Towards Implementing RTL Microprocessor Agile Design Using Feature Oriented Programming

Hongji Zou, Mingchuan Shi, Tun Li, Wanxia Qu
School of Computer Science, *National University of Defense Technology*
Changsha, Hunan, P.R.China, 410073
{zouhongji, mc_16, tunli, quwanxia}@nudt.edu.cn

***Abstract*—Recently, hardware agile design methods have been developed to improve the design productivity. However, the modeling methods hinder further design productivity improvements. In this paper, we propose and implement a microprocessor agile design method using feature oriented programming technology to improve design productivity. In this method, designs could be uniquely partitioned and constructed incrementally to explore various functional design features flexibly and efficiently. The key techniques to improve design productivity are flexible modeling extension and on-the-fly feature composing mechanisms. The evaluations on RISC-V and OR1200 CPU pipelines show the effectiveness of the proposed method on duplicate codes reduction and flexible feature composing while avoiding design resource overheads.**

***Index Terms*—Agile Design Method, Feature Oriented Programming, PyRTL**

I. Introduction

With the ending of Moore's Law and Dennard scaling, modern SoC trends towards incorporating a large and growing number of specialized modules for specific applications, such as AI acceleration units. This growing complexity has led to a design productivity crisis. The hardware agile design methodology (HADM) [1] is developed recently to enable the completion of complex chip designs on schedule and within budget.

The improvement of the HADM productivity depends on the ability of extensive design-space exploration of alternative system microarchitectures. There are several technologies that enable the HADM to have this capability. One depends on using another better language (e.g. Perl) as a macro processing language [2] for an underlying hardware description language (HDL), such as Verilog and VHDL. An alternative and dominate approach is to begin from a domain-specific application programming language from which a hardware block is generated.

The intention of developing hardware domain specific languages (HDSLs) is to make use of the characteristics of modern programming language, such as object-oriented programming (OOP), functional programming, abstract data types, and reflection, to speed up the development cycle by making the design as "soft" as possible. The state-of-the-art HDSLs include Chisel [3] on Scala, PyRTL [4] and PyMTL [5] on Python. The designs described in HDSLs are elaborated through execution and therefore these HDSLs are also called hardware construction languages (HCLs). At present, the abstraction level of most HDSLs is still at register-transfer level (RTL).

Typically, when using HDSLs, a complex design can be decomposed into manageable units, and be described by OOP techniques. However, the object oriented design does not necessarily lead to a unique partition. The non-unique partitioning makes some objects have to implement multiple concepts, and a concept may be found in multiple classes. In addition, the non-unique partition will prevent the HADM from effectively integrating various design features. Therefore, more sophisticated software design techniques should be explored to make the HADM work around rapid design feature integration challenge.

In software design, aspect-oriented programming (AOP) [6] and feature oriented programming (FOP) [7] are two types of generative programming [8] techniques which make a software design integrate various design features incrementally and conveniently. Generally, AOP focuses on weaving the non-functional aspects, while FOP focuses on functional concerns. At RTL, designers concerns are more on functional features than non-functional aspects. Therefore, we prefer using FOP to improve the HADM.

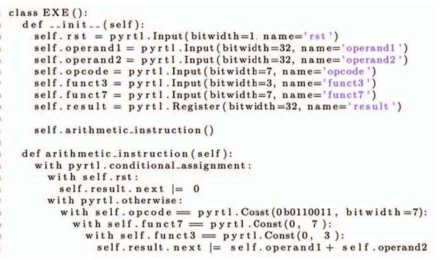
In this paper, we propose a microprocessor agile design method at RTL using FOP technology. The method first extends PyRTL to enable it to describe reusable design features independently, and then provides a feature composing mechanism to construct design on-the-fly in an incremental manner. To avoid dealing with diverse Python programming mechanisms, we make the composing algorithm run on an unified intermediate representation (IR). As PyRTL have added support to emit FIRRTL [9] and Verilog, the composed designs can be used by following design process seamlessly.

The method will construct various potential designs by partitioning a design uniquely and integrating partitions flexibly with or without relations, thus enhancing the HADM. Consequently, designers are able to explore various design features according to the integration results. We apply the method to 2 CPU designs to evaluate the effectiveness of the method. The promising evaluation results show that the method has flexible feature composing and duplicate code reduction ability, while avoiding overhead of design resource usage. To the best of our knowledge, it is the first work on adopt FOP in HCLs.

