Tricks of Computer Architects and Inductive Refinement Maps

Thomas, Arvind

OPLSS

Outline

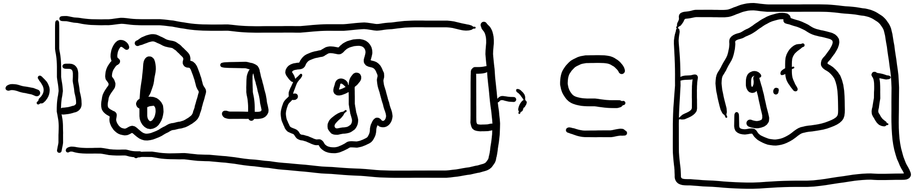
"There is plenty of room at the top" a bit more motivation for proving correctness of architecture

A nice way to do refinement maps - Inductive Refinement Maps

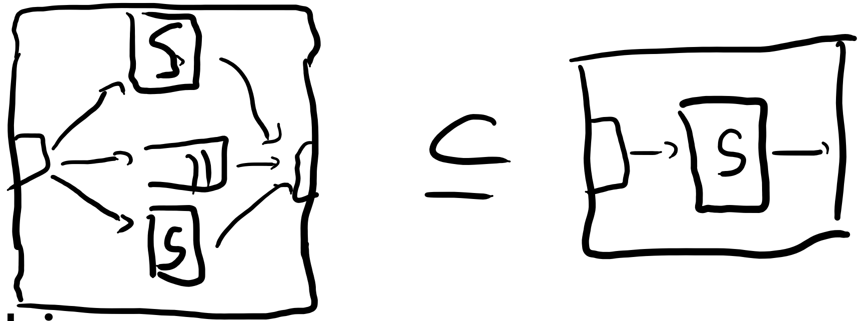
Maybe 3. The Problem

Tricks of Architects – Codex Preview

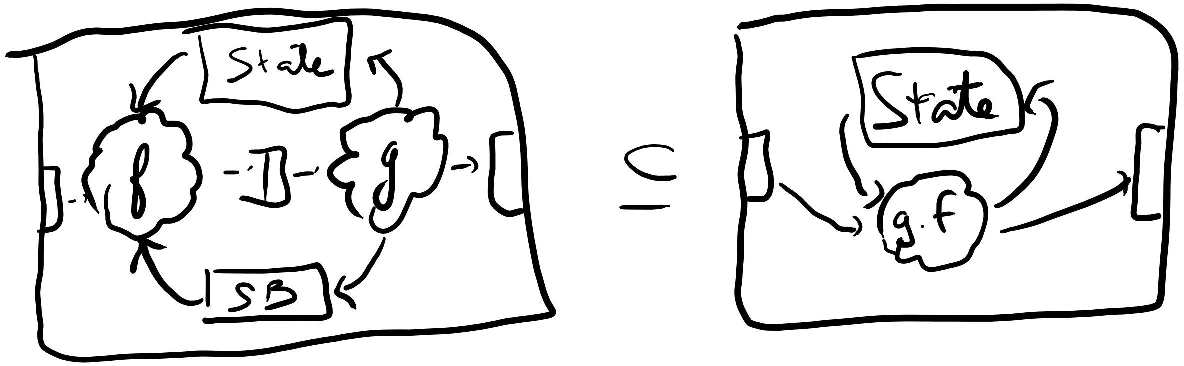
Pure pipelining



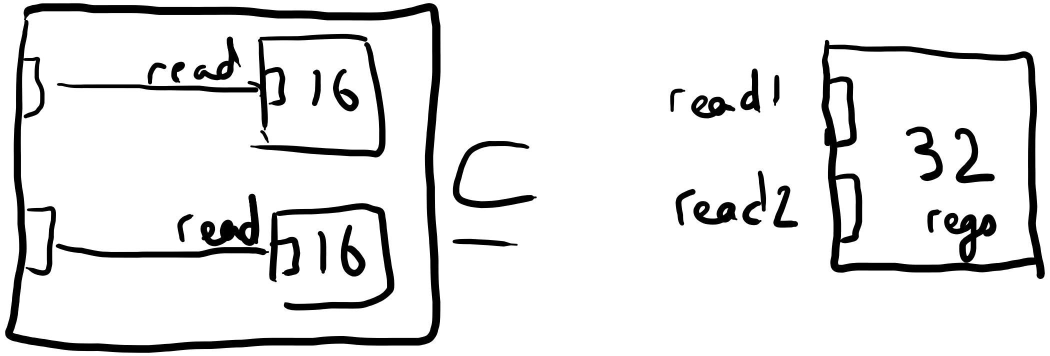
Duplicating (Parallel lanes)



Stateful pipelining



Banking



Folding, Caching, Cache Coherency, Vectorization, Pipelining with Control Flow, Reordering, Associativity, Queueing through Network (NoC)

Where is there room for performance?

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00

There is plenty of room at the top[...], Science 2020, C. Leiserson & al.

Where is there room for performance?

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01

There is plenty of room at the top[...], Science 2020, C. Leiserson & al.

Where is there room for performance?

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03

There is plenty of room at the top[…], Science 2020, C. Leiserson & al.

