Technical Report UEC-IS-2003-4, Version 2003-06-04
Graduate School of Information Systems, University of Electro-Communications

スカラプロセッサシミュレータの実装と動作検証

吉瀬 謙二　片桐 孝洋　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

C++を用いてパイプライン化されたスカラプロセッサのシミュレータを構築した．本稿の前半では，動作検証の機能を効率的に実装するため，SimAlpha Version 1.0 に拡張を施す．この版を SimAlpha Version 1.1 とし，その詳細を述べる．本稿の後半では，SimAlpha Version 1.1 をベースとして，6 段の命令パイプライン構成を持つ単純なスカラプロセッサのシミュレータを構築する．また，シミュレータ構築の信頼性を向上させるために，構築したスカラプロセッサシミュレータに動作検証の仕組みを追加する．

キーワード プロセッサシミュレータ，スカラプロセッサ，動作検証，SimAlpha

Implementation and Verification of Scalar Processor Simulator

Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba
Graduate School of Information Systems,
University of Electro-Communications

The simulator of a pipelining scalar processor was constructed using the processor simulator SimAlpha. In the first half, SimAlpha Version 1.0 is extended in order to implement the behavior verification efficiently. This version is defined as SimAlpha Version 1.1, and the details are given. In the second half, the simulator of a scalar processor with the six pipeline stages is constructed using SimAlpha Version 1.1. In order to improve the reliability of simulator construction, the verification mechanism is installed in the constructed scalar processor simulator.

Key-words processor simulator, scalar processor, verification, 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 [2, 3, 4] 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．我々は，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プロセッサシミュレータ SimAlpha[5, 6] の開発を続けている．

新たに，パイプライン化されたスカラプロセッサのシミュレータを構築した．本稿の前半では，動作検証の機能を効率的に実装するため， SimAlpha Version 1.0 に拡張を施す．この版を SimAlpha Version 1.1 とし，その詳細を述べる．本稿の後半では，SimAlpha Version 1.1 を用いて，6 段の命令パイプライン構成を持つ単純なスカラプロセッサのシミュレータを構築する．また，シミュレータ構築の信頼性を向上させるために，構築したスカラプロセッサシミュレータに動作検証の仕組みを追加する．

2 SimAlpha Version 1.1

動作検証を効率的に実装するため，SimAlpha Version 1.0 に拡張を施す．この版を SimAlpha Version 1.1 と定義し，拡張をおこなったソースコードを示しながら，その詳細をまとめる．

まず，変更前の，SimAlpha Version 1.0 のメイン関数を示す．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．

```
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;
}
/** main function of SimAlpha Version 1.0.0 **/
```

SimAlpha 1.0 の実装では，simple_chip 型のプロセッサが一つのオブジェクトとして実装されている．このため，比較的単純に，2 つのプロセッサを生成し，処理を進めることができる．c1 と c2 という 2 つのプロセッサを生成し，1 命令毎に相互に処理を進めるように変更を加えたメイン関数を示す．

```
int main(int argc, char **argv){
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */
  simple_chip *c1 =  new simple_chip(p, opt);
  simple_chip *c2 =  new simple_chip(p, opt);

  while(c1->step() & c2->step());

  delete c1;
  delete c2;
  return 0;
}
/** main function to simulate two chips **/
```

上に示したコードは，同じプログラム名と実行オプションを利用して，c1 と c2 という 2 つのプロセッサを生成し，どちらかの処理が終了するまでループを繰り返す．

本稿で対象とする動作検証の意味を明確にするために，より現実的なメイン関数の例を示す．

```
int main(int argc, char **argv){
  printf("%s\n", VER);
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */
  simple_chip *c1 = new simple_chip(p, opt);
  simple_chip *c2 = new simple_chip(p, opt);

  int run=1;
  while(run){
    run = c1->step();
    c2->step();
    int error = 0;
    for(int i=0; i<65; i++){
      if(c1->rr(i)!=c2->rr(i)) error=1;
    }
    if(error){printf("Verify Error\n"); exit(1);}
  }

  delete c1;
  delete c2;
  return 0;
}
/** main function with verification **/
```

c1 と c2 という 2 つのプロセッサが存在し，構築中で検証を必要とするプロセッサを c1, 正しい実行が保証された simple_chip 型のプロセッサを c2 として区別する．c1 の処理を進め，1 つの命令の処理が完了した後に，c2 の処理を進める．その後，2 つのプロセッサのアーキテクチャステートを構成する 65 個のレジスタの値を比較して，一つでも値が異なる場合には，変数 error をセットし，処理を停止する．この時，必要に応じてレジスタ値の表示等をおこない，検証に失敗した原因を調査する．

上のコードでは，アーキテクチャステートを構成するレジスタの値を返すメソッド rr が利用されている．このメソッドは SimSlpha Version 1.1 で追加されたものである．simple_chip のメソッド rr のコードを示す．

```
uint64_t architecture_state::rr(int n){
  if(n==0) return pc.ld();
  if(n>=1  && n<=32) return r[n-1].ld();
  if(n>=33 && n<=64) return f[n-33].ld();
  printf("Error:architecture_state::rr(%d)\n",n);
  exit(1);
}
```

SimAlpha を用いた動作検証の枠組みに関して述べてきた．SPEC CINT95 の 099.go の様に，ファイル操作をおこなわないベンチマークプログラムを用いて動作を検証する場合には，ここで示したメイン関数を利用すればよい．しかし，ファイル操作が必要となる多くのベンチマークプログラムでは，これに対処するため，更に SimAlpha の拡張が必要となる．

