✓ Swiss Verification Day 2025

13 February 2025, USI East Campus, Lugano

Reasoning about out-of-order execution in dataflow circuits

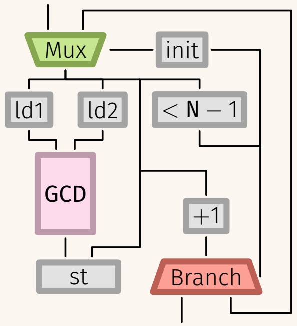
Yann Herklotz, Ayatallah Elakhras, Martina Camaioni, Lana Josipović, Paolo Ienne, Thomas Bourgeat

February 13, 2025

High-level synthesis with dataflow graphs

```
for (int i = 0; i < N; i++) {
  result[i] = gcd(arr1[i], arr2[i]);
}
```

Dynamic high-level synthesis tools translate C code into dataflow circuits.



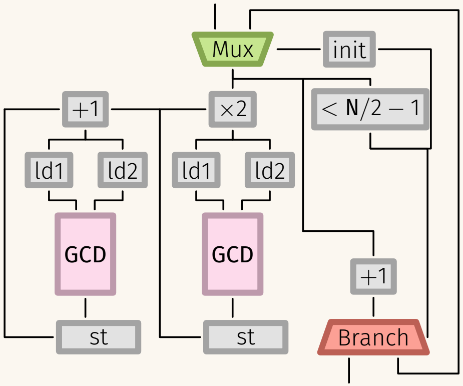
max 1 GCD operation at a time.

High-level synthesis with dataflow graphs

```
for (int i = 0; i < N/2; i++) {
  result[2*i] =
    gcd(arr1[2*i], arr2[2*i]);
  result[2*i+1] =
    gcd(arr1[2*i+1], arr2[2*i+1]);
}
```

Dynamic high-level synthesis tools translate C code into dataflow circuits.

Manual optimisations can help throughput.

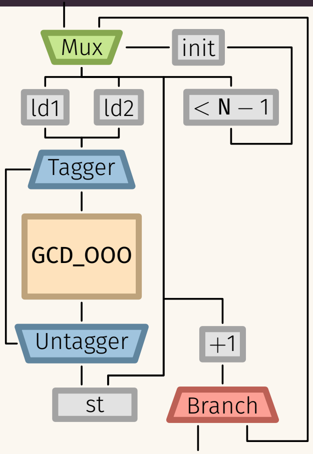


up to 2 GCD operation at a time.

Out-of-order in dataflow circuits

We can generalise the arbiter idea by defining and out-of-order GCD component.

A more general tagger/untagger pair keeps track of the correct order of tokens.



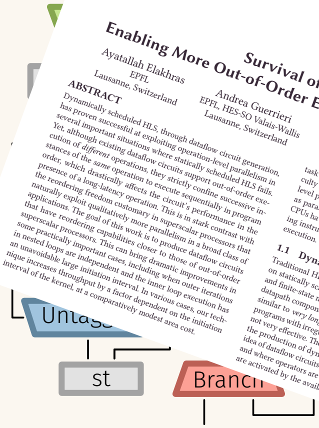
n GCD operation at a time.

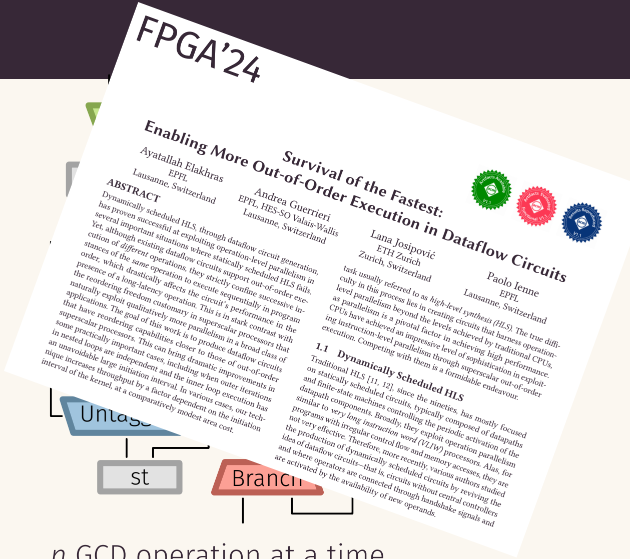
Out-of-order in dataflow circuits

We can generalise the arbiter idea by defining and out-of-order GCD component.

