一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2019-71,CPSY2019-69,RECONF2019-61(2020-01)

RISC-V 機械語プログラムからのバイナリ合成

浜名　将輝†　石浦菜岐佐†

† 関西学院大学 理工学部 〒 669–1337 兵庫県三田市学園 2-1

あらまし　本稿では, RISC-V 機械語プログラムからハードウェアを自動合成する手法を提案する. RISC-V を用いた CPU には BSD ライセンスが適用されるため, 内部設計の公開やライセンス料が不要であり, 自由に改変や複製を行うことができる. 高位合成の一種であるバイナリ合成は通常のコンパイラが存在する高級言語に加え, インラインアセンブリやアセンブリプログラムも合成対象とすることができ, アセンブリで書かれた割り込みハンドラをハードウェアに自動合成することも可能である. 本稿では, RISC-V の RV32IM 命令セットによる機械語プログラムを入力として, それを実行する CPU と機能等価なハードウェアを自動合成する. 本稿のバイナリ合成は, 固定サイクルで実行されるカスタム命令を含む機械語も, その命令を実行するハードウェアが独立していれば通常のスケジューリングとバインディングのフローに含めることによって合成対象とすることができる. 提案手法に基づいて合成系を実装し, 実験を行った結果, 160 命令以下のプログラムで Rocket Chip よりも小さな回路規模を実現しつつ実行時間を最大で 14 倍程度短縮できた. また飽和処理を含む SIMD 加算を行うカスタム命令を使用したプログラムでの実験では使用しなかった場合に比べ実行時間を 3.5 倍程度短縮できた.

キーワード　バイナリ合成, 高位合成, RISC-V, ハードウェア/ソフトウェア協調設計, 組込みシステム

Binary Synthesis from RISC-V Executables

Shoki HAMANA† and Nagisa ISHIURA†

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This article presents a method of synthesizing hardware from RISC-V binary codes. RISC-V is an open source instruction set architecture, where several CPU designs are provided under BSD licenses. Binary synthesis, a variant of high-level synthesis, can auto-generate hardware from assembly programs or inline assembly codes, and can be used to synthesize interrupt handler written in assebmly language into hardware. This article presents the first binary synthesizer which takes an executable binary codes for RV32IM and synthesizes a hardware module which is functionally equivalent with a CPU that runs the code. A CDFG is generated from a linked executable binary code, from which an RTL description is generated by the conventional high-level synthesis flow. This method can incorporate custum instructions into the synthesis flow, provided they are executed in fixed cycles and the execution units for them are separately designed from the CPU's datapath. From small scale codes consisting of less than 160 instructions, a prototype synthesizer has generated smaller and faster hardware modules than a Rocket Chip. A code containing SIMD add-saturate instruction has been also synthesized to accelerate the resulting hardware.

Key words　Binary sythesis, High-level synthesis, RISC-V, Hardware/Software codesign, Embedded systems

1. はじめに

RISC-V [2] はオープンソースの命令セットアーキテクチャ (ISA) であり, コンパイラやデバッガ, シミュレータなどのツールチェーンもすべて公開されている. RISC-V を用いた CPU には BSD ライセンスが適用されるため, ISA に対するライセンス料を支払うことなく CPU の開発を行えるのに加え, その複製, 改変, 再配布, 販売ができる. 更にその内部設計の公開も不要のため商用利用にも適している. また RISC-V は高いスケーラビリティを持ち, 小規模な組込みシステムや IoT デバイスから高性能コンピュータまで幅広い応用に用いることができる.

RISC-V の CPU を用いたシステムを更に高速化する必要がある場合には, カスタム命令による拡張や, 処理の一部を専用ハードウェア化するなどの手法が考えられる. 後者のハードウェアを効率的に設計する手法としては, 高位合成技術 [1] を利用した設計手法が提案されている [4], [5].

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2020 by IEICE

しかし外部機器を制御するようなシステムでは, プログラム中にアセンブリで書かれた外部割り込みハンドラを含むことがあり, そのような場合, 高位合成ではそのままハードウェアに合成することができない. RISC-V の割り込みハンドラごとハードウェア化したい場合や, インラインアセンブリを含むプログラムをハードウェア化したい場合にはバイナリ合成を用いる手法が考えられる.

