Evaluation of an Address Space as a Means of Interaction with Internal State of Synchronized Sequential Logic

Matthijs Bos
m.bos1@student.uva.nl
Student nr. 10073558

Supervisor:
T. Walstra (UvA)
T.R.Walstra@uva.nl

January 5, 2017

1 Introduction

Previous work [1] has presented a model for the application of field-programmable gate arrays (FPGA) as a tool for experimentation with digital logic. In this model, the concept of an address space has been adopted as a means of interaction with experiment setup logic. The address space serves as a primary means of interaction with an experiment setup, through the projection of experiment setup state on this address space.

In the evaluation of this model however, the address space projections have been limited to experiment setup input and output signals. It has been suggested that the concept of an address space may also be suitable as a means of interaction with an experiment setup's elements of internal state, such as registers and memories. Although the model has been shown to support cycle-accurate control over experiment setups defined through synchronous sequential logic, interaction with stateful elements inside the experiment setup's logic is a significant objective in achieving practical applicability of the model. This article evaluates the capabilities of an address space as a means of interaction with an experiment setup's stateful elements within the context of the previously proposed model.

2 Exposing Internal Stateful Elements

The nature of HDL modelling languages such as VHDL and Verilog HDL encourage the instantiation of design entities at lower levels of the design hierarchy, at the location where these are referenced and interconnected. As can be observed in figure 1a, this habit of local instantiation results in a situation in which entities only contain references to lower-level instances. Additionally, an entity's interface signal definition must be satisfied on instantiation.

2.1 Dependency Injection Principle

Application of the dependency injection principle is a common practice within the domain of object-oriented programming. A reference to the required components is passed to a lower-level object instance, such that instantiation can be transferred to a higher-level object instance. The dependency between two objects is often abstracted through the introduction of a separate interface definition.

Hardware modelling languages such as VHDL and Verilog HDL do not offer a means for passing of instance references. A set of signal definitions can however be considered equal to an interface definition. The interconnection of two entities' signals can furthermore be considered equal to reference passing of an instance that implements a particular interface. Newer HDL modelling languages such as SystemVerilog natively include the concept of an interface as a means for the collective identification of a set of signals. Following the mapping between these concepts, the dependency injection principle can be considered applicable to the domain of HDL modelling, such that the instantiation of an entity instance can be transferred to a higher-level entity instance. Figure 1b provides a graphical display of the dependency injection principle.

Since a signal's direction is of importance within the context of HDL modelling, the distinction between an interface and its complementary interface for consumption is made. In order to transfer the instantiation of a lower-level entity to a higher-level entity, the instantiating entity's definition is extended to include the inverse definition of the lower-level entity's. The lower-level entity is then instantiated in the higher-level entity and connected to its original parent.

2.2 Dual-Interface Elements

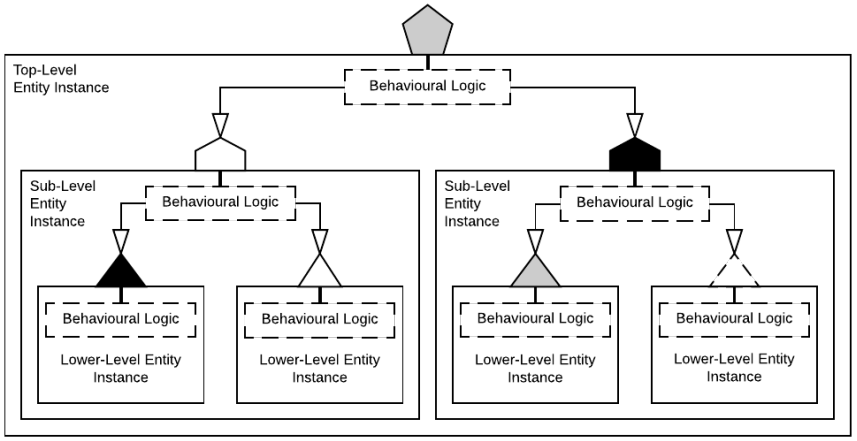
Application of the dependency inversion principle allows for the creation of design entities that are consumed by separate sibling entity instances. Figure 1c displays an example design in which two entity instances consume a shared entity instance. Consumption of a shared instance allows for two logic components to share a common stateful element.

As displayed in figure 1d, instantiation of a shared element does not necessarily have to happen in isolation. A dependency relationship between two entity instances can be established in order to allow for one to consume the other. By passing through one the signals, one can allow for interaction with a shared component that is instantiated locally, while reducing the amount of interfacing logic.

