Technical Report UEC-IS-2002-2
Graduate School of Information Systems, University of Electro-Communications

SimAlpha: シンプルで理解しやすいコード記述を目指したAlphaプロセッサシミュレータ

吉瀬 謙二　本多 弘樹　弓場 敏嗣
電気通信大学 大学院情報システム学研究科

プロセッサアーキテクチャ研究とプロセッサ教育における利用を目的として、プロセッサシミュレータ SimAlpha Version 1.0 を構築した。シンプルで理解しやすいコードであること、コードの変更が容易であることを第一の条件として C++を用いて記述した。インクルードファイルを含むコード量は約 2800 行と少ない。本稿では、実際のコードを参照しながら、SimAlpha のソフトウェアアーキテクチャを明らかにする。また、SimAlpha の利用例として SPEC CINT95 と CINT2000 の理想的な命令レベル並列性を測定した結果を報告する。

SimAlpha: Alpha Processor Simulator with Readable Source Code

Kenji KISE, Hiroki HONDA, and Toshitsugu YUBA
Graduate School of Information Systems,
The University of Electro-Communications

We developed a processor simulator SimAlpha Version 1.0 for research and education activities. Its design policy is to maintain the simplicity and the readability of source code. SimAlpha is written in C++ and is only 2,800 lines of code. This paper describes the software architecture of SimAlpha referring to its source code. As an actual example of use, we report the oracle IPC of SPEC CINT95 and CINT2000 benchmarks measured with SimAlpha.

1　はじめに

プロセッサアーキテクチャ研究のツールとして、あるいはプロセッサ教育のツールとして様々なプロセッサシミュレータ [7] が利用されている。近年の PC の高速化やクラスタの普及により、プロセッサシミュレータを動作させる環境は劇的に向上している。その一方で、シミュレータ構築に費す時間は、実装したいアイデアの複雑化に伴い増加する傾向にある。既存のツールをベースとして開発を進めたとしても、シミュレータの構築に数カ月を必要とする一方で、その評価は数週間で終るようなケースも珍しくない。プロセッサ研究などの目的で広く利用されているプロセッサシミュレータとして SimpleScalar Tool Set[3](以下 SimpleScalar) が有名だが、高速なシミュレーションを目的の一つとして実装されているために、必ずしも変更が容易なコードにはなっていない。

シンプルで理解しやすいコードであること、コードの変更が容易であることを第一の条件として Alpha プロセッサシミュレータ SimAlpha Version 1.0 を構築した。SimAlpha は SimpleScalar/Alpha の機能シミュレータと同等の機能をもつ。現在の実装では、命令間でオーバーラップすることなく、1 つの命令毎にプロセッサの動作を模倣する。パイプライン処理やアウトオブオーダ実行を模倣するクロックレベルのシミュレータではないが、それらへの拡張を視野に入れたコード記述となっている。SimAlpha は SimpleScalar とは異なるポリシーで、ゼロから記述したシミュレータである。C++を用いて記述され、インクルードファイルを含むコード量は約 2800 行と少ない。SimAlpha の実装においては、可読性を考慮して、グローバル変数、goto 文、条件付きコンパイルを利用していない。SimAlpha は SimpleScalar の置き換えを狙っているものではなく、異なるポリシーを持つもう一つの実装を示すことが構築の狙いである。プロセッサシミュレータをツールとして捉えると、多くの選択肢の中から適切なツールを選択できる利点は大きい。SimAlpha は研究と教育のためのツールとしてのもう一つの選択肢を提供する。

2 SimAlphaの準備

本章では、SimAlphaのコンパイル方法、シミュレーションをおこなうための実行イメージファイルの作成方法といったSimAlphaを動作させるまでの準備段階について説明する。

2.1 SimAlphaのコンパイル

SimAlphaはIntelアーキテクチャ上のLinux環境で動作する。RedHat Linux 7.2, RedHat Linux 6.2, Vine Linux 2.5CR上でコンパイルとその動作を確認している。

```
$ tar xvfz SimAlpha-release-1.0.0.tgz
$ cd SimAlpha-1.0.0
$ make
```

図 1: コンパイルとインストールのためのコマンド

コンパイルとインストールのためのUNIXコマンドを図1にまとめる。makeによるコンパイル時に「egcsというコマンドがありません。」というメッセージが表示される場合には、Makefileを編集してegcsという記述をg++に変更する。Red Hat Linux 7.2の環境で試したところ、g++と比較してegcsが約1割高速なコードを生成した。このため、 Makefileの中ではデフォルトのコンパイラをegcsに設定している。make中にエラーが発生しなければ、実行ファイルSimAlphaが生成される。このファイルを適切なディレクトリにコピーして利用する。

```
 (1)   340 COPYING
 (2)   356 README.txt
 (3)   368 define.h
 (4)    15 sim.cc
 (5)    73 chip.cc
 (6)   187 instruction.cc
 (7)   271 etc.cc
 (8)   179 memory.cc
 (9)   871 arithmetic.cc
(10)   562 syscall.cc
(11)   201 debug.cc
(12)    73 Makefile
 ------------------------
      3496 total
```

図 2: SimAlpha-release-1.0.0.tgzを展開して得られるファイルとその行数

コマンドtarによりディレクトリSimAlpha-1.0.0の中に展開されるファイルと、それぞれのファイルの行数を図2に示す。一つのインクルードファイル、8つのC++のソースファイル、ドキュメントREADME.txt、COPYING、Makefileから構成される。図2の左端の括弧に囲まれた数字は、コードを読む際の優先順位を表している。

2.2 実行イメージファイルの形式

SimAlphaはプロセッサシミュレータであり、これを動かすためには、その上で動作させるアプリケーションを用意する必要がある。SimpleScalarでは、Alphaの実機で動作するバイナリファイルを入力としてシミュレーションを開始するが、SimAlphaでは、独自形式の実行イメージファイルを入力とする。独自の形式の実行イメージファイルを用いる利点は、ELFやCOFFといった実行ファイル形式[4]の知識が不要となること、複雑なローダの実装をSimAlphaから切り離せる点にある。

```
/* SimAlpha 1.0 Image File */
/*** Registers ***/
/@reg 16 0000000000000003
/@reg 17 000000011ff97008
/@pc  32 0000000120007d80
/*** Memory    ***/
@11ff97000 3
@11ff97008 11ff97188
```

図 3: SimAlphaの実行イメージファイルの例

実行イメージファイルの一部を図3に示す。実行イメージファイルはテキスト形式で記述されたファイルで、2つの部分から成る。前半はレジスタの設定、後半はメモリの設定となっている。

前半のレジスタ設定における図3の例では、整数レジスタの16番に16進数の値3を代入し、整数レジスタの17番に16進数の値11ff97008を代入し、プログラムカウンタに16進数の値120007d80を代入する。番号0から31までを整数レジスタに、番号32をプログラムカウンタに割り当てた。これら指定のないレジスタは値0に初期化される。また、浮動小数点レジスタの内容は全て値0で初期化される。

後半のメモリ設定における図3の例では、16進数のアドレス11ff97000に値3を代入し、アドレス11ff97008に値11ff97188を代入する。これらは、8バイト単位でメモリのアドレスとデータ値との組を列挙する。指定のないメモリの内容は全て値0を持つものとして初期化される。

2.3 実行イメージファイルの作成

図 3 に示す形式の実行イメージファイルを作成する方法を示す。幾つかの実行イメージファイルのサンプルは SimAlpha のサイトからダウンロードできるので、これらを利用して SimAlpha を動作させる場合には本節をスキップしてかまわない。

