NICTA

Lyrebird

David Cock

davec@cse.unsw.edu.au

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

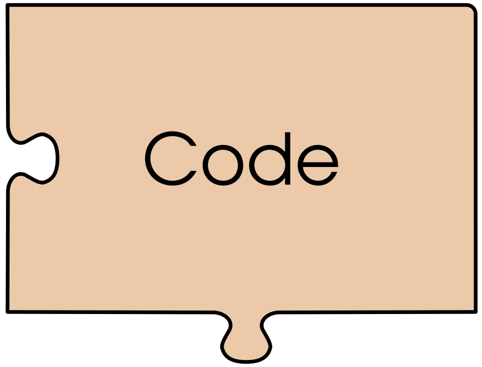
ANU THE AUSTRALIAN NATIONAL UNIVERSITY
UNSW THE UNIVERSITY OF NEW SOUTH WALES
NSW GOVERNMENT Industry & Investment
ACT GOVERNMENT
Victoria The Place To Be
THE UNIVERSITY OF MELBOURNE
THE UNIVERSITY OF SYDNEY
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
THE UNIVERSITY OF QUEENSLAND AUSTRALIA

What is the Motivation?

Program proof is important, but there's more to do.

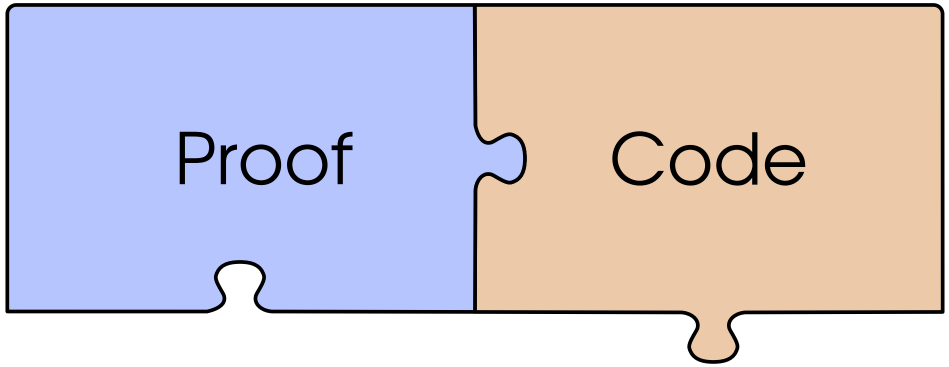
What is the Motivation?

Program proof is important, but there's more to do.



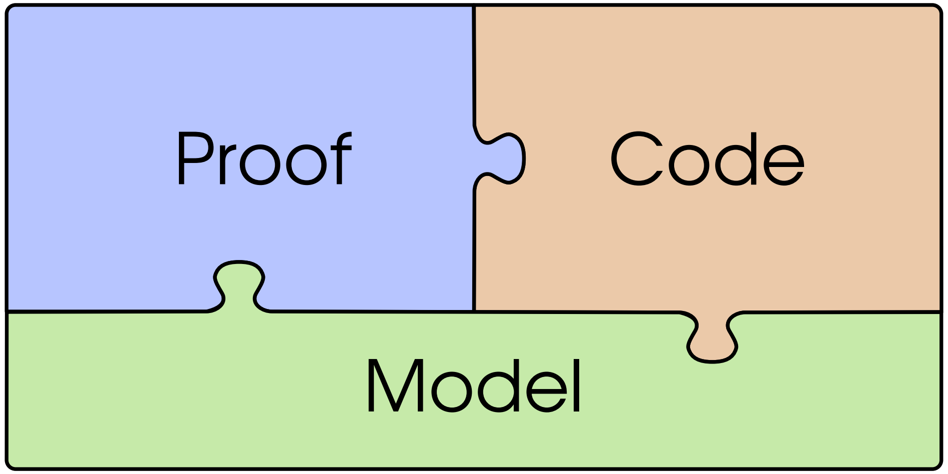
What is the Motivation?

Program proof is important, but there's more to do.



What is the Motivation?

Program proof is important, but there's more to do.

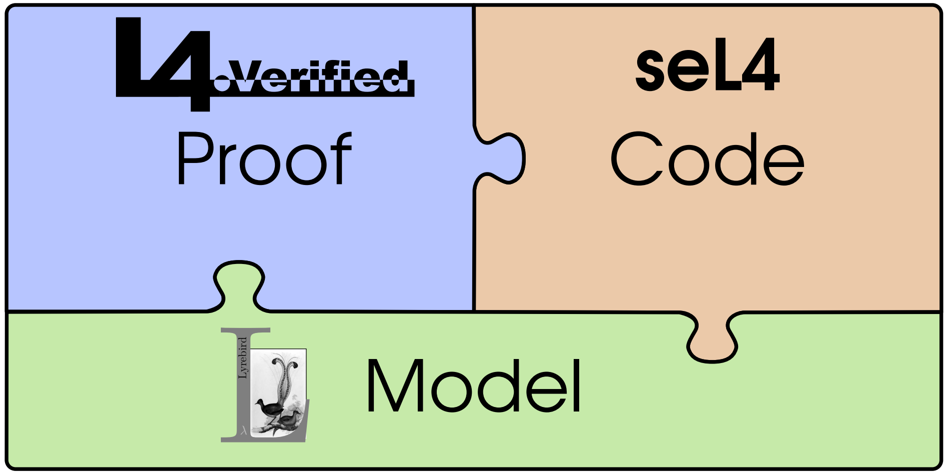


Any statement "P is True" is incomplete:
It must be read as ", under Q - my model of the world".

Goal: *Development outcomes: program, proof and model.*

What is the Motivation?

Program proof is important, but there's more to do.

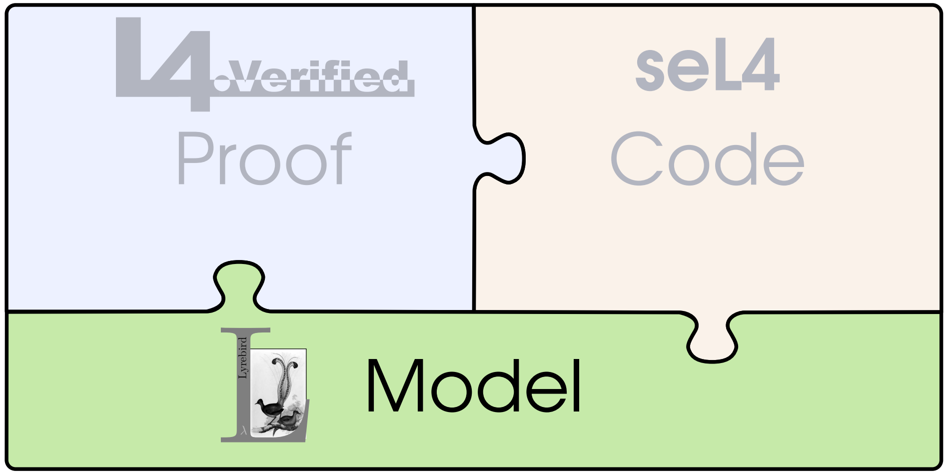


Any statement "P is True" is incomplete:
It must be read as ", under Q - my model of the world".

Goal: *Development outcomes: program, proof and model.*

What is the Motivation?

Program proof is important, but there's more to do.

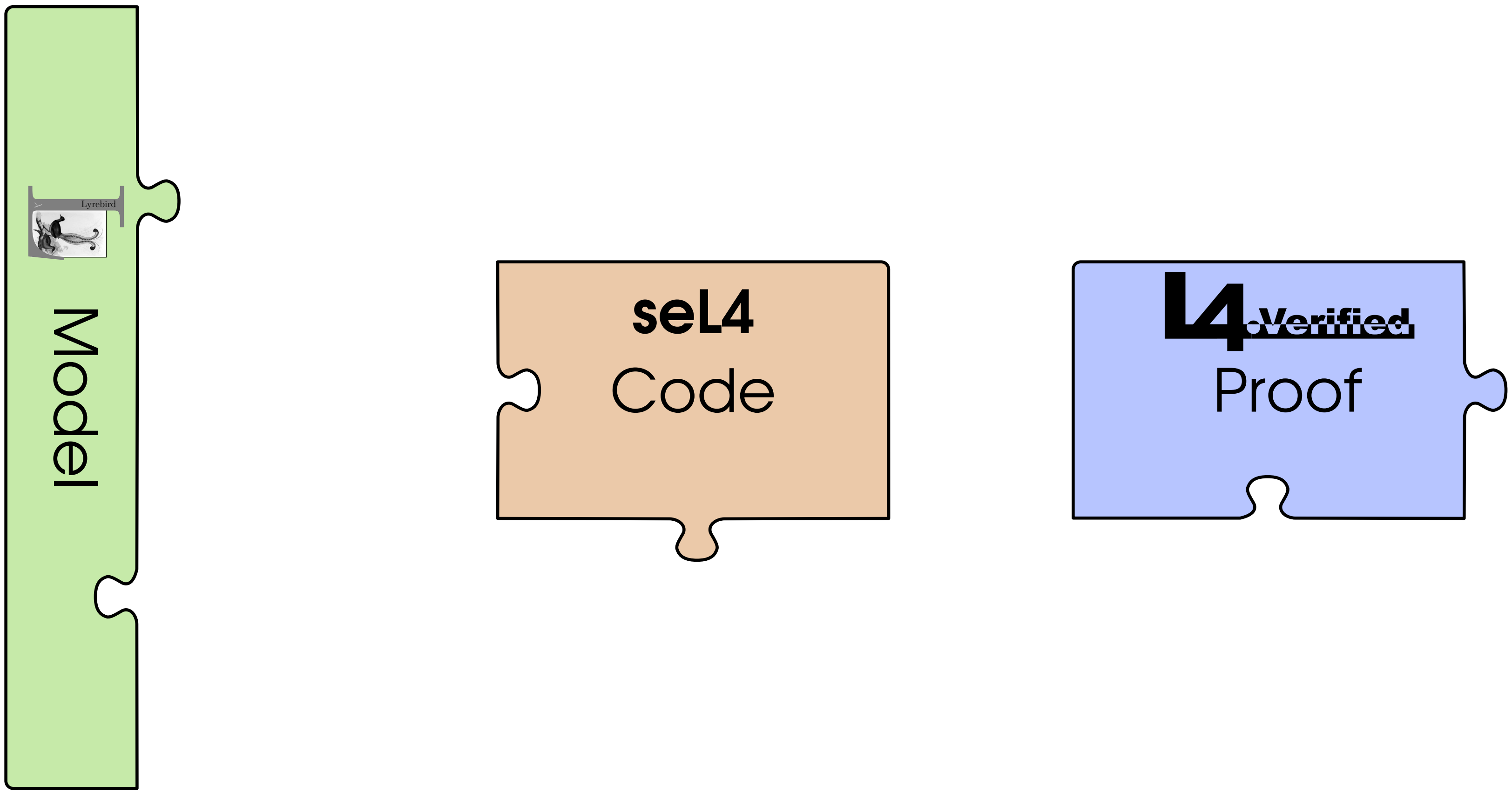


Any statement "P is True" is incomplete:
It must be read as ", under Q - my model of the world".

