C-Based RTL Design Framework for Processor and Hardware-IP Synthesis

Tsuyoshi Isshiki, Koshiro Date, Daisuke Kugimiya, Dongju Li, Hiroaki Kunieda
Dept. of Communications and Computer Engineering
Tokyo Institute of Technology
{isshiki, date, kugimiya, dongju, kunieda}@vlsi.ce.titech.ac.jp

Abstract *Although high-level synthesis tools and processor synthesis tools have emerged to improve the design productivity of SoC components, we have yet to see a practical solution for the challenging tasks of system-level integration and verification of these individual components involving different design languages, different tool sets and different debugging environments. In this paper, we propose a new C-based design framework where the RTL structure is directly described on dataflow C coding style, while the same C code serves as a fast simulation model. Design example on image signal processing pipeline shows the effectiveness of the proposed C-based tool framework where the dataflow C codes have 1/5 of the number of lines compared to HDLs, can generate high performance circuits with enormously high parallelism of 4000 operations/cycle. Also for RISC processor designs, our dataflow C coding style effectively captures the behavior of the instruction set simulator with less than 1000 lines of C code that runs at 11M cycles/sec speed which is 42x faster than RTL simulation, that can also be directly transformed into RTL description.*

1. Introduction

With the increasing design complexities of state-of-the-art SoCs and shorter time-to-market, there are ever growing demands for enhancing the design productivity as well as design quality. High-level synthesis (HLS) tools have gained industrial adoptions in the last decade [1][2], many focusing on dataflow-oriented DSP application domains with software programming language design entry (such as C/C++/SystemC). Processor synthesis tools have also emerged [3][4][5] that enable easy custom extensions to the instruction-set architecture (ISA) or completely define new ISAs for designing application specific instruction-set processors (ASIPs), where proprietary description languages are used here. Although these efforts in high-level design tools have contributed in enhancing the design productivity and design quality for individual components (IPs) of complex SoCs, system-level design integration and verification of these individual components involving different design languages, different tool sets and different debugging environments is still an enormously time consuming and difficult task.

In this paper, we propose a different approach from conventional works in HLS and processor design, where we use C language itself to directly describe the RTL structures of HW IP blocks and processors without any extensions on the C semantics. The designer expresses the algorithm in dataflow style on C, while a set of hardware attributes required for RTL generation (such as bit-width, clock boundary, memory, etc) is described by C pragmas. The same C code serves as a simulation model (compiled as pure ANSI-C code) as well as the RTL structural model. This capability of expressing RTL structure in C can then be utilized to construct a fast system-level verification model composed of these IP blocks and processors.

This paper is organized as follows: section 2 describes the related works on HLS and processor synthesis, section 3 describes our data-flow C coding style and hardware attributes, section 4 describes the overall tool framework especially on the verification flows, section 5 describes the synthesis flow, section 6 gives several design examples on image processing and processor design, followed by conclusions on section 7.

2. Related Works

There are vast amount of works in HLS over the past three decades [1], where design capture based on standard programming languages such as C, C++ and SystemC coupled with advanced compiler optimization techniques have finally lead many commercial HLS tools to be adopted in real product development. At the very core of these HLS tools is a set of synthesis engines composed of *operation scheduling*, *resource allocation* and *binding* to generate the RTL structural description. The quality of the generated RTL descriptions depends on a number of factors [2], not only on the quality of the internal synthesis engines but also on the various annotations in the design entry for guiding these synthesis engines as well as the coding style of the design input. Thus, optimizing the generated RTL requires a firm understanding of the characteristics of tools' internal engines and the appropriate coding style.

Processor design tools have also emerged from various EDA vendors over the last decade which are mainly adopted for developing custom DSPs. Design entry in these tools includes the definition of instruction-set, resources (register files, memories, registers) and behavioral descriptions of each instruction, using proprietary architecture description languages such as MIMOLA[3], LISA[4] and nML[5]. The use of these proprietary description languages may become an obstacle for inexperienced designers. Also, key components such as caches are usually out of scope of the processor synthesis for these tools.

