Technical Report UEC-IS-2003-5, Version 2003-07-07
Graduate School of Information Systems, University of Electro-Communications

SimAlpha-Loader の実装とクロス開発環境の構築

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

SimAlpha の独自形式の入力ファイルを生成するローダ SimAlpha-Loader Version 1.0 の実装に関して述べる．また，Intel PC 上で Alpha アーキテクチャの実行ファイルを構築するためのアセンブラとリンカによるクロス開発環境の構築方法に関して述べる．これらの環境を利用すれば，アセンブラで記述した命令列を，SimAlpha の実行形式に変換することができる．SimAlpha-Loader Version 1.0 と，構築したクロス開発環境を用いることで，通常のコンパイル環境で生成した COFF 形式と ELF 形式の実行ファイルから，SimAlpha の独自形式の実行ファイルを構築することができる．

キーワード　SimAlpha, SimAlpha-Loader, ローダ

Implementation of SimAlpha-Loader and Construction of Cross-Development Environment

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

Implementation of SimAlpha-Loader Version 1.0 which generates the input file of the SimAlpha original format is described. The construction of the cross-development environment for building the executable file of the Alpha architecture on Intel PC is described. By using the constructed environment, the executable file of the original format of SimAlpha can be build from the executable file of the COFF and ELF format generated in the common compile environment.

Key-words　SimAlpha, SimAlpha-Loader, loader

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, 5] 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[6, 7] の開発を続けている．

本稿では，SimAlpha から切り離していた実行ファイルのローダ SimAlpha-Loader の実装に関して述べる．また，GNU binutils を利用して，Intel PC 上で Alpha の実行ファイルを構築するためのアセンブラとリンカのクロス開発環境を構築する．これらの環境を利用すれば，アセンブラで記述した命令列を，SimAlpha の実行形式に変換することができる．SimAlpha-Loader Version 1.0 と，構築したクロス開発環境を用いることで，通常のコンパイル環境で生成した COFF(Common Object File Format) 形式と ELF(Executable and Linking Format) 形式の実行ファイルから，SimAlpha の独自形式の実行ファイルを構築することができる．幾つかのアセンブラコードを例に出しながら，SimAlpha の実行ファイルの構築方法を示す．

本稿の構成を示す．2 章で SimAlpha の実行イメージファイルを構築するためのローダの実装に関して述べる．3 章では，Intel PC 上で Alpha の実行ファイルを構築するためのクロス開発環境の構築方法をまとめる．4 章で

は，構築した開発環境においてアセンブラによる Alpha の実行ファイルの例を示す．5 章で本稿をまとめる．

2 SimAlpha-Loader

2.1 実行ファイルの形式

SimAlpha を利用するためには，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，独自形式の実行ファイルを入力とする．独自形式の実行ファイルを用いる利点は，COFF や ELF 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 1: SimAlpha の実行ファイルの例

実行ファイルの一部を図 1 に示す．実行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 1 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 1 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．

2.2 SimpleScalar を利用した実行イメージファイルの作成手法

本節では，図 1 に示す形式の実行イメージファイルを SimpleScalar のローダを利用して作成する方法を示す．

下に示すコードを 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);
}
```

上のコードを 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．

2.3 SimAlpha-Loader の実装

本節では，図 1 に示す形式の実行イメージファイルを作成するローダ　SimAlpha-Loader Version 1.0 の実装をまとめる．

COFF 形式の実行ファイルを入力として，SimAlpha の実行ファイルを出力するプログラム SimAlpha-Loader Version 1.0 を構築した．Tru64 UNIX が標準で生成する実行ファイル形式は COFF 形式であるため，COFF 形式の実行ファイル SimAlpha-Loader の入力とした．Linux 等で標準となっている ELF 形式の実行ファイルを COFF 形式に変換する方法は，3 章において説明する．

SimAlpha-Loader は C++ を用いて記述されている．SimAlpha-Loader Version 1.0 に含まれるソースコード

とインクルードファイルの行数をまとめる.

```
    149 etc.cc
    187 main.cc
    155 memory.cc
    375 define.h
     42 loader.h
  -----------------
    908 total
```

コード量の合計は 908 行と非常に少ない.

ソースコード内で利用される構造体を定義する loader.h の内容を示す.