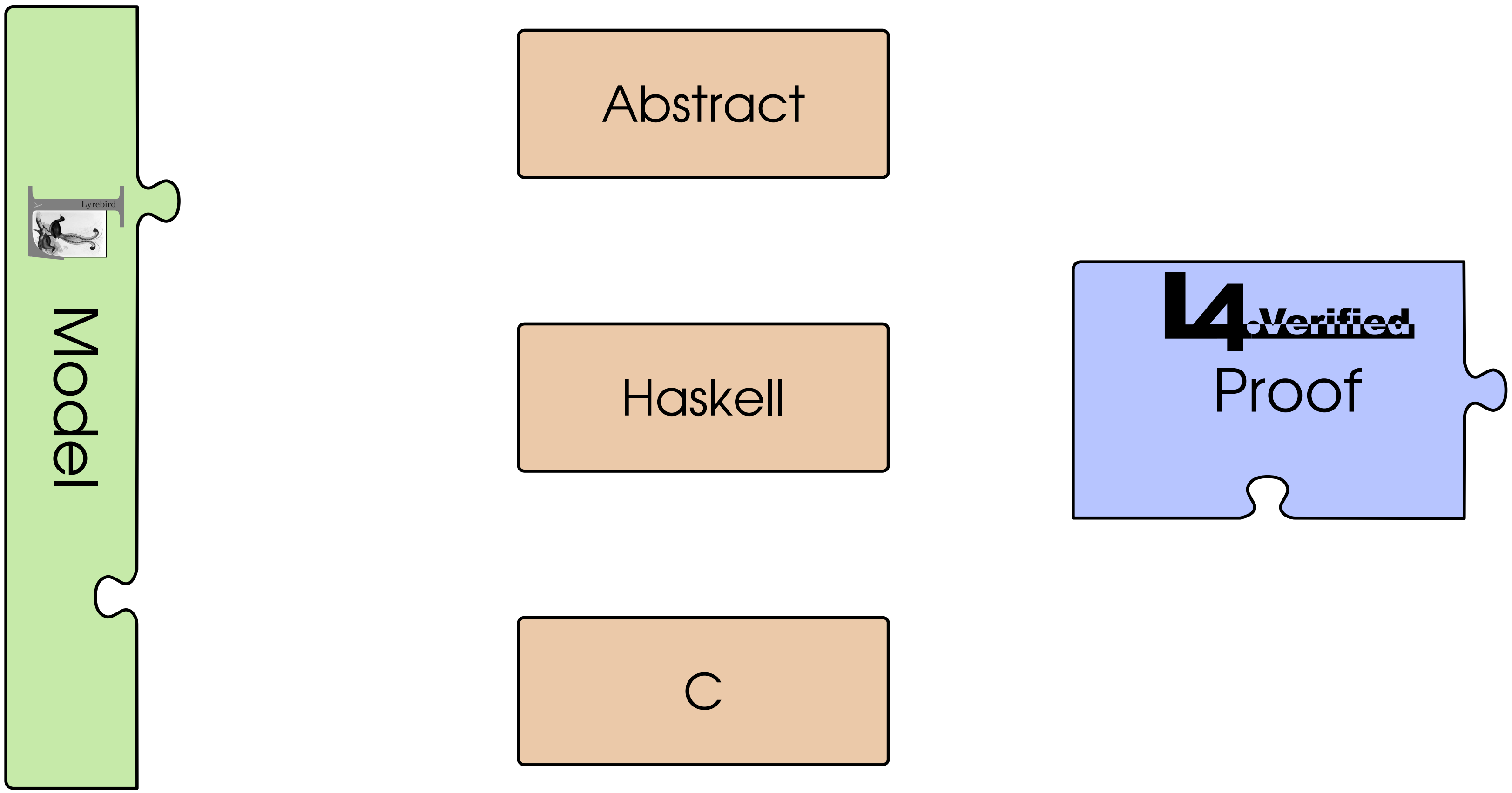
Goal: *Development outcomes: program, proof and model.*

Our approach is a language framework: *Lyrebird*.

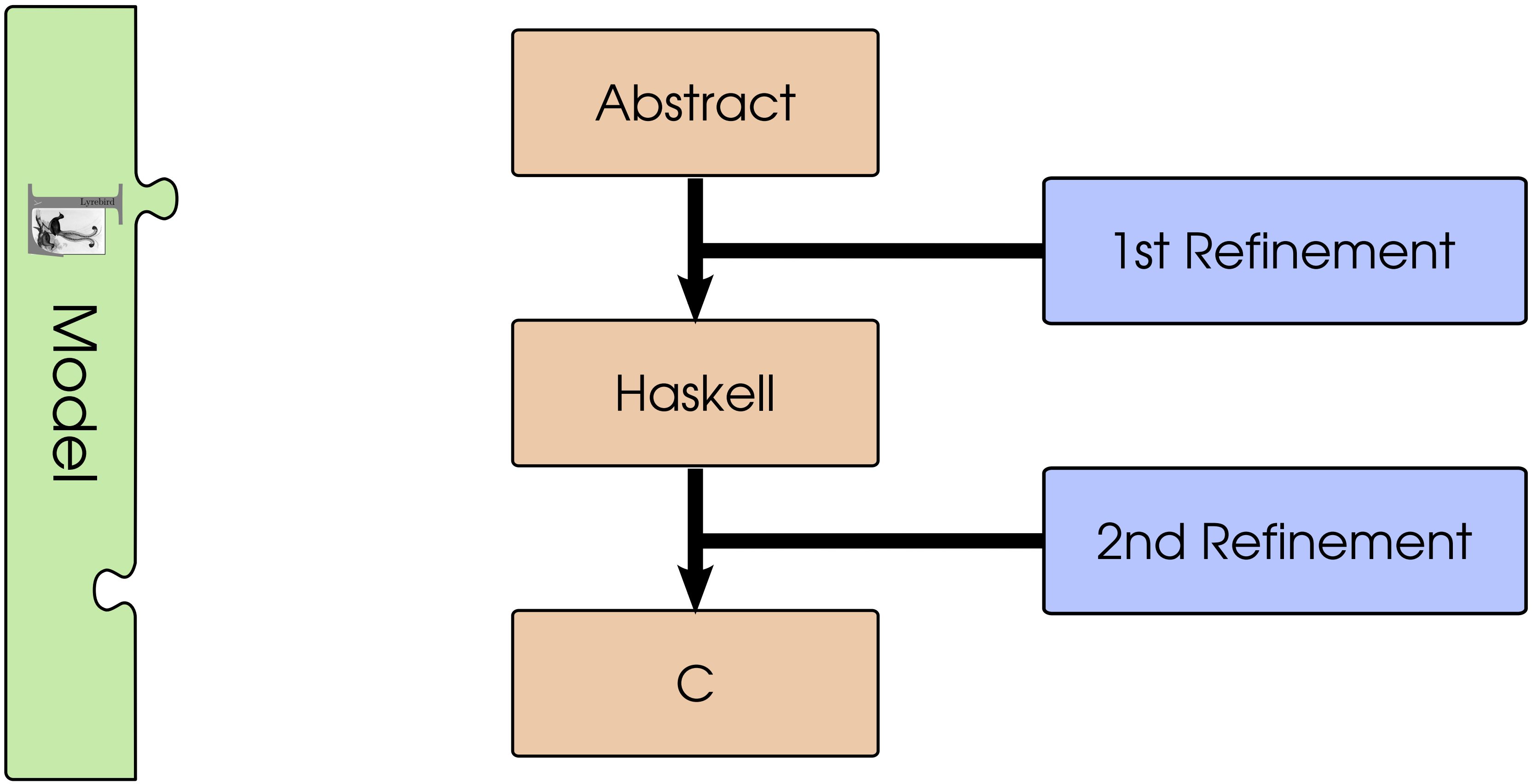
What is the Motivation?



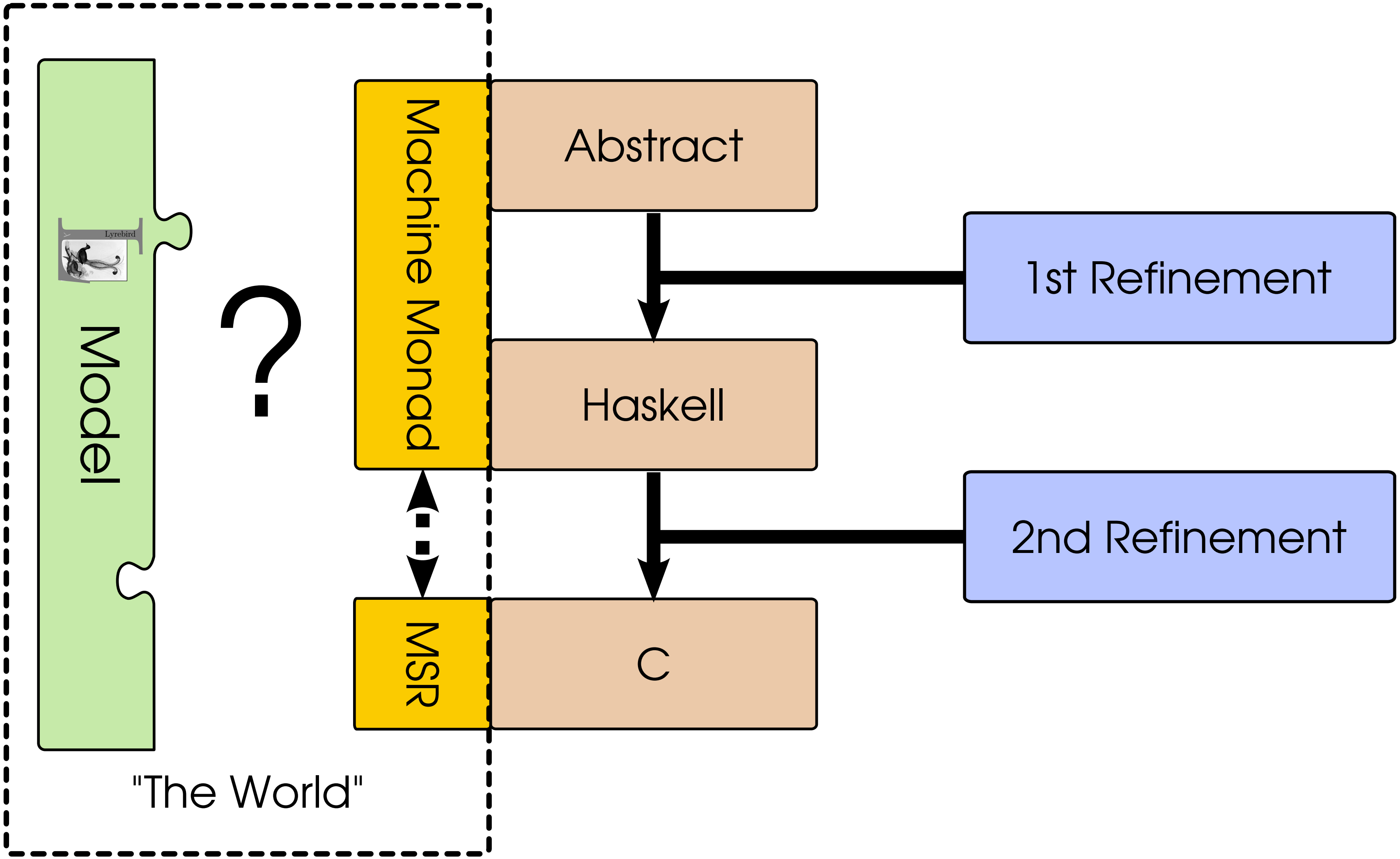
What is the Motivation?