The key concept of our proposed framework is the use of C language to directly capture the RTL structure of both HW IP blocks and processors using data-flow coding style, while the C code itself serves as a fast simulation model. Our tool framework views the C data-flow description merely as a *netlist representation of the RTL structure*, and therefore does not involve the synthesis steps of scheduling, resource allocation and binding included in conventional HLS tools. Therefore, we do not provide the architecture exploration capability (generating multiple RTL descriptions with area/time tradeoffs) seen in HLS tools. On the other hand, what our framework provides to the designer is the expressive capability equivalent to HDLs for describing any RTL structure on the C code. Unlike the HLS tools where correspondence between the input C code and the generated RTL is often obscure and hard to predict, our framework provides a direct correspondence between the C code and the generated RTL, that is, the property of WYSWYG: *what you write on your C code is what you get*

on your RTL code.

```
#pragma _TCT_verilog_bit_width 10 INT10
#pragma _TCT_verilog_bit_width 12 INT12
#pragma _TCT_verilog_state S_INT12
typedef int INT10, INT12;
typedef INT12 S_INT12; // typedef inheritance
int func0(int a, int b){ ... }
void top0(INT10 a, INT10 b, INT12 c, INT12 *d){
 S_INT12 e;
static S_INT12 f;
 e = func0(a, b);
 f = func0(f, c) + e;
 *d = e;
}
```

Figure 1: C coding example

3. C Dataflow Modeling Scheme for RTL Structural Description

This section describes the C dataflow modeling scheme for enabling C to describe the RTL structure.

3.1. Hardware Attribute Annotations via Pragma

A number of attributes required for constructing the RTL structure are specified with the use of C pragmas such as bit-width, clock boundary (register insertion) and storage elements (register-files, memories). Fig.1 shows a C code example with hardware attribute annotations where the *attribute type* is denoted as **TCT_verilog_XXX**, and these attributes are attached to user-defined **typedef** names. Multiple attributes can be annotated efficiently by **typedef** inheritance. Here, variables **a** and **b** are type **INT10** which is annotated with 10-bit *bit-width* attribute, variables **c** and ***d** are type **INT12** with 12-bit *bit-width* attribute. Variables **e** and **f** are type **S_INT12** annotated with two attributes, 12-bit *bit-width* attribute (inherited from **INT12**), and a *state* attribute. *State* attribute is used to specify the clock boundary, that is, registers will be allocated to such variables where the read operations are delayed by one cycle after the write operation to these variables. Attributes for specifying *memory* and *register-file* are also provided, where these attributes also imply clock boundary at the read ports. While these hardware attributes will be used by the compiler during RTL generation, they have no effect on the C semantics.

3.2. Single Clock Cycle Behavior Restriction

Primary coding restriction in the C dataflow modeling is related to the behavioral model of the top-level function targeted for RTL generation, that is, a single call to the top-level function needs to model *the single clock cycle behavior* of the entire circuit. Here, we assume that the generated RTL model is structured in a pipelined fashion, where the single clock cycle behavior corresponds to that of each pipeline stage operating in sequence (detailed explanation is given in section 3.5). This *single cycle behavior restriction* enforces the C code to be written in a dataflow style, thus make the direct translation from C code to RTL possible. Clock boundary attributes in the C code will be interpreted as *pipeline stage boundaries*. On the other hand, due to this restriction, multiple sequences of read/write operations to the same *state variable* (annotated with *state* attribute) or *memory variable* (annotated with *memory* or *register-file* attributes) are prohibited, since RTL behavior of such description requires multiple cycles.

Parameter variables of the top-level function correspond to inputs and outputs of the circuit, where input/output port direction for each parameter variable is automatically detected by our compiler. In the sample code on Fig.1, assuming that function **top0** is the top-level function, parameters **a, b, c** are the input ports and ***d** is the output port.

3.3. Function Calls and Loops

Any levels of function calls from the top-level function are allowed as long as they are not recursive. Loops such as *for-loops* are allowed as long as the iteration count can be deduced to be constant during compile time (through *constant propagation*). As explained in section 5.2, all function calls reachable from the top-level function will be completely *inlined* (flattened) and all loops will be completely *unrolled* during the compilation phase.

