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

Frédéric Blanqui, Claude Helmstetter[1], Vania Joloboff, Jean-François Monin[2], and Shi Xiaomu[3]

SimSoC-CERT project, **FORMES** team of the **LIAMA** laboratory (**INRIA**-**CNRS**), located at **Tsinghua** University

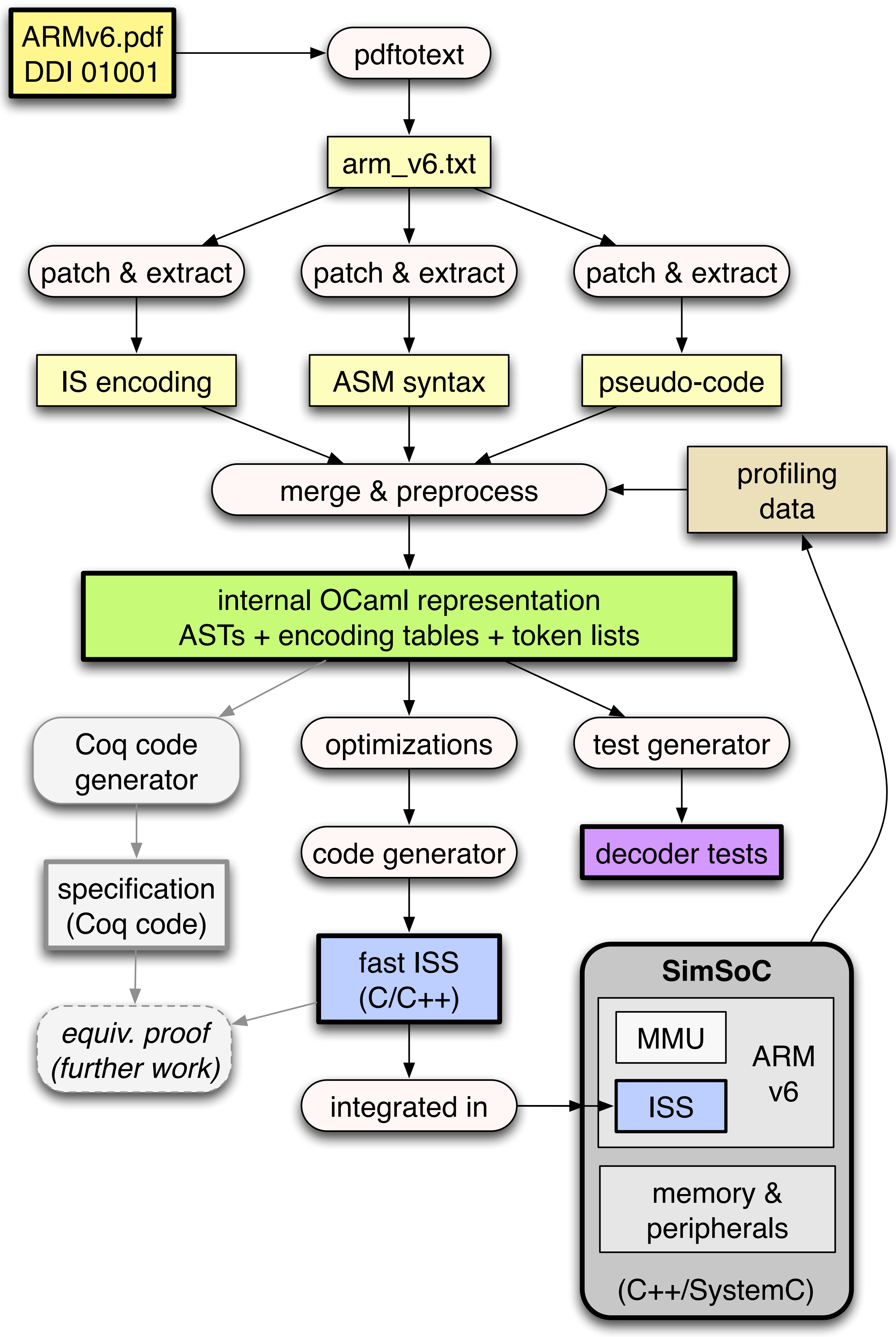
[1]claude.helmstetter@inria.fr, [2]jean-francois.monin@imag.fr, [3]xshi0811@gmail.com

Introduction

context: Simulation of embedded systems, using SystemC / TLM-LT (loosely timed) models

objective: Shorten the development time of **fast** CPU simulators (ISS, Instruction Set Simulator)

main idea: Extraction of pseudo-formal data from the architecture reference manual
+ **optimizations** on the intermediate representation + **code generation**



Results

Lines of code:
Generator: 5400 LoC, Generated code: 74,000 Loc
+ 4000 LoC for the ISS environment (hand-written)

Functional validation:

- test generator: 1 serious bug + 2 minor bugs found & fixed
- basic tests: same result than our previous ARMv5 hand-written simulator
- 2 boards are complete enough to boot Linux

Performances: (compared to hand-written ISS)

- Linux 64: 107 Mips vs. 103 Mips (+4.1%)
- Linux 32: 78 Mips vs. 84 Mips (–6.8%)
- MacOS 64: 92 Mips vs. 88 Mips (+4.5%)

