PCI バス シミュレーション環境

永見 康一[1]

1997/03/14 (Revision: 1.3)

[1]NTT 情報通信研究所 (nagami@isl.ntt.co.jp, tel:0468-59-3599)

更新履歴

Revision: 1.2 初版

Revision: 1.3

1. 規定課題の仕様変更 (Revision 1.4 → Revision 1.5) にともない, シミュレーションログ (リスト C.1, C.2, C.3) が若干変わった.

目次

はじめに

この資料は, ASIC デザインコンテスト規定課題である,「PCI バスインターフェース」を設計する上で, 設計回路の動作を `SECONDS` でシミュレーションする際に利用できるシミュレーション環境について説明するものです. 以降では, この環境を *PCI* 環境と呼びます.

1　概要

規定課題の「PCI バスインターフェース」では, *PCI* の信号プロトコルを実装してもらうことがメインとなります. *PCI* とはバス, すなわちデータ転送路の規格ですから, 設計の段階で回路を繰り返しデバッグする際には (あるいはコンテストの応募作品の動作確認をするためには), 設計した回路で正しくデータ転送が行なえることを確認するためのシミュレーション環境が不可欠です. これを踏まえて, *PCI* 環境は次のシミュレーション要素を提供します.

マイクロ・プロセッサ DLX　計算機システムのホスト CPU となるマイクロ・プロセッサとして, 文献 [2] などで知られる DLX (デラックス) に準拠したプロセッサのシミュレーション回路を用意しました. モジュール階層は図 1のようになっています. なお, 実装されている命令セットについては, 資料 [3] を参照して下さい.

DLX は 5 段パイプラインのロード / ストア型 RISC プロセッサです. 用意した回路では, 分岐命令, ロード命令がともに遅延命令となっています. すなわち, ジャンプ命令, 分岐命令の直後に遅延スロットが 1 命令必要であり, またロード命令 (`lb`, `lbu`, `lh`, `lhu`, `lw`) のデスティネーション・レジスタは, 直後の命令で参照することはできず, 1 命令の遅延スロットが必要です.

DLX 用内部キャッシュ　DLX 用の内部キャッシュ回路として次のような仕様のキャッシュ回路を用意しました.

- 命令・データ混合キャッシュ
- 容量: 4 words / line, × 16 line (total 256 bytes)
- マップ方式: direct map
- 書き込み: write through, no write allocate

このキャッシュ回路と, DLX を組み合わせて, 図 2 のようなキャッシュ内蔵 DLX を用意しました.

計算機システム `system0`　キャッシュを持たないプロセッサ, 二基のメモリ, および I/O 装置を結合した計算機システムです. このシステムは, 回路規模が小さく, かつ DLX プログラムの実行シミュレーションにかかるクロック数も最も小さいため, DLX プログラム自体のデバッグに利用すると便利です.

計算機システム `system1`　内部キャッシュを持つプロセッサと, 二基のメモリ, および I/O 装置を結合した計算機システムです. 結合バスは *PCI* ではありませんが, *PCI* が定めるコンフィギュレーション・レジスタを二基 (プロセッサ用と, 片方のメモリ用) と, コンフィギュレーション・メカニズム 1 を扱うブリッジ回路が用意されているため, DLX プログラム内でコンフィギュレーション・メカニズム 1 に基づいて各レジスタを操作することができます.

計算機システム `system2`　内部キャッシュを持つプロセッサと, 二基のメモリ, および I/O 装置を結合した計算機システムです. プロセッサと, 片方のメモリは *PCI* バスインタフェース回路により, *PCI* バス経由で接続されています. また, `system1` で述べたブリッジ回路も, *PCI* バスインタフェース回路によって *PCI* バスに接続されており, コンフィギュレーション・メカニズム 1 に基づいて *PCI* バス上のコンフィギュレーション・サイクルのイニシエータ動作を司ります.

コンテストの応募作品の動作確認では, プロセッサと, 片方のメモリを *PCI* バスに接続する *PCI* バスインタフェース回路 (モジュール名 `PCI`, ソースファイル `PCI.sfl`, インスタンス名 `DPCI`, `UPCI`) を, 設計回路で置き換えてシミュレーションを行ないます. ブリッジ回路用の *PCI* バスインタフェース回路 (モジュール名 `PCIb`, ソースファイル `PCIb.sfl`, インスタンス名 `BPCI`) については, 設計回路がコンフィギュレーション・サイクルのイニシエータ動作をサポートしている場合は置き換えることができますが, 課題として強制はしません.

例題プログラム `fact.asm`　各計算機システムによる実行シミュレーションを行なうための DLX プログラムの例題です. 内容は, コンフィギュレーション・メカニズム 1 に基づく簡単なコンフィギュレーション・シーケンスと, 10 の階乗を再帰関数呼び出しにより求めるコードから成ります.

SECONDS 用 **DLX** アセンブラ dasm　DLX のアセンブリ言語で記述された DLX プログラムを, system0, 1, 2 上で実行シミュレーションするための SECONDS スクリプトに変換するアセンブラです. SparcStation + SunOS 4.1.4 で動作確認したバイナリを用意しました. これを用いて, 自前の DLX プログラムをシミュレーション環境で実行することもできます.

サポートされている DLX 命令セットやアセンブリ言語の構文などについては, 資料 [3] を参照して下さい.

2 各シミュレーション環境

DLX プロセッサを用いたシミュレーション環境を 3 種類 (system0, system1, system2) 用意しました. いずれの環境においても, 以下のようにしてシミュレーションを行なうことができます.

1. まず, シミュレーションのための DLX プログラムを用意します. 具体的には, DLX の命令セットを用いてアセンブリ・プログラムを記述した .asm ファイルを作成します. サンプルとして fact.asm が用意されています.