バイナリ合成 [6] は機械語プログラムを入力としてハードウェア記述を自動合成する手法である. コンパイラが存在する高級言語に加えてアセンブリやインラインアセンブリからのハードウェア合成も可能である. 文献 [6] は MIPS, ARM, MicroBlaze の機械語からハードウェアを自動合成する. また文献 [3] は MIPS R3000 の機械語からハードウェアを自動合成し, 更に外部割り込みハンドラを独立したモジュールとして自動合成している [7].

本稿では, RISC-V の機械語を対象としてバイナリ合成を行う手法を提案する. 対象とする命令は RISC-V の RV32IM 命令とカスタム命令である. 本手法により RV32IM 命令で構成されたテストプログラムを用いて動作確認と性能評価を行った結果, 160 命令以下のプログラムで Rocket-Chip よりも小さい LUT を実現しながら 最大で約 75 % サイクル数を削減することができた. 更にカスタム命令を含むプログラムの合成では期待通りサイクル数の削減を確認することができた.

2. RISC-V とバイナリ合成

2.1 RISC-V

2.1.1 RISC-V の命令

RISC-V ISA にはカテゴリごとに命令セットが定義されている. 図 1 に命令セットモデルの一部を示す. CPU の設計を行う際には, ベースとして 32bit, 64bit, 128bit のいずれの基本命令を使用するかを選択し, その基本命令に加えて整数乗除算命令 (M) や浮動小数点数演算命令 (F, D, Q) などの命令拡張を指定する. ユーザは CPU の性能に対する要求やリソース制約を考慮して使用する命令セットを選択する.

RISC-V の 32 bit 命令には表 1 に示すように 6 種類の命令タイプがあり, 基本的なエンコーディングの方法が決められている.

BASE	
RV32I	32bit base instructions
RV32E	32bit base instructions(16 registers)
RV64I	64bit base instructions
RV128I	128bit base instructions
Extension	
M	Integer Multiplication and division
A	Atomic Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
Q	Quad-Precision Floating-Point
L	Decimal Floating-Point
C	Compressed Instructions
B	Bit Manipulation
J	Dynamically Trandlated Languages
T	Transactional Memory
P	Packed-SIMD Instructions
V	Vector Operations

図 1: RISC-V 命令セットモデル

2.1.2 カスタム命令

RISC-V CPU を用いたシステムを高速化する手法の一つとして, ユーザが挙動を自由に設計できるカスタム命令を用いる方法がある. 命令を実行するアクセラレータを適切に設計することにより, そのシステムに特化した CPU を作ることが可能である. 32 bit の RISC-V では下位 2 から 6 bit を設定することにより最大 4 種類のカスタム命令を定義できる. 設計者はカスタム命令のタイプを 6 種類の命令タイプの中から選択する. R, I, S, B タイプの場合は “funct 領域” を用いることにより派生命令を作ることもできる.

表 1: RISC-V 命令タイプ一覧

R タイプ	2 つのレジスタを対象とする命令用
I タイプ	短い即値を使用する命令と, ロード命令用
S タイプ	ストア命令用
B タイプ	条件分岐命令用
U タイプ	長い即値を使用する命令用
J タイプ	無条件ジャンプ命令用

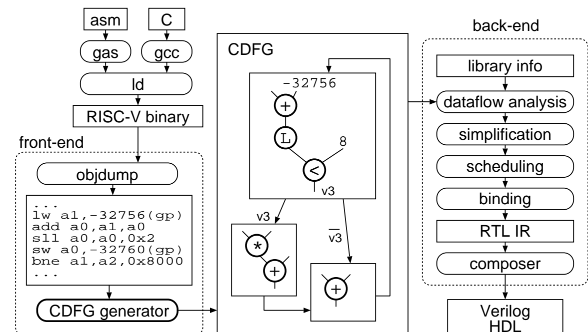

図 2: ACAP での合成の流れ [3]

2.2 バイナリ合成

高位合成 [1] は, C 言語等の手続き型言語で書かれた動作記述から論理合成可能なレジスタ転送レベルのハードウェア設計記述を自動合成する技術である. 与えられた動作記述から, 中間表現である CDFG (control dataflow graph) を生成し, スケジューリング, バインディングを経て論理合成可能なハードウェア設計記述を生成する.

これに対し, バイナリ合成 [6] はプログラムをコンパイル, リンクして得られる機械語プログラムより CDFG を生成し, ここからハードウェア記述を合成する. これにより, 様々な高級言語の記述に加え, アセンブリコードやインラインアセンブリを含むコードからハードウェア設計記述を生成することができる.

