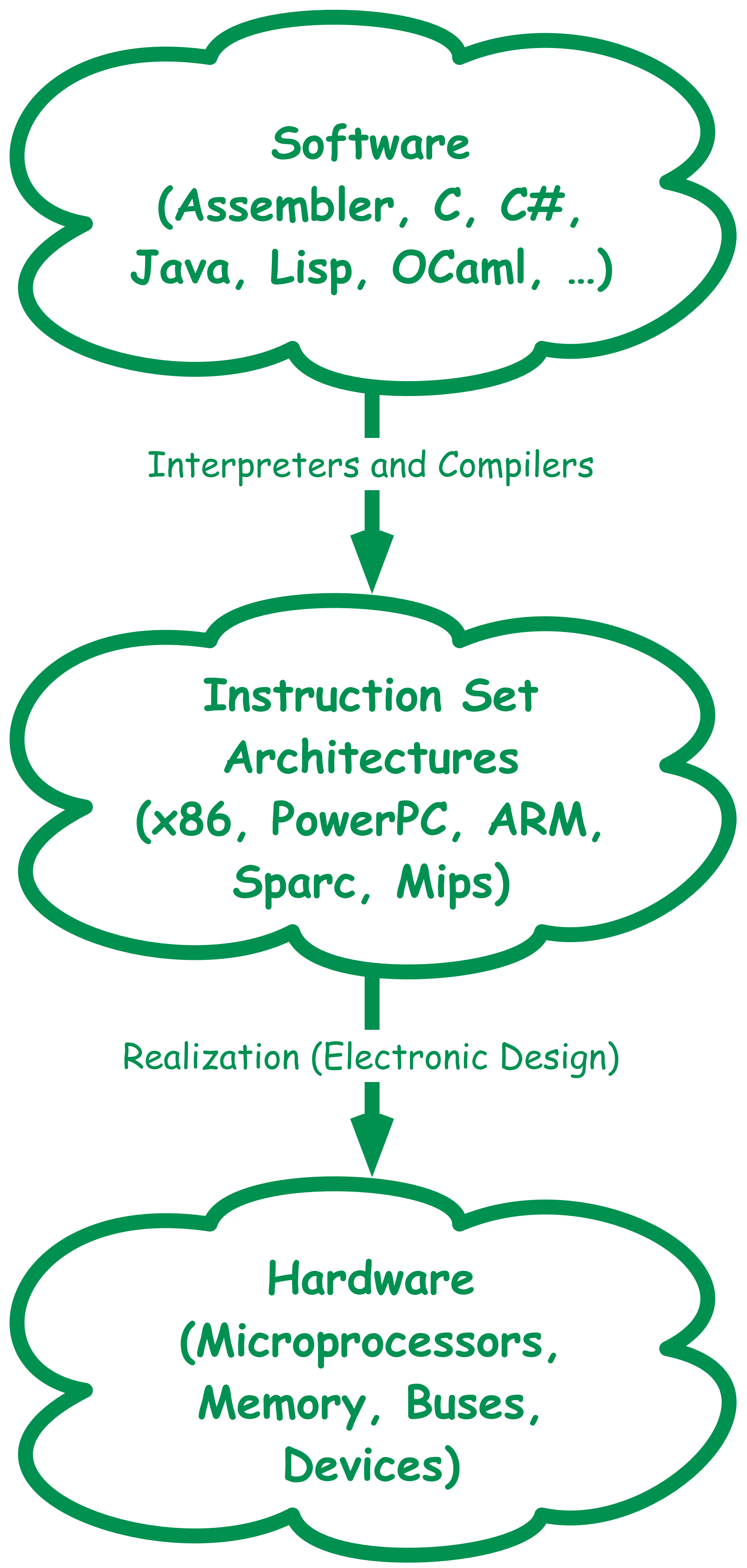
A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture

