# ISIS-SimpleScalar User's Guide

# 目 次

# 第 1 章

# プラットフォーム

現在、以下のプラットフォームでの動作が確認済みである。

- isis-1.1.1-2003101601-ss (テスト版 ISIS 対応 ISIS-SimpleScalar)

| コンパイラ / OS | gcc-2.95.3 | gcc-3.3.5 |
|---|---|---|
| Fedora Core 3 (Linux) | ○ | ○ |
| Plamo 4 (Linux) | ○ | ○ |

- isis-1.1.1-ss (開発版 ISIS 対応 ISIS-SimpleScalar)

| コンパイラ / OS | gcc-2.95.3 | gcc-3.3.5 |
|---|---|---|
| Fedora Core 3 (Linux) | ○ | × |
| Plamo 4 (Linux) | ○ | × |

以後、OS を Fedora Core 3 として説明を行う。尚、他のプラットフォームでの利用方法などは ISIS-SimpleScalar のホームページ (http://www.am.ics.keio.ac.jp/proj/snail/isis-ss/index.html) に記述してある。今後、ISIS-SimpleScalar に更新があった場合にも、このページ上で報告する予定である。

# 第 2 章

# ISIS-SimpleScalar

現在、次の ISIS-SimpleScalar が利用可能である。

- テスト版 ISIS 対応の ISIS-SimpleScalar – isis-1.1.1-2003101601-ss
- 開発版 ISIS 対応の ISIS-SimpleScalar – isis-1.1.1-ss

isis-1.1.1-2003101601-ss での説明を行う。

## 2.1 ダウンロード

ISIS のホームページ (http://www.am.ics.keio.ac.jp/isis/index-ja.html) と、ISIS-SimpleScalar のホームページ (http://www.am.ics.keio.ac.jp/proj/snail/isis-ss/index.html) から、次をダウンロードする。

- isis-1.1.1-2003101601.tar.gz
- isis-1.1.1-2003101601-ss.tar.gz

## 2.2 解凍

以下の順序で解凍することによって、ISIS に ISIS-SimpleScalar のライブラリが投入され、Makefile なども ISIS-SimpleScalar 対応のものに更新される。

```
% tar zvxf isis-1.1.1-2003101601.tar.gz
% tar zvxf isis-1.1.1-2003101601-ss.tar.gz
```

解凍すると、isis-1.1.1/lib/simoutorder/に ISIS-SimpleScalar に関連する次のファイルが入っている。

### addr_order_buffer{.h,.cc}

addr_order_buffer クラスが定義してある。Load 命令および Store 命令のメモリアクセスアドレスを保持し、メモリアクセスの順序管理を行う。

### address.h

ISIS-SimpleScalar が扱う共有メモリ領域のアドレス空間が定義してある。

**arg{.h,.cc}**

arg クラスが定義してある。コマンドラインオプションとして指定されたプロセッサ数、プロセッサが外部システムと通信するためのポート数を読み取る。

**bpred{.h,.cc}**

SimpleScalar が提供する bpred.c をクラス化した bpred_simple クラスが定義してある。branch predictor を表し、branch predictor の初期化、参照、更新を行う。

**cache{.h,.cc}**

SimpleScalar が提供する cache.c をクラス化した cache_simple クラスが定義してある。ISIS-SimpleScalar においては 0x00000000〜0x7fffffff の領域の命令やローカルデータを格納するローカルキャッシュを表す。命令やデータは実際には格納しておらず、キャッシュアクセスが発生したときには、キャッシュヒット/ミスが判定され、キャッシュアクセス時間が返される。命令やデータはメモリから読み出される。

**ecoff.h**

SimpleScalar ECOFF 形式のバイナリを生成するための定義がなされている。

**endian_s{.h,.cc}**

ホストマシンとターゲットマシン上のエンディアンを決めるための関数が定義してある。

**exec_buffer{.h,.cc}**

execution_buffer クラスが定義されている。実行中の命令を管理するクラスである。命令は実行される時刻 (Functional Unit での計算が終了する時刻、またはローカルメモリへのアクセスが終了する時刻) で昇順に並べられ、同じ時刻に実行される命令は命令 ID で昇順に並べられる。

**host.h**

ホストマシンに依存する関数の定義変更などがなされている。

**instqueue{.h,.cc}**

instqueue クラスが定義してある。各命令に対するあらゆる情報を保持し、命令は in-order で並べられる。Register Update Unit および reservation station の役割を担う。

### isim_bus_packet{.h,.cc}

isim_bus_packet クラスが定義してある。ISIS-SimpleScalarが提供するプロセッサ(isim_processor)とプロセッサ外部のシステムとの間でやり取りされるパケットを表す。

### isim_bus_port{.h,.cc}

isim_bus_port クラスが定義してある。isim_processor とプロセッサ外部にあるユニットとの接続に使用されるポートを表すクラスである。入力できるパケットは isim_bus_packet クラスで定義してあるもののみである。

### isim_processor{.h,.cc}、instruction.cc、inst_simple.cc

isim_processor クラスが定義してある。
isim_processor.cc には、パイプライン内の各 Stage の動作が記述してある。

- void ruu_fetch(void) – Fetch Stage
- void ruu_dispatch(void) – Dispatch Stage
- void ruu_issue(void) – Schedule Stage
- void ruu_writeback(void) – Exec Stage と Writeback Stage
- void ruu_commit(void) – Commit Stage

instruction.cc には、isim_processor が実行する各命令の処理が定義されている。

- void ＊_impl(void) – 値の読み出しや計算結果の生成、読み出した値を確保する部分
- void ＊_addr_impl(void) – メモリアクセスアドレスを計算する部分
- void ＊_set_req_impl(void) – 共有メモリに読み出し要求を発行する部分
- void ＊_reg_impl(void) – レジスタを更新する部分

inst_simple.cc には、レジスタの更新やプロセッサ外部への要求発行など、複数の命令で行われる処理をメンバ関数としてまとめて定義し、各命令が呼び出せるようにした。