プロセッサが外部機器を制御する場合には割り込み処理がアセンブリで書かれることがある. 文献 [7] では, バイナリ合成により割り込みハンドラをハードウェア化することにより, 割り込み処理のレイテンシを削減している.

また一般的な高位合成ツールでは, 複数の関数でグローバル変数を共有する場合, ソースコードの修正やプラグマの指定が必要になる. グローバル変数の読み書きは, 機械語プログラム中ではデータメモリへのアクセスにコンパイルされているため, 複数の関数が共有するグローバル変数が存在しても, バイナリ合成ではソースコードを修正することなく, そのままハードウェアに合成ができる.

文献 [6] は MIPS, ARM, MicroBlaze の機械語プログラムの一部をハードウェアにするバイナリ合成システムを開発した. また ACAP [3] は MIPS 機械語プログラムの一部をコプロセッサとして生成するか, 全部をハードウェアモジュールとして生成す

る．しかし, RISC-V の機械語プログラムからハードウェアを生成するバイナリ合成システムは存在しない.

3. RISC-V 機械語プログラムからのバイナリ合成

本稿では RISC-V の機械語プログラムからハードウェア設計記述を生成する．基本的な合成手法は [3] と同様である．対象とする命令は RV32IM (32bit 基本命令と整数乗除算命令) である.

本稿のバイナリ合成は文献 [3] の "Full synthesis mode" であり, 図 3 に示すように CPU と命令メモリをハードウェアに変換する．このハードウェアは命令メモリ内の機械語プログラムを実行する CPU と機能的に等価であるが, 演算の並列化やチェイニングにより処理が高速化される．また, 命令メモリはハードウェアの制御部に置き換えられるため, 小規模なプログラムでは回路規模が削減される.

3.1 合成の流れ

合成処理の流れを図 2 示す．入力プログラムは C , インラインアセンブリを含む C, またはアセンブリである．これらをコンパイル (gcc), アセンブル (gas) し, 実行可能なバイナリコードへとリンク (ld) する．フロントエンド部分ではこのバイナリコードを逆アセンブル (objdump) したものから CDFG を生成する．バックエンド部分ではこの CDFG に通常の高位合成と同様にスケジューリング, バインディングの処理を施し, Verilog HDL による記述を生成する.

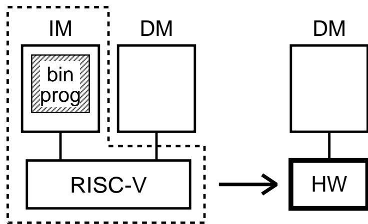

図 3: 本稿のバイナリ合成の概要

3.2 CDFG の生成

入力となるアセンブリコードは基本ブロックに分割し, 各基本ブロックを DFG (dataflow graph) に変換する．1 演算命令はその機能を実現する CDFG の 演算ノードに変換する．分岐命令やジャンプ命令は, 分岐条件を計算する演算ノードと DFG 間の遷移に変換する．レジスタジャンプは, プログラムアドレスをハードウェアの状態に変換するハードウェアを生成することにより実現する.

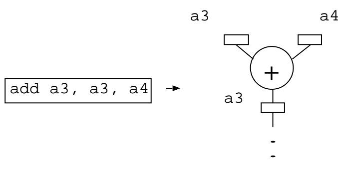

図 4: 基本的な算術/論理演算ノードへの変換

3.2.1 演算命令とロード/ストア命令

算術演算や論理演算命令は各命令に対応する CDFG の演算ノードに変換し, レジスタアクセスは DFG の値ノードとデータ依存関係に変換する．例えば 図 4 に示すように RISC-V の add 命令は DFG の加算ノードに変換する．レジスタ a3, a4 の参照は DFG 中の対応する値ノードからのデータ依存に変換する．この命令が更新するレジスタは参照した a3 であるが, 値が更新された場合はどのレジスタかに関わらず新しい値ノードを生成する.