Where is there room for performance?

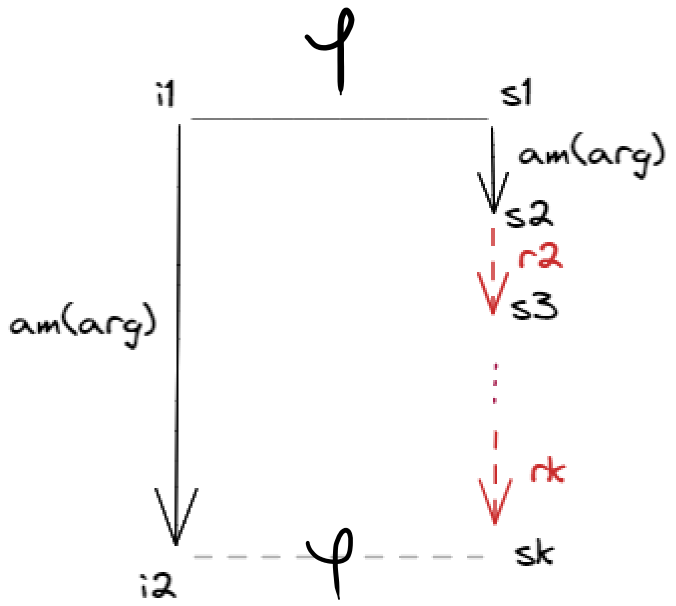
Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

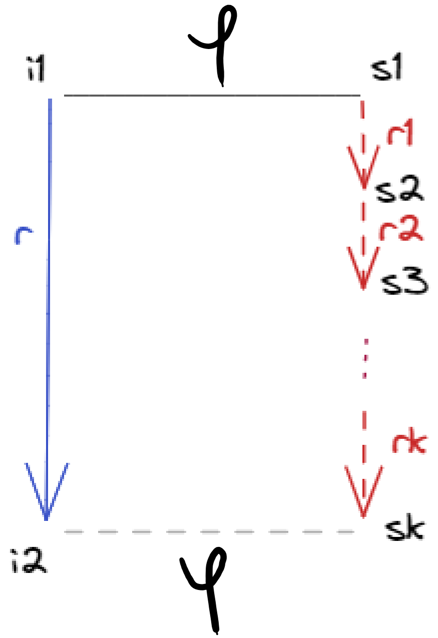
There is plenty of room at the top[...], Science 2020, C. Leiserson & al.

Weak Simulation/Refinement Map (Abadi/Lamport)

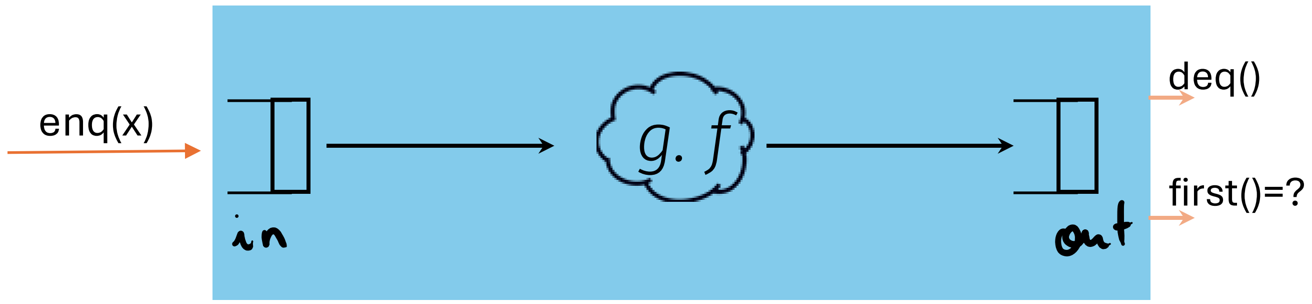
Find a mapping `(phi: stateImpl -> stateSpec -> Prop)` such that:

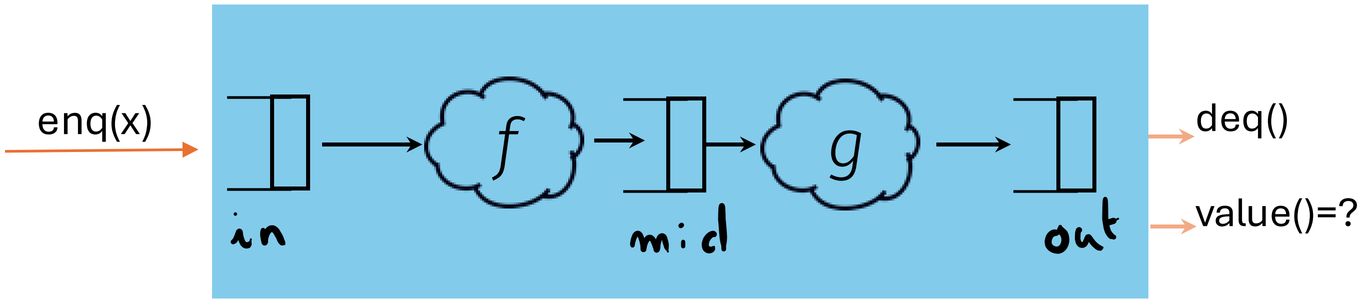
- `phi i0 s0`
- `forall i s, phi i s -> i < s`
- And phi is preserved forward:





Simulation Relations – First Examples






```
phi i s :=
  map (g • f) s.in ++ s.out =
  map (g • f) i.in ++ map g i.mid  ++ i.out
```

Shelving A Few Issues

How hard is it to write phi?