A more general tagger/untagger pair keeps track of the correct order of tokens.

Systematic approach developed by the Dynamatic lab.





FPGA'24

Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits

Ayatallah Elakhras
EPFL
Lausanne, Switzerland

Andrea Guerrieri
EPFL, HES-SO Valais-Wallis
Lausanne, Switzerland

Lana Josipović
ETH Zurich
Zurich, Switzerland

Paolo Ienne
EPFL
Lausanne, Switzerland

ABSTRACT

Dynamically scheduled HLS, through dataflow circuit generation, has proven successful at exploiting operation-level parallelism in several important situations where statically scheduled HLS fails. Yet, although existing dataflow circuits support out-of-order execution of *different* operations, they strictly confine successive instances of the *same* operation to execute sequentially in program order, which drastically affects the circuit's performance in the presence of a long-latency operation. This is in stark contrast with the reordering freedom customary in superscalar processors that naturally exploit qualitatively more parallelism in a broad class of applications. The goal of this work is to produce dataflow circuits that have reordering capabilities closer to those of out-of-order superscalar processors. This can bring dramatic improvements in some practically important cases, including when outer iterations in nested loops are independent and the inner loop execution has an unavoidable large initiation interval. In various cases, our technique increases throughput by a factor dependent on the initiation interval of the kernel, at a comparatively modest area cost.

task usually referred to as *high-level synthesis (HLS)*. The true difficulty in this process lies in creating circuits that harness operation-level parallelism beyond the levels achieved by traditional CPUs, as parallelism is a pivotal factor in achieving high performance. CPUs have achieved an impressive level of sophistication in exploiting instruction-level parallelism through superscalar out-of-order execution. Competing with them is a formidable endeavour.

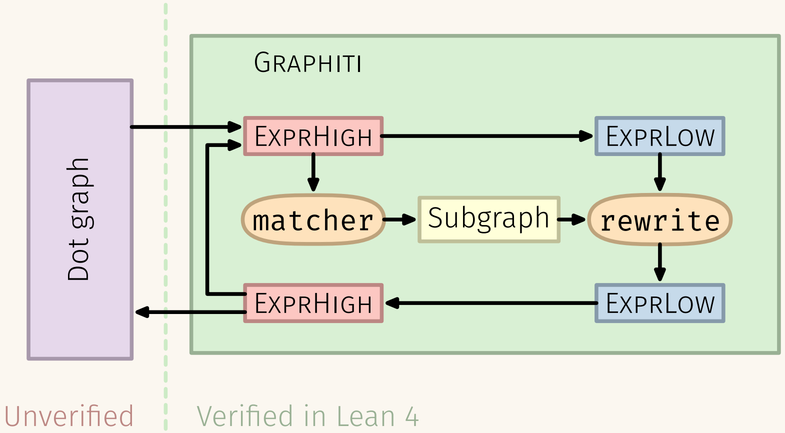
1.1 Dynamically Scheduled HLS

Traditional HLS [11, 12], since the nineties, has mostly focused on statically scheduled circuits, typically composed of datapaths and finite-state machines controlling the periodic activation of the datapath components. Broadly, they exploit operation parallelism similar to *very long instruction word (VLIW)* processors. Alas, for programs with irregular control flow and memory accesses, they are not very effective. Therefore, more recently, various authors studied the production of dynamically scheduled circuits by reviving the idea of dataflow circuits—that is, circuits without central controllers and where operators are connected through handshake signals and are activated by the availability of new operands.

n GCD operation at a time.