### loader_s{.h,.cc}

SimpleScalar が提供する loader.c をクラス化した loader_simple クラスが定義してある。シミュレータが実行するバイナリを読み込み、0x00400000〜0x0fffffff に格納する。

### machine{.h,.cc}

Functional Unit、immediate field、メモリ領域など、プロセッサが必要とするさまざまな定義がなされている。

**machine.def**

isim_processor で実行される各命令の識別子、オペコード、使用する Functional Unit 名、使用するレジスタ番号などの情報が定義してある。

**memory_s{.h,.cc}**

SimpleScalar が提供する memory.c をクラス化した memory_simple クラスが定義してある。ISIS-SimpleScalar においては、命令やローカルデータを格納するローカルメモリを表す。メモリ領域は 0x00000000〜0x7fffffff である。

**misc{.h,.cc}**

メッセージ出力など、多数の便利なサポート関数などを定義している。

**only_syscall.h**

システムコールに関する定義がなされている。

**options{.h,.cc}**

SimpleScalar が提供する options.c をクラス化した option_simple クラスが定義してある。キャッシュ、branch predictor など、システムに関するさまざまなコマンドラインオプションを解釈する。

**order_buffer{.h,.cc}**

addr_order_buffer クラスが定義してある。各命令の命令 ID と使用するレジスタ番号を保持し、レジスタ間の依存関係によって実行の順序を管理する。

**regs_head.h**

レジスタが構造体として定義してある。

**resource{.h,.cc}**

resource_simple クラスが定義してある。Functional Unit を生成する。

**stats{.h,.cc}**

SimpleScalar が提供する stats.c をクラス化した stats_simple クラスが定義してある。シミュレーション中にプロセッサ毎に取られる、実行クロック数やローカルキャッシュのヒット率などの統計データの指定、初期化、出力などを行う。

### version.h

SimpleScalar のバージョンに関する定義がなされている。

### writeback_buffer{.h,.cc}

writeback_buffer クラスが定義してある。Writeback イベントを終了した命令と、各命令がレジスタを更新する際の値 (計算結果やメモリから読み出した値) を保持し、reorder buffer の役割を担う。命令は命令 ID で昇順に並べられる。

## 2.3 インストール

インストールには ANSI/ISO C++コンパイラ (gcc-2.95.3、gcc-3.3.5 で動作確認済) が必要である。/usr/local/sim/isis-ss-inst にインストールするとする。

```
% cd isis-1.1.1
% ./configure --prefix=/usr/local/sim/isis-ss-inst \
 --disable-sample --disable-shared
% make
% make install
```

# 第 3 章

# SimpleScalarのインストール

SimpleScalarが提供するプロセッサシミュレータ上で実行できる実行バイナリは、ISIS-SimpleScalarを用いて実装したシミュレータ上でも実行することができる。その実行バイナリを生成するために必要なツールのインストール方法を以下に示す。

## 3.1 ダウンロード

SimpleScalar のホームページ (http://www.simplescalar.com/) から、以下をダウンロードする。

- simpleutils-990811.tar.gz
- gcc-2.7.2.3.tar.gz
- simpletools-2v0.tar.gz

## 3.2 binutils のインストール

インストールには ANSI/ISO C++コンパイラ (gcc-3.3.5 でインストール確認済) が必要である。/usr/local/sim/simplescalar-inst にインストールするとする。

```
% tar zvxf simpleutils-990811.tar.gz
% cd simpleutils-990811
% ./configure --host=i386-intel-linux --build=i386-intel-linux \
 --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld \
 --prefix=/usr/local/sim/simplescalar-inst
% make
% make install
```

## 3.3 gcc のインストール

インストールには ANSI/ISO C++コンパイラ (gcc-3.3.5 でインストール確認済) が必要である。/usr/local/sim/simplescalar-inst にインストールするとする。

```
% tar zvxf gcc-2.7.2.3.tar.gz
% cd gcc-2.7.2.3
```

libgcc2.c の 98 行目に以下を追加

- #define BITS_PER_UNIT 8

```
% ./configure --host=i386-intel-linux --build=i386-intel-linux \
 --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld \
 --prefix=/usr/local/sim/simplescalar-inst
% make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc" \
 LIBGCC2_INCLUDES="-I/usr/include"
```

エラーが出たら、insn-output.c の 675、750、823 行目の最後に”\”を追加

- return "FIXME\n\

```
% make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc" \
 LIBGCC2_INCLUDES="-I/usr/include"
% make install LANGUAGES="c c++" CFLAGS="-O3" CC="gcc" \
 LIBGCC2_INCLUDES="-I/usr/include"
```

## 3.4 crt0.o と libc.a の投入

```
% tar vxfz simpletools-2v0.tar.gz
% cp sslittle-na-sstrix/lib/crt0.o \
 /usr/local/sim/simplescalar-inst/sslittle-na-sstrix/lib/
% cp sslittle-na-sstrix/lib/libc.a \
 /usr/local/sim/simplescalar-inst/sslittle-na-sstrix/lib/
```

# 第 4 章

# サンプルシミュレータ

ISIS-SimpleScalar で実装されたサンプルシミュレータを実行する方法を示す。ここで紹介するサンプルシミュレータは、図 4.1 のような構成のシステムのシミュレータである。複数のプロセッシングユニット (PU) と 1 つの共有メモリ (shared memory) が単純な共有バス (shared bus) で接続されている。