The contributions of our work includes:

- We propose a method to apply FOP on IR forms, instead

This work was supported in part by the National Natural Science Foundation of China under Grant U19A2062. *(Corresponding author: Tun Li, Wanxia Qu)*

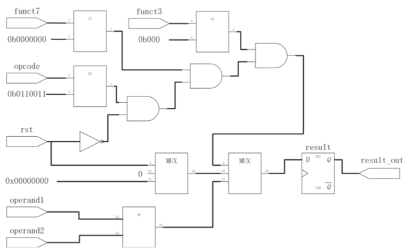


```python
class EXE():
  def __init__(self):
    self.rst = pyrtl.Input(bitwidth=1, name='rst')
    self.operand1 = pyrtl.Input(bitwidth=32, name='operand1')
    self.operand2 = pyrtl.Input(bitwidth=32, name='operand2')
    self.opcode = pyrtl.Input(bitwidth=7, name='opcode')
    self.funct3 = pyrtl.Input(bitwidth=3, name='funct3')
    self.funct7 = pyrtl.Input(bitwidth=7, name='funct7')
    self.result = pyrtl.Register(bitwidth=32, name='result')

    self.arithmetic_instruction()

  def arithmetic_instruction(self):
    with pyrtl.conditional_assignment:
      with self.rst:
        self.result.next |= 0
      with pyrtl.otherwise:
        with self.opcode == pyrtl.Const(0b0110011, bitwidth=7):
          with self.funct7 == pyrtl.Const(0, 7):
            with self.funct3 == pyrtl.Const(0, 3):
              self.result.next |= self.operand1 + self.operand2
```

(a) The EXE stage



(b) The corresponding IR

Fig. 1: A PyRTL sample design

of applying to programming language level, which is different from previous methods;

- We provide an easy-to-use description method, which enables designers to describe design features independently and require as little knowledge of the base design as possible .
- We propose a new feature composing algorithm to generate composed design on-the-fly while not just a preprocessing step. The composing results in no design resource usage overhead.

The related work and some preliminaries are presented in Section II and III, respectively. Our proposed method is explained in detail in Section IV. In section V we present the case study and evaluations of our method. Finally we present our conclusions in Section VI.

II. Related Work

Several works have already addressed the use of AOP concepts for RTL hardware design. In [10], the authors have first discussed how the separation of concerns may relate to different levels of algorithmic abstraction. The use of AOP concepts to design very simple and low level sequential logic designs is proposed in [11]. For actual hardware design, the presentation and assessment of possible applications of AOP by using SystemC with AspectC++ [12] was discussed in [13]. A 128-bit floating-point adder was implemented in SystemC using AOP techniques in a similar work [14].

In addition to using AOP, some works have extended the existing HDLs to support AOP design methods. A new HDL named ADH based on AOP is mentioned but without further details in [10]. In [15], the nature of cross-cutting concerns in VHDL-based hardware designs was discussed and a hypothetical AOP extension for VHDL was proposed. ASystemC [16] and AspectVHDL [17] extended SystemC and VHDL with their own developed aspect weaver, respectively. In addition, AspectVHDL could yield synthesizable code. However, there are few works on adopting AOP to enable HDSLs to compose design features quickly. For applying AOP to HCLs, we only notice that now there is a new "aop" sub-direction in Chisel[1] for now.

When referring to the use of FOP concepts for RTL design, the application of Aspectual Feature Module (AFM) [18] to HDLs was analyzed and shown by several examples on how AFM enables incrementally developing hardware [19] by using SystemC and FeatureC++ [20]. In [21], the authors proposed FeatureVerilog language to extend Verilog to support FOP.

The existing works on extending HDLs capability to apply AOP or FOP techniques limits the potential for reusability since OOP features are not supported. In addition, the weaving of aspects or composing of features was operated as a preprocessing step.

There are two differences between our work and previous research:

- Our method is applied to HDSLs/HCLs based on OOP languages.
- The feature-composing operation is applied to PyRTL IR and results in elaborated composed designs while not just a preprocessing step.

Although our method is implemented in PyRTL, it could be applied to other HCLs, such as Chisel.

There are a number of languages and synthesis tools have been developed for custom processor design, such as EXPRESSION [22] and LISA [23]. In some sense, these works treated "instructions" as features and added new instructions automatically. However, this category of approach works at microprocessor architecture level by generating a retargetable environment for design exploration, and can only handle "instructions" features. While our method works at RTL and is able to handle various "sub-design" that could be expressed as features besides "instructions" features.