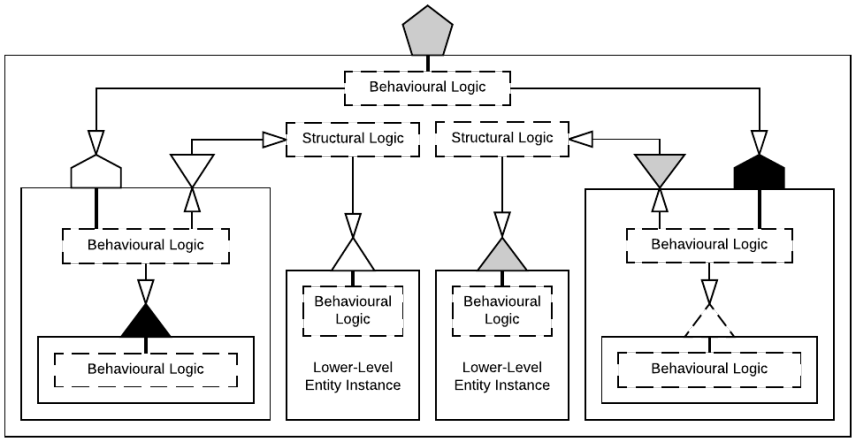
As demonstrated in section 4, the application of the dependency injection principle as well as the creation of dual interface elements are important steps in the process of projection of stateful elements on the experiment setup's address space. The dependency injection principle is applied in order to isolate the stateful elements from the original experiment setup logic. This isolation then allows for these stateful elements to be converted to dual-interface elements, such that these allow to be interacted with through the experiment setup controller.

Figure 1: Illustration of the dependency injection principle applied to a hierarchy of entities in order to share a common instance.

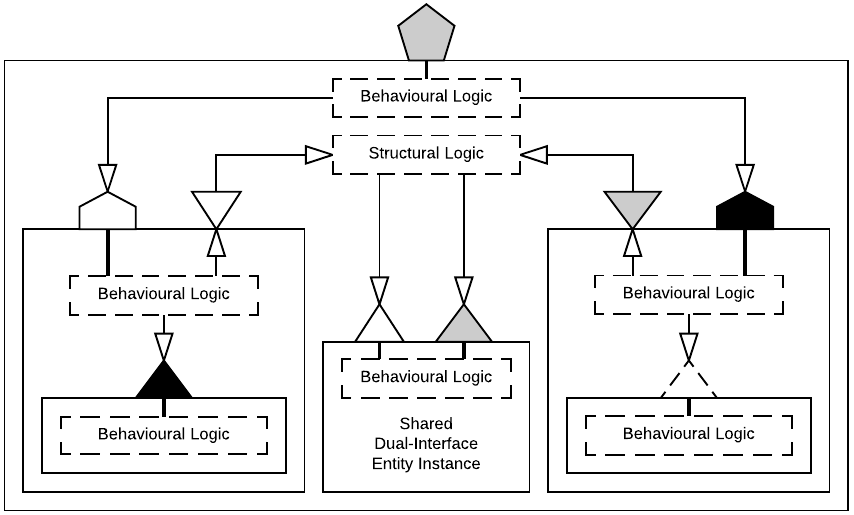
(a) Composition of entities in a hierarchical tree structure. The polygons represent different entity interfaces. An inverted polygon represents an interface's complementary definition for consumption.



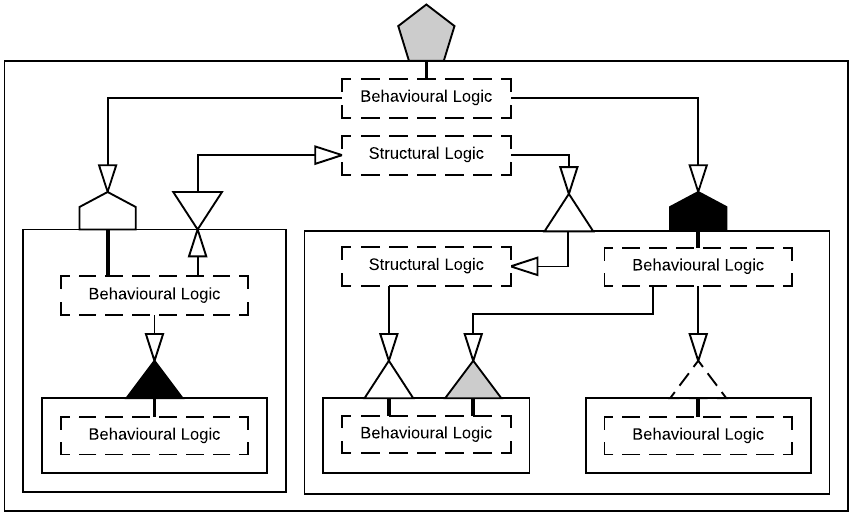
(b) The dependency injection principle applied. The sub-level entity interfaces have been extended in order to allow for interconnection with lower-level entity instances that are instantiated in the top-level entity instance.



