Liam: An Actor Based Programming Model for HDLs

Haven Skinner
Dept. of Computer Engineering
hskinner@ucsc.edu

Rafael Trapani Possignolo
Dept. of Computer Engineering
rpossign@ucsc.edu

Jose Renau
Dept. of Computer Engineering
renau@ucsc.edu

ABSTRACT

The traditional model for developing synthesizable digital architectures is a clock-synchronous pipeline. Latency-insensitive systems are an alternative, where communication between blocks is viewed as message passing. In hardware this is generally managed by control bits such as valid/stop signals or token credit mechanisms. Although prior work has shown that there are numerous benefits to building latency-insensitive digital hardware, a major factor discouraging its use is the difficulty of managing the additional logic and infrastructure required to implement it.

We propose a new programming paradigm for Hardware Description Languages, built on the Actor Model, to implicitly implement latency-insensitive hardware designs. We call this model Liam, for Latency-Insensitive Actor-based programming Model. Liam's paradigm allows the programmer to manage the logic of receiving, processing, and sending messages between pipeline stages implicitly, and independently from the underlying infrastructure. We show Liam programming model with a simple, Python-like HDL called Pyrope which has a few, non-intrusive constructs added to manage elastic behavior. To demonstrate the results, we discuss the simulation and synthesis results for a variety of digital hardware systems, including several variations of 32 and 64 bit RISC-V processors.

ACM Reference Format:
Haven Skinner, Rafael Trapani Possignolo, and Jose Renau. 2017. Liam: An Actor Based Programming Model for HDLs. In *Proceedings of MEMOCODE '17*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3127041.3127060

1 INTRODUCTION

The dominant model for developing synthesizable digital hardware is a clock-synchronous pipeline. The system is modeled as the interaction between one or more *stages*, which update their internal and external state in sync with one or more global clock signals.

This model presents some fundamental challenges to the developer and synthesis tools. The developer is constrained by the fact that the longest combinational logic path, or *critical* path, in each stage should be similar, to achieve good efficiency. The synthesis tools are also constrained because they can not change the number of pipeline stages. Taken together, these result in the desired clock frequency being locked down early in the development process, and adds a high cost in effort and man-hours, to make high-level changes to the system. Although these challenges can be managed, they represent fundamental obstacles to developing large-scale systems.

An alternative is a latency-insensitive design, where the system is abstractly seen as being comprised of nodes, which communicate via messages. Synchronous hardware systems can be automatically transformed to latency-insensitive designs, which we refer to as elastic systems [1–3]. Alternately, the designer can manually control the latency-insensitive backend [4, 5], we call such systems Fluid Pipelines. The advantage of exposing the elasticity to the designer is an increase in performance and flexibility in transformations [4, 6], the disadvantage is that the developer is responsible for handling logic driving the handshake signals, needed to manage message passing and back pressure between stages.

Some previous infrastructures implementing event driven simulations [7–10] use a handshake between pipeline stages similar to the handshake used by Fluid Pipelines. There are also designs like OpenCores [11], which implement blocks that use some valid and stop/ack handshakes between stages. In both cases, implementing the handshake was not done to leverage the advantages of Fluid Pipeline transformations but for other reasons like clarity or time multiplexing. Nevertheless, in all the cases the designer has to manage the handshake manually.

In this paper, we propose a new latency-insensitive programming model that can be applied to hardware description languages, called Liam. Liam is a programming model for HDLs based around a simplified Actor Model, designed to implement elastic behavior implicitly. Liam stands for Latency-Insensitive Actor-based programming Model for HDLs. It allows the programmer to focus on the high-level logic of the architecture, independently from the underlying infrastructure. To demonstrate Liam, we use a Python-like HDL called Pyrope, which can be compiled using a custom, DSL compiler, into a C++ or Verilog implementation of the system. In this paper we show that building digital architecture under a paradigm like this gives the designer more control over the development, simulation, and synthesis process.

In Section 3, we give some background on elastic pipelines and the Actor Model. In Section 4 we describe the Liam programming model, how it is integrated into the Pyrope HDL, and how it is compiled. Section 5 describes optimizations for simulation and synthesis which are enabled by Liam. In Section 7 we evaluate several hardware architectures in Liam, and compare their simulation and synthesis results against similar Verilog implemetations. Section 8 adds some final thoughts.

2 RELATED WORK

Liam is a hardware description language (HDL) for elastic pipelines built on the Actor Model. As such, this section first discusses related

MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria

ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127060

work with regard to the Actor Model, then discusses related work with regard to elastic pipelines and other HDLs.

The Actor Model was first proposed in 1973 [12] as an artificial intelligence computational model. It proposed viewing the system as an interaction between *actors*, which are atomic units that communicate via message passing. It has since been applied in managing concurrency [13–15] between heterogeneous components, and as a general paradigm for parallel and distributed system programming [16]. Labview uses an Actor Model-based backend to manage message passing between multi-threaded applications [17]. UC Berkeley's Ptolemy Project [18, 19] uses an Actor Model-based simulator to model heterogeneous embedded systems and embedded system networks. Pulsar [14] and Akka [15] provide Actor Model-based libraries for managing concurrency for Python and Java respectively. When studying the problem of developing elastic systems in hardware, we saw potential in the idea of viewing them as a type of concurrency problem. We ultimately decided on using the Actor Model as the basis behind Liam's paradigm, because of its previous applications in managing concurrency, and the analogies between the behavior of an actor, and a stage in an elastic pipeline. This is further discussed in Section 3.4.

There are a variety of projects which have created libraries and frameworks for implementing the actor model in traditional programming languages, such as CAF (C++ Actor Framework) [20], Pykka [21] (Python), Aojet [22] (Swift), Orbit [16] (Java).

