Designing a CPU model: from a pseudo-formal document to fast code

Frédéric Blanqui, Claude Helmstetter, Vania Joloboff, Jean-François Monin, Xiaomu Shi

INRIA - LIAMA - FORMES

http://formes.asia/

Functional simulators of Systems-on-Chip

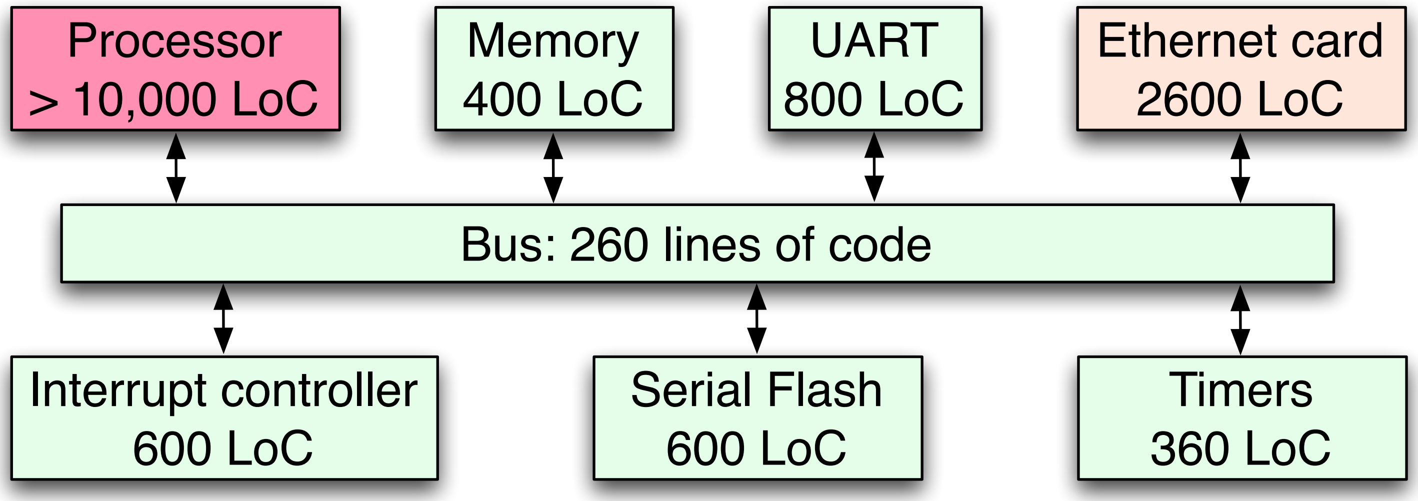
- Functional full-system models (instruction-accurate):
  - allows fast simulation of the real embedded software
  - allows verification of system-level properties
  - used as golden model for hardware verification
  - loosely timed, because of low level code using “timeouts”
- Abstraction level: functional full-system models are:
  - less abstract than “Software Development Kit”, used for the development of applications (e.g., IPhone SDK)
    - use native simulation => cannot simulate low level code
  - more abstract than time-accurate models, used for performance evaluation
    - more accurate => simulations are slower

The open-source SimSoC simulator

- SimSoC: Simulator of Systems-on-Chip
  - Based on SystemC and OSCI TLM-2.0.1 (Loosely-Timed level)
  - Library of component models
    - many processors models: ARM, PowerPC, MIPS (with GDB servers)
    - Bus, memory, timers, interrupt controllers, UARTs, Ethernet, etc
  - Many platforms (complete enough to boot Linux):
    - 2 models of SoC based on ARMv5
    - 1 model of SoC based on PowerPC (dual-core)
  - Distributed as open-source
    - libraries under LGPL license
    - programs under GPL license
- http://gforge.inria.fr/projects/simsoc/

Development of simulators

- Developing a simulator is costly: ~50.000 lines of code
- Some parts can be reused (if norms are respected)
- Processors are the most complicated parts
  - more and more instructions in new instructions sets
  - bottleneck for simulation speed, so optimizations are needed