III. Preliminaries

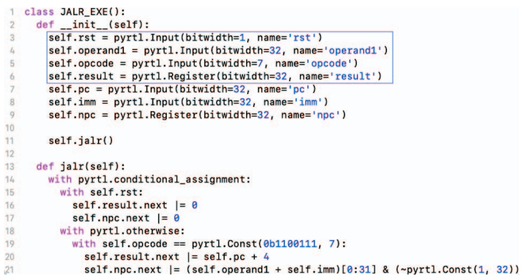
A. PyRTL Background

PyRTL[2] defines a collection of Python classes such as $WireVector$ and $LogicNet$ to model RTL designs. Designs modeled using PyRTL are elaborated-through-execution into pre-defined IR form, and a design in IR form is called a **block**.

The $WireVector$ objects are just used to represent connecting lines between $LogicNet$ objects. While $LogicNet$ objects are used to represent various logic modules at RTL. Each $LogicNet$ object is a 4-tuple and is defined in expression (1).

[1]https://github.com/chipsalliance/chisel3/tree/master/src/main/scala/chisel3/aop

[2]https://github.com/UCSBarchlab/PyRTL

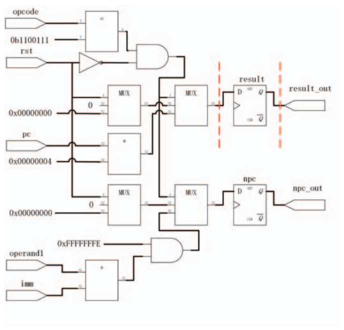


```
class JALR_EXE():
  def __init__(self):
    self.rst = pyrtl.Input(bitwidth=1, name='rst')
    self.operand1 = pyrtl.Input(bitwidth=32, name='operand1')
    self.opcode = pyrtl.Input(bitwidth=7, name='opcode')
    self.result = pyrtl.Register(bitwidth=32, name='result')
    self.pc = pyrtl.Input(bitwidth=32, name='pc')
    self.imm = pyrtl.Input(bitwidth=32, name='imm')
    self.npc = pyrtl.Register(bitwidth=32, name='npc')

    self.jalr()

  def jalr(self):
    with pyrtl.conditional_assignment:
      with self.rst:
        self.result.next |= 0
        self.npc.next |= 0
      with pyrtl.otherwise:
        with self.opcode == pyrtl.Const(0b1100111, 7):
          self.result.next |= self.pc + 4
          self.npc.next |= (self.operand1 + self.imm)[0:31] & (~pyrtl.Const(1, 32))
```

(a) The JALR instruction feature



(b) The IR of the new feature

Fig. 2: A running example

$$(operation, parameters, args, dests) \quad (1)$$

Here $operation$ and $parameters$ define the function and the set of any additional parameters of a $LogicNet$ object, respectively, while $args$ and $dests$ define the set of $WireVector$ objects connected as inputs and driven as outputs for the operation, respectively.

Fig.1(a) and Fig.1(b) show the 3rd stage (EXE) of a simple CPU's pipeline with only one addition instruction and its corresponding PyRTL IR, respectively. The signals from other stages are defined as $Input$ or $Register$. Every conditional assignment is elaborated to a 2:1 multiplexer. The True-case (when the select signal takes value "1") is connected to the result on the right side of the underline assignment statement, while the False-case (when the select signal takes value "0") is connected to a constant value signal "0" or the output of the multiplexers corresponding to the previous conditional level. A constant value signal is denoted by default-value-signal (DVS). For example, the bottom line of the right MUX in Fig.1(b) is connected to the output of a 32-bit adder, which is the right value of the statement "$result| = operand1 + operand2$".

B. FOP Background

Here, we will give a brief introduction to FOP from the point of view of hardware design. Readers could refer to [20] for detailed introduction. When using FOP for hardware design, a system under development is first analyzed from the view of functionality to identify all the features in the design. Then, each feature is implemented as an individual design module. Finally, a number of designs are automatically generated by composing selected feature modules to explore the design space.

In FOP, designers usually use feature expression to construct designs. The expression takes features as operands and "·" as composition operator. For example, "$P = C \cdot B \cdot A$" will generate a design named P, in which feature A and B are composed first, and the result is composed with C.