We are not aware of any Actor-oriented framework available for a sythesizable HDLs. For this project we decided instead to create a custom DSL rather than try and implement a framework in an existing HDL for two reasons: First, the traditional HDLs, (System) Verilog [23] and VHDL [24], have limited abstraction and meta-programming capabilities, and there are a variety of ways to implement elastic pipelines: queues, handshakes, or often some combination. We wanted a framework that could produce re-usable elastic pipeline code which could be shared between different types of backends. Second, a major challenge in working with elastic pipelines comes from making mistakes with regard to the underlying protocol, for example not shifting data from the queue or completing the handshake at the wrong time can cause deadlocks or dropped data. We wanted our framework to handle the low-level implementation and provide high-level constructs which the programmer could use to manipulate the pipeline's behavior, but restrict them to working within the bounds of the underlying elastic protocol. This combination of needing more freedom of abstraction in one regard and wanting to restrict potential behavior in another encouraged us to develop our our DSL and compiler for this project.

Elastic pipelines were first proposed to solve the problem of latency in long wires across a chip [25], but have been later studied in the context of pipeline transformations [1, 2]. In particular, Latency-Insensitive systems allows for changing the number of pipeline stages without breaking the system functionality. One early issue with these transformations is the addition of bubbles to the pipeline when new stages are added, which have been shown to reduce throughput [3, 26]. Fluid Pipelines [4] were recently proposed to reduce the penalties associated with pipeline transformations and improve area and energy utilization of such systems. Notably, Fluid Pipelines do not have any throughput penalty [4].

Elastic pipelines have also been used on the backends of hardware modeling tools. The Liberty Simulation environment [27] is an HDL and simulation infrastructure designed for modeling digital systems. On the backend the compiler inserts a simple handshake, driven by *enable* and *ack* signals, which is analogous to the *valid* and *stop* signals that Liam's compiler inserts between blocks to generate a fluid pipeline. Liberty is for modeling, not synthesis, however, and models only synchronous systems. Liberty also does not support multiple clock domains.

Li-BDN [10] describes a method for compiling an asynchronous (elastic) pipeline into a network of FPGAs for faster simulation. In doing so, the Li-BDN method sacrifices cycle-accuracy, and it is not possible to compile the non-cycle-accurate Li-BDN network into a synthesizable version of the original system.

Newer hardware description languages like Chisel [28] include syntax features designed for implementing asynchronous/elastic interfaces. This can simplify the process of managing the behavior of the required control bits, but it still requires the control logic to be manually implemented by the programmer.

Bluespec [29] has the "maybe" modifier for variables, which is functionally somewhat similar to associating a valid signal to the variable. It is different from elasticity because it does not provide support for back pressure.

Liam's programming model also has some similarities to Synchronous Reactive programming languages like Esterel [30]. Esterel programs are divided into modules, which communicate via signals. Signals are either present or absent at any given instant, and Esterel provides constructs to react to the presence or absence of input signals and trigger outputs accordingly. Prior work has also proposed using Esterel for RTL synthesis [31]. Esterel does not contain any built-in, synthesizable constructs to handle back pressure, so for elastic backends like Fluid Pipelines, Esterel would still require the developer to write the logic for managing a synthesized handshake manually.

Another language with a paradigm that has some similarities to Liam's is Lustre [32]. Lustre is a *synchronous dataflow programming language* designed for implementing reactive systems. Synchronous dataflow programming languages are based on a the concept of a dataflow graph, a graph of nodes which represent operations and are connected to their dependencies. Lustre programs are comprised of blocks called nodes, which can take one or more inputs and produce one or more outputs. Unlike imperative programming languages like C or Java, which are based on sequential operations, Lustre programs are a network of nodes which react to changes on their inputs. Lustre has also been called a *stream processing* language [33], and has found applications in verifying real-time systems.

Lustre has not been used for RTL synthesis, but other stream processing languages have. Lime [34] is a Java-based HDL which divides the system into nodes, and uses a task-based model to process inputs and update outputs. Both Lustre and Lime's models have clear analogies to elastic systems, and we believe that these languages could be compiled into a fluid pipeline as we do with Liam. For this project, however, we chose to base our paradigm on an imperative model, rather than a dataflow one, and use the concept of aborting execution to allow the programmer to manage elastic behavior. This is further discussed in Section 4.

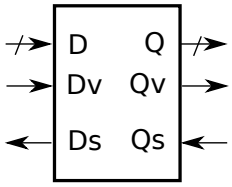

Figure 1: A fluid register. D and Q are data input and output ports, and Dv and Qv and their corresponding valid bits. Ds and Qs are the stop bits for this register, which travel in the opposite direction of data to handle back pressure.

3 BACKGROUND

3.1 Latency Insensitive Behavior

Table 1: The behavior of a multiplexer latched with a synchronous register.

Clock Cycle	1	2	3	4	5	6	
In0	0	1	5	2	4	3	
In1	1	4	2	3	3	1	
S	1	1	0	1	0	0	
Out		1	4	5	2	4	3

Table 2: An "elastic multiplexer". Blank spaces represent invalid data. The multiplexer waits to receive sufficient valid inputs before producing valid output.

Clock Cycle	1	2	3	4	5	6
In0	0	0		2	2	
In1		1			3	
S		1	0	0	0	
Out			1			2

Tables 1 and 2 shows a comparison of synchronous and elastic behavior with regards to a multiplexer. The first chart shows traditional mux, while the second chart shows the behavior of an "elastic mux", which waits for valid data before producing an output.