Even for small designs, too hard

How big is phi?

Even for small designs, too big

How much does phi changes when doing a little design update?

Too much

Part 2 -
Inductive Refinement Maps

Defining processor correctness – Burch&Dill '94

At a time when there was no clean notion of interface in hardware

Burch & Dill *defined* correctness as "commuting with Flushing":

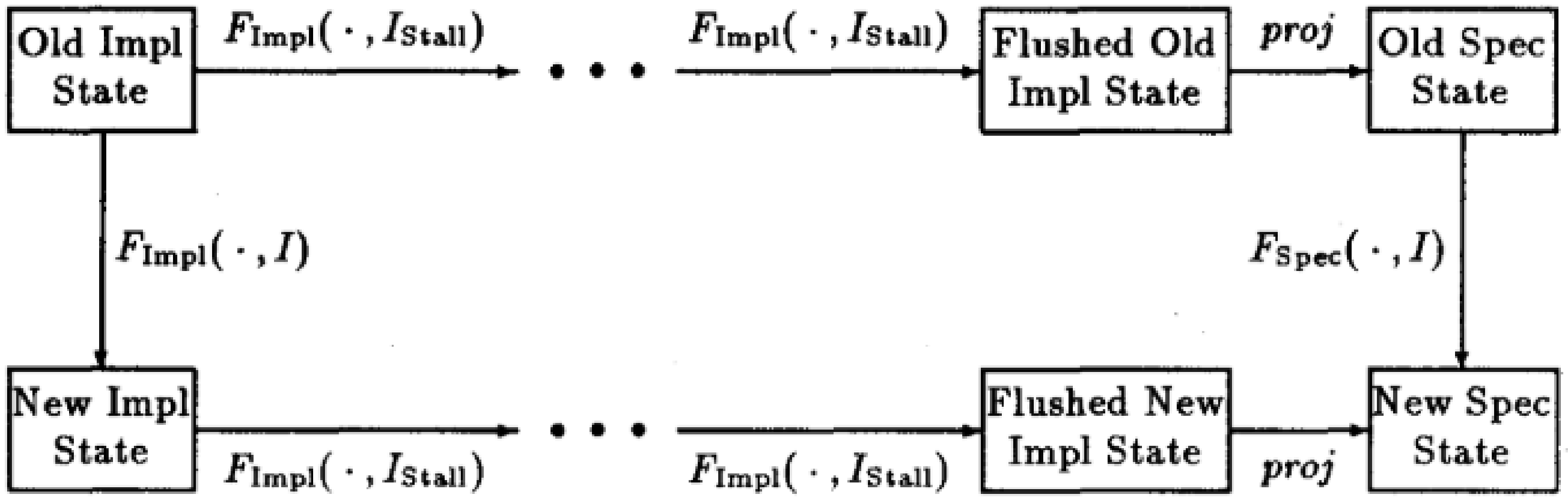
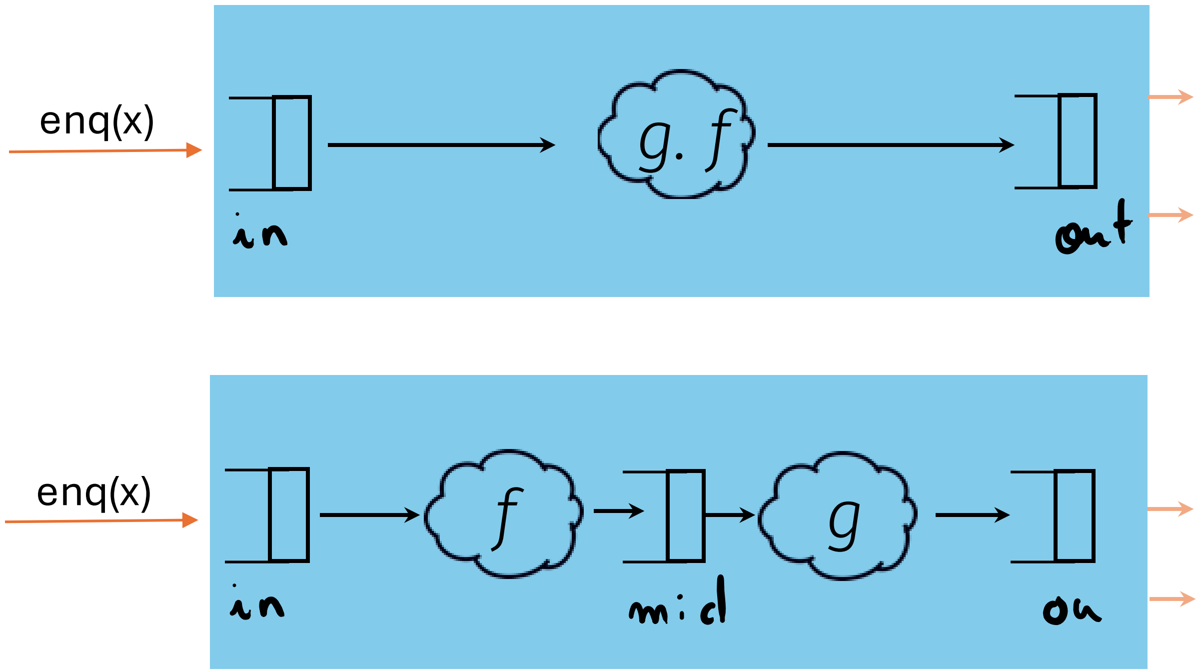