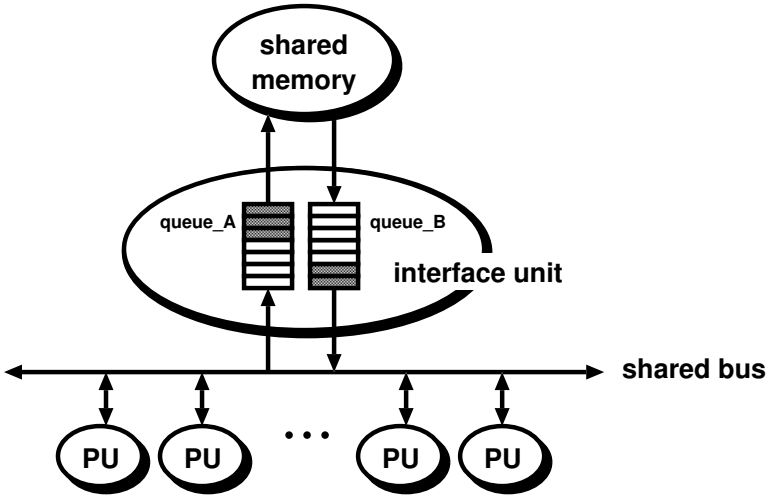


図 4.1: サンプルシミュレータ

また、各 PU は図 4.2 のような構成となっているが、各パラメータはコマンドラインオプションを用いて簡単に変更できる。

## 4.1 ダウンロード

ISIS-SimpleScalar のホームページ (http://www.am.ics.keio.ac.jp/proj/snail/isis-ss/index.html) から、以下をダウンロードする。

- simple_bus_system.tar.gz

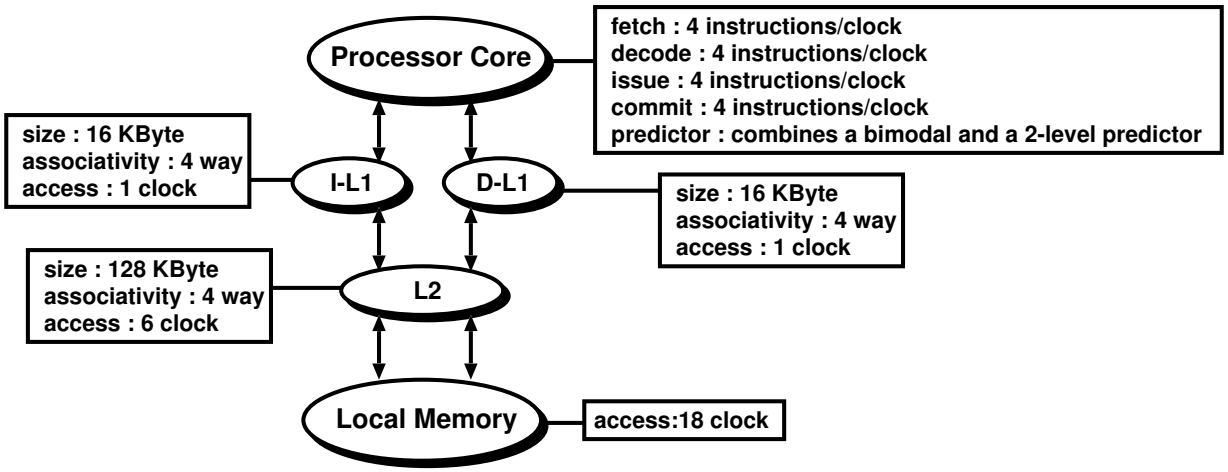


図 4.2: 各 PU の構成

## 4.2 解凍

```
% tar zvxf simple_bus_system.tar.gz
% cd simple_bus_system
```

解凍すると、以下のファイルが入っている。

- Makefile
- main.cc
- shared_memory_access_queue{.h,.cc}
- shared_bus{.h,.cc}
- shared_memory{.h,.cc}
- isim_system{.h,.cc}

### shared_memory_access_queue{.h,.cc}

アドレス、データ、データサイズ、命令 ID、プロセッサ ID などの要素をリスト管理する。
メンバ関数として、

- `void insertlist(int inst_id, md_addr_t addr, word_t dw1, word_t dw2, half_t dh, byte_t db, int data_size, int rw, int puid, int time, bool flag)`
  - キューの末尾に要素を追加する
- `void deletelist(int inst_id, md_addr_t addr, int puid)`

  - 該当するキュー内の要素を削除する

- `void get_data(int* id, md_addr_t* addr, word_t* dw1, word_t* dw2, half_t* dh, byte_t* db, int* data_size, int* rw, int* puid, int now, bool* flag)`

  - キューの先頭要素を渡す

が定義してある。

### shared_bus{.h,.cc}

図 4.1 の shared bus を表す。interface unit と PU に接続され、要求やデータを送信したい interface unit や PU に対してアービトレーションを行い、それらに順次使用権を与える。要求やデータの管理には shared_memory_access_queue{.h,.cc} で定義されたキューを用いる。

メンバ関数として、

- `void get_packet(void)`

  - PU および interface unit からの要求を受け取り、キューの末尾に順次追加する

- `void send_packet(void)`

  - キューの先頭から要素を取り出し、その要求に応じて PU または interface unit に転送する

が定義してある。

### shared_memory{.h,.cc}

図 4.1 の interface unit、shared memory を表す。shared bus を介して転送されてきた要求に対して、shared memory からの読み出しや shared memory への書き込みを行う。また、読み出し要求に対しては、PU への読み出しデータ送信要求を shared bus に発行する。要求やデータの管理には shared_memory_access_queue{.h,.cc} で定義されたキューを用いる。

メンバ関数として、

- `void clock(void)`

  - shared bus から転送されてきた要求を受け取り、キュー A の末尾に順次追加する
  - キュー A の先頭から要求を取り出し、その要求に応じたメモリアクセスを行う
  - 読み出しデータ送信要求をキュー B の末尾に順次追加する
  - キュー B の先頭から要求を取り出し、shared bus に発行する