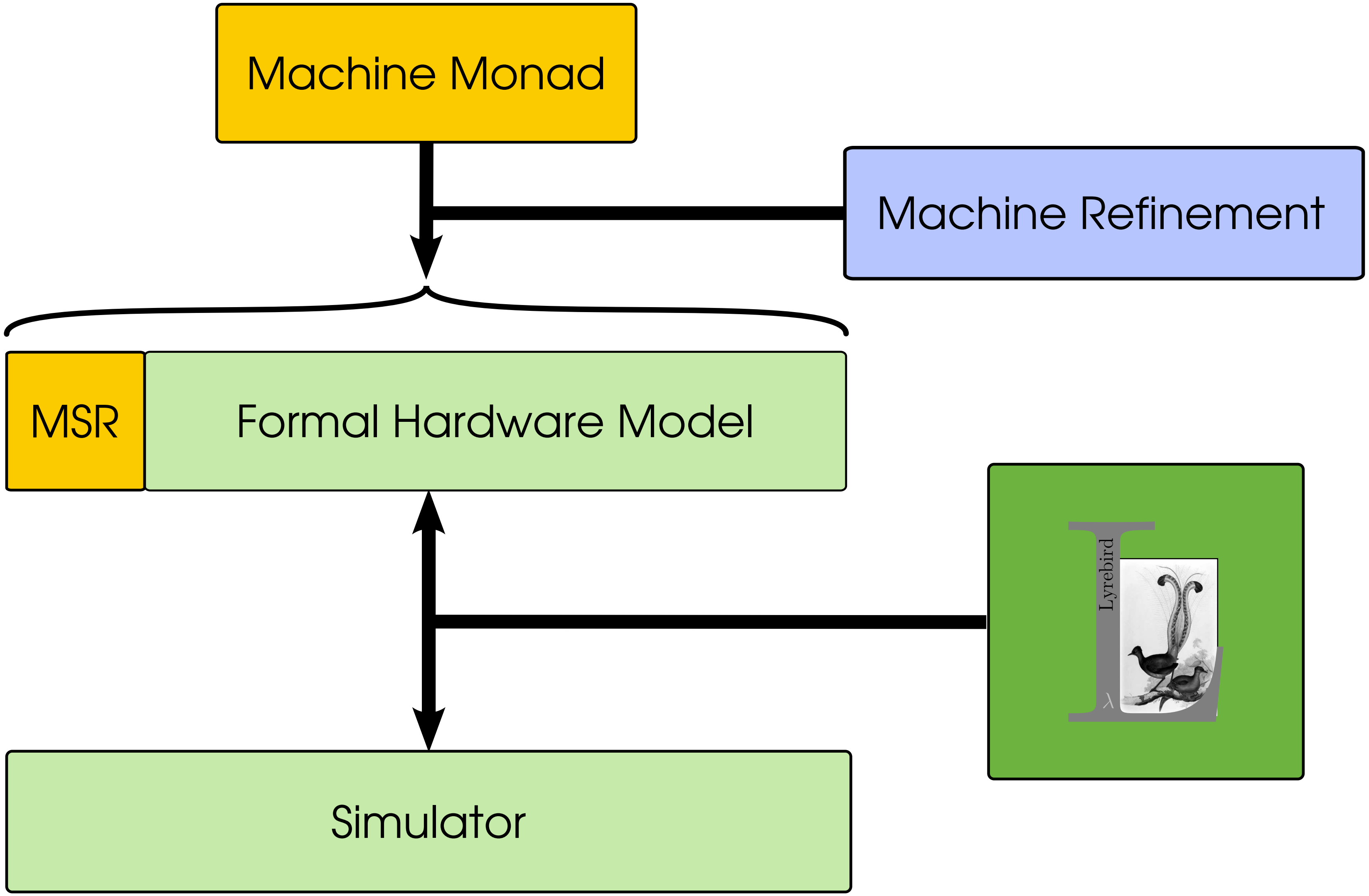
What is the Motivation?



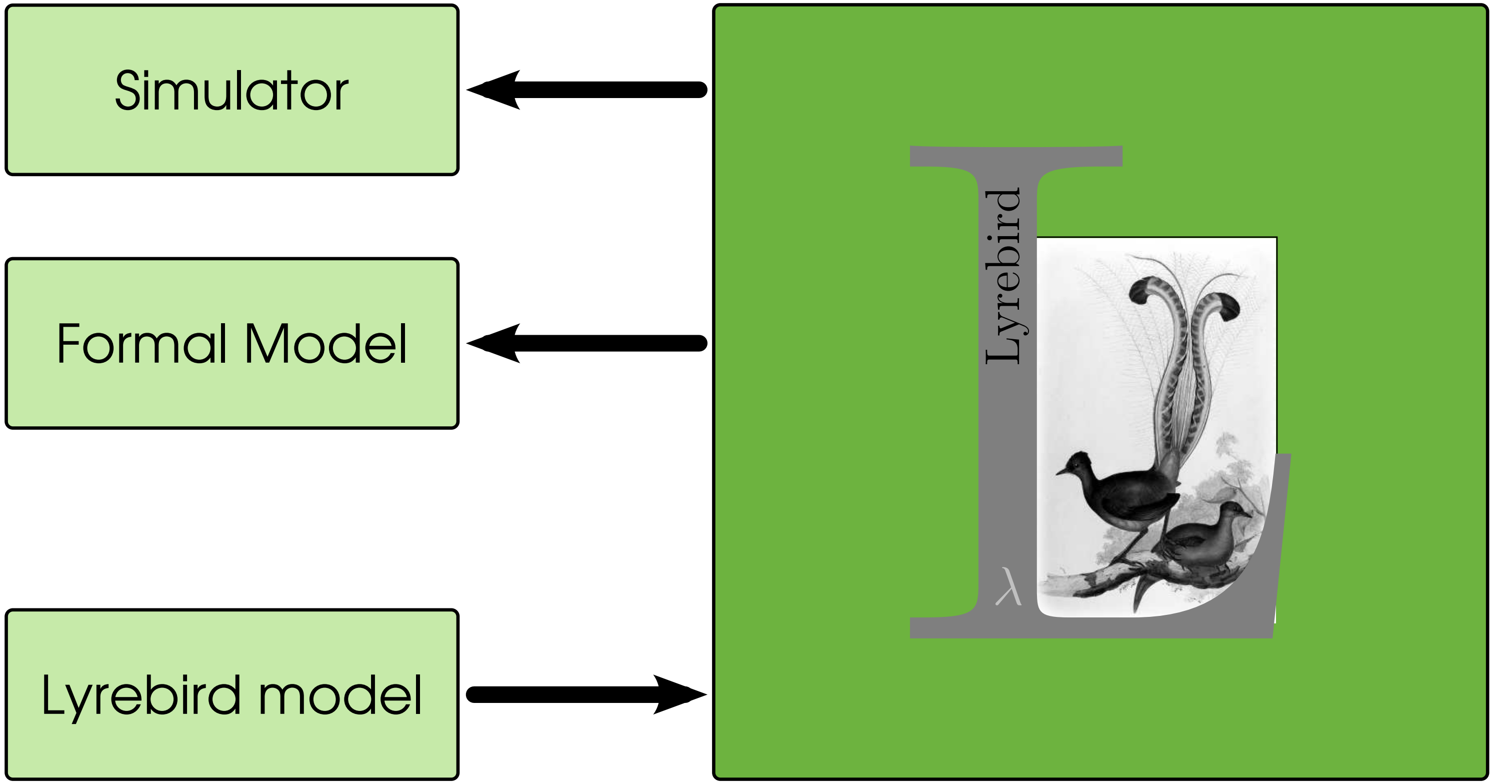
What is the Motivation?



What is the Motivation?



What is the Motivation?

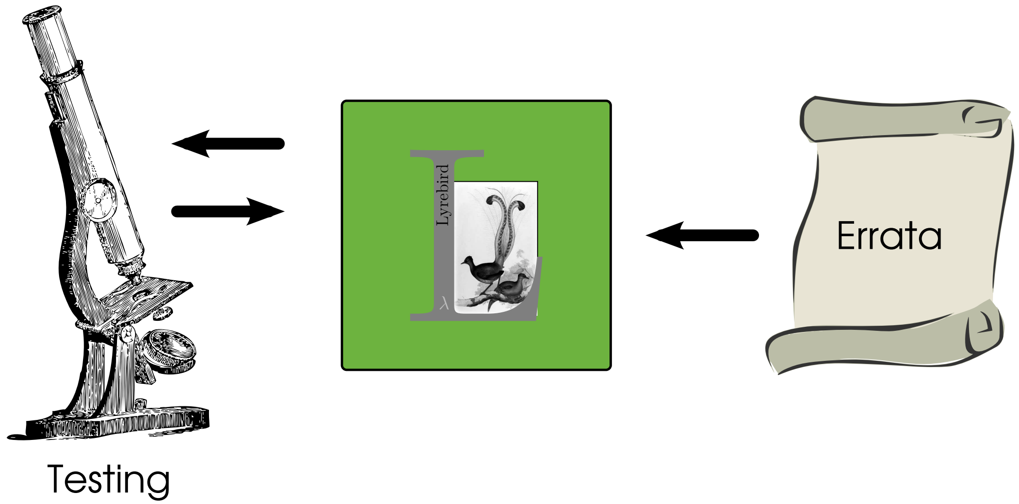


Lyrebird is a framework built around a modelling language.

Tools are included to generate simulators and formal models.

Observations

NICTA



The Model Should be Progressively Refined:

Even the manufacturer doesn't have a complete model, they publish errata when they find mistakes.

Goal: *Updating the model should be easy.*

Observations

To a program, the world is the machine.

Building machine models is hard, often boring work.

It's easy to get started, and cover the part that's well behaved.

Handling the rest, and *getting it right* takes a lot longer.

It's also mind-numbingly, soul-destroyingly dull.

So only model those parts that we actually need.

Example

What does this code do? What ends up in `r1`?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]					
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

Example

What does this code do? What ends up in `r1`?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

Example

What does this code do? What ends up in `r1`?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]	42	100	108	42	42
1008	e2811001	add r1, r1, #1					

Example

What does this code do? What ends up in `r1`?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	ldr r1, [r2]	42	100	108	42	...
1004	e5832000	str r1, [r3]	42	100	108	42	42
1008	e2811001	add r1, r1, #1	**43**	100	108	42	42

Example

What does this code do? What ends up in `r1`?

address	data	instruction	r1	r2	r3	@100	@108
...	...	...	...	100	108	42	...
1000	e5921000	`ldr r1, [r2]`	42	100	108	42	...
1004	e5832000	`str r1, [r3]`	42	100	108	42	42
1008	e2811001	`add r1, r1, #1`	**43**	100	108	42	42

Most code is like the above, and it's easy to understand;
The challenge here is how to express that formally.

Goal: *Easy things should be straightforward.*

Example

90% is not too bad and moreover it's been done.

We should focus on the 10%, the hard parts.

Example

90% is not too bad and moreover it's been done.

We should focus on the 10%, the hard parts.

So what *is* a hard part?

Example

90% is not too bad and moreover it's been done.

We should focus on the 10%, the hard parts.

So what *is* a hard part?

Let's have another look at that example. . .

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]					
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]					
1008	e2811001	add r1, r1, #1					

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1					

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000

Wait a minute, what was that address? Didn't we just overwrite this instruction?

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000
Wait a minute, what was that address? Didn't we just overwrite this instruction?							
1008	e5921000	ldr r1, [r2]	**e5921000**	1000	1008	e5921000	e5921000

Example

Another look at the example:

What value ends up in `r1` now?

			r1	r2	r3	@1000	@1008
...	...	...	...	**1000**	**1008**	e5921000	...
1000	e5921000	ldr r1, [r2]	e5921000	1000	1008	e5921000	...
1004	e5832000	str r1, [r3]	e5921000	1000	1008	e5921000	e5921000
1008	e2811001	add r1, r1, #1	**e5921001**	1000	1008	e5921000	e5921000
Wait a minute, what was that address? Didn't we just overwrite this instruction?							
1008	e5921000	ldr r1, [r2]	**e5921000**	1000	1008	e5921000	e5921000

Which of these is the right answer?