2.1 動作オプションを取得する仕組み

複数のプロセッサを生成する際に，それぞれのプロセッサが異なる動作オプションを利用できると便利である．動作検証のための準備として，プロセッサの動作オプションをファイルから取得する仕組みを追加する．

動作オプションを列挙したファイルの名前を指定して，動作オプションを設定する関数 option_from_file を SimAlpha Version 1.1 に追加した．このコードを示す．

```
void option_from_file(char **opt, char *name){
  char *buf;
  for(int i=0; i<OPTION_MAX; i++) opt[i]=NULL;
  FILE *fp = fopen(name,"r");
  if(fp!=NULL){
    printf("*** Use %s\n", name);
    for(int i=0; !feof(fp);){
      buf = (char *)malloc(256);
      fscanf(fp, "%s", buf);
      if(buf[0]=='-') opt[i++] = buf;
    }
  }
}
```

動作オプションを記述したファイルの例を下に示す．テキスト形式で各行の先頭が–で始まっている行をオプションとして認識する．下の例では，動作オプションとして -c と -v10000 を設定する．

```
# This file is used to verify SimAlpha.
#                2003-05-21 Kenji KISE
-c
-v10000
```

プロセッサ c1 のオプションはコマンドラインから取得し，プロセッサ c2 のオプションの内容を verify.txt という名前のファイルから取得するメイン関数のコード例を示す．

```
int main(int argc, char **argv){
  printf("%s\n", VER);
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */
  simple_chip *c1 =  new simple_chip(p, opt);

  char *op[OPTION_MAX];
  option_from_file(op, "verify.txt");
  simple_chip *c2 = new simple_chip(p, op);

  int run=1;
  while(run){
    run = c1->step();
    c2->step();
    int error = 0;
    for(int i=0; i<65; i++){
      if(c1->as->rr(i)!=c2->as->rr(i)) error=1;
    }
    if(error){printf("Verify Error\n"); exit(1);}
  }

  delete c1;
  delete c2;
  return 0;
}
/** another main function with verification **/
```

2.2 ファイル記述子の調整

システムコール open が利用されると，ファイル記述子として重複しない適切な整数値が返り値として渡される．ここで，実行している 2 つのプロセッサイメージ c1, c2 がファイルを開いた場合には，例えば c1 にはファイル識別子 3 が，c2 にはファイル識別子 4 が返され，これらの値がレジスタに格納される．このため，c1 と c2 のアーキテクチャステートが一致せずに，動作検証に失敗する．

この問題を回避するために，得られたファイル識別子を変換する仕組みが必要となる．この仕組みを実現するために，fd_table という変換テーブルを用意する．この値は，下に示す system_manager のコンストラクタの一部において初期化される．

```
  /** for example, fd_table[6]=7        **/
  /** maps real fd(6) to logical fd(7). **/
  /*************************************/
  fd_table[0] = 0; // STDIN
  fd_table[1] = 1; // STDOUT
  fd_table[2] = 2; // STDERR
  for(int i=3; i<FD_MAPPING_MAX; i++)
    fd_table[i] = i + 2 - ms->fd_adjust;
```

ファイル記述子 0,1,2 という値は，標準入力，標準出力，標準エラー出力として予約されている．これらの値は変換しない．それ以外の値は，得られたファイル記述子に 2 を加えた値を，返還後の値として利用する．動作オプション –f を利用して fd_adjust の値を設定することで，ファイル記述子の値を調整する．

変換されたファイル記述子から，本来のファイル記述子を得るためには，関数 translate_fd を利用する．このコードを示す．

```
int system_manager::translate_fd(DATA_TYPE fd){
  for(int i=0; i<FD_MAPPING_MAX; i++){
    if(fd_table[i] == fd) return i;
  }
  printf("*** Error in translate_fd\n");
  exit(1);
}
```

ファイル記述子を調整する仕組みの利用方法を考えてみる．2つのプロセッサイメージ c1, c2 を用いた動作検証を考える．この時，c2 の実行オプションとして –f を利用することで，c2 の fd_adjust の値を 1 に設定する．c1 は，fd_adjust の値を 0 に設定する．2 つのプロセッサ c1, c2 がファイルを開いて，c1 にはファイル記述子 3 が，c2 にはファイル記述子 4 が返されたとする．この時，fd_adjust の値を用いて調整することで，c1 のファイル記述子を 4 に保ちながら，c2 のファイル記述子を 4 に変更できる．このように，得られたファイル記述子を調整することで，2 つのアーキテクチャステートを同じ状態に保つことができる．

ファイル記述子はシステムコール open, close, read, write, unlink 等で利用される．

2.3 ファイル名の調整

SPEC CINT95 の 126.gcc の様に，実行結果をファイルに格納する場合を考える．例えば，126.gcc が genrecog.i を入力ファイルとすると，コンパイル結果を genrecog.s という名前のファイルに格納する．この時，2 つのプロセッサ c1, c2 が同じファイル名を利用すると，c1 と c2 が一つのファイル genrecog.s を変更するために，正しい内容のファイルが得られないという問題が生じる．

この問題を解決するために，起動オプション –f が利用された場合に，生成されるファイル名を変更するようにする．起動オプション –f が利用された場合には，ファイル名の 2 文字目を'S' に，3 文字目を'A' に，4 文字目を-f の依存する文字に変更する．ファイル名を変更するためのコードを示す．

```
      if(ms->fd_adjust){ /** rename **/
        buf[1]='S';
        buf[2]='A';
        buf[3]='0'+ ms->fd_adjust;
      }
```

buf は，ファイル名を格納している文字配列であり，fd_adjust が 0 でない場合に，2 文字目から 4 文字目を変更する．SA は SimAlpha の略である．SPEC CINT95 の実行においては問題にならないが，ファイル名が 4 文字以下の場合に，上に示したコードはエラーとなる．この場合には，コードを適切に修正する必要がある．

