# 汎用高位合成系をバックエンドとする RISC-V 機械語からのバイナリ合成

中道 凌[1]　石浦 菜岐佐[2]　近藤 匠[1]

**概要**：本稿では, バイナリ合成系の容易な実装手法として汎用の高位合成システムをバックエンドとして利用する方法を提案し, これに基づいて RISC-V 機械語からのバイナリ合成系を実装する. 本手法ではリンク済みの機械語プログラムを入力として与え, これを実行する CPU と機能等価なハードウェアの設計記述を合成する. この際, 機械語プログラムから CDFG (control data flow graph) ではなく高位合成可能な C プログラムを生成し, これを高位合成システムの入力としてハードウェアの設計記述を合成する. 提案手法に基づいて RISC-V の RV32IM 命令セットを対象とするバイナリ合成系を実装した結果, C プログラムを高位合成したものと比べ回路規模は 1.04–3.75 倍に増加するものの, 実行サイクル数とクリティカルパス遅延はほとんど同等のハードウェアを合成することができた.

**キーワード**：システム設計技術, バイナリ合成, 高位合成, RISC-V

# Binary Synthesis from RISC-V Executable Code Using General-Purpose High-Level Synthesizer

Nakamichi Ryo[1]　Ishiura Nagisa[2]　Kondo Takumi[1]

***Abstract:*** This artcle proposes a facile way to implement binary synthesizers which utilize existing high-level synthesizers as their backends, and demonstrates its application to implement a binary synthesizer for the RISC-V instruction set. We assume a type of binary synthesizers which take a linked executable binary code as input and generate a design description of hardware which is functionally exquivalent to a CPU running the code. In our method, a C program in place of a CDFG (control dataflow graph) is generated from a binary code, which is fed into existing high-level synthesizers to produce a hardware design. In an experiment using a commercial high-level synthesizer, the execution cycles and critical path delay of the circuits, generated by our binary synthesizer from RV32IM binaries compiled from C codes, are almost the same as those of the circuits generated by the high-level synthesizer from the C codes, though the circuit size is 1.04 to 3.73 times larger.

***Keywords:*** System design technology, binary synthesis, high-level synthesis, RISC-V

## 1. はじめに

現在, 組込みシステムは高機能化が進む一方, 性能だけでなく小型化や低消費電力化の要求が益々厳しくなっている. これを解決する一手法として, ソフトウェアで行っていた処理の一部または全部を, 専用ハードウェア化する手法がある.

一般にハードウェアの設計は抽象度が低く, 開発コストが高くなる傾向がある. そこで, 効率的なハードウェア設計の手法として, 高位合成技術 (high-level synthesis) [1] を利用した設計手法が提案されている. 既存のプログラムを元に高位合成によって専用ハードウェアを合成することに

[1] 関西学院大学 大学院理工学研究科
Graduate School of Science and Technology, Kwansei Gakuin Univ.
[2] 関西学院大学 工学部
School of Engineering, Kwansei Gakuin Univ.

より, 開発コストの削減が期待できる.

しかし, 外部機器を制御するようなシステムでは, 割り込みハンドラや特殊な命令・レジスタを利用するプログラムがアセンブリやインラインアセンブリで書かれることがあり, そのような場合には高位合成をそのまま適用することができない.

バイナリ合成 [2] は高位合成と同様の技術を用いて, 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手法である. 文献 [2] は MIPS, ARM, MicroBlaze, 文献 [3] では MIPS, 文献 [4] では RISC-V の機械語からハードウェア設計記述を生成している. 文献 [5], [6] では MIPS のアセンブリで書かれた割り込みハンドラを含むプログラムをバイナリ合成によりハードウェア化している.

バイナリ合成系は命令セットアーキテクチャ (ISA) ごとに処理系が必要となる. 既開発の高位合成系やオープンソースの高位合成系 [7] が利用できる場合には, 機械語プログラムを中間表現の CDFG (control dataflow graph) に変換する処理系を作成すればバイナリ合成系が実装できる [4]. しかし, その際にもデータフロー解析の処理が必要になる等, フロントエンドの CDFG へのポーティングにも比較的大きな工数が必要になることがある.

