LIAMA

Challenges and Results in Virtual Prototyping

Vania Joloboff

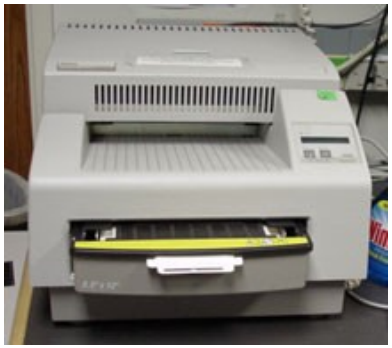
INRIA

Also Senior Visiting Scientist at Shenzhen Institute of Advanced Technologies

LIAMA

- LIAMA is the Sino French Laboratory of Informatics, Applied Mathematics and Automation
  - Established initially in 1997 as a single place laboratory at the Chinese Academy of Sciences Institute of Automation
- LIAMA is now a distributed laboratory with many partners
  - France: INRIA, CNRS, Ecoles Centrales, U. Grenoble
  - Europe: U. of Brussels, U. of Wagenigen
  - China: Beida, Beihang, Tsinghua and CAS institutes: CASIA, ISCAS and SIAT
- The FORMES project is a joint project between INRIA, CNRS, Tsinghua University and CAS SIAT
  - With contribution from Beihang and Harbin Eng. University

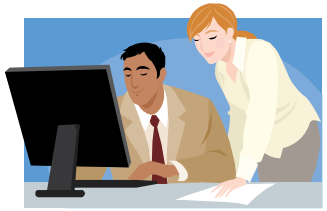
Embedded Systems



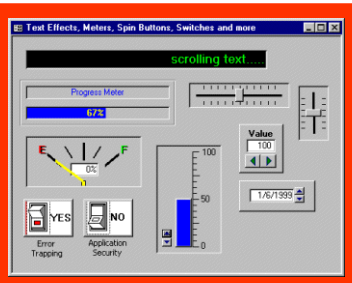
An industrial embedded systems product is a set of hardware components running dedicated application software.

It should have no bugs and adequate performance

Towards Full Virtual Prototyping



Software engineers design software models



Model design and verification

Hardware engineers design hardware models

Verification Tools

Application Software

Code generator

Generated code

Platform integration (RTOS, application specific libraries)

