RTL Design Framework for Embedded Processor by using C++ Description

Eiji Yoshiya
Information and Communications Engineering
Tokyo Institute of Technology
Tokyo, Japan
yoshiya.e.aa@m.titech.ac.jp

Tomoya Nakanishi
Information and Communications Engineering
Tokyo Institute of Technology
Tokyo, Japan
nakanishi@vlsi.ict.e.titech.ac.jp

Tsuyoshi Isshiki
Information and Communications Engineering
Tokyo Institute of Technology
Tokyo, Japan
isshiki@ict.e.titech.ac.jp

***Abstract*—In this paper, we propose a method to directly describe the RTL structure of a pipelined RISC-V processor with cache, memory management unit (MMU) and AXI bus interface using C++ language. This processor C++ model serves as a near cycle-accurate simulation model of the RISC-V core, while our C2RTL framework *translates* the processor C++ model into cycle-accurate RTL description in Verilog-HDL and RTL-equivalent C model. Our design methodology is unique compared to other existing methodologies since both the simulation model and the RTL model are derived from the same C++ source, which greatly simplifies the design verification and optimization processes. The effectiveness of our design methodology is demonstrated on a RISC-V processor which runs Linux OS on an FPGA board as well as significantly short simulation time of the original C++ processor model and RTL-equivalent C model compared to commercial RTL simulator.**

***Index Terms*—IoT, Embedded processor, RISC-V, RTL, C++**

I. Introduction

For realizing IoT devices, it is essential to design an SoC with power efficient processors and dedicated hardware components in order to meet the tight system power budget and high performance requirements. There is an increased attention to open-source processor cores such as RISC-V [1] as a new solution to IoT devices, instead of vendor-specific processors. An effective SoC design methodology and tools are therefore crucial in developing and verifying such extensible processors with various system components, especially involving a complex software system including operating systems.

High-level synthesis (HLS) tools are mostly focusing on dataflow-oriented DSP applications [2] [3]. These HLS tools take the design written in C++ or SystemC as input, and generate RTL designs according to resource and time constraints. Recently, Rokicki et al. [4] proposed a pipelined processor design method using HLS. Despite the effectiveness of HLS tools, it is often difficult to control its cycle-level behavior directly from the software description, often requiring tool-specific pragma annotations and coding styles.

Processor design tools make it easy to design application specific instruction-set processors (ASIPs) [5] [6] [7], using proprietary description languages. These tools are capable of defining an instruction set and its microarchitectures as well as generating synthesizable RTL codes and a software tool chain. However, the capability of designing cache, MMU, and bus interfaces are often limited.

Chisel [8] extends the Scala language for introducing high-level object-oriented programming features in RTL designs which has been successfully applied to developing a series of RISC-V processors. While Chisel offers rich programming features for implementing a wide variety of circuit generators that facilitate design reuse for general digital logic designs, mastering a new design language is still a challenge for many hardware and software designers who are used to working on common HDLs and programming languages such as C/C++.

In this paper, we propose a design framework named *C2RTL* that can directly describe the *pipelined* RTL structure using C++ language. Authors' earlier works [9] which used C language, have been extended to C++ language and redesigned using LLVM compiler infrastructure [10]. The designer describes the cycle-level behavior of a logic design in C++ language in a *data flow style*, and specifies hardware attributes using GCC-style attribute keyword. This C++ data flow description serves as a simulation model of the logic design, while our C2RTL framework generates the corresponding RTL description in Verilog-HDL and *RTL-equivalent* (cycle-accurate) C model which is an order of magnitude faster than a commercial HDL simulator. Our digital system design methodology on C++ using C2RTL give an enormous advantage in design efficiency since it does not involve any proprietary languages, facilitates the use of any standard C++ programming environment, fast simulation compared to RTL, and full transparency in C++ of the entire processor components including the basic processor microarchitecture, memory subsystem and bus interface.

II. C2RTL Framework Overview

C2RTL framework parses the C++ source codes with dedicated hardware attributes (bit-width, register/memory designations, RTL generation scopes) and converts into the *internal representation* (IR). Distinct feature of the C2RTL framework is ensuring that the RTL structure intended by the designer is preserved in the generated RTL description, which is realized by the combination of the following coding constraint and a set of translation processes applied on the IR.