Anthony Fox and Magnus O. Myreen
University of Cambridge

Background

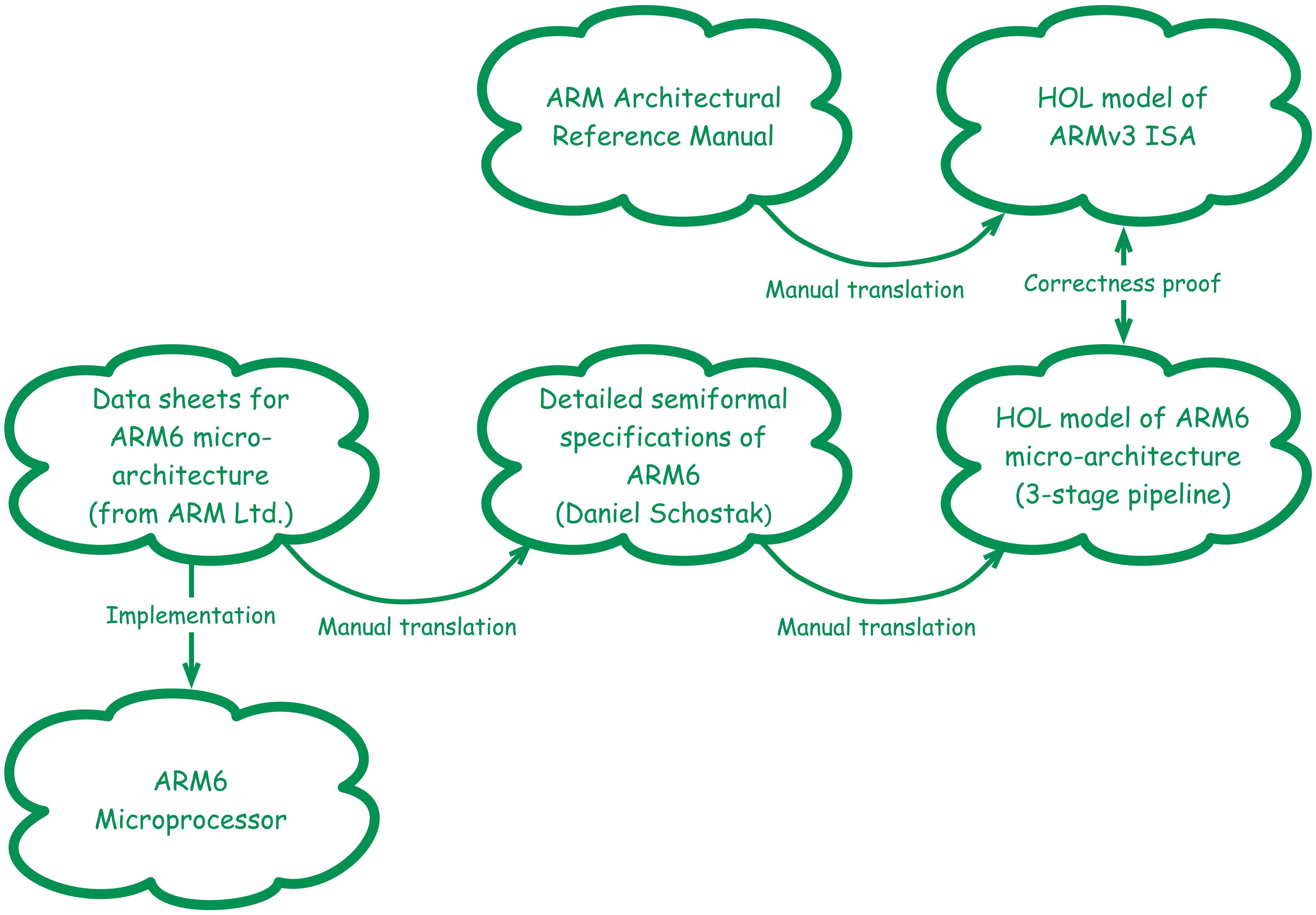
- Instruction set architectures play an important role in computing.
- They provide an **interface** between hardware and software.
- ISA models are needed for reasoning about:
  - interpreters and compilers
  - operating systems (device drivers and I/O)
  - micro-architecture designs.


Software (Assembler, C, C#, Java, Lisp, OCaml, ...)
Interpreters and Compilers
Instruction Set Architectures (x86, PowerPC, ARM, Sparc, Mips)
Realization (Electronic Design)
Hardware (Microprocessors, Memory, Buses, Devices)

ISA models

- Many instructions — even with RISC architectures.
- Architecture manuals are big, verbose and open to misinterpretation.
- Precise implementation details are often proprietary and protected by IP rights.
- Industry makes heavy use of simulators/emulators (often C based) and large validation suites. (Early processor designs become reference semantics.)
- Formal reasoning at the ISA level produces concrete results/artefacts with direct industrial relevance.
- Not many complete, high-fidelity, formal models for commercial ISAs in the public domain.