```
struct DU_filehdr {
  unsigned short f_magic;
  unsigned short f_nscns; /* use this */
  int            f_timdat;
  long long      f_symptr;
  int            f_nsyms;
  unsigned short f_opthdr;
  unsigned short f_flags;
};

struct DU_aouthdr {
  short magic;
  short vstamp;
  short   bldrev;
  short   padcell;
  long long tsize;      /* use this */
  long long dsize;      /* use this */
  long long bsize;
  long long entry;      /* use this */
  long long text_start;
  long long data_start;
  long long bbs_start;
  int   gprmask;
  int   fprmask;
  long long     gp_value;
};

struct DU_scnhdr {
  char        s_name[8]; /* use this */
  long long   s_paddr;   /* use this */
  long long   s_vaddr;
  long long   s_size;    /* use this */
  long long   s_scnptr;  /* use this */
  long long   s_relptr;
  long long   s_lnnoptr;
  unsigned short s_nreloc;
  unsigned short s_nlnno;
  int           s_flags;
};
```

SimAlpha-Loader のメイン関数を示す.

```
int main(int argc, char **argv){
  fprintf(stderr, "%s\n", VER);
  if(argc==1) usage_loader();
  char *p = argv[1]; /* program name */
  architecture_state as;
  memory_system mem;

  format_check(p);
  init_section(p, &mem, &as);
  init_stack(argc-1, &argv[1], &mem, &as);
  create_image(&as, &mem);
  return 0;
}
```

実行ファイルを解析して得られるテキストやデータの情報を格納するために，アーキテクチャステート as と，メモリシステム mem を生成する．関数 format_check は，実行ファイルの名前を引数として，入力ファイルの形式が適切かどうかを判定する．関数 init_section は，テキストやデータセクション等の内容を，メモリシステムとアーキテクチャステートに格納する．関数 init_stack は，アプリケーションを起動する際の引数から，スタック領域を設定する．これらの設定が終了してから，関数 create_image により，独自形式の実行ファイルを生成する．

ファイルの形式を検査する関数 format_check のコードを示す.

```
void format_check(char *p){
  FILE *fp;
  struct DU_filehdr hdr;
  struct DU_aouthdr aout;
  struct DU_scnhdr  st;

  if ((fp = fopen(p, "r")) == NULL) {
    fprintf(stderr, "Bad file name: %s\n", p);
    exit(0);
  }
  fread(&hdr,  sizeof(hdr), 1, fp);
  fread(&aout, sizeof(aout),1, fp);

  fprintf(stderr, "hdr.f_magic: %d,\n",
          hdr.f_magic);
  if(hdr.f_magic!=387){ /* elf 17791 */
    printf("This is not coff executable: %s.\n",p);
    exit(0);
  }

  for(int i=0; i<hdr.f_nscns; i++){
    fread(&st, sizeof(st), 1, fp);
    fprintf(stderr, "name: %8s  %09qx - %09qx \n",
            st.s_name, st.s_paddr,
            st.s_paddr + st.s_size);
    if(!strncmp(st.s_name, ".dynamic", 8)){
      printf("Using the shared library.\n");
      printf("opt -static for gcc.\n");
      printf("opt -non_shared for Digital.\n");
      exit(0);
```

```
    }
  }
  fclose(fp);
}
```

指定したファイル名のファイルが存在しない場合には，コメントを表示して終了する．ファイルヘッダのマジックナンバーが 387 でない場合には，COFF 形式のファイルではないので，コメントを表示して終了する．SimAlpha は，静的にライブラリをリンクしていない実行ファイルのシミュレーションをサポートしていない．このため，静的にリンクされていない実行ファイルの場合には，コメントを表示して終了する．

セクションの内容を設定する関数 init_section のコードを示す．

```
void init_section(char *p, memory_system *mem,
                  architecture_state *as){
  struct DU_filehdr hdr;
  struct DU_aouthdr aout;
  struct DU_scnhdr  st;