Hardware Models

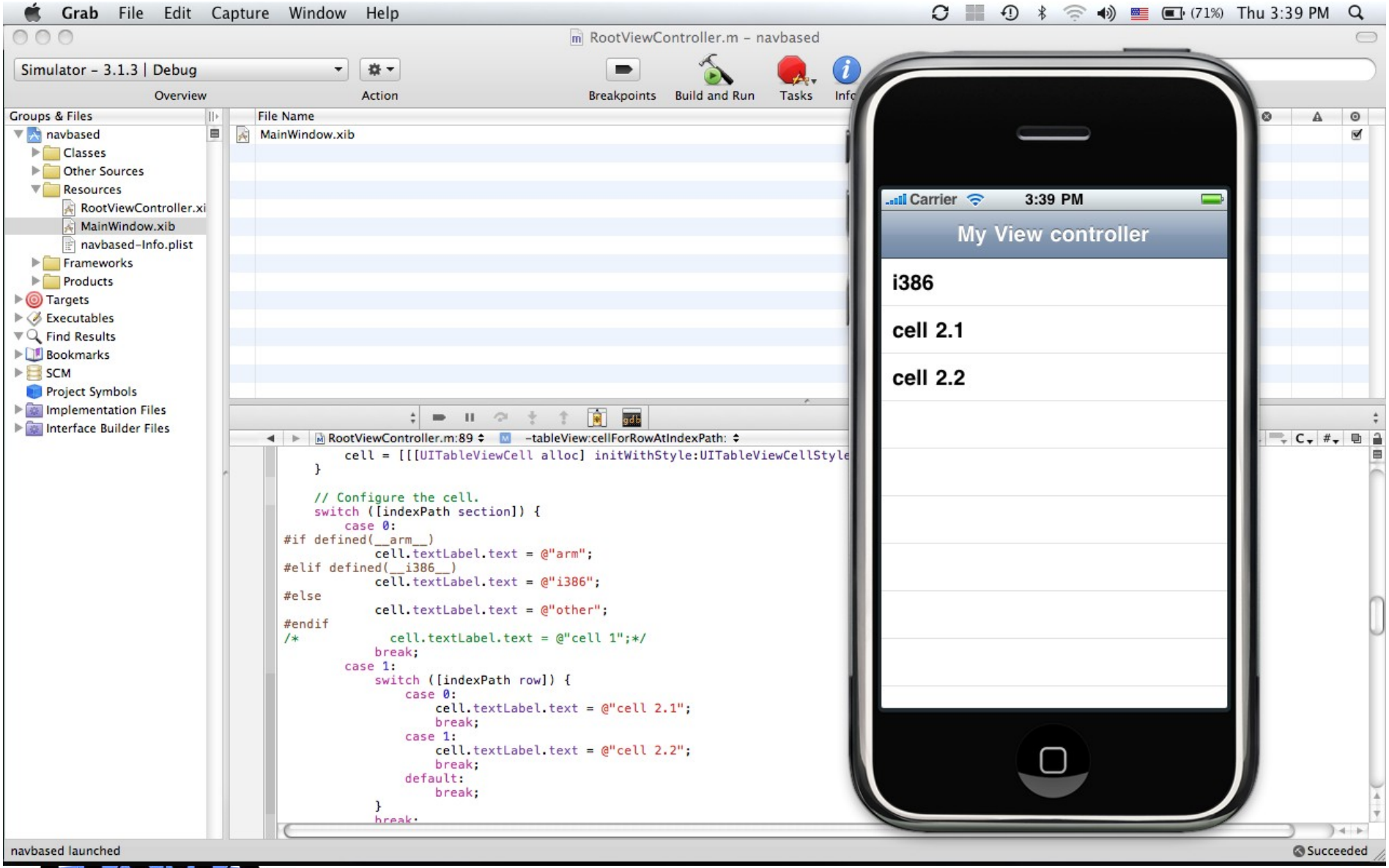
Virtual Prototype

Real System Prototype

Virtual Prototyping Platform to run the application software and validate hardware functionality and performance

LIAMA

Virtual Prototyping example: i-Phone



Virtual Prototyping with Full System Simulation

Build an executable model of the embedded system electronics (the virtual prototype) and run the application software on top of this virtual hardware

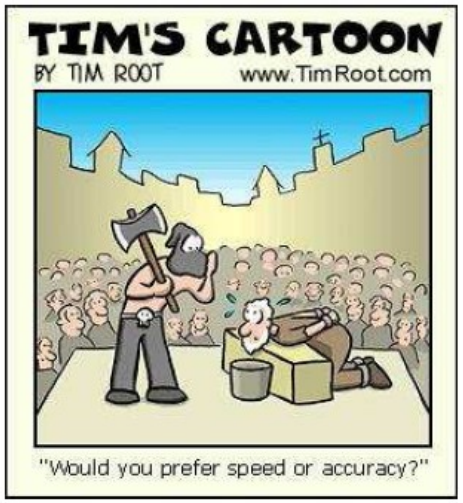
What is the appropriate technique to achieve virtual prototyping?

From the software point of view: Hardware simulation must be fast enough to run the programs in a few minutes, possibly seconds, not hours

From the hardware point of view: Performance prediction and power consumption

From both: *The simulation must produce the same results as the real hardware*

Today, this is a dream but we are making progress towards the goal



Virtual Prototyping Research

- New architectures, new chips
  - Today, we simulate ARM and PowerPC and MIPS
  - Started the SH simulator
  - Support for the new variable length encoding for Power
  - We need many students on this topic…
- Fast Simulation
  - Continue to improve our simulation speed
    - Explore parallel simulation
- Certified Simulation
  - Prove simulation is correct
- Approximately Timed
  - Provide performance estimate of the simulated hardware