先に例として用いた SPEC CINT95 の 126.gcc の場合には，c1 が genrecog.s という名前のファイルを生成し，c2 が gSA1ecog.s という名前のファイルを生成する．3 つ目のプロセッサを生成して fd_adjust の値を 2 に設定すると， gSA2ecog.s という名前のファイルが生成される．

2.4 その他の変更点

SimAlpha Version 1.0 では，クラス instruction の変数 Npc に，当該命令のプログラムカウンタに 4 を加えた値を格納し，これを用いて，実行ステージにける分岐先アドレスを計算していた．

SimAlpha Version 1.1 では，動作確認のために，当該命令の次の命令のアドレスを変数 Npc に格納し，当該命令のプログラムカウンタを格納するために，新たに Cpc という変数を導入する．この変更にともない，クラス instruction の定義，instruction.cc 内の命令フェッチと，実行ステージの動作を変更する．

クラス instruction の定義の変更点を示す．Cpc を追加し，Npc のコメントを変更した．

```
  data_t  Cpc;  /* PC of the instruction   */
  data_t  Npc;  /* PC of the next inst.    */
```

命令フェッチステージのコードを示す．当該命令のプログラムカウンタの値を変数 Cpc に格納する．当該命令の命令アドレスに 4 加えた値を変数 Npc に格納する．

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

実行ステージのコードを示す．当該命令が分岐命令の場合には，計算されたターゲットアドレスを用いて，Npc の値を更新する．

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

  return 0;
}
```

条件付き移動命令 (CMOV 命令) を処理するためには，例外的な動作が必要となる．この命令の判定を容易にするために，クラス instruction の変数 CM を追加する．命令デコード時に，当該命令が条件付き移動命令の場合には変数 CM をセットする．

SimAlpha Version 1.1.0 では，これまでに述べてきた変更点に加えて，記述スタイルの統一を進めている．

2.5 SimAlpha Version 1.1 のコード量

SimAlpha Version 1.1 に含まれる一つのインクルードファイルと 8 つの C++のソースファイルの行数を図 1 にまとめる．コード量の合計は 2,792 行と非常に少ない．

```
   377 define.h
   873 arithmetic.cc
    78 chip.cc
   204 debug.cc
   304 etc.cc
   204 instruction.cc
   180 memory.cc
    19 sim.cc
   553 syscall.cc
------------------------
  2792 total
```

図 1: SimAlpha Version 1.1 のコード量

SimAlpha Version 1.0 のコード量が 2,727 行である．Version 1.1 への拡張に伴って約 70 行が増加している．

表 1: SimAlpha のシミュレーション速度 (MIPS: Million Instructions Per Second) の比較

Program	version 1.0	version 1.1	verify
099.go	2.23	2.05	0.48
124.m88ksim	2.23	2.09	0.48
126.gcc	2.23	2.11	0.47
129.compress	2.24	2.12	0.48
130.li	2.24	2.13	0.48
132.ijpeg	2.32	2.18	0.46
134.perl	2.19	2.09	0.47
147.vortex	2.20	2.08	0.47
average	2.23	2.10	0.47

2.6 SimAlpha Version 1.1 の動作速度

SimAlpha の動作速度を測定した結果をまとめる．本稿の評価には，SPEC[1] CINT95 の 8 本の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は文献 [5] と同じものを利用する．

SimAlpha Vresion 1.0 と Version 1.1 の動作速度を測定した結果を表 1 にまとめる．表 1 の 2 列目に SimAlpha Version 1.0 の動作速度，3 列目に SimAlpha Version 1.1 の動作速度，4 列目には SimAlpha Version 1.1 をベースとして 2 つのプロセッサ生成し，動作検証の処理を追加した場合の動作速度を示す．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) で示す．SimAlpha のコンパイルには gcc version 2.96，最適化オプション O2 を用いる．測定には，Intel Xeon 2.8GHz プロセッサを 2 個，メモリ 4GB を搭載した計算機を利用する．

8 個のベンチマークの算術平均を用いて SimAlpha のシミュレーション速度を比較する．

SimAlpha Version 1.0 のシミュレーション速度 2.23 MIPS に対して，SimAlpha Version 1.1 のシミュレーション速度は 2.10 MIPS となった．instruction.cc に，新しい変数 Cpc を追加したために，僅かに動作速度が低下している．

動作検証の処理を追加した場合には， 0.47 MIPS と極端に遅くなる．アーキテクチャステートを構成する全てのレジスタの値を比較していることが速度低下の大きな原因である．SPEC CINT95 の場合には，浮動小数点演算をほとんど含んでいないので，汎用レジスタとプログラムカウンタのみを検証の対象とするといった比較対象

のレジスタ数の制限や，比較のためのデータを連続的に読み出すといった工夫により高速化が可能である．

3 単純なスカラプロセッサの構築

SimAlpha Version 1.1 を土台として，6 段の命令パイプライン構成を持つ単純なスカラプロセッサを構築する．このシミュレータは，サイクル毎にプロセッサの状態を更新する (トレース駆動ではない) 実行ベースのシミュレータである．構築するシミュレータはプロセッサの性能等を評価するために利用できる．また，シミュレータ構築の信頼性を向上させるために，シミュレータに動作検証の仕組みを追加する．

3.1 プロセッサアーキテクチャの定義

構築するプロセッサの命令パイプラインを図 2 に示す．命令フェッチ (IF)，命令デコード (ID)，レジスタ参照 (RR)，実行 (EX)，メモリ参照 (MM)，ライトバック (WB) という 6 段の命令パイプラインを採用する．

IF	ID	RR	EX	MM	WB
stg[0]	stg[1]	stg[2]	stg[3]	stg[4]	stg[5]

図 2: 命令パイプラインの構成

命令キャッシュとデータキャッシュは実装しない．1 サイクルという理想的な遅延で，必要とする命令とデータを供給できると仮定する．

データフォワーディングはおこなわない．また，同一サイクルにおいて，前半でレジスタファイルへの書き込みをおこない，後半で読み出しをおこなうといった動作をおこなわない．このため，データ依存のある後続命令は，先行する命令がレジスタファイルにデータを書き戻した後に，レジスタ参照 (RR) ステージでオペランドを取得する．

一般的な分岐予測器を実装しない．実装を単純化するために，分岐を不成立と仮定して，命令フェッチを継続する．

PAL 命令がライトバックステージに進んだ際に，先行する全ての命令をフラッシュし，再度，PAL 命令の次の命令から処理を再開する．

条件付き移動 (CMOV) 命令は，2 オペランドの命令として実現し，ライトバックステージに進んだ際に適切な処置をおこなう．

3.2 スカラプロセッサの動作例

先に定義したプロセッサアーキテクチャの動作例として，プログラムが始まってから，20 命令が完了するまでの処理の様子 (ログ) を示す．ここに示すログは–t オプションを用いてスカラプロセッサ版の SimAlpha を起動することで得ることができる．

ログの先頭と分岐予測ミスが起こった場合を除いて，1 サイクルの処理が終わった際に，1 行のログを出力する．各行の左端の数字は，クロックサイクルを示している．続いてパイプラインの各ステージに存在する命令の識別子を表示する．対応するパイプラインステージに命令が投入されていない場合 (バブルの場合) には’——’ を表示する．各命令の識別子として 3 桁の整数を用いる．各ステージの命令が当該サイクルに処理された場合に，識別子の後ろにアスタリスク’*’ を付ける．アスタリスクが無い命令は，ストールしている命令を表す．次の 0 あるいは 1 という一桁の数字は，データ依存関係の有無を表現する．この数字が 1 の場合には，命令デコード (ID) の命令と先行して実行している命令との間にデータ依存関係がある．この時，命令フェッチ (IF) と命令デコード (ID) をストールさせる．命令の処理が完了したサイクルには，右端に，完了した命令の数を表示する．