The first thing to notice, is that there's a subtlety to the behavior of the elastic mux. It will only produce output if it has valid data in all three ports. For some applications, it may instead be desirable for the system to produce data as soon as possible, and drop a packet on the non-selected port when it arrived. It may also be desirable to never drop packets, but hold it until it is selected, making this more of a switch.

The existence of valid/invalid data and the decision of when to read input and produce output adds another vector of control to the system, which must be accessible to the programmer. This demonstrates why it is difficult to abstract away this extra logic, or turn a synchronous system into an optimized elastic one with an automated tool.

3.2 Fluid Pipelines

A fluid pipeline are a type of latency-insensitive system. It is structurally similar to a synchronous pipeline, but instead of clock-synchronous registers being inserted between stages, fluid registers, shown in Figure 1, are used instead. Along with the data input and output ports present in synchronous registers, fluid registers add input and output ports for two control bits: *valid* and *stop*. Valid travels in the same direction as data, and stop travels in the opposite direction to handle back pressure. Internally, the fluid register buffers the sent packet, and holds it if the receiver asserts stop. The fluid register also passes the stop signal backwards through the pipeline, forcing earlier stages to stop as well. This is how fluid pipelines handle back pressure without a throughput penalty or the risk of buffer overflow.

3.3 Implementing Fluid Pipelines Manually

Listing 1: A pseudo-code implementation the elastic mux depicted in Table 2. Even a rather simple block requires a fair amount of backend logic to implement.

```
1  stage elastic_mux(
2    output out,
3    input a, input b, input s):
4
5    out.valid = false
6    if s.valid: s.retry = true
7    if a.valid: a.retry = true
8    if b.valid: b.retry = true
9
10   if s.valid and not out.retry:
11     if s == 0 and a.valid and b.valid:
12       s.retry = false
13       a.retry = false
14       b.retry = false
15       out = a
16       out.valid = true
17     elif s == 1 and b.valid and a.valid:
18       s.retry = false
19       a.retry = false
20       b.retry = false
21       out = b
22       out.valid = true
23     else:
24       out.valid = false
25   else:
26     out.valid = false
```

Listing 1 shows a pseudo-code implementation which would produce output consistent with Table 2. Each input and output port is attached to a fluid register, shown in Figure 1. As per the fluid pipeline handshake protocol, described in Section 3.2, each output valid but must be asserted if it contains valid data, cleared otherwise. Likewise, the input retry flags must be cleared if the inputs have been consumed, and set otherwise. As we can see, elasticity adds a non-trivial amount of additional logic which is needed to decide when to produce output, and when to hold or consume input. It is not so easy to abstract this logic into an API, because it's inherent to the logic of the stage. Further, these vectors of control must be available to the programmer.

Our solution is to propose a new programming paradigm which adds some axioms and constructs to allow the programmer to manage the elastic behavior of the system implicitly, and independent from the underlying infrastructure. In the upcoming Section 3.4 we describe the Actor programming model, and how we build on it to accomplish this.

3.4 The Actor Model

The *Actor Model* is a programming paradigm first proposed by Hewitt, Biship, and Steiger [12] in 1973, where the program is modeled as the interaction between any number of independent *actors*. Actors are atomic units which communicate with each other via messages, they have the following capabilities:

- Update their local state.
- Send a message to another actor.
- Decide to act on a message they received.
- Create more actors.

Elastic pipelines, which are made up of stages that communicate through latency-insensitive interfaces, can naturally map to this paradigm. In an elastic pipeline the latency between the time when a stage receives an input and produces an output is not defined, just as in the actor model, the actor can decide when to update its internal state or respond to a message.

There are differences between the traditional Actor Model and Liam's paradigm as well.

The pipeline in Liam is divided into stages, which are analogous to actors. One capability of an actor model actor, is the ability to create more actors. Considering Liam's intended application is digital hardware synthesis, we decided that this would be problematic to implement. Instead the compiler is fed what we call a topology file which lists which stages are used in the pipeline. For this reason one may call Liam's programming paradigm a reduced actor model, since the number of actors is fixed at compile time.

The Actor Model also makes no provision for back pressure, which is required by some fluid and elastic pipeline backends. Liam implements this by inferring logic which reads the values of status flags that signal back pressure. The inferred logic prevents the stage from processing new data, and asserts the back pressure flags on its own inputs. This can be thought of as adding a new condition onto all Liam stages, that they will not send new messages until they get an acknowledge from the recipients of the last message that it was received. The process of compiling elastic control logic, with and without back pressure, is discussed in detail in Section 4.2.2.

Another small, but important difference is that in the traditional actor model, actors are independent, atomic units. We model stages in Liam this way, but add recursive dimension as well. Since the overall pipeline behavior is independent of latency, its functionality is independent of the latency through any given stage. Thus, each stage can be viewed as a pipeline of its own. This gives Liam a conceptual basis to manage pipeline transformations laid out in previous work on elastic pipelines [1, 2, 4].

4 LIAM

4.1 Compiling Pyrope with Liam

Pyrope is designed to simplify large-scale digital architecture design. The system is organized into *stages*, which are compiled into combinational logic. The compiler is also fed what we call a topology file, which lists all the stage-to-stage connections, and lists which stage input and output ports map to input and output ports of the pipeline. When compiling with Liam, Pyrope instead inserts fluid registers, as shown in Figure 1, and generates logic to drive the fluid registers' control signals based on the axioms described below.