LIAMA

Very Fast Simulation

Execute only a few host machine instructions for each application software simulated instruction

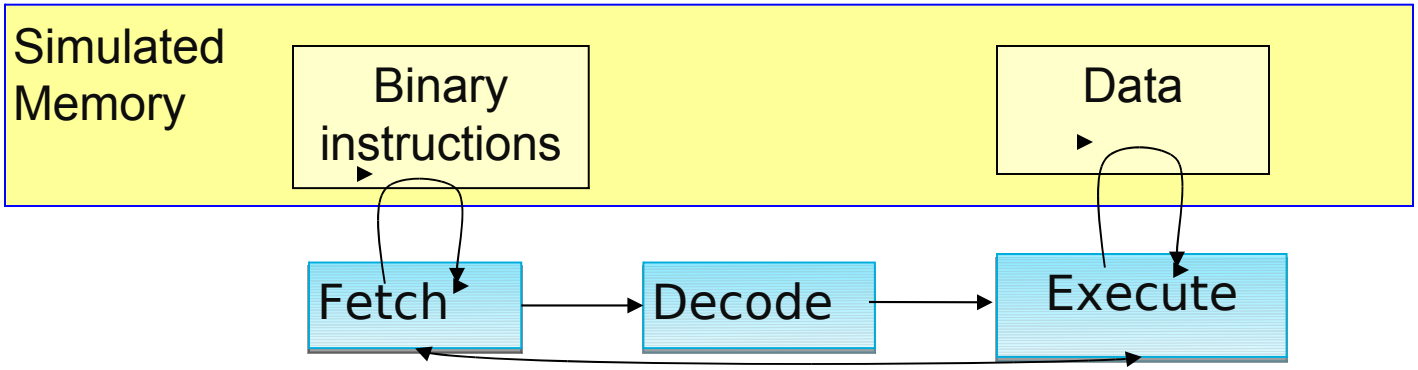
Parallelize multi-core simulation

Interpreted Instruction Set Simulation (ISS)

Early simulation: **Interpreted Simulation**

- Simulate the instruction fetch/decode/execute of the target processor
- Simulator code does essentially

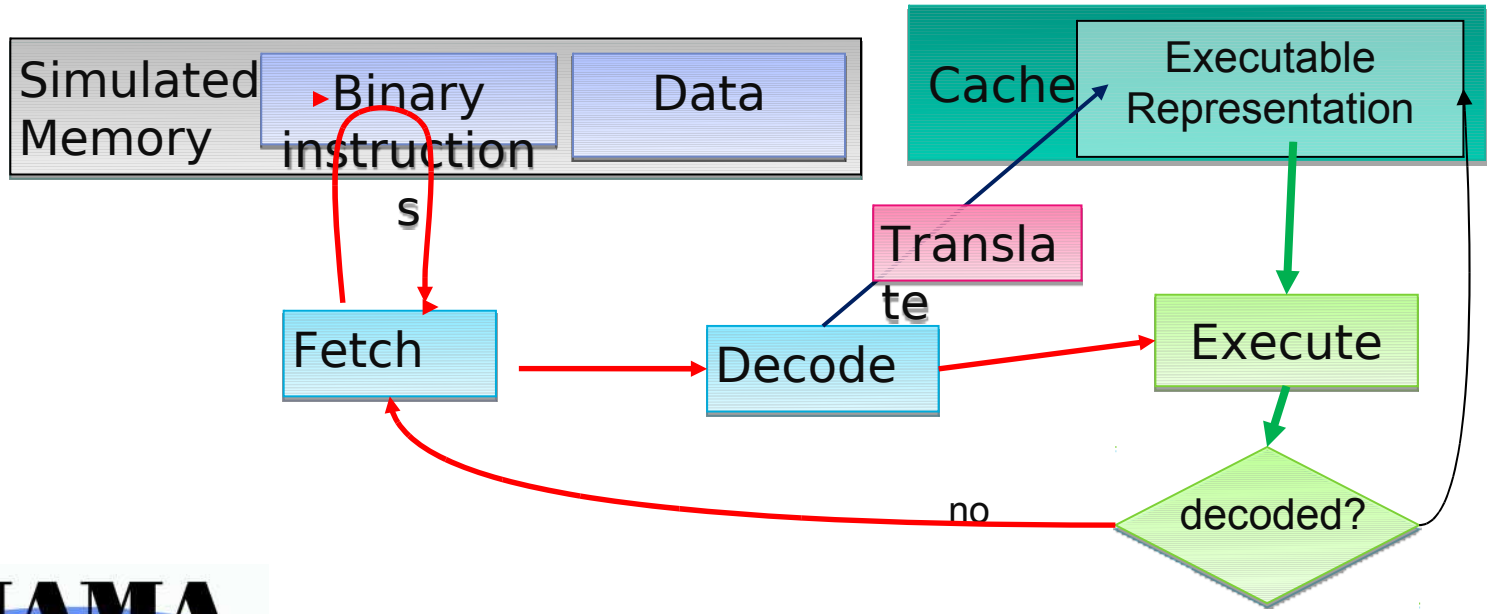
```
do {
instruction = Fetch (current_pc); // result: 01110011001100011…
Decode (instruction); // result: "this is an addition instruction"
Execute (instruction); // result: the operands are added
} until End Of Program
```