```
 cycle:  IF   ID   RR   EX   MM   WB  :dependency:
000001: 001* ---- ---- ---- ---- ---- :0:
000002: 002* 001* ---- ---- ---- ---- :0:
000003: 003* 002* 001* ---- ---- ---- :0:
000004: 003  002  ---- 001* ---- ---- :1:
000005: 003  002  ---- ---- 001* ---- :1:
000006: 003  002  ---- ---- ---- 001* :1: 0001
000007: 004* 003* 002* ---- ---- ---- :0:
000008: 005* 004* 003* 002* ---- ---- :0:
000009: 006* 005* 004* 003* 002* ---- :0:
000010: 007* 006* 005* 004* 003* 002* :0: 0002
000011: 007  006  ---- 005* 004* 003* :1: 0003
000012: 008* 007* 006* ---- 005* 004* :0: 0004
000013: 008  007  ---- 006* ---- 005* :1: 0005
000014: 009* 008* 007* ---- 006* ---- :0:
000015: 009  008  ---- 007* ---- 006* :1: 0006
000016: 009  008  ---- ---- 007* ---- :1:
000017: 009  008  ---- ---- ---- 007* :1: 0007
000018: 010* 009* 008* ---- ---- ---- :0:
000019: 011* 010* 009* 008* ---- ---- :0:
000020: 011  010  ---- 009* 008* ---- :1:
000021: 011  010  ---- ---- 009* 008* :1: 0008
000022: 012* 011* 010* ---- ---- 009* :0: 0009
000023: 012  011  ---- 010* ---- ---- :1:
000024: 012  011  ---- ---- 010* ---- :1:
000025: 012  011  ---- ---- ---- 010* :1: 0010
000026: 013* 012* 011* ---- ---- ---- :0:
000027: 014* 013* 012* 011* ---- ---- :0:
000028: 014  013  ---- 012* 011* ---- :1:
```

```
000029: 014  013  ---- ---- 012* 011* :1: 0011
000030: 015* 014* 013* ---- ---- 012* :0: 0012
000031: ---- ---- ---- 013* ---- ---- :0:
Branch Miss Prediction 00001

000034: 017* ---- ---- ---- 013* ---- :0:
000035: 018* 017* ---- ---- ---- 013* :0: 0013
000036: 019* 018* 017* ---- ---- ---- :0:
000037: 020* 019* 018* 017* ---- ---- :0:
000038: 020  019  ---- 018* 017* ---- :1:
000039: 020  019  ---- ---- 018* 017* :1: 0014
000040: 021* 020* 019* ---- ---- 018* :0: 0015
000041: ---- ---- ---- 019* ---- ---- :0:
Branch Miss Prediction 00002

000044: 023* ---- ---- ---- 019* ---- :0:
000045: 024* 023* ---- ---- ---- 019* :0: 0016
000046: 025* 024* 023* ---- ---- ---- :0:
000047: 026* 025* 024* 023* ---- ---- :0:
000048: 026  025  ---- 024* 023* ---- :1:
000049: 026  025  ---- ---- 024* 023* :1: 0017
000050: 027* 026* 025* ---- ---- 024* :0: 0018
000051: ---- ---- ---- 025* ---- ---- :0:
Branch Miss Prediction 00003

