LYREBIRD

David Cock
davec@cse.unsw.edu.au
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.

Code
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.

Proof

Model

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?

Model

seL4
Code

L4
Proof
What is the Motivation?

Model

Abstract

Haskell

C

\( \lambda \)

\( \text{L4\textsuperscript{Verified}} \)

Proof
What is the Motivation?

Model

Abstract

Haskell

C

1st Refinement

2nd Refinement
What is the Motivation?

Model

"The World"

Machine Monad

Abstract

Haskell

C

1st Refinement

2nd Refinement
What is the Motivation?

- Machine Monad
- Formal Hardware Model
- Simulator

MSR

Machine Refinement

NICTA Copyright © 2011 From Imagination to Impact
What is the Motivation?

Lyrebird is a framework built around a modelling language. Tools are included to generate simulators and formal models.
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?

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>instruction</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>@100</th>
<th>@108</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What does this code do? What ends up in r1?

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>instruction</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>@100</th>
<th>@108</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

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

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>instruction</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>@100</th>
<th>@108</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

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

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>instruction</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>@100</th>
<th>@108</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td><strong>43</strong></td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>
Example

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

<table>
<thead>
<tr>
<th>address</th>
<th>data</th>
<th>instruction</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>@100</th>
<th>@108</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>...</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td>42</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td>43</td>
<td>100</td>
<td>108</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>

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.
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?
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...
Another look at the example:

What value ends up in $r1$ now?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>@1000</th>
<th>@1008</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr  r1, [r2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str  r1, [r3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add  r1, r1, #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td>e5921000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1008</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example

Another look at the example:

What value ends up in r1 now?

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>e5921000</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@1000</td>
<td>1000</td>
<td>1008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@1008</td>
<td>e5921000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another look at the example:

What value ends up in `r1` now?

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
ldr r1, [r2]
str r1, [r3]
add r1, r1, #1
```
Another look at the example:

What value ends up in \( r1 \) now?

<table>
<thead>
<tr>
<th>( \ldots )</th>
<th>( \ldots )</th>
<th>( \ldots )</th>
<th>( r1 )</th>
<th>( r2 )</th>
<th>( r3 )</th>
<th>( @1000 )</th>
<th>( @1008 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr ( r1, [r2] )</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str ( r1, [r3] )</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>e5921000</td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add ( r1, r1, #1 )</td>
<td>e5921001</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>e5921000</td>
</tr>
</tbody>
</table>
Another look at the example:

What value ends up in \( r1 \) now?

<table>
<thead>
<tr>
<th>( \ldots )</th>
<th>( \ldots )</th>
<th>( \ldots )</th>
<th>( r1 )</th>
<th>( r2 )</th>
<th>( r3 )</th>
<th>( @1000 )</th>
<th>( @1008 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>( \ldots )</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr ( r1, [r2] )</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str ( r1, [r3] )</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>e5921000</td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add ( r1, r1, #1 )</td>
<td>e5921001</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
<td>e5921000</td>
</tr>
</tbody>
</table>

Wait a minute, what was that address? Didn’t we just overwrite this instruction?
Another look at the example:

What value ends up in \( r1 \) now?

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>1008</td>
</tr>
<tr>
<td>1000</td>
<td>\text{e5921000}</td>
<td>\text{ldr} \ r1, \ [r2]</td>
<td>\text{e5921000}</td>
<td>1000</td>
<td>1008</td>
</tr>
<tr>
<td>1004</td>
<td>\text{e5832000}</td>
<td>\text{str} \ r1, \ [r3]</td>
<td>\text{e5921000}</td>
<td>1000</td>
<td>1008</td>
</tr>
<tr>
<td>1008</td>
<td>\text{e2811001}</td>
<td>\text{add} \ r1, \ r1, \ #1</td>
<td>\text{e5921001}</td>
<td>1000</td>
<td>1008</td>
</tr>
</tbody>
</table>

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>1008</td>
</tr>
<tr>
<td>1008</td>
<td>\text{e5921000}</td>
<td>\text{ldr} \ r1, \ [r2]</td>
<td>\text{e5921000}</td>
<td>1000</td>
<td>1008</td>
</tr>
</tbody>
</table>
Another look at the example:

What value ends up in \( r1 \) now?

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
</tr>
<tr>
<td>1000</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
</tr>
<tr>
<td>1004</td>
<td>e5832000</td>
<td>str r1, [r3]</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
</tr>
<tr>
<td>1008</td>
<td>e2811001</td>
<td>add r1, r1, #1</td>
<td>e5921001</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
</tr>
</tbody>
</table>

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1008</td>
<td>e5921000</td>
<td>ldr r1, [r2]</td>
<td>e5921000</td>
<td>1000</td>
<td>1008</td>
<td>e5921000</td>
</tr>
</tbody>
</table>

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.
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.
This workflow requires a common language.

Our solution is *Lyrebird*
A simple model of a CPU connected to RAM.
Modules are written in Lyrebird.