Example

It depends . . . on the CPU, the cache, and the state.

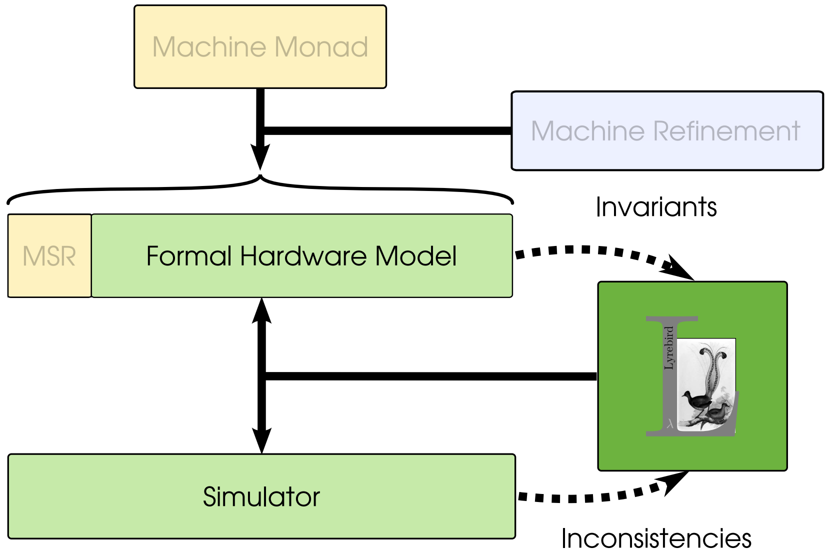
This isn't hypothetical;
We need to write code to memory and then run it
. . . and we need to make sure we do it right.

In a formal model, this is a corner case and it's abstracted.

Sometimes, however, you've got to get your hands dirty.

Goal: *Hard things should be possible.*

How to Build Models

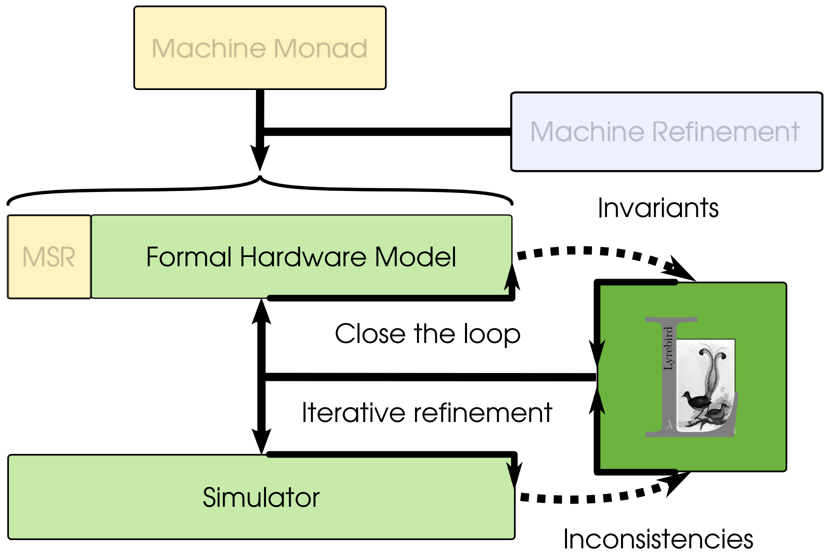


Verification uncovers what the machine *should* do.
These models are too abstract.

Programming uncovers what the machine *does*.
These models are too informal.

We must combine this knowledge rigorously.

How to Build Models



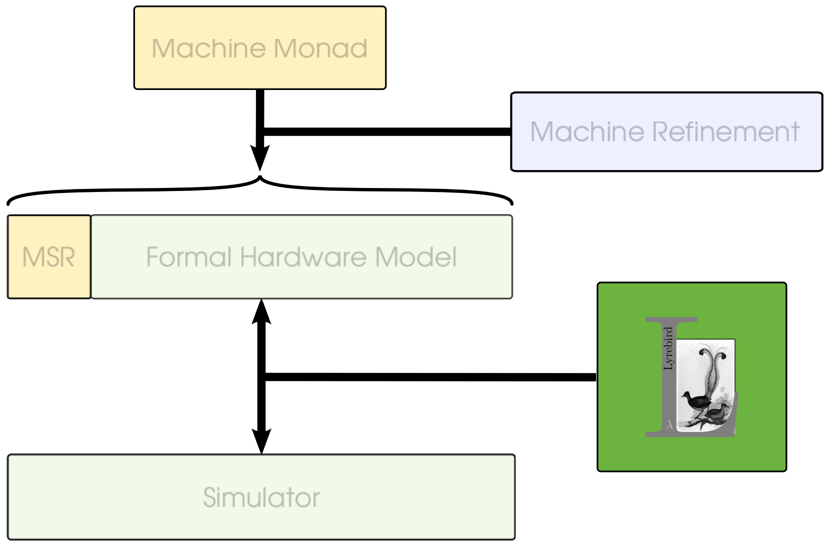
Work Iteratively:

Start with a simple model and only add details as required.

When verification uncovers a requirement, update the model.

When programming discovers a behaviour, update the model.

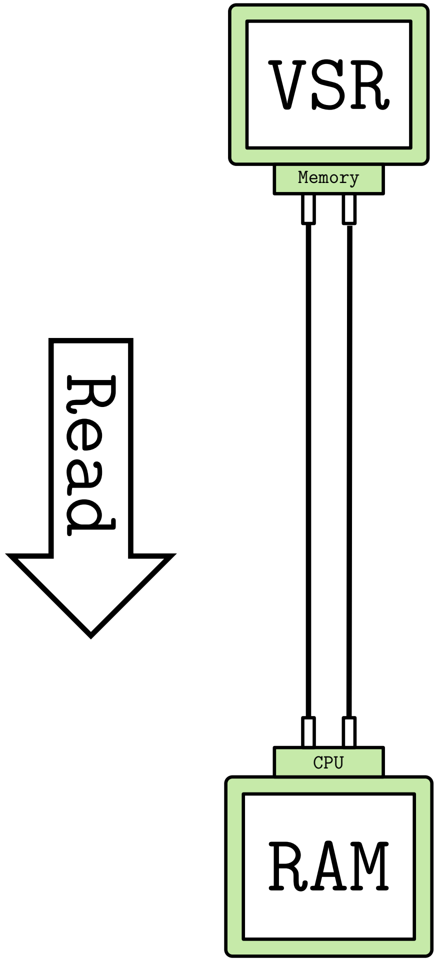
How to Build Models



This workflow requires a common language.