000054: 029* ---- ---- ---- 025* ---- :0:
000055: 030* 029* ---- ---- ---- 025* :0: 0019
000056: 031* 030* 029* ---- ---- ---- :0:
000057: 032* 031* 030* 029* ---- ---- :0:
000058: 032  031  ---- 030* 029* ---- :1:
000059: 032  031  ---- ---- 030* 029* :1: 0020
```

最初の命令 (識別子 001) は，サイクル 1 でフェッチされ，その後，順調にパイプラインを流れていく．この命令は，サイクル 6 で実行結果をライトバックして，その処理を完了する．

次の命令 (識別子 002) は，最初の命令 (識別子 001) との間にデータ依存関係を持つ．このため，最初の命令 (識別子 001) のライトバックが終わるまで，レジスタ参照を開始することができない．サイクル 6 で最初の命令 (識別子 001) のライトバックが終了して，必要とするデータがレジスタファイルに格納された後に，サイクル 7 でレジスタ参照の処理を再開する．

サイクル 31 では，分岐予測ミスが検出されている．実行ステージの命令 (識別子 013) が分岐命令であり，その実行結果と分岐予測結果とが異なっている場合に，分岐予測ミスが検出される．このため，最初の 3 つのパイプラインに存在する 3 つの命令 (識別子 014, 015, 016) をフラッシュする．また，分岐予測ミスのペナルティの 2 サイクルが予測ミスからの復帰のために費やされる．このため，分岐予測ミスが検出された後に出力されるログはサイクル 34 となる．

3.3 スカラプロセッサの実装

本節では，スカラプロセッサの実装の詳細をまとめる．構成する一つのインクルードファイルと 8 つの C++のソースファイルの行数を図 3 にまとめる．コード量の合計は 3,063 行と非常に少ない．

```
   385 define.h       **
   873 arithmetic.cc
    78 chip.cc
   204 debug.cc
   316 etc.cc         **
   204 instruction.cc
   180 memory.cc
   270 scalar.cc      **
   553 syscall.cc
  -------------------------
  3063 total
```

図 3: スカラプロセッサのコード量

SimAlpha Version 1.1 を土台として，define.h, etc.cc, scalar.cc という 3 つのファイルに変更が加えられている．SimAlpha Version 1.1 からの変更点は，約 300 行と非常に少ない．

3.3.1 スカラプロセッサ版のソースコード

パイプラインの段数と分岐予測ミスのペナルティ (分岐予測ミスにより生じる副作用を取り除くためのサイクル数) の定義を示す．分岐予測ミスのペナルティには適切な整数を指定する．ここでは 2 サイクルとする．

```
#define PIPELINE_DEPTH     6
#define PBRED_MISS_PENALTY 2
```

スカラプロセッサを実現するクラス scalar_chip の定義と，コンストラクタ，デストラクタを示す．

```
class scalar_chip{
  debug              *deb;
public:
  data_t pc;         /* Program Counter       */
  int depth;         /* Pipeline Depth        */
  instruction **stg; /* Pipeline Stage        */
  int stall;         /* data dependency stall */
  int flush_pal;     /* flush by PAL code     */
  int flush_bpred;   /* flush by branch miss  */
  int work[PIPELINE_DEPTH];
  system_config      *sc;
  system_manager     *sys;
  evaluation_result  *e;
  memory_system      *mem;
  architecture_state *as;
```

```
  scalar_chip(char *, char **);
  ~scalar_chip();
  void step(int *);
  void flush();
  void update();
};

scalar_chip::scalar_chip(char *prog, char **opt){
  depth = PIPELINE_DEPTH;
  stg = new instruction*[depth];
  for(int i=0; i<depth; i++) stg[i] = NULL;
  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);
  pc.init(as->pc.ld());
}

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

ここで定義するアーキテクチャステート as には，インオーダステートが格納される．scalar_chip の pc は命令フェッチのためのもので，アーキテクチャステート内の pc はインオーダに完了した命令の次にフェッチする命令のアドレスを格納する．配列 stg には，命令パイプラインに存在する命令のポインタを格納する．図 2 の下段に，配列 stg と命令パイプラインとの関係を示した．

パイプラインストールを検出するためには，データ依存関係の有無を判定する必要がある．二つの命令のポインタ r(後続命令), w(先行命令) を引数として，それらの間のデータ依存関係の有無を返す関数 data_dep_a と data_dep_b のコードを示す．

```
int data_dep_a(instruction *r, instruction *w){
  if(r==NULL || w==NULL || w->WB==31) return 0;
  int ww = (w->WF << 5) | w->WB;
  int ra = (r->Ai << 6) | (r->Af << 5) | r->RA;
  if(ra==ww) return 1;
  return 0; /* No data dependency */
}

int data_dep_b(instruction *r, instruction *w){
  if(r==NULL || w==NULL || w->WB==31) return 0;
  int ww = (w->WF << 5) | w->WB;
  int rb = (r->Bi << 6) | (r->Bf << 5) | r->RB;
  if(rb==ww) return 1;
  return 0; /* No data dependency */
}
```

入力オペランドは Rav, Rbv という 2 つが存在する．Rav とのデータ依存を求める関数として data_dep_a，Rbv とのデータ依存を求める関数として data_dep_b を利用する．ra, rb という変数には 7 ビットの値を格納する．最上位ビットは，即値からオペランドを生成した場合に 1 となる．次のビットは，浮動小数点レジスタから読み出してオペランドを生成する場合 1 となる．下位の 5 ビットには，読み出したレジスタの番号を格納する．これら 7 ビットの値と変数 ww とを比較することで，依存関係の有無を判定する．