Listing 2: A Pyrope implementation of a mux, which has the same behavior the elastic mux in Table 2, if it is compiled with Liam.

```
1  stage mux(output out,
2    input a, input b, input s):
3    if s == 0:
4      out = a
5      consume(b)
6    else:
7      out = b
8      consume(a)
```

Listing 3: A Pyrope implementation of a switch. Similar to Listing 2, but it will only read and consume the selector and the selected data input.

```
1  stage mux(output out,
2    input a, input b, input s):
3    if s == 0:
4      out = a
5    else:
6      out = b
```

Listing 2 shows a Pyrope *stage* which implements the same behavior as the "elastic multiplexer", described in Table 2, when compiled with Liam. The data-path behavior is laid out in roughly six lines of code which sets *out* equal to either *a* or *b* based on the value of *s*. The status flag behavior, which is responsible for implementing elasticity, is based on the axioms 1 (Atomicity) 2 (Abortion) 3 (Isolation).

The state of the stage is made up by the state of its outputs and the state of its internal registers. The state of a stage's outputs is made up of the values of the stage's output data ports and the state of its status flags. As stated in Axiom 1 (Atomicity), stages are atomic units that either update or abort every cycle. Axiom 2 (Abortion) describes the conditions which cause a stage to abort, and Axiom 3 (Isolation) describes what happens when a stage does abort.

Axiom 1 (Atomicity). *A stage executes from the top down. If it completes without aborting, it updates its state.*

Axiom 2 (Abortion). *A stage aborts when an invalid input is read, or an output with back-pressure is written.*

Axiom 3 (Isolation). *Aborted stages do not consume any input, generate any valid output, or make any other changes to its state.*

Taking these axioms into account, we can see that the code in Listing 2 matches the example elastic mux behavior shown in Table 2. In order for the stage to finish executing, there are only two paths it can follow, the $s = 0$ path and the $s = 1$ path. Both paths read all three inputs, lines 4 and 7 force a read on the unselected input, assuring that the stage will abort unless all inputs are valid. If this stage is compiled with an elastic infrastructure that supports back pressure, like Fluid Pipelines, Liam will also compile logic to handle back-pressure, as required by Axiom 3 (Isolation). Both paths require a write to *out*, which for Fluid Pipelines will cause an abort if the *stop* flag is asserted.

By contrast, if we remove the statements on lines 4 and 7, as we do in Listing 3, we effectively have a switch. Each path will only read the selector and the selected input port, so the unselected port will be ignored. If it is valid it will not be consumed, if it is invalid it will not be read.

4.2 Internal State

Inputs and outputs constitute a stage's external state, but a stage has an internal state as well, which is implemented through *registers*. Pyrope borrows syntax from Ruby, using the @ symbol to represent a register variable, which unlike regular variables remembers its value from the previous, non-aborted, iteration. In Pyrope, registers are initialized to 0.

When compiling with Liam, registers are considered part of the stage's state, so they are updated at the end of the stage's execution, assuming it doesn't abort. Reading them will never produce an abort since they are initialized to zero. A register can also be an output, however, and like any output, it could have back pressure. Writing to an output register will produce an abort if that output has back pressure.

4.2.1 Added Constructs. Liam adds some new constructs to allow the programmer to manage elastic behavior. These are listed in Table 3. They allow the programmer to control elastic behavior abstractly, independent from any one specific type of backend.

Table 3: Liam adds some new constructs which helps the programmer manage elastic behavior abstractly.

Name	Code form	Description
Valid Check	*var*?	Checks if an input or output is valid
Stop Check	*var*!	Checks if an input or output stop flag is asserted.
Keep Statement	keep *input*	Prevents an input from being consumed.
Consume Statement	consume *input*	Explicitly consume an input.

Although we have had ideas for other useful operators, this minimum set is what was needed for our early evaluation of Liam, where we implemented the designs discussed in the evaluation, Section 7.

The valid and stop operators allow the programmer to check whether an input is valid or whether an output has back pressure, without triggering an abort. If the stage is compiled to a backend which doesn't support back pressure, the stop flag is compiled as a constant false. The "keep" statement will prevent an input from being consumed. The "consume" statement is not strictly necessary, since an input is implicitly consumed when it is read. It is employed strictly for readability.

Using these additional constructs, we can modify the multiplexer from Listing 2 with more subtlety. We may wish to modify the mux to have it drop data on the non-selected port if that port has data, but produce output regardless. This is implemented in Listing 4. By using the valid operator, we check whether the non-selected data port is valid before reading, preventing an abort.

Listing 5 is similar to Listing 4 in that it produces an output as soon as it has a valid data packet which matches a valid selector packet, but unlike Listing 4, it will always drop a packet from the non-selected data path. If there is no data available when the output is ready, the stage uses a register to remember that it will drop the next incoming packet on that port. This is accomplished by setting the registers *@dropa* or *@dropb* to true. If either of those registers are true, on the next cycle the stage will attempt to consume the input and set the corresponding register to false. This will cause an abort until there is a valid input on that port, meanwhile if other valid inputs arrive, they are implicitly held until the stages reaches an execution path that consumes them.

Listing 4: An alternative elastic mux, which will drop the non-selected input port if it is valid, and produce an output regardless.

```
1  stage mux(out, a, b, s):
2    if s == 0:
3      out = a
4      if b?: consume b          # read b
5    else:
6      out = b
7      if a?: consume a          # read a
```

Listing 5: Another version, which produces output as soon as the selector and matching data input port is valid, but also drops a packet from the non-selected data input port, whenever that packet arrives.

```
1  stage mux(out, a, b, s):
2    if @dropa:
3      consume a
4      @dropa = false
5    elif @dropb:
6      consume b
7      @dropb = false
8    else:
9      if s == 0:
10       out = a
11       @dropb = true
12     else:
13       out = b
14       @dropa = true
```

These constructs, along with our core axioms, allow the programmer to manage the condition at which inputs and outputs will be consumed and updated. We do not provide constructs to override elastic behavior however. For example, the valid operator (denoted

with a ?) can be used to read a valid flag but not write one. Providing this would be problematic, because if the stage receiving the data asserts the stop flag, it is expecting to receive the same data on the next cycle. In some cases an engineer may be tempted to violate the elastic protocol because it may solve a problem they have run into in a given application. Doing so remains risky nonetheless because in the project employs any of the optimizations described in this paper, or the pipeline transformations described in other work [1, 2, 4], doing so could cause bugs if there are bugs in the underlying elastic behavior. By placing these restrictions on the programmer, we can guarantee that pipelines compiled in Liam behave elastically and can be transformed safely. Section 5 shows how this elastic guarantee can be leveraged by the synthesis and simulation tools.