(c) The application of the dependency injection principle allows for two branches in the instance hierarchy to share a common entity instance. This shared instance is instantiated in the top-level entity instance.



(d) An alternative composition in which a shared entity is instantiated in a sub-level entity. The containing sub-level entity's interface has been extended to include a producer interface.



3 Asynchronous Read Behaviour

Experiment setup logic designs may contain elements of sequential logic which' read behaviour is defined to be asynchronous. One such example is the MIPS computer architecture as developed in [2, Ch. 4]. For reading operations, all elements of sequential logic in this design are defined to reflect changes on their address bus without a clock signal triggering this change. Figures 2 and 3 display a small design which emphasizes this type of behaviour.

Figure 2: An example logic design in which a ROM is defined to reflect changes on the address bus asynchronously.

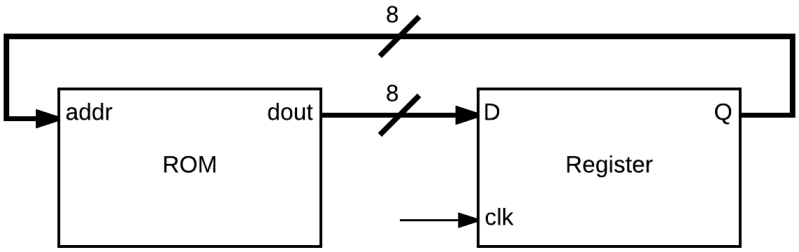
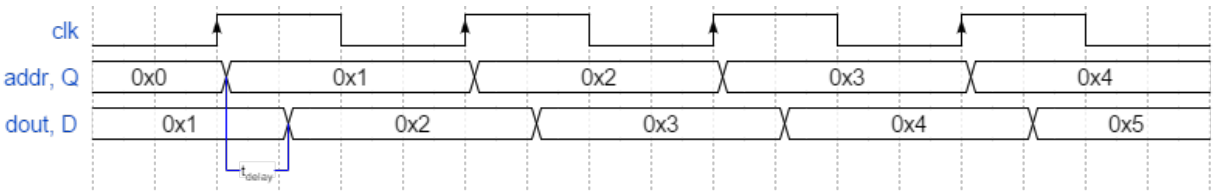


Figure 3: Timing diagram displaying the behaviour for the logic elements displayed in figure 2. The contents of the ROM are defined as such that $value(addr) = addr+1$. The ROM's propagation delay is identified by $\mathtt{t_{delay}}$.



Projecting this immediate read behaviour on a FPGA proves to be problematic in the case of large memories. Small stateful elements, such as registers or register files can be translated to CLB flip-flops, allowing for preservation of the asynchronous read behaviour. Large memory elements however, are generally projected on a FPGA's block ram elements which' read and write behaviour is synchronous, requiring one or more clock cycles to complete its operations. A design requiring even larger memories may make use of the FPGA's peripheral memory devices, such as SRAM or DRAM ICs. These peripheral devices generally introduce significant latencies for reading and writing operations.

3.1 Clock Delay

Through intentional delay of the experiment setup clock, these relatively slow memory devices can still be utilized to behave asynchronously. The experiment setup logic clock signal can be considered a subsample of the global clock signal. More specifically, this signal operates at irregular intervals between global clock pulses. Through operation of the slow memory devices off the global clock signal, the idle time in-between experiment setup clock cycles allow for these synchronous elements to update their data outputs and expose asynchronous read behaviour from the perspective of the experiment setup logic.

One could compare this type of behaviour to a situation of two clock domains in which a high-latency memory element operating at high frequency is consumed by another component operating at a relatively lower frequency. The current approach avoids challenges related to clock domain crossing however, since both clocks are defined to be similar, except for the fact that the experiment setup's clock signal is occasionally disabled.

Figure 4: Example logic design similar to figure 2 that features a synchronous `ROM`, requiring a clock input. The triangle represents a clock gating buffer that is controlled by the `exp_clk_en` signal.