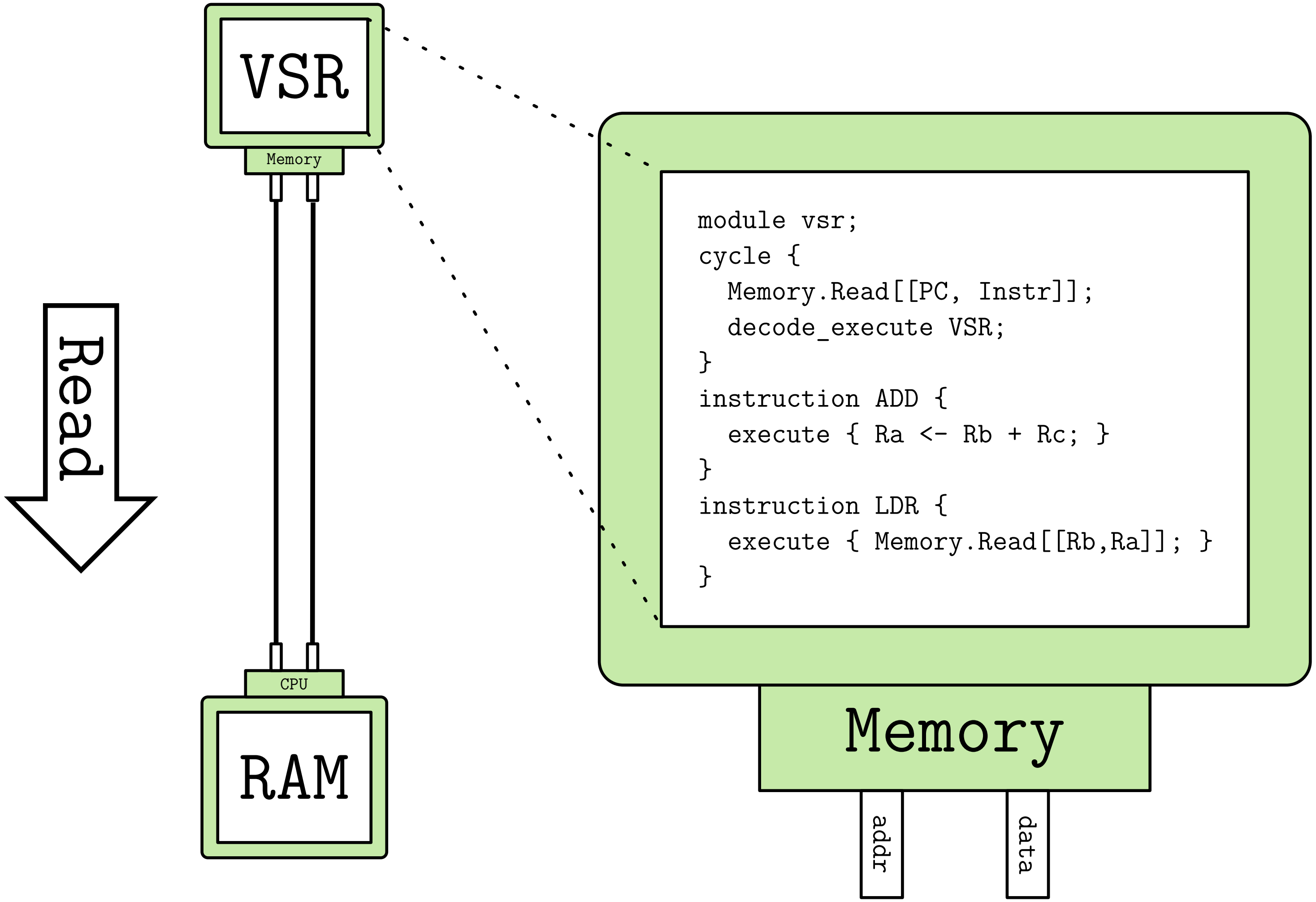
Our solution is *Lyrebird*

Lyrebird



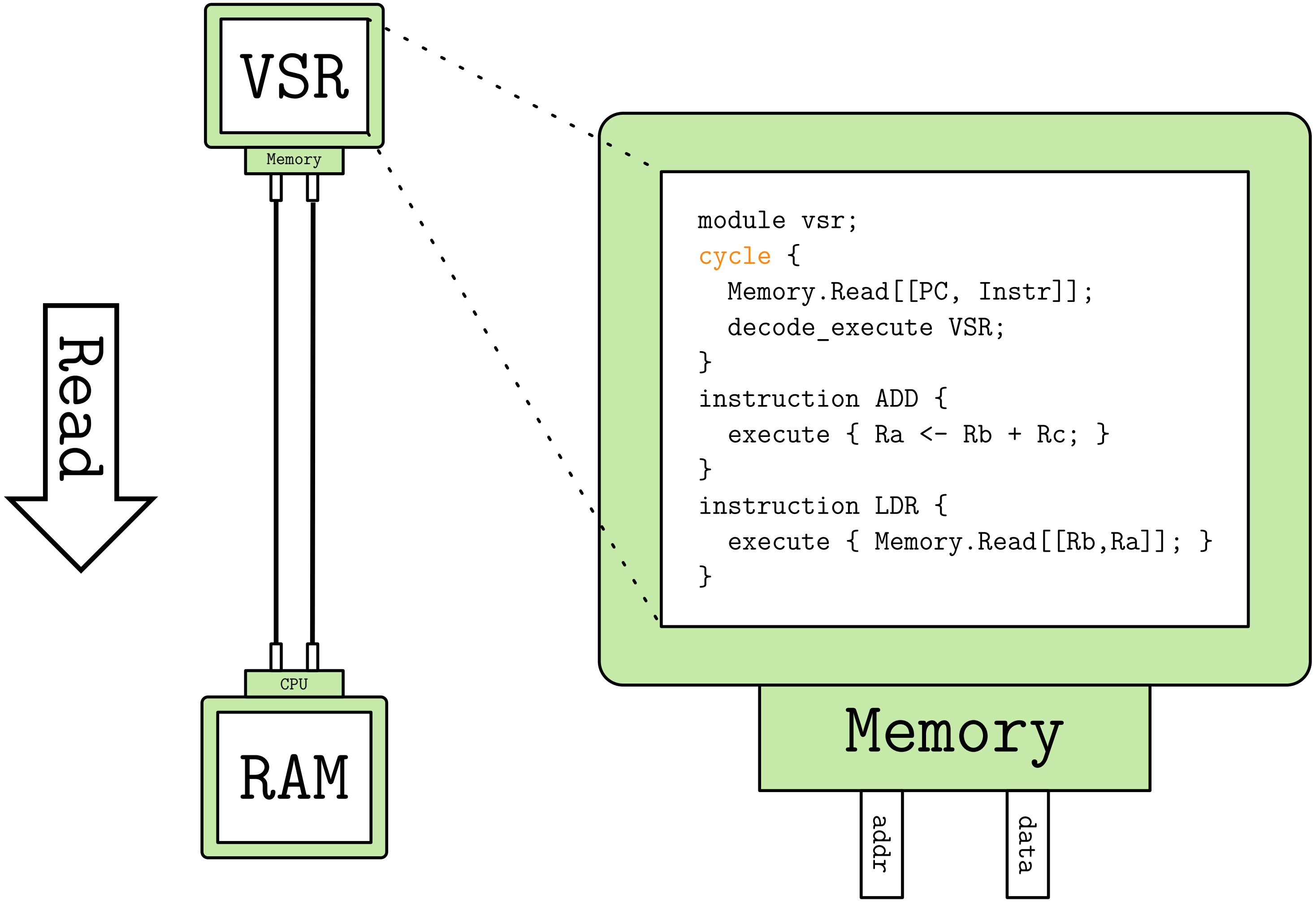
A simple model of a CPU connected to RAM.

Lyrebird



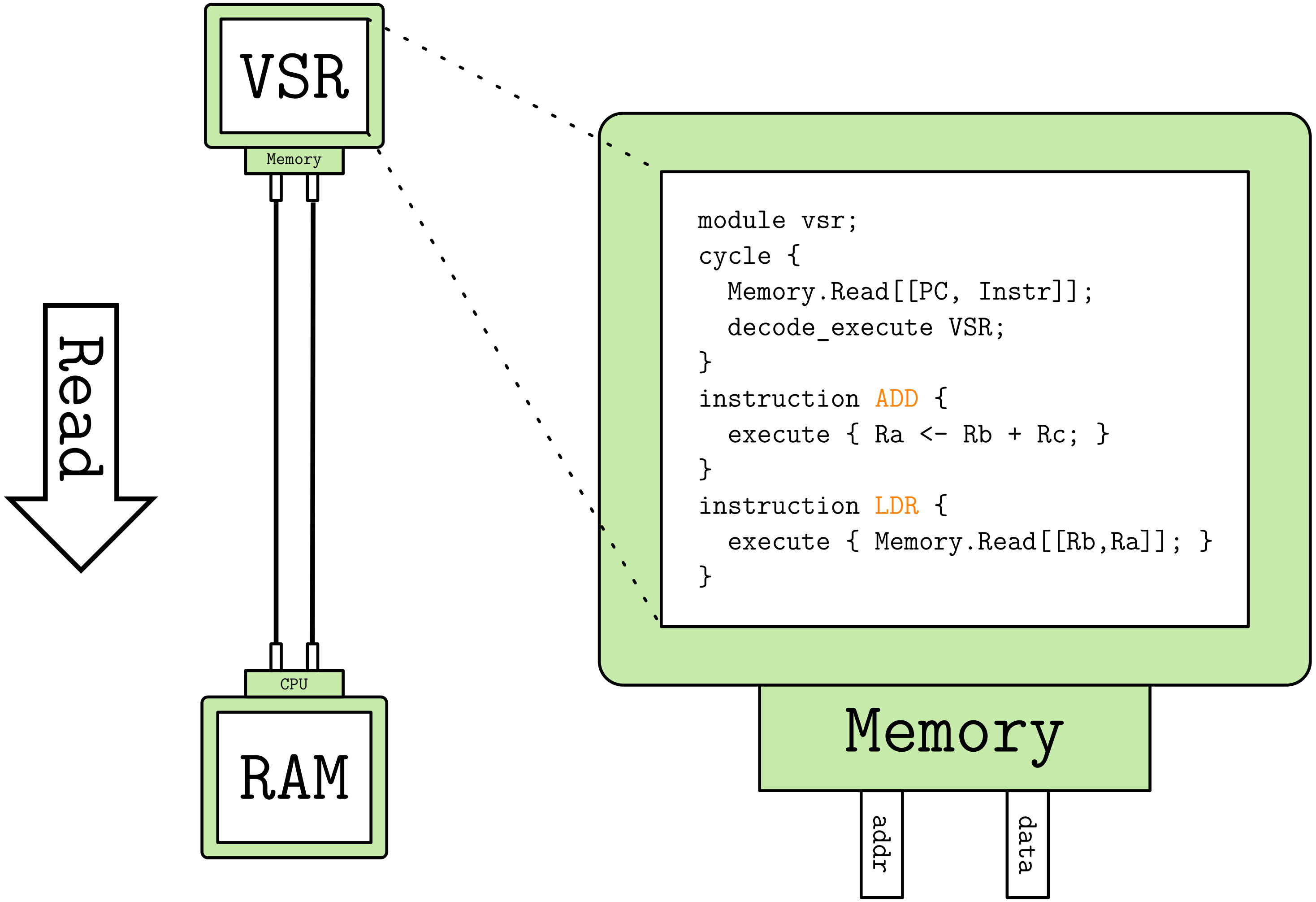
Modules are written in Lyrebird.

Lyrebird



The **cycle** specifies asynchronous behaviour.