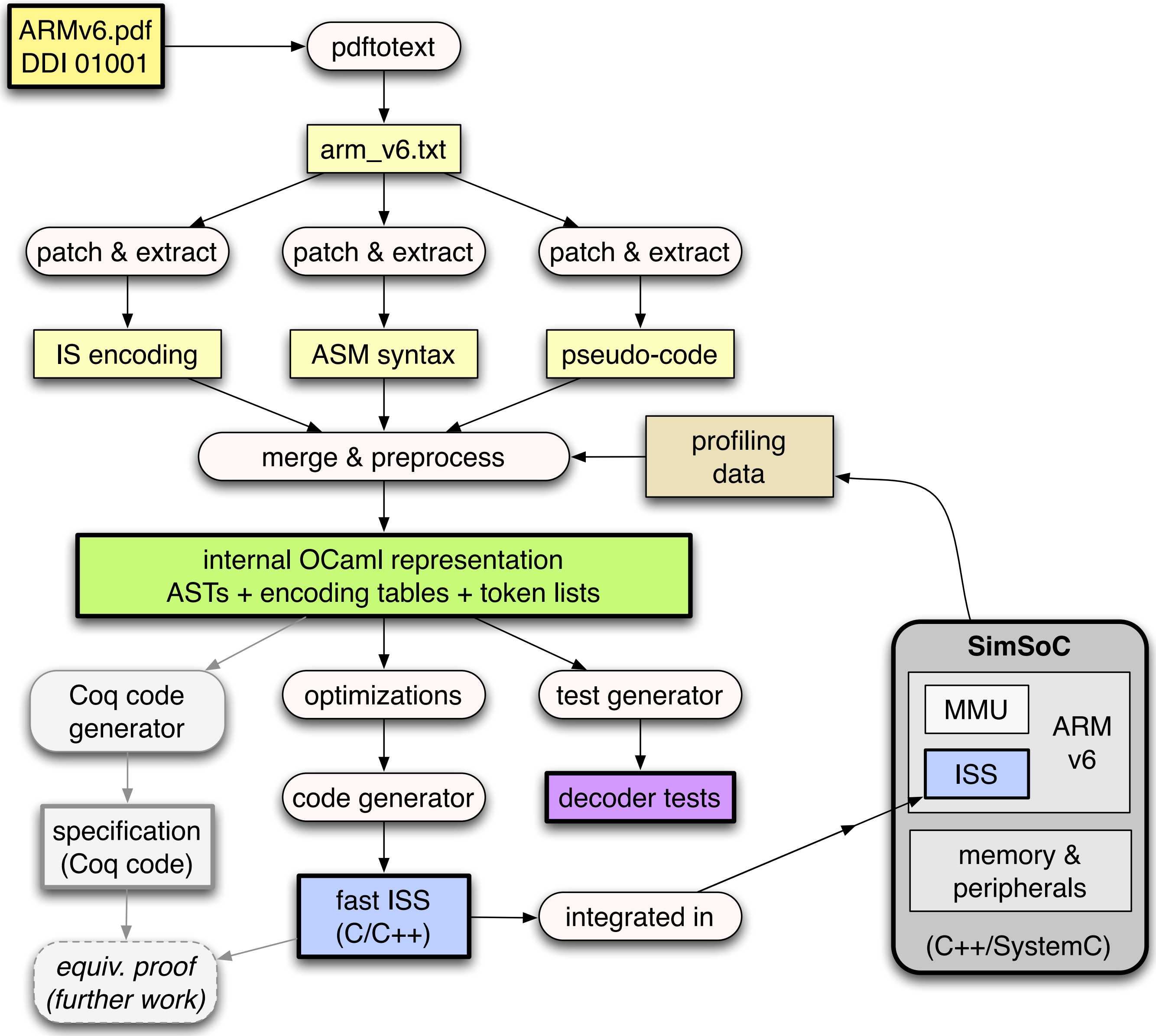


Generation of an ARMv6 ISS

- The code of an Instruction Set Simulator (ISS) is repetitive
- 1st idea: generate the code of the ISS from an in-memory description
  - Apply transformations and analysis before code generation
- Reference manuals contains pseudo-formal parts:
  - The semantics of each instruction is described by pseudo-code
  - Instruction syntax, instruction encoding
- 2nd idea: extract automatically the formal description from the reference manual
- Application to the **ARMv6** instruction set
  - Note: SimSoC provides an hand-written **ARMv5** ISS

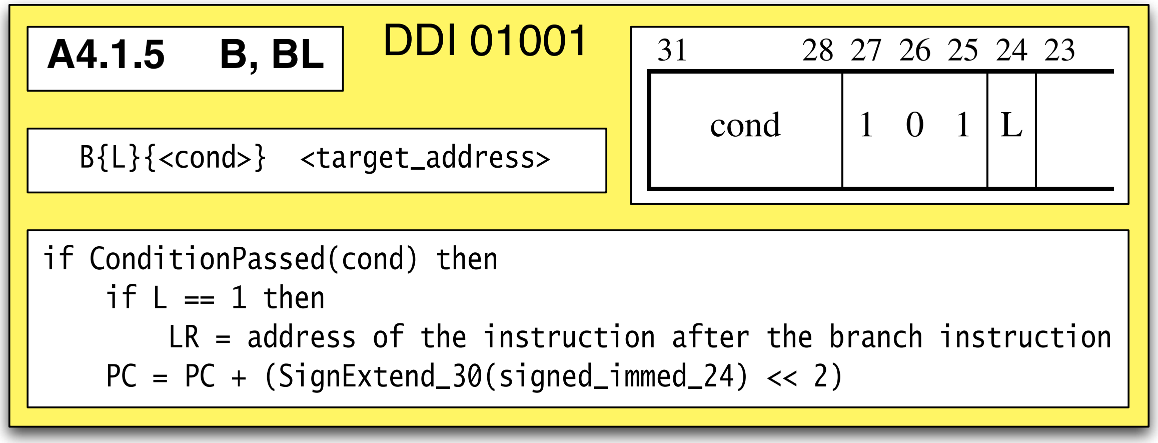
Outline & Architecture overview

- Introduction
- **Extraction**
- **Transformations**
- Generation
- Experiments
- Conclusion



The pseudo-formal parts of the manual

- Many parts of the instruction descriptions can be extracted



A4.1.5 B, BL

DDI 01001

31 … 28	27 26 25	24	23 …
cond	1 0 1	L	

```
B{L}{<cond>}   <target_address>
```

```
if ConditionPassed(cond) then
    if L == 1 then
        LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend_30(signed_immed_24) << 2)
```

- Other parts (English text) are either:
  - ignored (e.g., examples, instruction usage, etc)
  - interpreted manually and injected into the OCaml analyser and generator (e.g., validity constraints such as “W ⇒ Rn != PC”)
  - interpreted manually and included in C/C++ libraries (e.g., saturated arithmetic, memory model, etc)

Automatic extraction

- Automatic extraction **avoids a manual step**
  - manual translation could introduce errors
  - the extractor code is ad hoc but its development is simple
- Issues
  - the pseudo-code contains **ambiguities**
    - type information
    - ordering of side-effects
    - exceptions not described in the code
  - the pseudo-code contains **bugs** (in document ref. DDI 01001)
    - syntax errors (e.g., unclosed parenthesis)
    - code not conform with the textual description
      (e.g., condition check missing in CLZ,
      wrong assignment at the end of LDRBT)

