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
VLD2023-95,RECONF2023-98(2024-01)

汎用高位合成系を用いたバイナリ合成における外部メモリアクセスの実装

岸本　　匠†　石浦菜岐佐†

† 関西学院大学　〒669–1330 兵庫県三田市学園上ケ原 1

あらまし　本稿では, 汎用高位合成系を用いたバイナリ合成における外部メモリアクセスの実装を提案する. バイナリ合成は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り, アセンブリプログラムやインラインアセンブリを含むプログラムからのハードウェア合成が可能である. 中道らはバイナリ合成系の容易な実装法として, 機械語プログラムを C プログラムに変換し, これを汎用の高位合成によりハードウェア化する手法を提案している. しかしこの手法ではデータメモリを合成したハードウェアの内部に実装しているため, 他のハードウェアとのメモリ共有や, メモリマップト I/O に対応できないという課題がある. 本稿では中道らのバイナリ合成手法において, ポートを介した外部メモリへのアクセスを可能にするとともに, 共有領域のみを外部メモリに実装する手法や, メモリマップト I/O を実装する手法を提案する. 本手法に基づくバイナリ合成法で RISC-V 機械語プログラムからハードウェアを合成した結果, 内部にメモリを実装した場合と比べ, 回路規模, 実行サイクル数とも大きなオーバヘッドなく外部メモリアクセスとメモリマップト I/O を実現できた.

キーワード　バイナリ合成, 高位合成, RISC-V, ISA, 組込みシステム

Implementation of External Memory Access for Binary Synthesis Using General-Purpose High-Level Synthesizer

Sho KISHIMOTO† and Nagisa ISHIURA†

† Kwansei Gakuin University,　1 Uegahara, Gakuen, Sanda, Hyogo, 669-1330, Japan

Abstract　In this article, we present a method for implementing external memory access within the context of binary synthesis utilizing commercial high-level synthesis systems. Binary synthesis, which translates binary program codes into hardware designs, enables synthesis of hardware from programs using assembly or inline assembly. Nakamichi et al. has proposed an approach for facilitating implementation of binary synthezers, in which binary programs are once translated into C programs and then processed by a high-level synthezer. However, binary synthesizers developed so far using this method embed memory within the synthesized hardware, thereby impeding data sharing among various hardware components and memory-mapped I/O. This paper aims to enhance Nakamichi's method to enable external memory access through I/O ports of synthesized hardware, facilitating memory-mapped I/O as well. A binary synthesizer, implemented based on the proposed method, demonstrates that external memory access and memory-mapped I/O are achievable without incurring significant overhead in terms of circuit size and execution cycle count.

Key words　Binary sythesis, High-level synthesis, RISC-V, ISA, Embedded systems

1. はじめに

現在, 組み込みシステムには益々高い機能が求められる一方で, 小型化や低消費電力等の厳しい資源節約が求められる. これを解決する手法の一つとして, ソフトウェアで行っていた処理の一部または全てをハードウェア化する手法がある. しかし一般にハードウェア開発のコストはソフトウェアよりも高くなる. 高位合成技術 (high-level synthesis) [1] は C 言語等のプログラムからハードウェア設計記述を生成する技術であり, これを利用したマイグレーションが開発効率化の一手法として考えられる.

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・レジスタへのアクセスが必要になる. このようなプログラムはアセンブリやインラインアセンブリで記述されるため, これらをそのまま高位合成の入力とすることは

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技術であり, 高位合成では扱えなかったアセンブリやインラインアセンブリで書かれたプログラムも, ハードウェア化が可能である. 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.

しかし, バイナリ合成の処理系は命令セットアーキテクチャ毎に開発する必要がある. これに対し文献 [7] は汎用高位合成系を利用したバイナリ合成システムの低コストな実装手法を提案している. 機械語プログラムを C プログラムに変換し, これを高位合成系への入力とするというものである.

しかし, 文献 [7] の実装ではデータメモリをハードウェアの内部に配置することを想定しているため, メモリを介してデータを他の CPU やハードウェアと共有できない. また, メモリマップト I/O に対応できないという課題がある.

本稿では, 汎用高位合成系を利用したバイナリ合成における外部メモリアクセスの実装を提案する. ポートを介した外部メモリへのアクセスを可能にするとともに, 共有領域のみを外部メモリに配置する手法や, メモリマップト I/O を実装する.