4.2.2 Inferring Status Logic. In order to synthesize a synchronous pipeline, the Pyrope compiler compiles the code into combinational logic which implements the data operations. When compiling with Liam into a fluid pipeline, the compiler also compiles logic for the status flags, *valid* and *stop*, which drive the fluid registers, based on the Axioms listed in Section 4.1.

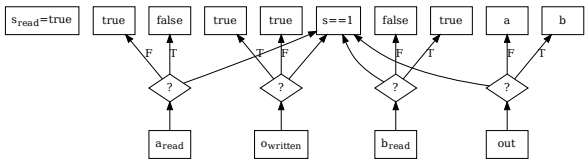

Figure 2: The dataflow graph generated by Listing 3.

The first step to doing this is shown in Listing 6. The compiler inserts flags indicating when input and output ports have been read or written. The compiler then generates a dataflow graph [35] for the stage, shown in Figure 2. In a dataflow graph, the stage's operations are represented as nodes on the graph attached to their dependencies. This is the same method that traditional HDL compilers use to synthesize logic from code. From the final value of each *read* and *write* flag's dataflow net, logic can be generated to determine if the stage will abort, shown in Figure 3. The *abort* flag can then be used to generate logic for each valid and stop output, shown in Figure 4 and 5. These generated logic networks do not add a significant cost in clock frequency, because they can largely be executed in parallel.

Although these equations are designed for implementing the Liam system as a fluid pipeline, they can be adapted to other sorts of elastic system backends as well. Li-BDNs [10] use a queue to provide elastic behavior, and do not account for back pressure. To compile control signals for a Li-BDN system, we set all $output_{stop}$ input signals to constant 0, which simplifies the abort equation, Figure 3. We generate the $output_{valid}$ output signals as before, and have no need to generate the $input_{stop}$ outputs. This fulfills one of core design goals for Liam, which is that Liam code should be independent from the elastic backend.

Figure 3: A boolean flag indicating whether or not the stage will abort can be set with the equation below.

$$abort = (\exists input : \neg input_{valid} \wedge input_{read}) \vee (\exists output : output_{stop} \wedge output_{written}) \quad (1)$$

Figure 4: A given output is valid if the stage did not abort and it has been written.

$$output_{valid} = \neg abort \wedge output_{written} \quad (2)$$

Figure 5: A given input asserts its stop flag if it is valid and either the stage aborted, or it was not read this cycle.

$$input_{stop} = input_{valid} \wedge (abort \vee \neg input_{read}) \quad (3)$$

Listing 6: To compile the switch described in Listing 3, with Liam, the compiler first inserts flags to indicate when IO ports are read or written.

```
1  stage mux(output out,
2    input a, input b, input s):
3    s_read = true
4    if s == 0:
5      a_read = true
6      out = a
7      out_written = true
8    else:
9      b_read = true
10     out = b
11     out_written = true
```

4.3 Memory

Listing 7: A Liam implementation of a register file for a RISCV processor. Note that it is necessary to check for valid data manually, to avoid triggering an abort, but the logic for consuming input can still be handled implicitly.

```
1  stage regfile(
2    outputs data1, data2,
3    inputs addr1, addr2, waddr, wdata):
4
5    int32 @registers[RISCV_REG_COUNT]
6
7    if addr1?: data1 = @registers[addr1]
8    if addr2?: data2 = @registers[addr2]
9
10   if waddr? and wdata?:
11     @registers[waddr] = wdata
```

Listing 7 shows our Liam implementation of a register file for our RISCV processor. We use the @ to mark *register* as a register, and

use the ? operator check whether inputs are valid before reading them, to avoid an abort.

Like other registers, updates to register arrays under Liam happen after the stage executes, and are canceled by an abort.

4.4 Loops

Loops are generally not used in HDL code intended for synthesis, because of the difficulty of managing them with regards to the critical path. A loop for a constant number of cycles can be synthesized, but a loop for a variable number of cycles cannot be synthesized without adding control bits, which is generally done manually. We actually think Liam's latency-independent paradigm would be helpful in this regard, but have not explored the topic at this time.

4.5 Functions and Other Higher-Level Constructs

The processor we implemented for Section 7 was done using six stages, only basic arithmetic expressions, and if-then-else control blocks. Higher level structures, like functions could be added, but they would likely have to be inlined in the calling stage as is done in Verilog and other HDLs. Since the focus of this paper is the elastic semantics, we consider further exploration in this area outside its scope.

4.6 Integrating Liam into other HDLs

This paper uses Pyrope, a Python-like HDL, as a platform for compiling Liam into fluid pipelines. In Pyrope, stages are normally compiled to combinational logic and inserted between clock synchronous registers. When the compiler compiles with Liam enabled, it uses fluid registers instead. To drive the valid and stop status flags of those fluid registers, it analyzes the source of each stage, and generates logic based on when the stage's inputs and outputs are read and written.

Implementing Liam in the traditional HDLs, Verilog and VHDL, would be possible in theory if similar modifications were made to one of their compilers. The compiler must be modified to generate fluid registers, or an equivalent, for module-to-module communication. There would also need to be support for an equivalent to the constructs we added in Table 3, which could possibly be accomplished with domain-specific macros or functions.