- **Single-cycle behavior coding constraint:** the top-level function which designates the RTL generation scope must

978-3-9819263-5-4/DATE21/©2021 EDAA

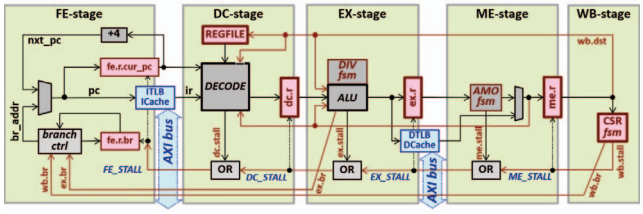

Fig. 1. Five-stage pipelined RISC-V processor.

```
#define _BW(b)          __attribute__((C2RTL_bit_width(b)))
#define _T(t)           __attribute__((C2RTL_type(t)))
#define _C2R_FUNC(n)    __attribute__((C2RTL_function(n)))
typedef unsigned char   BIT _BW(1), UINT3 _BW(3);
typedef unsigned int    UINT32 _BW(32);
typedef UINT32          ST_UINT32 _T(state);
```

```
struct BranchStatus {
  BIT active; UINT32 addr;};
struct DstStatus {
  BIT valid; UINT6 id;
  UINT32 out;};
struct FEState {
  struct Reg {
    UINT32 pc;
    BranchStatus br;
    BIT pending, ready,flush;
  } r _T(state);
};
struct DCState {
  BIT    stall, sw_mode;
  BranchStatus br;
  struct Reg {
    UINT32 pc, src[3];
    UINT6 op;
    UINT5 src_id[2], dst_id;
    ... /// other members
  } r _T(state);
};
```

```
struct CPU {
  ICache<...> icache;
  DCache<...> dcache;
  ST_UINT32 gpr[32]; // reg-file: 32bx32
  UINT32    ir;  // RV-inst word (wire)
  FEState   fe;  // FE wires & registers
  DCState   dc;  // DC wires & registers
  EXState   ex;  // EX wires & registers
  MEState   me;  // ME wires & registers
  WBState   wb;  // WB wires & registers
  .............
  _C2R_FUNC(5) /// top-level function
  int step(AXI::CH *axi_i,
           AXI::CH *axi_d, UINT32 intr) {
    fetch(axi_i);            // FE method call
    decode();                // DC method call
    execute();               // EX method call
    memory(axi_d, intr);     // ME method call
    writeback();             // WB method call
    return (halt == 1);      // HALT condition
  }
};
```

Fig. 2. Resource Structure of RISC-V Processor.

represent the single-cycle behavior of the digital system, where registers and memories are updated at most *once*.

- **Acyclic-CFG translation:** under the top-level function, all loops are completely unrolled and all function calls are recursively inlined which results in a single *control flow graph* (CFG) in a *direct acyclic graph* form.
- **CFG-to-DFG translation:** the acyclic-CFG is derived from the LLVM-IR which employs the *single static assignment form* (SSA form). Translation from the acyclic-CFG to the data flow graph (DFG) is accomplished by *def-use chain analysis* and replacing the ϕ-functions in the SSA form with multiplexers controlled by control flow flags.
- **Pipelined RTL structure extraction:** our data flow coding style with single-cycle behavior constraint enables an explicit description of the pipelined RTL structure.

III. Pipelined Processor C++ Description

Fig.1 shows the RTL structure of a RISC-V processor described by the C++ dataflow description. Supported features are RV32-IMA (integer multiplication and division, atomic) and *privileged mode* for Linux operating system. Fig.2 shows a part of the hardware resource C++ descriptions for the RTL structure in Fig.1. It consists of C++ structure declarations corresponding to general-purpose register-file, pipeline registers, pipeline control signals, cache, etc. Hardware attributes for binding C++ data flow entities with RTL entities are embedded using **__attribute__** keyword, which are summarized as follows.