実行イメージファイルは Alpha バイナリから構成できる。SimAlpha の実行イメージファイルは SimpleScalar のローダを利用して作成できる。図 4 に示すコードを SimpleScalar Version 3.0 の loader.c の 728 行目に挿入する。このコードは、SimpleScalar がレジスタとメモリの内容をセットした後に、これらの値を読みだして値をファイルに書き出す。

```
{ /***** This is added by Kenji KISE *****/
  unsigned long long a;   /* Address */
  unsigned long long dat;
  FILE *fp = fopen("aout.txt", "w");
  fprintf(fp,"/* SimAlpha 1.0 Image File */\n");

  fprintf(fp,"/*** Registers ***/\n");
  for(i=0; i<32; i++){
    fprintf(fp, "/@reg %02d %016llx\n",
            i, regs->regs_R[i]);
  }
  fprintf(fp,"/@pc  %02d %016llx\n",
          i, regs->regs_PC);

  fprintf(fp,"/*** Memory    ***/\n");
  for(a=0x11f000000; a<0x121000000; a+=8){
    mem_access(mem, Read, a, &dat, 8);
    if(dat!=0)
      fprintf(fp, "@%llx %llx\n", a, dat);
  }

  for(a=0x140000000; a<0x141000000; a+=8){
    mem_access(mem, Read, a, &dat, 8);
    if(dat!=0)
      fprintf(fp,"@%llx %llx\n", a, dat);
  }
  fclose(fp);
}
```

図 4: 実行イメージファイルを生成するために SimpleScalar の loader.c 728 行目に追加するコード

図 4 に示すコードを loader.c に追加した後に sim-safe をコンパイルする。構築した sim-safe を用いてアプリケーションを実行する際に、SimAlpha の実行イメージファイル aout.txt が作成される。

2.4 SimApha の実行オプション

SimAlpha の実行イメージファイルは通常 aout.txt というファイル名を持つので、通常は、コマンド
SimAlpha aout.txt
によりシミュレーションを開始する。この際に
SimAlpha -e10m aout.txt
の様にオプションを与えることができる。利用できるオプションを以下に列挙する。

-d オプション シミュレーション開始と同時にデバッグモードに入る。デバッグモードでは、リターンの入力により 1 つの命令を実行してレジスタの内容を表示する。また、指定した命令数の実行が終了した時点でレジスタの値を表示する c コマンド, 特定のアドレスの命令が実行された時点でレジスタの値を表示する s コマンド, 現在のメモリの内容を表示する x コマンドを利用できる。ソースコード debug.cc の中の関数は、-d オプションを利用する場合にのみ利用される。

-c オプション システムコールの実行トレースを表示する。Linux 環境では、strace コマンドを用いてアプリケーションを実行した場合に近い結果を得ることができる。

-v オプション 一定数の命令のシミュレーションが進んだ際にピリオドを表示し、シミュレーションの進行具合を確認できる。-v の後に、ピリオドを表示する間隔を指定する。-v1000 は千命令毎にピリオドを表示する。また、数字の後に m を追加することで million 単位で間隔を指定できる。-v10m は 1 千万命令毎にピリオドを表示することを意味する。

-e オプション 指定した命令数のシミュレーションが終了した時点でシミュレーションを終了する。-e の後に、シミュレーションが終了するまでの命令数を指定する。また、数字の後に m を追加することで million 単位で間隔を指定できる。-e10m というオプションは 1 千万命令でシミュレーションを終了することを意味する。

-i オプション 標準入力の代わりに指定したファイル名のファイルの内容を利用する。-i の後に、利用するファイル名を指定する。
SimAlpha aout.txt < input.file
という実行と、
SimAlpha -iinput.file aout.txt

```
class system_config{
 public:
  char program_name[512];
  uint64_t end_tc;
  int verbose_interval;
  int debug_flag;
  int syscall_flag;
  int fd_stdin_flag;
  int fd_stdin; /* file descripter for STDIN */
  system_config(char *, char**);
};

system_config::system_config(char *program,
                             char **option){
  memset(this, 0, sizeof(system_config));
  strcpy(program_name, program);

  if(option==NULL) return;
  for(int i=0; option[i]!=NULL; i++){
    char opt[1024];
    strcpy(opt, option[i]);
    if(opt[0]!='-') continue;
    switch(opt[1]){
    case 'c': syscall_flag = 1; break;
    case 'd': debug_flag   = 1; break;
    case 'v':
      verbose_interval =atoi(&opt[2]);
      if(verbose_interval == 0)
          verbose_interval  =  100000;
      if(opt[strlen(opt)-1]=='m')
          verbose_interval *= 1000000;
      printf("*** verbose interval: %d\n",
             verbose_interval);
      break;
    case 'e':
      end_tc =atoi(&opt[2]);
      if(opt[strlen(opt)-1]=='m')
        end_tc *= 1000000;
      printf("*** Finish at %qd code execution.\n",
             end_tc);
      break;
    case 'i':
      fd_stdin_flag = 1;
      fd_stdin = open(&opt[2], O_RDONLY);
      printf("*** SimAlpha: use [%s] as STDIN.\n",
              &opt[2]);
      if(fd_stdin<0){
        printf("Can not open file: %s\n", &opt[2]);
        exit(1);
      }
      break;
    default:
      printf("*** Error: [%s] Wrong option!!\n\n",
             opt);
      usage();
      exit(1);
    }
  }
}
```

図 5: クラス system_config の定義とコンストラクタ

という実行は同じ結果をもたらす。例えば、2 つのシミュレーションイメージを作成した場合に、それぞれが標準入力からのデータを必要とする場合を考える。この時、標準入力を複製することは困難なので、それぞれのシミュレーションイメージがファイルを開いて、そこからデータを得るようにする。このように、-i オプションは、複数のシミュレーションイメージを構築したり、SimAlpha と他のシミュレータとの動作検証の際に利用される。

オプションを解析した結果は、シミュレーションの対象となるシステム設定の情報などが格納される system_config 型のオブジェクト sc に格納される。クラス system_config の定義とコンストラクタのコードを図 5 に示す。まず、memset を用いて全ての変数を値 0 で初期化し、次に、入力オプションを解析して変数の値を設定する。

2.5 ベンチマークプログラムと測定に用いる PC の構成

本稿の評価には、SPEC[1] CINT95 の 8 本のプログラムと、SPEC CINT2000 の 12 本のプログラムの合計 20 本のベンチマークプログラムを利用する。各ベンチマークの入力データセットと実行命令数を図 6 にまとめる。CINT2000 の 9 つのベンチマークに関しては University of Minnesota が提供している縮小入力セット MinneSPEC[2] 利用する。それ以外のベンチマークにおいては、シミュレーションの命令数を抑えるように入力パラメタを調整する。SPEC CINT95 のバイナリは DEC C コンパイラ、最適化オプション O4 を用いて生成する。SPEC CINT2000 のバイナリは、SimpleScalar のサイトからダウンロードしたものを利用する。

表 1: 測定に利用する PC の構成

Machine	GS-SR101 1U Rackmount Server
Mother	GA-6VXDR7 (Dual Socket 370)
Processor	Intel Pentium III 1GHz x 2
DRAM	512MB (SD 512MB PC133/3/R-ECC x 1)
HD	80GB RAID-1 mirroring (Seagate ST380021A x 2)
OS	Red Hat Linux 7.2 (Fast Trak RAID Driver)