IV. Our Method

Our method uses FOP technology to enhance PyRTL in 2 aspects. One aspect is to extend the implementation of PyRTL in Python to support FOP modeling in syntax and semantics. The other one is an on-the-fly composing algorithm.

A. A Running Example

Here, we first give a running example to show the basic idea of our method. Suppose we want to extend the design (the **base** design) in Fig.1(a) to add support for a jump and link register (JALR) instruction. The JALR instruction could be defined as a **feature**. In order to implement the JALR feature, designers only need to know that the JALR instruction will use the base design's signals circled by the blue box (Fig.2(a)). Then, the feature itself is elaborated independently as an individual design (Fig.2(b)). The composed design is shown in Fig.3, where the units in blue and brown color are from the JALR feature, the units without coloring are from the base design.

B. PyRTL Language Extension for FOP

We define several mechanisms to help designers flexibly model and compose features to increase design reusability. First, we syntactically define "+" operator as "composing" operator. Then semantically, we overload the "__*add*__" method of PyRTL $Block$ class to return block of the composed circuit. Therefore, the composing formula in Section III-B could be expressed as "P=A+B+C".

We further overload the "__*getattr*__" method in PyRTL $Block$ class. Therefore, designers can select corresponding signals directly from the block using the signals' name. Finally, we add a new "$input\_circuit$" method to the $Block$ class to elaborate designs into PyRTL IR block, which enables our composing algorithm generate composed design on-the-fly.

There is one constraint added on writing PyRTL codes to use FOP–all the $Inputs$, $Outputs$, $WireVector$ and $Registers$ objects in both base and feature designs should be named obviously, as the second parameter of the $Input$ in Fig.1(a). Except having the information on the name of $Inputs$, $Outputs$, and $Registers$ in the base design which are used in the feature designs, designers could model feature designs independently and freely without any other information about the base design.

C. The Composing Algorithm

The idea behind the composing algorithm is to "merge" the **identical signal** (a signal which appears both in the base and

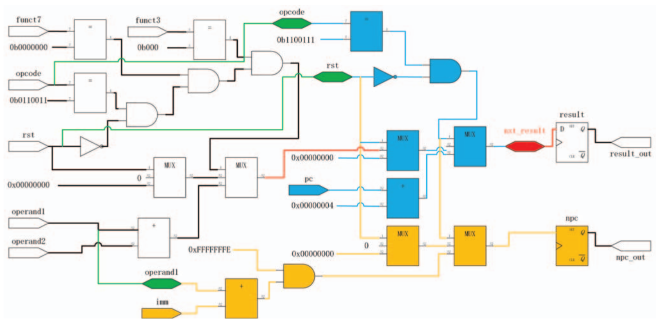


Fig. 3: The IR of the composed design

```
Algorithm 1: Compose($B$, $F$)
  Input: blocks of the base circuit $B$ and the feature
         circuit $F$
  Result: block of the composed circuit $result$
1  $F'$ = Preprocessing($B$, $F$);
2  $relevant\_outputs$ = Analyze_Outputs($B$, $F'$);
3  $result$ = $\emptyset$;
4  foreach $output \in relevant\_outputs$ do
5      $default\_signals$ = Trace_Back($F'$, $output$);
6      if $default\_signals \neq \emptyset$ then
7          $result$ = Compose_1($B$, $F$);
8      else
9          Exit("Could not be composed");
10 $result$ = Connect_Other_Identical_Signals(B, F');
11 Postprocessing($result$);
12 return $result$;
```

feature design with the same name). If the identical signals are driver signals, the "merge" is done by selecting the outputs of the identical signals between base and feature designs according to the driven conditions of the signals in both designs. On the other hand, if the identical signals are primary inputs, only one copy of signals are remained.

The composing algorithm is listed in Algorithm 1. The 2 inputs are the blocks of a base and a feature design to be composed, respectively. The blocks are generated using "$input\_circuit$" method of the $Block$ class. The algorithm will return the composed design in the form of PyRTL IR.

In the following, the set of inputs, outputs, wires and registers in a design D are denoted by $Input_D$, $Output_D$, $WireVector_D$ and $Register_D$, respectively. For example, $Input_B$ denotes the set of inputs of the base design in the algorithm.

