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

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, Vienna, Austria, September 29-October 2, 2017,* 4 pages.
https://doi.org/10.1145/3127041.3127060

1 LIAM

The Liam paradigm is a collection of constructs aimed at managing latency-insensitive behavior. These constructs are language-agnostic, although implementing them requires integrating them into the compilater. For this paper, we use a custom, Python-like HDL called Pyrope to both provide concise code samples, and because the Pyrope compiler was designed with Liam-integration in mind.

1.1 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, and generates logic to drive the fluid registers' control signals based on the axioms described below.

Listing 1: A Pyrope implementation of a mux, which when compiled with Liam enabled will act more like a switch. Liam's axioms 1 (Atomicity) 2 (Abortion) 3 (Isolation) add additional behavior to the stage, guaranteeing it will wait for valid data before producing a result, and consume any inputs once they are used.

```
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 1 shows a Pyrope implementation of a mux, which when compiled with Liam will act more like a switch.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*. Liam's axioms 1 (Atomicity) 2 (Abortion) 3 (Isolation) add elastic behavior to the system by controlling when the stage updates its outputs and internal registers, referred to collectively as its state. As stated in Axiom 1 (Atomicity), the stage is evaluated like a function, from the top down. If it completes the state is updated. If it reads an input that is invald or writes an output with back pressure, execution aborts and the state is left unchanged.

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.*

1.2 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 zero.

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.

1.3 Added Constructs

Liam adds some new constructs to allow the programmer to manage elastic behavior. These are listed in Table 1. They allow the programmer to control elastic behavior abstractly, independent from any one specific type of backend.

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 Section 3.

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 for readability.

Using these additional constructs, we can modify the multiplexer from Listing 1 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,

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

Table 1: 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, meaning back-pressure.
Keep Statement	keep *input*	Prevents an input from being consumed.
Consume Statement	consume *input*	Explicitly consume an input.

but produce output regardless. This is implemented in Listing 2. By using the valid operator, we check whether the non-selected data port is valid before reading, preventing an abort. By contrast, Listing 3 assures that all inputs will be read every cycle. This means the stage will only produce output when all of its inputs are valid.

Listing 2: 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 3: 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 s == 0:
3      out = a
4      consume b
5    else:
6      out = b
7      consume a
```

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 that would break latency-insensitivity. A major goal of this project is to provide a platform to allow programmers to take advantage of pipeline transformations and other optimizations described in prior work [1, 3, 5]. These depend on the system being latency-insensitive. By placing these restrictions on the programmer, we can guarantee that pipelines compiled in Liam behave elastically and can be transformed safely.

1.4 Inferring Status Logic

There are a variety of ways to implement latency-insensitivity in hardware, which generally boil down to using some type of simple handshake [5] or queues [4] between stages. A major goal for this project was a method of implementing elastic systems independent of the underlying infrastructure.

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 [5], as we do for this paper, the compiler also compiles logic for the status flags, *valid* and *stop*, which drive the fluid registers, based on the Axioms listed in Section 1.1.

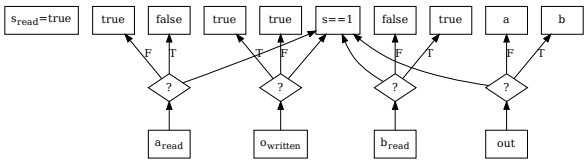

Figure 1: The dataflow graph generated by Listing 1.

The first step to doing this is shown in Listing 4. The compiler inserts flags indicating when input and output ports have been read or written. The compiler then generates a dataflow graph [2] for the stage, shown in Figure 1. 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 2. The *abort* flag can then be used to generate logic for each valid and stop output, shown in Figure 3 and 4. 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 [4] 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 2. We generate the $output_{valid}$ output signals as before, and have no need to generate the $input_{stop}$ outputs.

Listing 4: To compile the switch described in Listing 1, 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
```

Figure 2: 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 3: 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 4: A given input asserts its stop flag if it is valid and either the stage aborted, or it was not read this cycle.

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

1.5 Memory

Listing 5: 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? and not data1!:
8      data1 = @registers[addr1]
9
10   if addr2? and not data2!:
11     data2 = @registers[addr2]
12
13   if waddr? and wdata?:
14     @registers[waddr] = wdata
```

In some cases, the default behavior of Liam may not be desirable. Listing 5 shows a register file for a 32-bit RISCV processor with two read ports and one write port. We want this stage to execute the reads and writes for the valid data it receives, and ignore the invalid ones. If we allow the stage to abort, that will mean no inputs or outputs are consumed or updated, which is not ideal. Instead, we insert manual checks to make sure all reads and writes are only evaluated if they can be done safely. Liam still provides us some assistance, though, since the logic for consuming inputs and holding of outputs (if the stage receiving the output asserts back pressure) is handled by the compiler.

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

2 SETUP

Synthesis was done with the latest Yosys v0.7 using a NanGate 15nm library, which internally uses ABC for synthesis. Delay reported by ABC does not include wire delay, just cell delay, and the number of cells reported is the total cell count for the NanGate library. The design was flattened before optimization to allow for across module optimization. The Fluid Flops used for this implementation are not optimized, they use two flops, instead of a master-slave latch, which is recommended in the fluid pipeline paper [5]. This 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.

3 EVALUATION

To evaluate Liam, we implemented three different RISCV cores that are comparable with existing RISCV implementations, we also implemented a RISCV core directly in Verilog, using Fluid Pipelines, but not using Pyrope and Liam. We call our implemented cores Cliff. We compared these against two public Verilog implementations of similar cores. **Cliff64** is a Pyrope+Liam 64bit RISCV core with small internal memories. It is an in-order core with four pipeline stages. **Cliff** is a Pyrope+Liam 32bit RISCV core with small internal memories. Cliff is similar to Cliff64, but is 32-bit. **Cliff No-Mem** is a Pyrope+Liam 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, it also has one additional pipeline stage and some 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. Unlike the other cores, this is a three 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 four pipeline stages.

Table 2 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

Table 2: 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

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. We compare against Pico32, mainly 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. Pyrope+Liam automatically generated code is faster and smaller. We see this as an indication the automatically generated Liam code does not add overhead, and equally important using the Liam Fluid Pipeline actor model is as efficient or more than non-actor model Fluid Pipeline implementation.

We then evaluate the simulation results of a set of three simple benchmarks in Table 3: 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, which suggests Pyrope+Liam generated C++, where blocks are implemented as class objects which communicate only through their pre-defined IO ports, and only on positive clock edges, has some advantages over the Verilator-generated C++ which implements a global netlist.

Table 3: 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*

4 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 1.1, 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 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 overhead 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 [6] to the Liam toolchain which could allow for seamless, high-level manipulation of the pipeline critical path/clock frequency without any cost in throughput.

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.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] J. Cortadella, M. Kishinevsky, and B. Grundmann. 2006. SELF: Specification and Design of Synchronous Elastic Circuits. In *TAU'06.*

[2] Ron Cytron, Jeanne Ferrante, Barry Rossen, Mark Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* (1991).

[3] J. Cortadella et. al. 2010. Elastic Systems. In *MEMOCODE'10.* 149–158.

[4] T.S. et. al. Harris. 2011. Techniques for Li-BDN Synthesis for Hybrid Micro-Architectural Simulation. *ICCD'11* (October 2011).

[5] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. 2016. FluidPipelines: Elastic Circuitry meets Out-of-Order Execution. In *ICCD'16.*

[6] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. 2016. FluidPipelines: Elastic Circuitry meets Out-of-Order Execution. In *Computer Design (ICCD), Proceedings of the 34th International Conference on.*