Generating gates from C: a test with PPSi

Fabrizio Ferrandi - Pietro Fezzardi

Politecnico di Milano
Dipartimento di Elettronica ed Informazione
fabrizio.ferrandi@elet.polimi.it pietrofezzardi@gmail.com
Outline

- PandA framework
- High-level synthesis with bambu
- PPSi test
High-Level Synthesis

- Design and implementation of a digital circuit starting from a behavioral representation

- Different state-of-art algorithms have been implemented
  - Function allocation and sharing
  - Memory allocation
  - Bit-width analysis
  - Module allocation and binding
  - Register allocation and binding
  - Interconnection allocation and binding
  - Controller and Datapath generation

- The output is a synthesizable code in a common hardware description language (e.g., Verilog, VHDL)
General features of bambu

- Open source project: GPLv3 license
- ANSI C support
  - Complete with few exceptions: recursive functions
- Support of single and double precision floating point computation
- Unaligned memory accesses and dynamic pointers resolution
- GCC compilers supported: v4.5, v4.6, v4.7, v4.8 and v4.9
  - All gcc options can be passed to bambu
- Support of GCC vectorization
- Linux distributions supported:
  - Ubuntu 12.04LTS, 13.04, 13.10, 14.04LTS
  - Debian 7.3 (Wheezy), unstable (Sid)
  - CentOS/Scientific Linux
  - ArchLinux
Functional resources

- Resource library composed of about 320 components
- 155 components are actually templates:
  - Parametrizable with respect to the operation data sizes (8, 16, 32, 64)
- Several builtins are available:
  - `exit, abort, printf, puts, putchar` for debugging purposes
  - `abs, memcpy, memset, memcmp, malloc, free`
  - `libm functions: sinf, cosf, cbrt, acosh, ...`
- `Libm functions, malloc` and `free` supported through a C based library
- Floating point modules generated by exploiting FloPoCo library or the SoftFloat library
- Supported pipelined, multicycling and unbounded functional resources
- Different memory models
- Resource library described in XML or in C
  - Easily extendible
Accelerator verification

- bambu HLS tool is able to pretty-print the IR as C code
- Direct execution of a program based on this code could be used to produce expected functional values
- Simulation of the HDL description of the accelerator is done comparing the expected results produced by the C code and the values obtained by the hardware simulation
  - Testbench generation done automatically
  - Testbench generation could be customized through XML files
- Hardware simulators currently supported
  - ISIM: Xilinx Simulator
  - XSIM: Xilinx Simulator
  - Modelsim from Mentor
Regression tests currently used

  - DFADD, DFMUL, DFDIV, DFSIN: Double-precision floating-point
  - MIPS: Simplified MIPS processor
  - ADPCM: Adaptive differential pulse code modulation decoder and encoder
  - GSM: Linear predictive coding analysis of global system for mobile communications
  - JPEG: JPEG image decompression
  - MOTION: Motion vector decoding of the MPEG-2
  - AES: Advanced encryption standard
  - BLOWFISH: Data encryption standard
  - SHA: Secure hash algorithm

- Subset of GCC regression suite: 800
Accelerators synthesis

- Automatic generation of synthesis scripts based on XML configuration for different tool flows:
  - FPGA:
    - Xilinx ISE
    - Xilinx VIVADO
    - Altera Quartus
    - Lattice Diamond
  - ASIC
    - Synopsys Design Compiler
PPSi test
THE ORIGINAL IDEA

- Try out HLS with PPSi, to see if and how it could be replaced by HW accelerators:
  - System is predictable
  - System latency is deterministic
  - Easier to design than a HDL based project

- Ignore UI and WR support to make things easier at an initial stage.

- Original C code size:
  - standard PTP state machine: 2130 lines
  - arch-wrpc: 454 lines
  - time-wrpc: 203 lines
  - headers: 1232 lines
  - Total: ~4000 lines
FIRST PROBLEMS

- As we proceeded in the work, it turned out we needed to include more and more wrpc-sw stuff to make things work properly.

- SOLUTION: use HLS on the whole wrpc-sw + ppsi.

- GOAL: produce a design for an FPGA block acting as wrpc-sw + ppsi. This would allow to build a full HW core not relying on an external processor (e.g., LM32, etc.)

- We didn't expect the resulting synthesis to be smaller than the LM32.
- The aim was to show a new possible design approach.
- There would have been time for optimization later.
THE FINAL CODEBASE

- from PPSi:
  - standard PTP state machine: 2130 lines
  - WRPTP state machine: 1537 lines
  - arch-wrpc + time-wrpc: 657 lines
  - headers: ~1200 lines
  - total: ~5500 lines

- from wrpc-sw:
  - softpll: 1407 lines
  - dev: ~3000 lines
  - headers: ~3000 lines
  - total: ~7400 lines

- wrpc-sw + PPSi = ~12900 lines of C code

- WHAT STAYS OUT: printf, diagnostics, uart, shell and any other UI. ptp-no-posix support. new sdb-lib in wrpc-sw.
WORKING WITH PANDA

- PandA does not support all the features of C language yet:
  - bitfields
  - function pointers
  - recursive functions
  - forward declarations of data structures
  - struct returned by copy

- Not available in PPSi and wrpc-sw
  - little-endian operations (abs(), noths(), ntohl(), htons(), htonl())

- PandA specifically needs definitions _in C for every function it uses

- Namely we had to add C definitions for:
  - __builtin_swap32()
  - strcpy()
  - empty stubs for irq_enable() and irq_disable()
RESULTS

- The work took two man-weeks, including fixes to bugs discovered in PandA, PPSi and wrpc-sw during the development.