  FILE *fp = fopen(p, "r");
  fread(&hdr, sizeof(hdr), 1, fp);
  fread(&aout,sizeof(aout),1, fp);

  for(int i=0; i<hdr.f_nscns; i++){
    fread(&st, sizeof(st), 1, fp);
    if(!strcmp(st.s_name, ".sbss") ||
       !strcmp(st.s_name, ".bss") ||
       !strcmp(st.s_name, ".comment")) continue;

    int size = st.s_size / 4;
    int *data = new int[size];
    {
      FILE *f = fopen(p, "r");
      fseek(f, st.s_scnptr, SEEK_SET);
      fread(data, st.s_size, 1, f);
      fclose(f);
    }
    for(int j=0; j<size; j++){
      data_t dat, adr;
      dat.st(data[j]);
      adr.st(st.s_paddr + 4*j);
      mem->st_nbyte(4, &adr, &dat); /* store */
    }
    delete data;
  }
  fclose(fp);
  as->pc.st(aout.entry); /* Program Counter */
}
```

3 つのセクション (.sbss, .bss, .comment) を除いて，各セクションの内容をメモリシステム mem に格納する．また，ヘッダから取得した情報を用いて，プログラムの開始アドレスを設定する．

アプリケーションプログラムの起動時の引数を用いて，スタック領域を設定する関数 init_stack のコードを示す．

```
#define BASE_STACK 0x11ff97000ull
#define BASE_ARGV  0x11ff98000ull
#define BASE_ARGP  0x11ff99000ull
void init_stack(int argc, char **argv,
                memory_system *mem,
                architecture_state *as){
  data_t dat, adr;
  adr.st(BASE_STACK);
  as->r[30] = adr; /* stack pointer */

  int argpc = 1;
  static char *argp[] ={"HOME=/"};

  /***** Error Checck *****/
  if(argc>15 || argpc>15){
    printf("Error: argc %d, argpc %d\n",
           argc, argpc);
    exit(1);
  }
  /***** argc *****/
  dat.st(argc);
  mem->st_nbyte(8, &adr, &dat);
  /***** argv *****/
  for(int i=0; i<argc; i++){
    adr.st(adr.ld()+8);
    dat.st(BASE_ARGV + 0x100*i);
    mem->st_nbyte(8, &adr, &dat);

    char *c = argv[i];
    for(unsigned int j=0; j<strlen(c); j++){
      data_t d, a;
      a.st(dat.ld() + j);
      d.st(c[j]);
      mem->st_nbyte(1, &a, &d);
    }
  }
  adr.st(adr.ld()+8);
  /***** argp *****/
  for(int i=0; i<argpc; i++){
    adr.st(adr.ld()+8);
    dat.st(BASE_ARGP + 0x100*i);
    mem->st_nbyte(8, &adr, &dat); /* argp */

    char *c = argp[i];
    for(unsigned int j=0; j<strlen(c); j++){
      data_t d, a;
      a.st(dat.ld() + j);
      d.st(c[j]);
      mem->st_nbyte(1, &a, &d);
    }
  }
}
```

本実装では，プログラム開始時のスタックポインタの値を 0x11ff97000 に固定した．アプリケーションプログラムの実行引数 ARGV の格納アドレスを 0x11ff9800 に固定した．環境変数 ARGP の格納アドレスを 0x11ff9900 に固定した．環境変数は利用している計算機システムから取得すべきだが，それでは，実行するディレクトリやユーザ名などにより，シミュレーションの結果が異なってしまう．シミュレーション結果を計算機の環境変数と非依存とするために，環境変数を以下の様に固定している．

```
static char *argp[] ={"HOME=/"};
```

スタック領域のメモリ配置例を図 2 に示す．この例では，ARGV が 3 個，ARGP が 1 個の場合の配置を示している．

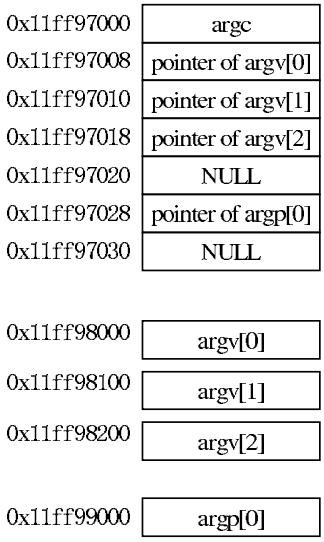

図 2: スタック領域のメモリ配置例

SimAlpha の独自形式により実行ファイルの内容を出力する関数 create_image のコードを示す．