Transformations and optimizations

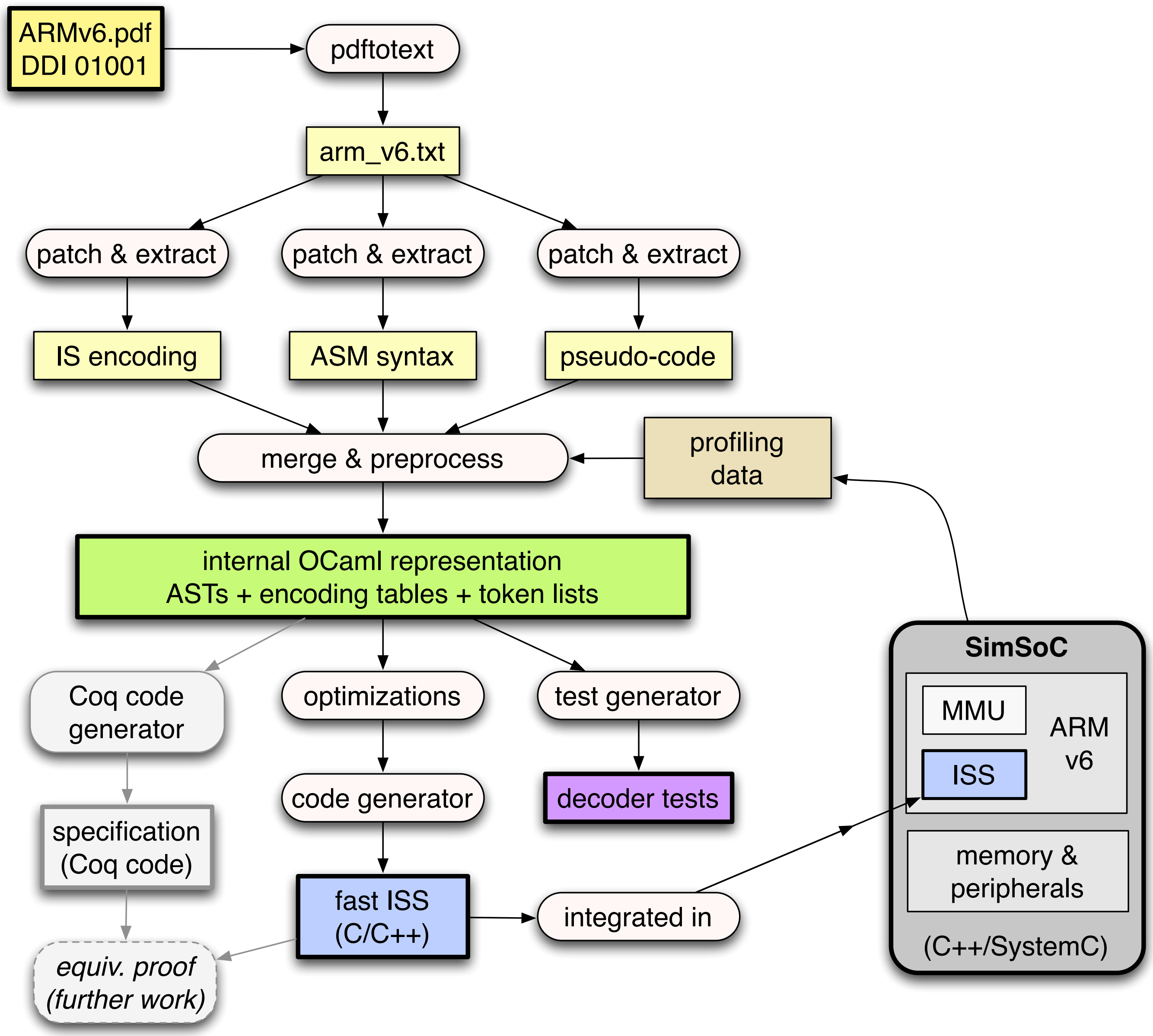
- Transformations fixing ambiguities
  - Transform “CarryFrom(A+B)” in “CarryFromAdd(A,B)”
  - Correct the addressing mode of SRS and RFE
  - Move the base register write-back to a proper place
- Optimizations
  - Flattening: given some instructions $I_1,..,I_n$ and the related addressing modes $M_1,..,M_k$, we generate $n{\times}k$ instructions I_iM_j
    - Append the code, merge the binary encoding and the assembly syntax
  - Pre-computation of static sub-expression
    - some sub-expressions can be computed at decode-time instead of execution time (e.g., “NbOfSetBitsIn(reg_list)*4”)
  - Specialize instructions, using feedback from the simulator