が定義してある。

**isim_system{.h,.cc}**

シミュレータ全体のシステムを 1 つにまとめ、ユニットの接続やクロックの分配を行う。
メンバ関数として、

- `void initialize(unsigned int, int)`
  - 引数として与えられた数分の PU を用意する
  - 引数として与えられた数分のポートを用意する
  - PU と shared bus、interface unit と shared bus を接続する
  - shared memory の領域を指定する
  - 各 PU に ID を割り当てる
- `void sim_main(void)`
  - シミュレータにクロックを入れる

等が定義してある。

## 4.3 シミュレータの make

Makefile を修正する。CC には、ISIS-SimpleScalar をインストールした際と同じバージョンの ANSI/ISO C++コンパイラを指定し、isisdir には、ISIS-SimpleScalar がインストールしてあるディレクトリを指定する。
<例>

- `CC = /usr/local/gcc/bin/g++`
- `isisdir = /usr/local/sim/isis-ss-inst`

```
% make
```

“simulator”という実行ファイルができれば、成功である。

# 第 5 章

# シミュレータ上でアプリケーションの実行

シミュレータ上でのアプリケーションの実行について説明する。/usr/lib に libstdc++.so.5 がない場合には、ホストマシンの gcc/lib/libstdc++.so.5 のリンクを貼る必要がある。

<例>

```
% ln -s /usr/local/gcc-3.3.5/lib/libstdc++.so.5 /usr/lib
```

## 5.1 用意されている実行バイナリの実行

simple_bus_system/tests 内に、以下の実行バイナリが入っている。

- SimpleScalar が提供するテストプログラム
  - tests/bin.little/test-fmath
  - tests/bin.little/test-llong
  - tests/bin.little/test-lswlr.out
  - tests/bin.little/test-math
  - tests/bin.little/test-print
- マルチプロセッサシステムシンプルテストプログラム
  - tests/test/kuku.out (九九の並列計算)

実行は次のようにする。

```
% ./simulator (simulator オプション) 実行バイナリ (実行バイナリオプション)
```

実行例は次の通りである。

- SimpleScalar が提供するテストプログラム (シングルプロセッサ用)

```
% ./simulator tests/bin.little/test-lswlr.out
% ./simulator tests/bin.little/test-fmath
```

- マルチプロセッサシステムシミュレーション用
  - コマンドの末尾の-p2 や-p16 というオプションは、プロセッサ数を指定する実行バイナリへのオプションである

```
% ./simulator -p 2 tests/test/test.out -p2
% ./simulator -p 16 tests/test/test.out -p16
```

## 5.2 実行バイナリの作成

独自に作成したプログラムの実行バイナリを生成し、それをシミュレータ上で実行する方法を説明する。

### プログラムの作成

<例> hello_world.c

```
#include <stdio.h>

int main (void) {
    printf("Hello World!\n");
    return 0;
}
```

### コンパイル

SimpleScalar が/usr/local/sim/simplescalar-inst にインストールされているとする。

```
% /usr/local/sim/simplescalar-inst/bin/sslittle-na-sstrix-gcc \
 hello_world.c -I/usr/include -o hello_world.out
```

### 実行

```
% ./simlator hello_world.out
```

**実行結果**

```
-----------------以上省略 -----------------
sim: ** starting performance simulation **
Hello World!

sim: ** simulation statistics **
-----------------以下省略 -----------------
```

## 5.3 マルチプロセッサシステムシミュレーション用並列計算プログラムの作成

マルチプロセッサシステム上で実行する並列計算プログラムの作成方法について説明する。simple_bus_system/tests/test 内に、並列計算プログラムの作成を助ける次のプログラムが入っている。これらのプログラムのコンパイルには、同じディレクトリ内に ISIS-SimpleScalar の address.h へのリンクが貼ってある必要がある。

```
ln -s /usr/local/sim/isis-ss/include/isis/address.h address.h
```

### get_puid.c

プロセッサ ID を読み出すためのアドレスへアクセスするプログラムである。main プログラムに、

```
unsigned int puid = get_puid();
if ( puid == 0 ) {
  処理 A;
}
```

と記述すれば、プロセッサ ID が 0 のプロセッサのみが処理 A を行う。

### get_punum.c

システム上のプロセッサ数を読み出すためのアドレスへアクセスするプログラムである。main プログラムに、

```
unsigned int punum = get_punum();
```

と記述すれば、punum にプロセッサ数が代入される。

**barrier.c**

すべてのプロセッサで同期をとるためのプログラムである。main プログラムに、

```
barrier();
```

と記述すれば、すべてのプロセッサがこの barrier() に達するまで、他のプロセッサは barrier() から抜けない。つまり、

```
処理 A;
barrier();
処理 B;
```

と記述すれば、すべてのプロセッサが処理 A を終了してから、各プロセッサは処理 B を始める。

**shared_addr_allocate.c**

address.h で定義された SHARED_MEM_TOP〜SHARED_MEM_BOTTOM の範囲からデータ領域を確保するプログラムである。main プログラムに、

```
unsigned int *mem;
mem = (unsigned int *)shared_addr_allocate(sizeof(unsigned int));
```

と記述すれば、すべてのプロセッサで共有される unsigned int 型の大きさの領域を SHARED_MEM_TOP〜SHARED_MEM_BOTTOM の範囲から確保する。

**shared_addr_for_sync_allocate.c**

address.h で定義された SHARED_MEM_FOR_SYNC_TOP〜SHARED_MEM_FOR_SYNC_BOTTOM の範囲からデータ領域を確保するプログラムである。main プログラムに、

```
unsigned int *mem;
mem = (unsigned int *)shared_addr_for_sync_allocate(sizeof(unsigned int));
```

と記述すれば、すべてのプロセッサで共有される unsigned int 型の大きさの領域を SHARED_MEM_FOR_SYNC_TOP〜SHARED_MEM_FOR_SYNC_BOTTOM の範囲から確保する。