2. 次に, 用意した DLX プログラム (ここでは仮に fact.asm とします) をアセンブルし, SECONDS が読める形式の .bin ファイルを作成します. 具体的には, 次のようにします.

   ```
   % ./dasm -a fact.adr fact.asm > fact.bin
   または,
   % make fact.bin
   ```

   dasm の -a fact.adr というオプションにより, fact.asm に現れるすべてのラベルについて, それらのアドレス対応を記したファイル fact.adr が作成されます.

   fact.bin は, fact.asm プログラムの動作を seconds でシミュレーションするための, seconds の命令群で構成されたテキストファイルです.

3. 次に, SECONDS シミュレーション用に, 各回路の SFL 記述のシミュレーション・イメージをコンパイルしておきます. ここでいうコンパイルとは, SECONDS を起動し, SFL 記述を sflread で読み込み, sflsave でモジュール・クラス・ファイルを出力することを指します.

   このようにしてシミュレーション・イメージをコンパイルしておくことにより, シミュレーションをやり直すたびに, SFL の構文解析に時間を費やす必要がなくなります.

   具体的には, 次のようにします.

   ```
   % make mc
   ```

   これにより, *PCI* 環境に含まれる各 .sfl ファイルについて, 拡張子を .mc としたモジュール・クラス・ファイルが出来上がります. このモジュール・クラス・ファイルがあれば, SECONDS で

   ```
   % sflread dlx.sfl
   ```

   とする代わりに

   ```
   % sflload dlx.mc
   ```

   とすることにより, 読み込み速度がかなり高速になります.

   このようなコンパイル・作業は, 基本的には一度だけ行なえばよい作業です. しかし, .sfl ファイルや .h ファイルの内容に変更を加えた場合, その都度行なう必要があります. その際にも,

   ```
   % make mc
   ```

   とすれば, 再コンパイルの必要な SFL ファイルだけが自動的にコンパイルされます.

4. いよいよ, それぞれのシミュレーション環境を用いて, 用意した DLX プログラム (.bin ファイル) の実行シミュレーションを行ないます. 例えば system0 を用いる場合,

   ```
   % seconds
   % SECONDS> system0.sim fact.bin
   ```

とすることにより, シミュレーション・イメージの構築と, 環境のリセット (2 クロック進みます) が行なわれます. どのようなことを SECONDS に処理させているのかについては, system0(1, 2).sim の各スクリプトを参照し, 呼び出されているスクリプトをたどってみて下さい.

このあとは, 通常の SECONDS のコマンド (forward, print など) により, シミュレーションを進めていくことができます.

各環境は, rpt_add 文により, クロック毎に環境の各種状態をレポートします. このレポートはかなり大量の情報になるため, 次のようにしてレポートをログ・ファイルに保存しておくようにすればよいでしょう. また, シミュレーション中に生じたエラー情報もファイルに保存しておけば, 後で解析する際に便利です.

```
% 1seconds
1SECONDS> speak ログ・ファイル名; claim エラー・ファイル名
としてから
1SECONDS> system0.sim fact.bin
1SECONdS> ... (シミュレーションを続行)
```

2.1 system0

system0 は, キャッシュなしの DLX プロセッサと, 二基のメモリおよび I/O ポートを接続した環境です. 図 3は, 各要素の論理的な接続関係を表しています.

二基のメモリ SYS, USR は, ともに モジュール memory のインスタンスで, 128 Mb のメモリ空間を持っています. 実メモリを二基に分けた理由は, system0, 1, 2 で, アセンブリ・ソース変更しなくても扱えるようにするためです. 後述しますが, system2 では, *PCI* のコンフィギュレーション・メカニズム 1 をサポートしており, system1 でもこのメカニズムにならったコンフィギュレーションをサポートしています. つまり, コンフィギュレーションをプロセッサのプログラム実行によって行なうわけですから, コンフィギュレーションが完了するよりも以前に, プロセッサがアクセスできるメモリ空間が必要になります. そこで, プロセッサは常に SYS にアクセスできるようにします.

また, I/O ポートは図 3 の通り memory の内部に実装されており, 1 word (4 byte) 単位で 4 ポートあります. 各ポートの機能について説明すると, まず, memory は内部に Time, Rcnt, Wcnt という 3 個の 32 bit カウンタを内蔵しています. Time はクロックをカウントします. Rcnt は memory に対するリード・アクセスの回数をカウントします. Wcnt はライト・アクセスのカウンタです. 先の I/O ポートをアクセスすることにより, これらのカウンタの値にアクセスすることができます.

アドレス	機能 (リード)	機能 (ライト)
00000000 - 00000003	Time の値を返す	Time := 書き込み値
00000004 - 00000007	Rcnt の値を返す	Rcnt := 書き込み値
00000008 - 0000000B	Wcnt の値を返す	Wcnt := 書き込み値
00000008 - 0000000B	Rcnt + Wcnt の値を返す	Rcnt := 書き込み値, Wcnt := 書き込み値

なお, 実際に使っているのは USR 内部の I/O ポートのみで, SYS 内部の I/O ポートは使っていません.

さて, DLX のアドレス空間は, 単一の 32 ビット空間であり, メモリ空間と I/O 空間を区別しません. したがって, 上記のメモリおよび I/O ポートは, 図 6 のようにマップします. SYS の 128 Mbyte メモリは DLX アドレス 00000000 - 00FFFFFF にマップされます. USR の 128 Mb メモリは DLX アドレス 08000000 - 0FFFFFFF にマップされます. また, DLX アドレス FFFF0000 - FFFFFFFF の 64 Kbyte の空間は I/O 空間にマップされます. DLX がこれ以外のアドレスにアクセスすると address error となり, シミュレーション上は SECONDS の Multiple Write エラーが発生するようにしてあります.