ARM6 formal verification (completed 2005)

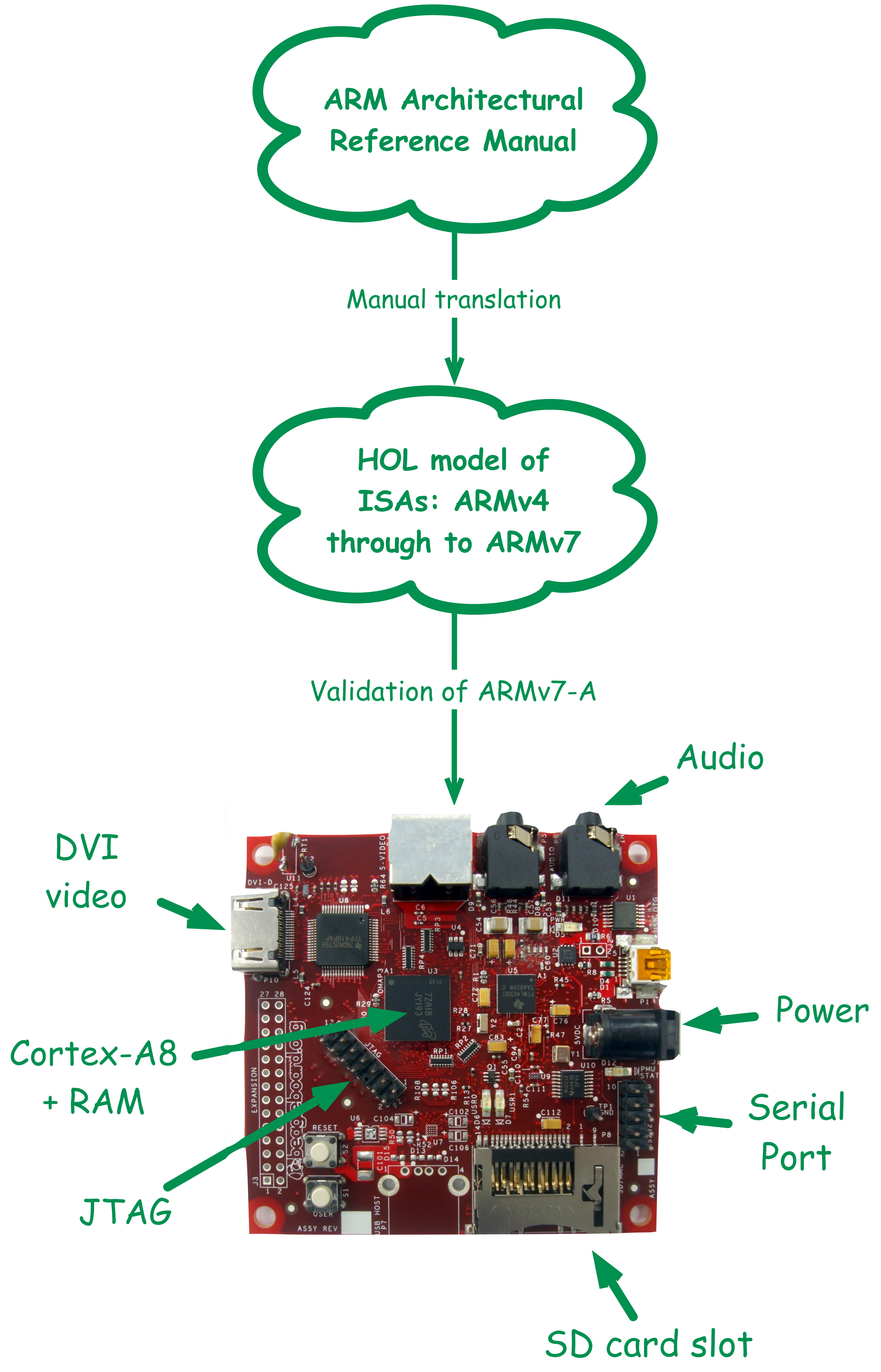

ARM Architectural Reference Manual
HOL model of ARMv3 ISA
Manual translation
Correctness proof
Data sheets for ARM6 micro-architecture (from ARM Ltd.)
Detailed semiformal specifications of ARM6 (Daniel Schostak)
HOL model of ARM6 micro-architecture (3-stage pipeline)
Implementation
Manual translation
Manual translation
ARM6 Microprocessor