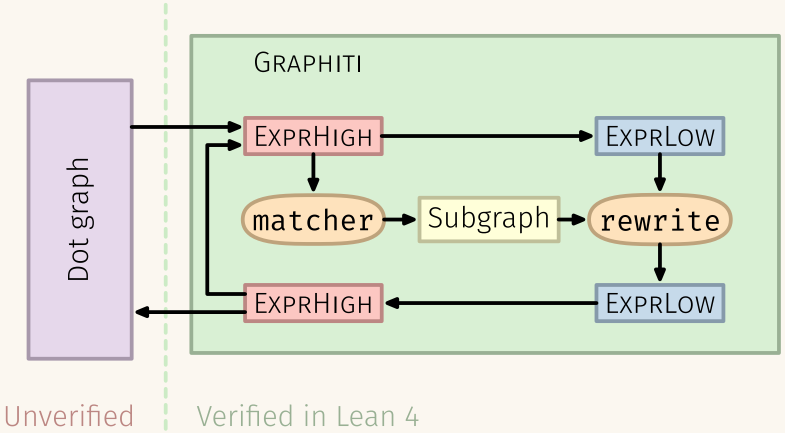
Ayatallah Elakhras, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. **"Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits".** In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 44–54. ISBN: 9798400704185. DOI: 10.1145/3626202.3637556

Rewrites are a nice way to perform optimisations in a verified framework



Started collaborating on a verified dataflow rewrite framework.

Rewrites are a nice way to perform optimisations in a verified framework



Started collaborating on a verified dataflow rewrite framework.

Use the rewrite framework to implement the FPGA'24 algorithm.

Quick overview of the syntax and semantics

Language descriptions

`ExprHigh` Graph language represented by sets of nodes and edges.

`ExprLow` Inductive graph language with products and connections.

Graph denotation and semantics

```
def join T T' : NatModule (List T × List T') :=
  { inputs := [ (0, ( T, fun ol el nl => nl.1 = ol.1.concat el /\ nl.2 = ol.2))
              , (1, ( T', fun ol el nl => nl.2 = ol.2.concat el /\ nl.1 = ol.1))].toAssocList,
    outputs := [(0, ( T × T', fun ol el nl => ol.1 = el.1 :: nl.1 /\ ol.2 = el.2 :: nl.2 ))].toAssocList }
```

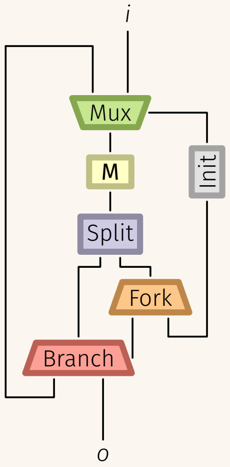
Top-level correctness theorem

$$(rw_{\mathrm{rhs}} \Downarrow \varepsilon) \sqsubseteq (rw_{\mathrm{lhs}} \Downarrow \varepsilon) \rightarrow$$

$$(e[rw_{\mathrm{lhs}} := rw_{\mathrm{lhs}}] \Downarrow \varepsilon) \sqsubseteq (e \Downarrow \varepsilon)$$

In particular we only need one general rewrite replicate many of the FPGA'24 results

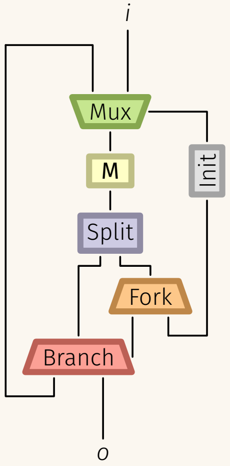
Given a **sequential** loop over an arbitrary loop body *M*.

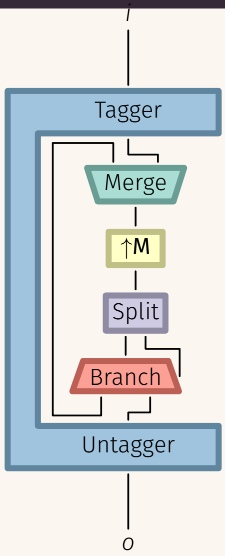


In particular we only need one general rewrite replicate many of the FPGA’24 results

Given a **sequential** loop over an arbitrary loop body *M*.