- Most time was spent to find out that we had to synthesize all wrpc-sw + PPSi, then rewriting some functions in the codebase to make the code more amenable for synthesis with PandA/bambu.

- Four types of FPGA considered: SPEC, SPEC100, Xilinx Zynq, Altera CycloneV.

SPECv4 results

Slice Logic Utilization:
- Number of Slice Registers: 26782 out of 54576 49%
- Number of Slice LUTs: 45818 out of 27288 167% (*)
  Number used as Logic: 44739 out of 27288 163% (*)
  Number used as Memory: 1079 out of 6408 16%
  Number used as SRL: 1079

Slice Logic Distribution:
- Number of LUT Flip Flop pairs used: 51017
  Number with an unused Flip Flop: 24235 out of 51017 47%
  Number with an unused LUT: 5199 out of 51017 10%
  Number of fully used LUT-FF pairs: 21583 out of 51017 42%
  Number of unique control sets: 660

IO Utilization:
- Number of IOs: 194
  Number of bonded IOBs: 161 out of 296 54%

Specific Feature Utilization:
- Number of BUFG/BUFGCTRLs: 1 out of 16 6%
- Number of DSP48A1s: 54 out of 58 93%
Spec100 results

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xilinx Spartan6 xc6slx100t-3fgg484</strong></td>
<td></td>
</tr>
<tr>
<td>Area</td>
<td>42839 LUT/FF pairs</td>
</tr>
<tr>
<td>Design min period</td>
<td>11.778 ns</td>
</tr>
<tr>
<td>Design max frequency</td>
<td>84.9 MHz</td>
</tr>
<tr>
<td>Design slack</td>
<td>-3.778 ns</td>
</tr>
<tr>
<td>Registers</td>
<td>31823</td>
</tr>
<tr>
<td>DSPs</td>
<td>29</td>
</tr>
<tr>
<td>RAMs</td>
<td>27224 bytes</td>
</tr>
<tr>
<td>HLS execution Time</td>
<td>151.1 seconds</td>
</tr>
<tr>
<td>Total Execution Time</td>
<td>15588.21 seconds</td>
</tr>
</tbody>
</table>

- Xilinx ISE based synthesis, version 14.7
A different architecture: Xilinx Zynq

Zynq-7000 AP SoC Block Diagram
### Zynq results

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xilinx Zynq xc7z020-1clg484</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Area</strong></td>
<td>39272 LUT/FF pairs</td>
</tr>
<tr>
<td><strong>Design min period</strong></td>
<td>7.987 ns</td>
</tr>
<tr>
<td><strong>Design max frequency</strong></td>
<td>125.2 MHz</td>
</tr>
<tr>
<td><strong>Design slack</strong></td>
<td>0.013 ns</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>25165</td>
</tr>
<tr>
<td><strong>DSPs</strong></td>
<td>115</td>
</tr>
<tr>
<td><strong>RAMs</strong></td>
<td>27224 bytes</td>
</tr>
<tr>
<td><strong>HLS execution Time</strong></td>
<td>172.77 seconds</td>
</tr>
<tr>
<td><strong>Total Execution Time</strong></td>
<td>1774.03 seconds</td>
</tr>
</tbody>
</table>

- Xilinx VIVADO RTL based synthesis, version 2014.2
# Cyclone V results

<table>
<thead>
<tr>
<th>Altera CycloneV 5CSEMA5F31C6</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>24415 ALM</td>
</tr>
<tr>
<td>Design min period</td>
<td>8.983 ns</td>
</tr>
<tr>
<td>Design max frequency</td>
<td>111.32 MHz</td>
</tr>
<tr>
<td>Design slack</td>
<td>-0.983 ns</td>
</tr>
<tr>
<td>Registers</td>
<td>27902</td>
</tr>
<tr>
<td>DSPs</td>
<td>82</td>
</tr>
<tr>
<td>RAMs</td>
<td>27224 bytes</td>
</tr>
<tr>
<td>HLS execution Time</td>
<td>165.12 seconds</td>
</tr>
<tr>
<td>Total Execution Time</td>
<td>2145.29 seconds</td>
</tr>
</tbody>
</table>

- Quartus II based synthesis, 13.0sp1
What about HLS Commercial tool?

- It stops on this function:

```c
static int wrap_copy_out(struct sockq *q, void *src, size_t len)
{
    char *sptr = src;
    int i = len;

    TRACE_WRAP("copy_out: head %d avail %d len %d\n", q->head, q->avail, len);

    while (i--)
    {
        q->buf[q->head++] = *sptr++;
        if (q->head == NET_SKBUF_SIZE)
            q->head = 0;
    }
    return len;
}
```
THANK YOU!

GPL v3 source code available at
http://panda.dei.polimi.it