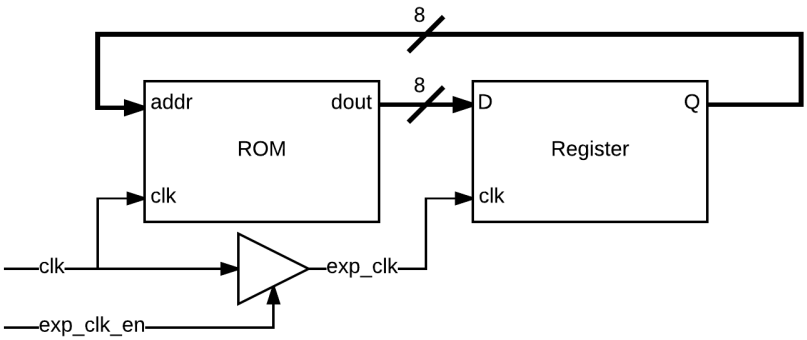
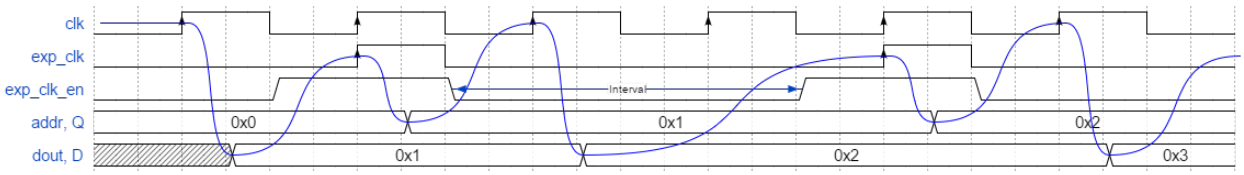


Figure 5: Timing diagram displaying the behaviour for the logic elements displayed in figure 4. The blue curved line follows a sequence of events that show how the signal changes propagate.

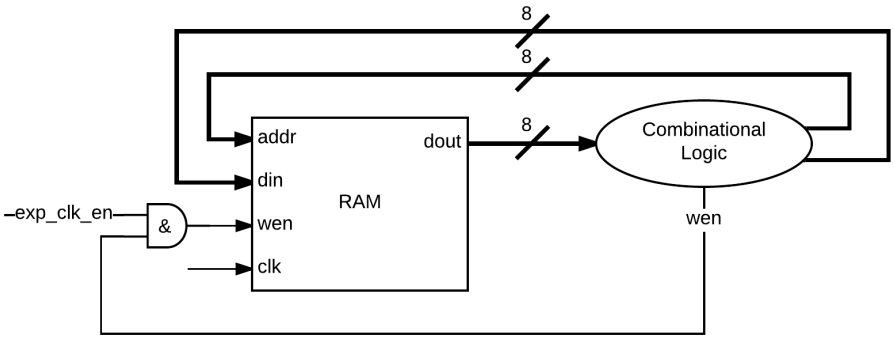


3.2 Synchronous Write

A consequence of a memory element operating off the continuous global clock signal is that write operations are synchronized with this signal as well. Write operations are generally triggered trough a pulse on a write enable signal. Since the write enable signal is controlled through experiment setup logic, this pulse may span multiple global clock cycles, triggering multiple write operations.

In order for a write operation to be executed only once on the experiment setup clock signal's rising edge, additional logic is required. Figure 6 displays an example RAM which operates off the global clock signal. An and-gate is included to prevent the RAM's write enable signal `wen` to be raised unless an experiment setup clock cycle is executed. This is achieved trough conjunction of the experiment setup's `wen` signal and the `exp_clk_en` signal. As displayed in figure 5, the `exp_clk_en` is raised before the experiment setup clock signal's rising edge occurs.

Figure 6: Example logic design in which additional logic is included to prevent the the RAM's `wen` signal to be raised in-between experiment setup clock cycles.



4 Implementation

An existing MIPS implementation has been modified to support interaction through an address space. More specifically, this implementation features an implementation of a simple single-cycle MIPS computer architecture, very similar to the design developed in [2, Sec. 4.4]. Although limited in the number of supported instructions, the implementation follows the instruction format specification as established in [2, Ch. 2]. Source code for the original version[1] as well as this article's modified version[2] are publicly available on GitHub. This article's implementation is targeted at a Digilent Nexys 4 FPGA development board, featuring a Xilinx Artix-7 FPGA. The Xilinx Vivado IDE has been used for compilation of HDL sources.

4.1 Types of Stateful Elements

Although limited in terms of complexity and optimizations, the single-cycle MIPS design makes use of a variety of types of stateful elements, making it suitable as a means for establishing a proof of concept for the theory introduced in section **??**. Although optimizations such as pipelining introduce additional stateful elements into the design, these elements do not contribute to an increased variation of stateful elements types, since primitive registers can be used for this purpose.

- A 32-bit register is used as a program counter.
- The register file features a combination of immediate read behaviour as well as synchronous write behaviour.
- The instruction memory is defined to provide immediate read behaviour.
- The data memory is defined to provide immediate read behaviour while allowing for synchronous write operations.