We can generate an **out-of-order** version of the loop by **overlapping** loop executions.



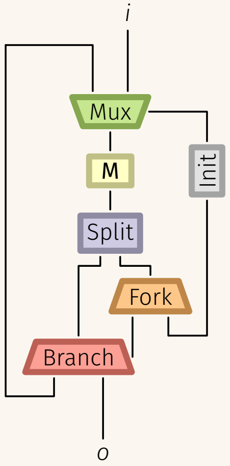


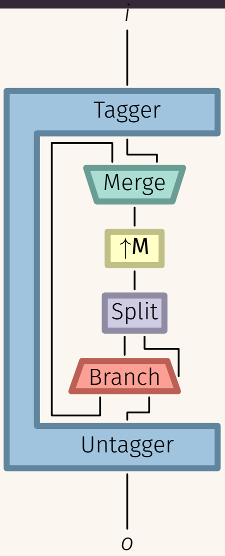
In particular we only need one general rewrite replicate many of the FPGA'24 results

Given a **sequential** loop over an arbitrary loop body *M*.

We can generate an **out-of-order** version of the loop by **overlapping** loop executions.

Does this hold for an **arbitrary** module M?





In particular we only need one general rewrite replicate many of the FPGA'24 results

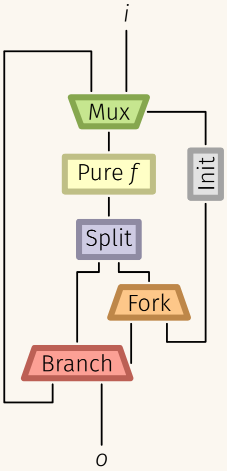
Given a sequential loop over an arbitrary loop body *M*.

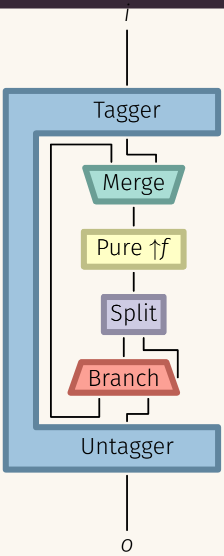
We can generate an out-of-order version of the loop by overlapping loop executions.

Does this hold for an arbitrary module M?

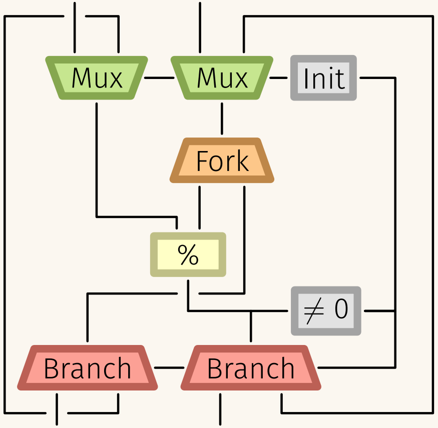
No, we need:

- 1-in-1-out behaviour for the module.
- It needs to be stateless.