シミュレーション速度などのデータは、表 1 にまとめ

SPEC CINT95	099.go 9 9		*1		138,956,479
	124.m88ksim < ctl.in (train)		*1		127,441,803
	126.gcc	*10	*1		150,965,671
	129.compress < {3000 q 2131}		*1		142,023,912
	130.li train.lsp	*11	*1		208,158,264
	132.ijpeg	*12	*1		136,720,409
	134.perl	*13	*1		153,959,895
	147.vortex	*14	*1		184,606,636
SPEC CINT2000	164.gzip mdred.graphic 1		*2	*3	596,563,251
	175.vpr *20	*20	*2	*3	17,616,865
	176.cc1 mdred.rtlanal.i		*2	*3	551,226,252
	181.mcf smred.in		*2	*3	188,605,056
	186.crafty < crafty.in(test)		*2		4,264,764,568
	197.parser	*21	*2	*3	611,705,488
	252.eon	*22	*2		94,038,802
	253.perlbmk	*23	*2	*3	200,692,300
	254.gap	*24	*2		1,169,578,113
	255.vortex	*25	*2	*3	147,895,384
	256.bzip2 lgred.source 1		*2	*3	1,819,780,238
	300.twolf smred/smred		*2	*3	91,901,750

```
*1 : binary is compiled with DEC C compiler -O4 option
*2 : binary is downloaded from SimpleScalar LLC
*3 : used the reduced input sets from University of Minnesota
*10: cc1 -quiet -funroll-loops -fforce-mem -fcse-follow-jumps
     -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce
     -fpeephole -fschedule-insns -finline-functions
     -fschedule-insns2 -O genrecog.i -o genrecog.s
*11: ijpeg -image_file specmun.ppm -compression.quality 50
     -compression.optimize_coding 0 -compression.smoothing_factor
     50 -difference.image 1 -difference.x_stride 10
     -difference.y_stride 10 -verbose 1 -GO.findoptcomp
*12: perl scrabbl.pl scrabbl.in ("admits" in 1/5 input)
*13: train/vortex.in (PART_COUNT 10)
*20: vpr smred.net small.arch.in place.out other.out -nodisp
     -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num
*21: parser 2.1.dict -batch < mdred.in
*22: eon chair.control.cook chair.camera chair.surfaces
     chair.cook.ppm ppm pixels_out.cook (test)
*23: perlbmk -I. -I./lib smred.makerand.pl
*24: gap -l ./ -q -m 64M < test.in (test)
*25: vortex smred.raw (used a person.1k instead of a person.250)
```

図 6: ベンチマークプログラムの入力データセットと実行命令数。各プログラムの右端の数字が実行命令数を表す。

る SMP 型の PC を利用して測定する。

2.6 SimAlpha のシミュレーション速度

SimAlpha と sim-safe[3] の動作速度を測定した結果を表 2 にまとめる。単位は 1 秒当たりに何百万命令をシミュレーションできるかという数値 (Million Instructions Per Second) で示した。SimAlpha のコンパイルには egcs-1.1.2 release、最適化オプション O2 を用いている。

表 2: sim-safe と SimAlpha のシミュレーション速度 (Million Instructions Per Second) の比較

Program	sim-safe	SimAlpha
099.go	3.30	1.17
124.m88ksim	3.26	1.17
126.gcc	3.21	1.17
129.compress	3.08	1.19
130.li	3.15	1.16
132.ijpeg	3.44	1.26
134.perl	3.07	1.14
147.vortex	2.97	1.12
164.gzip	3.20	1.23
175.vpr	3.52	1.04
176.gcc	3.06	1.14
181.mcf	2.73	1.16
186.crafty	3.17	1.13
197.parser	3.18	1.14
252.eon	2.98	0.96
253.perlbmk	3.15	1.09
254.gap	2.84	1.16
255.vortex	3.18	1.10
256.bzip2	3.06	1.17
300.twolf	3.03	1.04
average	3.13 MIPS	1.14 MIPS

20 個のベンチマークの算術平均を計算すると、sim-safe のシミュレーション速度 3.1 MIPS に対して、SimAlpha のシミュレーション速度は 1.1 MIPS という結果となった。SimpleScalar と比較して、約 3 倍のシミュレーション時間が必要となる点が SimAlpha の欠点の一つとなっている。ただし、シミュレータ開発の時間が支配的な場合に、その開発時間を短縮できるとすれば、シミュレーション速度で 3 倍の速度差は問題にならない範囲といえる。

3 SimAlpha の動作検証手法

SimAlpha の開発において、アーキテクチャステートの動作レベルで、SimpleScalar との互換性を確認した。SPEC CINT95 と CINT2000 の 20 個のベンチマークプログラムを利用して、1 つの命令を実行する度に、SimAlpha のアーキテクチャステート (プログラムカウンタ, 32 本の整数レジスタ、32 本の浮動小数点レジスタ) と、SimpleScalar のアーキテクチャステートの全ての値を比較し、一致することを確認している

```
extern "C" int simple_cpu(int, uint64_t *);

int simple_cpu(int job, uint64_t *p){
  static simple_chip *chip;

  switch(job){
  case 1: {
    static char *opt[OPTION_MAX];
    char *buf;
    for(int i=0; i<OPTION_MAX; i++) opt[i]=NULL;
    FILE *fp;
    if((fp=fopen("SimAlpha_Option.txt","r"))
        !=NULL){
      printf("*** Use SimAlpha_Option.txt\n");
      int i=0;
      while(!feof(fp)){
        buf = (char *)malloc(256);
        fscanf(fp, "%s", buf);
        if(buf[0]=='-'){
          opt[i++] = buf;
          i++;
        }
      }
    }
    chip = new simple_chip(DEF_FILE_NAME, opt);
  }
  break;
  case 2: {
    *(p++) = chip->as->pc.ld();
    int running = chip->step();

    for(int i=0; i<31; i++)
      *(p++)=chip->as->r[i].ld();
    for(int i=0; i<31; i++)
      *(p++)=chip->as->f[i].ld();

    if(running==0) delete chip;
    break;
  }
  default:
    printf("Warning: %d siple_cpu.\n", job);
  }
  return 0;
}
```

図 7: 動作検証に利用する C のインターフェース

2つのシミュレータのアーキテクチャステートの値を一致させるために、SimAlpha の動作を、SimpleScalar の望ましくない挙動と一致させた箇所が 3 つ存在する。また、getdirentries, stat, getrlimit, setrlimit という 4 つのシステムコールの実装において SimpleScalar のコードを変更した。

従来のシミュレータでは、グローバル変数の存在などにより、一つのプロセス内に 2 つのシミュレーションのイメージを作ることは難しかった。このため、正しい実行結果をトレースとしてファイルに保存し、これと比較することで、もうひとつのシミュレータの検証をおこなっていた。SimAlpha の場合には、高速かつ確実に検証をおこなう手段として、SimpleScalar などのプログラムに SimAlpha のオブジェクトを埋め込む機能を提供する。これを実現するために、図 7 に示す C のインターフェース simple_chip を提供する。

SimpleScalar の一部として SimAlpha を動作させる場合には、システムコール open で得られるファイル識別子の値が一致しないという問題が発生する。例えば、SimAlpha がシステムコール open を呼び出し 5 番というファイル識別子を得たとすると、SimpleScalar の同様のシステムコール open は 6 番のファイル識別子を取得する。これらのファイル識別子はレジスタ内に格納されるために、2 つのシミュレーション結果が異なる結果を得たとして、シミュレーションが止まってしまう。これを回避するため、SimAlpha の実装においては、得られたファイル識別子とレジスタに格納する値との間の変換テーブルを持つことで、2 つのシミュレーションイメージがファイル識別子として同じ値を利用するように工夫を凝らしている。

検証のために SimpleScalar に追加するコードを図 8 に示す。これらのコードを sim-safe.c の 398 行目に挿入する。図 8 に示すように検証のために必要となるコードは約 40 行であり、また、変更するソースコードは 1ヶ所と少ない。

図 8 のコードを挿入した SimpleScalar のコードをコンパイルするためには、SimAlpha のオブジェクト (ライブラリ) を結合する必要がある。SimAlpha のディレクトリ SimAlpha-1.0.0 の中でコマンド make libsimalpha.lib により、ライブラリ libsimalpha.lib が生成される。その後、生成されたライブラリを SimpleScalar のディレクトリにコピーして、SimpleScalar の Makefile の OBJS の部分に libsimalpha.lib を追加する。その後、コマンド make sim-safe より動作検証のための実行ファイルをコンパイルする。