Inefficiency due to decode multiple times the same instructions : speed < 10 Mips

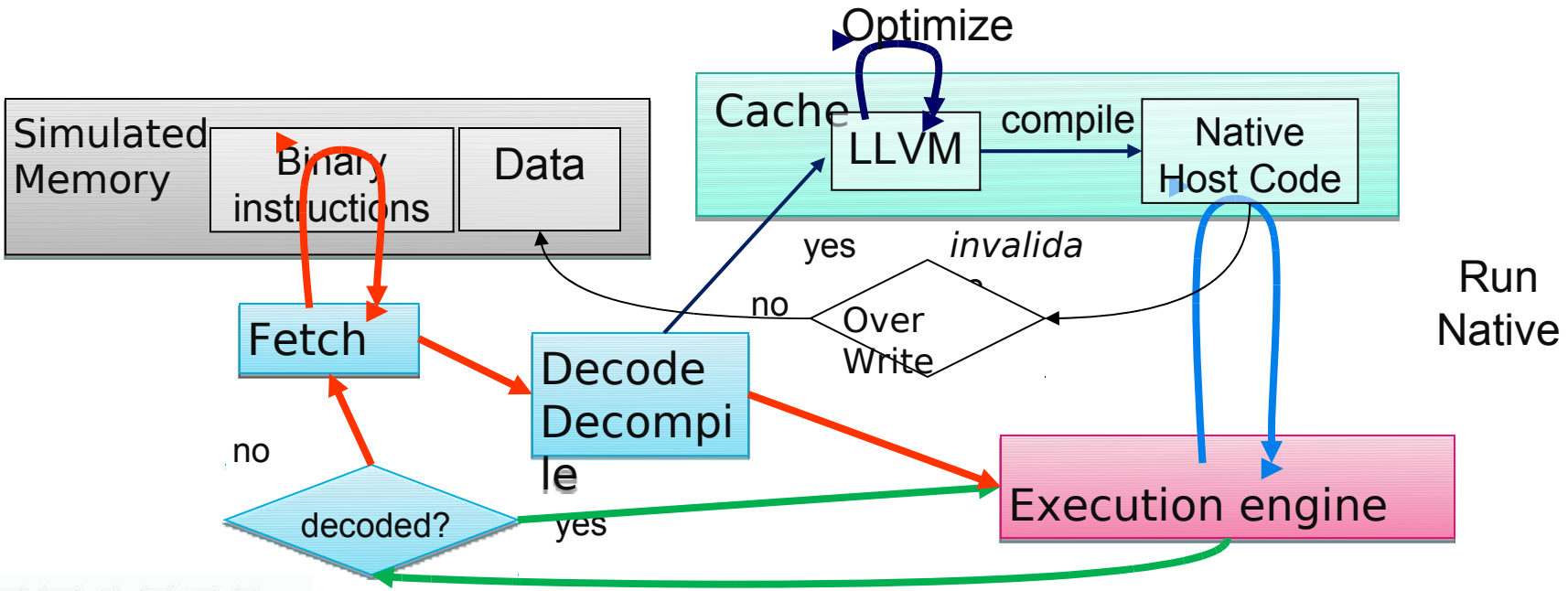
How to do better ?