The University of California, Berkeley, has developed their own hardware description language, called Chisel [28]. This language is built on Scala, giving it far greater abstraction capabilities than Verilog or VHDL. In Chisel, variables and ports are Scala objects; these types can be inherited from and their operators can be overloaded. PyMTL [36] is another newer HDL developed at Cornell, based on Python, which also boosts greater abstraction and metaprogramming capabilities than Verilog or VHDL. Integrating Liam into Chisel or PyMTL should be far easier for these reasons.

Table 4 proposes some possible syntax for integrating Liam into other HDLs. Verilog would be the most difficult, as it would require macros or some other type of specialized functions. Chisel and PyMTL could likely implement Liam by extending their already existing types.

5 FLUID PIPELINE OPTIMIZATION

This paper makes the case for Liam, an HDL with a programming paradigm that implements elastic behavior implicitly. The reason we are making this case, and the reason we developed Liam, is because we believe that developing large digital architectures under an elastic paradigm can provide a lot of benefits, including optimizations which leverage latency-insensitivity, and cannot be safely done on synchronous pipelines.

Recycling and Retiming are pipeline transformations which can be used to automatically increase or decrease the frequency and area of an Elastic System, without breaking functionality [37]. Prior work has shown that this can be done without a throughput penalty [4].

The Liam paradigm, like the Actor Model it's based on, describes the system as an interaction between atomic units which communicate via message passing. This model of a system provides some inherent benefits toward parallelization, since the system can be reliably broken into isolated, simulatable blocks or regions, which are ideally suited to being integrated into a sort of event/update based queue, commonly used for optimizing on a multi-core system. This is in contrast to traditional HDLs like Verilog and VHDL which compile to a global netlist, and must be updated in its entirety on every state change.

We demonstrate how this can be leveraged in the Evaluation, Section 7. In each of our benchmarks, we evaluation Liam against a non-elastic Verilog implementation, compiled with Verilator and GCC. Compiling Liam into Verilog, and running it in the same manner results in a slightly slower simulation speed due to the added logic required to implement latency-insensitivity, as shown in Table 6. Due to the locality of Liam stages, however, we can also compile the stages as C++ objects which update in an event loop.

Despite the fact that Verilator generates C++, Liam compiled into C++ objects outperforms Liam compiled to Verilog modules. This is because when the system is implemented as C++ objects, the vast majority of the pipeline's operations are localized into the scope of a class method, while in a Verilog implementation the global netlist must be compiled into the same scope. This type of compilation is possible due to Liam's paradigm.

We further leverage the event loop structure that Liam pipelines have when compiled into C++ in Table 7. The testbench is designed to utilize only a small number of stages in the overall FPU pipeline, allowing the event loop based C++ implementation to ignore many of the FPU stages. This gives it a dramatically faster execution speed than the traditional Verilog implementation which needs to update the entire global state every cycle.

Latency independence can also be leveraged to provide other sorts of optimizations that would be difficult to do in an automated manner otherwise. For example, consider an architecture which includes a floating-point unit. Doing an extensive verification on this system would include running on the order of 1-10 million operations. If the floating-point unit does not need to be verified on its own, one idea for an optimization might be to replace the floating point unit with a simple software operation. Switching the floating point unit with a module that does the calculation in software can be safely done automatically in a latency-insensitive pipeline. In a synchronous pipeline, however, there would be a risk of introducing a timing bug.

Table 4: Possible syntax for integrating Liam into other HDLs.

HDL	Valid Check	Retry Check	Keep Statement	Consume Statement
Verilog	*IS_VALID*(*port*)	*IS_STOPPED*(*port*)	*KEEP*(*port*)	*CONSUME*(*port*)
Chisel	*io.port.is_valid*()	*io.port.is_stopped*()	*io.port.keep*()	*io.port.consume*()
PyMTL	*s.port.is_valid*()	*s.port.is_stopped*()	*s.port.keep*()	*s.port.consume*()
Pyrope	port?	port!	keep(port)	consume(port)

6 SETUP

Synthesis was done with the latest Yosys v0.7 using a NanGate 15nm library. Yosys internally uses ABC for synthesis. Delay reported by ABC does not include wire delay, just cell delay. The number of cells reported is the total cell count for the NanGate library. The design was flatten before optimization to include across module optimization. The Fluid Flops used for this implementation are not optimized and use two flops, instead of a master-slave latch which is recommended in Fluid Pipelines. This only accounts for higher flop overhead in the Fluid Pipeline designs.

The C++ benchmarks were compiled with GCC 6.3, and the Verilog were compiled with Verilator 3.9, with the same GCC. They were run on an Intel(R) Xeon(R) CPU E3-1275 v3 at 3.50GHz, and 16GB RAM.

7 EVALUATION

To evaluate Liam, we implemented 3 different RISCV cores that are comparable with existing cores, we also implemented a RISCV core directly in Verilog, using Fluid Pipelines, but not using Liam or Actor Model. We call our implemented cores Cliff. We compare against two public Verilog implementations of similar cores. **Cliff64** is a Liam automatically generated 64bit RISCV core with small internal memories. It is an in order core with 4 pipeline stages. **Cliff** is a Liam automatically generated 32bit RISCV core with small internal memories. Cliff is similar to Cliff64, the only difference is that it is a 32bit core. **Cliff No-Mem** is a Liam automatically generate 32bit RISCV core without internal memory. This core is created to be compared to VScale and PICO32, which also don't have internal memories. **MCliff** is a manually coded 64bit RISCV core with internal memories. This core is similar to Cliff64 with the exception of being handcoded, have one additional pipeline stage and have data hazard detection logic at execute instead of decode. **VScale** is a Berkeley 32bit RISCV core without Fluid Pipelines, it is publicly available and written in Verilog, not Chisel. Unlike the other cores, this is a 3 pipeline in order core without internal memories. **Pico32** is a 32 bit RISCV core written by Clifford Wolf publicly available, it has no Fluid Pipelines and it is coded in Verilog like VScale. It is an in order core with 4 pipeline stages.