SimAlpha のオブジェクトを組み込んだ sim-safe を実行することで、SimAlpha の動作を検証できる。各命令の実行後に、2 つのシミュレーションの結果を比較し、それらが異なる場合にはアーキテクチャステートの内容を表示してシミュレーションを中断する。sim-safe の実行オプションは SimpleScalar のものとして利用される。組み込まれた SimAlpha に対してオプションは、コマンドラインではなく、図 7 のコードに示す様に、SimAlpha_Option.txt というファイルの内容を利用するようにした。

```
{ /**** SimAlpha Verification by Kenji KISE ****/
static int init=0;
static unsigned long long *p;
int i;

if(init==0){ /**** Initialize ****/
  simple_cpu(1, NULL);
  p = (unsigned long long  *)calloc(65, 8);
  simple_cpu(2, p);      /** Step Execution **/
  init=1;
}

if(regs.regs_PC!=*p){
  printf("** PC Error %llx %llx \n",
         regs.regs_PC, *p); exit(0);
}
for(i=0; i<31; i++){
  if(*(p+i+1)!=regs.regs_R[i] ||
     *(p+i+32)!=regs.regs_F.q[i]){
    printf("\n** Index %2d **", i);

    myfprintf(stdout, "%7n [0x%08x] @ 0x%08p: ",
      sim_num_insn, md_xor_regs(&regs),
                                regs.regs_PC);
    md_print_insn(inst, regs.regs_PC, stdout);
    if (MD_OP_FLAGS(op) & F_MEM)
      myfprintf(stdout, "  mem: 0x%08p", addr);
    printf("\n    SimAlpha(Int)    ");
    printf("simplescalar(Int)");
    printf(" SimAlpha(Float) simplescalar(F)\n");
    for(i=0; i<31; i++){
      printf("%02d  %016llx:%016llx  ",
             i,*(p+i+1),regs.regs_R[i]);
      printf("%016llx:%016llx\n",
             *(p+i+32),regs.regs_F.q[i]);
    }
    exit(1);
  }
}
simple_cpu(2, p);      /** Step Execution **/
} /**** END of SimAlpha Verification ****/
```

図 8: 動作検証のために sim-safe.c に追加するコード

SimAlpha の提供する、複数のシミュレーションイメージを作成したり、あるシミュレータの中にオブジェクトを埋め込んで検証する手法は、SimAlpha と SimpleScalar の間の検証に利用されるだけではなく、開発中の幾つかのシミュレータの動作を検証するためにも利用すること

ができる。この機能を利用することで、開発中のシミュレータのバグを早期に発見したり、シミュレータの正当性を確認することが可能となる。

4 SimAlphaのソフトウェアアーキテクチャ

本章では、ソースコードの高い可読性を示すために、変更を加えていない C++の本物のコード (疑似コードではない) を示しながら、SimAlpha の実装内部の説明を試みる。ただし、本稿のスペースの都合上、ソースコードの改行位置を変更したり、ソースコード中のコメントを排除する場合がある。

まず、メイン関数の説明から始め、そこで生成されるオブジェクト chip のコンストラクタが、7 つのオブジェクトを生成することを見る。幾つかの重要なクラスの定義を見た後に、重要な役割を果たすクラス instruction の定義とコードを説明する。最後に、メモリシステムの実装を説明する。

4.1 メイン関数

SimAlpha のメイン関数を図 9 に示す。変数にプログラム名とオプションを設定してから、simple_chip 型のオブジェクト chip を生成する。1 つの命令を実行するメンバ関数 step は、実行する命令がなくなった時点 (アプリケーションのシミュレーションが終了した時点) で値 0 を返す。while ループにより、関数 step が値 0 を返すまで繰り返すことでシミュレーションを進めていく。実行が終わるとオブジェクト chip を解放し、その際に呼ばれるデストラクタがシミュレーション結果を表示する。

```
#include "define.h"

int main(int argc, char **argv){
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */

  simple_chip *chip =  new simple_chip(p, opt);
  while(chip->step());
  delete chip;

  return 0;
}
```

図 9: SimAlpha のメイン関数 (sim.cc)

4.2 クラス simple_chip

クラス simple_chip の定義、コンストラクタ、デストラクタを図 10 に示す。simple_chip 型のオブジェクトの生成時に呼び出されるコンストラクタが 7 種類のオブジェクトを生成する。

一方、シミュレーションの終了時に呼ばれるデストラクタは、関数 print_evaluation_result を呼び出してシミュレーション結果を表示した後に、7 つのオブジェクトを開放する。

```
class simple_chip{
  system_config      *sc;
  evaluation_result  *e;
  debug              *deb;
  system_manager     *sys;
  instruction        *p;
 public:
  memory_system      *mem;
  architecture_state *as;
  simple_chip(char *, char **);
  ~simple_chip();
  int step();
};

simple_chip::simple_chip(char *prog, char **opt){
  sc  = new system_config(prog, opt);
  e   = new evaluation_result;
  as  = new architecture_state(sc, e);
  mem = new memory_system(sc, e);
  deb = new debug(as, mem, sc, e);
  sys = new system_manager(as, mem, sc, e);
  p   = new instruction(as, mem, sys, sc, e);
}

simple_chip::~simple_chip(){   /** destructor **/
  print_evaluation_result(e, sc, sys);
  delete p;
  delete sys;
  delete deb;
  delete mem;
  delete as;
  delete e;
  delete sc;
}
```

図 10: クラス simple_chip の定義、コンストラクタ、デストラクタ

ステップ実行をおこなうクラス simple_chip のメンバ関数 step のコードを図 11 に示す。instruction 型のオブジェクト p に対して、パイプラインステージに対応する 7 つの関数と WriteBack の 8 つの関数を順番に呼び出すことで 1 つの命令を実行する。SimAlpha Version 1.0 では機能レベルシミュレータの能力しか提供しないが、コードの可読性と拡張性を考慮して、Alpha21264[6] の命令パ

```
int simple_chip::step(){
  p->Fetch(&as->pc);    /* pipeline stage 0 */
  p->Slot();            /* pipeline stage 1 */
  p->Rename();          /* pipeline stage 2 */
  p->Issue();           /* pipeline stage 3 */
  p->RegisterRead();    /* pipeline stage 4 */
  p->Execute(&as->pc);  /* pipeline stage 5 */
  p->Memory();          /* pipeline stage 6 */
  p->WriteBack();

  /* split a conditional move,see README.txt */
  execute_cmovb(p, as);

  e->retired_inst++;
  house_keeper(sys, sc, e, deb);

  return sys->running;
}
```

図 11: 関数 step のコード

イプラインを参考にしながら命令の動作を 8 つのステージに分割して記述した。条件付き移動命令 (CMOV 命令) に関しては、2 つの入力オペランドの命令として処理するために、1 つの命令を 2 つに分割して処理している。関数 execute_cmovb は、分割された 2 つ目の命令を処理する関数である。

文献 [6] の条件付き移動命令に関する記述を引用する。The 21264 splits each conditional move instruction into two new instructions only have two register sources. The first instruction places the move value into an internal register together with a 65th bit indicating the move's ultimate success or failure. The second instruction reads the first result (including the 65th bit) together with the old destination register value and produces the final destination register results.