Instruction specialization

- An ARM instruction such as “ADD”,
  - checks the **condition field**, to decide whether the instruction must be executed or skipped
  - checks the **S bit**, to decide whether the flags must be updated
- Most of the time (as known by running testbeds):
  - “ADD” is used with “S=false” and “condition=always”
- A specialized instruction “ADD_S0_Always” is generated
  - the AST is duplicated
  - the condition check is removed
  - S is replaced by false
  - That’s simple using OCaml

Outline & Architecture overview

- Introduction
- Extraction
- Transformations
- **Generation**
- Experiments
- Conclusion



Generation of a fast ISS

- Generated components:
  - The **types** used to store an instruction after decoding.
  - Two **decoders**: one for the main ARM instruction set and another for the Thumb instruction set.
  - The **semantics functions**, corresponding to the extracted and optimized pseudo-code.
  - The ASM **printers**, used to print debug traces.
  - The "*may_branch*" predicate that detects basic block terminators (i.e., branch instructions).

The “may_branch” predicate

- Fast simulation requires to recognize *“basic blocks”*
  - basic block = sequence of instructions always executed in a row (i.e., only the last instruction may be a branch)
- ARM architecture: PC = the general purpose register R15
  - => there are a lot of branch instructions (e.g., ADD R15, R0, #8)
  - an “ADD <rD>, <Rn>, <oper.>” instruction may branch if “d==15”
- For each instruction, the code is analyzed to deduce the “may_branch” condition (e.g., “d==15”)
  - Fully automatic for most instructions
  - Some special cases are managed by hand (e.g., LDM instruction)

More outputs: decoder tests

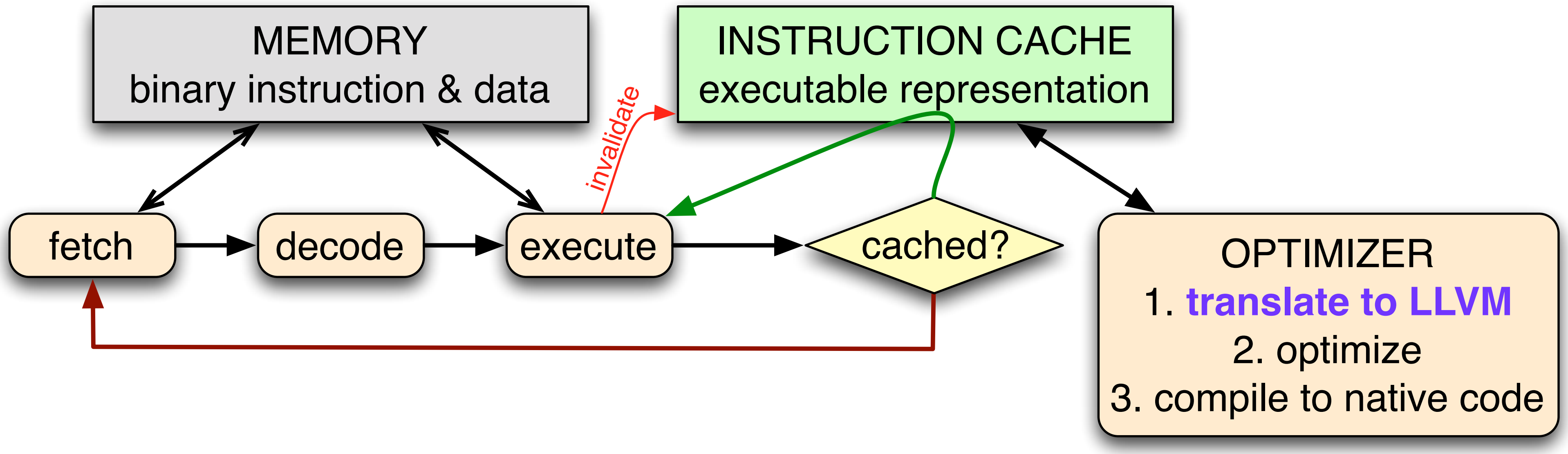
- A decoder test =
  - a binary word
  - + the corresponding instruction (e.g., in assembly syntax)
- Can be generated using the same internal representation
  - we have developed a random test generator
- Results:
  - 1 serious bug (BKPT not recognized)
  - 2 minors bugs (in printers) found