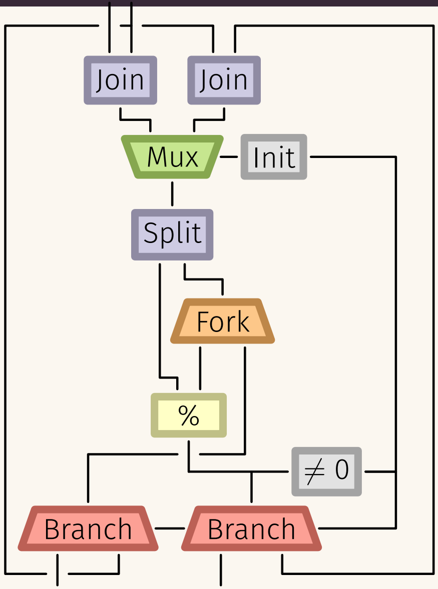


Applying the rewrite to the GCD example



We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

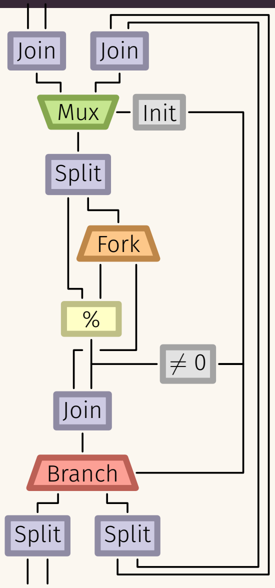
Applying the rewrite to the GCD example



We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

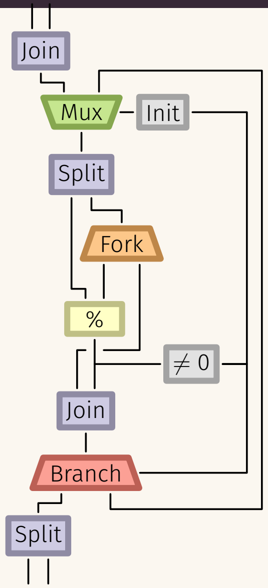
Applying the rewrite to the GCD example



We have a problem: too many muxes and branches, and we don’t know whether the body can be modelled as a pure function.

Combine muxes and branches.

Applying the rewrite to the GCD example

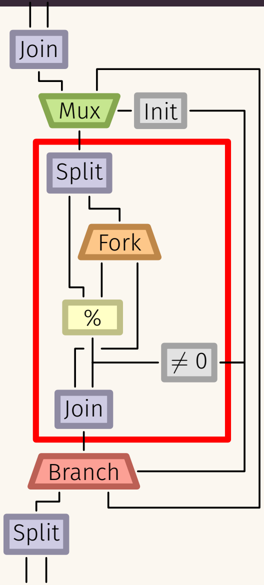


We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Applying the rewrite to the GCD example



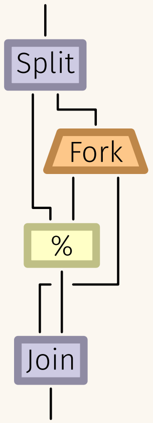
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Applying the rewrite to the GCD example



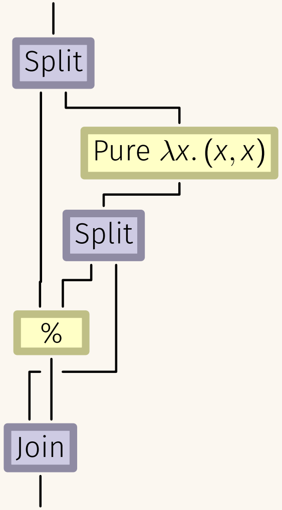
We have a problem: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: More rewrites!

Applying the rewrite to the GCD example



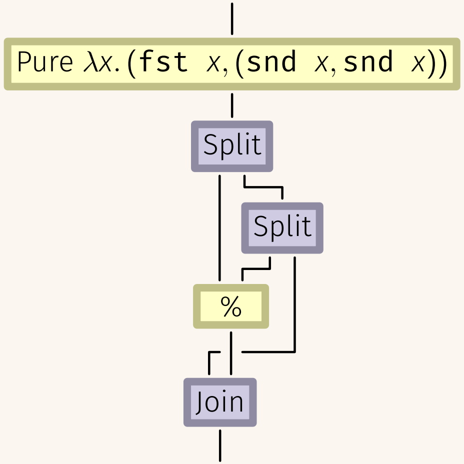
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Applying the rewrite to the GCD example


Pure $\lambda x.(\texttt{fst}\ x, (\texttt{snd}\ x, \texttt{snd}\ x))$
Split
Split
%
Join