At first, the base and feature to be composed are preprocessed to handle the identical registers (Line 1). These registers should be merged in the composed design. Therefore, for each identical register $r \in (Register_F \cap Register_B)$, the $Preprocessing$ procedure will divide the feature design logic related to r into 3 parts, which are the register r, the next-value combinational logic (NVL), and the register output logic (ROL). As the red dotted lines do in Fig.2(b). For the 3 parts, the $Preprocessing$ will:

1) delete r;
2) change the type of the output signals of the NVL to $Output$ type, and then rename it to the input signal's name of the identical register in the base design;
3) change the type of the signals in the ROL which take the value of the register output to $Input$ type, and name it using the name of r.

As shown in Fig.3, the $result$ register in Fig.2(b) is deleted. The NVL of $result$ is connected to an output called nxt_result which is identical to the input of the $result$ register in the base design.

The $Analyze\_Outputs$ procedure (Line 2) is used to calculate the set $relevant\_outputs = Output_{F'} \cap (WireVector_B \cup Register_B \cup Output_B)$, which contains all the $Output$ signals in the feature design that are identical to any non-$Input$ signals of the base design. For example, the red color nxt_result signal is an element in the $relevant\_outputs$ list.

Subsequently, for each $output$ in the $relevant\_outputs$ set (Line 4), the $Trace\_Back$ procedure (Line5) will traversal backwards from $output$ (the blue circuit in Fig.3) to primary inputs in the preprocessed feature F'. Any DVSs found will be added to the $default\_signals$ set during traversal.

The finding of any DVSs means that the two designs could be composed. In this situation, The $Compose\_1$ procedure is called to compose the two designs (Line7). In detail, there are 2 type of composing operations implemented for different DVSs situations:

- $output \in Output_B$: It will connect the identical output of the base design to the DVSs in the $default\_signals$ set. This connection will substitute the default values with the base design's output.
- $output \in WireVector_B \cup Register_B$: It will first cut the identical signal in the base design, then connect the left part of the cut to the DVSs in the $default\_signals$ set, and connect $output$ to the right part of the cut, respectively. The red lines in Fig.3 illustrate the results for this situation.

On the other hand, if no DVSs are encountered, the algorithm will report an error and exit. The principle behind the composing of 2 designs' relevant outputs is to make the composed design take one of the outputs of the 2 designs. Therefore, if the feature design's outputs are DVSs, the outputs of the composed design are determined by the base design, otherwise are determined by the feature design.

Based on this consideration, If the outputs of the feature circuit are deterministic without default values, the feature could not be composed with the base design. Because even if we can infer a multiplexer to select output from the 2 designs, we can not determine the input of the select signal of the inferred multiplexer. However, designers can avoid this situation by composing the base design into the feature design using "feature+base" expression.

Thereafter, the $Connect\_Other\_Identical\_Signals$ procedure (Line 10) will further handle other types of identical signals, such as:

- some feature design's $Output$ signals are identical to some base design's $Input$ signals ($Output_{F'} \subseteq Input_B$), or some base design's $Output$ signals are identical to some feature design's $Input$ signals ($Output_B \subseteq Input_{F'}$);
- some feature design's $Input$ signals are identical to some base design's $Input$ signals or $Register$ ($Input_{F'} \subseteq (Input_B \cup Register_B)$).

For identical signals in above situations, the $Connect\_Other\_Identical\_Signals$ procedure just connects each identical pair between the 2 designs. For example, as the the green lines shown in Fig.3.

Finally, we will do some post processing on the composed design $result$ (Line 11). Some $Input$ or $Output$ signals which have become internal signals in the composed design will be changed to $Wirevector$ (e.g. the green inputs and red nxt_result output in Fig.3) for sanity checking.

To ensure the correctness of the algorithm, we use dynamic simulation to verify that the composed designs are in agree with the design specifications.

V. Case Studies and Evaluation

We have implemented the proposed method in PyRTL in about 500 lines of code (LOC) with the help of the PyRTL predefined IR processing APIs. We take RISC-V[3] and OR 1200[4] as 2 cases to evaluate our method, and all the designs are implemented using PyRTL.

For a given CPU base design, 3 students use the proposed method (FOP), OOP inheritance (OOPI), and direct modification (DM) methods to extend the same base design with new features, respectively. The evaluation consists of two stages. In the first stage, 3 methods are used to extend the RISC-V CPU and the OR 1200 CPU with new instructions and the level of productivity in their respective designs are compared. In the second stage, the extended designs are synthesized to FPGA to compare the hardware overhead.

