What You Simulate is What You Synthesize: Designing a Processor Core from C++ Specifications

Simon Rokicki, Joseph Paturel, Davide Pala, Olivier Sentieux

Univ. Rennes, Inria, IRISA
• Traditional Processor Design Flow
  – Maintain two coherent models:
    • RTL and simulation (ISS) models
• 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**
Is HLS suitable for designing processors?

• The answer is yes but....
Advantages and Challenges

Advantages

• Improves readability, productivity, maintainability, and flexibility of the design

• Object-Oriented processor model can easily be modified, expanded and verified

Challenges

• How to specify core/un-core components, cache memory hierarchy, synchronization, etc.?

• How to specify parallel computing pipelines using HLS?
This work describes Comet
– 32-bit RISC-V instruction set
– In-order 5-stage pipeline micro-architecture

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

Is HLS suitable for processor design?

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 ≈ 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)

```c
struct FtoDC ftdc;
struct DCtoEx dctoex;
struct ExtoMem extomem;
struct MemtoWB memtowb;

while true do
    /* --- Execute the stages --- */
    ftdc_temp = fetch();
    dctoex_temp = decode(ftdc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    /* --- Commit the registers --- */
    ftdc = ftdc_temp;
    dctoex = dctoex_temp;
    extomem = extomem_temp;
    memtowb = memtowb_temp;
end
```
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 ($\Pi = 1$)
- Explicit stall mechanism
 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)
  
  ```
  while true do
      ftdc_temp = fetch();
      dctoex_temp = decode(ftdc);
      extomem_temp = execute(dctoex);
      memtowb_temp = memory(extomem);
      writeback(memtowb);
      bool forward = forwardLogic();
      bool stall = stallLogic();
      if !stall then
          ftdc = ftdc_temp;
          dctoex = dctoex_temp;
          extomem = extomem_temp;
          memtowb = memtowb_temp;
      end
      if forward then
          dctoex.value1 = extomem.result;
      end
  end
  ```

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

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

while true do
    ftdc_temp = fetch();
    dctoex_temp = decode(ftdc);
    extomem_temp = execute(dctoex);
    memtowb_temp = memory(extomem);
    writeback(memtowb);
    bool forward = forwardLogic();
    bool stall = stallLogic();
    if !stall then
        ftdc = ftdc_temp;
        dctoex = dctoex_temp;
        extomem = extomem_temp;
        memtowb = memtowb_temp;
    end
    if forward then
        dctoex.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
Comet Simulation Environment

Not Synthesizable
- Instrumentation
- Syscall emulation
- Elf reader
- Not-yet-synthesized extensions

Synthesizable

- A C++ simulator handles all features that cannot be synthesized
- Two ways of handling syscalls
  - Syscall emulation
  - Through an OS
(Ongoing work) Handling interrupts

- Support for CSR registers
- Ecall instruction fire an interrupt
- When external interrupt (if not masked)
  - Disable fetched instruction
  - Save current PC
  - Jump to ISR

- Not synthesized yet
- Simulator can boot RT os
Simulation performance
- 18 Millions cycles per sec.
- MiBench
- 8th-gen. Intel core i7

What about quality of the hardware?
Experimental Study - ASIC

- Comparison of Comet against similar implementations
- Target technology is STMicro 28nm FDSOI

<table>
<thead>
<tr>
<th>Core</th>
<th>ISA</th>
<th>freq. (MHz)</th>
<th>Area ($\mu m^2$)</th>
<th>Lang.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comet [14]</td>
<td>rv32i</td>
<td>700</td>
<td>8168</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td>rv32im</td>
<td></td>
<td>11099</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rv32imf</td>
<td></td>
<td>26760</td>
<td></td>
</tr>
<tr>
<td>Rocket [12]</td>
<td>rv32i</td>
<td></td>
<td>11114</td>
<td>Chisel</td>
</tr>
<tr>
<td></td>
<td>rv32im</td>
<td></td>
<td>12606</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rv32imf</td>
<td></td>
<td>26550</td>
<td></td>
</tr>
<tr>
<td>PicoRV [15]</td>
<td>rv32i</td>
<td></td>
<td>7747</td>
<td>Verilog</td>
</tr>
<tr>
<td></td>
<td>rv32im</td>
<td></td>
<td>11176</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design</th>
<th>Area ($\mu m^2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU</td>
<td>8147</td>
</tr>
<tr>
<td>Core w/o FPU</td>
<td>15299</td>
</tr>
<tr>
<td>Core w/ FPU</td>
<td>26760</td>
</tr>
</tbody>
</table>
Experimental Study - FPGA

- Comparison of Comet against similar implementations
- Target technology is Xilinx Artix 7 (Catapult + Vivado 2018.3)

<table>
<thead>
<tr>
<th>Core</th>
<th>ISA</th>
<th>freq. (MHz)</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LUT</td>
</tr>
<tr>
<td>Comet</td>
<td>rv32i</td>
<td>80</td>
<td>2032</td>
</tr>
<tr>
<td></td>
<td>rv32im</td>
<td>70</td>
<td>2910</td>
</tr>
<tr>
<td></td>
<td>rv32imf</td>
<td>74</td>
<td>6460</td>
</tr>
<tr>
<td>Rocket</td>
<td>rv32i</td>
<td>76</td>
<td>2253</td>
</tr>
<tr>
<td></td>
<td>rv32im</td>
<td></td>
<td>2570</td>
</tr>
<tr>
<td></td>
<td>rv32imf</td>
<td></td>
<td>8132</td>
</tr>
<tr>
<td>PicoRV</td>
<td>rv32i</td>
<td>140</td>
<td>880</td>
</tr>
<tr>
<td></td>
<td>rv32im</td>
<td>110</td>
<td>1977</td>
</tr>
</tbody>
</table>
Pros/Cons of Proposed Paradigm

• Pros
  – Debugging is done at C++ level
  – Fast simulation using the C++ simulator (\(\sim 20.10^6\) 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)

• Cons
  – 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. 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

• Perspectives
  – Dedicated source-to-source transformations for HLS
  – 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
Tracing – Code Instrumentation

• Instrumenting Comet simulator is simple
• Overload function `everyCycle()` from the simulator
  – Example: counting stall caused by cache miss
Adding Custom Instruction

- Custom FFT accelerator as a custom instruction
- Overview of FFT algorithm
Adding Custom Instruction

• Custom FFT accelerator as a custom instruction
• Overview of FFT algorithm

Butterfly computation
Adding Custom Instruction

• What can custom instructions do?
  – Read two registers
  – Write one result
  – Can stall if needed
  – Can encode internal state machine

• Custom instruction limitations
  – Cannot access memory
  – Cannot use more than 2R/1W
Adding Custom Instruction

• Butterfly computation:
  – Reads two complex values
    • Two 16-bit fixed-point values in a 32-bit register
  – Writes two complex values
    • Main custom instruction computes the two values and output first value
    • Second custom instruction output the second value
  – Uses two twiddle factors
    • Index encoded in immediate field
    • Use of internal state machine to select the one to use
Adding Custom Instruction

• Results:
  – Around 50 lines of C code in the ALU
  – Execution of FFT is 14x faster
  – Core area increased by 31% (baseline is rv32im)