また, USR の I/O ポートは I/O 空間内の 0000FF00 - 0000FF0F の 16 byte 空間にマップします. シミュレーション中にこれ以外の I/O 空間にアクセスがあった場合も, Mutiple Write エラーが発生します. 例外として, 0000cf8 - 0000cfb への word (4 byte) アクセス, および, 0000cfc - 0000cff 内の任意アクセスは許されます. この場合, ライト・アクセスは何の作用もなく, リード・アクセスには 00000000 が返ります. この例外的扱いは, system1, system2 で用いている *PCI* コンフィギュレーション・メカニズム 1 との兼ね合いです. すなわち, system1, system2 用の DLX プログラムに含まれるコンフィギュレーション・プログラムを, そのまま何の弊害もなく system0 で動作させるためです.

module map のインスタンス MAP は, DLX プロセッサのアドレスを監視し, メモリや I/O ポートにアクセスを振り分けるための制御を行なう回路です.

2.1.1 シミュレーション・ログの見方

system0 でシミュレーションを行なうと, 1 クロック進める毎にリスト C.1 のようなレポートが得られます. この例を使ってレポートの見方について簡単に説明します.

なお, 先頭が '! タグ名 !' で始まっている行は, SECONDS 上で

```
SECONDS> rpt_off タグ名
```

とすることにより, その行の表示を抑制することができます. 抑制を解きたい場合は

```
SECONDS> rpt_on タグ名
```

とすることにより, 再び表示するようになります.

1 行 環境 system0 を使って, プログラム fact.bin の実行をシミュレートしており, 現在 496 クロック目です.

3 行 (pcsv.rpt) DLX 内部のプログラム・カウンタの値と, プログラム・カウンタへのアクセス情報を表します. 現在のプログラム・カウンタ値は 08000044 です.

4 - 5 行 (dlx.rpt) DLX のメモリ・アクセスの様子を表します. アドレス 08000044 に対して 命令リード を要求しており, 値 0bfffffc が返ってきて (iok(1)) います.

14 - 17 行 (rfsv.rpt) DLX の各整数レジスタ値を表示しています.

19 - 21 行 (iaccess.rpt) SYS の iAccess インターフェースの状態を表示します.

23 - 25 行 (iaccess.rpt) USR の iAccess インターフェースの状態を表示します. アドレス 00000044, バイト・イネーブル 1111 の ReadMemReq 要求を受けて, 値 0bfffffc を返しています.

27 - 42 行 (fact.bin) 実行しているプログラム固有のデータ領域の値を表示します. アセンブリ・プログラムにおいて, ラベルが付けられたデータ領域でサイズが 1 byte 以上のものについては, 自動的にレポート内に表示します. 各データ領域の表示の抑制, および再表示化については

```
SECONDS> rpt_off ラベル名
SECONDS> rpt_on ラベル名
```

で, それぞれ行なえます.

2.2 system1

system1 は, キャッシュを内蔵した DLX プロセッサと, 二基のメモリおよび I/O ポートを接続した環境です. 図 4 は, 各要素の論理的な接続関係を表しています. SYS, USR, MAP は system0 と同じものです.

system1 では アドレス空間のマップとして図 7 の形態を採ります. このマッピングは, system0 で用いたマッピングと基本的には同じものになります. ただし, 図 7 において PCI メモリ空間および PCI I/O 空間のどの位置に, USR のメモリ空間および I/O 空間をマップするかを, コンフィギュレーションによって設定するところが system0 と違います. system0 では, これに相当するマッピングは固定でした.

system1 は *PCI* バスを備えているわけではありませんが, *PCI* のコンフィギュレーション・メカニズム 1 にならったコンフィギュレーションをサポートしています. このためにコンフィギュレーション・レジスタ二基 (DCR, UCR) およびブリッジ回路 (BRI) を備えています. DCR, UCR は, それぞれ DLX, USR のためのレジスタであり, DLX と USR との結合に関する設定を保持します. BRI は, DLX の I/O アクセスをスヌープし, DCR, UCR へのアクセスを司ります.

コンフィギュレーションによって, 次の各項目を設定する必要があります.

- UCR メモリ・ベース・アドレス, I/O ベース・アドレスを設定することにより, メモリ空間と I/O 空間の, PCI アドレス空間へのマッピングを設定する.
- DCR の command レジスタの, Bus Master bit, および UCR の command レジスタの Memory Space bit, I/O Space bit をセットすることにより, DLX から USR をアクセスできるようにする. これらのビットがセットされていない状態では, DLX - USR 間のバスは切断状態になります.
- DCR, UCR の Cache Line Size レジスタの値をともに 4 にする.

PCI のコンフィギュレーション・レジスタは 64 byte のヘッダを含む 256 byte のレジスタ・ファイルですが, 同一の byte で読み取り専用ビットと書き込み可能ビットが混在したり, 同じビットでもデバイスによって読み取り専用であったり書き込み可能であったりします. 特にベース・アドレス・レジスタなどは, デバイスの持つアドレス空間の大きさを, ビットの書き込み可能性で表現するため, 当然デバイスによって違いが生

じます．また, リセット後のレジスタの初期値がデバイスによって異なることもあり得ます．そういったレジスタの動作を同一の SFL モジュールとして記述するために, 次のような仕組みを採りました．

まず, *PCI* 環境では creg00.sfl の中で記述している creg00 というモジュールが, コンフィギュレーション・レジスタ回路です．このモジュール内では, レジスタ・ファイル内の 1 バイトを R，M，I の 3 バイトで表します．R はレジスタの値そのものを常に表します．M は書き込み可能マスクで, このバイト内で 1 が立っているビット位置は, 書き込み可能であることを表します．I は初期値で, creg00 にリセットがかかった時, I の値が R に代入されます．I，M は回路動作中に値を変化させることはありませんが, メモリとして記述されていますので, SECONDS の memset 文により値を設定できます．これにより, インスタンス毎にデバイス依存の情報を設定できます．例えば DCR，UCR はそれぞれ, DPCI.reg，UPCI.reg という SECONDS スクリプトによって設定しています．