Fig. 1. Commutative diagram for showing our correctness criteria.

What we like/don't like

- Yay:
  - Property of "implementation flushes and specification agree post flushing" is a very rich relation
  - Architecturally meaningful, I can think about it!
  - Criteria amenable to automatic verification when pipeline has bounded depth
- Abstain:
  - We can (almost) prove that the ultimately lazy machine is Burch&Dill correct
- Nay:
  - Requires to write the flushing steps as shadow logic (the machine does not really flush), error prone, what do we verify if error in that code?
  - Ambiguous – more than one way to "flush" -> are those notion of correctness equivalent

Flushing for our (f•g) pipeline: an inductive simulation relation!




```
Inductive phi : ImplState -> SpecState -> Prop :=
 | Flushed : forall l, phi ([],[],l) ([],l)

 | one_more_f : forall i i' s,
           (i ~( do_f )~> i') ->
           phi i' s -> phi i s

 | one_more_g : forall i i' s,
           (i ~( do_g )~> i') ->
           phi i' s -> phi i s.
```

Phi is preserved forward, by induction on phi

Only one case interesting:

$$\begin{tikzcd}
([\,],[\,],\ell) \arrow[r, dash, "\varphi"] \arrow[d, "enq(x)"'] & ([\,],\ell) \arrow[d, dotted, "enq(x)"] \\
([x],[\,],\ell) & {}
\end{tikzcd}$$

Phi is preserved forward, by induction on phi

Base case:

Only one case interesting:

$$\begin{tikzcd}
([\,],[\,],\ell) \arrow[r, dash, "\varphi"] \arrow[dd, "enq(x)"'] & ([\,],\ell) \arrow[d, "enq(x)"] \\
 & {} \arrow[d, dotted, "do_fg"] \\
([x],[\,],\ell) \arrow[r, dash, "\color{red}\varphi"] & ([\,], [g(f(x))] +\!\!+\, \ell)
\end{tikzcd}$$

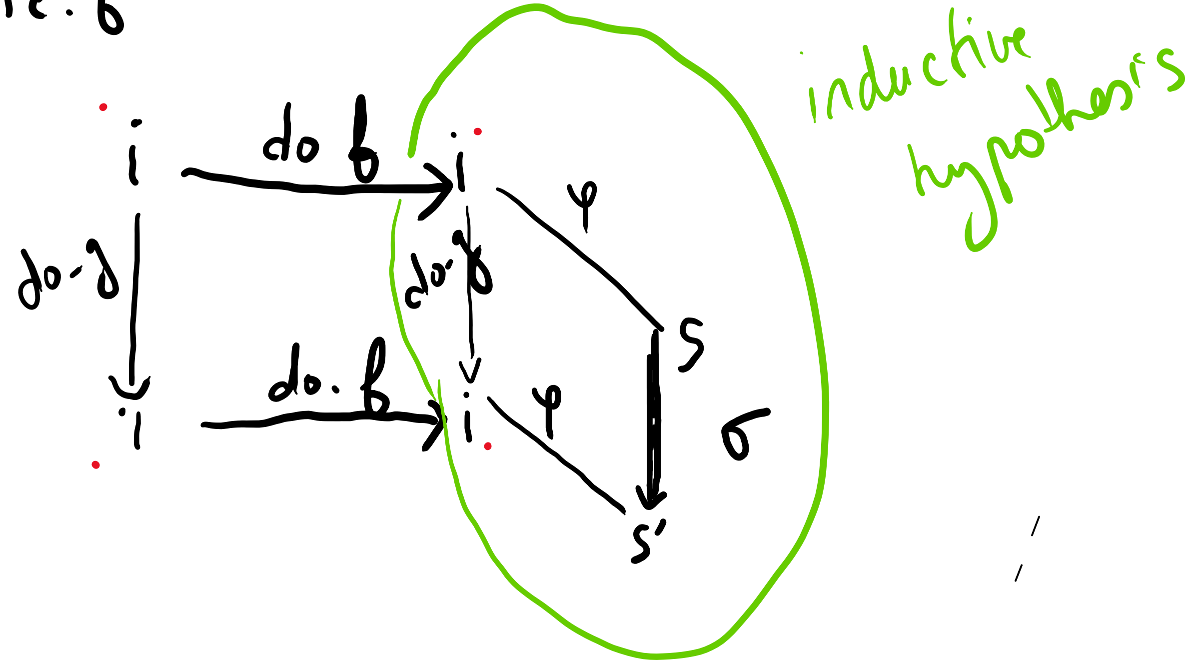
one_more_f

one_more_g, then flushed!