```
void create_image(architecture_state *as,
                  memory_system *mem){
  printf("/* SimAlpha 1.0 Image File */\n");

  printf("/*** Registers ***/\n");
  for(int i=0; i<32; i++)
    if(as->r[i].ld())
      printf("/@reg %02d %016qx\n",
             i, as->r[i].ld());
  printf("/@pc  32 %016qx\n", as->pc.ld());

  printf("/*** Memory    ***/\n");
  for(int i=0; i<BLOCK_TABLE_SIZE; i++){
    if(mem->mm->block_table[i]!=NULL){
      for(int j=0; j<BLOCK_SIZE/8; j++){
        data_t dat, adr;
        adr.st(0x100000000ull + (i<<13) + j*8);
        mem->ld_nbyte(8, &adr, &dat);
        if(dat.ld()!=0){
          printf("@%qx %qx\n",
                 adr.ld(), dat.ld());
        }
      }
    }
  }
}
```

2.4 SimAlpha-Loader の利用方法

アプリケーションプログラムの起動引数として 9 9 を指定して，go.decc_O4 を実行する場合の SimpleScalar のコマンドを示す．

```
$ sim-safe go.decc_O4 9 9
```

同様のシミュレーションを SimAlpha を用いて行う場合には，まず，SimAlpha-Loader を利用して独自形式の実行ファイルを作成し，その実行ファイルを用いてシミュレーションをおこなう．対応するコマンドを以下に示す．

```
$ SimAlpha-Loader go.decc_O4 9 9 > aout.txt
$ SimAlpha aout.txt
```

3 クロス開発環境の構築

GNU binutils を用いて，SimAlpha の簡単な動作検証のためのクロス開発環境を構築する．RedHat Linux 7.3 が動作している Intel PC 上で，Tru64 UNIX が動作している Alpha アーキテクチャのための実行ファイルを作成する方法をまとめる．

なお，Linux が動作している Alpha アーキテクチャで動作する ELF 形式の実行ファイルも同様の方法で構築することができる．

3.1 GNU binutils のインストール

GNU binutils に含まれる主なプログラムの名前と役割を示す．

alpha-linux-as アセンブラ，アセンブラコード (拡張子.s) をオブジェクトファイル (拡張子.o) に変換する．

alpha-linux-ld リンカ，オブジェクトファイル (拡張子.o) を実行形式のファイル (ファイル名 a.out) に変換する．

alpha-linux-objdump オブジェクトファイル (ファイル名 a.out) の内容を表示する．

alpha-linux-objcopy オブジェクトファイル (ファイル名 a.out) の形式を変換する．

標準的な拡張子とファイル名を括弧内に示した．ターゲットを alpha-linux として GNU binutils をコンパイルするので， alpha-linux- という文字がプログラム名の先頭に付加される．

GNU binutils をインストールする方法をまとめる．GNU のサイト [1] から binutils-2.13.1.tar.gz をダウンロードして，作業ディレクトリにコピーする．alpha-linux をターゲットとして，GNU binutils をコンパイル及びインストールするためのコマンドを示す．

```
$ tar xvfz binutils-2.13.1.tar.gz
$ cd binutils-2.13.1
$ mkdir alpha-linux
$ cd alpha-linux
$ ../configure --target=alpha-linux
                --prefix=/home/kis/bin/gnu/
$ make
$ make install
```

インストールや変更を頻繁におこなう様な場合には，ユーザの権限でユーザのホームディレクトリ以下にインストールすればよい．インストール先のディレクトリを指定するために， configure を起動オプション –prefix を利用する．上のコマンドでは /home/kis/bin/gnu/ 以下に生成されたファイルがインストールされる．ここに示したディレクトリが存在しないと，コンパイル時にエラーとなるので，事前にディレクトリを作成しておかなければならない．インストールの終了後，–prefix で指定したディレクトリをシェルの実行パスに加えておく．

3.2 クロス開発環境の動作確認

システムコールの exit のみを実行するシンプルなアセンブラコードを利用して GNU binutils の動作を確認する．main.s の内容を示す．

```
.global start
.data
        .byte 0
.text
start:
        mov 1, $0
        call_pal 0x83
```

データセクションが存在しないと SimpleScalar の起動時に警告が表示されるので，1 バイトの値 0 を格納するデータセクションを作成した．テキストセクションでは，汎用レジスタ 0 に値 1 を設定して，call_pal 命令を起動する．これにより，システムコールの exit を実行して，プログラムを終了する．命令はテキストセクションに記述し，プログラム開始の場所を start というラベルにより明示的に指定する．リンカからこのラベルが見えるように，.global という指示子を利用する．

オブジェクトファイル a.out を生成するための Makefile の内容を示す．