- Bit-width: **C2RTL_bit_width(N)** attaches bit-width of **N**
- Register: **C2RTL_type(state)** attaches "register" attribute

```
#define FE_STALL (dc.stall|DC_STALL)
#define DC_STALL (ex.stall|EX_STALL)
#define EX_STALL (me.stall|ME_STALL)
#define ME_STALL (wb.stall)
```

```
#define FE_SW_MODE (dc.sw_mode|DC_SW_MODE)
#define DC_SW_MODE (ex.sw_mode|EX_SW_MODE)
#define EX_SW_MODE (me.sw_mode|ME_SW_MODE)
#define ME_SW_MODE (wb.sw_mode)
```

```
void CPU::fetch(AXI::CH *axi) { /// axi : icache/itlb bus interface
  BIT sw_mode = FE_SW_MODE;     /// mode switching on DC/EX/ME/WB stages
  BIT br_active  = (wb.br.active || ex.br.active || fe.r.br.active);
  UINT32 br_addr = (wb.br.active) ? wb.br.addr :
                   (ex.br.active) ? ex.br.addr : fe.r.br.addr;
  UINT32 pc = fe.r.pc;
  BIT mmu_en = fe.r.mmu_en;
  if (!FE_STALL) {  /// no stalls on DC/EX/ME/WB stages
    if (!icache.busy && !sw_mode) {
      if (!fe.r.pending) { pc = (br_active) ? br_addr : pc + 4; }
      mmu_en = wb.mmu_en;
      fe.r.mmu_en = mmu_en;
      fe.r.pc = pc;
      br_active = 0;
    }
    BIT flush = sw_mode || (icache.busy && fe.r.flush);
    fe.r.flush = flush;
    fe.r.pending = 0; fe.r.ready = !flush;
  }
  else if (icache.busy)  { fe.r.pending = 1; fe.r.ready = 0; }
  else if (fe.r.pending) { fe.r.pending = 0; fe.r.ready = 1; }
  fe.r.sw_mode   = sw_mode;
  fe.r.br.active = br_active;
  fe.r.br.addr   = br_addr;
  ir = icache.fsm(mmu_en, pc, axi, csr.satp, dc, fe);
}
```

Fig. 3. Fetch stage code

- Memory: **C2RTL_type(memory)** attaches "memory" attribute

These hardware attributes can be attached to typedef names, variable declarations, and even to the entire structure. All variables are categorized in either *register variables* (with register attribute), *memory variables* (with memory attribute), or *wire variables* (without register or memory attribute).

In Fig.2, **gpr[32]** is a 32-bit general-purpose register-file where bit-width attribute and register attribute are attached. Wire variables and register variables at each pipeline stage are organized in structures such as **FEState**, **DCState**, etc., where each contains its inner structure **Reg** with register attribute attached. Wire variables such as those in direct members of **DCState** (**stall**, **br**, and **sw_mode**) indicate the status of its pipeline stage and are synthesized as combinational signals in the RTL, while **FEState::Reg** and **DCState::Reg** members are register variables which are used to propagate information to the next pipeline stage. **ICache** and **DCache** structures contain finite state machine (FSM) resources for cache and MMU.

Our implementation of RISC-V processor consists of a standard 5 stage pipeline which is described in the top-level function **CPU::step()** annotated with **C2RTL_function(N)** attribute using **_C2R_FUNC** macro. The five calls inside **CPU::step()** corresponds to the behavior of the five stages, which are, fetch (**FE**), decode (**DC**), execute (**EX**), memory (**ME**) and writeback (**WB**) stages. Operations at each stage consists of operations on a set of input values, and producing a set of output values. Here, input values include the *pipeline registers* of the previous stage, *pipeline status signals* of the later stages, and *state registers* of the current stage. Output values include the *pipeline registers* referred in the next stage, *pipeline status signals* referred in the earlier stages, and *state registers* of the current stage.

In the **FE** stage (**CPU::fetch()** in Fig.3), **FE_STALL** macro defines the *stall events* at the later stages, and **FE_SW_MODE** macro similarly defines the *mode switching events* (between "user" and "supervisor" modes) at later stages. These two control signals and **icache.busy** signal (indicating the ICache/I-TLB status), are used to determine the condition for updating

the program counter (**fe.r.pc**). Branch address (**br_addr**) is computed at **EX** stage for branch and jump instructions and at **WB** stage for privileged instructions (such as trap return). At the end of **CPU::fetch()** function, **icache.fsm()** is called on the updated program counter for fetching the next instruction. **FE** state registers (**FEState::Reg** members) are used to control the fetch logic, and to keep track of ICache/I-TLB status as well as stall, mode switching, and branch events at later stages.