4.2 Externalization of Stateful Elements

Applying the model as described in [1] results in a hierarchical design as displayed in figure 7 in which the stateful elements are instantiated locally. In order to project these elements on the experiment setup's address space, these need to be isolated and wrapped such that a dual-interface element is created. This is achieved through application of the dependency injection principle as described in section 2.1. Figure 9 displays the result of this process. Instantiation of the MIPS' internal stateful elements has been delegated to a higher-level entity and these have been wrapped such that a dual-interface component is created. The introduction of a dual-interface element allows for the stateful element to be consumed by the MIPS implementation as well as the experiment setup controller.

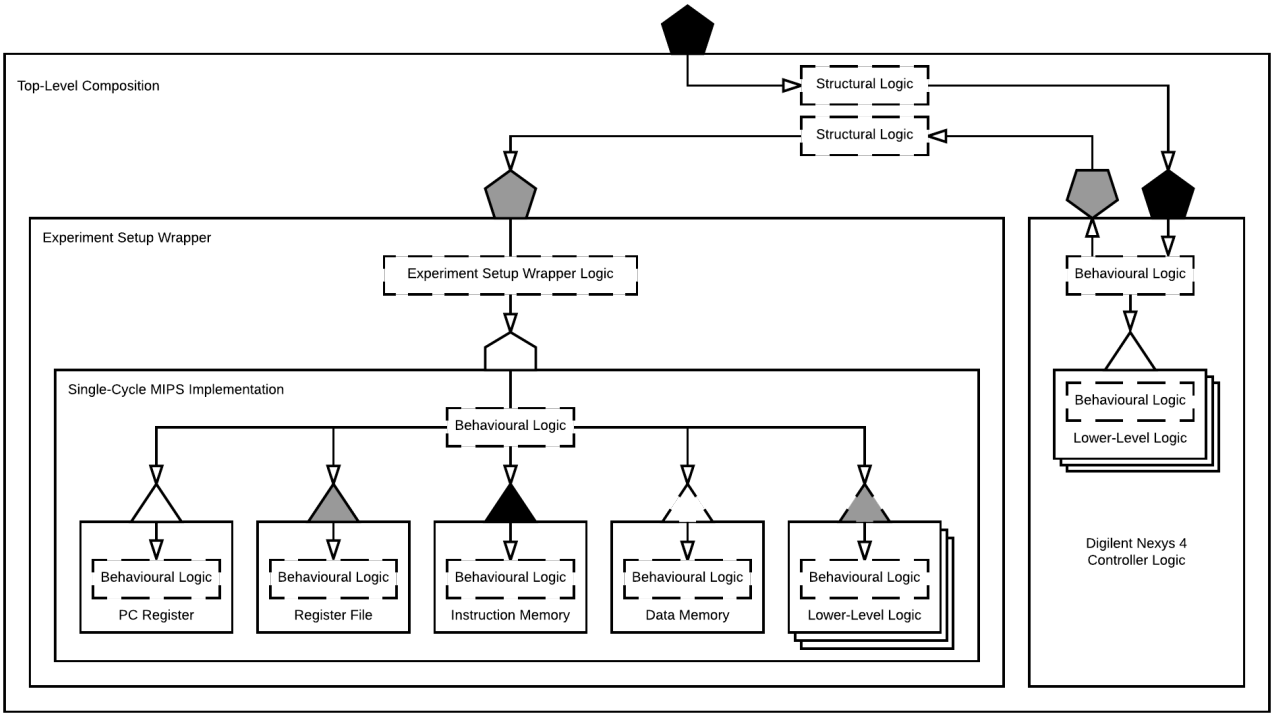
For the purpose of this implementation, the FPGA's internal block RAM elements have been utilized for the implementation of the instruction memory as well as the data memory. Both memory elements have been set to contain 1024 elements of 32-bit data words, resulting in bus widths of 10 and 32 for the address and data buses respectively. Although the size of these memories is relatively small, these suit the purpose of the proof of concept. Any memory size can be selected, as long as the FPGA's memory resources can accommodate the requirements.

Like FPGAs from other manufacturers, Xilinx 7 Series FPGAs come equipped with dual-port block RAM elements. The presence of this type of functionality eases the process of creating dual-interface stateful elements. Instead of adding additional logic for the multiplexing of the consuming interface, the FPGA provides this functionality natively. The theory introduced in section 3 has been applied to the wrapping logic in order to allow for both memories to expose asynchronous read behaviour.

An early design decision has been to extend the original experiment setup component interface to include the `exp_clk_en` signal, such that this signal's level can be used to select between the consuming component. This addition is of particular relevance for the handling of write operations. As can be observed in the timing diagram displayed in figure 5, the `exp_clk_en` signal is raised during an entire experiment setup clock cycle, indicating an upcoming rising edge on the experiment setup clock. The interface displayed in figure 8c includes the additional input signal.