```
TARG   = a.out
OPT    = -entry=start
FORMAT = --oformat ecoff-littlealpha

$(TARG): main.s
        alpha-linux-as main.s -o main.o
        alpha-linux-ld $(FORMAT) $(OPT) main.o
```

COFF 形式のオブジェクトファイルを生成するために，リンカのオプション –oformat ecoff-littlealpha を利用する．

アセンブラコードに記述したラベル start をプログラムの開始アドレスとして利用するために， -entry=start オプションを利用する．

alpha-linux-objdump を用いてディスアセンブルすることで，構築したオブジェクトファイル a.out の内容を確認する．コマンドと出力結果を示す．

```
$ alpha-linux-objdump -d a.out

a.out:     file format ecoff-littlealpha

Disassembly of section .text:

0000000120000170 <start>:
   120000170:   00 34 e0 47     mov     0x1,v0
   120000174:   83 00 00 00     callsys
```

命令アドレスと，16 進数で表示した内容とアセンブラが表示される．

SimAlpha-Loader を利用して，構築した実行ファイルを独自形式のファイル aout.txt に変換した内容を示す．

```
/* SimAlpha 1.0 Image File */
/*** Registers ***/
/@reg 30 000000011ff97000
/@pc  32 0000000120000170
/*** Memory    ***/
@11ff97000 1
@11ff97008 11ff98000
```

```
@11ff97010 11ff99000
@11ff98000 74756f2e61
@11ff99000 2f3d454d4f48
@120000170 8347e03400
```

aout.txt の内部では，図 2 に示した様にスタック領域が構成されていることを確認できる．

3.2.1 リンカスクリプトを用いたアドレス指定

先に示したアセンブラの例では，ディスアセンブルして得られるプログラムの開始アドレスが 0x120000170 となっている．リンカスクリプトを利用することで，各セクションのアドレスを指定することができる．テキストセクションの開始アドレスを 0x120007000, データセクションの開始アドレスを 0x140000000 に指定するリンカスクリプト linkerscript の内容を示す．

```
SECTIONS
{
  . = 0x120007000;
  .text : { *(.text) }
  . = 0x140000000;
  .data : { *(.data) }
}
```

リンカスクリプトを利用する場合の Makefile の内容を示す．

```
TARG   = a.out
OPT    = -entry=start -T linkerscript
FORMAT = --oformat ecoff-littlealpha

$(TARG): main.s
        alpha-linux-as main.s -o main.o
        alpha-linux-ld $(FORMAT) $(OPT) main.o
```

リンカのオプション -T を用いて，リンカスクリプトのファイル名 linkerscript を指定する．

3.2.2 ELF 形式と COFF 形式に関する補足

オペレーティングシステムとして Linux を動作させている場合には標準で ELF 形式の実行ファイルが生成され，Tru64 Unix を動作させている場合には標準で COFF 形式の実行ファイルが生成される．これらの形式を変換するためには，以下のコマンドを利用する．

ELF 形式のオブジェクトファイル obj を COFF 形式のオブジェクトファイル a.out に変換するコマンドを示す．

```
$ alpha-linux-objcopy -O alpha-dec-osf obj a.out
```

COFF 形式のオブジェクトファイル obj を ELF 形式のオブジェクトファイル a.out に変換するコマンドを示す．

```
$ alpha-linux-objcopy -O elf64-alpha obj a.out
```

GNU binutils を用いて，標準で COFF 形式を採用するツールを作成するためには target として alpha-dec-osf を指定する．構築するためのコマンドを示す．