本手法に基づくバイナリ合成法で RISC-V 機械語プログラムをバイナリ合成した結果, 内部にメモリを実装した時と比べ, 回路規模, 実行サイクル数とも大きなオーバヘッドなく外部メモリアクセスとメモリマップト I/O を実現できた.

2. 汎用高位合成系を用いたバイナリ合成

2.1 バイナリ合成

バイナリ合成 [2] は機械語プログラムからレジスタ転送レベルのハードウェア設計記述を合成する技術である.

バイナリ合成は機械語に翻訳されて実行されるものであれば言語に依存しないため, アセンブリやインラインアセンブリで書かれたプログラムも合成対象とすることができる. 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.

バイナリ合成はプログラム中のメモリアクセスを行う記述も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ために, 機械語プログラムを CPU とともにハードウェア化してしまうという用途も考えられる.

バイナリ合成系の処理の流れを図 1 に示す. バイナリ合成の処理 (b) は高位合成 (a) と大部分が同じである. 高位合成は C 言語等のプログラミング言語で書かれた動作記述を解析して得られる中間表現 CDFG (control dataflow graph) に対して, 演算の実行タイミングを決めるスケジューリングや, 演算や値を演算器やレジスタに割り当てるバインディングを行い, RTL (register transfer level) の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は機械語プログラムを入力として CDFG を生成するが, 以降の処理は高位合成と同じである.

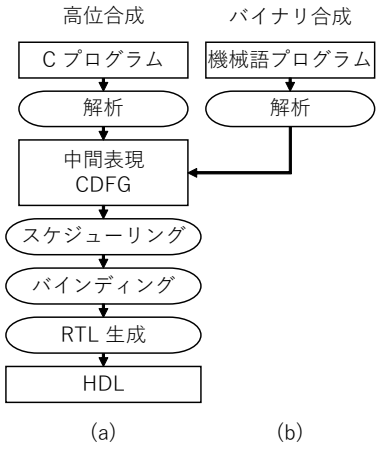


図 1 高位合成/バイナリ合成の流れ

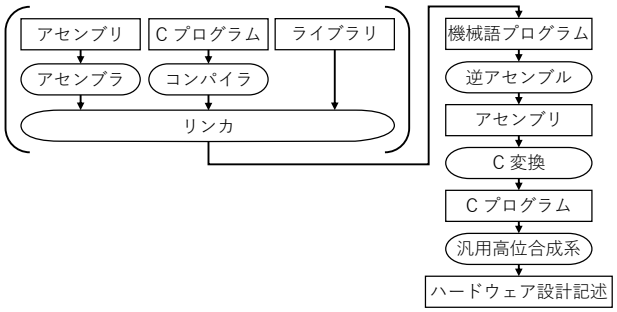


図 2 文献 [7] のバイナリ合成の流れ

バイナリ合成系は命令セットアーキテクチャ毎に処理系の開発が必要になる. 一般に高位合成系に CDFG を入力するインタフェースは公開されていないため, スケジューリングやバインディング等の処理部分も実装する必要がある. 自ら開発した高位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は, 機械語プログラムから CDFG を生成するまでの部分だけを新たに実装すればよい. 文献 [4] の RISC-V を対象としたバイナリ合成系は, MIPS を対象としたバイナリ合成系 [3] の CDFG 生成部分のみを新たに開発して実装している. しかしこの場合にも, データフロー解析や制御解析等が必要になり, CDFG 生成部の実装にも応分の工数が必要になる.

2.2 汎用高位合成系を用いたバイナリ合成

2.2.1 合成の流れ

文献 [7] 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, これを一般的な高位合成系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している.