As explained here, C++ description of the processor pipeline behavior is written in an intuitive and easy to understand coding style. We call this the *data flow coding style* in which the series of operations for updating the processor states are described in normal C++ statements. Hardware attribute notations are used to construct the *RTL semantics* (registers and wires), but do not have any influence on the *software semantics*, and therefore these C++ descriptions act as a normal instruction-set simulator.

IV. Mapping Rule for Pipelined RTL Structure

The unique feature of our C2RTL framework is the direct mapping rule from the original C++ data flow description to the RTL structure. First, we define two types of value references.

- **Forward reference**: a value reference which occurs after it has been assigned inside the top-function.
- **Backward reference**: a value reference which does not occur after it has been assigned inside the top-function.

The mapping rule of the procedural operations to the RTL structure depends on the variable types, namely *register variables*, *memory variables* and *wire variables*. Next, we categorize the signal connections into the following four types, and define the mapping rules from the two value reference types.

- **Forward inter-pipeline connection**: Forward signal propagation between adjacent stages occurs via registers where there is one clock delay between the input and output. Therefore, a *forward reference* of a register variable is mapped to this forward inter-pipeline signal connection.
- **Backward intra-pipeline connection**: The *current state* value of a register is a *backward reference* of this register variable, and therefore, the statement which references the current state value of a register is mapped to the *same* pipeline stage that assigns to this register.
- **Forward intra-pipeline connection**: Combinational signals transfer values within the pipeline stage, which correspond to the *forward references* of wire variables.
- **Backward inter-pipeline connection**: Backward signals that cross pipeline boundaries are used to notify pipeline hazards and transfer data forwarding information. Here, a *backward reference* of a wire variable is mapped to this backward inter-pipeline signal connection.

With these simple and direct mapping rules, we can make use of the data flow coding style with hardware attribute annotations to *explicitly describe the intended RTL structure*.

V. Cache Memory and MMU Designs

Our RISC-V C++ model contains the cache/MMU pairs for instruction and data memory. We use a *virtually indexed, physically tagged* (VIPT) cache, and the MMU conforms to

```
template<int BW, int BI> struct TLB {
  UINT16    m_tag[N_WAY][N_LINE]  _T(memory);
  UINT32    m_pte[N_WAY][N_LINE]  _T(memory);
  UINT16    out_tag[N_WAY];       /// fetched tag (wire)
  UINT32    out_pte[N_WAY], pte;  /// fetched pte (wire)
  BIT       hit, page_fault;
  ST_UINT2  state; /// fsm state register
  ...... }; /// other members and methods

template<int BW, int BI, int BO, int BT> struct Cache {
  RV_AXI4L  axim; /// AXI-master
  UINT32    data; /// latched AXI data
  UINT32    word[N_WAY][N_LINE*N_WORD]  _T(memory);
  unsigned  vtag[N_WAY][N_LINE]  _T(memory)  _BW(2+BT);
  UINT32    out_word[N_WAY];            /// fetched word (wire)
  unsigned  out_vtag[N_WAY]  _BW(2+BT); /// fetched tag  (wire)
  ST_UINT3  state;  /// fsm state register
  BIT       busy;   /// busy status
  ST_UINT32 prv_inst; /// previous fetched instruction
  struct AddrFields { unsigned tag, idx, ofs;} adf;
  struct TagStatus { unsigned hit_way; BIT hit, dirty;} tstt;
  ...... }; /// other members and methods

template<int BW_t, int BI_t, int BW, int BI, int BO, int BT>
struct ICache : Cache<BW, BI, BO, BT> {
  TLB<BW_t, BI_t> tlb; /// tlb : a member of ICache
  ......}; /// other members and methods
```

```
#define N_WAY  (1 << BW)
#define N_LINE (1 << BI)
#define N_WORD (1 << BO)
```

```
BT : tag bits
```