Phi is "inductive" by induction on phi

One.more.f

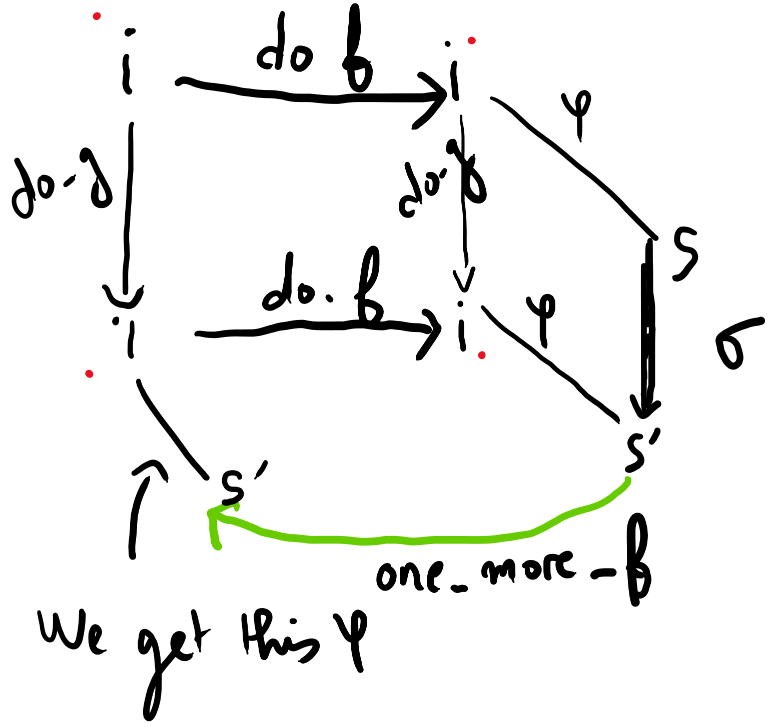
Inductive cases



Phi is "inductive" by induction on phi

one_more_f

Inductive cases



`phi i s -> i < s`

- By induction on phi:
  - Base case: `phi ([],[],l) ([],l)` , masquerading all good

  - Inductive case (do_f easy, do_g not completely immediate, left as an exercise)

Unshelving The Issues

How hard is it to write phi?

Did you consider inductive flushing?

How big is phi?

Linear in the size (number of transitions < 10 when doing hierarchical proofs) of the system

How much does phi changes when doing a little design update?

Very little **(the proof might change though)**

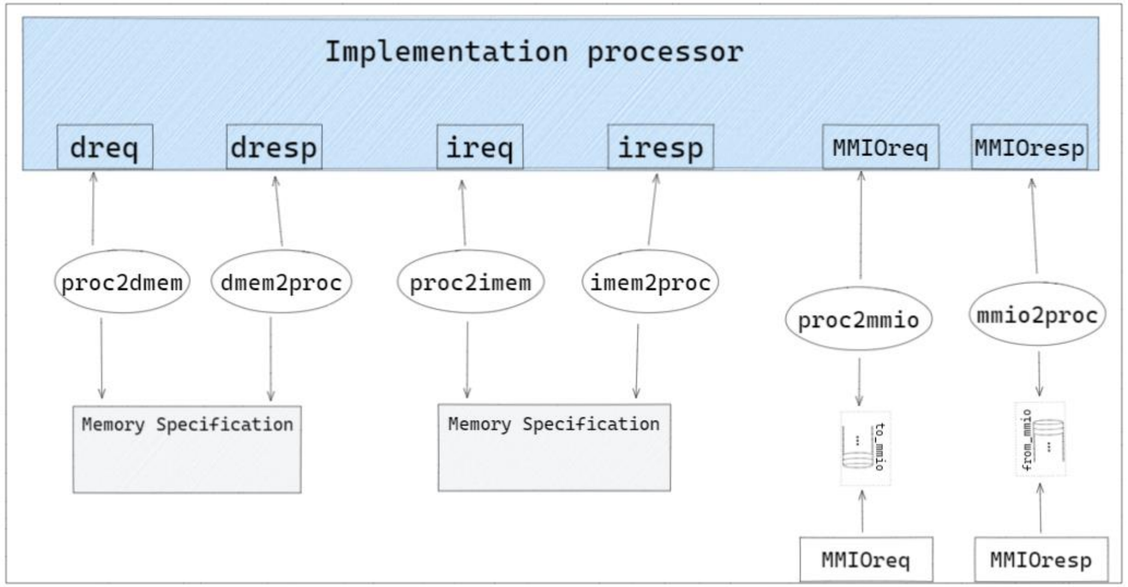
What' s the scam?