3.4. Variable Lifetime and Reference Directions for Describing Finite State Machines

Previous single cycle behavior restriction also dictates how finite state machines (FSMs) should be described in our C dataflow model. In an FSM, *states* are updated to *next-states* that are accessible in the next cycle. This behavior can be coded in C by the use of *state variables* whose lifetime is longer than that of the top-level function (variables declared above the top-level function, or global lifetime variables). In the code sample on Fig.1, state variable **f** is declared as **static**, and therefore its value will exist after **top0** returns and is reused upon the next **top0** call modeling the behavior of the next cycle. Reference directions to these state variables will determine whether the *current-state* or *next-state* is being referenced. Here, we will call the references that occurs after the assignment to that variable as *forward reference*, and the references that occurs before the assignment as *backward reference*. Forward reference of a state variable require one cycle delay after the assignment, thus implies a clock boundary. Backward reference of a state variable corresponds to accessing the *current-state*. As will be needed in pipelined processor modeling explained in section 6.2, we also allow *non-state* variables (corresponding to combinational logic output signals) to be backward referenced as well, which will be translated as *feedback signals to the previous pipeline stages*. Variables with backward references require that their lifetimes extend beyond the top-level function scope.

In the statement **f = func0(f, c) + e**, the reference to **e** is a forward reference which implies one clock delay after the previous assignment **e = func0(e, c)**, where as the reference to **f** in **func0(f, c)** is a backward reference, therefore its value is computed by this assignment statement on the *previous* call to **top0**, that is, on the *previous cycle*.

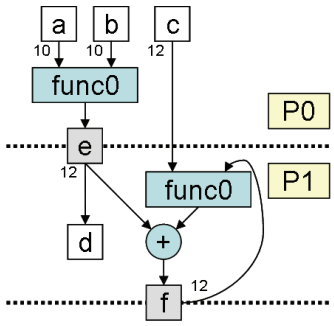

Figure 2: Pipelined RTL structure of C code in Fig.1

3.5. C Behavioral Model vs. RTL Structural Model

Fig.2 shows the pipelined RTL structure of the dataflow C model in Fig.1. Here, we assume that clock boundaries are not inserted inside **func0**. Pipeline boundaries are inserted on state variables **e** and **f**, and two function calls to **func0** are inlined twice. Assignment to **f** contains a forward reference to **e**, and thus **f** sits on the second pipeline stage. Also, the second call to **func0** is also placed in the second pipeline stage due to the backward reference to **f**, where the computation **func0(f, c)** needs to occur within the same cycle as its assignment..

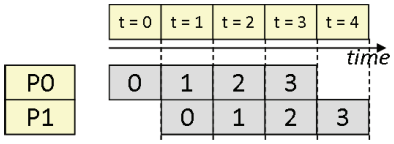

Figure 3: Pipelined RTL behavior of C code in Fig.1

Next, let us compare the behavior of the original C code and the pipelined RTL structure. Figure 3 shows the pipelined RTL behavior of the C code in Fig.1 where the numbers in the shaded boxes correspond to the call sequence indices to the top-level function **top0** ("0" corresponds to the first call, "1" corresponds to the second call, etc). Compared with the pipelined RTL model where each pipeline stage operates on the same cycle (t = 0, 1, 2, …), *the single clock behavior model* of the C dataflow model executes the pipeline stages in sequence, virtually ignoring the pipeline boundaries. Despite the difference in computation ordering between the "untimed" C dataflow model and "timed" pipelined RTL model, they result in the same computational model, that is, sequence of values observed at each variable will be identical. With our proposed C dataflow modeling with pragma-based hardware attribute annotation scheme, we are able to express virtually any kind of RTL structure on the C-language. Pipelined structure is expressed naturally by the use of state attributes to insert clock boundaries. FSMs can also be expressed naturally by the use of backward references to state variables, and can even be distributed among different pipeline stages.