本稿では, バイナリ合成系の容易な実装手法として汎用高位合成システムをバックエンドとして利用する方法を提案する. 機械語プログラムから高位合成可能な C プログラムを生成し, これを汎用高位合成システムによってハードウェア化する. データフロー解析, スケジューリング, バインディング等の処理は汎用高位合成系が行うため, アセンブリコードから C プログラムへの変換系を実装するだけで容易にバイナリ合成系を開発できる.

本手法に基づき RISC-V 機械語プログラムを入力とするバイナリ合成系を実装した. C プログラムをコンパイルして得た RISC-V 機械語プログラムを用いて評価実験を行った結果, 元のプログラムから高位合成を行った場合と比べて回路規模は 1.04–3.75 倍に増加するものの, 実行サイクル数とクリティカルパス遅延はほとんど同等のハードウェアを合成することができた.

## 2. 高位合成とバイナリ合成

### 2.1 高位合成

高位合成 [1] は, C 言語等の手続き型言語で書かれたハードウェアの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.

一般的な高位合成の処理の流れを図 1 に示す. まず, 与えられた動作記述を解析し, 中間表現である CDFG を生成する. CDFG に対して演算の実行のタイミングを決定するスケジューリング, 演算器やレジスタの割当を決定するバインディングを行い, レジスタ転送レベルのハードウェア設計記述 (HDL; Hardware Description Language) を生成する.

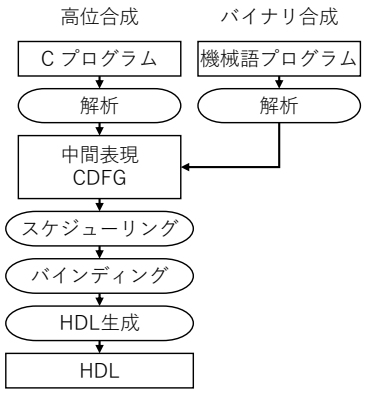


**図 1** 高位合成/バイナリ合成の流れ

C 言語等のプログラミング言語は単一のメモリ空間を備えた計算機構による実行を想定しているが, 高位合成で合成される回路の計算モデルは必ずしもこれに一致しない. このため, グローバル変数やポインタを用いて複数の関数やスレッド間でデータを共有するようなプログラムを高位合成でハードウェア化するためには, メモリアクセスを陽に扱うための書き換えが必要になる. また, 機器を制御するようなシステムは, 割り込みハンドラや特殊な命令・レジスタを利用するプログラムがアセンブリやインラインアセンブリで書かれることがあり, このようなシステムには直接高位合成を適用することができない.

### 2.2 バイナリ合成

バイナリ合成 [2] は機械語プログラムの一部または全部からハードウェア記述を合成する技術である.

文献 [2] では MIPS, ARM, MicroBlaze の機械語プログラムの指定部分をハードウェアに合成するバイナリ合成手法を提案している. また 文献 [3] は MIPS R3000 を対象に, 実行可能な機械語プログラム全体をハードウェア化する手法, および, 機械語プログラムの指定部分を CPU と密結合したコプロセッサに合成する手法を提案している.

バイナリ合成は, ソースプログラムが書かれている言語に依存することなく (あるいはソースコードが存在せず機械語プログラムのみ入手可能な場合でも) ソフトウェアをハードウェア化できる. 文献 [5], [6] では, MIPS のアセンブリやインラインアセンブリによって書かれた外部割り込みハンドラを含むシステムをバイナリ合成によりハードウェア化している. 文献 [4] では, RISC-V からのバイナリ合成系において, インラインアセンブリで書かれたカスタム命令を含むプログラムをハードウェア化している.

バイナリ合成ではプログラム中に出現するメモリアクセスをそのままハードウェア化できるため, グローバル変数やポインタを用いたプログラムもそのまま合成対象にできる. また, 機械語プログラムの解析や盗用を避ける目的でバイナリ合成により CPU とコードをハードウェアに置き

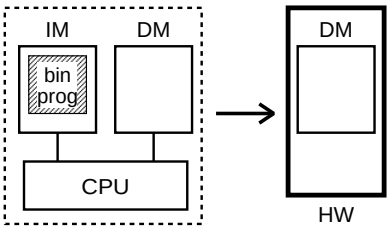


**図 2** 本稿で想定するバイナリ合成

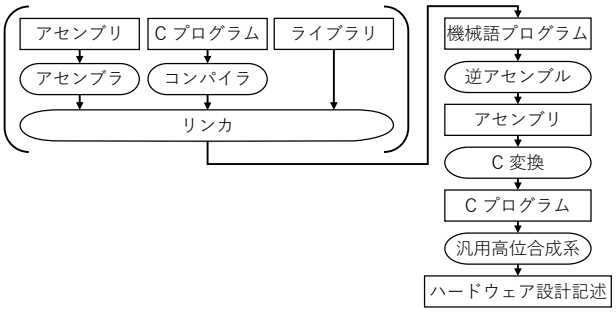


**図 3** 本手法によるバイナリ合成の流れ

換えるという用法も考えられる.