Moving forward…

- Version ARMv3 is now “obsolete”, i.e. no longer supported by ARM Ltd.
- ARMv3 model was extended to ARMv4T:
    - Still widely used version — ARMv4 implemented by ARM7TDMI processor.
    - The correctness proof has not been extended.
    - The micro-architecture data is not available and we have already shown it can be done.
- Shifted efforts to machine code verification, e.g. Magnus’ Lisp interpreter.
- Now wish to look at I/O and further code verification…

Latest ISA model

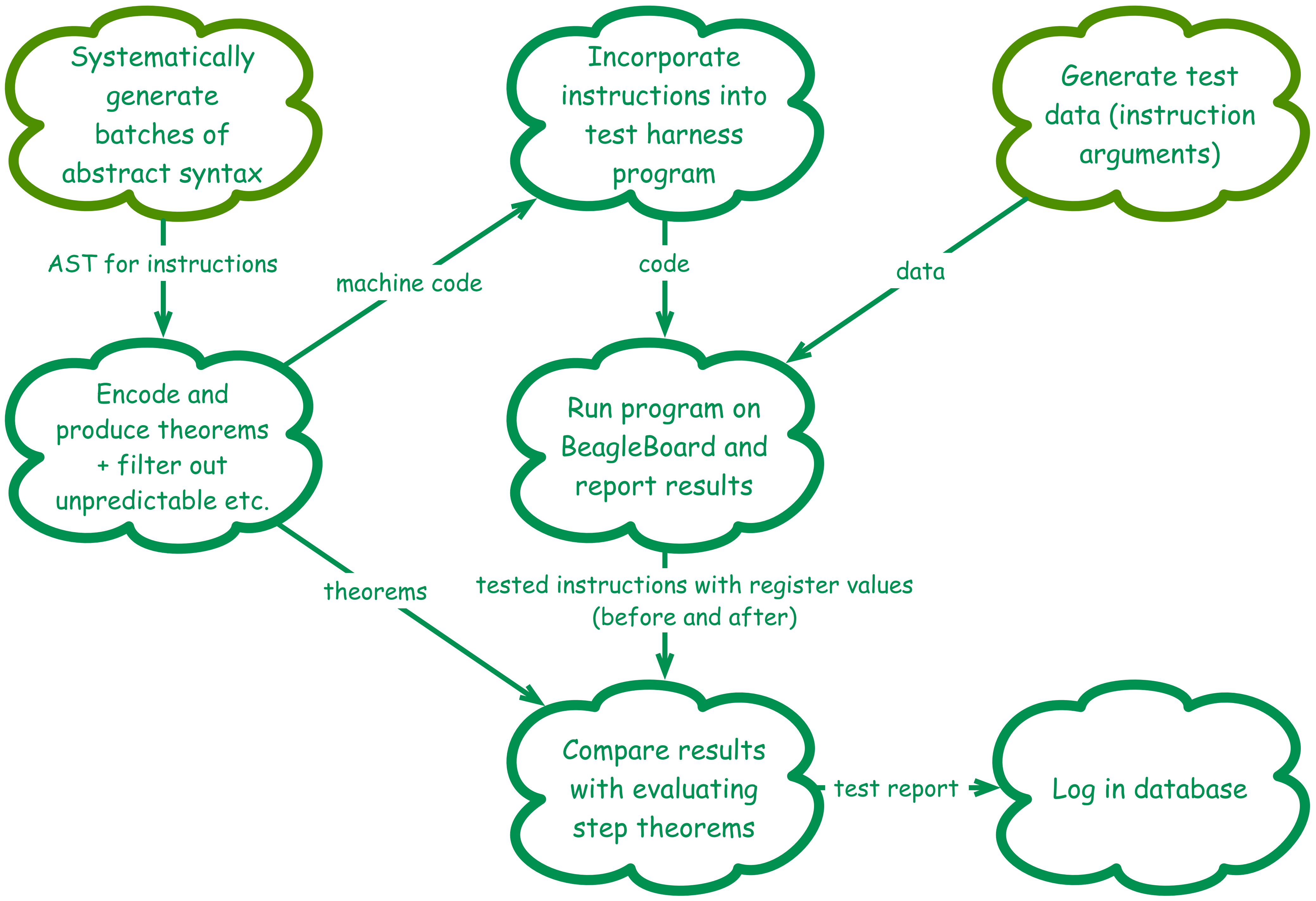
- Adopts monadic style.
- Covers multiple ISA versions:
  - all instructions, including Thumb-2 and ThumbEE.
- Validated against hardware.
- ARMv7 is version for latest Cortex processors.
- Extensive tool support:
  - custom built assembler and evaluator.
- **25K lines of development.**