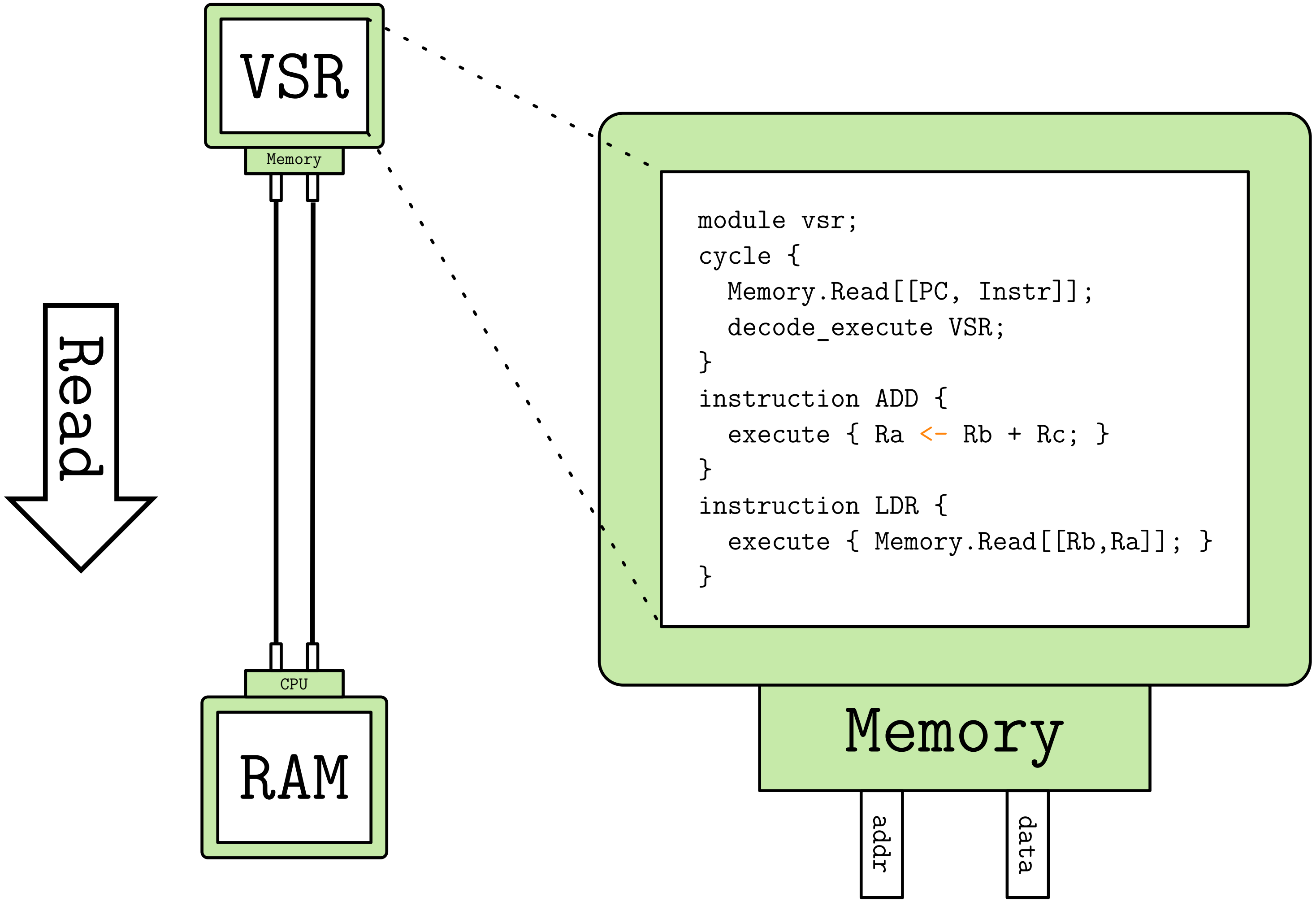
Lyrebird



Modules export **instructions.**

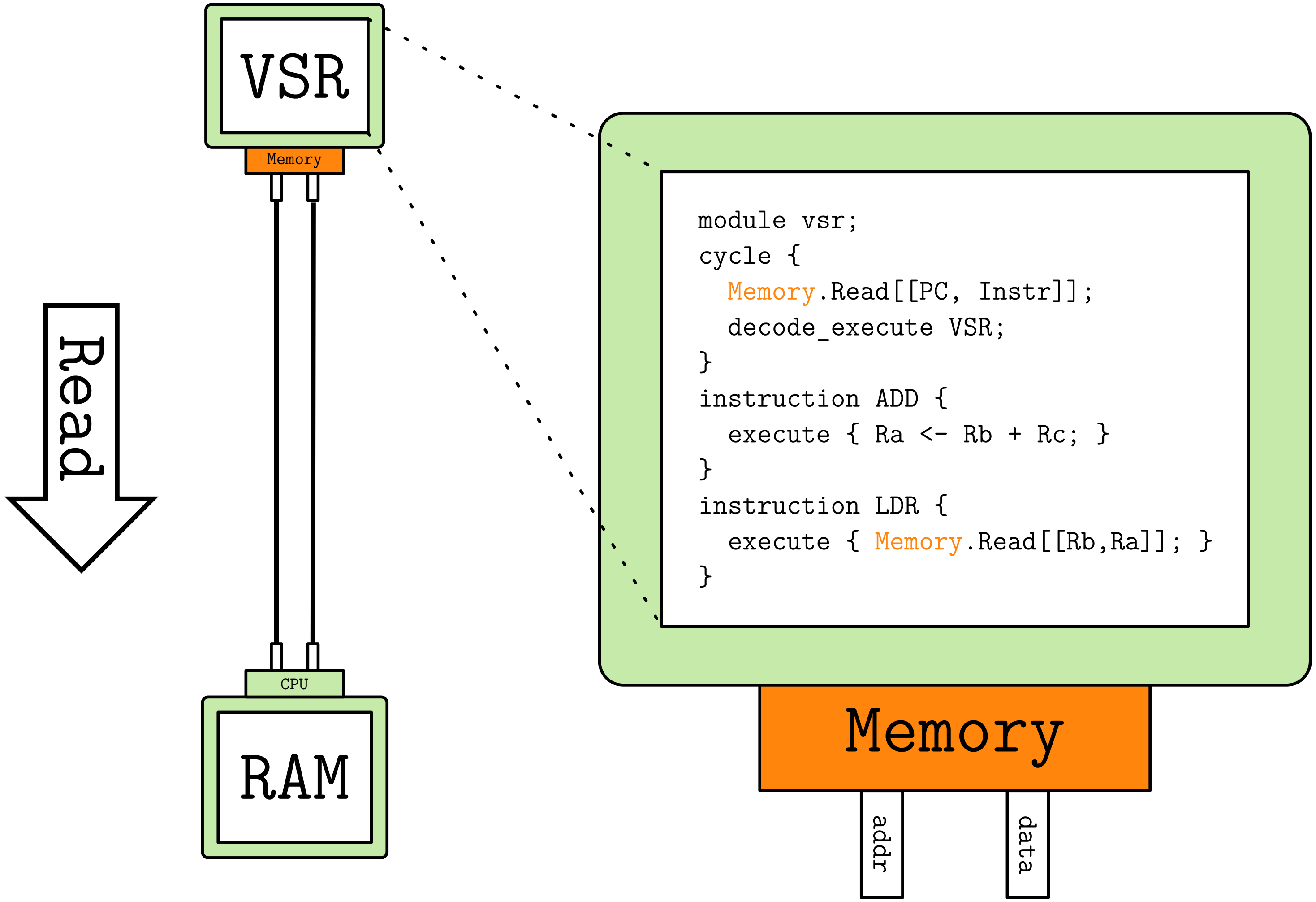
Lyrebird

NICTA



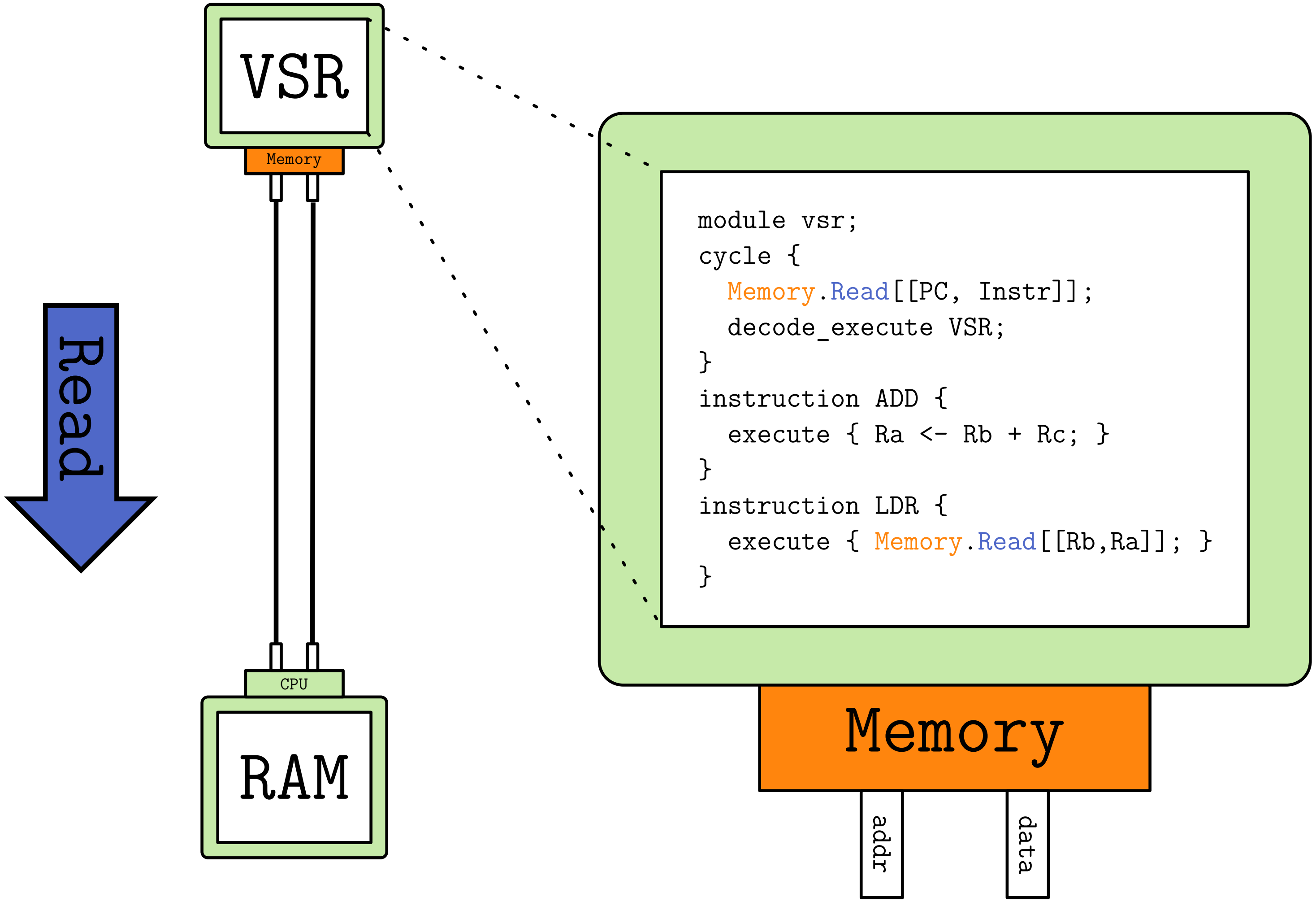
All behaviour is built from **register transfers.**

Lyrebird



Modules are linked by **interfaces**.

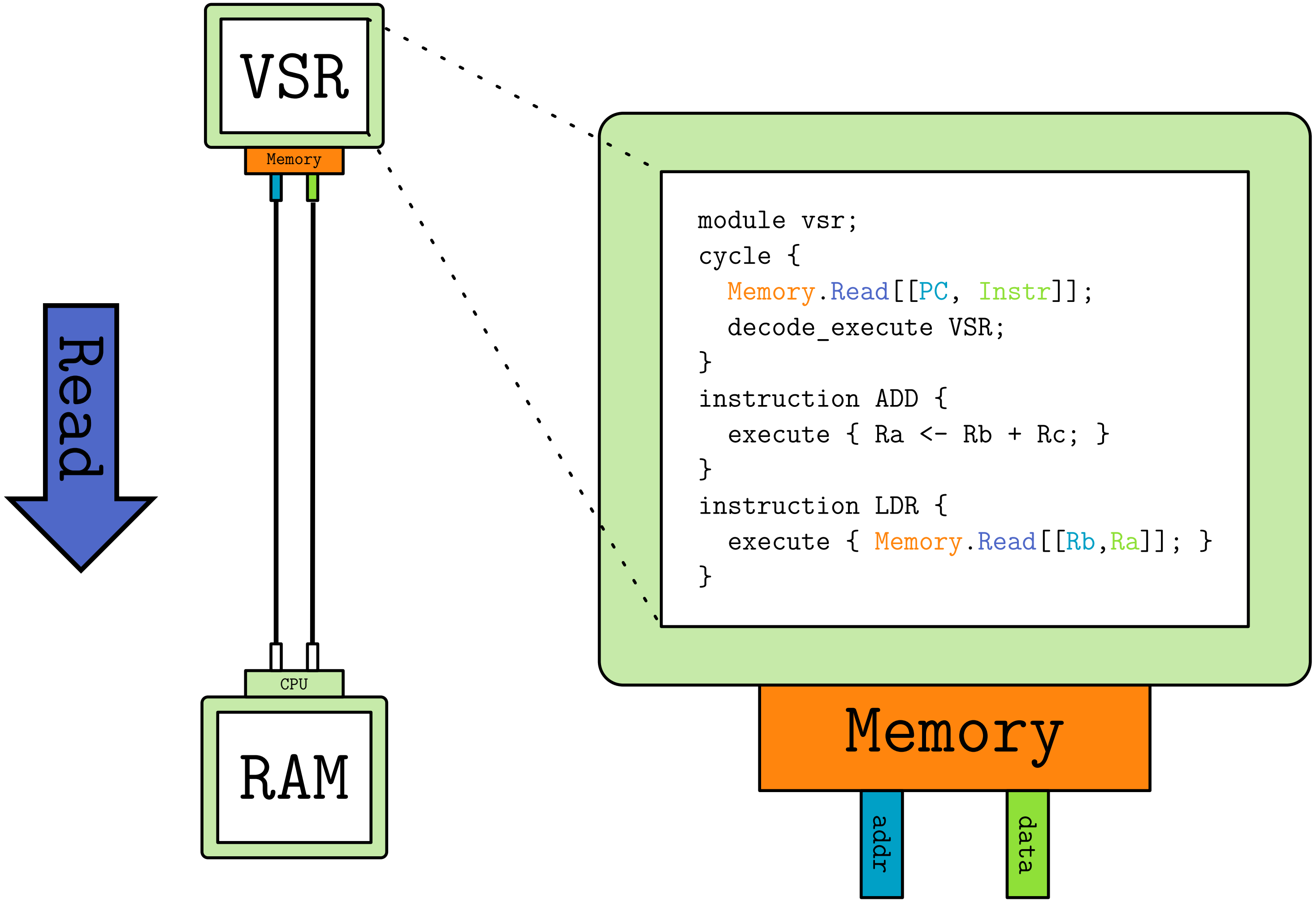
Lyrebird




```
module vsr;
cycle {
  Memory.Read[[PC, Instr]];
  decode_execute VSR;
}
instruction ADD {
  execute { Ra <- Rb + Rc; }
}
instruction LDR {
  execute { Memory.Read[[Rb,Ra]]; }
}
```

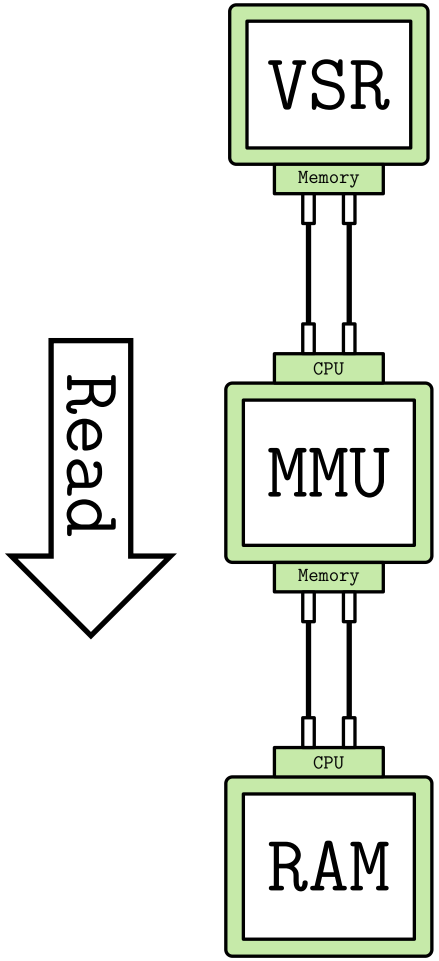
Interfaces define **transactions**.