- Technique: Dynamic binary translation
  - Decode Only Once: The simulated binary proam (typically the operating system binary, eg. Linux kernel) is dynamically translated into another representation run on the simulation host
    - Eliminate most of the decode time, speed up the execute time
  - Cache the translated code for re-use (optimize)
  - Translation can be done on segment or page basis
- Speed increases significantly > 15 Mips



High Speed Simulation

- Dramatically improve simulation speed using most recent compiling technologies: dynamically translate simulated binary code into optimized host code
- The machine code is first decompiled into a Control Flow Graph, translated first in some *Intermediate Language (*LLVM from UIUC). , *then optimized,* then recompiled into host machine code and executed under control of execution engine



Compiling speed

- The compiling speed becomes an issue
  - If it takes time T_i to execute an instruction in interpreted mode, and time C to compile, resulting in code whose execution takes time T_e, then it is only worth compiling when the instruction is executed more than N times such that
    - $N * T_i > C + N * T_e \quad => \quad C < (T_i - T_e) / N$
- Only frequently executed instructions are worth compiling, those over some threshold N
  - Value of N depends on compilation speed, in our case about 1000 instructions per second
- We always start simulation in interpreted mode, run dynamic profiling and then selectively compile “hot” basic blocks

Results

- Our progress

Speed in Mips	2007	2008	2009	2011
ARM32	6.62	15	75	120

- Can we do better ?
  - Yes, but doing compilation on a separate processor, parallelizing dynamic translation and execution
  - Yes, by compiling larger chunks than basic blocks

Certified Simulation

Ideal: Certify that the simulator behaves exactly as the real hardware

Assumption: there exists a formal specification of the HW, which may not be available from the vendors (e.g ARM, IBM, Intel…) but that can be developed or extracted from vendor's specifications.

HW Formal specs (in Coq)

Goal: Prove

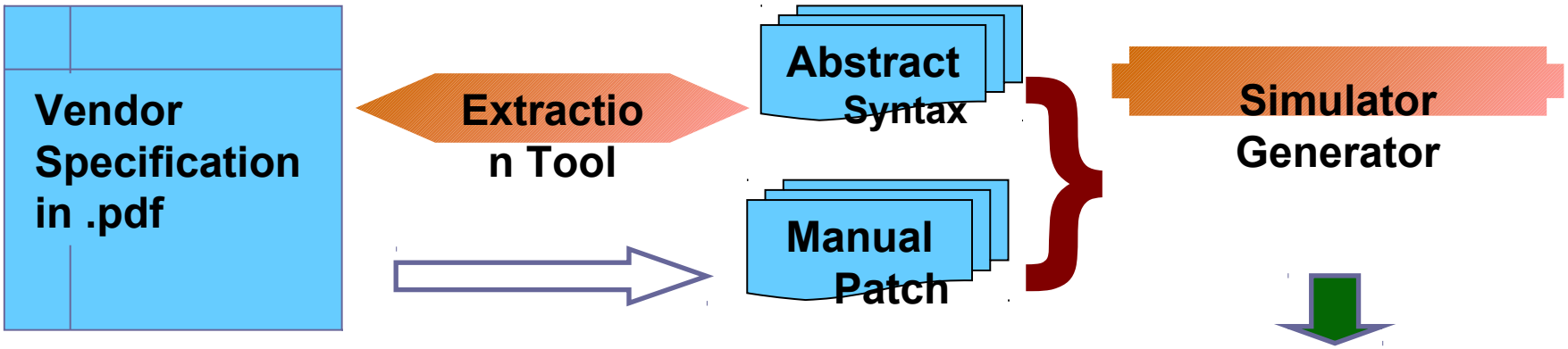
(with a theorem prover) that the C program implements the specs