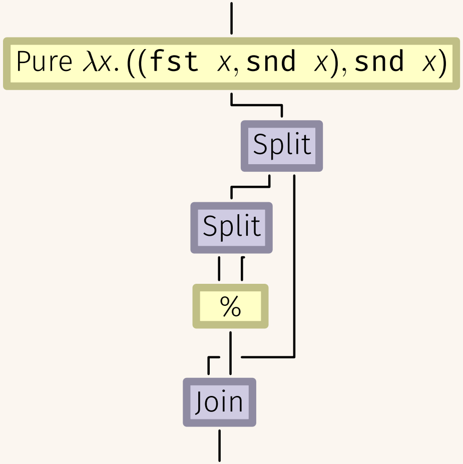
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Applying the rewrite to the GCD example


Pure $\lambda x.((\texttt{fst}\ x, \texttt{snd}\ x), \texttt{snd}\ x)$
Split
Split
%
Join

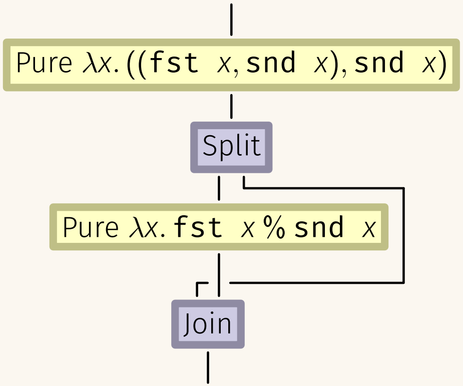
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Applying the rewrite to the GCD example


Pure $\lambda x.\,((\texttt{fst}\ x, \texttt{snd}\ x), \texttt{snd}\ x)$
Split
Pure $\lambda x.\,\texttt{fst}\ x \,\%\, \texttt{snd}\ x$
Join

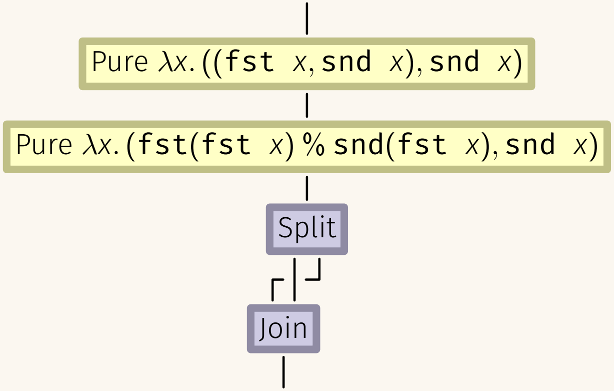
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Applying the rewrite to the GCD example


Pure $\lambda x.((\texttt{fst}\ x, \texttt{snd}\ x), \texttt{snd}\ x)$
Pure $\lambda x.(\texttt{fst}(\texttt{fst}\ x) \% \texttt{snd}(\texttt{fst}\ x), \texttt{snd}\ x)$
Split
Join

We have a problem: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: More rewrites!

Applying the rewrite to the GCD example

$$\begin{tikzcd}
{} \arrow[d, no head] \\
\boxed{\mathrm{Pure}\ \lambda x.\,((\texttt{fst}\ x, \texttt{snd}\ x), \texttt{snd}\ x)} \arrow[d, no head] \\
\boxed{\mathrm{Pure}\ \lambda x.\,(\texttt{fst}(\texttt{fst}\ x)\ \%\ \texttt{snd}(\texttt{fst}\ x), \texttt{snd}\ x)} \arrow[d, no head] \\
\boxed{\mathrm{Pure}\ \lambda x.\,(\texttt{snd}\ x, \texttt{fst}\ x)} \arrow[d, no head] \\
{}
\end{tikzcd}$$

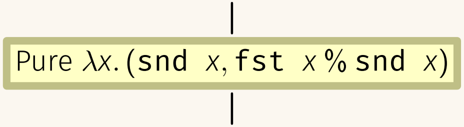
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Applying the rewrite to the GCD example