バイナリ合成系の処理の流れを図 1 に示す. 機械語プログラムから CDFG を生成すれば, 後続の処理は高位合成と共通である. 文献 [4] の RISC-V 用バイナリ合成系は, MIPS 用バイナリ合成系 [3] の CDFG 生成部だけを新規開発して実装している. このように既開発の高位合成系/バイナリ合成系やオープンソースの高位合成系が利用できる場合には CDFG をインタフェースとしてフロントエンドを実装すればバイナリ合成系が得られる. しかし, 高級言語レベルでデータフロー解析や制御解析を行なっている場合には, フロントエンドの実装にも工数を要することがある.

## 3. 高位合成系をバックエンドとするバイナリ合成

本稿では, 新しい命令セットアーキテクチャに対するバイナリ合成系を容易に実装する手法として, 機械語プログラムから CDFG ではなく高位合成可能な C プログラムを生成し, これを汎用高位合成システムの入力としてハードウェアの設計記述を合成する方法を提案する.

### 3.1 合成の流れ

本稿のバイナリ合成では, 図 2 に示すように, 命令メモリ内のプログラムとこれを実行する CPU およびデータメモリを機能的に等価なハードウェアに変換する.

図 3 に本手法によるバイナリ合成の流れを示す. 入力はリンク済みの実行可能な機械語プログラムである. これを逆アセンブルしたものから, プロセッサ上でその命令列を実行するのと機能等価な C プログラムを生成し, これを汎用高位合成系に入力して, ハードウェア設計記述を生成する. データフロー解析, スケジューリング, バインディング等の処理は汎用高位合成系が行うため, アセンブリコードから C プログラムへの変換処理系を実装するだけで容易にバイナリ合成系を開発できる.

アセンブリコードから生成する C プログラムの構成法は種々考えられるが, 本稿では図 4 に示すように 1 つのアセンブリコードから 1 つの C 言語の関数を生成する. 関数は, レジスタとメモリをローカル変数として宣言する部分 (図中 (1)(2)) と, 命令列を C 言語のコードに変換した部分 (図中 (3)) から成る.

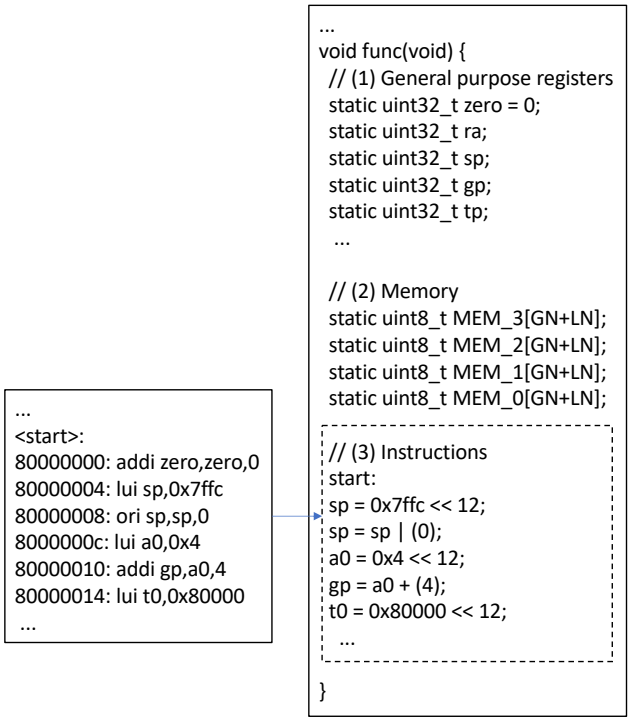


**図 4** 本手法で生成する C プログラムの構成

### 3.2 ALU 演算命令と符号の扱い

レジスタはレジスタと同じビット長の符号なし整数型のローカル変数に変換する. 図 4 の (1) は 32 ビットのレジスタの変換例であり, `uint32_t` はバックエンドの高位合成系が 32 ビット符号なし整数として扱う C 言語の型になるように定義する.