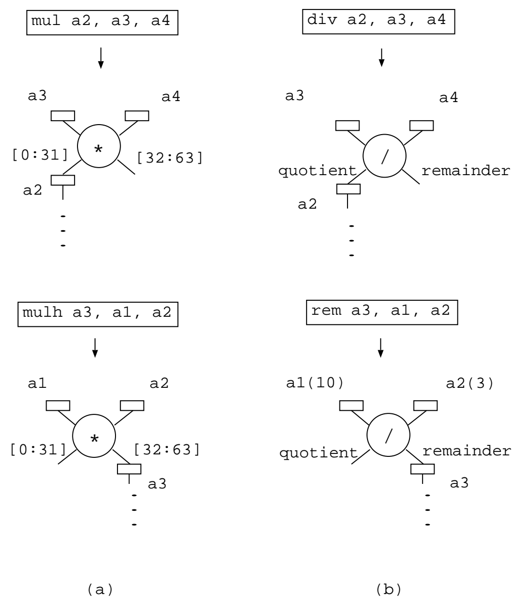

図 5: 乗算除算処理ノードへの変換

乗算命令の場合の変換例を図 5(a) に示す．RV32M には結果が 32 bit のオーバーフローを無視する命令 (mul) と, 計算結果の上位 32 bit だけをレジスタに格納する命令 (mulh) が存在する．CDFG の演算としては積の結果を 64 bit で計算するものを用意しておき, mul 命令の場合は下位 32bit を, mulh の場合には上位 32bit を値ノードに格納する.

同様に, 2 変数の商を計算する命令 (div) と剰余を計算する命令 (rem) は, 図 5(b) に示すように, CDFG の除算処理として商と剰余を計算する処理を用意しておき, div 命令の場合は商部分を, rem 命令の場合は剰余部分をそれぞれ新しい値ノードに格納する.

ロード/ストア命令は, 図 6 のように入力オペランドから実効アドレスを計算するノード (+) と, ロード命令ならメモリからの読み出しを行う処理ノード (L), ストア命令なら書き込みを行う処理ノード (S) に変換する．ロード命令の場合はメモリから読み出した値を, lb, lh 命令なら 32bit にゼロ拡張, lbu, lhu 命令なら符号拡張を行って値ノードに格納する. ストア命令の場合は新たな値ノードは生成しない.

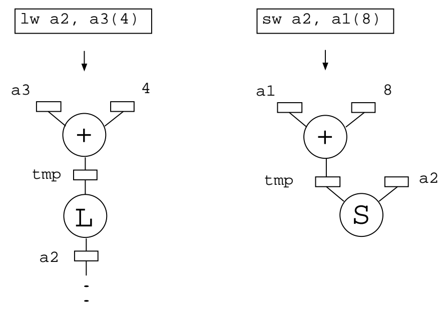

図 6: ロード/ストア処理ノードへの変換

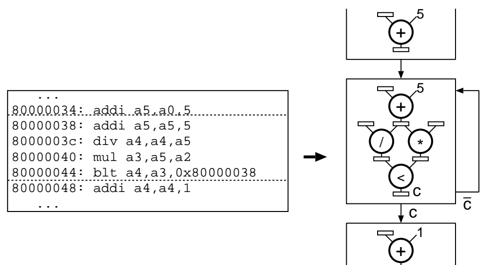

図 7: 分岐命令/即値ジャンプ命令の変換

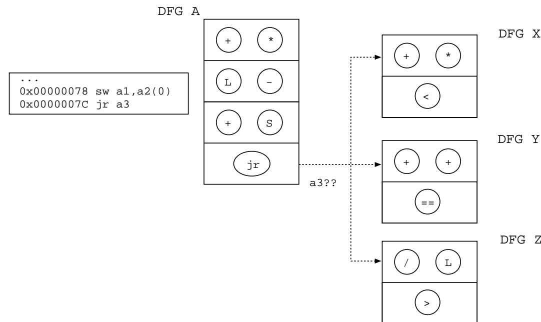

図 8: レジスタジャンプでの遷移

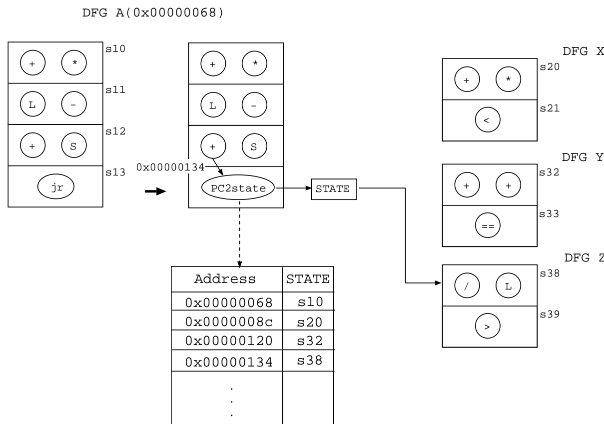

図 9: PC2STATE モジュールを利用したレジスタジャンプ

3.2.2 分岐命令と即値ジャンプ命令

即値ジャンプ命令 (j) は, 分岐先アドレスに対応する DFG への分岐に変換する. 分岐命令は, 分岐条件を計算する比較演算を行うノードを生成し, その結果に応じて次の DFG へ分岐する処理に変換する. プログラムアドレスから遷移先の DFG を特定するため, 各 DFG には元になった命令の先頭アドレスを保持しておく.