**fad_addr_allocate.c**

同期機構を実現するための Fetch and Dec を行うためのプログラムである。main プログラムに、

```
unsigned *mem;
mem = (unsigned int*)fad_addr_allocate(sizeof(unsigned int));
```

と記述すれば、すべてのプロセッサで共有される unsigned int 型の大きさの領域を address.h で定義された FAD_MEM_TOP〜FAD_MEM_BOTTOM の範囲から確保する。その後、

```
if ( get_puid() == 0 )
    *mem = get_punum();

barrier();

unsigned int *a;
a = (unsigned int *)shared_addr_allocate(sizeof(unsigned int));
*a = 0;

if ( *mem > 1 )
    while(!(*a)) {}
else
    *a = 1;
```

と記述し、シミュレータの FAD_MEM_TOP〜FAD_MEM_BOTTOM へのメモリアクセスに対する処理を

- メモリの値を読み出し、その値から 1 引いた数をメモリに書き込む
- プロセッサには、1 引く前の値を返す

とすれば、すべてのプロセッサ間で同期を取ることができる。これを応用したものが上記した barrier.c である。

### tas_addr_allocate.c

同期機構を実現するための Test and Set を行うためのプログラムである。main プログラムに、

```
unsigned *idlock;
idlock = (unsigned*)tas_addr_allocate(sizeof(unsigned));
```

と記述すれば、すべてのプロセッサで共有される unsigned 型の大きさの領域を address.h で定義された TAS_MEM_TOP〜TAS_MEM_BOTTOM の範囲から確保する。その後、

```
unsigned int *id;
id = (unsigned*)shared_addr_allocate(sizeof(unsigned));
*id = 0;
unsigned int Mynum;
*idlock = 1;

barrier();

while (*idlock) {}

Mynum = id;
id++;

*(Global->idlock) = 1;
```

と記述し、シミュレータの TAS_MEM_TOP〜TAS_MEM_BOTTOM へのメモリアクセスに対する処理を

- メモリの値が 1 ならばメモリに 0 を書き込みプロセッサに 0 を返す
- メモリの値が 0 ならプロセッサに 1 を返す

とすれば、各プロセッサの Mynum がすべて異なる値となる。

### system_clock.c

このプログラムは、統計データを取得するためのものであり並列計算プログラムの作成を助けるためのものではないが、説明する。ISIS-SimpleScalar では、0xd0000000 番地へのメモリアクセスがあってから 0xd0000100 番地へのメモリアクセスがあるまでカウントされる特定部分統計データの取得が可能である。0xd0000000 番地へアクセスするコードと 0xd0000100 番地へアクセスするコードをシミュレータが実行するプログラムに埋め込むことによって、ユーザは任意の部分の統計データを取ることができる。

main プログラムに、

```
unsigned int start = calc_start();
```

と記述すれば、0xd0000000 番地へのアクセスが発行される。つまり、シミュレータ側で特定部分統計データの取得が始まる。

```
unsigned int end = calc_end();
```

と記述すれば、0xd0000100 番地へのアクセスが発行される。つまり、特定部分統計データの取得が終了する。取得された統計データは、実行終了時に全時間統計データとは別に出力される。

### Makefile

SimpleScalar が/usr/local/sim/simplescalar-inst にインストールされているとする。

```
CC = /usr/local/sim/simplescalar-inst/bin/sslittle-na-sstrix-gcc
```

PROGRAM でコンパイルソースを指定する。

```
PROGRAM = main.c \
          get_puid.c \
          get_punum.c \
          barrier.c \
          shared_addr_allocate.c \
          shared_addr_for_sync_allocate.c \
          fad_addr_allocate.c \
          tas_addr_allocate.c \
          system_clock.c
```

# 第 6 章

# 新しいシミュレータの実装

ISIS-SimpleScalarを用いたシミュレータの構築にはISISが提供するライブラリの利用が可能であるため、さまざまなシステムを比較的容易に構築できるというISISの利点がそのまま享受できる。ISISについては、isis-1.1.1 online manual (http://www.am.ics.keio.ac.jp/isis/manual/index-ja.html) を参考にして頂きたい。

新しいシミュレータの実装には、サンプルシミュレータのコードが参考になる。shared_bus{.h,.cc} や shared_memory{.h,.cc} を参考にプロセッサ外部のモジュールを記述し、`isim_system` クラス (isim_system{.h,.cc}) のメンバ関数 `void initialize(unsigned int, int)` でモジュールのポート同士を接続する記述を行い、`void sim_main(void)` で各モジュールにクロックを入れる記述を行えば良い。

サンプルシミュレータのコードを示し、具体的に説明する。

`void initialize(unsigned int, int)`

システムを構成するモジュールを表すクラスのインスタンス生成や、それらの接続を行う。

```
 1:   void
 2:   isim_system::initialize(unsigned int punum, int portnum)
 3:   {
 4:       soo.resize(punum);
 5:       shared_mem_isis.set_top(0x80000000);
 6:       shared_mem_isis.resize(0x7fffffff);
 7:       shmem.connect_mem(shared_mem_isis);
 8:       shbus.set_each_pu_port_num(portnum);
 9:       shbus.set_num_of_pu_port(portnum*punum);
10:       shmem.ref_shbus_port().connect(shbus.ref_mm_port());
11:       for ( unsigned int i = 0; i < punum; i++ ) {
12:           soo[i].set_puid(i);
13:           soo[i].set_punum(punum);
14:           soo[i].set_punum(portnum);
15:           for ( int p = 0; p < portnum; p++ ) {
16:               shbus.ref_to_pu_port(i*portnum+p).connect(soo[i].ref_to_pu_
     port(p));
```

```
17:                 shbus.ref_from_pu_port(i*portnum+p).connect(soo[i].ref_from
      _pu_port(p));
18:         }
19:     }
```