4.3 幾つかの重要なクラスの定義

4.3.1 データを表現するクラス data_t

計算結果は、レジスタファイルやメモリ内に格納されるが、これらは図 12 に示すクラス data_t のオブジェクトの集合として定義される。

data_t 型のオブジェクトの値を設定する場合には関数 st を利用する。また、値を読みだす場合には関数 ld を利用する。これ以外に、新しいデータを生成する場合には関数 init を利用する。

```
class data_t{
  uint64_t value;
 public:
  int cmov;
  uint64_t ld();
  int   st(uint64_t);
  int init(uint64_t);
};

int data_t::init(uint64_t d){
  value = d;
  cmov  = 0;
  return 0;
}

uint64_t data_t::ld(){
  return value;
}

int data_t::st(uint64_t d){
  value = d;
  return 0;
}
```

図 12: データオブジェクト data_t の定義と関数

4.3.2 アーキテクチャステート

プログラムカウンタ、整数レジスタ、浮動小数点レジスタから成るクラス architecture_state の定義とコンストラクタを図 13 に示す。コンストラクタでは、まず、全ての変数を値 0 で初期化した後に、実行イメージファイルを読み込み、その前半のレジスタ設定の記述に従いレジスタファイルの内容とプログラムカウンタの値を設定する。

4.3.3 クラス evaluation_result

evaluation_result 型のオブジェクトには評価中のデータを保存する。変数の値はシミュレーションの実行中に更新されるが、これらの値がシミュレーションの実行に影響を与えることはない。クラス evaluation_result の定義を図 14 に示す。それぞれの変数が、実行した命令数、メインメモリにおいて利用したページ数、シミュレーションを開始した時刻を格納する。

4.3.4 system_config

system_config 型のオブジェクトには、シミュレーションの対象となるシステム設定の情報などが格納される。これらの値はシミュレーションの開始前に定義され、原則として、シミュレーション中に値が変わることはない。クラス system_config の定義とコンストラクタのコード

```
class architecture_state{
 public:
  data_t pc;    /* program counter      */
  data_t r[32]; /* general purpose regs */
  data_t f[32]; /* floating point  regs */
  architecture_state(system_config *,
                     evaluation_result *);
};

architecture_state::architecture_state(
                    system_config *sc,
                    evaluation_result *e){
  for(int i=0; i<32; i++) r[i].init(0);
  for(int i=0; i<32; i++) f[i].init(0);
  char *program_name = sc->program_name;
  FILE *fp;
  if((fp = fopen(program_name, "r")) == NULL) {
    fprintf(stderr, "Bad file name: %s\n",
            program_name);
    exit(1);
  }
  int i;
  DATA_TYPE dat;
  char buf[4096];
  fgets(buf, 4096, fp);
  if(strncmp(buf+3, "SimAlpha", 8)){
    fprintf(stderr,
      "%s: Not a SimAlpha data file.\n",
       program_name);
    exit(1);
  }
  while(!feof(fp)){
    fgets(buf, 4096, fp);

    if(*buf=='/' && *(buf+1)=='@'){
      sscanf(buf+6,"%d %llx\n", &i, &dat);
      if(i==32) pc.init(dat);
       else r[i].init(dat);
    }
    if(i==32) break;
  }
  fclose(fp);
}
```

図 13: アーキテクチャステートの定義とコンストラクタ

```
class evaluation_result{
 public:
  uint64_t retired_inst;
  int used_memory_block;
  time_t time_begin;     /* start time stamp */
  struct timeval tp;     /* start time stamp */
  struct timezone tzp;   /* start time stamp */
  evaluation_result();
};
```

図 14: クラス evaluation_result の定義

は図 5 に示してある。

4.4 クラス instruction

本節では、クラス instruction の定義と、8 つのステージに対応する関数のコードを説明する。ただし、関数 Rename は処理をおこなっていない空の関数なので、関数 Rename の説明は省略する。

```
class instruction{
  evaluation_result  *e;
  architecture_state *as;
  system_manager     *sys;
  memory_system      *mem;
  INST_TYPE ir; /* 32bit instruction code  */
  int Op;       /* Opcode field            */
  int RA;       /* Ra field of the inst    */
  int RB;       /* Rb field of the inst    */
  int RC;       /* Rc field of the inst    */
  int ST;       /* store inst ?            */
  int LD;       /* load inst ?             */
  int LA;       /* load address inst ?     */
  int BR;       /* branch inst ?           */
  int Ai;       /* Rav is immediate ?      */
  int Bi;       /* Rbv is immediate ?      */
  int Af;       /* Rav from floating-reg ? */
  int Bf;       /* Rbv from floating-reg ? */
  int WF;       /* Write to the f-reg ?    */
  int WB;       /* Writeback reg index     */
  data_t  Npc;  /* Update PC or PC + 4     */
  data_t  Imm;  /* immediate               */
  data_t  Adr;  /* load & store address    */
  data_t  Rav;  /* Ra                      */
  data_t  Rbv;  /* Rb                      */
  data_t  Rcv;  /* Rc                      */
 public:
  int Fetch(data_t *);
  int Fetch(data_t *, INST_TYPE);
  int Slot();
  int Rename();
  int Issue();
  int RegisterRead();
  int Execute(data_t *);
  int Memory();
  int WriteBack();
  INST_TYPE get_ir();
  int data_ld(data_t *, data_t *);
  int data_st(data_t *, data_t *);
  instruction(architecture_state *,
              memory_system *,
              system_manager *,
              system_config *,
              evaluation_result *);
};
```

図 15: クラス instruction の定義

クラス instruction の定義を図 15 に示す。パイプライ

ンステージに対応する関数が呼びだされ、命令の処理が進むに従って、オブジェクト内のプライベート変数の値が計算されていく。int 型で定義されている 14 個の変数は命令コード ir から、デコードされた値を保持する。data_t 型の変数は、メモリやレジスタファイルからロードした値や、メモリやレジスタファイルに格納する値を保持する変数である。

4.4.1 命令フェッチステージ

図 16 に示す様に、命令フェッチのための関数 Fetch は 2 つ存在する。図 16 の上に示したコードは通常の命令 (CMOV 以外の命令) のための関数 Fetch で、プログラムカウンタ PC の示すアドレスから 4 バイトの命令をロードして変数 ir に格納する。また、PC に値 4 を加えて得られる次命令のアドレスを Npc に格納する。

図 16 の下に示したコードは、条件付き移動命令の分割された 2 つ目の命令のフェッチ時に利用する。条件付き移動命令は 2 つに分割されるが、分割された 2 つ目の命令は最初の命令の命令コードから生成される。このため、命令コードを引数の一つとして関数 Fetch を呼び出すことになる。

```
int instruction::Fetch(data_t *pc){
  mem->ld_inst(pc, &ir);
  Npc.init(pc->ld() + 4);
  return 0;
}

int instruction::Fetch(data_t *pc,
                       INST_TYPE ir_t){
  ir = ir_t;
  Npc.init(pc->ld());
  return 0;
}
```

図 16: 命令フェッチステージのコード

4.4.2 スロットステージ

スロットステージのコードを図 17 示す。先のステージでフェッチした命令コード ir を用いて、幾つかの変数の値をデコードする。

図 17 の記述の様に、変数にデコードした値を代入する替わりに、マクロを用いてコードを記述することができる。後者の方法を用いることで、シミュレーション時間の向上を期待できるが、コードの可読性を考慮して変数に代入する方法を選択した。また、このような記述は Verilog-HDL の記述と近く、C++と Verilog-HDL との間の部分的なコードの再利用を可能とする。

```
int instruction::Slot(){
  Op  = (ir>>26) &  0x3F;
  RA  = (ir>>21) &  0x1F;
  RB  = (ir>>16) &  0x1F;
  RC  = (ir    ) &  0x1F;
  WF  = ((Op&MSK2)==0x14 || (Op&MSK2)==0x20);
  LA  = (Op==0x08 || Op==0x09);
  LD  = (Op==0x0a || Op==0x0b || Op==0x0c ||
         (Op&MSK2)==0x20 || (Op&MSK2)==0x28);
  ST  = (Op==0x0d || Op==0x0e || Op==0x0f ||
         (Op&MSK2)==0x24 || (Op&MSK2)==0x2c);
  BR  = ((Op&MSK4)==0x30);
  WB  = (LD || (Op&MSK2)==0x08 || Op==0x1a ||
         Op==0x30 || Op==0x34) ? RA :
        ((Op&MSK3)==0x10 || Op==0x1c) ? RC : 31;
  Af  = (Op==0x15 || Op==0x16 || Op==0x17 ||
         Op==0x1c ||
         (Op&MSK2)==0x24 || (Op&MSK3)==0x30);
  Bf  = ((Op&MSK2)==0x14);
  Ai = (Op==0x08 || Op==0x09 || LD);
  Bi = (BR || (Op&MSK2)==0x10 && (ir & BIT12));

  /** For the CMOV Split Code (CMOV1) **/
  if(cmov_ir_create(ir)){
    RB = RC;
    Bi = 0;
  }
  return 0;
}
```