例えば図 7 に示すように, プログラム中に分岐命令である blt 命令があるとすると, 分岐先ターゲットアドレス (0x00000038) と, blt 命令自身の命令アドレス (0x00000044) で基本ブロックを分割し, それぞれを DFG に変換する. blt 命令は入力値ノードを比較する演算ノードに変換し a4 < a3 が真であるならこの DFG を入力オペランドがに対応する DFG に遷移させ, 偽であるなら次の DFG 遷移させる.

jal 命令の場合は, リターンアドレスを入力オペランドに格納する処理が含まれるので, jal 命令の次の命令のアドレスを入力オペランドの値ノードに格納する.

3.2.3 レジスタジャンプ命令

レジスタジャンプ命令は, サブルーチン呼び出しからの戻りや, C 言語の switch 文のコード生成時に用いられるが, バイナリ合成では, 分岐先が実行時の値に依存するレジスタジャンプ命令の扱いが課題となる. 例えば図 8 のようにレジスタジャンプ命令 (jr) があるとすると, 命令の引数レジスタ (a3) に渡されるレジスタの値は CDFG を生成する段階では不明であり, 静的に遷移先 DFG を決定することができない.

本稿では [3] と同様, プログラムアドレスの値を DFG の先頭状態に変換する「PC2STATE 演算」を導入することによりこの課題を解決する. PC2STATE 演算はプログラムアドレスを状態に変換する演算である. 例えば, 図 9 において, DFG A の最後にレジスタジャンプ (jr) が存在する. 図中の s10 から s39 は合成されるハードウェアの状態である. 状態機械の状態を PC2STATE の出力する状態に更新することによりレジスタジャンプと等価な遷移が実現できる. この演算は, プログラムアドレスと状態の符号の対応を計算するように合成された組み合わせ回路にバインディングする.

4. カスタム命令を含む機械語プログラムからのバイナリ合成

本稿では下記のようなカスタム命令を合成対象にする.

- R タイプの命令
- 分岐やジャンプ処理を行わない
- 固定サイクルで実行可能
- カスタム命令用は独立した演算器で実行される

カスタム命令は R タイプの命令に従うため, オペランド数は 3 個である.

カスタム命令から処理ノードに変換するには, 前章で述べた方法と同じ方法を取る. カスタム命令は分岐やジャンプ処理を行わない想定であるので, カスタム命令に一つの処理ノードを割り当てる. 合成後のハードウェアの概要を図 10 に示す. カスタム命令を含むプログラムをバイナリ合成すると合成後のハードウェアのデータパスにカスタム命令の演算器が追加される.

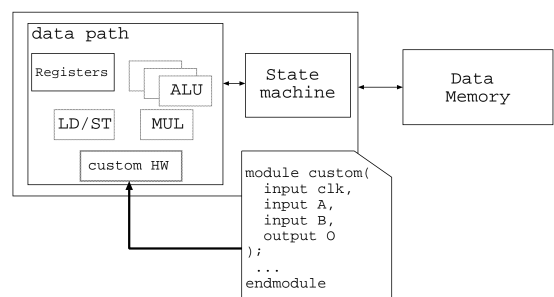

図 10: RISC-V HW

5. 実装と実験

5.1 RV32IM 命令を含むプログラムからのバイナリ合成結果

本手法に基づき, RISC-V の機械語 (RV32IM) を入力としてバイナリ合成を行うシステムを Perl5 で実装した. ただし fence 命令や ebreak, ecall 命令, コントロールステータスレジスタに関する命令は対象外である. 動作環境は Ubuntu, Mac OS X で

表 2: RV32IM 命令を含むプログラムからのバイナリ合成結果

program	#insn	RISC-V #cycle	RISC-V #LUT	RISC-V delay [ns]	本稿 #cycle	本稿 #LUT	本稿 delay [ns]
fibonacci	34	1,483			274	1,533	7.2
arith-test	70	642	3,202	14.7	273	2,665	11.6
FSM	160	777			164	2,362	13.3

Synthesizer: Xilinx Vivado (2017.4)
Target: Xilinx Artix-7 (XC7Z020-1CLG400I)

表 3: カスタム命令を含むプログラムからのバイナリ合成結果

	#cycle	#LUT	delay [ns]
with custom insn	1,013	2,170	8.8
without custom insn	3,596	2,337	8.8

Synthesizer: Xilinx Vivado (2017.4)
Target: Xilinx Artix-7 (XC7Z020-1CLG400I)

ある. スケジューリング, バインディング等は [3] で使用されているものを用いた. 合成されたハードウェアは Verilog HDL で記述される.