[1] `https://github.com/tcamolesi/simple-mips`
[2] `https://github.com/matthijsbos/simple-mips-fpgaedu`

Figure 7: Hierarchical composition of single-cycle MIPS implementation. The MIPS' internal stateful elements are instantiated locally.



Although the instruction memory solely offers read-only functionality to the MIPS implementation, a fully-featured RAM element has been used to implement this behaviour. This decision has allowed for modification of the instruction memory's contents through the address space projection. Being unable to modify the ROM's contents at runtime would limit the capability of setting the instruction memory's contents at compile time during bitstream generation.

Utilization of the FPGAs built-in memory resources for the MIPS' memories forces a clean separation between the stateful component and the wrapping logic. The wrapper implementation for the MIPS' register components however, has allowed for a more integral approach in which the original components have been extended to include an interface for interaction through an address space.

4.3 Address Space Unification

The externalization and wrapping of the separate stateful elements creates separate 'miniature' experiment setups with their own address space and corresponding interface. Additional logic is required in order for these separate components to be projected on a single address space. As can be observed in figure 9, address space partitioning logic is added to the design that interconnects the different wrapper instances.

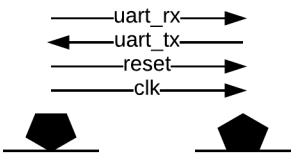
In the current design, a single implementation of a binary address space partitioning element is used. This element is used to select between two child experiment setup components based on the address value on the master address bus as provided by the controller. The component is parametrized such that the address ranges for both partitions can be set. An additional parameter is provided for both child experiment setup components that subtracts a given value from the master bus' address. This feature allows for an address in the global address space to be translated to an address in a local address space.

In this article's implementation, four instances of this binary component have been daisy-chained together in order to allow for five local address spaces to be projected on the experiment setup's global address space. Table 1 provides a detailed description of the resulting address space. These four paritioning elements could have been replaced by a single design entity to reduce the amount of logic elements and interconnections. For the purpose of simplicity and extensibility however, the current approach has been taken.

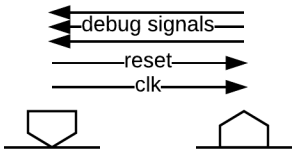
Table 1: An description of the experiment setup's unified address space. The different local address spaces have been projected on five partitions.

Partition		**Signal**			
Description	**Addr. Offset**	**Description**	**Addr. Offset**	**Abs. Addr.**	**Read/Write**
MIPS Debug	0	PC	0	0	Read-only
		PC next	1	1	Read-only
		Opcode	2	2	Read-only
		RS addr	3	3	Read-only
		RT addr	4	4	Read-only
		RD addr	5	5	Read-only
		Funct	6	6	Read-only
		Immed	7	7	Read-only
		Reg q0	8	8	Read-only
		Reg q1	9	9	Read-only
		Reg wdata	10	10	Read-only
		Instr data	11	11	Read-only
		Instr. addr	12	12	Read-only
		Data mem wen	13	13	Read-only
		Data mem ren	14	14	Read-only
		Data mem wdata	15	15	Read-only
		Data mem addr	16	16	Read-only
		Data mem rdata	17	17	Read-only
Program Counter	99	PC Register	0	99	Read+Write
Register File	100	Register 0, $zero	0	100	Read+Write
		⋮			
		Register 31, $ra	31	131	Read+Write
Instruction Memory	10000	Instruction 0	0	10000	Read+Write
		⋮			
		Instruction 1023	1023	11023	Read+Write
Data Memory	20000	Data word 0	0	20000	Read+Write
		⋮			
		Data word 1023	1023	21023	Read+Write

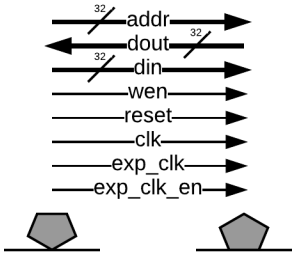
Figure 8: Interface defintions for those displayed in figures 7 and 9.