加減算や論理演算, シフト演算を行う命令は, 各命令の動作を C 言語の文で表現する. RISC-V の 32 ビット ALU 命令の変換例を表 1 に示す. 例えば “`add rd,rs1,rs2`” という命令は “`rd = rs1 + rs2;`” という C 言語の文に変換すればよい.

レジスタを符号なし整数型の変数で表現しているため, 命令がオペランドを符号付き整数として扱っている場合には C プログラム中に型変換 (キャスト) が必要になる. しかし, 2 の補数表現を用いる CPU であれば, 符号なし整数として演算を行っても計算結果は一致する. 例えば, “`addi sp,sp,-32`” は “`sp = sp + (-32);`” に変換すれば, C 言語の型変換で $-32$ は 4294967264 に変換されて `sp` に加算

```
#define SINT8(u)  ((int8_t)((u) & 0x7f) + (((u) & 0x80) ? -128 : 0))
#define SINT16(u) ((int16_t)((u) & 0x7fff) + (((u) & 0x8000) ? -32768 : 0))
#define SINT32(u) ((int32_t)((u) & 0x7fffffff) + (((u) & 0x80000000) ? -2147483648 : 0))
```

**図 5** 符号なしから符号付きへの変換例

**表 1** ALU 演算命令の変換例

| 命令 | C プログラム |
|---|---|
| add a5,a4,a5 | a5 = a4 + a5; |
| addi sp,sp,-32 | sp = sp + (-32); |
| and a4,a4,a5 | a5 = a4 & a5; |
| sll a5,a5,a0 | a5 = a5 << (a0 & 31); |
| sra a6,a4,a6 | a6 = SINT32(a4) >> (a6 & 31); |
| srl a7,a4,a7 | a7 = a4 >> (a7 & 31); |

され, 結果の下位 32 ビットが sp に代入されるが, 得られる表現を符号付き整数と見れば, sp の値に −32 を加算した結果に等しくなる.

しかし, 算術シフト命令 (表 1 の sra) のように, 必ず C 言語の符号付き整数型の演算を用いなければならない場合には, 型変換 (キャスト) が必要になる. 多くの C コンパイラでは, 符号なし整数を同精度の符号付き整数変数に代入した場合には単純にコピーを行うコードを生成する. しかし, 最上位ビットが 1 となっている符号なし整数を同精度の符号付き整数型に変換すると, C 言語の仕様上は未定義動作となり結果は保証されない. そこで安全のため, 図 5 に示すマクロ (SINT32, SINT16, SINT8) を用いて未定義動作のない型変換を行うようにする. [*1]

### 3.3 乗算命令

計算の中間結果が汎用レジスタのビット数を超える場合は, ビット数に応じた整数型を用いる. RISC-V の 32 ビット乗算命令の変換例を表 2 に示す.

- mul 命令は符号なし 32 ビットレジスタ rs1, rs2 の乗算結果の下位 32 ビットをレジスタ rd に格納する命令である. C 言語の 32 ビット符号なし整数型の演算では結果の下位 32 ビットだけが残るので, 単に “rd = rs1 * rs2;” という文に変換すればよい.
- mulh 命令は符号付き 32 ビットレジスタレジスタ rs1, rs2 の乗算結果の上位 32 ビットをレジスタ rd に格納する命令である. これは rs1 と rs2 を符号付き 32 ビット整数型に変換した後, 符号付き 64 ビット整数型に拡張して計算を行い, 結果を 32 ビット右シフトしたものを rd に代入すればよい.
- mulhu 命令は符号なし 32 ビットレジスタレジスタ rs1, rs2 の乗算結果の上位 32 ビットをレジスタ rd に格納する命令である. これは rs1 と rs2 を符号なし 64 ビット整数型に拡張して乗算を行った結果を 32 ビット右シフトすればよい.
- mulhsu 命令は符号付き 32 ビットの rs1 と 符号なし 32 ビットの rs2 の乗算結果の上位 32 ビットをレジスタ rd に格納する命令である. これは 両オペランドを符号付き 64 ビット整数型に拡張して乗算を行い, 結果を 32 ビット右シフトすればよい.