Lyrebird



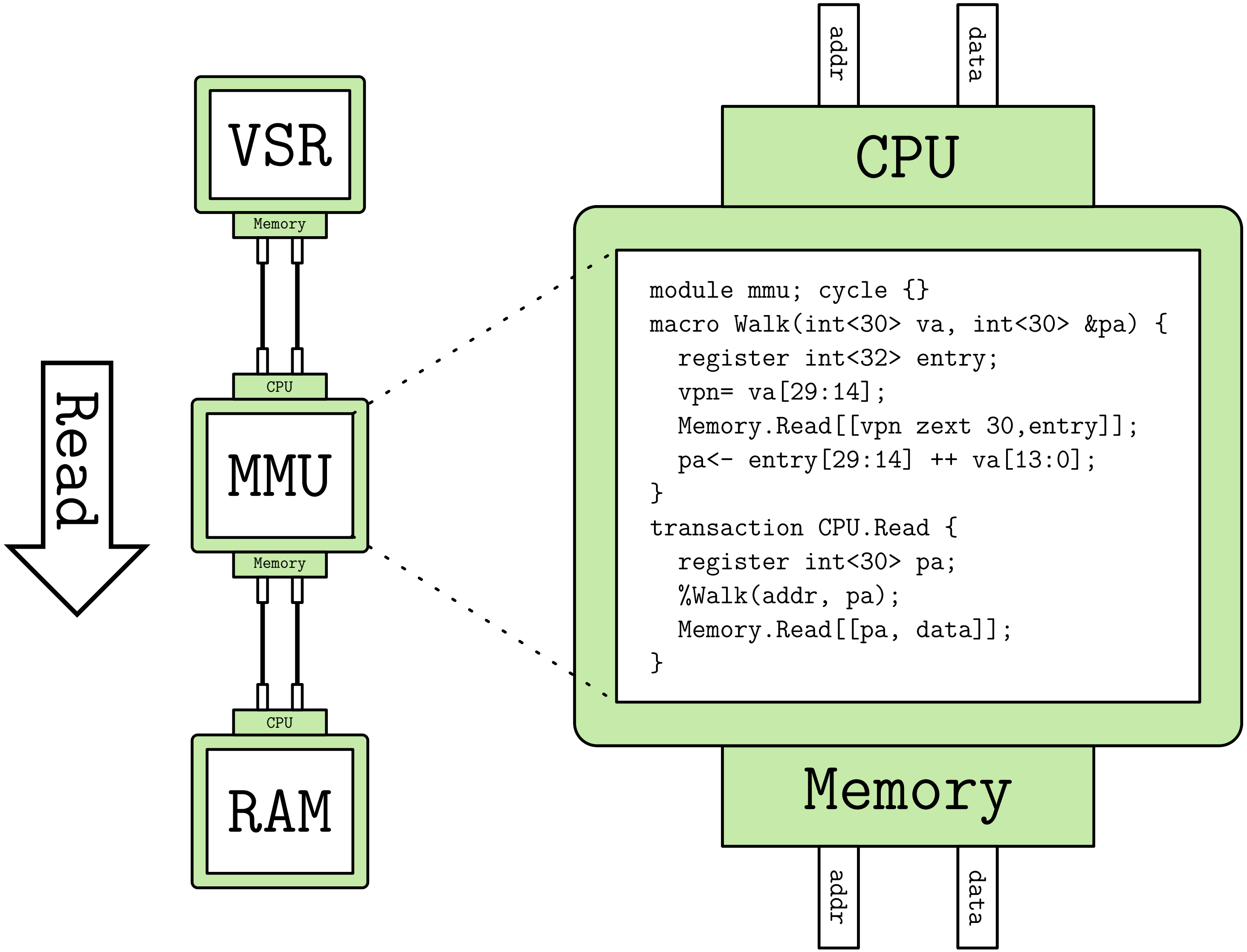
Transactions access the **datapath**.

Lyrebird



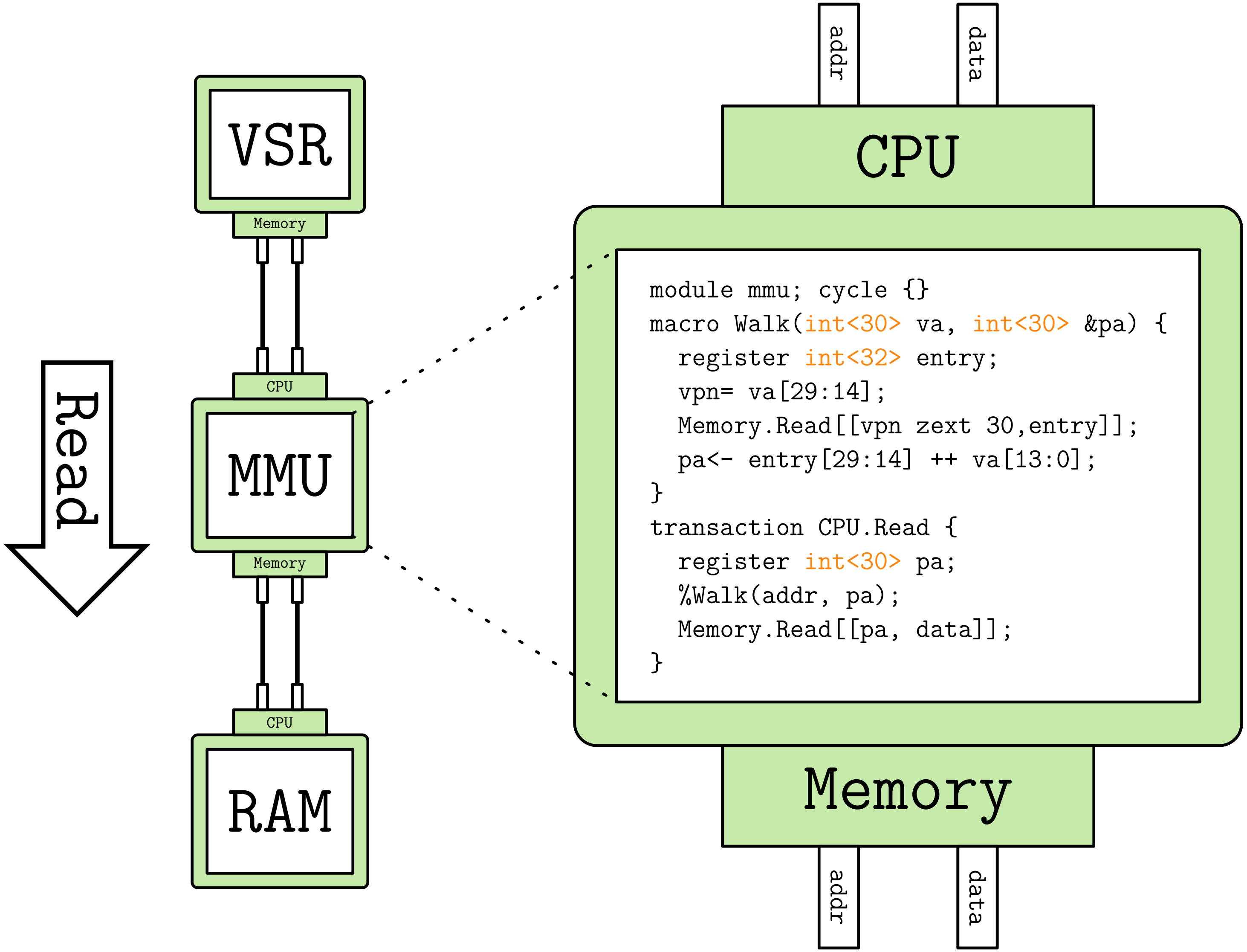
Interfaces and modules allow different implementations.

Lyrebird



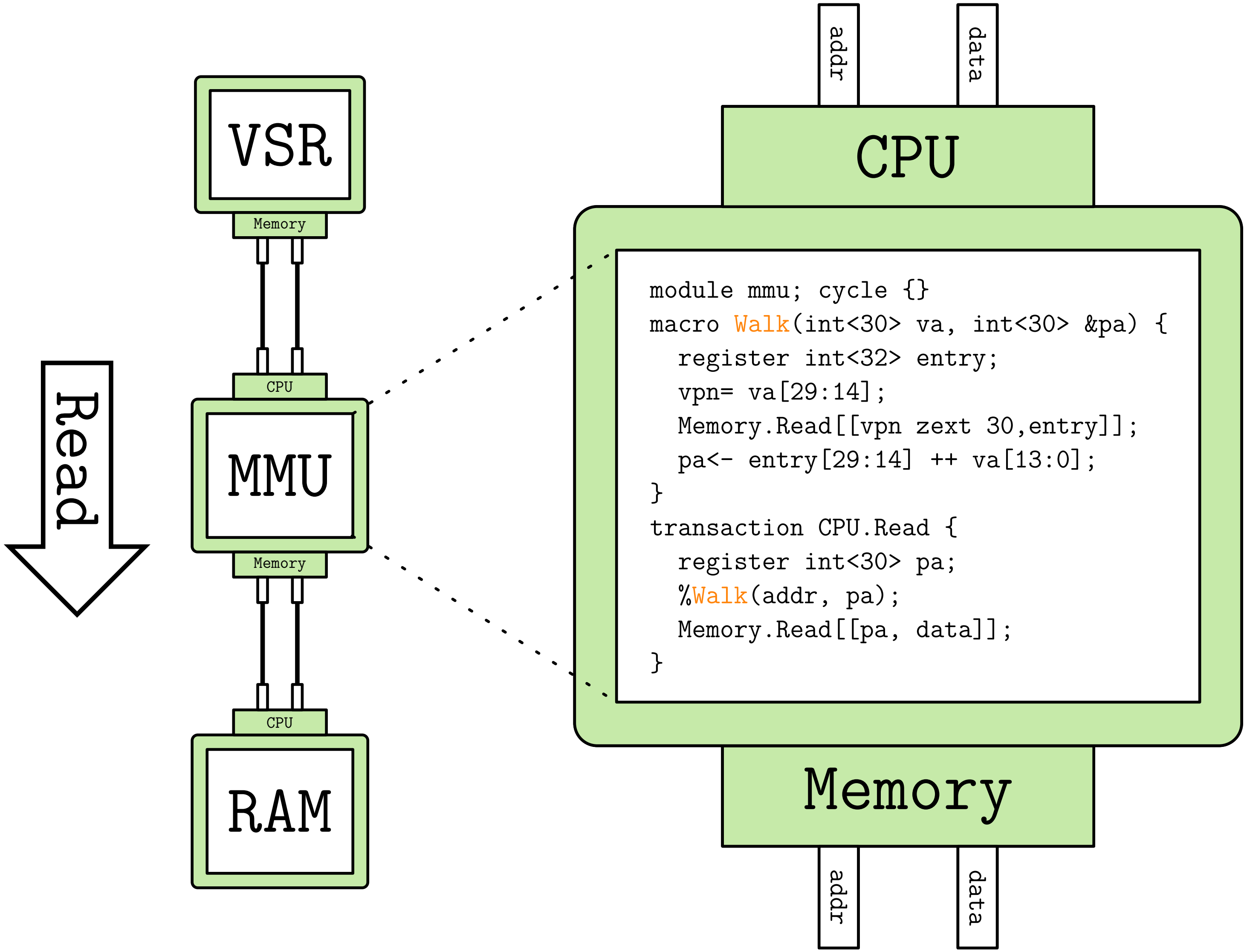
Lyrebird can also be used to model **devices**.