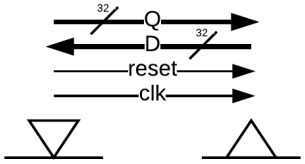
(a) Nexys 4 controller



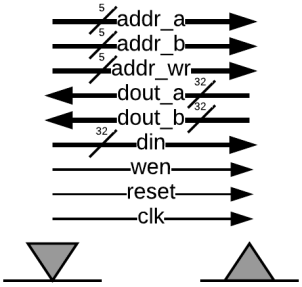
(b) Single-cycle MIPS



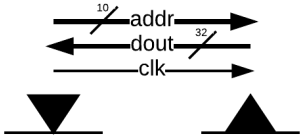
(c) Experiment setup component



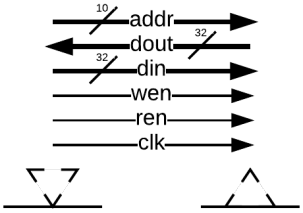
(d) PC register



(e) Register file

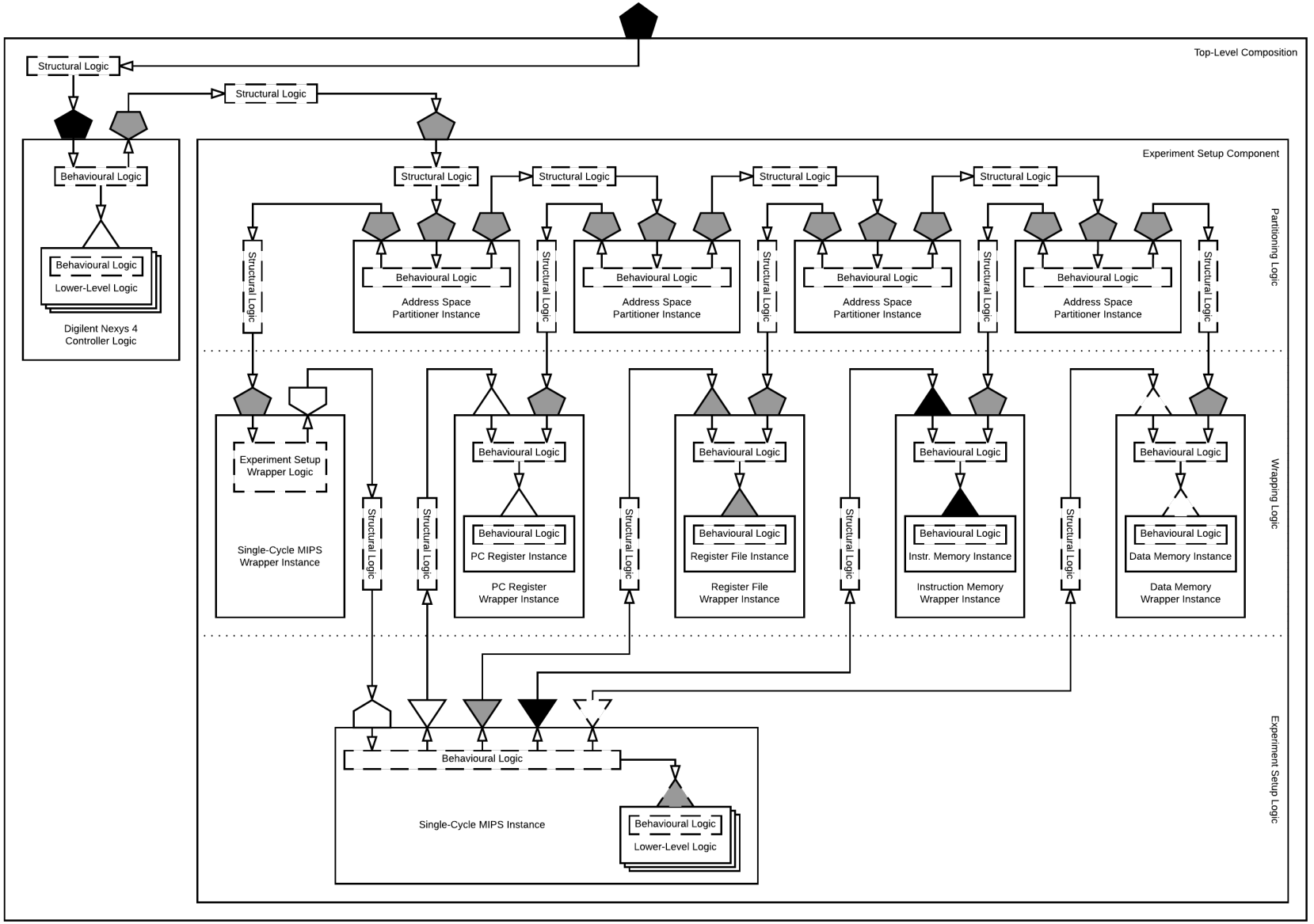


(f) Instruction memory



(g) Data memory

Figure 9: Composition of a single-cycle MIPS implementation. The stateful elements have been externalized through application of the dependency injection principle. Wrapping logic has been introduced to encapsulate the stateful elements, such that shared dual-interface elements are created. The address spaces for the individual stateful elements are unified trough daisy-chaining of four address space partitioning elements.