図 17: スロットステージのコード

4.4.3 発行ステージ

発行ステージのコードを図 18 示す。ここでは、命令の種類に応じて必要となるイミディエート Imm を作成する。

4.4.4 レジスタリードステージ

レジスタリードステージのコードを図 19 を示す。レジスタリードステージでは、イミディエート値、浮動小数点レジスタファイル、整数レジスタファイルから選択してデータを読みだし、Rav と Rbv それぞれのを設定する。

4.4.5 実行ステージ

実行ステージのコードを図 20 に示す。実行ステージでは、3 つのデータを更新する。通常の算術論理命令は Rav と Rbv を入力として Rcv の値を計算する。ロード

```
int instruction::Issue(){
  DATA_TYPE Lit, D16, D21, tmp, d21e, d16e;
  d21e = ((ir & MASK21) | EXTND21) << 2;
  d16e  = (ir & MASK16) | EXTND16;

  Lit = (ir>>13) & 0xFF;
  D21 = (ir & BIT20) ? d21e : (ir&MASK21)<<2;
  D16 = (ir & BIT15) ? d16e : (ir&MASK16);
  if(Op==0x09) D16 = (D16 << 16);

  tmp = (LA||LD||ST) ? D16 : (BR) ? D21 : Lit;
  Imm.init(tmp);
  return 0;
}
```

図 18: 発行ステージのコード

```
int instruction::RegisterRead(){
  Rav = Ai ? Imm : Af ? as->f[RA] : as->r[RA];
  Rbv = Bi ? Imm : Bf ? as->f[RB] : as->r[RB];
  return 0;
}
```

図 19: レジスタリードステージのコード

ストア命令はメモリ参照アドレス Adr を計算する。分岐命令は分岐先アドレス Tpc を計算する。

4.4.6 メモリアクセスステージ

メモリアクセスステージのコードを図 21 に示す。ストア命令の場合には Rav の値をメモリにストアする。ロード命令の場合にはロードした値を Rcv に保存する。

4.4.7 ライトバック

ライトバックのコードを図 22 示す。実行している命令が PAL(Privileged Architecture Library) コードである場合には、execute_pal 関数を呼びだす。結果を生成する通常の命令の場合は、データ Rcv をレジスタファイルに格納して当該命令の実行を完了する。

4.5 メモリシステム

SimAlpha Version 1.0 のメモリシステムは、キャッシュを含む階層的なものではなく、メインメモリのみのシンプルな構成として実現する。Alpha AXP アーキテクチャのアドレスは 64 ビット幅だが、SimAlpha Version 1.0 においては、アドレスの上位 32 ビットを無視し、下位の 32 ビットのみを利用する。通常のコンパイルにより生成さ

```
int instruction::Execute(data_t *Tpc){
  /*** Update Rcv ***/
  if(BR || Op==OP_JSR){
    Rcv=Npc;
  }
  else if(!LD){
    ALU(ir, &Rav, &Rbv, &Rcv);
  }

  /*** Update Adr ***/
  Adr.init(0);
  if(LD || ST){
    ALU(ir, &Imm, &Rbv, &Adr);
  }

  /*** Update Tpc ***/
  *Tpc = Npc;
  if(Op==OP_JSR){
    *Tpc = Rbv;
    Tpc->st(Tpc->ld() & ~3ull);
  }
  if(BR){
    BRU(ir, &Rav, &Rbv, &Npc, Tpc);
  }
  return 0;
}
```

図 20: 実行ステージのコード

```
int instruction::Memory(){
  if(ST) data_st(&Adr, &Rav);
  if(LD) data_ld(&Adr, &Rcv);
  return 0;
}
```

図 21: メモリアクセスステージのコード

```
int instruction::WriteBack(){
  if(Op==OP_PAL){
    sys->execute_pal(this);
  }

  if(!WF && WB!=31) as->r[WB] = Rcv;
  if( WF && WB!=31) as->f[WB] = Rcv;
  return 0;
}
```

図 22: ライトバックのコード

れるコードにおいて、上位 32 ビットの値は 0x00000001 に固定されるので、このような実装で問題になることはない。クラス memory_system の定義を図 23 に示す。

```
class memory_system{
  evaluation_result *e;
  class main_memory *mm;
  void ld_8byte(data_t *, data_t *);
  void st_8byte(data_t *, data_t *, DATA_TYPE);
 public:
  void ld_inst(data_t *, INST_TYPE *);
  void ld_nbyte(int, data_t *, data_t *);
  void st_nbyte(int, data_t *, data_t *);
  ~memory_system();
  memory_system(system_config *,
                evaluation_result *);
};
```

図 23: クラス memory_system の定義

4.5.1 メインメモリの実装

メインメモリは、data_t 型のオブジェクトの配列として実現される。32 ビットのアドレスで指定されるメモリ空間を、8KB のページを単位として管理する。

```
class main_memory{
  evaluation_result *e;
  data_t *block_table[BLOCK_TABLE_SIZE];
  data_t *allocblock(data_t *);
 public:
  void ld_8byte(data_t *, data_t *);
  void st_8byte(data_t *, data_t *, DATA_TYPE);
  main_memory(evaluation_result *);
};
```

図 24: クラス main_memory の定義

メインメモリの定義を図 24 に示す。ポインタを格納する配列 block_table は、32 ビットのアドレスを 8KB のページサイズで割った数のエントリ数を持つ。メインメモリの実装を図 25 に示す。メインメモリのコンストラクタにおいて、配列 block_table の全てのエントリを NULL で初期化する。

メインメモリの参照には、8 バイト単位でロードする関数 ld_8byte と、8 バイト単位のマスク付きストアを実現する関数 st_8byte を利用する。

関数 ld_8byte は、一つ目の引数としてロードするアドレスを指定し、ロードしたデータを二つ目の引数として指定したポインタの実体にコピーする。もし、指定したアドレスを含むページが最初の参照だった場合には、関数 allocblock を呼び出して、必要とするページを割り当てた後にデータをロードする。

```
main_memory::main_memory(evaluation_result *ev){
  e = ev;
  for(int i=0; i<BLOCK_TABLE_SIZE; i++)
    block_table[i]=NULL;
}

data_t *main_memory::allocblock(data_t *a){
  data_t *ret=
    (data_t *)calloc(BLOCK_SIZE/DATA_T_SIZE,
                     sizeof(class data_t));
  block_table[MM_TABLE_INDEX(a->ld())]=ret;
  if(ret==NULL){
    printf("** Error in allocblock.\n");
    exit(0);
  }
  e->used_memory_block++;
  return ret;
}

void main_memory::ld_8byte(data_t *a, data_t *d){
  ADDR_TYPE adr = a->ld() & ~7;
  data_t *ptr = block_table[MM_TABLE_INDEX(adr)];
  unsigned int offset =
    (adr & BLOCK_MASK)/DATA_T_SIZE;
  if(ptr==NULL) ptr=allocblock(a);
  *d = *(ptr + offset); /** COPY **/
}

void main_memory::st_8byte(data_t *a, data_t *d,
                           DATA_TYPE msk){
  ADDR_TYPE adr = a->ld() & ~7;
  data_t *ptr = block_table[MM_TABLE_INDEX(adr)];
  unsigned int offset =
    (adr & BLOCK_MASK)/DATA_T_SIZE;
  if(ptr==NULL) ptr=allocblock(a);
  (ptr + offset)->st(((ptr + offset)->ld() & msk)
                     | d->ld());
}
```