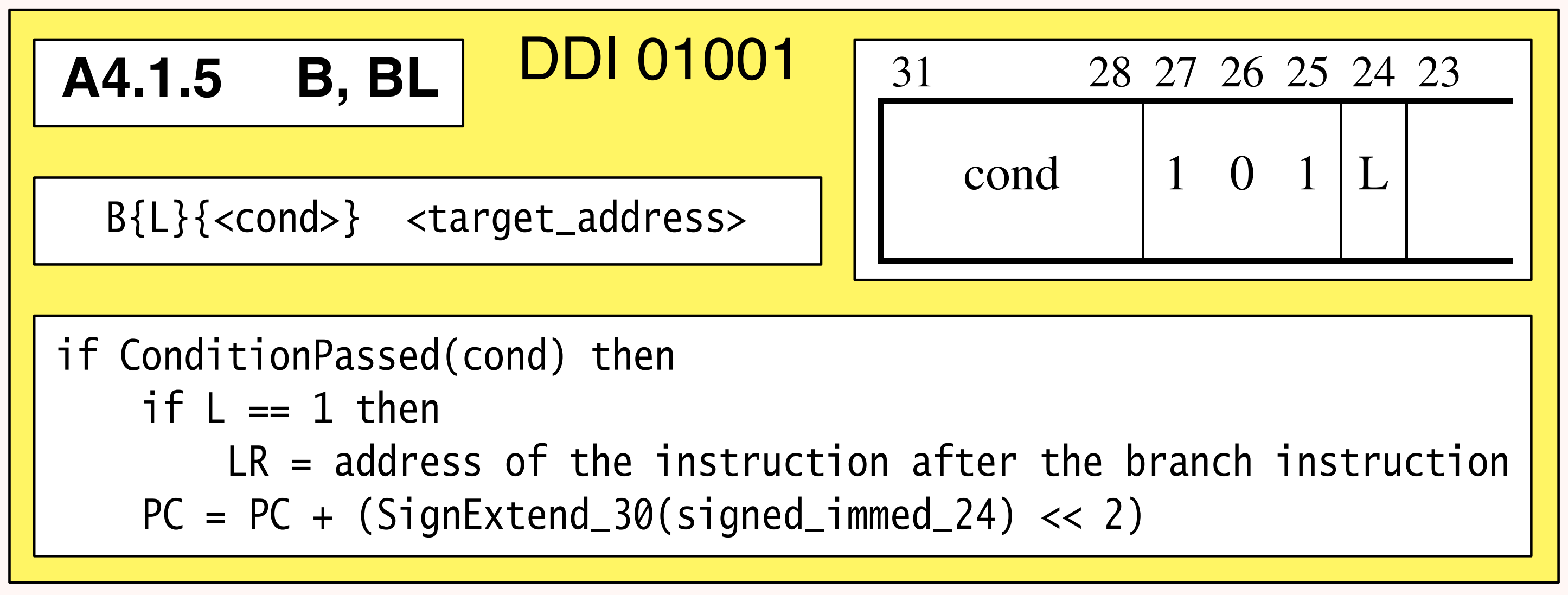
References

- **SystemC 2.2.0 and OSCI TLM-2.0.1:** cf. http://www.systemc.org/
- **SimSoC (software):** cf. http://gforge.inria.fr/projects/simsoc/
- **SimSoC (paper):** Claude Helmstetter, Vania Joloboff, Hui Xiao. *SimSoC: A full system simulation software for embedded systems.* In IEEE, editor, OSSC'09, 2009.

Transformation Flow

1. Extraction of the pseudo-formal sections

header
+
syntax
+
encoding
+
pseudo-code



A4.1.5 B, BL

DDI 01001

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

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

```
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)
```

2. Normalization and optimizations

- Some instructions are described in two parts: body + operand. The operand is described in the same way than an instruction. Generating efficient code requires to **flatten** the description by inlining the pseudo-code of the operand and merging the encoding tables.
- Most ARM instructions have a condition field, which is evaluated before executing the instruction itself. However, the condition is often *"always"*,1 thus the condition check is useless. For each conditional instruction, we derive an unconditional variant by **specialization**.
- As an instruction is generally decoded once and executed a lot of times, sub-expressions that depend only upon the instruction parameters are moved from the execute function to the decoder. Thus, the sub-expression `NumberOfSetBitsIn(register_list)*4` is replaced by a new parameter `nb_reg_x4`, which is **pre-computed** at decode-time.
- Compiling to native code requires to recognize the *basic blocks* (i.e., a sequence of instructions always executed in a row). An instruction is a basic-clock terminator if it may branch for some states of the processor. Some particular instructions are managed manually, but the **may-branch condition** (such as `Rd==PC`) is computed automatically for most instructions.
- etc

3. Code generation

Many modules are generated:

- a C **type** used to store instructions
- 2 **decoders**: ARM32 + Thumb instruction sets
- 1 **semantics function** per instruction
- the **`may_branch`** function that detects basic block terminators
- ASM **printers**, used to print debug traces.

Conclusion

- We successfully obtained a **stable and fast CPU simulator**, using automatic extraction + automatic transformation + code generation
- The ARMv6 specification (DDI 01001) contains **ambiguities** and **bugs**. Those are fixed in the new ARMv7 reference manual.
- Code **refactoring** (e.g., to try a new optimization) is clearly **easier** using this approach, compared to a hand-written ISS.
- Part of the generator can be **reused** for other architectures. We have started the generation of an SH4 CPU simulator.
- Other back-ends: Coq specification (for formal proofs), ARM to LLVM translator (for dynamic compilation and faster simulation)