4. Proposed "C2PixPipe" Tool Framework

Fig.4 illustrates the overall flow of our proposed tool framework. Our tool framework which we call *C2PixPipe* (since our initial target for this framework was image processing) parses the C behavioral model coded in dataflow style and generates the RTL structural descriptions in two languages, Verilog-HDL for circuit implementation and cycle-level verification, and also in C code whose behavior is equivalent to RTL model (*RTL-equivalent C model*). These RTL models serve as verification models against the original C dataflow model according to the below verification flow:

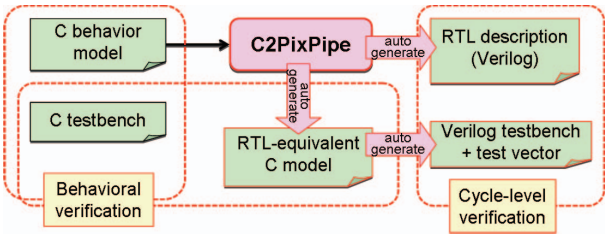

Figure 4: Proposed *C2PixPipe* tool framework

- ***RTL-equivalent C model verification*:** Top-level function of the RTL-equivalent C model contains the same list of function parameters as the top-level function of the original C dataflow model. Thus the same testbench for verifying the C dataflow model can be used for verifying the RTL-equivalent C model, although the circuit outputs will be delayed by the number of pipeline stages.
- ***Verilog RTL model verification*:** RTL-equivalent C model is also equipped with the functionality of recording all input/output port signals during the entire simulation run. We also provide a C pragma to specify a list of signals to be probed during the RTL verification, in which case these probe signals will also be recorded. Verilog testbench model is also generated that reads in the recorded test vectors, drives the input signals of the top-level RTL model, and verifies the output and probe signals.

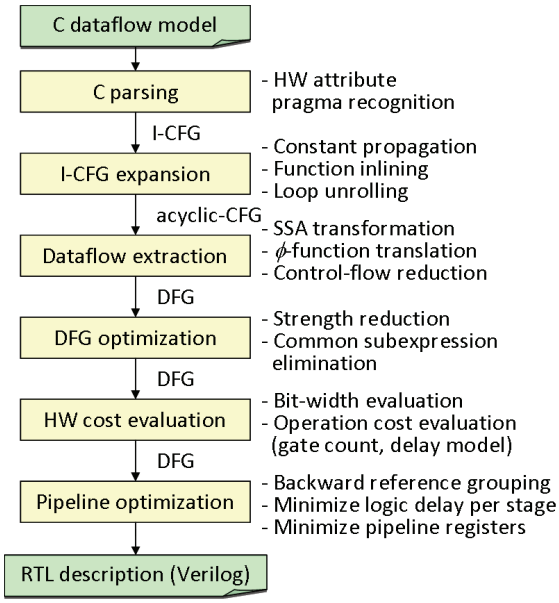

Figure 5: Synthesis flow of our tool framework

5. RTL Synthesis Flow from C Dataflow Model

As explained earlier, our tool framework views the C dataflow model as a *direct netlist representation* of the RTL structure, and therefore the required synthesis steps are quite different from conventional HLS. Fig. 5 shows the overall synthesis flow in our framework which consists of straightforward transformations using compiler and graph

techniques.

5.1. C Parsing

This part was realized by the compiler framework originally targeted for program parallelization in our previous work [6]. C dataflow description is parsed as pure ANSI-C code and transformed into interprocedural control flow graph (I-CFG). Also, C pragmas for annotating hardware attributes are also recognized and passed on to the later synthesis stages.

5.2. I-CFG Expansion

As mentioned earlier, all function calls under the top-level function will be completely inlined and loops will be completely unrolled. *Constant propagation* is performed during the function inlining and before the loop unrolling in order to deduce the constant iteration count. This procedure results in a *flat* CFG (no function hierarchy) which is also *acyclic* (does not contain *cycles* in the CFG).

5.3. DFG Extraction

This flat acyclic CFG is then transformed into a dataflow graph (DFG) by first transforming each C statement into *static single assignment* (SSA) form, where ϕ-functions are inserted on the multiply assigned variables on mutually exclusive control flows. These ϕ-functions are then replaced with multiplexers controlled by the conditional bits which indicate the executed control flow path. Since the input CFG to this step is flat and acyclic, control-flow nodes (branches and joins) can be totally eliminated such that the *entire design is represented by a single DFG*.