Lyrebird



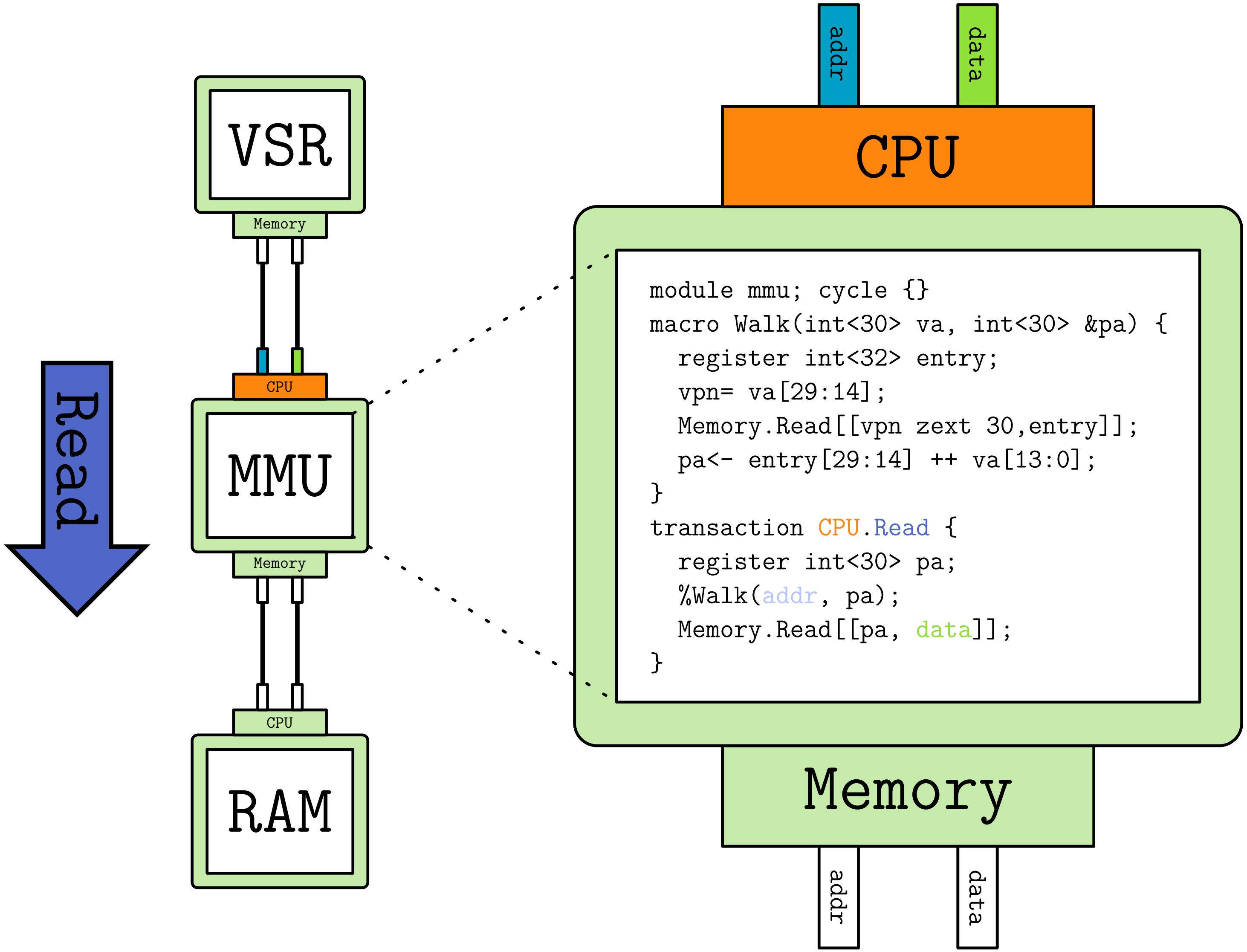
Register types have explicit **width**.

Lyrebird



Type-checked **macros** minimize duplication.

Lyrebird



Transactions are **implemented** by modules.

Lyrebird

ARMv6 Model:

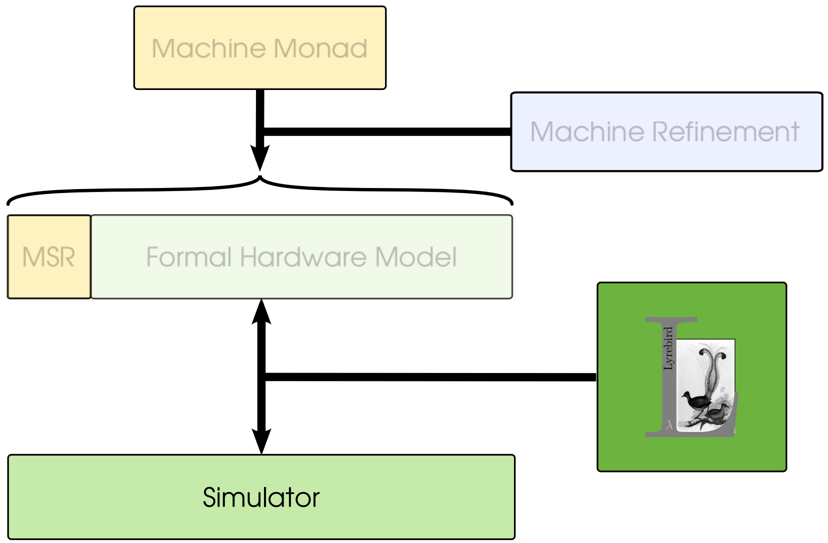
We have an ARMv6 user-level integer instruction model.

Floating-point and vector operations are excluded.

The complete model is approximately 1600 lines.

We used it to validate the seL4 Haskell prototype.

Lyrebird



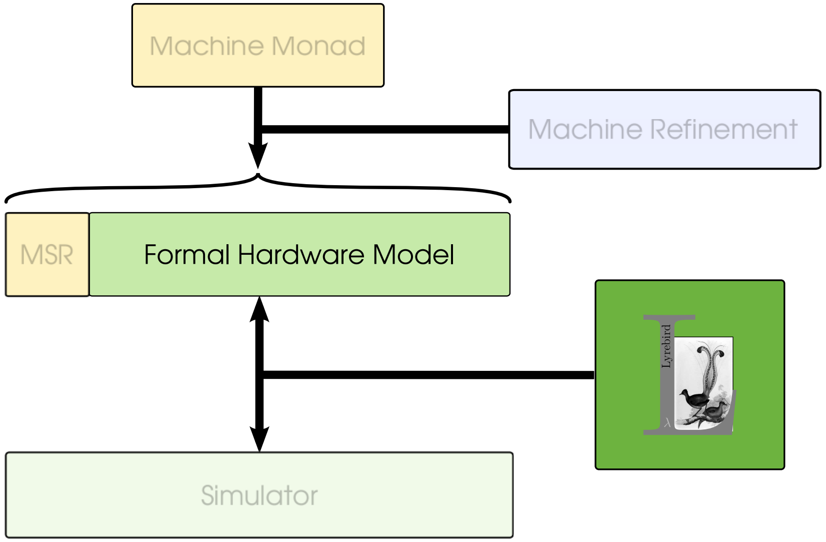
Simulation:

Register transfer is easy to simulate.

The simulator is portable and fast — 10MIPS for ARMv6 user.

The output is a single C module;
It is easily incorporated into larger simulations.

Lyrebird



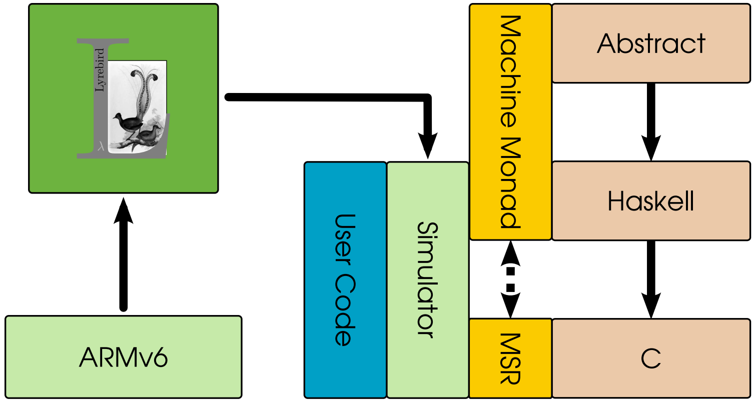
Generated Models:

An Isabelle model is generated by a tool.

We co-generate code and proofs for kernel objects.

We should be able to do the same for device structures.

Lyrebird



Rapid Modelling and Early Simulation:

We ran real user code against the Haskell seL4 model.

We found bugs in both the machine model and the kernel.

We *tested* the model against the implementation;
We fixed things before we tried to prove them.

Project Status

NICTA

Goals:

- *Development outcomes: program, proof and model.*
- *Updating the model should be easy.*
- *Easy things should be straightforward.*
- *Hard things should be possible.*

Project Status

Goals:

- ➜ *Development outcomes: program, proof and model.*

  Yes - The model is generated automatically.
- ➜ *Updating the model should be easy.*
- ➜ *Easy things should be straightforward.*
- ➜ *Hard things should be possible.*

Project Status

Goals:

➜ *Development outcomes: program, proof and model.*

Yes - The model is generated automatically.

➜ *Updating the model should be easy.*

Yes - Recompile for a new formal model.

➜ *Easy things should be straightforward.*

➜ *Hard things should be possible.*

Project Status

Goals:

- ➜ *Development outcomes: program, proof and model.*

  Yes - The model is generated automatically.
- ➜ *Updating the model should be easy.*

  Yes - Recompile for a new formal model.
- ➜ *Easy things should be straightforward.*

  Yes - User-level ARMv6 in 1600 lines.
- ➜ *Hard things should be possible.*

Project Status

Goals:

- *Development outcomes: program, proof and model.*

  Yes - The model is generated automatically.
- *Updating the model should be easy.*

  Yes - Recompile for a new formal model.
- *Easy things should be straightforward.*

  Yes - User-level ARMv6 in 1600 lines.
- *Hard things should be possible.*

  Maybe - Work is ongoing.

Future Work

Semantics:

Model generation is not ideal, the generator is trusted.

A statement's meaning should be intrinsic.

Building a semantics early will force discipline.

Underspecification:

Behaviour is often undefined or non-deterministic.

Should be modelled by underspecification and assertions.

Future Work

NICTA

The Abstract Model Stack:

We should end up with a very detailed model of the machine.

We'd rather reason about a simple, abstract machine.

We'll build the simpler model in *layers*.

Validation:

Any model must be extensively validated against hardware.

It must also be consistent with existing models e.g. Fox et. al.

Many models exist in different formalisms, this is a challenge.

QUESTIONS?