以上のコードにより、図 6.1 のようなシステムが構築される。

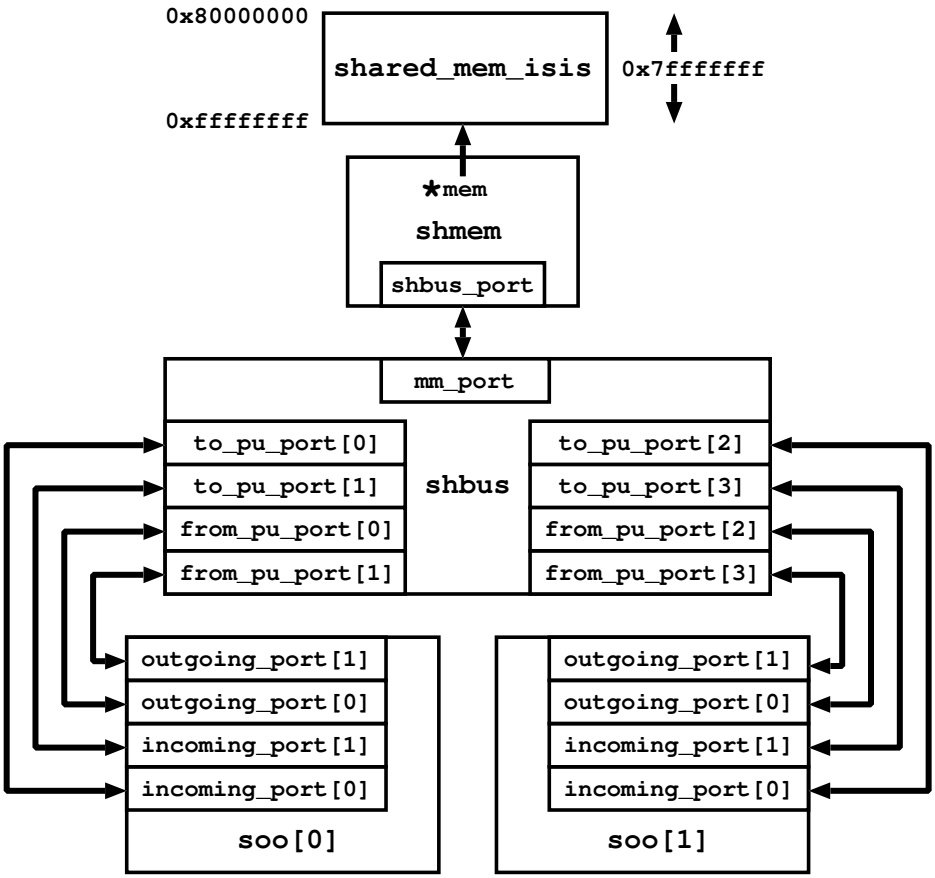


図 6.1: 上記コードによって構築されるシステム

サンプルコードに沿って、システムが構成される流れを示す。

2: 引数としてプロセッサ数とポート数を受け取る

4: isim_processor クラスのインスタンス soo の数を指定されたプロセッサ数分にする

- 図 6.1 はプロセッサ数が 2 の場合

5: mapped_memory クラス (ISIS が提供するメモリ) のインスタンス shared_mem_isis の先頭アドレスを 0x80000000 に設定する

6: shared_mem_isis のサイズを 0x7fffffff に設定する (終端アドレス = 0xffffffff)

7: shmem がアクセスするメモリを shared_mem_isis に設定する

8: shbus に soo のポート数を通知する

9: shbus の soo 側のポートを生成
   - to_pu_port*soo 毎のポート数*soo 数
   - from_pu_port*soo 毎のポート数*soo 数
10: shmem の shbus_port と shbus の mm_port を接続する
12: soo にプロセッサ ID を割り振る
13: soo にシステム上の soo 数を通知する
14: soo が外部と通信するためのポート数を指定された数にする
16: shbus の to_pu_port と soo の incoming_port を接続する
17: shbus の from_pu_port と soo の outgoing_port を接続する

void sim_main(void)

各モジュールにクロックを入れる。パケットの送信や受信を行う記述をしている場合には、クロックが入力されることで、void initialize(unsigned int, int) において接続したモジュール間でパケットがやり取りされる。

```
     void
     isim_system::sim_main(void)
     {

                ..............
                ..............

 1:      for (;;) {
 2:          for ( unsigned int i = 0; i < soo.size(); i++ )
 3:              soo[i].clock_out();
 4:
 5:          shbus.get_packet();
 6:          shbus.send_packet();
 7:          shmem.clock();
 8:
 9:          for ( unsigned int i = 0; i < soo.size(); i++ ) {
10:              soo[i].clock_in();

                ..............
                ..............

            }
        }
     }
```

サンプルコードに沿って、システムが動作する流れを示す。

1: シミュレータはプログラムの実行が終了するまで、1 ループ 1 クロックとして処理を進める

3: soo が clock_out() の処理 (ruu_commit()、ruu_writeback()、ruu_issue()) を行う

- 共有メモリへの要求がある場合には、soo の出力ポート (outgoing_port) にパケットが発行される

5: soo および shmem からパケットが発行されている場合には、shbus がそれらを受信する

6: shbus が get_packet() で受信したパケットを送信先に転送する

7: shmem が clock() の処理を行う

- shbus からパケットが送信されてきている場合には、そのパケットを受信する
- メモリアクセスを行う
- メモリアクセスが終了したら、読み出し要求に対するデータが shmem の shbus_port に発行される

10: soo が clock_in() の処理 (ruu_dispatch()、ruu_fetch()、get_data()) を行う

- shmem からの読み出しデータが転送されてきている場合には、それを受信する

以上のように、作成したプロセッサ外部のモジュールを void initialize(unsigned int, int) で繋げ、void sim_main(void) で各モジュールにクロックを入力すれば、シミュレータが動作する。