5.4. DFG Optimization

This step consists of standard compiler optimization procedures such as *strength reduction* and *common subexpression elimination*. Strength reduction includes the following transformations:

- Constant multiplications into adds and shifts
- Power-of-2 divisions into shifts
- Divisions into series of shift-subtract operations

5.5. HW Cost Evaluation

This step consists of analyzing each operations in the DFG to evaluate the required bit-widths of the intermediate variables (DFG edges), and also estimating the gate counts and delay model for each operation (DFG vertices). Bit-width evaluation is done by computing the value ranges of each operation from the value ranges of its inputs and propagating the value ranges from the primary inputs of the DFG towards its primary outputs. Gate count and delay models are estimated by a fairly abstract datapath circuit model for each operation type.

5.6. Pipeline Optimization

This step determines the optimal pipeline boundaries on the optimized DFG which is formulated as a problem of finding the optimal *cuts* on the DFG. Due to the clock boundary annotations given by the C pragmas in the C dataflow model, there exists a *lower bound* on the number of pipeline stages. This lower bound is determined by examining *all possible paths* from the primary inputs to primary outputs of the DFG and counting the maximum number of clock boundaries that each path crosses. The designer can choose any desired number of pipeline stages equal to or above this lower bound. Thus the problem is formulated as finding the *optimal locations* of *predetermined number of cuts* on the DFG which is achieved by the following procedures:

1. ***Backward reference grouping*:** For each state variable, any paths from its *backward reference nodes* to its assignment node cannot cross any clock boundaries (otherwise, the *current-state* will delay more than one cycle from the *next-state*). Thus, nodes on these paths are grouped into a single node to assure that these nodes will be located on the same pipeline stage. Also, the combinational logic feedback signals explained in section 3.4 will be disconnected. By this node reduction procedure, all cycles in the original DFG will be contained inside these node groups and thus the reduced DFG will become *acyclic*, in other words, a *directed acyclic graph* (DAG).
2. ***Initial cut placement*:** After DFG reduction into DAG, initial cut placement is determined. This is accomplished by a simple DAG longest path algorithm where the distance is measured by the number of clock boundary edges on the path from primary inputs to each node. Edges connecting nodes with different distances become the locations of the cuts.
3. ***Cut placement optimization by simulated annealing*:** Cut placement is gradually modified by a sequence of local node moves between adjacent pipeline stages by *simulated annealing*. Here we divide the annealing process into two phases, where first phase attempts to minimize the maximum logic delay among the pipeline stages and the second phase attempts to minimize the total cut cost, that is, the number of bits crossing the pipeline boundaries.

6. Design Case Studies

In this section, we introduce two design examples, a camera front-end image signal processing system and pipelined RISC processor.

6.1. Image Signal Processing System

Image signal processing (ISP) system for camera sensor front-end [7] was our initial motivation for developing our C-based framework (thus the name *C2PixPipe*). The target ISP system architecture for this project consisted of the following properties:

- ISP pipeline blocks operates on the same clock frequency as the camera sensor interface delivering one pixel per clock with vertical and horizontal blank periods..
- C algorithm models for each ISP pipeline stages will be modeled as one clock cycle behavior.
- All input pixel data at each pixel pipeline stages are delivered from the pixel stream interface between each ISP blocks and line-buffers, thus direct access from the frame-buffer was not considered.