2.2.1 シミュレーション・ログの見方

system1 でシミュレーションを行なうと, 1 クロック進める毎にリスト C.2 のようなレポートが得られます．

18 - 19 行 (cache.rpt)　/DLX/DLX から, キャッシュへのアクセスの状態を表示します．現在 /DLX/DLX がアドレス 08000048 に対して命令リード (inst:read(1)) を要求しており, それがキャッシュラインにヒット (hit(1)) して, 値 (00000020) を返しています．

20 - 22 行 (cache.rpt)　キャッシュから DLX 外部へ向かう, メモリアクセスの状態を表示します．

23 - 30 行 (cache.rpt)　各キャッシュラインの状態を表示します．

31 - 33 行 (iaccess.rpt)　DLX の iAccess インタフェースの状態を表示します．

31 - 33 行 (iaccess.rpt)　DLX の iAccess インタフェースの状態を表示します．

43 - 46 行 (creg00.rpt)　DCR (コンフィギュレーション・レジスタ) のヘッダ部分を表示します．

48 - 51 行 (creg00.rpt)　UCR (コンフィギュレーション・レジスタ) のヘッダ部分を表示します．

2.3 system2

system2 は, キャッシュを内蔵した DLX プロセッサと, 二基のメモリおよび I/O ポートを接続した環境です．図 5は, 各要素の論理的な接続関係を表しています．DLX，SYS，USR，MAP，BRI は system1 と同じものです．DLX と USR の間は, *PCI* バスによってデータ転送が行なわれます．*PCI* バスの信号接続は, PCIbus モジュールのインスタンス BUS が行ないます．

コンテストの規定課題である *PCI* バス・インターフェース回路 PCI のインスタンス DPCI，UPCI が, それぞれ DLX，USR を *PCI* バスに接続しています．*PCI* 環境は, サンプルの PCI モジュールを用意しています．設計した PCI をデバッグする際にはこのサンプルを設計回路で置き換えて下さい．

PCI バスには, DLX，USR の他に BRI も接続されています．BRI は system0,1 で用いたものと同じブリッジ回路です．BRI と *PCI* バスの間は *PCI* バス・インターフェース回路 PCIb のインスタンス BPCI が接続します．もし, 設計回路がコンフィギュレーション・サイクルのイニシエータ動作をサポートしているならば, その記述を PCIb.sfl というファイル名でコピーし, モジュール名を PCIb と替えることによって, サンプル回路を置き換えてみて下さい．

2.3.1 シミュレーション・ログの見方

system2 でシミュレーションを行なうと, 1 クロック進める毎にリスト C.3 のようなレポートが得られます．

43 - 46 行 (creg00.rpt)　DPCI 内部のコンフィギュレーション・レジスタのヘッダ部分を表示します．

48 - 51 行 (creg00.rpt)　UPCI 内部のコンフィギュレーション・レジスタのヘッダ部分を表示します．

53 - 56 行 (creg00.rpt)　BPCI 内部のコンフィギュレーション・レジスタのヘッダ部分を表示します．

58 - 59 行 (PCIbus.rpt)　*PCI* バスの各信号線の値を表示します．

60 - 62 行 (PCIbus.rpt)　*PCI* バスの各信号線の値を, どの PCI インタフェース回路がドライブしているかを表示します. enb1, enb2, enb3 は, それぞれ DPCI, UPCI, BPCI に該当します.

64 - 68 行 (PCI.rpt)　DPCI の各種状態を表示します. 現在, イニシエータとして動作中です.

70 - 74 行 (PCI.rpt)　UPCI の各種状態を表示します. 現在, アイドルです.

76 - 80 行 (PCI.rpt)　BPCI の各種状態を表示します. 現在, アイドルです.

3 例題プログラム

3.1 fact.asm

5 - 10 行　図 7 のアドレス・マップに基づいて, system area, user area, mapped I/O の各領域の先頭番地に, それぞれ sys, user, io というラベルをつけています.

13 - 23 行　USR の I/O ポートにアクセスするための各アドレスに, それぞれラベル Ccnt, Rcnt, Wcnt, RWcnt をつけています. また, この領域の先頭に IOcnt というラベルをつけています.

26 - 66 行　ブート・プログラムのコード部分です. このセグメントは, アドレス 00000000 から始まらなければなりません.

69 - 132 行　ブート・プログラムで行なうコンフィギュレーションのためのデータ領域です. このセグメントは, アドレス空間中の system area 内に配置されなければなりません. コンフィギュレーションデータは, 次の 4 ワード (16 バイト) を 1 エントリとし, エントリの配列の形態をとります. 1 エントリにつき 1 ワードのコンフィギュレーション・レジスタを設定します.

ワード 0	CONFIG_ADDRESS に書き込む値
ワード 1	コンフィギュレーション・レジスタに書き込む値
ワード 2	コンフィギュレーション・レジスタに書き込む値の e ビット毎のイネーブルマスク
ワード 3	エンド・マーク (00000000 が, 配列の末尾をあらわす)

134 - 243 行　この部分が, ユーザ・プログラムです. プログラムの内容は次のようなものです. なお, この部分については, 文献 [1] を参考にしました.