文献 [7] のバイナリ合成の流れを図 2 に示す. 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て得られるアセンブリプログラムから C プログラムを生成

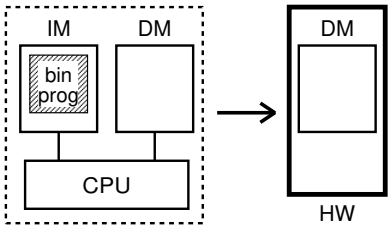


図 3　文献 [7] のバイナリ合成が生成するハードウェア構成

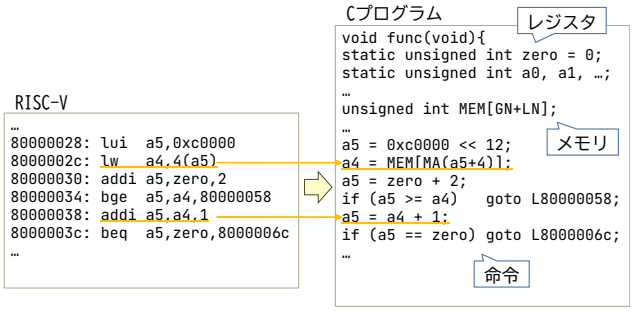


図 4　アセンブリから C プログラムへの変換例 [7]

し, それを高位合成の入力とする. これにより, 機械語プログラムを C プログラムへ変換するシステムを実装するだけでバイナリ合成系が開発できる.

文献 [7] のバイナリ合成では, 図 3 に示すように, 命令メモリ (IM) 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 (DM) を機能的に等価なハードウェアに変換している.

2.2.2　命令の変換

アセンブリコードから生成する C プログラムの構成法として, 文献 [7] では, 図 4 のように 1 つのアセンブリコードから 1 つの C 言語の関数を生成している. 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, および命令列を C 言語の文に変換した部分から成る.

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で表現する. 32 ビットの RISC-V (RV32I) の場合であれば, 図 4 のように汎用レジスタを 32 ビットの符号なし整数型のローカル変数として宣言する. メモリは後述する通りローカル配列で表現する.

加減算や論理演算, シフト演算を行う命令は, 図 4 のように, 各命令の動作を表す C 言語の文に変換する. 例えば “`addi a5,a4,1`” という命令は “`a5 = a4 + 1;`” という C 言語の文に変換する.

分岐命令は分岐条件を計算する文と goto 文に変換する. 例えば, 図 5 のように, “`blt a1,a5,800000e4`” という命令は, 分岐条件が成立すれば, 分岐先アドレスに対応する文にジャンプする文に変換する. 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.

レジスタジャンプ命令は, 分岐先が実行時にしか確定しないため, ジャンプ先の候補をあらかじめ列挙しておき, switch 文によって分岐アドレスに対応した文に分岐させる. 例えば, 図 6 のように, “`jalr zero,0(ra)`” という命令は, 分岐先アドレスのハッシュ値により分岐先を選択する文に変換する. 一旦 “`regjump`” に分岐させているのは, レジスタジャンプの処理を集約して回路規模を削減するためである.

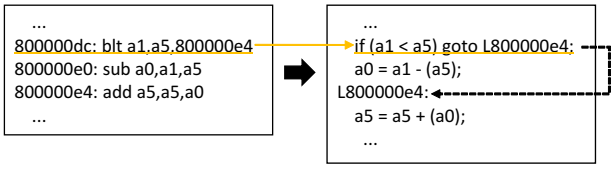


図 5　分岐命令の変換 [7]

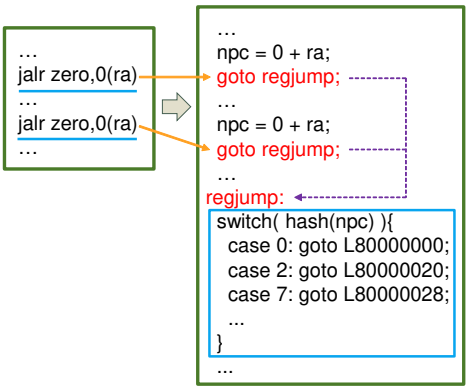


図 6　レジスタジャンプ命令の変換 [7]