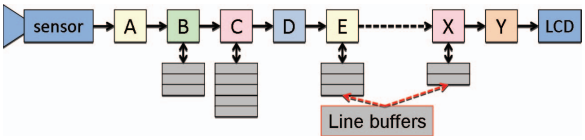

Figure 6: ISP pipeline architecture

```
void IpxTop(        IpxTopCtx* ctx,
                    const XSVI_UINTXX* xsviIn,     XSVI_RGB08* xsviOut,
                    const IpxSVIParam* sp,         const IpxTopParam* tp)
{
        IpxSVI      svi0,svi1,svi2,svi3,svi4,svi5,svi6,svi7,svi8;
        UINTXX      data0;
        SINTXX      data2,data3,data4;
        SINTXXx3    data5,data6,data7;
        RGB08       data8;
                                        Function calls to pixel pipeline stages
        Xsvi2Ipx_UINTXX     (&ctx->Vif, xsviIn, &svi0, &data0, sp);
        IpxOBC      (&ctx->OBC, &svi0, &svi1, &data0, &data1, NULL, &tp->pOBC);
        IpxWBG      (&ctx->WBG, &svi1, &svi2, &data1, &data2, NULL, &tp->pWBG);
        IpxBPF      (&ctx->BPF, &svi2, &svi3, &data2, &data3, sp, &tp->pBPF);
        IpxWNR      (&ctx->WBR, &svi3, &svi4, &data3, &data4, sp, &tp->pWBR);
        IpxACI      (&ctx->ACI, &svi4, &svi4, &data4, &data5, sp, &tp->pACI);
        IpxCCM      (&ctx->CCM, &svi5, &svi5, &data5, &data6, NULL, &tp->pCCM);
        IpxYCN      (&ctx->YCN, &svi6, &svi6, &data6, &data7, sp, &tp->pYCN);
        IpxGAM      (&ctx->GAM, &svi7, &svi8, &data7, &data8, NULL, &tp->pGAM);
        Ipx2Xsvi_RGB08      (&svi8, &data8, xsviOut, sp);
}
```

Figure 7: Top-level function of the ISP pipeline

Table 1: RTL synthesis results of ISP pipeline

ISP functions: Pixel interface conversion, color adjustments, Bayer-RGB demosaic, denoising, color conversion, gamma correction, etc.	
C code size	5,000 lines, 15 source files
# operations	4,195 ops (76 multiplications)
line buffer memory	64 lines (2,304 pixels wide), 1.9M bits
# pipeline stages	32
RTL code size	27,000 lines (Verilog), 20,000 lines (C)
comb. circuit size	225,279 gates (approx.)
FF count	23,393 bits
max clock freq.	300 MHz (approx., @90nm CMOS)
synthesis time	56.8 seconds (excl. C parsing)

Fig.6 shows the assumed ISP pixel pipeline architecture in this design case. Fig.7 shows the top-level function of the ISP pixel pipeline where a total of 10 ISP blocks are called, each having a common pixel stream interface. Table 1 shows the RTL synthesis results of this ISP pipeline design. The entire C dataflow model consists of more than 5000 lines distributed over 15 source files. After parsing these C source files and undergoing the synthesis steps, the final optimized DFG consisted of 4,195 operation nodes, including 76 multiplications whose bit-widths were distributed between 9 bits and 36 bits. It should be noted that this vast amount of operations occur concurrently on the pipelined RTL, thus parallelism of 4,195 operations are achieved by the C dataflow model. A total of 64 line buffers (2,304 pixels wide) are instantiated inside the ISP blocks. The generated RTL consists of 32 pipeline stages (user-specified) where the circuit size is 225K gates for combinational logic, 23K bits of flip-flops, and maximum clock frequency of 300 MHz using 90nm CMOS (here, gate count and clock frequency are approximated by the tool's gate model). It takes a total of 56.8 seconds to generate the RTL descriptions which consists of 27,000 lines of Verilog code and 21,000 lines of RTL-equivalent C code.

6.2. RISC Processor

Upon the successful generation of the previous ISP pipeline architecture with our framework, we set our next target to the RISC processor design. The RISC processor architecture used here comes from authors' previous works on MPSoC designs [8] originally on manual RTL coding and later by Synopsys Processor Designer using LISA language [9][10][11]. Our RISC core is a 32-bit architecture with 4 pipeline stages (FE: fetch, DE: decode, EX: execute, WB: writeback), 32 general purpose registers, with integer multiplier and multi-cycle integer divider (Fig. 8), and has been shown to have comparable performance to that of ARM9 processor [9].