Pure $\lambda x.(\texttt{snd}\ x, \texttt{fst}\ x \,\%\, \texttt{snd}\ x)$

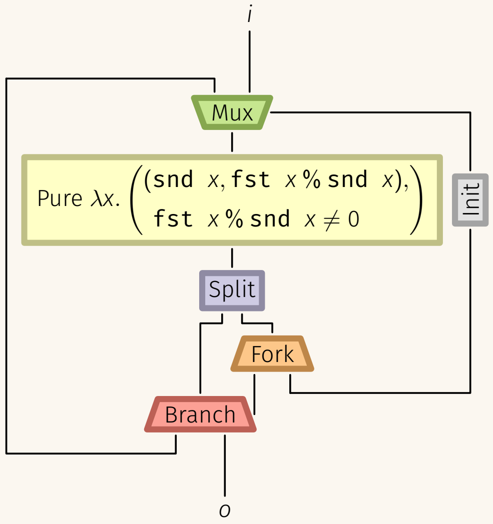
We **have a problem**: too many muxes and branches, and we don't know whether the body can be modelled as a pure function.

Combine muxes and branches.

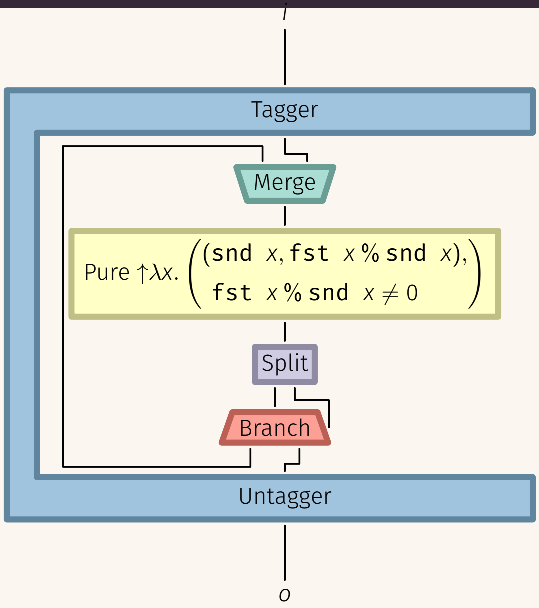
Remove unnecessary splits and joins.

Prove that the body of the loop is `Pure`: **More rewrites!**

Finally we can parallelise our GCD!



i
Mux
Pure $\lambda x. \begin{pmatrix} (\texttt{snd}\ x, \texttt{fst}\ x \,\%\, \texttt{snd}\ x), \\ \texttt{fst}\ x \,\%\, \texttt{snd}\ x \neq 0 \end{pmatrix}$
Init
Split
Fork
Branch
o



i
Tagger
Merge
Pure $\uparrow\lambda x. \begin{pmatrix} (\texttt{snd}\ x, \texttt{fst}\ x \,\%\, \texttt{snd}\ x), \\ \texttt{fst}\ x \,\%\, \texttt{snd}\ x \neq 0 \end{pmatrix}$
Split
Branch
Untagger
o

Hypothetical results

Benchmark	N	Cycle count		CP (ns)		Execution time (μs)			LUTs			FFs			DSPs
		[18]	Ours	[18]	Ours	[18]	Ours		[18]	Ours		[18]	Ours		
img-avg	4	1,722	634	6.42	8.21	11.05	5.21	**2.1x**	1,415	1,593	**+13%**	1,320	1,206	**-9%**	5
gsum_many	10	68,523	32,874	7.91	9.95	541.81	327.0	**1.7x**	2,835	3,657	**+29%**	3,256	3,725	**+14%**	22
gsum_single	10	6,703	9,333	6.90	9.01	46.25	84.11	**0.55x**	2,736	2,677	**-2%**	3,142	3,114	**-1%**	22
gemm	20	68,825	8,144	6.75	12.81	464.78	104.30	**4.5x**	3,214	5,937	**+85%**	2,693	3,688	**+37%**	11
matvec	50	7,936	918	6.41	13.51	50.86	12.40	**4.1x**	1,272	4,396	**+246%**	1,373	3,423	**+149%**	5
mvt	10	7,940	2,044	6.27	11.70	49.79	23.92	**2.1x**	2,886	5,544	**+92%**	2,701	3,730	**+38%**	10
bicg	10	7,936	1,000	6.43	11.27	51.06	11.27	**4.5x**	2,051	3,229	**+57%**	2,182	2,737	**+25%**	10