各パイプラインステージの命令に処理を施すメソッド step のコードを示す．

```
void scalar_chip::step(int *run){
  for(int i=0; i<depth; i++) work[i]=0;
  flush_pal   = 0;
  flush_bpred = 0;

  if(stg[5]!=NULL){ /** WriteBack **/
    e->retired_inst++;
    as->pc.st(stg[5]->Npc.ld());
    stg[5]->WriteBack(); work[5]=1;

    if(stg[5]->Op==OP_PAL){
      /* sigreturn updates as->ps */
      stg[5]->Npc.st(as->pc.ld());
      as->pc.st(stg[5]->Npc.ld());
      flush_pal = 1;
    }
    if(stg[5]->CM){ /* CMOV instruction  */
      uint64_t npc = stg[5]->Npc.ld();
      execute_cmovb(stg[5], as);
      stg[5]->Npc.st(npc);
    }
  }

  if(stg[4]!=NULL){ /** Memory Access **/
    stg[4]->Memory(); work[4]=1;
  }

  if(stg[3]!=NULL){ /** Execute **/
    data_t target_pc;
    stg[3]->Execute(&target_pc); work[3]=1;
    if(stg[3]->Cpc.ld()+4 != target_pc.ld())
      flush_bpred = 1;
  }

  if(stg[2]!=NULL){ /** RegisterRead **/
    stg[2]->Issue();
    stg[2]->RegisterRead(); work[2]=1;
```

```
  }

  if(!stall){ /** no data dependency **/
    if(stg[1]!=NULL){ /** I-Decode **/
      stg[1]->Slot(); work[1]=1;
    }

    /*** I-Fetch ***/
    static uint64_t fetchd_inst = 0;
    stg[0]=new instruction(as,mem,sys,sc,e);
    stg[0]->Fetch(&pc); work[0]=1;
    stg[0]->id.st(++fetchd_inst);
    pc.st(pc.ld() + 4);
  }

  house_keeper(sys, sc, e, deb);
  *run = sys->running;
}
```

メソッド step において，命令パイプラインの各ステージに存在する命令の処理をおこなう．ライトバックステージでは，PAL 命令と条件付き移動 (CMOV) 命令に関して，特別な処理を必要とする．条件付き移動命令の実装に関しては，現実的な構成により改良をおこなう必要がある．データ依存関係が検出され変数 stall がセットされている場合に，命令フェッチと命令デコードのステージの処理をストールさせる．

分岐予測ミスが検出された時と PAL コードが実行される時にパイプラインをフラッシュするためのメソッド flush のコードを示す．

```
void scalar_chip::flush(){
  if(flush_pal){
    for(int i=0; i<=4; i++){
      if(stg[i]!=NULL){
        delete stg[i];
        stg[i]=NULL;
      }
    }
    pc.st(stg[5]->Npc.ld());
  }
  else if(flush_bpred){
    for(int i=0; i<=2; i++){
      if(stg[i]!=NULL){
        delete stg[i];
        stg[i]=NULL;
      }
    }
    pc.st(stg[3]->Npc.ld());
  }
}
```

命令パイプラインの状態を更新するためのメソッド update のコードを示す．

```
void scalar_chip::update(){
  stall = 0;
  stall |= data_dep_a(stg[1], stg[2]);
  stall |= data_dep_b(stg[1], stg[2]);

  stall |= data_dep_a(stg[1], stg[3]);
  stall |= data_dep_b(stg[1], stg[3]);

  stall |= data_dep_a(stg[1], stg[4]);
  stall |= data_dep_b(stg[1], stg[4]);

  if(stg[5]!=NULL){ /** Code Retire **/
    delete stg[5];
  }
  stg[5] = stg[4];
  stg[4] = stg[3];
  stg[3] = stg[2];
  stg[2] = NULL;

  if(!stall){
    stg[2] = stg[1];
    stg[1] = stg[0];
    stg[0] = NULL;
  }
}
```

メソッド update の前半において，命令デコードステージの命令と，レジスタ参照 (RR)，実行 (EX)，メモリ参照 (MM) ステージの命令との間の依存関係を解析し，データ依存関係が存在する場合には変数 stall をセットする．後半では，パイプラインステージを進める．ただし，変数 stall がセットされている場合には，最初の 3 つのパイプラインステージをストールさせる．

メイン関数のコードを示す．

```
int main(int argc, char **argv){
  printf("%s\n", VER);
  if(argc==1) usage();
  char *p    = argv[argc-1]; /* program name */
  char **opt = argv;         /* options      */
  scalar_chip *c1 = new scalar_chip(p, opt);

  simple_chip *c2=NULL;
  if(c1->sc->verify_flag){
    char *op[OPTION_MAX];
    option_from_file(op, "verify.txt");
    c2 = new simple_chip(p, op);
  }

  int run=1;
  while(run){
    c1->e->clock++;
    c1->step(&run);
    c1->flush();
    if(c1->sc->trace_flag) print_pipeline(c1);
```

```
    if(c1->sc->verify_flag) verify(c2, c1);
    c1->update();
    if(c1->flush_bpred){
      c1->e->bpred_miss++;
      c1->e->clock += PBRED_MISS_PENALTY;
    }
  }

  if(c1->sc->verify_flag) delete c2;
  delete c1;
  return 0;
}
```