自然数 n の階乗 $n!$ は, 次の C 言語のプログラムによって求めることができます.

```
int fact(int n);

int main(void)
{
    int n;

    return fact(n);
}

int fact(int n)
{
    return n > 1 ? n * fact(n - 1) : 1;
}
```

この fact.c を hand compile したものです. データ領域 N は, 階乗を求める数を格納しています. ここでは 10 進数で 10 としています. FACTN には $n!$ が格納されます. $10! =$ 00375f00 です. 領域 TIME には, このユーザプログラムの実行に掛かったクロック数 (の概算) が格納され, 領域 ACCESS には, このユーザ・プログラムの実行中に USR メモリに対して生じたアクセスの累計 (の概算) が格納されます. また, ユーザ・プログラムの実行が終了すると END に 0000eeee が書き込まれ, プログラムの実行は無限ループに落ちます.

stack00 ～ stack10 は 関数 fact() のスタック・フレームで, stack10, stack09, ..., stack00 の順に伸びていきます. $fp が現在のスタックフレームの先頭を指し, $sp が次のスタック・フレームの先頭を指しています. 1 つのスタック・フレームは 4 ワード (16 バイト) の大きさを持ち, 順に

ワード 0	`fact` への引数
ワード 1	`fact` の戻り値
ワード 2	`$fp` の保存値
ワード 3	戻り番地

の 4 ワードが格納されていきます.

3.1.1 プログラムの終了と動作確認

プログラム `fact.asm` の終了時点は,

「領域 `END` の値が `uuuuuuuu` から `0000eeee` に変化したクロック」

と定義します. ちなみに筆者の用意した各シミュレーション環境では, 終了時点は次のようになりました.

system0	496
system1	623
system2	1007

もちろん, `PCI` を設計回路で置き換えると `system2` における値が変わる可能性があります.
プログラムが正常に実行されたかどうかは, 終了時点において, 領域 `FACTN`, `END`, `stack00` $\sim$ `stack10` の値が正しいかどうかで判定します. 正しい値とは, リスト C.1 のシミュレーション・ログに示されている値です.

参考文献

[1] David A.Patterson, John L.Hennessy: 成田 光彰訳. コンピュータの構成と設計 下. 日経 BP 社, 1996.

[2] David A.Patterson, John L.Hennessy: 富田眞治 村上 和彰 新實治男訳. コンピュータ・アーキテクチャ. 日経 BP 社, 1992.

[3] 永見康一. DASM – DLX assembler for SECONDS –. http://www.kecl.ntt.co.jp/car/parthe/html/psfile/dasm_man.ps, 1996.

A シミュレーション環境のファイル構成

SECONDS スクリプト		
ファイル名	対象 SFL module	備考
system0.sim	-	system0 環境のメイン・スクリプト
system1.sim	-	system1 環境のメイン・スクリプト
system2.sim	-	system2 環境のメイン・スクリプト
PCI.rpt	PCI	PCI インタフェース回路の各種状態を rpt_add
PCIb.rpt	PCIb	PCI インタフェース回路の各種状態を rpt_add
cache.rpt	cache	DLX 内部キャッシュの各種状態を rpt_add
creg00.rpt	creg00	Config. Reg. 回路の各レジスタ値を rpt_add
dlx.rpt	dlx	DLX プロセッサの各種状態を rpt_add
dlxc.rpt	dlxc	DLX プロセッサと内部キャッシュの各種状態を rpt_add
iaccess.rpt	iAccess を使う module	iAccess インタフェースの各信号値を rpt_add
pcsv.rpt	pcsv	DLX プロセッサの PC 制御部の各種状態を rpt_add
rfsv.rpt	rfsv	DLX プロセッサのレジスタファイルの状態を rpt_add
dlx.ld		dlx を構成する module の .mc ファイルを sflload
dlxc.ld		dlxc を構成する module の .mc ファイルを sflload
DPCI.reg	creg00	bridge (ターゲット) 用 Config. Reg. の初期設定
UPCI.reg	creg00	DLX (イニシエータ) 用 Config. Reg. の初期設定
BCI.reg	creg00	birdge (ブリッジ) 用 Config. Reg. の初期設定

SFL 記述	
ファイル名	備考
PCI.{h, sfl}	PCI インタフェース回路のサンプル
PCIb.{h, sfl}	PCI インタフェース回路 (ブリッジ用) のサンプル
	(実際は, PCIb.sfl は PCI.sfl と同一)
PCI.tpl	PCI.sfl, PCIb.sfl のテンプレート
PCIbus.{h, sfl}	system2 の PCI バス接続回路
alu32.{h, sfl}	dlx の ALU 回路
idec.{h, sfl}	dlx の命令デコード回路
malign.{h, sfl}	dlx のメモリ・アクセス整列化回路
mult32.{h, sfl}	dlx の整数乗算回路
pcsv.{h, sfl}	dlx の PC 制御回路
rfsv.{h, sfl}	dlx のレジスタ・ファイル制御回路
regfile.{h, sfl}	rfsv 内部のレジスタ・アレイ回路
dlx.{h, sfl}	DLX プロセッサ本体
cache.{h, sfl}	dlxc の内部キャッシュ回路
dlxc.{h, sfl}	内部キャッシュつき DLX プロセッサ本体
system0.{h, sfl}	system0 環境
system1.{h, sfl}	system1 環境
system2.{h, sfl}	system2 環境
bridge.{h, sfl}	system[12] のブリッジ回路
map.{h, sfl}	system[012] のアドレス空間マッパ
memory.{h, sfl}	128 Mb メモリ + 4 I/O ポート回路
creg00.{h, sfl}	コンフィギュレーション・レジスタ回路
iaccess.{pin, cpa, pca}	iAccess 信号線の宣言テンプレート