N^2 cases in the inductive case

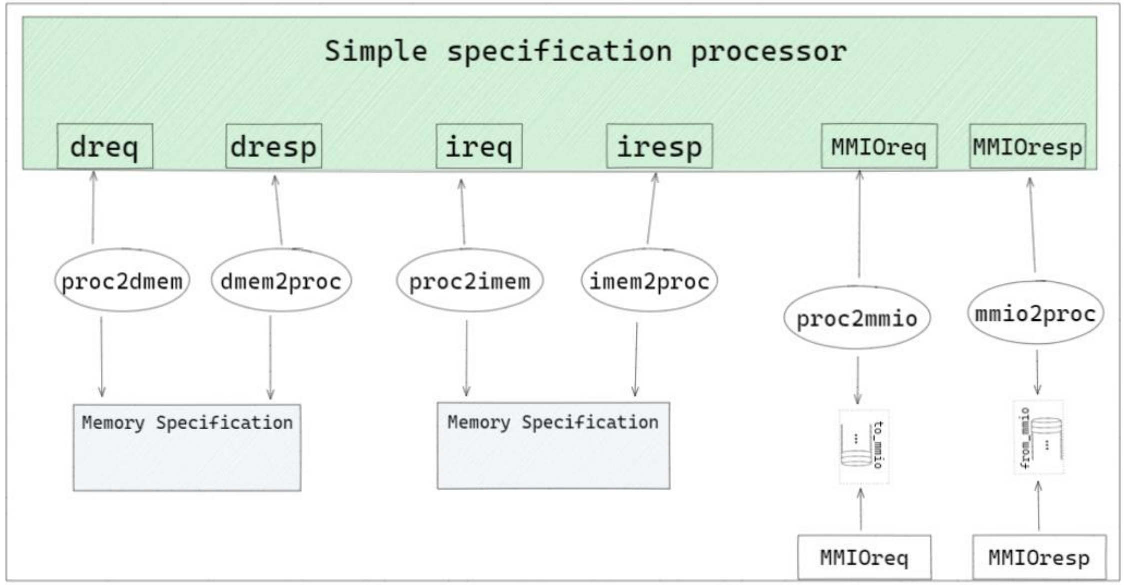
Maybe 3 – Actual Problems with Refinements

Coming with with specification of submodules is hard, but worth it!

Two Systems: Implementation and a Specification



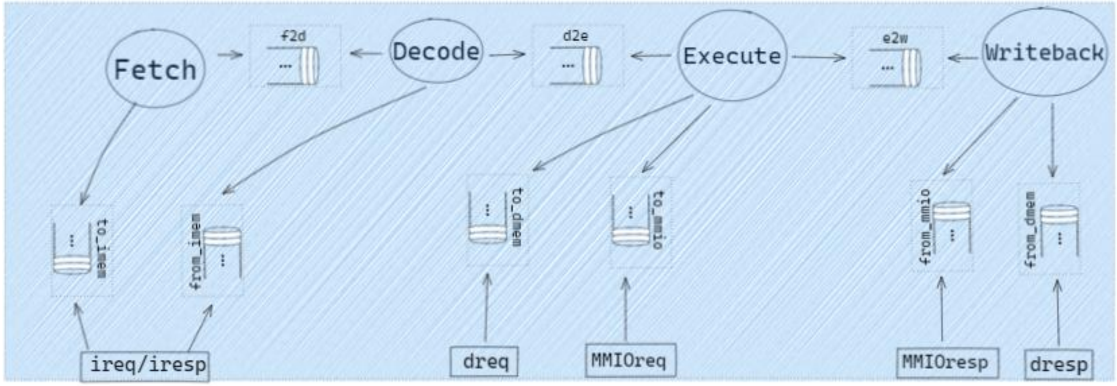
$\sqsubseteq$



Instruction and data memory are separate for now
Memories have request/response interfaces

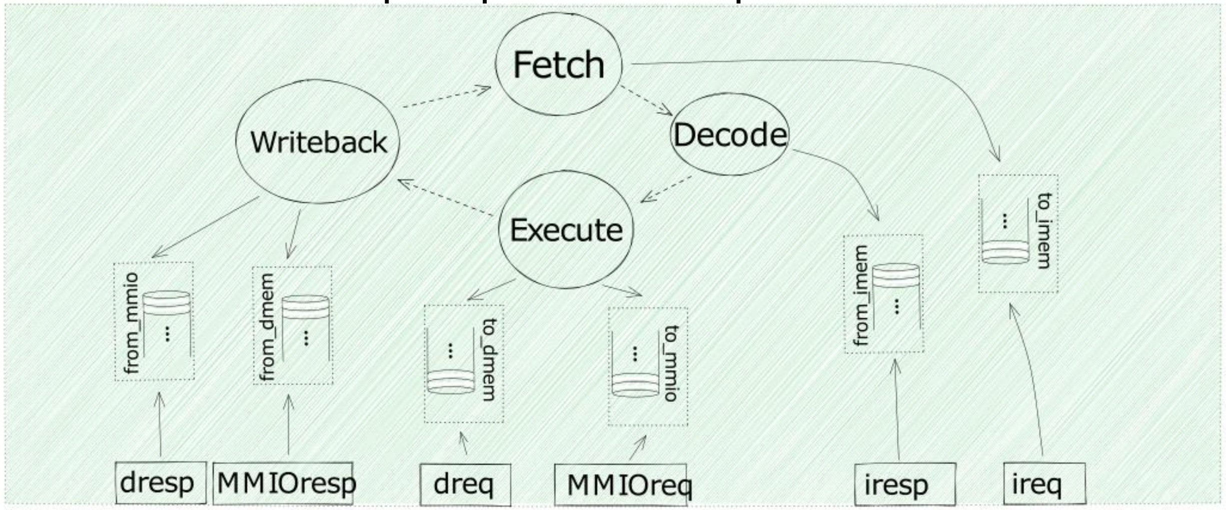
Invalid processor specification!