メイン関数では，scalar_chip 型のオブジェクト c1 を生成し，while ループにより処理を進める．このコードには，動作検証のための処理が追加されている．起動オプションとして –z を指定して verify_flag をセットすることで，動作検証のためのプロセッサ c2 が構築される．関数 verify の内部では，1 つの命令の処理が完了した時点で c1 と c2 のアーキテクチャステートを比較し，異なっている場合には処理を停止する．

3.4 データフォワーディングの追加

記述を簡潔にするために，データフォワーディングをおこなわない単純なスカラプロセッサの実装を説明してきた．実際には，僅かな変更で，データフォワーディングの機能を追加することができる．本節では，データフォワーディングを実現するためのコードを示す．

データフォワーディングをおこない，プログラムが始まってから 20 命令が完了するまでの処理の様子を示す．

```
 cycle:  IF   ID   RR   EX   MM   WB  :dependency:
000001: 001* ---- ---- ---- ---- ---- :0:
000002: 002* 001* ---- ---- ---- ---- :0:
000003: 003* 002* 001* ---- ---- ---- :0:
000004: 004* 003* 002* 001* ---- ---- :0:
000005: 005* 004* 003* 002* 001* ---- :0:
000006: 006* 005* 004* 003* 002* 001* :0: 0001
000007: 007* 006* 005* 004* 003* 002* :0: 0002
000008: 008* 007* 006* 005* 004* 003* :0: 0003
000009: 009* 008* 007* 006* 005* 004* :0: 0004
000010: 010* 009* 008* 007* 006* 005* :0: 0005
000011: 011* 010* 009* 008* 007* 006* :0: 0006
000012: 012* 011* 010* 009* 008* 007* :0: 0007
000013: 013* 012* 011* 010* 009* 008* :0: 0008
000014: 014* 013* 012* 011* 010* 009* :0: 0009
000015: 015* 014* 013* 012* 011* 010* :0: 0010
000016: ---- ---- ---- 013* 012* 011* :0: 0011
Branch Miss Prediction 00001

000019: 017* ---- ---- ---- 013* 012* :0: 0012
000020: 018* 017* ---- ---- ---- 013* :0: 0013
000021: 019* 018* 017* ---- ---- ---- :0:
000022: 020* 019* 018* 017* ---- ---- :0:
000023: 021* 020* 019* 018* 017* ---- :0:
000024: ---- ---- ---- 019* 018* 017* :0: 0014
Branch Miss Prediction 00002

000027: 023* ---- ---- ---- 019* 018* :0: 0015
000028: 024* 023* ---- ---- ---- 019* :0: 0016
000029: 025* 024* 023* ---- ---- ---- :0:
000030: 026* 025* 024* 023* ---- ---- :0:
000031: 027* 026* 025* 024* 023* ---- :0:
000032: ---- ---- ---- 025* 024* 023* :0: 0017
Branch Miss Prediction 00003

000035: 029* ---- ---- ---- 025* 024* :0: 0018
000036: 030* 029* ---- ---- ---- 025* :0: 0019
000037: 031* 030* 029* ---- ---- ---- :0:
000038: 032* 031* 030* 029* ---- ---- :0:
000039: 033* 032* 031* 030* 029* ---- :0:
000040: ---- ---- ---- 031* 030* 029* :0: 0020
```

データフォワーディングを利用しない場合には 20 命令の処理に 59 サイクルを要したが，データフォワーディングを用いることで 40 サイクルへと減少することがわかる．

3.4.1 データフォワーディングを追加したソースコード

データフォワーディングを追加するために変更を加えた update のコードを示す．

```
void scalar_chip::update(){
  stall = 0;
  stall |= data_dep_a(stg[1], stg[2]);
  stall |= data_dep_b(stg[1], stg[2]);
  stall &= ((stg[2]!=NULL) &
            (stg[2]->LD | stg[2]->CM));

  if(stg[5]!=NULL){ /** Code Retire **/
    delete stg[5];
  }
  stg[5] = stg[4];
  stg[4] = stg[3];
  stg[3] = stg[2];
  stg[2] = NULL;

  if(!stall){
    stg[2] = stg[1];
    stg[1] = stg[0];
    stg[0] = NULL;
  }
}
```

ロード命令と，それが生成したデータを利用する命令が連続してパイプラインに投入された場合にデータ依存関係によるストールが発生する．また，本実装では，条件付き移動命令の結果はライトバックステージにおいて生

成されるために，条件付き移動命令とのデータ依存関係が発生した場合にもストールさせる．これらを実現するために，変数 stall の生成の条件が変更されている．

データフォワーディングを追加するために変更を加えた step の実行ステージのコードを示す．

```
if(stg[3]!=NULL){ /** Execute **/

  instruction *p = stg[3];  /* FWD */
  if(data_dep_a(p, stg[5])) /* FWD */
    p->Rav = stg[5]->Rcv;   /* FWD */
  if(data_dep_b(p, stg[5])) /* FWD */
    p->Rbv = stg[5]->Rcv;   /* FWD */
  if(data_dep_a(p, stg[4])) /* FWD */
    p->Rav = stg[4]->Rcv;   /* FWD */
  if(data_dep_b(p, stg[4])) /* FWD */
    p->Rbv = stg[4]->Rcv;   /* FWD */

  data_t target_pc;
  stg[3]->Execute(&target_pc); work[3]=1;
  if(stg[3]->Cpc.ld()+4 != target_pc.ld())
    flush_bpred = 1;
}
```