We take each (type of) instruction in the instruction set architecture (ISA) of RISC-V or OR 1200 as a feature. Therefore, we extend a base CPU design by composing features to add new instructions supports. In Table I, each row indicates a feature. For RISC-V CPU, we implement a five-stage pipeline for the addition instruction (384 LOC) as base design. Then we extended it with RV32I ISA (R/I/S/U-Type) and two types of immediate data processing instructions (B/J-Type). For OR1200 CPU, we implement a pipeline (4659 LOC) for the basic ISA, such as addition, logic operation, and jump instructions. Then, two features corresponding to MAC (multiplication) and CUST (concatenation) instructions are implemented.

[3]https://riscv.org
[4]https://opencores.org/projects/or1k_old/openrisc 1200

A. Design Overhead and Productivity

For each method, the LOC added to extend functions (the "+LOC" columns) and the time each student spent (the "Time" columns) on function extensions are collected and compared in Table I. The time cost includes comprehending base design and implementing feature designs. In addition, we list the number of $LogicNet$s elaborated for each extension in the 3 methods in the 4th, 7th and 10th columns in Table I, respectively.

From Table I, we can find that the LOC added by FOP are a few more than those by DM, but less than by OOPI. The reason is that both FOP and DM methods solely focus on the implementation of new features. However, each feature described in FOP is a unique circuit and therefore various necessary signals have to be declared, which brings some redundancy for each feature design. In addition, both FOP and DM extend base designs by modifying designs at the code level, while OOPI at the function level. Therefore, when the base design is well designed and the module is divided carefully, the LOC added by OOPI is close to those by DM. Conversely, when the base design is designed not so well and part of the base design needs to be rewritten to integrate new features, a lot of redundant code should be added.

In terms of design and implementation productivity, DM has similar time overhead to OOPI, while FOP has the lowest time overhead. With the DM design approach, designers have to drill down to the code level to understand all the implementation details of the base design for design extensions. On the other hand, with OOPI design approach, designers should first comprehend the implementation framework of a base design which is at the function level. For extending new functionalities, designers only need to pay attention to the specific implementation of the corresponding function. Therefore, the cost of implementing DM for a well-designed circuit is similar to that of OOP.

Finally, with the FOP design approach, designers should only have the signals definition information of a base design which is at the module level. Therefore, designers dont need to understand the detailed implementation of the base circuit, but the basic function of the base design module and module interface. In addition, FOP is a method of direct synthesis of new features into the base design circuit, which avoids the optimizations on HDL source codes.

B. Hardware Overhead

As the scale of RISC-V CPU is too small to be compared, we only list the number of Look Up Tables(LUTs) for the extended OR 1200 CPU in the 3 methods in Table II, respectively.

TABLE I: Design Overhead and Productivity

Design	Features	FOP			OOPI			DM		
		+LOC	LogicNets	Time	+LOC	LogicNets	Time	+LOC	LogicNets	Time
RISC-V	R-Type (9 instructions)	33	650	1 Hour	35	649	2 Hours	26	662	2.2 Hours
	I-Type (9 instructions)	42	1004		45	1006		33	1040	
	U-Type (1 instructions)	13	1108		31	1117		4	1168	
	J-Type (1 instructions)	13	1684		33	1703		3	2222	
	B-Type (6 instructions)	95	1697		107	1727		76	2244	
	S-Type (2 instructions)	13	1021		33	1030		2	1062	
OR1200	CUST	84	13430	0.6 Hour	442	13646	1 Hour	34	13646	1.2 Hours
	MAC	243	13204	1.2 Hours	737	13175	2.5 Hours	152	13308	2.5 Hours

TABLE II: Hardware Overheads

Design	Features	Number of LUTs		
		FOP	OOPI	DM
OR1200	CUST	2377	2363	2377
	MAC	3719	3784	3764

It is obvious that the number of LUTs of the 3 designs are very close, and there is no extra hardware overhead in FOP design. On the contrary, the FOP method could optimize the circuits in some situation (MAC features), as FOP compose the base circuit and feature circuit directly on the IR form rather than on the code level. For CUST feature, as FOP design only pays attention to interfaces of module, when the design of base circuit is in coarse granularity, the FOP may produce extra hardware overhead. For example, if we need an internal signal, we have to use the $Input$s of the module to get the same signal. Consequently, these circuits are extra overhead. But it is easy to eliminate it by the common subexpressions elimination optimization.

Finally, from Table I and II, we can conclude that the proposed FOP method allows designers to explore various design features with no hardware overhead while with high productivity and low design effort overhead.