Monadic specification

- New monadic model prompted by Peter Sewell's group, who are working on multi-processor memory models.
- Provides a useful abstraction layer — hides underlying computational semantics.
- Nicely suited to modelling systems with "state" — no need to explicitly pass state as a parameter (with heavy use of clunky *let*-statements).
- Provides clean mechanisms for handling:
  - multiple ISA versions
  - memory and I/O
  - parallelism
  - error states

Establishing trust


Systematically generate batches of abstract syntax
AST for instructions
Encode and produce theorems + filter out unpredictable etc.
machine code
Incorporate instructions into test harness program
code
Generate test data (instruction arguments)
data
Run program on BeagleBoard and report results
theorems
tested instructions with register values (before and after)
Compare results with evaluating step theorems
test report
Log in database

Single step theorems (specializing the model)

- We wish to know the effect of running a particular op-code, say 0xE2921003.
- This can be expressed with a theorem of the form:

$$\forall s.\ \mathrm{P}(s) \Rightarrow (\mathrm{NEXT}(s) = \mathrm{f}_{\mathrm{P}}(s))$$

- NEXT is the next-state function for the ARM model.
- Predicate P characterises states in which the op-code is about to be run.
- f_{P} is a state update function for the op-code.
- Originally used for production of Hoare triples, then used for validation.

Tool for single step theorems

- A tool has been developed in HOL to produce single step theorems on-the-fly using forward proof (symbolic evaluation).
- This has a number of advantages:
    - Users don’t have to be intimate with the HOL model or all the intricacies of the ARM architecture. This becomes essential for large, complex models.
    - Users can simply ask **“what does this instruction do?”**.
    - Hoare triples for instructions can be generated on demand — manual proof is not required, nor a big database of theorems.

Demo…

Implementation

- Implemented in Standard ML using HOL4's conversions, rules and other primitive functions.

- Simple interactive interface, for example:

```
> armLib.arm_step "v5t, thumb" "1889"
```

or

```
> armLib.arm_steps_from_string "v5t" "thumb\n adds r1,r2"
```

Basic approach

- Create predicate P
  - P is used to direct the evaluation (e.g. assign T or F to conditional statements).
  - It selects the correct architecture/configuration and instruction set.
  - It ensures the instruction is in memory at the PC address.
  - Extra side-conditions are added to avoid “unpredictable” cases, e.g. registers must be aligned when used as memory addresses. The op-code must be examined (decoded) to achieve this.
- Term-rewrite NEXT(s) using condition P(s).

Rewriting with side-conditions

- The HOL4 simplifier can rewrite with side-conditions. For example:

  $$b \Rightarrow (\text{if } b \text{ then } x \text{ else } y)$$

  will simplify to

  $$b \Rightarrow x$$

- But the simplifier is pretty slow and we want a fast tool.
- HOL's call-by-value rewriter EVAL is fast but doesn't allow side-conditions.
- Technique has been developed for doing bulk of work with EVAL.
- Details are in the paper.

Validation (some observations)

- Able to test most instructions and fix some bugs.
- Data processing (arithmetic/logic) instructions easiest to test.
- Loads/Stores and Branches a bit more complicated — need to be a little cleverer in writing the test harness code.
- System instructions problematic — user code can’t access privileged state and system calls exit to OS’s exception handler code. (BeagleBoard runs Ångström Linux.)
- Maybe need to use JTAG interface.
- Further testing was done through EmitML. Compiled MD5 to ARM.
- Result available at: http://www.cl.cam.ac.uk/~acjf3/arm

Summary

- ISA models are very useful.
- Not easy to accurately formalize — **large**, with **lots of intricacies**.
- It is important to provide high-fidelity, wide coverage and good tool support:
  - An abstract “assembly code” model leaves a fairly big gap to the ISA.
  - Accurately represent the programmer’s model state space (use machine words).
  - Define instruction decoder functions (work with machine code) — better than working with ASTs alone.
  - This all facilitates direct comparison with hardware.
- Structured and extensive validation becomes essential.
- Questions?