関数 step 内の実行ステージのコードにおいて，処理を開始する前に，必要なデータをフォワーディングする処理を追加する．

これら 2 つの関数の変更により，データフォワーディングの処理を追加することができる．

3.5 動作速度とプロセッサ性能の測定結果

SimAlpha の動作速度の測定結果を表 2 にまとめる．比較対象として，表の 2 列目に SimAlpha Version 1.1 の動作速度 (MIPS) を示す．表の 3 列目に動作検証を利用しないスカラプロセッサシミュレータの動作速度を，4 列目には動作検証を利用した場合の動作速度をまとめる．表 2 にまとめたスカラプロセッサは，データフォワーディングを利用しない版を利用する．各ベンチマークの入力データセットは文献 [5] と同じものを利用する．

SimAlpha Version 1.1 のシミュレーション速度 2.10 MIPS に対して，スカラプロセッサ版 SimAlpha のシミュレーション速度は 0.88 MIPS まで低下することがわかる．また，スカラプロセッサ版 SimAlpha に動作検証を追加した場合には，0.35 MIPS まで低下する．機能レベルシミュレータとクロックレベルのパフォーマンスシミュレータの間には一桁以上の速度差が生じることが一般的である．しかしながら，構築したアーキテクチャが，単純なスカラプロセッサであり，分岐予測やキャッシュを実装していないために，そこまで大きな速度低下は起こっていない．

表 2: シミュレーション速度 (MIPS: Million Instructions Per Second)

Program	version 1.1	scalar	verify
099.go	2.05	0.90	0.35
124.m88ksim	2.09	0.88	0.35
126.gcc	2.11	0.89	0.35
129.compress	2.12	0.90	0.35
130.li	2.13	0.87	0.35
132.ijpeg	2.18	0.84	0.37
134.perl	2.09	0.86	0.35
147.vortex	2.08	0.94	0.36
average	2.10	0.88	0.35

表 3: スカラプロセッサの性能 (CPI: clock cycles per instruction)

Program	no forwarding	forwarding
099.go	2.43	1.68
124.m88ksim	2.50	1.75
126.gcc	2.48	1.78
129.compress	2.29	1.60
130.li	2.46	1.84
132.ijpeg	2.07	1.43
134.perl	2.54	1.81
147.vortex	1.89	1.53
harmonic mean	2.30	1.66

実装したスカラプロセッサの CPI (clock cycles per instruction) の測定結果を表 3 にまとめる．CPI は 1 命令を処理するために必要となる平均サイクル数を意味するので，この値が低い方がプロセッサとしての性能が高い．本稿で実装したスカラプロセッサは，サイクル当たり高々 1 命令しかフェッチしないスカラプロセッサであり，CPI の上限は 1 である．表の 2 列目にはデータフォワーディングを利用しない場合の CPI を示す．表の 3 列目にはデータフォワーディングをおこなう場合の CPI を示す．調和平均を用いて比較すると，データフォワーディングを利用しない場合の CPI 2.30 に対して，データフォワーディングを用いることで CPI 1.66 まで，性能を向上できることを確認できる．

3.6 議論

SimAlpha Version 1.1 を用いて，パイプライン化されたスカラプロセッサのシミュレータを構築した．

スカラプロセッサのシミュレータを構築するためには，データ依存関係と制御依存関係を考慮して，パイプラインの状態を更新すればよい．SimAlpha を用いることで，短いコード量で，これらの記述が可能となる．SimAlpha Version 1.1 を土台として，スカラプロセッサのシミュレータを構築するための変更点は，約 300 行と非常に少ない．

本稿で示した実装は 6 段の命令パイプラインを持つ単純なアーキテクチャのスカラプロセッサであるが，命令パイプラインの深さや構成は，容易に変更可能である．近年のプロセッサ構成は，命令パイプラインの段数を増やして，動作周波数の向上を狙う傾向にある．SimAlpha は，パイプライン構成を変更して様々な挙動を測定するような場合のシミュレータの構築に適している．

シミュレータを短期間で効率的に構築するためには，その動作検証をおこないながら実装を進めていくことが重要となる．SimAlpha を用いることで，一つの命令の処理が完了した時点で，アーキテクチャステートの内容を検証する機能を簡潔に追加できることを示した．この機能を利用することで，シミュレータ構築の時間を大幅に短縮できる．

4 おわりに

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プロセッサシミュレータ SimAlpha の開発を続けている．

新たに，パイプライン化されたスカラプロセッサのシミュレータを構築した．本稿では，構築したスカラプロセッサ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，そのソフトウェアアーキテクチャを明らかにした．また，構築したスカラプロセッサ版のシミュレータの動作速度及び，構築したプロセッサの性能を測定した．

本稿の前半では，動作検証の機能を効率的に実装するため，SimAlpha Version 1.0 に拡張を施した．この版を SimAlpha Version 1.1 とし，その詳細を示した．

本稿の後半では，SimAlpha Version 1.1 をベースとして，6 段の命令パイプライン構成を持つスカラプロセッサのシミュレータを構築した．また，シミュレータ構築の信頼性を向上させるために，構築したスカラプロセッサシミュレータに動作検証の仕組みを追加した．

SimAlpha Version 1.1 と，本稿で示したスカラプロセッサの実装は次の URL からダウンロードできる．

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

参考文献

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

[2] The MicroLib.org Project Homepage. http://www.microlib.org/.

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

[5] 吉瀬謙二, 本多弘樹, 弓場敏嗣. SimAlpha: C++で記述したもうひとつの Alpha プロセッサシミュレータ. 情報処理学会研究報告 2002-ARC-149, pp. 163–168, August 2002.

[6] 吉瀬謙二, 本多弘樹, 弓場敏嗣. 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. Technical Report UEC-IS-2002-2, 電気通信大学 大学院情報システム学研究科, July 2002.