More outputs: Coq specification

- Generation of a formal Coq specification (done)
  - based on the same internal representation
  - no optimizations are applied
  - allows to simulate simple tests
    - extremely slow, because the code is tailored to formal proof
- Long-term goal: (!ongoing work!)
  - Evaluate whether a proof assistant such as Coq can be used to improve the confidence in virtual prototypes
  - Idea: prove that the C code used in the ISS is equivalent to the Coq code
  - Work not finished; still too early to say whether it will succeed.

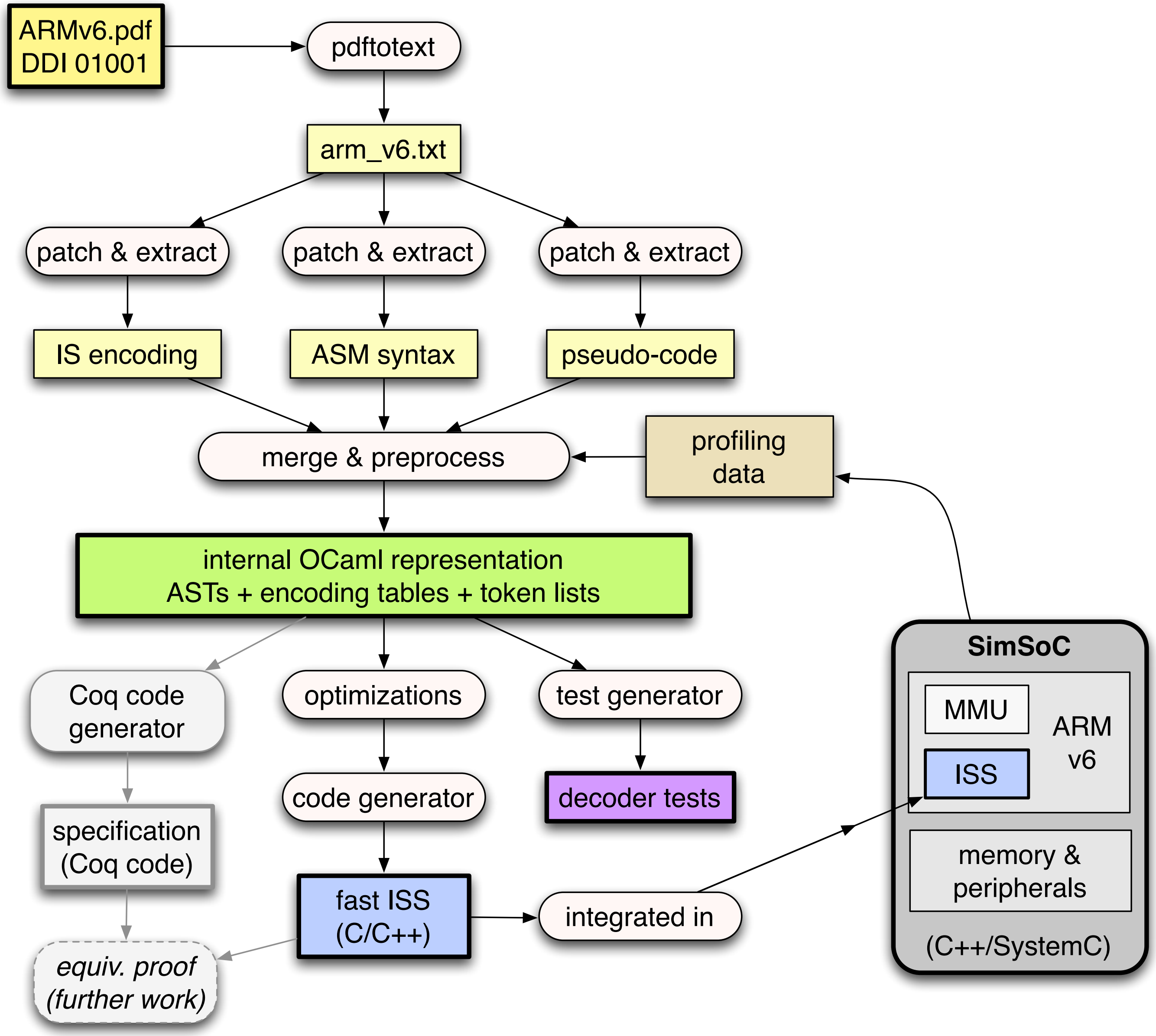
More outputs: ARM → LLVM translator

- LLVM: library including an optimizing runtime compiler
- Compiling frequently executed ARM code to optimized native code allows to speed up the simulations
- Part of the ARM to LLVM translator is generated from the same internal representation



Outline & Architecture overview

- Introduction
- Extraction
- Transformations
- Generation
- **Experiments**
- Conclusion



ISS validation

- ISS first validated and debugged using unitary tests
- Decoder validated using the automatically generated tests
- Next, after integration of the ISS into SimSoC
  - Linux boot on the STMicroelectronics SPEArPlus600 SoC
    - a few bugs found (e.g., in case of memory exception)
  - Linux boot on the Texas Instrument AM1707 SoC
    - no more bugs

ISS performances

- 3 benchmarks