5 Evaluation

This article's implementation has been evaluated through functional validation. The original MIPS implementation includes a number of test bench scripts that verify the MIPS' behaviour through the execution of short instruction sequences. These same instruction sequences have been reused in order to validate the behaviour of this article's implementation. The original test bench scripts initialized the instruction memory's contents at compile time and were designed to run in a HDL simulator environment. For this article's implementation however, a different approach has been taken that initialized the instruction memory at runtime while the implementation was executing on a physical FPGA.

A simple preexisting command shell utility was used for interaction with the experiment setup's address space. The test bench instructions could be manually loaded into the MIPS' instruction memory and the program counter could be set to point to the correct position in instruction memory. Through the execution of cycles, these instructions were executed and their effects could be inspected through the read-only debug signals as well as through inspection of the values for the different stateful elements.

6 Conclusion

This article has explored the applicability of the concept of an address space as a means for interaction with an experiment setup's stateful elements. A number of theories and practices have been developed that allow for the projection of such elements on an experiment setup's address space. These theories and practices have been applied to modify a preexisting single-cycle MIPS implementation, allowing for the experiment setup's address space to provide a means of interaction with the different registers and memories that support the MIPS' operation.

This article's implementation has contributed to the practical applicability of the model as developed in [1]. The concept of an address space has proven to be a suitable means of interaction with various types of stateful elements. The experiment setup development process has increased terms of complexity and work required. From the experimenter's point of view however, the additional logic introduced for address space projection does not have a negative impact on a transparent means of interaction with experiment setup logic contained within a FPGA development board.

6.1 Future Work

6.1.1 Automated Projection

The additional work and complexity introduced into the experiment setup development process raises the need for further automation to aid in the development of new experiment setups. The repetitive structures as observed in figure 9 as well as during this article's implementation process suggests that the process of projecting a stateful element on an address space is a candidate for automation. Similar to the previously demonstrated functionality for automated wrapper generation, code generation can play an important in achieving this goal of automated projection of stateful elements.

While the automated detection of design entity input and output signals is a relatively simple task, the automated detection, externalization and wrapping of stateful elements is considered significantly more challenging. The complexities of HDL modelling languages will likely prove the automated detection of these elements to be especially difficult. One might instead take a hybrid approach in which the process of identification is manually done during the development process, instructing an automated process that generates the logic for address space projection. One approach might be to provide a standard library of stateful components that developers can then use in their experiment implementations. The detection of these standard components can be considered a relatively easy task. Externalization of these standard components can then be achieved through automated refactoring of the HDL design, while the wrapping logic as well as the partitioning logic can be automatically generated based on templates.

6.1.2 Memory Latencies

Although mentioned in section 3, this article's implementation has not utilized the peripheral SRAM memory device that is available on the Nexys 4 development board. Many FPGA development boards come equipped with large memory devices whose capacity may increase the applicability of the model as a means of hosting experiments setups with large memory requirements. Current work has not included these memory devices

in their implementations, but has only included the FPGA's built-in block RAM devices. It is likely that the relatively high latencies of peripheral memory devices can be compensated for through intentional delaying of the experiment setup clock as described in section 3.1.

An underexposed subject in this article however, is the required length of this delay. The block RAM elements used in this article's implementation require a minimum delay of 1 clock cycles if asynchronous read behaviour is desired. However, n cycles of delay are required if n of these asynchronous elements are daisy-chained together in order for one change to propagate through. Including high-latency peripheral memory devices in the design further complicates determining the required duration for delay. The current implementation does not enforce a specific delay in any way. A future implementation might include some form of feedback mechanism for a stateful element to indicate that a new cycle on the experiment setup clock can be initiated.

6.1.3 Experimentation Software

An increased size of the experiment setup's address space description makes it more and more difficult to interact with the experiment setup's address space through simple read and write operations. The address space description of this article's implementation as displayed in table 1 has already proven to be difficult to interact with as the complexity of the actual experiment rises. Loading a large program into the MIPS' instruction memory for example, proves to be impractival trough manual commands. At the time of writing however, work has already started on an improved command line shell application[3] that aims to tackle these issues. The new software makes use of the provided address space description to aid the experimenter in executing some tasks automatically and prevent manual lookup of addresses.

[3] https://github.com/fpgaedu/fpgaedu-shell

References

[1] M. Bos, "A Model for Experiment Setups on FPGA Development Boards," Bachelor's thesis, Univeristy of Amsterdam, Aug. 2016. [Online]. Available: `https://esc.fnwi.uva.nl/thesis/centraal/files/f7614483.pdf`.

[2] D. A. Patterson and J. L. Hennessy, *Computer organization and design: The hardware/software interface.* Newnes, 2013.