## 6.1 プロセッサから発行される要求に対する処理

ISIS-SimpleScalar が提供するプロセッサ isim_processor は、実行するプログラム中に共有メモリ領域へのアクセスがあると、共有メモリへの要求を外部出力ポートに発行する。isim_processor 外部のシステムは、isim_processor からの要求を受け取り、要求に応じた処理を行わなければならない。

### 6.1.1 isim_processor が扱うパケット

isim_processor から発行されるパケットには、以下の情報が付加されている。

```
md_addr_t addr;          //メモリアクセスアドレス
int data_size;           //メモリアクセスする際の byte 幅
int inst_id;             //命令 ID
int rw;                  //Read or Write or Reply
int puid;                //送信元プロセッサの ID
/* データ */
word_t data, data2;      //unsigned int 型
half_t data_half;        //unsigned short 型
byte_t data_byte;        //unsigned char 型
```

### 6.1.2 isim_processor から発行されるパケットの取得

isim_processor 外部のシステムは、isim_processor の外部出力ポート (outgoing_port) に発行されたパケットに記載されている情報を元に、それぞれに応じた処理を行う必要がある。パケットに記載された情報は、次のようにすれば取得できる。isim_processor の外部出力ポート (outgoing_port) に port_A という名の外部システムのポートが接続されているとする。

```
if ( port_A.have_packet() && port_A.inst_id() != -1 ) {
    int inst_id      = port_A.inst_id();
    md_addr_t addr   = port_A.addr();
    word_t data      = port_A.data();
    word_t data2     = port_A.data2();
    half_t data_half = port_A.data_half();
    byte_t data_byte = port_A.data_byte();
    int data_size    = port_A.data_size();
    int rw           = port_A.rw();
    int puid         = port_A.puid();
    port_A.reset_packet();

    /* パケットに応じた処理を以下に記述 */

                ..............
                ..............


}
```

パケットの初期状態における inst_id の値は-1 であり、isim_processor から発行されるパケットの inst_id には正の値が入るため、パケットの情報取得の条件に port_A.inst_id() != -1 を入れることによって、無駄な処理を省くことができる。また、情報を取得した後、

```
port_A.reset_packet();
```

とすることによって、パケットは初期化され、isim_processor は新しいパケットを外部出力ポート (outgoing_port) に発行できるようになる。

### 6.1.3 isim_processor へのパケットの送信

メモリからの読み出しデータを isim_processor に返したい場合には、次のようにすればプロセッサに送信できる。isim_processor の入力ポート (incoming_port) に port_B という名の外部システムのポートが接続されているとする。

```
if ( !port_B.have_packet()
     || ( port_B.have_packet()
          && port_B.inst_id() == -1 ) ) {
    port_B.set_inst_id(inst_id);
    port_B.set_addr(addr);
    port_B.set_data(data);
    port_B.set_data2(data2);
    port_B.set_data_half(data_half);
    port_B.set_data_byte(data_byte);
    port_B.set_data_size(data_size);
    port_B.set_reply();
    port_B.set_puid(puid);
}
```

どの要求に対する読み出しデータなのかを isim_processor に知らせるため、puid、inst_id には受信したときと同じ値を代入する必要がある。また、読み出しデータの送信要求であることを set_reply() によって通知する。if 文の条件に port_B.inst_id() == -1 を付加することによって、情報取得されていないパケットへの上書きを避けることができる。

### 6.1.4 外部システムがすべき具体的な処理内容

isim_processor からの要求に対して外部システムが行うべき処理について、パケットに記された情報別に説明する。

- 1byte データの読み出し要求 (rw==0、data_size==1) が発行されたとき
  - addr 番地からデータを読み出す
  - 読み出したデータを data_byte に代入して isim_processor に返す
- 2byte データの読み出し要求 (rw==0、data_size==2) が発行されたとき
  - addr 番地からデータを読み出す
  - 読み出したデータを data_half に代入して isim_processor に返す
- 4byte データの読み出し要求 (rw==0、data_size==4) が発行されたとき
  - Fetch and Dec を行う領域へのアクセス (addr >= FAD_MEM_TOP && addr <= FAD_MEM_BOTTOM)
    - addr 番地からデータを読み出す
    - 読み出したデータから 1 引いた値を addr 番地に書き込む
    - 読み出したデータを data に代入して isim_processor に返す
  - Test and Set を行う領域へのアクセス (addr >= TAS_MEM_TOP && addr <= TAS_MEM_BOTTOM)
    - addr 番地からデータを読み出す
    - 読み出したデータが 1 ならば addr 番地に 0 を書き込み、data に 0 を代入して isim_processor に返す

        * 読み出したデータが 0 ならば data に 1 を代入して isim_processor に返す
    - 通常のメモリアクセスを行う領域へのアクセス (上記条件以外の addr)
        * addr 番地からデータを読み出す
        * 読み出したデータを data に代入して isim_processor に返す

- 8byte データの読み出し要求 (rw==0、data_size==8) が発行されたとき
    - addr 番地と addr+4 番地からデータを読み出す
    - addr 番地から読み出したデータを data に、addr+4 番地から読み出したデータを data2 に代入して isim_processor に返す
- 1byte データの書き込み要求 (rw==1、data_size==1) が発行されたとき
    - addr 番地に data_byte の値を書き込む
- 2byte データの書き込み要求 (rw==1、data_size==2) が発行されたとき
    - addr 番地に data_half の値を書き込む
- 4byte データの書き込み要求 (rw==1、data_size==4) が発行されたとき
    - addr 番地に data の値を書き込む
- 8byte データの書き込み要求 (rw==1、data_size==8) が発行されたとき
    - addr 番地に data の値を、addr+4 番地に data2 の値を書き込む