  - loop (extremely simple), sorting (pretty simple), crypto (more complex)
  - compiled with different compilations flags (O0/O3, thumb mode)
  - tested using 3 computers: Linux 32, Linux 64, MacOSX (64)
  - benchmarks compatible with ARMv5
- Compared to our previous ARMv5 hand-written ISS *(without using LLVM)*
  - on Linux 64: 107 Mi/s vs. 103 Mi/s (+4.3 %)
  - on Linux 32: 78 Mi/s vs. 84 Mi/s (-6.8 %)
  - on MacOSX: 92 Mi/s vs. 88 Mi/s (+4.5 %)

Reusability: SH4 (!ongoing work!)

- Question: Is the framework reusable for another architecture?
- We (Frédéric Blanqui and Frédéric Tuong) have started the same work for the SH4 architecure
- In the SH4 reference manual
  - no problem for syntax and binary encoding
  - the instruction semantics are described by C-like code (>90% is real C code)
- Method:
  - new extraction code, new parser (based on G.C.Necula parser)
  - same OCaml internal representations (same ASTs, same coding tables, etc)

Conclusion

- Development cost
  - developing the generation framework is *likely* longer than developing one simple ISS
  - Refactoring the ISS is a lot faster if it has been generated from the presented framework
  - Adding one new output is a lot easier with this framework
- Documentation uses “**pseudo**-code”. Why not “code”? Using code:
  - Easier to generate ISS, tests, etc
  - Allow to validate the documentation
- The generated code is distributed (in SimSoC 0.7.1)
  - the generator will be.