[*1] LLVM では最適化によってこれが単純なコピーにコンパイルされる.

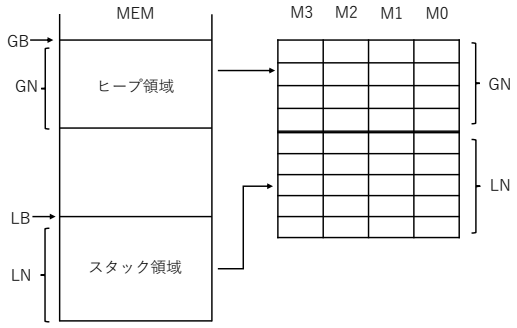


**図 6** メモリの扱い

```
#define LB 0x7ffbfec
#define GB 0xc0000000
#define MA_G(a) ( (((a) - GB) / 4) )
#define MA_L(a) ( (((a) - LB) / 4) + GN )
#define MA(a) ( ((a) >= GB) ? MA_G(a) : MA_L(a) )
```

**図 7** メモリアドレスのインデックスへの変換例

### 3.4 ロード/ストア命令

本稿では, バイト単位でアドレッシングが可能で, かつ主としてワード単位でアクセスが行われるメモリを想定し, 図 4 のように 1 バイトデータの配列変数をワードのバイト数だけ用意して記憶する.

配列変数は必要なサイズだけを用意し, CPU がアクセスするアドレスを配列のインデックスに変換する. 例えば図 6 のように, スタック領域の開始番地と必要サイズがそれぞれ LB, LN (ワード), ヒープ領域 (グローバル領域) の開始番地と必要サイズがそれぞれ GB, GN (ワード) の場合, 合計 LN+GN ワード分の配列を用意し, メモリのアドレスがどちらの領域のものかに応じてその開始番地を減じることによりアドレスをインデックスに変換する. 変換のためのマクロの例を図 7 に示す. MA(a) は, アドレス a に対応するインデックスを与える.

複数バイトのロード/ストアは, 複数バイトデータの並列読み書きに変換可能な C コードに変換する. 変換例を表 3 に示す. “lw”, “lh”, “lb” はそれぞれ 4 バイト, 2 バイト, 1 バイトのロード, “sw”, “sh”, “sb” はそれぞれ 4 バイト, 2 バイト, 1 バイトのストア命令である.

**表 2** 乗算命令の変換例

| 命令 | C プログラム |
|---|---|
| mul rd,rs1,rs2 | rd = rs1 * rs2; |
| mulh rd,rs1,rs2 | rd = ((int64_t)SINT32(rs1) * (int64_t)SINT32(rs2)) >> 32; |
| mulhu rd,rs1,rs2 | rd = ((uint64_t)rs1 * (uint64_t)rs2) >> 32; |
| mulhsu rd,rs1,rs2 | rd = ((int64_t)SINT32(rs1) * (int64_t)rs2) >> 32; |

**表 3** ロード/ストア命令の変換例

| 命令 | C プログラム |
|---|---|
| lw rd,offset(rs) | rd = (MEM_3[MA(rs+offset)] << 24)<br>+ (MEM_2[MA(rs+offset)] << 16)<br>+ (MEM_1[MA(rs+offset)] << 8)<br>+ (MEM_0[MA(rs+offset)]); |
| lh rd,offset(rs) | rd = (MEM_1[MA(rs+offset)] << 8)<br>+ (MEM_0[MA(rs+offset)]); |
| lb rd,offset(rs) | rd = (MEM_0[MA(rs+offset)]); |
| sw rs2,offset(rs1) | MEM_3[MA(rs1+offset)] = (rs2 >> 24);<br>MEM_2[MA(rs1+offset)] = (rs2 >> 16);<br>MEM_1[MA(rs1+offset)] = (rs2 >> 8);<br>MEM_0[MA(rs1+offset)] = (rs2 & 255); |
| sh rs2,offset(rs1) | MEM_1[MA(rs1+offset)] = (rs2 >> 8);<br>MEM_0[MA(rs1+offset)] = (rs2 & 255); |
| sb rs2,offset(rs1) | MEM_0[MA(rs1+offset)] = (rs2 & 255); |