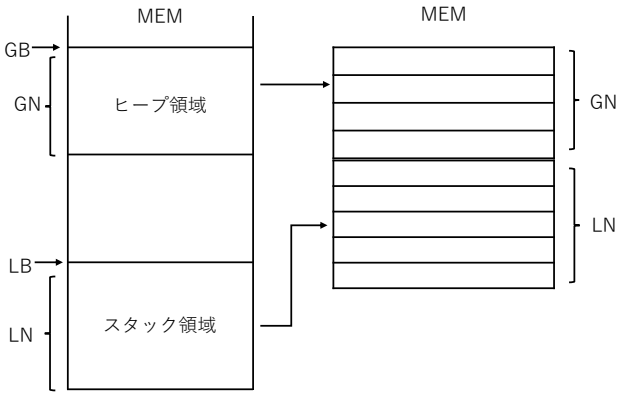


図 7　メモリ領域の集約 [7]

```
#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 8　メモリアドレスの要素番号への変換 [7]

2.2.3　ロード/ストア命令

データメモリは関数内のローカル配列で表現する. 図 7 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), 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アドレスを要素番号に変換する.

変換のためのマクロの例を図 8 に示す. `MA(a)` は, アドレス `a` に対応する要素番号を与える. アクセスされるメモリアドレス (a) が共有領域の開始アドレス (GB) より大きければ共有領域に

表 1　ロード/ストア命令の変換 [7]

命令	off	C プログラム
lw rd,offset(rs)	0	rd = (MEM[MA(rs+offset)]);
lh rd,offset(rs)	0	rd = (MEM[MA(rs+offset)] » 0) & 0xFFFF;
	1	rd = (MEM[MA(rs+offset)] » 16) & 0xFFFF;
lb rd,offset(rs)	0	rd = (MEM[MA(rs+offset)] » 0) & 0xFF;
	1	rd = (MEM[MA(rs+offset)] » 8) & 0xFF;
	2	rd = (MEM[MA(rs+offset)] » 16) & 0xFF;
	3	rd = (MEM[MA(rs+offset)] » 24) & 0xFF;
sw rs2,offset(rs1)	0	MEM[MA(rs1+offset)] = rs2;
sh rs2,offset(rs1)	0	MEM[MA(rs1+offset)] = (MEM[MA(rs1+offset)] & 0xFFFF0000) | (rs2 « 0);
	1	MEM[MA(rs1+offset)] = (MEM[MA(rs1+offset)] & 0x0000FFFF) | (rs2 « 16);
sb rs2,offset(rs1)	0	MEM[MA(rs1+offset)] = (MEM[MA(rs1+offset)] & 0xFFFFFF00) | (rs2 « 0);
	1	MEM[MA(rs1+offset)] = (MEM[MA(rs1+offset)] & 0xFFFF00FF) | (rs2 « 8);
	2	MEM[MA(rs1+offset)] = (MEM[MA(rs1+offset)] & 0xFF00FFFF) | (rs2 « 16);
	3	MEM[MA(rs1+offset)] = (MEM[MA(rs1+offset)] & 0x00FFFFFF) | (rs2 « 24);

対応した要素番号 (MA_G(a)) に, アドレスが共有領域の開始アドレスより小さければ, 非共有領域に対応した要素番号 (MA_L(a)) に変換する.

1 ワードのロード/ストア命令は, 配列変数を用いて 1 ワードデータを格納する. 1 ワードに満たないデータのロード/ストア命令は, 読み書きするデータをオフセットに応じて変換する. RV32I のロード/ストア命令の変換例を表 1 に示す. 例えば 1 ワードのロード命令 “lw rd,offset(rs)” の場合, “rd = (MEM[MA(rs+offset)])” という代入文へ変換する.

文献 [7] の手法ではデータメモリはハードウェアの内部にあることを想定しているため, 他のハードウェアや CPU とメモリを共有できない. また, 外部機器を制御するメモリマップト I/O に対応できないという課題がある.

3. 外部メモリアクセスの実装

本研究では汎用高位合成系を用いたバイナリ合成において, 外部に配置したメモリへのアクセスを実装する. これにより, メモリを介した他の CPU やハードウェアとのデータ共有を可能にする. また, データメモリのうち, 共有領域のみを外部に実装する手法およびメモリマップト I/O の実現手法も提案する.

3.1 ポートを介した外部メモリアクセスの実装

メモリをハードウェア外部に配置する構成を図 9 に示す. 従来 (a) のようにハードウェアの内部に配置していたメモリを, 外部に配置し, ポートとインタフェースを介してデータをやりとりする構成 (b) にする.

この変更は機械語プログラムから生成する C プログラムにおいて, メモリを表現する配列の宣言を変更することにより行う. 例えば高位合成ツールの Vivado HLS では, 記述を図 10 のように変更すればよい. 従来は (a) の 7 行目のようにメモリを関数内の配列として記述していたものを, (b) の 4 行目のように関数の引数として記述することによりポートとして扱われるようにする. 5 行目のプラグマはこのポートに対するメモリインタフェースを指定するものである. これ以外, ロード/ストア命令の変換は従来手法と同じである.

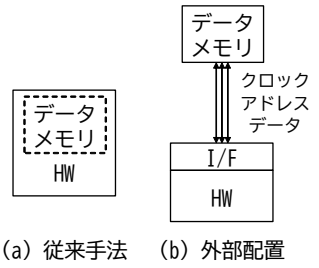


図 9　データメモリの外部配置