Table 5 summarizes the synthesis results showing delay, combinational cells and flops. Pico32 is a little bit slower than Cliff No-Mem, but for synthesis variation reasons could be considered equal. This means that a Fluid Pipeline automatically generated with Liam has no timing overhead versus a handcoded optimized Verilog. Pico32 and VScale have the lowest number of combinational cells and flops. Cliff No-Mem, which is the equivalent core, has higher combinational cells and flops. It has more flops in part because the Fluid Flops used have two internal flops instead of a master-slave latch. If such library was available and used, we would expect a number of flops in Cliff No-Mem to be between Pico32 and VScale. The number of logic cells is around 15% more in Cliff No-Mem than in VScale. We think that the main reason for that is because VScale has one less pipeline stage than Cliff No-Mem which allows for more logic optimization. When comparing to Pico32, the main reason is because Pico32 is highly optimized. Another reason is that there is a small number of cells added to handle the handshake of Fluid Pipelines, but as the timing results show, those are not in the critical path. Cliff64 and MCliff are very similar cores with the main difference being that MCliff was handcoded in Verilog not using actor model. Liam automatically generated code is faster and smaller. We see this as an indication the automatically generated Liam code does not add overheads, and equally important using the Liam Fluid Pipeline actor model is as efficient or more than non-actor model Fluid Pipeline implementation.

Table 5: Synthesis results for the CPU benchmarks.

Core	Delay (ps)	Comb Cells	Flops
Cliff64	247	43195	6102
Cliff	112	22610	3190
Cliff No-Mem	124	14163	2153
MCliff	299	46344	7272
VScale	242	12239	1862
Pico32	130	11046	1545

We then evaluate the simulation results of a set of three simple benchmarks in Table 6: a greatest-common divisor (GCD) benchmark, which computes the greatest common divisor of two numbers; a simple ring network of four nodes, each of which can be sent a packet, on which it will do a computation and pass it back to the sender; and a few RISCV CPUs.

Liam's Verilog implementation is consistently a little slower than the non-fluid pipeline Verilog implementation due to the added overhead required for simulating fluid registers. The C++ implementation is consistently a little faster, likely due to inherent advantages native C++ has over Verilator-generated C++.

Due to the nature of Verilog, simulation requires updating the entire system every clock cycle. Although Verilator compiles to C++ and compiles with the same version of GCC, the global nature of Verilog requires that the resulting program keep a large amount of memory in a global scope, which impacts execution speed. When Liam is used to generate C++, it leverages the fact that stages are atomic blocks that only communicate through message passing. This allows C++ to compile objects which do computations locally, better utilizing CPU resources and providing a boost in execution speed. This type of compilation can be safely done due to the structure of the Pyrope/Liam programming model.

As an additional demonstration of Liam, we present an FPU benchmark which is designed to take advantage of Liam's C++ compilation target. Table 7 compares a normal (non-fluid) Verilog FPU against an FPU created in Pyrope/Liam. We created such a stark difference in execution speed by only sending floating point add operations into the benchmark, and compiling Liam into a C++ backend that only updates stages when they receive valid data. Since the different FPU instructions are sent along different paths in the pipeline, Liam's C++ FPU only executes stages related to the FPU add operation, while the Verilog FPU still needs to update its entire global state. Although these types of optimizations are common in high-level hardware simulators, the Pyrope/Liam compiler can implement it with unmodified, synthesizable code.

Table 6: Benchmark results in MHz. Liam is compiled into both C++ and Verilog, and is compared against a similar Verilog implementation. The entries with the (*) are non-fluid implementations

	Liam C++	Liam Verilog	Verilog
Ring Network	46.12	23.56	38.79*
GCD	73.14	28.94	71.85*
Cliff64	9.42	5.32	N/A
MCliff	N/A	N/A	12.45
VScale	N/A	N/A	8.80*

Table 7: Benchmark results in MHz for the FPU testbench. This testbench was designed to take full advantage of Liam's C++ compilation target.

Non-fluid Verilog FPU	Liam FPU
1.92	10.96

8 CONCLUSION

In this paper we present Liam, a new paradigm for hardware description languages which implements elastic behavior implicitly. We propose Liam because we feel that a major factor which discourages the use of latency-insensitive systems in digital hardware design is the complexity they add, due to the additional logic and backend infrastructure required to implement them. Liam addresses this by adding a set of axioms, in Section 4.1, which describes elastic behavior implicitly. When the HDL source code is compiled, the behavior described by those axioms is compiled into logic which drives the control signals for the latency-insensitive backend infrastructure. This both frees the developer from having to implement the backend protocol manually, and allows Liam source code to be shared between different types of latency-insensitive pipelines. We demonstrate and evaluate Liam by integrating it into a specialized HDL called Pyrope, but also describe how it may be added to other HDL toolchains.

Our evaluation shows that Liam/Pyrope does not add overheads in the implemented core either in timing or area when comparing against an equivalent Fluid Pipeline core implemented by hand in Verilog. When comparing Fluid Pipelines versus non-Fluid Pipelines, we observed a small overhead in area, but with no overhead in delay.