Simulator program coded in C

Fortunately, the C semantics in Coq have been developed by the Compcert C program

We can re-use of lot of Compcert-C compiler code to develop the simulator proof

Automated, Certified Simulation



execi(M,S) -> M', S'

Prove that the C code for each instruction ends up with the same result as the formal definition

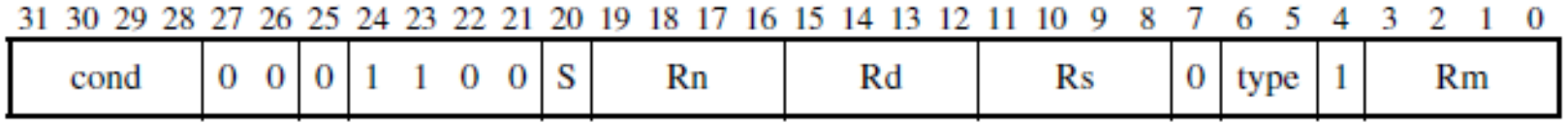
Coq Proof

```
Add(P) {
int v = p.reg[d] += p.reg[src];
if (v > 0) p.neg = true else if (v>0)
      p.pos=true else p.zero=true;
}
```

LIAMA

Example of .pdf for ORR instruction

Decoding info



31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 0 0	S	Rn	Rd	Rs	0	type	1	Rm

Semantics info

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;

if ConditionPassed() then
    EncodingSpecificOperations();
    Shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
      APSR.N = result<31>; APSR.Z = IsZeroBit(result); APSR.C = carry;
```

Results

- We have completely generated the ARM V6 simulator from the .pdf
  - It runs at 95% the speed of the manually generated
  - We found bugs in the documentation that created bugs in the simulator (reported to ARM)
  - Need some manual complement because the specification is not enough strongly typed, or there are english sentences
  - Strongly tested, runs a Linux platform
- We have completed a formal spec of ARM instruction set
- We have now complete proof for one instruction (100 more to go…)
- We have completely generated the SH instruction set
  - Not tested yet, but proof of concept we can generate two simulators for two architectures from the same abstract syntax

Approximately Timed Simulation

- Ideal: At the end of the simulation, the simulator reports exactly how many clock cycles have elapsed to run the software
- Cycle accurate simulators are extremely slow: unusable for virtual prototyping.
- Reminder: the modern processors are designed to execute at least 1 instruction per cycle (sometimes more) with architecture support (caches, pipe line, etc). If they don't, it's because there is a blocking factor…
- Idea: simulate enough of the system with a model to compute the blocking factors and evaluate the delays with approximation, without really simulating the HW
- Expectation : to get 90% of accuracy with >10 times the speed of a CA simulator

Approximately Timed Simulation

- Example
  - Processors have an instruction cache and instruction buffer with a complex pre-fetch process. One can approximate the pre-fetch by calculating cache misses and resulting delays with abstract simulation of the cache, the bus and memory.
  - Processors have data cache. May be possibility of fast calculation of the cache miss with a different algorithm than the HW
- Under development
  - An abstract cache simulator and an abstract pipe line to evaluate the delays created

We are recruiting intern students

- Motivated students
- Reasonably good english : reading, speaking, writing
- Computer science background: we are looking for students having *at least one* of these competences
  - Real time systems, process control, concurrent // computing
  - Modeling language experience: UML, SystemC
  - Good object oriented C++ programming
  - Compiler and operating systems, code generation
  - Networking protocols: Ethernet, TCP/IP
- Experience with LINUX and handling software with sophisticated control tools: subversion (svn), autoconf, automake, make, etc. is a plus
- Write to vania.joloboff@inria.fr

LIAMA

谢谢，
Thank You,
Merci