```
1  void func(
2   unsigned int target,
3   unsigned int *res){
4    static unsigned int zero = 0;
5    static unsigned int a0, a1, …;
6    …
7    unsigned int MEM[GN+LN];
8    …
9    a4 = MEM[MA(a5+4)];
10   …
11   MEM[MA(sp+8)] = s0;
12   …
13 }
```

```
1  void func(
2   unsigned int target,
3   unsigned int *res,
4   unsigned int MEM[GN+LN]){
5  #pragma HLS interface bram port=MEM
6    static unsigned int zero = 0;
7    static unsigned int a0, a1, …;
8    …
9    a4 = MEM[MA(a5+4)];
10   …
11   MEM[MA(sp+8)] = s0;
12   …
13 }
```

図 10　メモリを外部に配置する記述例

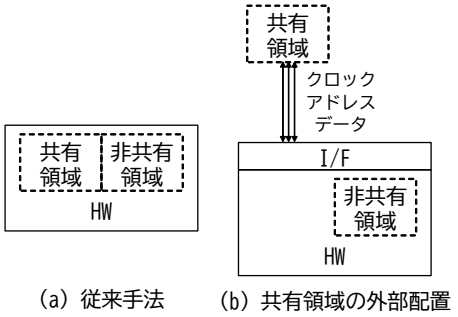


図 11　共有領域の外部配置

3.2 共有領域のみの外部実装

メモリは全て外部に置くだけでなく, 図 11 のように, 共有領域のみを外部に置くこともできる. 従来はデータメモリを全て内部に実装していたが (a), 共有領域を外部に配置し非共有領域を内部に実装する構成 (b) にする. ここで, 共有領域とはデータメモリのうち他の CPU やハードウェアと共有される可能性のあるグローバル領域を指し, 非共有領域はスタック領域等のように合成対象のハードウェアだけが使用する領域を指す.

例えば Vivado HLS では, メモリを共有領域と非共有領域に分

```
void func(int target, int *res, unsigned int MEM_EX[GN]){
#pragma HLS interface bram port=MEM_EX
static unsigned int zero = 0;
static unsigned int a0, a1, …;
…
static unsigned int MEM_IN[LN] = {};
…
}
```

図 12　メモリを共有領域と非共有領域に分けた記述例

```
#define MA_L(a) ( (((a) - LB) / 4) + GN )
#define MA_G(a) ( (((a) - GB) / 4) )
#define MA(a) ( ((a) >= GB) ? MA_G(a) : MA_L(a) )
...
t4 = MEM[MA(a3 + 0)];
```

⇩