### 3.5 制御移転命令

分岐命令は分岐条件を計算する文と goto 文に変換する．blt (branch if less than) 命令の変換例を図 8 に示す．分岐条件が成立すれば, 分岐先アドレスに対応するラベルの文にジャンプする．これを実現するため, 前処理で関数の先頭や分岐先に指定されているアドレスを列挙し, これらのアドレスにある命令に対応する文にラベルを付与するようにする．

本稿では, 1 つのアセンブリコードから 1 つの C 言語の関数を生成するので, サブルーチンの呼び出しと戻りについても goto 文を用いて書き換えられる．

呼び出しに用いられるジャンプ命令は, 仕様に従って戻り番地を記憶した後にジャンプするようにすればよい．jal (jump and link) 命令の変換例を図 9 に示す．ra に次の命令のアドレスを戻り番地として格納し, アドレス 800000d8 に対応するラベル L800000d8 にジャンプしている．

サブルーチンからの戻りに用いられるレジスタジャンプ命令は分岐先が実行時の値に依存するため, ジャンプ先の候補を前処理で列挙しておき, 条件分岐文によって分岐先が選択されるようにする．jalr (jump and link register) 命令の変換例を図 10 に示す．レジスタ ra にオフセット (図においては 0) を加えたアドレスに対応するラベルにジャンプする．

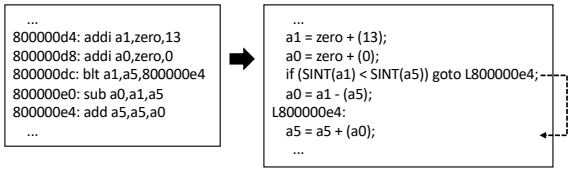


**図 8** 分岐命令の変換例

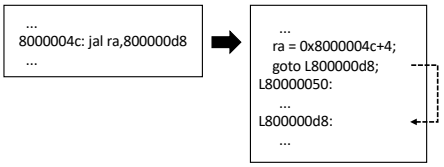


**図 9** ジャンプ命令の変換例

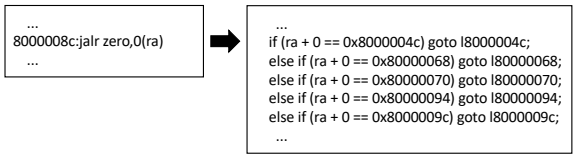


**図 10** レジスタジャンプ命令の変換例

## 4. 実装と実験

提案手法に基づき, RISC-V 機械語プログラムを入力とするバイナリ合成系を実装した．対象とした命令セットは RV32IM であり, 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を除く全 55 命令中 45 命令を処理可能とした．バックエンドの高位合成系としては Vivado HLS (2020.01) を用い, アセンブリを C プログラムに変換する処理系は Python 3.7.7 で実装した．

実装したシステムのテストは, C プログラムを入力とする高位合成系と本手法のバイナリ合成系でそれぞれ回路を生成し, それらのシミュレーション結果が一致するかを確認することにより行った．テスト手法を図 11 に示す．まず, 合成対象の関数とテストベンチを用意する．合成対象の関数はこのまま高位合成に入力することができ, テストベンチを用いてハード/ソフト協調シミュレーション (Co-Simulation) を行って “result_hls” を得る．一方, バイナリ合成は合成対象の関数は関数名を “main” へと書き換えてコンパイルすることによりリンク済みの機械語プログラムを得る．この機械語プログラムを入力として本手法によるバイナリ合成を行い, Co-Simulation によって “result_bs” を得る．“result_hls” と “result_bs” を照合することにより動作確認が行える．

6 本の C プログラムから得られる RISC-V 機械語プログラムに対して本手法でバイナリ合成を行った．コンパイラには riscv32-unknown-elf をターゲットとする gcc-10.2.0 を用い, 最適化オプションには –O3 を指定した．結果を表 4 に示す．“binary_search” は整数配列に対して二分探