```
$ tar xvfz binutils-2.13.1.tar.gz
$ cd binutils-2.13.1
$ mkdir alpha-dec-osf
$ cd alpha-linux
$ ../configure --target=alpha-dec-osf
               --prefix=/home/kis/bin/gnu/
$ make
$ make install
```

ただし，ターゲットを alpha-dec-osf として構築したツールの動作は不安定でありお勧めできない．先に示した様に target として alpha-linux を指定して，必要に応じて COFF 形式に変換する方が安定動作する．

4 サンプルプログラム

Intel PC 上で Alpha-AXP アーキテクチャのアセンブラプログラミング環境を構築することができた．ライブラリが提供されていないので，原始的なプログラミング環境だが，アセンブラによるプログラミング環境はシミュレータのデバッグ等に効果的である．ここでは，幾つかのアセンブラによるプログラム例を示す．

4.1 レジスタ間の加算: sum

汎用レジスタ 10 に値 1 をセットし，汎用レジスタ 11 に値 2 をセットし，これら 2 つのレジスタの値の和を求めるアセンブラコード main.s の内容を示す．

```
.global start
.data
        .byte 0
.text
start:
        mov 1, $10
        mov 2, $11
        addq $10, $11, $12
callsys_exit:
        mov 1, $0
        call_pal 0x83
```

Alpha-AXP では，64 ビット長のレジスタを quad word と呼ぶ．加算のニーモニック addq は，quad word の加算を意味する．

アセンブラ，リンク，SimAlpha の独自実行形式への変換というコマンドの流れを下に示す．

```
$ alpha-linux-as main.s -o main.o
$ alpha-linux-ld --oformat ecoff-littlealpha
                 -entry=start main.o
$ SimAlpha-Loader a.out > aout.txt
$ SimAlpha -d aout.txt
```

得られた SimAlpha の実行ファイル aout.txt の内容を示す．

```
/* SimAlpha 1.0 Image File */
/*** Registers ***/
/@reg 30 000000011ff97000
/@pc  32 0000000120000170
/*** Memory    ***/
@11ff97000 1
@11ff97008 11ff98000
@11ff97018 11ff99000
@11ff98000 74756f2e61
@11ff99000 2f3d454d4f48
@120000170 47e0540b47e0340a
@120000178 47e03400414b040c
@120000180 83
```

SimAlpha を起動して，ステップ実行をおこなうことで，汎用レジスタ 12 に値 3 が格納されることを確認できる．

4.2 1から 100 の合計値: loop

1 から 100 までの合計値を求めるアセンブラコード main.s の内容を示す．

```
.global start
.data
       .byte 0
.text
start:
        mov   0, $11
        mov 100, $10
loop1:
        addq $11,$10,$11
        subq $10,  1,$10
        bne  $10,loop1
callsys_exit:
        mov 1, $0
        call_pal 0x83
```

汎用レジスタ 11 に加算結果を格納する．汎用レジスタ 10 の初期値を 100 として，毎ループ値をデクリメントする．また，汎用レジスタ 10 の値が 0 でなかったら，ラベル loop1 に分岐して，加算を繰り返す．bne は branch if register not equal to zero の略で，レジスタの値がゼロでない場合に分岐する命令である．終了時には，汎用レジスタ 11 に値 5050 が格納される．

上のアセンブラコードは，下に示す C++のコードに対応する．

```
int main(){
  int sum=0;
  for(int i=100; i!=0; i--) sum += i;
}
```

汎用レジスタの名前を適切な変数名に置き換えることで，アセンブラコードの可読性が向上する．指示子 define を用いて書き換えたアセンブラコードを示す．

```
#define   i    $10
#define sum    $11

.global start
.data
       .byte 0
.text
start:
        mov   0, sum
        mov 100, i
loop1:
        addq sum, i, sum
        subq i,  1, i
        bne  i, loop1
callsys_exit:
        mov 1, $0
        call_pal 0x83
```

上のコードは，プリプロセッサ cpp を利用して定義を展開した後に，alpha-linux-as を用いてアセンブルする．

4.3 THE WORLD の表示: write

THE WORLD という文字列を画面に出力するアセンブラコード main.s の内容を示す．リンカスクリプトを用いて，データセクションの開始アドレスを 0x140000000 に設定している．ソースコードに行番号を付加した．

```
 1  .global start
 2  .data
 3          .ascii "THE WORLD\n"
 4  .text
 5  start:
 6  callsys_write:
 7          mov  1, $16
 8          ldah $17, 0x1400
 9          sll  $17, 4
10          mov  10, $18
```