```
#define MA_L(a) ( (((a) - LB) / 4) )
#define MA_G(a) ( (((a) - GB) / 4) )
...
#define load_addr_id(reg, a)\
  ((a) >= GB) ?\
    ((reg) = MEM_EX[MA_G(a)])) :\
    ((reg) = MEM_IN[MA_G(a)]))
...
load_addr_id(t4, (a3+0));
...
```

図 13　メモリアドレスの配列要素番号への変換

けた C プログラムを図 12 のように生成すればよい. 共有領域は外部メモリとして実装するよう関数の引数として宣言し, 非共有領域は通常の配列変数として記述する.

外部メモリと内部メモリとの振り分けは, アドレスを配列の要素番号に変換するマクロ (MA) で行う. 図 13 のように, アクセスされるメモリアドレス (a) が共有領域の開始アドレス (GB) より大きければ外部の配列の要素 (MEM_EX[MA_G(a)]) にアクセスし, アドレスが共有領域の開始アドレスより小さければ内部の配列に対応する要素 (MEM_IN[MA_G(a)]) にアクセスする.

3.3　メモリマップト I/O の実装

本稿で想定するメモリマップト I/O 実装の構成を図 14 に示す. メモリはすべてハードウェアの内部に配置し, 特定のアドレスに対してのみポートを介したメモリアクセスが行われるようにする.

このような構成は例えば Vivado HLS では図 15 のように記述できる. 共有領域も非共有領域もともにハードウェア内部で実装するよう関数内で記述する. 共有領域は関数の引数としても記述しておくが, メモリマップト I/O でアクセスされるアドレスに限り外部メモリにアクセスされるようにする.

メモリマップト I/O に対応したマクロ (MA) の記述例を図 16 に示す. 例えばメモリマップト I/O でアクセスされる領域が 3 つあり, それぞれの開始アドレスを A, B, C, ワード数を l, m, n とする. このとき, アクセスされるメモリアドレス (a) が共有領域の開始アドレス (GB) より小さければ, そのアドレスは非共有領域のものなので, 内部の配列に対応する要素 (MEM_IN[MA_L(a)]) にアクセスする. アドレスが共有領域の開始アドレス以上の数値ならばさらに場合分けする. アドレスが A 以上 A+l 未満, または B 以上 B+m 未満, または C 以上 C+n 未満ならば外部の配列に対応する要素 (MEM_EX[MA_G(a)]) にアクセスし, そうでなければ内部の配列に対応する要素 (MEM_IN[MA_G(a)]) にアクセスする.

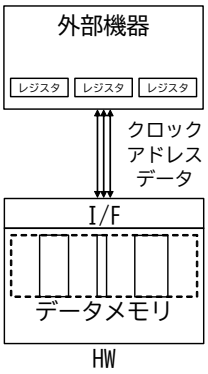
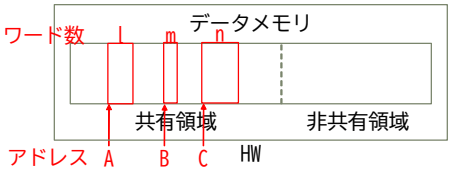


図 14　メモリマップト I/O を実装するための構成

```
void func(int target, int *res, unsigned int MEM_EX[GN]){
#pragma HLS interface bram port=MEM_EX
  static unsigned int zero = 0;
  static unsigned int a0, a1, …;
  …
  static unsigned int MEM_IN[GN+LN];
  …
}
```

図 15　メモリマップト I/O 実装の記述例