簡単なテストプログラムからバイナリ合成したものと, RISC-V のソフトコアである Rocket-chip(注1) との比較結果を 表 2 に示す. 論理合成は FPGA Xilinx Artix-7 (XC7Z020-1CLG400I) をターゲットに Vivado 2017.4 で行った. “#insn” は入力プログラムの命令数, “#cycle” は実行サイクル数, “#LUT” は LUT 数, “delay [ns]” はクリティカルパス遅延を表す. Rocket-chip の生成スクリプトのモードは DefaultRV32Config である. 160 命令以下のプログラムで Rocket-chip よりも小さな LUT で合成することができた. 更にサイクル数も削減できている.

5.2 カスタム命令を含むプログラムからのバイナリ合成

さらにカスタム命令を含むプログラムからもバイナリ合成を行えるようにした. C プログラム中にインラインアセンブリとしてカスタム命令を記述することによってカスタム命令の機械語を出力する. 本稿で使用する binutils (注2) のソースコード中にカスタム命令の命令名, 命令タイプ等を記述する.

合成対象のプログラムは 8bit 変数を 4 つ含む構造体を 100 個用意して, 各構造体の 4 変数に対して加算処理を行うプログラムである. 各 8bit 変数がオーバーフローする際は飽和処理を行う. カスタム命令の演算器が行う処理は 8bit 変数の加算を 4 並列での実行とオーバフロー時の処理で, この処理は独立した演算器として設計した. カスタム命令を用いた場合と用いなかった場合との比較結果を表 3 に示す. “#cycle” の括弧内はカスタム命令を適用している処理にかかるサイクル数である. 両者とも LUT 数や遅延はほとんど差異がないが, カスタム命令を使用した場合サイクル数が削減できている.

6. む す び

本稿では, RISC-V 機械語プログラムからバイナリ合成を行う手法を提案した. 本手法により RISC-V 機械語プログラムを経て生成したハードウェアは, 小さなプログラムではあるが, 性能が向上していることを確認した. また, カスタム命令とそれを実行する演算器を扱えるようにしたことで更に性能の向上を図ることが可能になった.

現在は小さなプログラムのみの実験しか行っていないが, より実用的なプログラムからバイナリ合成を行えるようにすることと, RISC-V の外部割り込みを単一のモジュールとして合成できるようにし, 外部から呼び出せるようにすること, 本稿のシステムの内部仕様を知らないユーザでも簡単にカスタム命令の演算器を設計できるような枠組みを提供することが今後の課題である.

謝　辞

本研究に関して有益な御助言を頂いた元立命館大学の中谷嵩之氏, 元関西学院大学の田村真平氏に感謝致します. 本研究は一部 JSPS 科研費 19H04081 の助成による.

文　献

[1] D.D. Gajski, N.D. Dutt, A.C-H Wu, and S.Y-L Lin: High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers (1992).
[2] D. Patterson and A. Waterman: *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon (Nov. 2017).
[3] N. Ishiura, H. Kanbara, and H. Tomiyama: “ACAP: Binary Synthesizer Based on MIPS Object Codes,” in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).
[4] 田村真平, 石浦菜岐佐, 神原弘之, 冨山宏之: “CPU 密結合型アクセラレータの機械語プログラムからの自動合成,” 信学技報, VLD2013-133 (Jan. 2014).
[5] 本田晋也, 冨山宏之, 高田広章, “システムレベル設計環境：SystemBuilder,” 信学論, Vol. J88-D-I, No. 2, pp. 163–174 (Feb. 2005).
[6] G. Stitt and F. Vahid: “Binary synthesis,” *ACM TODAES*, vol. 12, no. 3, article 34 (Aug. 2007).
[7] N. Ito, Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: “Binary synthesis implementing external interrupt handler as independent module,” in *Proc. RSP 2017*, pp. 92–98 (Oct. 2017).

(注1)：https://github.com/freechipsproject/rocket-chip
(注2)：https://github.com/riscv/riscv-gnu-toolchain