**表 4** 合成結果

| | HLS | | | | | 本手法 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | FF | LUT | DSP | cycle | delay [ns] | 命令数 | FF | LUT | DSP | cycle | delay [ns] |
| binary_search | 164 | 405 | 0 | 26 | 6.720 | 38 | 230 | 523 (1.31) | 0 | 21 (0.80) | 5.924 |
| bubble_sort | 112 | 190 | 0 | 90,300 | 6.514 | 37 | 170 | 639 (3.36) | 0 | 90,002 (1.00) | 6.514 |
| heap_sort | 247 | 544 | 0 | 11,300 | 7.040 | 103 | 1,152 | 2,031 (3.73) | 0 | 10,416 (0.92) | 6.830 |
| lcm | 1,419 | 1,363 | 3 | 211 | 8.470 | 39 | 1,526 | 1,413 (1.04) | 3 | 216 (1.02) | 8.470 |
| prime | 464 | 499 | 0 | 2,679 | 8.250 | 29 | 546 | 534 (1.07) | 0 | 2,443 (0.91) | 6.880 |
| FSM | 2,917 | 4,639 | 64 | 257 | 8.470 | 221 | 3,757 | 4,975 (1.07) | 67 | 264 (1.03) | 8.470 |

GCC optimization: –O3, High-Level Synthesizer: Xilinx Vivado (2020.1), Target: Xilinx Artix-7 (xc7a100tcsg324-1)

**表 5** FSM において状態数を変化させた結果

| | 高位合成 | | | | | 本手法 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 状態数 | FF | LUT | DSP | cycle | delay [ns] | 命令数 | FF | LUT | DSP | cycle | delay [ns] |
| 16 | 1,639 | 2,379 | 32 | 273 | 8.470 | 127 | 1,708 | 2,505 | 34 | 274 | 8.470 |
| 32 | 3,078 | 4,679 | 64 | 266 | 8.470 | 221 | 3,757 | 4,975 | 67 | 264 | 8.470 |
| 64 | 5,798 | 9,364 | 128 | 257 | 8.470 | 423 | 6,708 | 9,117 | 194 | 256 | 8.470 |
| 128 | 11,822 | 18,732 | 256 | 258 | 8.644 | 809 | 10,826 | 16,512 | 258 | 258 | 8.644 |
| 256 | 24,350 | 37,302 | 512 | 257 | 8.670 | 1,645 | 24,807 | 30,121 | 770 | 257 | 8.670 |
| 512 | 60,810 | 72,386 | 1,024 | 258 | 8.470 | 3,627 | 53,983 | 66,440 | 1,026 | 257 | 8.470 |

**表 6** 最適化オプションを変化させたバイナリ合成結果の比較

(a) binary_search

| | 命令数 | FF | LUT | cycle | delay [ns] |
|---|---|---|---|---|---|
| HLS | | 164 | 405 | 22 | 6.720 |
| –O0 | 65 | 407 | 1,530 | 104 | 8.661 |
| 本手法 –O1 | 37 | 264 | 625 | 29 | 7.557 |
| –O2 | 42 | 198 | 530 | 21 | 7.113 |
| –O3 | 38 | 230 | 523 | 21 | 5.924 |

(b) heap_sort

| | 命令数 | FF | LUT | cycle | delay [ns] |
|---|---|---|---|---|---|
| HLS | | 247 | 544 | 11,300 | 7.040 |
| –O0 | 159 | 3,553 | 9,881 | 64,956 | 8.661 |
| 本手法 –O1 | 92 | 3,299 | 4,335 | 13,106 | 8.661 |
| –O2 | 84 | 3,441 | 4,790 | 15,365 | 8.591 |
| –O3 | 103 | 1,152 | 2,031 | 10,416 | 6.830 |

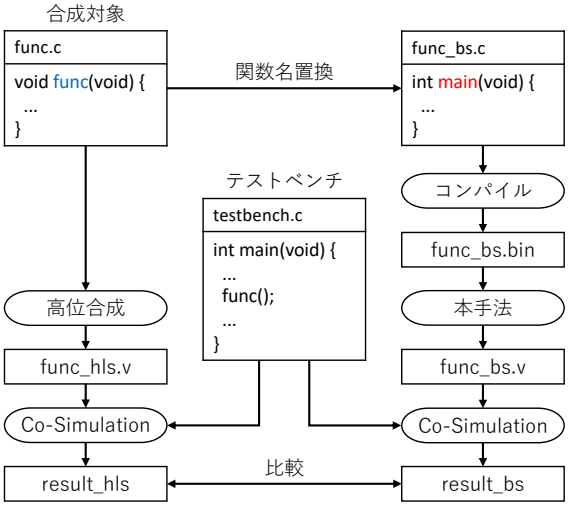


**図 11** 実装したシステムの動作概要とテスト手法

索を行うプログラム, “bubble_sort”, “heap_sort” は整数配列をそれぞれバブルソート, ヒープソートでソートするプログラム, “lcm” は 2 整数の最小公倍数を求めるプログラム, “prime” は素数判定を行うプログラム, “FSM” はランダムに生成した状態遷移機械 (32 状態で, 遷移先の計算に剰余演算を用いている) で 128 回の状態遷移を行うプログラムである. “HLS” は参考のために元の C プログラムを Vivado HLS で直接高位合成した結果である. “FF”, “LUT”, “DSP” はそれぞれの使用数, “cycle” は実行サイクル数, “delay” はクリティカルパス遅延 (ns) を示す. “命令数” は機械語プログラムの命令数である.

