What You Synthesize is What You Simulate: Design of a RISC-V Core from C++ Specification

Simon Rokicki, Joseph Paturel, Davide Pala, Olivier Sentieys

Univ. Rennes, Inria, IRISA
Traditional Processor Design Flow

- Maintain two coherent models:
  - RTL and simulation (ISS) models
What You Simulate is What You Synthesize

• Traditional Processor Design Flow
  – Maintain two coherent models:
    • RTL and simulation (ISS) models

• Proposed Flow
  – Design the processor as well as its software validation flow from a single high-level model
Advantages and Challenges

Advantages
- Improves readability, productivity, maintainability, and flexibility of the design
- Object-Oriented processor model can be easily modified, expanded and verified

Challenges
- How to specify core (and uncore) components, cache memory hierarchy, synchronization, etc.?
- How to specify parallel computing pipelines using HLS?
Comet

Is HLS suitable for such a complex processor design?

- This work describes Comet
  - 32-bit RISC-V instruction set RV32IM
  - In-order 5-stage pipeline micro-architecture

- Designed from a single C++ specification using High-Level Synthesis (HLS)

https://gitlab.inria.fr/srokicki/Comet
Synthesizing from an Instruction Set Simulator

- Main loop is pipelined
- Inter-iteration dependencies:
  - register file read/write dependencies
  - determining PC value
- Cycles per Instruction $\approx II = 3$

Need to explicit the pipeline and the stall/forwarding logic!
Explicitly Pipelined Simulator (1/2)

- Pipelined stages are explicit
- Pipeline registers are variables
- One iteration is the execution of each stage
- Main loop is pipelined (II=1)

- Explicit stall mechanism
- Explicit forwarding
Explicitly Pipelined Simulator (2/2)

```
struct FtoDC ftdc;
struct DCtoEx dctoe;
struct ExtoMem extomem;
struct MemtoWB memtowb;

while true do
    ftdc_temp = fetch();
    dctoe_temp = decode(ftdc);
    extomem_temp = execute(dctoe);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    bool forward = forwardLogic();
    bool stall = stallLogic();
    if !stall then
        ftdc = ftdc_temp;
        dctoe = dctoe_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
    if forward then
        dctoe.value1 = extomem.result;
    end
end
```
Dealing with Multi-Cycle Operators

- Multi-cycle operators combine state machine & execution logic
- State machine *encoded* in the C code using a switch/case

- Used for:
  - Division
  - Caches
  - FPU (work in progress)
Design and Validation Flow

Simulation performance
- 26 Millions cycles per sec.
- MiBench
- 8th-gen. Intel core i7

What about quality of the hardware?
Synthesis Results

- Comparison of Comet against similar implementations
- Target technology is STMicro 28nm FDSOI
- All cores are configured for rv32i

Area and Frequency Results for Different RISC-V Cores.

<table>
<thead>
<tr>
<th>Core</th>
<th>Language</th>
<th>Frequency Target (MHz)</th>
<th>Area ($\mu m^2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comet [1]</td>
<td>C++</td>
<td>700</td>
<td>8 476</td>
</tr>
<tr>
<td>PicoRV32 [3]</td>
<td>Verilog</td>
<td></td>
<td>7 830</td>
</tr>
</tbody>
</table>
Simulator Class

- Manages execution of the program and controls the non-synthesisable aspects of the simulation
  - reading binary file, emulating the system-calls, etc.

- Simulator object has full control over the core state
  - can access PC, register file, pipeline regs, etc.

```python
class Simulator:
    def run(self):
        exitFlag = False
        while not exitFlag:
            doCycle(core, 0)  # CPU cycle (HLS HW)
            solveSyscall()  # Emulated syscalls
            extend()  # Emulated HW extension
            printCycle()  # Per-cycle print/log
            printStat()  # End-of-execution print stats.
```
Discussions on the Proposed Paradigm

• Pros
  – Debugging is done at C++ level
  – Fast simulation using the C++ simulator (~20.10^6 cycles per second)
  – Simulator is equivalent to RTL model
  – Modifying the core is simpler than at HDL level
  – Software development techniques (e.g., continuous integration)

• Limitations
  – Some features are difficult to describe (e.g. multi-cycle operators)
  – Pipeline has to be explicit
  – HLS tools may have trouble synthesizing multi-core systems
  – Late modifications (e.g. gate/metal fix)
Conclusion & Roadmap

• Efficient processor core design (HW µarch + SW simulator) from a single C++ code

• Current projects
  – VLIW Dynamic Binary Translation, Non-Volatile Processor, Fault-Tolerant Multicore, etc.

• Perspectives
  – Automatic source-to-source transformations for HLS
    • From ISS-like specification to HLS-optimized C code
  – Support for floating point extension (Q3 2019)
  – Multi-core system with cache coherency (Q4 2019)
  – Many-core system with NOC (2020)
Thank you for your attention!

https://gitlab.inria.fr/srokicki/Comet