Future work on this project should include integrating the pipeline transformations described in prior work [4] to the Liam toolchain which could allow for seamless, high-level manipulation of the pipeline critical path/clock frequency without any cost in throughput. We also believe that Liam could provide an interface to allow computer architectures to be better integrated into other types of concurrency and real time distributed system analysis and simulation tools [13–15] because it represents a high-level description of latency-insensitive behavior.

As digital architectures continue to grow in size and complexity, we believe that there are a lot of advantages in using latency-insensitive design paradigms. Developing a platform to describe latency-insensitive behavior at a high level is a key step in leveraging these advantages.

REFERENCES

[1] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic Systems," in *Proceedings of the 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, ser. MEMOCODE '10, Jul. 2010, pp. 149–158.

[2] J. Cortadella, M. Kishinevsky, and B. Grundmann, "SELF: Specification and Design of Synchronous Elastic Circuits," in *Proceedings of the ACM/IEEE International Workshop on Timing Issues*, ser. TAU âĂŹ06, 2006.

[3] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proceedings of the 37th Design Automation Conference*. New York, NY, USA: ACM, 2000, pp. 361–367.

[4] R. T. Possignolo, E. Ebrahimi, H. Skinner, and J. Renau, "FluidPipelines: Elastic circuitry meets out-of-order execution," in *Computer Design (ICCD), Proceedings of the 34th International Conference on*, October 2016.

[5] M. Vijayaraghavan and A. Arvind, "Bounded Dataflow Networks and Latency-Insensitive Circuits," in *Proceedings of the 7th IEEE/ACM Int'l Conf. on Formal Methods and Models for Codesign*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 171–180.

[6] R. T. Possignolo, E. Ebrahimi, H. Skinner, and J. Renau, "FluidPipelines: Elastic circuitry without throughput penalty," in *Logic Synthesis (IWLS), Proceedings of the 2016 International Workshop on*, June 2016.

[7] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-ports: an efficient abstraction for cycle-accurate performance models on fpgas," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 87–96.

[8] M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August, "Microarchitectural exploration with liberty," in *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, 2002, pp. 271–282.

[9] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani, "Unisim: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, 2007.

[10] T. Harris, Z. Ruan, and D. Penry, "Techniques for li-bdn synthesis for hybrid microarchitectural simulation." *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, October 2011.

[11] "Opencores.org," http://opencores.org/.

[12] C. Hewitt, P. Biship, and Steiger, "A universal modular actor formalism for artificial intelligence," *Proceedings of the 3rd international joint conference on Artificial intelligence*, 1973.

[13] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 01, pp. 1–72, 1997.

[14] "Pulsar, a python concurrency framework," http://quantmind.github.io/pulsar/, accessed: 2017-04-22.

[15] "Akka: A concurrency framework for java and scala," http://akka.io/, accessed: 2017-05-25.

[16] "Orbit: a framework for writing distributed systems using virtual actors," https://github.com/orbit/orbit/wiki, accessed: 2017-04-22.

[17] "Labview actor framework "white paper"," https://forums.ni.com/ni/attachments/ni/7301/130/1/Using2011.

[18] "Pykka: A python implementation of the actor model," http://ptolemy.eecs.berkeley.edu/, accessed: 2017-05-25.

[19] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," *13th International Conference on Embedded Software (EMSOFT)*, 2013.

[20] "C++ actor framework," http://actor-framework.org/, accessed: 2017-04-22.

[21] "Pykka: A python implementation of the actor model," https://www.pykka.org/en/latest/, accessed: 2017-05-25.
[22] "Aojet, an actor model framework for swift," https://github.com/aojet/Aojet, accessed: 2017-04-22.
[23] J. Bergeron, *Verification Methodology Manual for SystemVerilog*. Springer, 2006. [Online]. Available: http://books.google.com/books?id=dcET3kKtmH4C
[24] "Vhsic hardware description language," https://en.wikipedia.org/wiki/VHDL, 2017-05-25.
[25] L. P. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A Methodology for Correct-by-construction Latency-insensitive Design," in *Computer-Aided Design. Int'l Conf. on*, 1999, pp. 309 –315.
[26] J. Julvez, J. Cortadella, and M. Kishinevsky, "Performance analysis of concurrent systems with early evaluation," in *Computer-Aided Design. Int'l Conf. on*, November 2006, pp. 448–455.
[27] M. Vachharajani, N. Vachharajani, D. Penry, J. B. andS. Malik, and D. August, "The Liberty Simulation Environment: A Deliberate Approach toHigh-Level System Modeling," Princeton University, Technical Report, March 2004.
[28] J. e. a. Bachrach, "Chisel: Constructing hardware in a scala embedded language." *Design Automation Conference (DAC)*, 2012.
[29] R. Nikhil, "Bluespec system verilog: efficient, correct RTL from high level specifications," *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70, Jun. 2004.
[30] "The esterel programming language," http://www-sop.inria.fr/meije/esterel/esterel-eng.html, accessed: 2017-05-08.
[31] S. A. Edwards, "High-level synthesis from the synchronous language esterel." in *IWLS*. Citeseer, 2002, pp. 401–406.
[32] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: a declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '87. New York, NY, USA: ACM, 1987, pp. 178–188. [Online]. Available: http://doi.acm.org/10.1145/41625.41641
[33] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.
[34] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 89–108.
[35] R. Cytron, J. Ferrante, B. Rossen, M. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, 1991.
[36] D. Lockhart, G. Zibrat, and C. Batten, "Pymtl: A unified framework for vertically integrated computer architecture research," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 280–292.
[37] D. Bufistov, J. Cortadella, M. Galceran-Oms, J. Julvez, and M. Kishinevsky, "Retiming and recycling for elastic systems with early evaluation," in *46th Design Automation Conference*, 2009, pp. 288–291.