図 25: クラス main_memory の定義とコード

関数 st_8byte は、一つ目の引数としてストアするアドレスを指定し、二つ目の引数としてストアするデータを指定する。また、三つ目の引数として、ストアする際のマスクを指定する。

st_8byte(a, d, 0);

例えば、上の呼び出しはマスク値 0 を利用する。すなわち、64 ビット全てのデータがメモリにストアされる。

st_8byte(a, d, 0xFFFFFFFF00000000ull);

上の呼び出しは、上位 32 ビットを除く下位 32 ビットの値のみがメモリにストアされる。

メインメモリに関しては、全てのアドレスは 8 バイト単位の整列されたアドレス指定による、8 バイト単位の

ロードストアにより参照される。このため、メインメモリへのアクセスに利用されるアドレスの下位 3 ビットと、アドレスの上位 32 ビットは無視される。

4.6 メモリシステムの実装

先に示したメインメモリは 8 バイト単位の参照しか許さない。1 バイト、2 バイト、4 バイト単位の参照を許す一般的なメモリ参照は、メモリシステムの機能として実現する。

```
void memory_system::ld_nbyte(int n,
                             data_t *a, data_t *d){
  if(a->ld()%n!=0)
    printf("*** ld_nbyte n=%d miss-alignment.\n",
           n);

  ld_8byte(a, d);

  int offset = a->ld() & 7;
  switch(n){
  case 1: {
    DATA_TYPE data= (d->ld() >> (offset * 8))
                  & 0xfflu;
    d->st(data);
    break;
  }
  case 2: {
    DATA_TYPE data= (d->ld() >> (offset * 8))
                  & 0xffffllu;
    d->st(data);
    break;
  }
  case 4: {
    DATA_TYPE data= (d->ld() >> (offset * 8))
                  & 0xffffffffllu;
    d->st(data);
    break;
  }
  case 8: {
    break;
  }
  default:
    printf("Case %d, Error in load_nbyte\n", n);
    exit(1);
  }
}
```

図 26: ld_nbyte のコード

ロードするバイト数を選択できる ld_nbyte のコードを図 26 に示す。最初の引数において、ロードするバイト数を指定する。

同様に、ストアするバイト数を選択できる st_nbyte のコードを図 27 に示す。

これらの関数を利用することで、1, 2, 4, 8 バイト単位でロードとストアが可能だが、指定したバイト数で整列したアドレスを用いる必要がある。

```
void memory_system::st_nbyte(int n, data_t *a,
                             data_t *d){
  if(a->ld()%n!=0)
    printf("*** st_nbyte n=%d miss-alignment.\n",
           n);

  int offset = a->ld() & 7;
  DATA_TYPE mask = 0;

  switch(n){
  case 1: {
    mask = ~(0xfflu << offset*8);
    DATA_TYPE data = (d->ld() & 0xfflu)
                   << offset*8;
    d->st(data);
    break;
  }
  case 2: {
    mask = ~(0xffffllu << offset*8);
    DATA_TYPE data = (d->ld() & 0xffffllu)
                   << offset*8;
    d->st(data);
    break;
  }
  case 4: {
    mask = ~(0xffffffffllu << offset*8);
    DATA_TYPE data = (d->ld() & 0xffffffffllu)
                   << offset*8;
    d->st(data);
    break;
  }
  case 8: {
    mask = 0;
    break;
  }
  default:
    printf("Case %d, Error in store_nbyte\n",
           n);
    exit(1);
  }

  st_8byte(a, d, mask);
}
```

図 27: st_nbyte のコード

5 SimAlpha の利用例

本章では、SimAlpha の拡張性を示す例として、制御依存関係や資源競合を解消し、データ依存関係のみを制約として得られる理想的な命令レベル並列性を測定するように SimAlpha を拡張する。また、これを用いて測定した結果を報告する。測定する値は文献 [5] の oracle IPC

と同様の意味を持つ。ベンチマークには SPEC CINT95 と CINT2000 を利用した。

5.1 クラス data_t の拡張

SimAlpha で扱うデータは、単なる unsigned long long 型の値ではなく、data_t 型のオブジェクトとして定義される。理想的な命令レベル並列性を測定するために、計算された全てのデータにデータフローグラフの高さに相当する値 (これをランクと呼ぶことにする) を格納するようにクラス data_t を変更する。物理メモリはクラス data_t のオブジェクトの配列として構築されている。ロードストア命令は 1 バイトから 8 バイトの粒度でメモリを参照するので、メモリ内のデータをオブジェクトとして表現する際の粒度には幾つかの選択肢があるが、8 バイト単位の整列したデータを一つのオブジェクトとした。

理想的な命令レベル並列性を測定するために変更されたクラス data_t の定義とコンストラクタを図 28 に示す。クラス data_t に uint32_t 型の変数 rank を追加した。この変数にデータのランクを格納する。コンストラクタでは変数 rank を値 0 に初期化するコードを追加した。

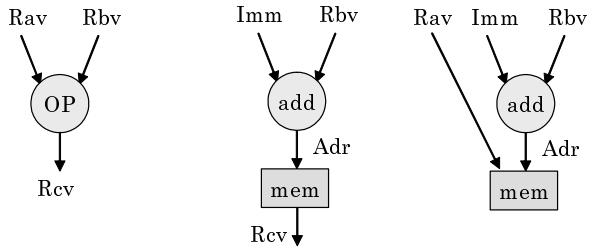
```
class data_t{
  uint64_t value;
 public:
  int cmov;
  uint32_t rank;  /* This line is inserted. */
  uint64_t ld();
  int   st(uint64_t);
  int init(uint64_t);
};

int data_t::init(uint64_t d){
  value = d;
  cmov  = 0;
  rank  = 0;      /* This line is inserted. */
  return 0;
}
```

図 28: 拡張したデータオブジェクト data_t の定義と拡張したコンストラクタ

5.2 ランクと理想的な命令レベル並列性の計算方法

ランクの計算方法を図 29 に示す。算術論理命令は、ALU によりデータが計算される時に、2 つの入力オペランド Rav と Rbv のランクの最大値に演算レイテンシを加えた値を計算結果 Rcv のランクとして格納する。ロード命令では、Rbv のランクにアドレス計算のためのレイテンシとメモリ参照レイテンシを加えることでランクを計算する。ストア命令は、メモリに書き込むデータ Rav と、アドレス計算により得られるランクとの最大値をデータのランクとしてメモリに格納する。



(a) rank(Rcv) = max(rank(Rav), rank(Rbv)) + latency(OP)
(b) rank(Rcv) = rank(Rbv) + latency(add) + latency(mem)
(c) rank = max(rank(Rav), rank(Rbv) + latency(add))

図 29: 命令タイプ毎のランクの計算方法

シミュレーション中には、個々のデータが持つランクとは別に、全データのランクの最大値を更新する。シミュレーションが終了した時点のランクの最大値がデータフローグラフの高さを表すので、この値と実行命令数から理想的な並列性を計算できる。ただし、データはシステムコールを越えてスケジューリングできないという制約を加えた。以下の評価において、ランクを計算する際の演算レイテンシとメモリ参照レイテンシを 1 サイクルとした。

5.3 SimAlpha の拡張

理想的な並列性を測定するために SimAlpha に変更を加えた。この変更の多くの部分はデータが生成される際のランクの計算であり、追加した全てのコードはわずか 26 行だった。結果表示のコードを除いた変更点を図 30 から図 34 に示す。追加した行に /* Added */ というコメントを追加した。