回路規模 (FF 数と LUT 数) は, “bubble_sort” や “heap_sort” で 3 倍以上の増加が見られるが, それ以外では高位合成と同程度である. 実行サイクル数やクリティカルパス遅延はすべてのプログラムで高位合成と同程度となった. “bubble_sort” や “heap_sort” で回路規模が増大した原因は現在調査中である.

表 6 は “binary_search” と “heap_sort” について, 機械語プログラムを得る際のコンパイラの最適化オプションを –O0, –O1, –O2, –O3 と変化させて実験した結果である. 最適化を行わない –O0 に比べて –O1, –O2, –O3 で回路規模と実行サイクル数が減少しているが, 最適化の強度が必ずしも回路の規模や性能の改善に直結しているわけではない. これは, CPU での実行に適した最適化が必ずしもバックエンドの高位合成における最適化にマッチしないためと考え

られる.

また, 機械語プログラムの規模 (命令数) と回路規模の関係を見るために, 状態遷移を行うプログラムの状態数を変化させて合成を行った. 結果を表 5 に示す. 遷移回数はすべて 128 回である. 命令数にほぼ比例して回路規模や演算器数は増加するが, 実行サイクル数やクリティカルパス遅延が著しく増加することはなかった.

## 5. おわりに

本稿ではバックエンドとして汎用高位合成システムを利用したバイナリ合成システムの容易な実装手法を提案した. 高位合成に比べて回路規模が増加することはあるが, 実行サイクル数やクリティカルパス遅延は同程度のハードウェアを合成することができた.

今回の方式では動作確認と高位合成との比較を行うためにデータメモリをハードウェア内部に組み込んだが, ヒープ (グローバル) 領域のデータは外部のメモリとして扱う必要があると考えている. また, レジスタジャンプやメモリアクセスの効率化を行う等, 高位合成の入力に適した C プログラムの生成が今後の課題として挙げられる.

**謝辞** 本研究に関して有益な御助言を頂いた京都高度技術研究所の神原弘之氏, 立命館大学の冨山宏之教授, 元立命館大学の中谷嵩之氏, 元関西学院大学の浜名将輝氏に感謝致します. また, 本研究に関してご協力, ご討議頂いた関西学院大学石浦研究室の諸氏に感謝いたします. 本研究は一部 JSPS 科研費 19H04081 の助成による.

### 参考文献

[1] Gajski, D. D., Dutt, N. D., Wu, A. C. and Lin, S. Y.: High—Level Synthesis: Introduction to Chip and System Design, Springer Science & Business Media (2012).
[2] Stitt, G. and Vahid, F.: Binary synthesis, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 12, No. 3, pp. 1–30 (2008).
[3] Ishiura, N., Kanbara, H. and Tomiyama, H.: ACAP: Binary synthesizer based on MIPS object codes, *Proc. ITC-CSCC 2014*, pp. 725–728 (2014).
[4] 浜名将輝, 石浦菜岐佐: RISC-V 機械語プログラムからのバイナリ合成, 研究報告システムと LSI の設計技術 (SLDM), Vol. 2020, No. 19, pp. 1–5 (2020).
[5] 伊藤直也, 石浦菜岐佐, 冨山宏之, 神原弘之: 外部割込みのハンドラを含むプログラムからの高位合成, DA シンポジウム 2014 論文集, Vol. 2014, pp. 121–126 (2014).
[6] Ito, N., Oosako, Y., Ishiura, N., Kanbara, H. and Tomiyama, H.: Binary synthesis implementing external interrupt handler as independent module, *2017 International Symposium on Rapid System Prototyping (RSP)*, IEEE, pp. 92–98 (2017).
[7] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S. D. and Anderson, J. H.: LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems, *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 13, No. 2, pp. 1–27 (2013).