その他	
ファイル名	備考
Makefile	シミュレーション環境のメイクファイル
fact.asm	例題プログラムのアセンブリ・ソース
fact.bin	fact.asm のアセンブル結果
dasm	SECONDS 用 DLX アセンブラ (SparcStation + SunOS 4.1.4)
fact0.log.Z	system0 上で fact.bin を実行した結果のログファイル (compress 圧縮形式)
fact1.log.Z	system1 上で fact.bin を実行した結果のログファイル (compress 圧縮形式)
fact2.log.Z	system2 上で fact.bin を実行した結果のログファイル (compress 圧縮形式)

B 図

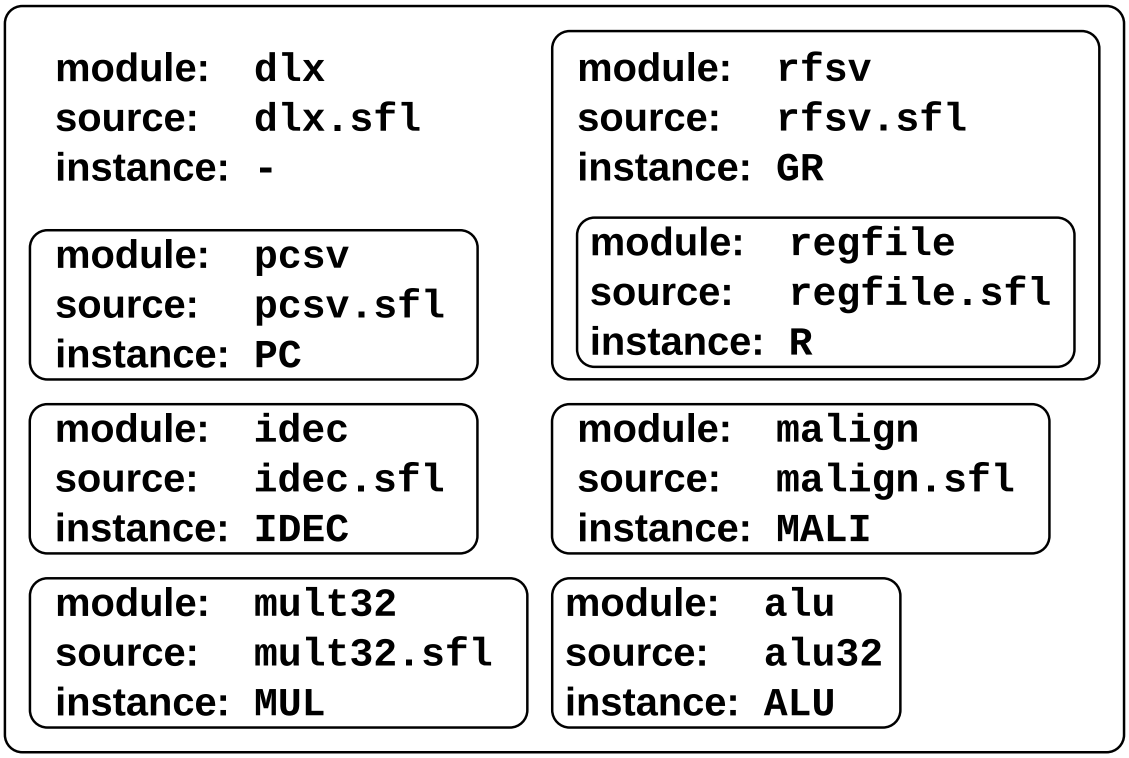

図 1: `dlx` の SFL モジュール階層

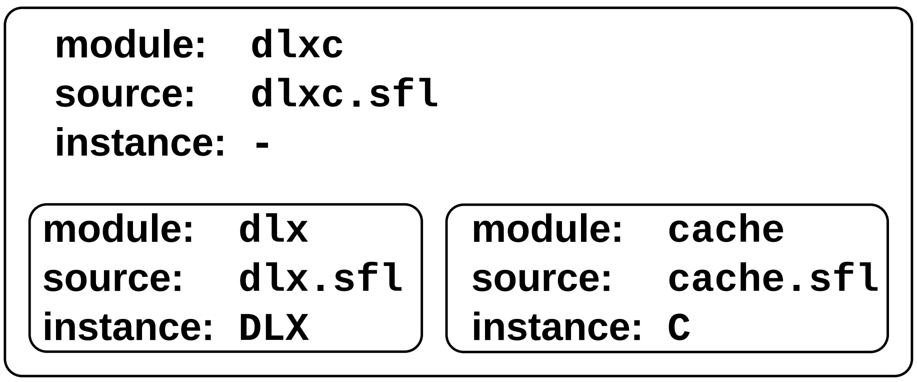

図 2: `dlxc` の SFL モジュール階層

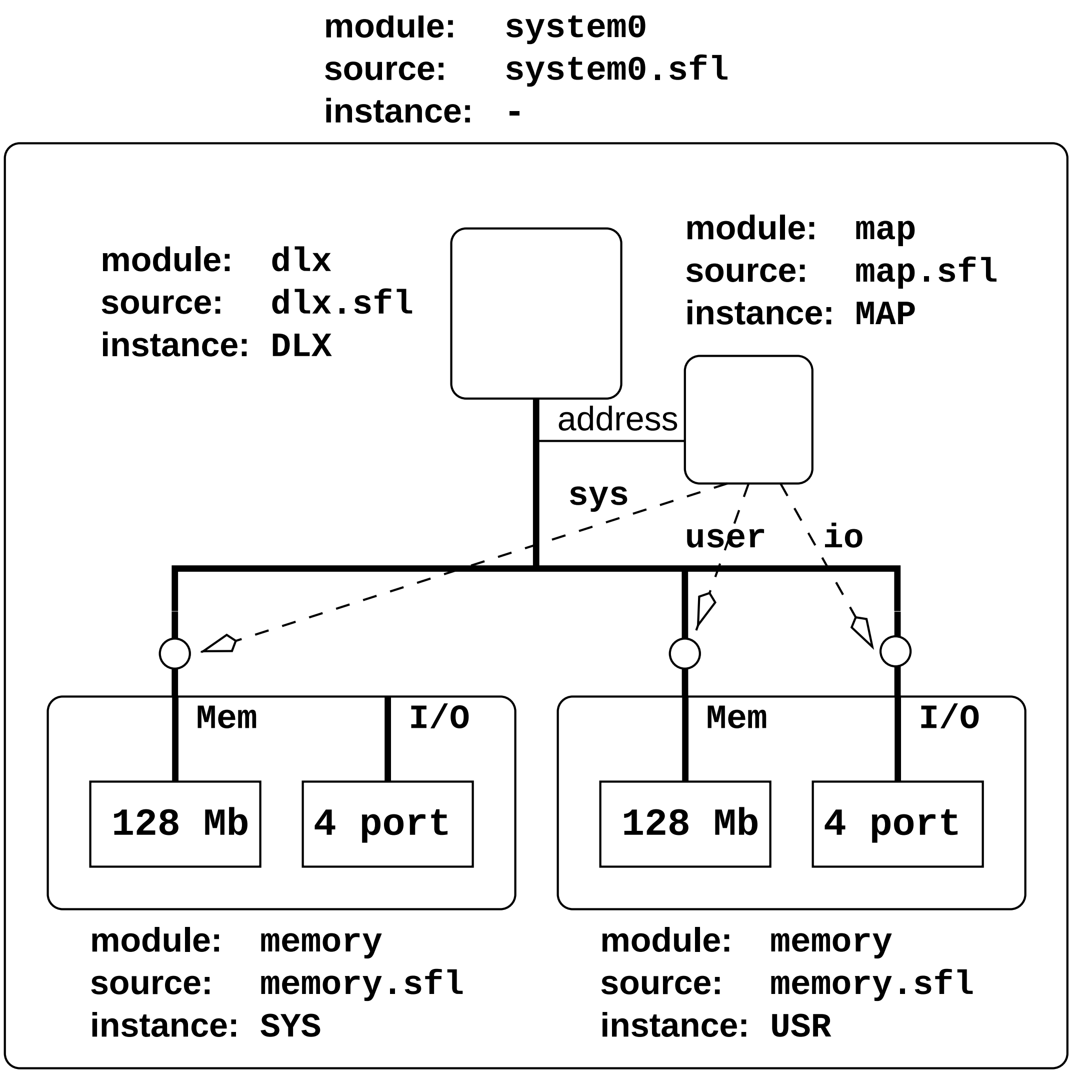
module: system0
source: system0.sfl
instance: -
module: dlx
source: dlx.sfl
instance: DLX
module: map
source: map.sfl
instance: MAP
address
sys
user
io
Mem
I/O
128 Mb
4 port