Execute ステージに追加したコードは、ALU において演算を施した後にデータ Rav, Rbv のランクの最大値に演算レイテンシ 1 を加えた値をデータ Rcv のランクとして設定している。

Memory ステージでは、ストアの場合には Rav をストアする前にそのランクを計算し、ロードの場合には、ロードしたデータ Rcv のランクを計算するコードを追加した。

WriteBack では、レジスタファイルにデータをコピー

```
int instruction::Execute(data_t *Tpc){
  /*** Update Rcv ***/
  if(BR || Op==OP_JSR){
    Rcv=Npc;
  }
  else if(!LD){
    ALU(ir, &Rav, &Rbv, &Rcv);
    Rcv.rank = (Rav.rank>Rbv.rank) ?  /* Added */
                Rav.rank : Rbv.rank;  /* Added */
    Rcv.rank += 1; /* ALU latency */  /* Added */
  }

  /*** Update Adr ***/
  Adr.init(0);
  if(LD || ST){
    ALU(ir, &Imm, &Rbv, &Adr);
    Adr.rank = (Imm.rank>Rbv.rank) ?  /* Added */
                Imm.rank : Rbv.rank;  /* Added */
    Adr.rank += 1; /* ALU latency */  /* Added */
  }

  /*** Update Tpc ***/
  *Tpc = Npc;
  if(Op==OP_JSR){
    *Tpc = Rbv;
    Tpc->st(Tpc->ld() & ~3ull);
  }
  if(BR){
    BRU(ir, &Rav, &Rbv, &Npc, Tpc);
  }
  return 0;
```

図 30: 拡張した Execute のコード

```
int instruction::Memory(){
  if(ST){
    Rav.rank = (Adr.rank > Rav.rank) ? /* Added */
                Adr.rank : Rav.rank;   /* Added */
    if(Rav.rank <e->systemcall_rank)   /* Added */
        Rav.rank = e->systemcall_rank; /* Added */
    data_st(&Adr, &Rav);
  }
  if(LD){
    data_ld(&Adr, &Rcv);
    Rcv.rank = (Adr.rank>Rcv.rank) ?   /* Added */
                Adr.rank : Rcv.rank;   /* Added */
    Rcv.rank += 1; /* Load latency */  /* Added */
  }
  return 0;
}
```

図 31: 拡張した Memory のコード

```
int instruction::WriteBack(){
  if(Op==OP_PAL){
    sys->execute_pal(this);
    e->systemcall_rank = e->max_rank;  /* Added */
  }

  if(WB!=31){                          /* Added */
    if(e->max_rank < Rcv.rank)         /* Added */
       e->max_rank = Rcv.rank;         /* Added */
    if(Rcv.rank < e->systemcall_rank)  /* Added */
       Rcv.rank = e->systemcall_rank;  /* Added */
  }
  if(!WF && WB!=31) as->r[WB] = Rcv;
  if( WF && WB!=31) as->f[WB] = Rcv;
  return 0;
}
```

図 32: 拡張した WriteBack のコード

```
void main_memory::st_8byte(data_t *a, data_t *d,
                           DATA_TYPE msk){
  ADDR_TYPE adr = a->ld() & ~7;
  data_t *ptr = block_table[MM_TABLE_INDEX(adr)];
  unsigned int offset =
    (adr & BLOCK_MASK)/DATA_T_SIZE;
  if(ptr==NULL) ptr=allocblock(a);
  (ptr + offset)->st(((ptr + offset)->ld() & msk)
                     | d->ld());

  if(msk==0)                           /* Added */
   (ptr+offset)->rank = d->rank;       /* Added */
  else if((ptr+offset)->rank < d->rank) /* Added */
   (ptr+offset)->rank = d->rank;       /* Added */
}
```

図 33: 拡張した st_8byte のコード

```
class evaluation_result{
 public:
  uint64_t retired_inst;
  int used_memory_block;
  time_t time_begin;
  struct timeval tp;
  struct timezone tzp;
  evaluation_result();
  int max_rank;                        /* Added */
  int systemcall_rank;                 /* Added */
};
```

図 34: 拡張した evaluation_result のコード

する際に、全データのランクの最大値を計算している。また、システムコールを跨いだ命令スケジューリングを禁止しているので、システムコールが呼ばれた際のランクの最大値よりデータのランクが小さくならないように調整をおこなっている。

メインメモリは 8 バイト単位でデータを管理するので、1、2、4 バイトの値をストアする場合には、すでにメモリに格納されているデータとストアするデータとのマージが発生する。拡張した st_8byte のコードでは、ストアするバイト数が 8 でない場合 (mask が 0 でない場合) には、マージする 2 つのデータの最大値を新しいデータのランクとして格納する。

5.4 理想的な命令レベル並列性の測定結果

理想的な命令レベル並列性の測定結果を表 3 に示す。変更後のクラス data_t の定義に示した様に、ランクを保持するための変数の追加によりシミュレーションのためのメモリ量が増加する。また、ランクを計算するための処理が追加されているにもかかわらず、シミュレーション時間に関して深刻な増加はみられなかった。変更後のシミュレーション速度も、ほぼ 1.1MIPS である。

理想的な IPC の測定結果を見ると、124.m88ksim と 253.perlbmk で低い値を示したが、それ以外のベンチマークでは数十というレベルから多いもので 186.crafty の 108 という高い並列性が内在していることを確認できた。

6 おわりに

プロセッサアーキテクチャ研究とプロセッサ教育における利用を目的として、プロセッサシミュレータ SimAlpha Version 1.0 を構築した。本稿では、コードの高い可読性を示すために、実際の C++のコードを示しながら SimAlpha のソフトウェアアーキテクチャを明らかにした。また、SimAlpha の利用例として、理想的な命令レベル並列性を測定する方法を示し、これを測定するための機能を数十行という少ない記述で追加できることを述べた。変更した SimAlpha を利用して SPEC CINT95 と CINT2000 の理想的な命令レベル並列性を測定し、その結果を報告した。

SimAlpha Version 1.0 と、理想的な命令レベル並列性を得るために変更を加えたソースコードは次の URL からダウンロードできる。

`http://www.yuba.is.uec.ac.jp/~kis/SimAlpha/`

表 3: 拡張した SimAlpha を用いて測定した実行命令数、シミュレーション速度、理想的な命令レベル並列性

Program	instruction count	MIPS	IPC
099.go	138 million	1.12	64.2
124.m88ksim	127 million	1.12	10.5
126.gcc	150 million	1.12	41.8
129.compress	142 million	1.14	56.6
130.li	208 million	1.11	20.0
132.ijpeg	172 million	1.21	107.0
134.perl	153 million	1.10	43.3
147.vortex	184 million	1.07	32.0
164.gzip	596 million	1.19	16.9
175.vpr	17 million	1.00	25.1
176.gcc	551 million	1.10	47.1
181.mcf	188 million	1.12	53.1
186.crafty	4264 million	1.10	108.0
197.parser	611 million	1.10	30.9
252.eon	94 million	0.93	49.7
253.perlbmk	200 million	1.05	8.4
254.gap	1169 million	1.12	32.1
255.vortex	147 million	1.06	29.3
256.bzip2	1819 million	1.12	43.6
300.twolf	91 million	1.00	21.9

参考文献

[1] Standard Performance Evaluation Corporation. SPEC benchmark suites. http://www.spec.org.

[2] AJ KleinOsowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. In *Computer Architecture Letters*, Vol. 1, June 2002.

[3] Doug Burger and Todd M. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin-Madison, June 1997.

[4] John R. Levine. *Linkers & Loaders*. オーム社, 2001.

[5] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *19th Annual International Symposium on Computer Architecture*, pp. 46–57, May 1992.

[6] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, Vol. 19, No. 2, pp. 25–36, March 1999.

[7] Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Performance Simulation Tools. *IEEE Computer*, Vol. 35, No. 2, pp. 38–39, February 2002.