```plaintext
module vsr;
  cycle {
    Memory.Read[[PC, Instr]];
    decode_execute VSR;
  }
  instruction ADD {
    execute { Ra <- Rb + Rc; }
  }
  instruction LDR {
    execute { Memory.Read[[Rb,Ra]]; }
  }
```
The cycle specifies asynchronous behaviour.
module vsr;
cycle {
    Memory.Read[[PC, Instr]];
    decode_execute VSR;
}
instruction ADD {
    execute { Ra <- Rb + Rc; }
}
instruction LDR {
    execute { Memory.Read[[Rb,Ra]]; }
}
All behaviour is built from register transfers.
module vsr;
cycle {
    Memory.Read[[PC, Instr]]; 
    decode_execute VSR;
}
instruction ADD {
    execute { Ra <- Rb + Rc; }
}
instruction LDR {
    execute { Memory.Read[[Rb,Ra]]; }
}

Modules are linked by interfaces.
module vsr;

cycle {
    Memory.Read[[PC, Instr]];
    decode_execute VSR;
}

instruction ADD {
    execute { Ra <- Rb + Rc; }
}

instruction LDR {
    execute { Memory.Read[[Rb,Ra]]; }
}
Transactions access the *datapath*. 

```python
module vsr;
    cycle {
        Memory.Read[[PC, Instr]]; 
        decode_execute VSR;
    }
    instruction ADD {
        execute { Ra <- Rb + Rc; }
    }
    instruction LDR {
        execute { Memory.Read[[Rb,Ra]]; }
    }
```
Interfaces and modules allow different implementations.
Lyrebird can also be used to model devices.
module mmu; cycle {}
macro Walk(int<30> va, int<30> &pa) {
    register int<32> entry;
    vpn= va[29:14];
    Memory.Read[[vpn zext 30,entry]];
    pa<- entry[29:14] ++ va[13:0];
}
transaction CPU.Read {
    register int<30> pa;
    %Walk(addr, pa);
    Memory.Read[[pa, data]];
}
module mmu; cycle {}
macro \textit{Walk}(\textit{int}<30> \textit{va}, \textit{int}<30> &\textit{pa}) {
  \textit{register int}<32> \textit{entry};
  \textit{vpn} = \textit{va}[29:14];
  Memory.Read[[\textit{vpn zext} 30,\textit{entry}]];
  \textit{pa} <- \textit{entry}[29:14] ++ \textit{va}[13:0];
}
transaction CPU.Read {
  \textit{register int}<30> \textit{pa};
  %\textit{Walk}(\textit{addr}, \textit{pa});
  Memory.Read[[\textit{pa}, \textit{data}]];}

Type-checked \textbf{macros} minimize duplication.
Transactions are **implemented** by modules.

```c
module mmu; cycle {}
macro Walk(int<30> va, int<30> &pa) {
  register int<32> entry;
  vpn= va[29:14];
  Memory.Read[[vpn zext 30,entry]];
  pa<- entry[29:14] ++ va[13:0];
}
transaction CPU.Read {
  register int<30> pa;
  %Walk(addr, pa);
  Memory.Read[[pa, data]];
}
```
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.
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.
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

Goals:

➜ Development outcomes: program, proof and model.

➜ Updating the model should be easy.

➜ Easy things should be straightforward.

➜ Hard things should be possible.
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.
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.
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.
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?