```
11          mov  4, $0
12          call_pal 0x83
13  callsys_exit:
14          mov 1, $0
15          call_pal 0x83
```

7行目は，システムコール writeの最初の引数として，標準出力 stdoutを意味する1を格納する．8行目から9行目で，システムコール writeが書き込む値を保持するアドレスを設定する．ここでは，レジスタ17にデータセクションのアドレス0x140000000を設定する．10行目は，システムコール writeで書き込む (画面に出力する) 文字列としてレジスタ18に10を格納する．11行目は，レジスタ0にシステムコール writeを意味する値4を格納する．

4.4 レジスタの内容表示: print

16桁の16進数でレジスタの内容を表示するアセンブラコードを示す．

```
.global start
.data
        .ascii "0123456789abcdef"
        .ascii "xxxxxxxxxxxxxxxx\n"
.text
start:
        ldah $8, 0x7777

##### data to print $8
        mov 64, $12
        ldah $11, 0x1400
        sll  $11, 4
        addq $11, 16, $10
loop_char:
        subq $12, 4, $12
        srl $8, $12, $19
        and $19, 0xf, $19
write_one_hex_digit:
        addq $11, $19, $17
        ldb $9, 0($17)
        stb $9, 0($10)
        addq $10, 1, $10
        bne $12, loop_char

        mov 1, $16
        addq $11, 16, $17
        mov  17, $18
        mov  4, $0
        call_pal 0x83
callsys_exit:
        mov 1, $0
        call_pal 0x83
```

データセクションの最初の16バイトに，16進数を構成する16種類の文字を格納する．レジスタの内容に応じて表示する文字を選択し，続く16文字の領域に格納していく．最後に，システムコール writeにより，改行を含む17文字を表示してプログラムを終了する．

4.5 配列要素間の演算: array

メモリ間の演算の例として，2つの配列 A, Bの各要素の差分の合計を計算するプログラムを考える．まず，対応する C++のコードを示す．

```
int main(){
  long long A[5] = {0,4,0x300,5,0x4000};
  long long B[5] = {0,2,0x200,4,0x1000};
  int sum=0;
  for(int i=4; i!=0; i--) sum += A[i]-B[i];
  printf("%016qx\n", sum);
}
```

配列要素間の演算をおこなうコードのアセンブラによる記述の一部を示す．

```
 1  #define   i    $2
 2  #define   A    $3
 3  #define   B    $4
 4  #define dif    $5
 5  #define adr    $6
 6  #define sum    $8
 7
 8  .global start
 9  .data
10          .ascii "0123456789abcdef"
11          .ascii "xxxxxxxxxxxxxxxx\n"
12  .org 0x100
13          .quad 0,4,0x300,5,0x4000
14  .org 0x200
15          .quad 0,2,0x200,4,0x1000
16  .text
17  start:
18          ldah adr, 0x1400
19          sll  adr, 4
20          mov 0, sum
21          mov 5, i
22  loop1:
23          ldq A, 0x100(adr)
24          ldq B, 0x200(adr)
25          addq adr, 8, adr
26
27          subq A, B, dif
28          addq sum, dif, sum
29
30          subq i, 1, i
31          bne i, loop1
```

指示子 orgを利用して，配列 Aを 0x140000100に，配列 Bを 0x140000200に配置した．

5 おわりに

本稿では，SimAlpha の独自形式の入力ファイルを生成するローダ SimAlpha-Loader の実装に関して述べた．また，Intel PC 上で Alpha アーキテクチャの実行ファイルを構築するためのアセンブラとリンカによるクロス開発環境の構築方法をまとめた．ライブラリが提供されていないので，原始的なプログラミング環境だが，アセンブラによるプログラミング環境はシミュレータのデバッグ等に効果的である．

本稿で示した環境を利用すれば，アセンブラで記述した命令列を，SimAlpha の実行形式に変換することができる．SimAlpha-Loader Version 1.0 と，構築したクロス開発環境を用いることで，通常のコンパイル環境で生成した COFF 形式と ELF 形式の実行ファイルから，SimAlpha の独自形式の実行ファイルを構築することができる．

環境構築の方法に加えて，幾つかのアセンブラコードを例に出しながら，SimAlpha の実行ファイルを生成するための具体的な方法を説明した．

SimAlpha-Loader Version 1.0 と，本稿で示したサンプルコードは次の URL からダウンロードできる．

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

参考文献

[1] GNU's Not Unix! http://www.gnu.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] John R. Levine. *Linkers & Loaders*. オーム社, 2001.

[5] 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.

[6] 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.

[7] 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.