Pipelined processor



$\not\sqsubseteq$

Simple specification processor



Indeed, the implementation can be queries for two instruction requests back-to-back

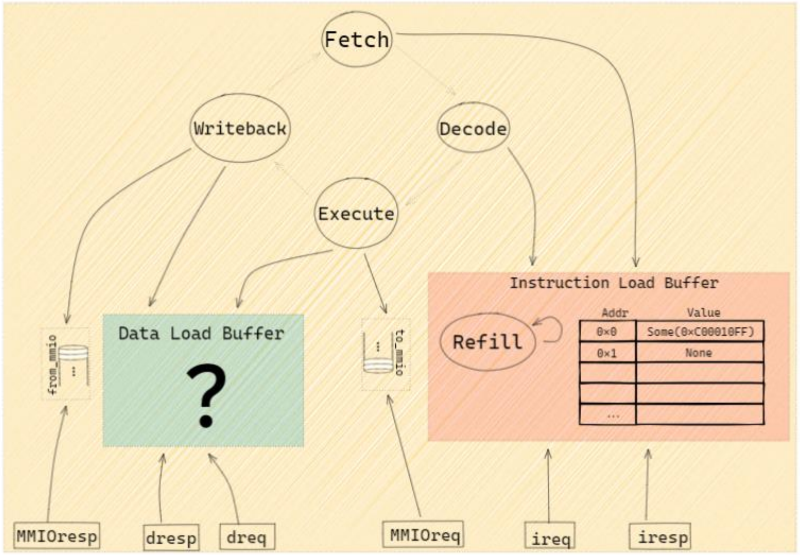
[ireq()->_; ireq()->_]

We do not have a valid specification just for the processor.

<u>We only have a specification for the full system</u>, which happened to be made of a "processor" and a "memory".

Generalizing the specification

Generalized specification processor



4 *sequential* steps:

- Fetch, Decode, Execute, Writeback
- Always works on exactly one instruction
- No speculation/prediction

Two *non-deterministic load machine*:

- Processor does not directly emit loads to memories
- Instead processor queries the load buffers
- Load buffers are refilled nondeterministically

Architectural intuition: *Some load speculation techniques can be wild, let's be conservative and just say that loads can be emitted at any time, for any address.*

Why is the generalization valid?

Architecturally it is obvious:

Loads don't matter

Is that intuition formalizable?

What prevents us to make a mistake?

We just changed our specification with no discussion?

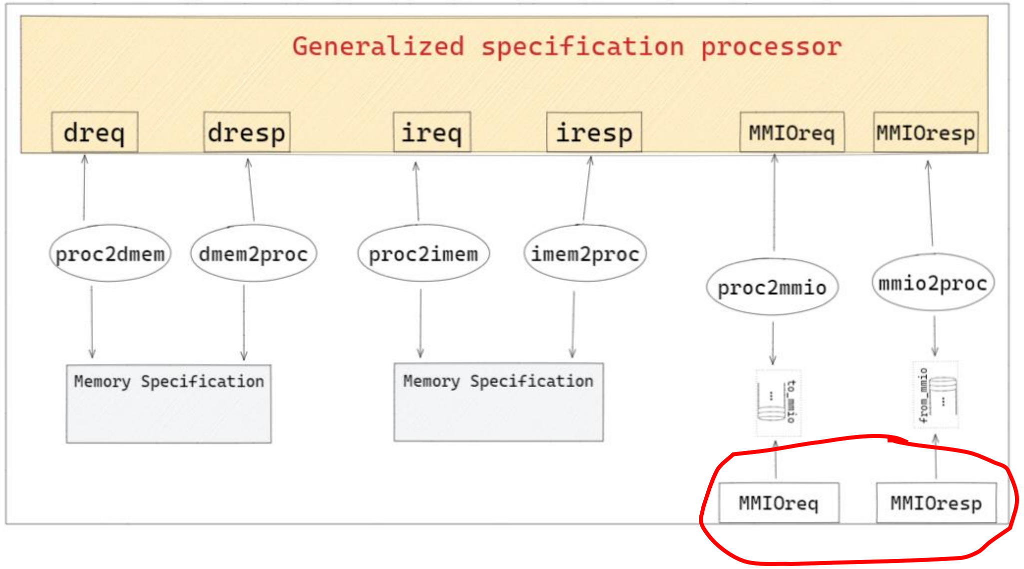
Generalized specification that emits random stores? Clearly wrong

Why is the generalization valid?