Mem
I/O
128 Mb
4 port
module: memory
source: memory.sfl
instance: SYS
module: memory
source: memory.sfl
instance: USR

図 3: system0

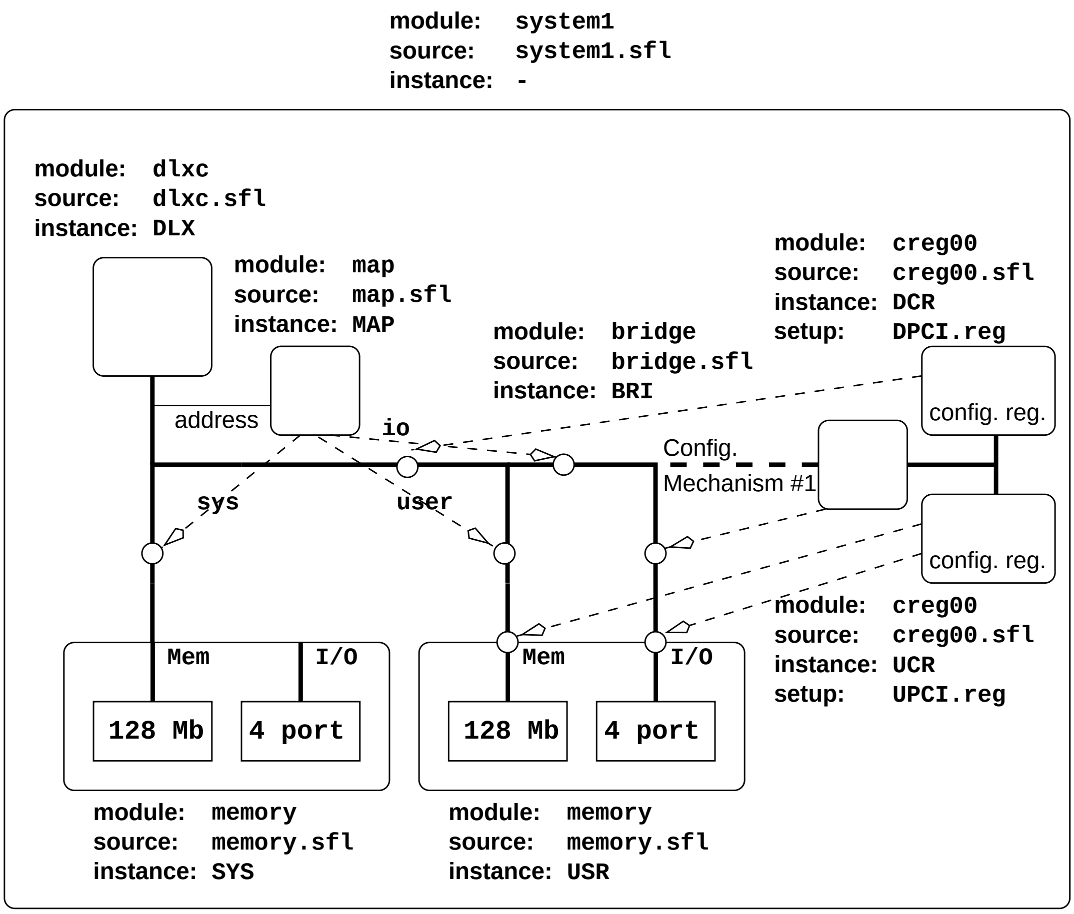
module: system1
source: system1.sfl
instance: -
module: dlxc
source: dlxc.sfl
instance: DLX
module: map
source: map.sfl
instance: MAP
module: bridge
source: bridge.sfl
instance: BRI
module: creg00
source: creg00.sfl
instance: DCR
setup: DPCI.reg
config. reg.
address
io
Config.
Mechanism #1
sys
user
config. reg.
module: creg00
source: creg00.sfl
instance: UCR
setup: UPCI.reg
Mem
I/O
128 Mb
4 port
Mem
I/O
128 Mb
4 port
module: memory
source: memory.sfl
instance: SYS
module: memory
source: memory.sfl
instance: USR