Fig. 4. Instruction MMU and Cache Resource Coding

```
template <..> void Cache::check_vtag() {
  tstt.reset();
  for (int w = 0; w < N_WAY; w++) {
    if (adf.tag == (out_vtag[w] & ~DIRTY_BIT)) {
      tstt.dirty = (out_vtag[w] & DIRTY_BIT) != 0;
      tstt.hit |= 1;
      tstt.hit_way = w;
}}}
```

```
#define DIRTY_BIT  (1 << (BT + 1))
#define VALID_BIT  (1 << (BT))
```

```
template<..> UINT32 ICache::fsm(BIT mmu_en, UINT32 pc,
  AXI4L::CH *axi, unsigned ptbr, DCState &dc, FEState &fe) {
  this->extract_addr_fields(pc); /// pc => adf
  unsigned vp = this->adf.tag;
  access_axi(mmu_en, axi, pc, ptbr);
  update_state(mmu_en, dc.iflush);
  tlb.i_fsm(mmu_en, dc.itflush, vp, pc, &this->axim);
  if (tlb.state != TLB_ST_INIT && mmu_en)
    this->adf.tag = (tlb.pte >> 10) | VALID_BIT;
  this->check_vtag();
  if (tlb.state == TLB_ST_INIT || !tlb.hit) this->busy = 1;/// TLB miss
  else if (tlb.page_fault)                  this->busy = 0;/// pg fault
  else                                      this->busy = !this->tstt.hit;
  UINT32 inst = RV_NOP_INST;
  if (!this->busy && !tlb.page_fault)  /// read cache word
    inst = this->out_word[this->tstt.hit_way];
  if (!fe.r.ready || fe.r.sw_mode) { tlb.page_fault = 0; }
  if (!fe.r.ready) { inst = prev_inst;}
  prev_inst = inst;
  return (fe.r.sw_mode) ? RV_NOP_INST : inst;
}
```

```
#define RV_NOP_INST  0x00000013
```

AXI-master FSM
I-Cache core FSM
I-TLB FSM
I-Cache tag check on physical tag

Fig. 5. I-Cache top-level FSM code with AXI-master FSM and TLB FSM.

RISC-V Sv32 specification [1]. Fig.4 shows the C++ description of the TLB and the cache, whose parameters such as the associativity (**N_WAY**), the number of cache lines (**N_LINE**) and the number of words per cache line (**N_WORD**) are defined by template parameters to give flexibility to the C++ description. Fig.5 shows the I-Cache code (**ICache::fsm()**), which consists of the following processes.

- ***AXI-master FSM*** (**access_axi()**) controls the access to the external DRAM for retrieving cache line in case of a cache miss or *page table entry* (PTE) in case of a TLB miss.
- ***I-Cache FSM*** (**update_state()**) controls the access to I-Cache for instruction fetch and cache line replacement. Fetched word/tag are stored in **out_word[N_WAY]** and **out_vtag[N_WAY]** for all *ways* simultaneously.
- ***I-TLB FSM*** (**tlb.i_fsm()**) controls the access to TLB memory for PTE fetch and search. If PTE is not found during the PTE search, **tlb.page_fault** is set to 1.
- ***I-Cache tag check*** (**check_vtag()**) is performed on **out_tag[N_WAY]** simultaneously (in the same clock) since the for-loop is unrolled by C2RTL, and the results are stored in **tstt** structure.

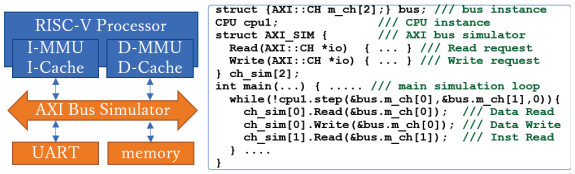

Fig. 6. RISC-V processor simulation testbech C++ code

TABLE I
SIMULATION TIME AND SIMULATED CYCLES ON LINUX BOOT SEQUENCE

model	cycles	time (sec)	cycles/sec	speedup
C++ data flow	175,936,607	30.208	5,824,172.636	236.071
RTL-C	179,012,788	128.423	1,393,930.900	58.887
Verilog (VCS)	179,012,788	7255.930	24,671.239	1.000

VI. SIMULATION AND RTL SYNTHESIS RESULTS

Fig.6 shows the processor simulation model diagram and its C++ testbench code for the simulation of C++ data flow processor code, as well as the simulation of the RTL-equivalent C model generated by the C2RTL. In addition, the synthesized RISC-V RTL model have been integrated into Zynq XC7Z020 FPGA board and have successfully booted the Linux OS.

We use the Linux boot sequence for evaluating the simulation speed. Table I shows the simulation time and simulated cycles of the C++ processor model (C++ data flow), RTL-equivalent C model (RTL-C) and Verilog simulation using Synopsys VCS. Simulated cycles of the C++ data flow model is about 2.2% less than that of RTL-C, which is accurate enough for early design exploration. C++ data flow and RTL-C are about 234x and 55x faster than VCS, respectively. This speedup factor against RTL simulator is significantly higher than that of Chisel-generated C++ simulator [8] which is reported to be 7.77x faster than VCS on 64-bit RISC-V processor with MMU and cache.