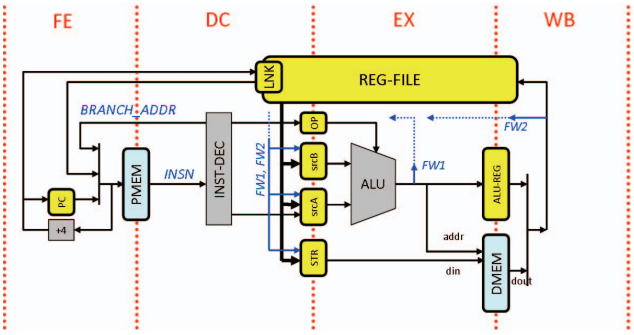

Figure 8: 4-stage RISC pipeline architecture

```
typedef struct ST_CPU
{
  ST_BIT       enable, halted;
  ST_UINT32 pc, cycle, ir_prev;
  ST_UINT4   p_key;
  RF_UINT32 gpr[GPR_COUNT];
  RF_UINT32 spr[SPR_COUNT];
  UINT32       ir;
  M_UINT32  pmem[PMEM_SIZE];
  M_UINT32 dmem[DMEM_SIZE];
```

```
  Insn        insn;
  FEState     fe_stt;
  DCState     dc_stt;
  EXState     ex_stt;
  WBState     wb_stt;
  FEReg       fe_reg;
  DCReg       dc_reg;
  EXReg       ex_reg;
  DIVReg      div_reg;
} CPU;
extern CPU cpu;
```

Figure 9: CPU resource data structure

```
int TCTProc(DI_UINT10 sw, UINT4 key, ...)
{
  BIT        sw0 = (sw) & 1;
  UINT4      key_in = detect_key(key, &cpu.p_key);
  BIT        key1_in = (key_in >> 1) & 1;
  if(cpu.enable){
    if(sw0 == 1){ cpu.enable = 0;}
    fetch();
    decode();
    execute();
    writeback();
    update_pipe_ctrl();
    show_cpu();
  }
  else if(key1_in){ cpu.enable = 1; }
  return (cpu.halted == 1);
}
```

Figure 10: Top-level function of RISC ISS code

In this design exercise, we have started by completely rewriting the processor model in C dataflow model as a simple interpretive instruction-set simulator (ISS) [12][13] which consists of 500 lines of C header file declaring RISC resource data structures (registers, register files, memories, etc) as shown in Fig.9, with another 500 lines of C code for describing the full details of the baseline pipeline behavior as shown in Fig.10. In accordance to the *single cycle behavior coding restriction* explained in section 3, one call to the top-level function **TCTProc** models the single cycle behavior of the RISC pipeline, such coding style comes naturally for such interpretive ISS.

Inside the top-level function **TCTProc**, lower-level functions corresponding to the 4 pipeline stages are called consecutively. Fig.11 shows the function for the FE-stage where program memory array **cpu.pmem[]** is accessed by a normal array indexing expression. The entire C dataflow

code, in fact, only uses standard C expressions without any use of tool-specific predefined macros, data structures or libraries. This makes our C dataflow modeling quite intuitive to any C programmers.

```
void fetch()
{
  INIT_PIPE_CTRL(cpu.fe_stt.pctl, FE_STALL);
  if(!cpu.fe_stt.pctl.stalled_flag){
    UINT32 pc;
    pc = (cpu.dc_stt.br.active) ? cpu.dc_stt.br.addr :
          cpu.fe_reg.nxt_pc;
    cpu.fe_reg.cur_pc = pc;
    cpu.ir = cpu.pmem[pc >> 2];
    cpu.ir_prev = cpu.ir;
    cpu.fe_reg.nxt_pc = (FC_UINT32)(pc + 4);
  }
  else{ cpu.ir = cpu.ir_prev;}
}
```

Figure 11: FE-stage C function

Fig.11 contains some special features required in RISC pipeline modeling, that is, the *feedback signals* from the later pipeline stages (described in section 3.4). **cpu.dc_stt.br.active** is a backward reference from the DC-stage which indicates that a branch instruction is decoded whose target address computed also at DC-stage as **cpu.dc_stt.br.addr** is used to access the program memory. Such backward references in the C dataflow model is necessary for modeling pipeline stalls and data forwarding.

Table 2: RTL synthesis results of RISC pipeline

RISC features: 32-bit data, 4 stage pipeline, 32 general purpose registers, integer multiply unit, multi-cycle integer division unit, UART port (including SerDes), interrupt handling logic	
C code size	850 lines, 5 source files
# operations	1,824 ops (1 multiplication)
# pipeline stages	4
RTL code size	4,100 lines (Verilog), 3,650 lines (C)
comb. circuit size	44,268 gates (approx.)
FF count	3,449 bits (pipeline registers + reg-file)
max clock freq.	300 MHz (approx., @90nm CMOS)
synthesis time	30.5 seconds (excl. C parsing)