```
#define MA_L(a) ( (((a) - LB) / 4) )
#define MA_G(a) ( (((a) - GB) / 4) )
…
#define load_addr_id(reg, a)\
  (a) >= GB) ?\
    (A <= MA_G(a)) && (MA_G(a) < A+l) || \
    (B <= MA_G(a)) && (MA_G(a) < B+m) || \
    (C <= MA_G(a)) && (MA_G(a) < C+n)  : \
      (reg) = MEM_EX[MA_G(a)] :\
      (reg) = MEM_IN[MA_G(a)] :\
    (reg) = MEM_IN[MA_L(a)])
…
load_addr_id(t4, (a3+0));
…
```

図 16　メモリマップト I/O 用アドレス変換マクロ

4.　実装と実験

文献 [7] の RISC-V を対象としたバイナリ合成系に本稿の提案手法を追加実装した. 本バイナリ合成系は Python で実装されており, UNIX 系 OS で動作する.

3 本の機械語プログラム (fibonacci, DFS, heap) に対して, 本手法の実験を行った. fibonacci はフィボナッチ数列を計算するプログラム, DFS は深さ優先探索を行うプログラム, heap はヒープソートを行うプログラムである. 各機械語プログラムは, C プログラムから riscv-32-unknown-elf をターゲットとする GCC (10.2.0) により最適化オプション-O3 を指定して生成した. 各プログラムの特徴を表 2 に示す. 共有領域と非共有領域それぞれについて,「ワード数」は使用するメモリのワード数,「命令数」

表 2　プログラム毎のメモリの使用量およびアクセス数

	共有領域			非共有領域		
	ワード数	命令数	実行命令数	ワード数	命令数	実行命令数
fibonacci	52	5	126	5,000	14	500
DFS	124	39	297	4,089	46	246
heap	302	13	7,551	5	8	8

表 3　合成結果: 回路規模 (LUT 数)

program	fibonacci	DFS	heap
in	4,746 (1.00)	15,222 (1.00)	6,760 (1.00)
MMI/O	4,831 (1.02)	19,027 (1.25)	7,234 (1.07)
hybrid	3,951 (0.83)	11,609 (0.76)	3,772 (0.56)
ex	2,033 (0.43)	11,609 (0.76)	4,870 (0.72)

表 4　合成結果: サイクル数

program	fibonacci	DFS	heap
in	835 (1.00)	695 (1.00)	14,757 (1.00)
MMI/O	753 (0.90)	772 (1.11)	14,459 (0.98)
hybrid	753 (0.90)	738 (1.06)	12,272 (0.83)
ex	753 (0.90)	739 (1.06)	18,925 (1.28)

および「実行命令数」はそれぞれ各領域にアクセスするロード/ストアの静的な命令数と動的な命令数である.

各機械語プログラムから生成された C プログラムを高位合成ツールに入力し, ハードウェアを生成した. 高位合成ツールには Xilinx Vivado HLS (2020.1) を用い, 合成ターゲットは Xilinx Artix-7 とした.

合成により得られた回路の規模 (LUT 数) を表 3 に示す. 「in」が従来手法であるメモリを全て内部に置いた構成, 「MMI/O」がメモリマップト I/O に対応した構成, 「hybrid」が共有領域を外に置いた構成, 「ex」がメモリを全て外部に置いた構成である. 数値の隣の括弧内の値は「in」の手法を 1.0 としたときの比率である.

「in」と比べて, 「MMI/O」の回路規模は 2%〜25% 増加した. これは要素番号を求める計算の記述量の増加が原因と考えられる. 「hybrid」と「ex」の回路規模が減少したのは「in」ではメモリが LUT を使って実装されていたためと考えられる. 「hybrid」と「ex」の回路規模は, 外部に配置するメモリのワード数に応じて減少するはずであるが, 必ずしもそのような結果にはならなかった. これは高位合成系の最適化による回路規模構成の変動が一因として考えられる.

実行サイクル数を表 4 に示す. 「in」と比べると, プログラムによって 0.83〜1.28 倍とばらつくが, 概ね同程度となった. このばらつきについても高位合成系の最適化がプログラムや回路規模毎に異なるためと考えられる.

結果として, 回路規模, 実行サイクル数ともに大きなオーバヘッドなく外部メモリアクセスやメモリマップト I/O を実現することができた.

5. むすび

本稿では, 汎用高位合成系を用いたバイナリ合成における外部メモリアクセスの実装手法を提案した. 本手法は生成したハードウェアの内部にメモリを実装した時と比べ, 回路規模, 実行サイクル数ともに大きなオーバヘッドなく外部メモリアクセスとメモリマップト I/O を実現できた.

これまでに開発したバイナリ合成系は 32 ビットの RISC-V (RV32I) を対象としたものだが, 64 ビット版 (RV64I) や, ARM 等の命令セットアーキテクチャに対応した処理系の開発にも取り組む予定である.

謝　辞

本研究に関して有益なご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に感謝致します. また, 元関西学院大学の中道凌氏はじめ, 本研究にご協力, ご討議頂いた関西学院大学工学部石浦研究室の諸氏に感謝致します.

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] G. Stitt and F. Vahid: "Binary synthesis," *ACM TODAES*, vol. 12, no. 3, article 34 (Aug. 2007).

[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] 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," 信学技報, VLD2019-71 (Jan. 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, pp. 121–126 (Aug. 2014).

[6] N. Ito, Y. Oosako, N. Ishiura, H. Kanbara, and H. Tomiyama: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. RSP 2017*, pp. 92–98 (Oct. 2017).

[7] R. Nakamichi, S. Kishimoto, N. Ishiura, and T. Kondo: "Binary synthesis using high-vevel synthesizer as its back-end," in *Proc. SASIMI 2022*, pp. 121–126 (Oct. 2022).