VI. Conclusions

In this paper, we propose a method for microprocessor agile design at RTL using feature oriented programming techniques and it is available open source: https://github.com/zousheng/PyRTLFOP. The experimental results shows the effect of our method. In future work, we intend to empirical evaluate our method on more real-world microprocessor designs. We are also planning to apply design optimization techniques to composed designs.

References

[1] Lee Y, Waterman A, Cook H, et al., "An Agile Approach to Building RISC-V Microprocessors," IEEE Micro, vol. 36, no. 2, pp. 8–20, 2016.

[2] O. Shacham, O. Azizi, et al., "Rethinking digital design: Why design must change," IEEE Micro, vol. 30, no. 6, pp. 9–24, 2010.

[3] J. Bachrach et al., "Chisel: Constructing Hardware in a Scala Embedded Language," DAC'12, pp. 1216–1225, 2012.

[4] D. Dangwal, G. Tzimpragos, T. Sherwood, "Agile Hardware Development and Instrumentation With PyRTL," IEEE Micro, vol. 40, no. 4, pp. 76–84, 2020.

[5] D. Lockhart, G. Zibrat, C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," IEEE Micro, pp. 280–292, 2014.

[6] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Trans. on Soft. Eng., no. 5, vol. 2, pp. 128–138, 1979.

[7] D. Batory, J. N. Sarvela, A. Rauschmayer, "Scaling Step-Wise Refinement," IEEE Trans. on Soft. Eng., no. 30, vol. 6, pp. 355–371, 2004.

[8] K. Czarnecki, U. W. Eisenecker, Generative programming: methods, tools, and applications, ACM Press/Addison-Wesley Publishing Co., 2000.

[9] Adam M. Izraelevitz, et.al., "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," ICCAD'17, pp. 209–216, 2017.

[10] A. Bainbridge-Smith, S.-H. Park, "ADH: an aspect described hardware programming language," FPT'05, pp. 283–284, 2005.

[11] P. Burapathana, P. Pitsatorn, B. Sowanwanichkul, "An applying aspect-oriented concept to sequential logic design," ITCC'05 , vol. 02, pp. 819–820, 2005.

[12] O. Spinczyk, A. Gal, W. Schröder-Preikschat, "AspectC++: an aspect-oriented extension to the C++ programming language," ICTP(OAMEA)'02, pp. 53–60, 2002.

[13] D. Dharbe, S. Medeiros, "Aspect-oriented design in systemC: implementation and applications," ICSD'06, pp. 119–124, 2006.

[14] F. Liu, Q. Tan, X. Song, N. Abbasi, "AOP-based high-level power estimation in SystemC," GLSVLSI'10, pp. 353–356, 2010.

[15] M. Engel, O. Spinczyk, "Aspects in hardware: what do they look like?," AOSD(ACPIS)'08, pp 5:1–5:6, 2008.

[16] Y. Endoh, "ASystemC: an AOP extension for hardware description language," ASDC'11, pp. 19–28, 2011.

[17] M. Meier, S. Hanenberg, O. Spinczyk, "AspectVHDL stage 1: the prototype of an aspect-oriented hardware description language," MSS'12, pp. 3–8, 2012.

[18] S. Apel, T. Leich, G. Saake, "Aspectual feature modules," IEEE Trans. on Soft. Eng., vol. 34, no. 2, pp. 162–180, 2008.

[19] J. Ye, T. Li, Q. Tan, "The application of aspectual feature module in the development and verification of SystemC models," FDL'09, pp. 1–6, 2009.

[20] S. Apel, T. Leich, M. Rosenmüller, G. Saake, "FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming," GPCE'05, pp. 125–140, 2005.

[21] J. Ye, Q. Tan, T. Li, G. Cao, "FeatureVerilog: Extending Verilog to Support Feature-Oriented Programming," PDPW'11, pp. 302–305, 2011.

[22] A. Halambi, P. Grun, V. Ganesh, A. Kahare, N. Dutt, A. Nocolau, "Expression: A language for architecture exploration through compiler/simulator retargetability," In DATE'99, pp. 485–490, 1999.

[23] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebushch, O. Wahlen, "A novel methodology for the design of application-specific instruction-set processors (asips) using a machine description language," IEEE Trans. on CAD of Integ. Circ. and Sys., no. 20, vol. 11, pp. 1338–1354, 2001.