Table II shows the FPGA mapping results on Xilinx KU3P-I1 device for 7 different TLB configurations, where the number of ways are set to 1, 2, 4, 8, 16, 32 and 64, while the total number of page-table entries are fixed to 64. On the bottom row, the estimated combinational gate counts calculated by the C2RTL framework are shown. Fig.7 shows the plot of # CLBs on the X-axis and the estimated combinational gates (by C2RTL) on the Y-axis which exhibits a relatively strong correlation except for the 64-ways (full-associative) case.

Even though the combinational gates estimated by C2RTL may not be accurate, this still is a large advantage especially in the early design phase since the RTL generation process by C2RTL takes only about 15 *seconds*, as oppose to about 15 *minutes* required in FPGA mapping. Also, comparison results with other related works on RISC-V synthesis are shown in Table III. Here, the quality of the RTL generated by the C2RTL framework is comparable to other solutions using Chisel [8] and HLS [4] in terms of logic size and clock frequency.

TABLE II
SYNTHESIS RESULTS (FPGA DEVICE : KU3P-I1)

# TLB ways	1	2	4	8	16	32	64
# CLBs	7,694	8,096	8,412	9,380	11,577	16,124	14,537
# FFs	4,129	4,292	4,496	4,888	5,598	7,135	10,144
freq.(MHz)	142.85	133.33	127.22	114.91	104.71	100.00	99.00
comb. gates (C2RTL)	35,317	35,988	37,203	39,633	44,678	54,957	64,605

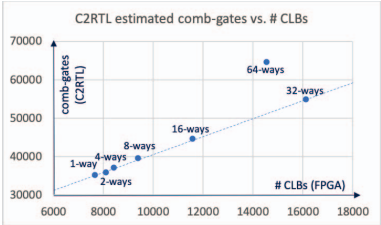

Fig. 7. C2RTL estimated combinational gate counts vs. # CLBs on FPGA

TABLE III
FPGA SYNTHESIS COMPARISONS OF RV32-IM WITH OTHER WORKS (FPGA DEVICE : XA7A12T)

Design	freq. (MHz)	Logic utilization			
		LUT	FF	MUX	DSP
Ours	70.4	2521	1499	275	5
Rocket [8] [4]	76	2570	1275	43	2
Comet [4]	70	2910	2244	227	3

VII. CONCLUSION

We proposed a method to directly describe the RTL structure of a pipelined processor with cache and MMU by using C++ description without any language extensions. The strength of our proposed C++ processor design methodology is the significantly short turnaround time for fast simulation and early circuit size estimation prior to the time consuming logic synthesis and mapping. We are currently working on extending the C2RTL framework for system-level design where IP-level components of SoC and the system integration are all described in C++.

REFERENCES

[1] RISC-V Instruction Set Manual. Volume I: Unprivileged ISA/Volume II: Privileged Architecture.
[2] W. Meeus, et.al, "An overview of today's high-level synthesis tools", Design Automation for Embedded Systems, vol.16, no.3, pp.31-51, 2012.
[3] B. C. Schafer and Z. Wang, "High-Level Synthesis Design Space Exploration: Past, Present and Future," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019.
[4] Simon Rokicki, et.al, 2019. "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications", IEEE/ACM International Conference on Computer-Aided Design (ICCAD).
[5] A. Hoffmann, et. al, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language", IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol.20, no.11, pp.1338 - 1354, 2001.
[6] S. Basu, R. Moona, "High level synthesis from Sim-nML processor models", 16th Int. Conf. VLSI Design, pp. 255 - 260, 2003.
[7] Achim Nohl; Frank Schirrmeister; Drew Taussig, "Application specific processor design: Architectures, design methods and tools", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2010
[8] Jonathan Bachrach, et.al, 2012. "Chisel: constructing hardware in a Scala embedded language". In Proceedings of the 49th Annual Design Automation Conference (DAC '12).
[9] Tsuyoshi Isshiki, et.al, 2015. "C-Based RTL Design Framework for Processor and Hardware-IP Synthesis", In SASIMI 2015 Proceedings.
[10] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In CGO '04, pages 75-86, 2004.