Table 1. Ayatallah Elakhras, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. **"Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits".** In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 44–54. ISBN: 9798400704185. DOI: `10.1145/3626202.3637556`

Conclusion and future (current) work

Conclusion

We verify a rewrite framework for dataflow circuits in Lean 4.

We implement one main rewrite to introduce out-of-order execution.

Future work

Still a work in progress, finish proofs and gather proper results.

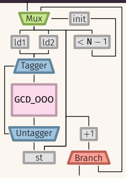
Develop better proof methods for verifying rewrites.

Hopeful we can extend the framework to other dataflow graphs (circuits?).

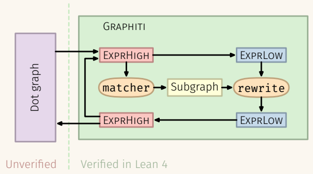
Out-of-order in dataflow circuits

We can generalise the arbiter idea by defining and out-of-order GCD component.

A more general tagger/untagger pair keeps track of the correct order of tokens.



Rewrites are a nice way to perform optimisations in a verified framework



Started collaborating on a verified dataflow rewrite framework.

Use the rewrite framework to implement the FPGA'24 algorithm.

Thanks! Any questions?

In particular we only need one general rewrite replicate many of the FPGA'24 results

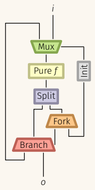
Given a sequential loop over an arbitrary loop body *M*.

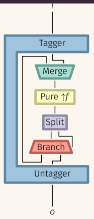
We can generate an out-of-order version of the loop by overlapping loop executions.

Does this hold for an arbitrary module M?

No, we need:

- 1-in-1-out behaviour for the module.
- It needs to be stateless.





Hypothetical results

Benchmark	*N*	Cycle count		CP (ns)		Execution time (*μs*)			LUTs			FFs			DSPs
		[18]	Ours	[18]	Ours	[18]	Ours		[18]	Ours		[18]	Ours		
img-avg	4	1,722	634	6.42	8.21	11.05	5.21	**2.1x**	1,415	1,593	**+13%**	1,320	1,206	**-9%**	5
gsum_many	10	68,523	32,874	7.91	9.95	541.81	327.0	**1.7x**	2,835	3,657	**+29%**	3,256	3,725	**+14%**	22
gsum_single	10	6,703	9,333	6.90	9.01	46.25	84.11	**0.55x**	2,736	2,677	**-2%**	3,142	3,114	**-1%**	22
gemm	20	68,825	8,144	6.75	12.81	464.78	104.30	**4.5x**	3,214	5,937	**+85%**	2,693	3,688	**+37%**	11
matvec	50	7,936	918	6.41	13.51	50.86	12.40	**4.1x**	1,272	4,396	**+246%**	1,373	3,423	**+149%**	5
mvt	10	7,940	2,044	6.27	11.70	49.79	23.92	**2.1x**	2,886	5,544	**+92%**	2,701	3,730	**+38%**	10
bicg	10	7,936	1,000	6.43	11.27	51.06	11.27	**4.5x**	2,051	3,229	**+57%**	2,182	2,737	**+25%**	10

Table 1. Ayatallah Elakhras, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. **"Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits".** In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 44–54. ISBN: 9798400704185. DOI: 10.1145/3626202.3637556

References i

Elakhras, Ayatallah, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. **“Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits”.** In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’24. Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 44–54. ISBN: 9798400704185. DOI: 10.1145/3626202.3637556.