以上の処理さえ正確に行えばよい。そのため、プロセッサ外部のネットワーク部やメモリシステムなどの記述は、ISIS のライブラリを使用しなくてもよい。

# 第 7 章

# コマンドラインオプション

ISIS-SimpleScalar が提供するコマンドラインオプションをまとめる。

`-config [file]`
: `[file]` に記述されたコマンドラインオプションを読み込む

`-dumpconfig [file]`
: 指定したコマンドラインオプションを `[file]` に出力する

`-p [number]`
: システムに搭載するプロセッサ数を指定する

`-redir:prog [file]`
: 実行結果を `[file]` に出力する

`-nice`
: シミュレータプロセスの優先度を指定する

`-max:inst [number]`
: シミュレータが `[number]` 個の命令を実行した時点で、実行を終了する

`-fetch:ifqsize [number]`
: IFQ(fetch された命令を保持する) のサイズを指定する

`-fetch:speed`
: 命令 fetch 幅を指定する

`-bpred {nottaken|taken|bimod|2lev|comb}`
: branch predictor を指定する

  `nottaken` – 常に分岐しないと予測

  `taken` – 常に分岐すると予測

  `bimod` – 2 ビットカウンタによる予測

  `2lev` – 2 レベル予測機構による予測

  `comb` – bimodal predictor と 2-level predictor を結合した branch predictor

`-bpred:bimod [number]`
テーブルのサイズを指定する (※ `-bpred bimod` を指定した時のみ有効)

`-bpred:2lev [l1size] [l2size] [hist_size] [xor]`
2-level predictor の設定を行う (※ `-bpred 2lev` を指定した時のみ有効)

`l1size` – L1 テーブル (分岐履歴テーブル) のサイズ

`l2size` – L2 テーブル (2 ビットカウンタ) のサイズ

`hist_list` – 分岐履歴の幅

`xor` – L2 テーブルにおいて、履歴とアドレスの XOR を行うかどうか

`-bpred:comb [number]`
bimodal predictor と 2-level predictor で共有するテーブルのサイズを指定する

`-bpred:btb [num_sets] [assoc]`
Branch Target Buffer のの設定を行う

`num_sets` – セット数

`assoc` – 連想度

`-decode:width [number]`
命令 decode 幅を指定する

`-issue:width [number]`
命令 issue 幅を指定する

`-commit:width [number]`
命令 commit 幅を指定する

`-ruu:size [number]`
RUU(命令が fetch されてから commit されるまで命令を保持する) のサイズを指定する

`-lsq:size [number]`
LSQ(Load 命令/Store 命令を保持する) のサイズを指定する

`-cache:dl1 {<config|none>}`
L1 ローカルデータキャッシュを設定する

`<config> = <name>:<nsets>:<bsize>:<assoc>:<repl>`

`<name>` – キャッシュの名前 (識別子)

`<nsets>` – セット数

`<bsize>` – 大きさ (KByte)

`<assoc>` – 連想度

`<repl>` – 置換アルゴリズム (l:LRU,f:FIFO,r:random)

`-cache:dl1lat [number]`
L1 ローカルデータキャッシュにヒットしたときのレイテンシを指定する

-cache:dl2 {<config|none>}
    L2 ローカルデータキャッシュを設定する

-cache:dl2lat [number]
    L2 ローカルデータキャッシュにヒットしたときのレイテンシを指定する

-cache:il1 {<config|none>}
    L1 ローカル命令キャッシュを設定する

-cache:il1lat [number]
    L1 ローカル命令キャッシュにヒットしたときのレイテンシを指定する

-cache:il2 {<config|none>}
    L2 ローカル命令キャッシュを設定する

-cache:il2lat [number]
    L2 ローカル命令キャッシュにヒットしたときのレイテンシを指定する

-cache:icompress
    64 ビットの命令アドレスを 32 ビットに置き替える

-mem:lat [fist] [next]
    ローカルメモリへのアクセスレイテンシを指定する

-mem:width [number]
    メモリのバス幅 (bytes)

-tlb:itlb
    命令 TLB(アドレス変換バッファ) を設定する

-tlb:dtlb
    データ TLB を設定する

-tlb:lat [number]
    TLB ミス時のレイテンシを指定する

-res:ialu [number]
    整数 ALU の数を指定する

-res:imult [number]
    整数乗除算器の数を指定する

-res:memport [number]
    ローカルメモリアクセスポートの数を指定する

-res:fpalu [number]
    浮動小数点 ALU の数を指定する

-res:fpmult [number]
    浮動小数点乗除算器の数を指定する

`-o [number]`
プロセッサが外部のシステムと通信するためのポート数を指定する

`-cache:flush`
特定部分統計データ取得時に限ったキャッシュに関する統計データを取得する (5.3 項 system_clock.c 参照)

- `0xd0000000` 番地へのアクセスがあったら特定部分統計データ取得用のキャッシュに切り替える
- `0xd0000100` 番地へのメモリアクセスがあったら、切り替える前のキャッシュに戻す

`-output [number]`
統計データの出力方法を指定する

0 – 統計データを出力しない
1 – 特定部分統計データのみ出力する
2 – 全時間の統計データのみ出力する
3 – 全時間の統計データと特定部分統計データの両方を出力する

# 第 8 章

# デバッグ出力

ISIS-SimpleScalar が提供するプロセッサ isim_processor のデバッグをサポートする出力として、次のものが用意してある。

## 8.1 処理直前、直後の出力

各命令に関わる処理がなされるとき、その命令のプログラムカウンタ、命令名とともに、実行直前のオペランドの内容や実行結果、メモリアクセスアドレスなどを出力する。instruction.cc 内に

```
#define DEBUG_ISIM_PROCESSOR
```

inst_simple.cc 内に

```
#define OBSL_DEBUG_ISIM_PROCESSOR
```

と記述し、インストールすれば良い。

## 8.2 パイプライン内命令の出力

各命令がパイプラインの Stage を移るときに、その命令のプログラムカウンタ、命令名、命令 ID、使用するレジスタ番号を出力する。isim_processor.cc 内に

```
#define COUT_PIPELINE
```

と記述し、インストールすれば良い。