Table 3: Instruction-set simulation speed comparison

test program: calculation of 200 prime numbers with UART message of output results (5,600 characters @ 115.2K bps)	
simulation cycles	10,764,342 cycles
ISS time (C dataflow)	0.918 sec (11.726 M cycles/sec)
ISS time (RTL-equiv. C)	5.558 sec (1.937 M cycles/sec)
ISS time (Verilog)	38.850 sec (0.277 M cycles/sec)

Table 2 shows the RTL synthesis results of the RISC pipeline design, where in addition to the baseline RISC pipeline (about 500 lines), additional features such as UART port with SerDes for pin-accurate serial IO modeling and interrupt handling logic are implemented in the C model, resulting in 850 lines of C code. The number of pipeline stages for pipeline optimization phase was obviously set to 4, where changing the number of pipeline stages involves reworking the pipeline control logic (pipeline stalls, data-forwarding) and cannot be simply handled by graph cut algorithm. Generated RTL codes were 4,100 lines of Verilog and 3,650 lines of RTL-equivalent C code. Table 3 shows the simulation speed comparison on the 3 different ISS (instruction-set simulator) models. The original C dataflow model has a throughput of 11.726 M cycles/sec, compared to 1.937 M cycles/sec for the RTL-equivalent C model and 0.277 M cycles/sec for Verilog model.

7. Conclusion

In this paper, we have proposed a novel C dataflow modeling scheme for directly describing the RTL structures of HW IP blocks and processors without any extensions on the C semantics, where a set of hardware attributes are annotated by C pragmas. The required synthesis steps are composed of standard compiler optimization and graph algorithms which allow the tool framework to directly handle relatively large designs exceeding 5000 lines of C code and generate deeply pipelined RTL structure of over 220K gates with parallelism of over 4000 operations per cycle in the case of ISP pipeline synthesis. Our C dataflow modeling applied to processors in turn is a cycle-level ISS with the full details of the processor pipeline behavior with pipeline control logic for pipeline stalls and data-forwarding. Our tool framework enables the designers to describe the RTL structures on C language, where the very same C model serves as a fast simulation model which can be integrated as a simulation component for the system-level verification. In the future, we plan to develop many C dataflow models for various IOs, accelerator engines for creating a wide range of ASIPs.

References

[1] G. Martin, G. Smith, "High-Level Synthesis: Past, Present, and Future", *IEEE Design & Test of Computers*, vol.26, no.4, pp.18 - 25, 2009

[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] P. Marwedel, "The MIMOLA Design System: Tools for the Design of Digital Processors", *Design Automation Conference*, pp.587 - 593, 1984

[4] 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

[5] S. Basu, R. Moona, "High level synthesis from Sim-nML processor models", *16th Int. Conf. VLSI Design*, pp. 255 - 260, 2003

[6] M. Z. Urfianto, et. al, "Decomposition of task-level concurrency on C programs applied to the design of multiprocessor SoC", *IEICE Trans. Fundamentals,* vol.91, no.7, pp.1748 - 1756, 2008

[7] R. Ramanath, et. al, "Color Image Processing Pipeline", *IEEE Signal Processing Magazine*, vol.22, no.1, pp. 34 - 43, 2005

[8] M. Z. Urfianto, et. al, "A multiprocessor SoC architecture with efficient communication infrastructure and advanced compiler support for easy application development", *IEICE Trans. Fundamentals*, vol.91, no.4, pp.1185 - 1196, 2008

[9] H. Xiao, et. al, "A Low-Cost and Energy-Efficient Multiprocessor System-on-Chip for UWB MAC Layer", *IEICE Trans. Information and Systems*, vol.95, no.8, pp. 2027 - 2038, 2012

[10] H. C. Liao, et. al, "A Design of High Performance Parallel Architecture and Communication for Multi-ASIP Based Image Processing Engine", *IEICE Trans. Fundamentals*, vol.E96-A, no.6, pp. 1222 - 1235, 2013

[11] http://www.synopsys.com

[12] R. Leupers, et. al, "Generation of interpretive and compiled instruction set simulators", ASP-DAC '99, pp. 339 - 342, 1999

[13] M. Reshadi, et. al, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation", *Design Automation Conference*, pp. 758 - 763, 2003