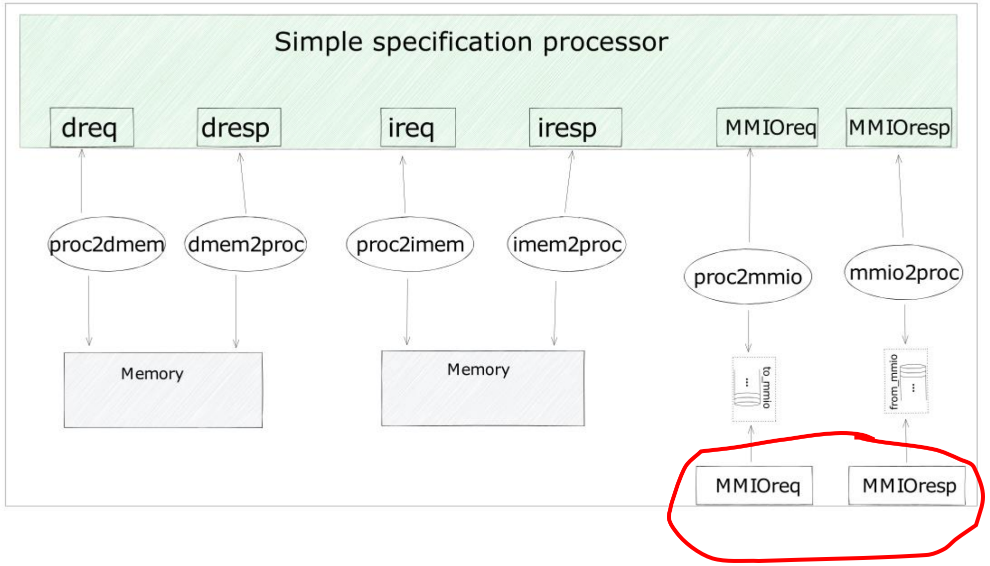
Architectural intuition:

“Loads don’t matter”

“Loads don’t matter, from the perspective of the MMIO trace of the full-system”

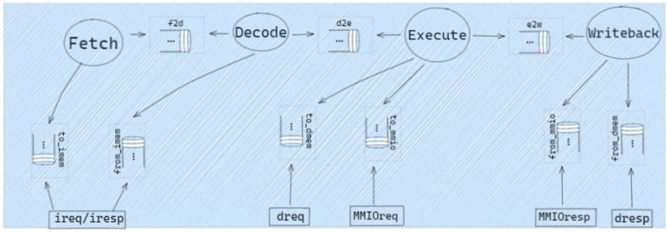


$\sqsubseteq$

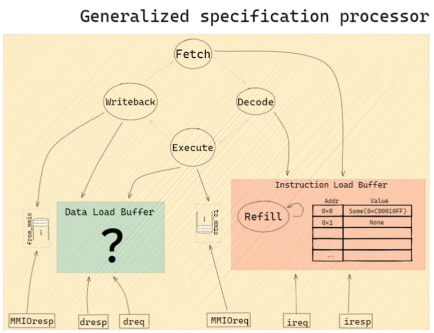


Modular proof

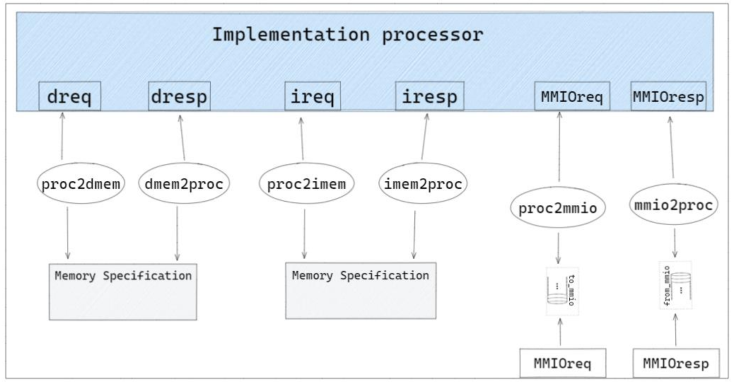
Has a chance to be true! (And it is actually true)
Note this theorem does not even mention the memory



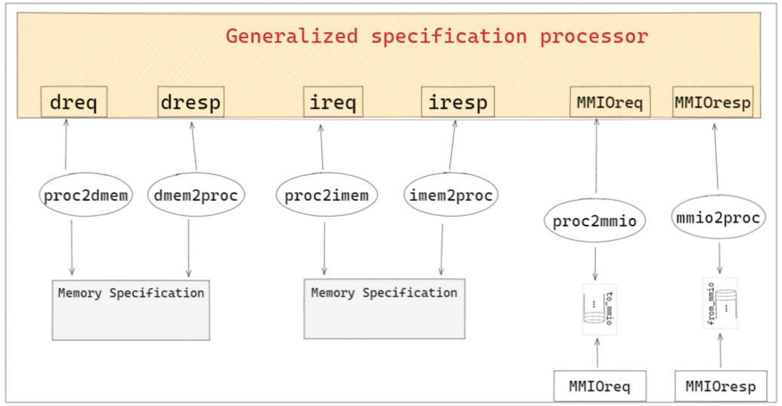
$\sqsubseteq$



Applying the refinement theorem



$\sqsubseteq$



$\sqsubseteq$