図 4: system1

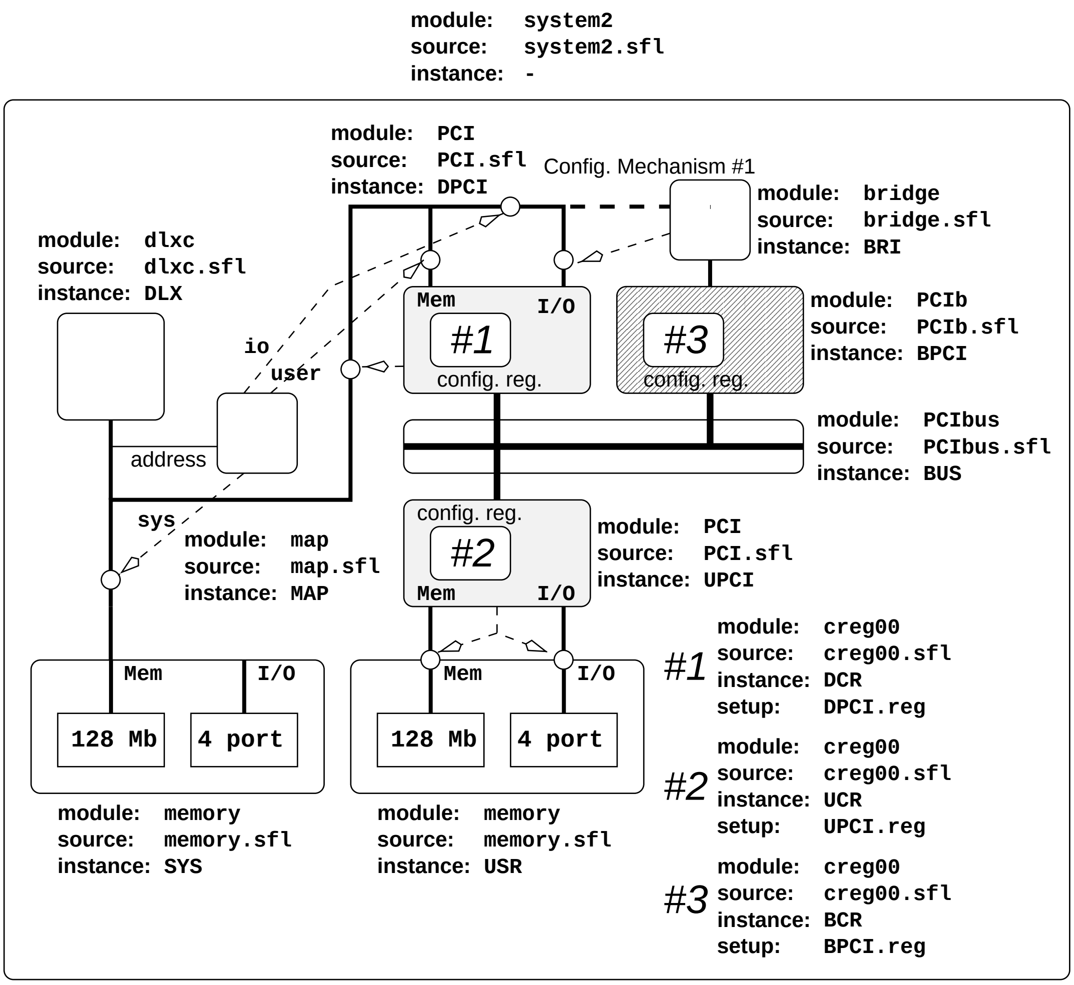
module: system2
source: system2.sfl
instance: -
module: PCI
source: PCI.sfl
instance: DPCI
Config. Mechanism #1
module: bridge
source: bridge.sfl
instance: BRI
module: dlxc
source: dlxc.sfl
instance: DLX
io
user
Mem
I/O
#1
config. reg.
#3
config. reg.
module: PCIb
source: PCIb.sfl
instance: BPCI
module: PCIbus
source: PCIbus.sfl
instance: BUS
address
sys
module: map
source: map.sfl
instance: MAP
config. reg.
#2
Mem
I/O
module: PCI
source: PCI.sfl
instance: UPCI
Mem
I/O
128 Mb
4 port
Mem
I/O
128 Mb
4 port
module: memory
source: memory.sfl
instance: SYS
module: memory
source: memory.sfl
instance: USR
#1
module: creg00
source: creg00.sfl
instance: DCR
setup: DPCI.reg
#2
module: creg00
source: creg00.sfl
instance: UCR
setup: UPCI.reg
#3
module: creg00
source: creg00.sfl
instance: BCR
setup: BPCI.reg

図 5: system2

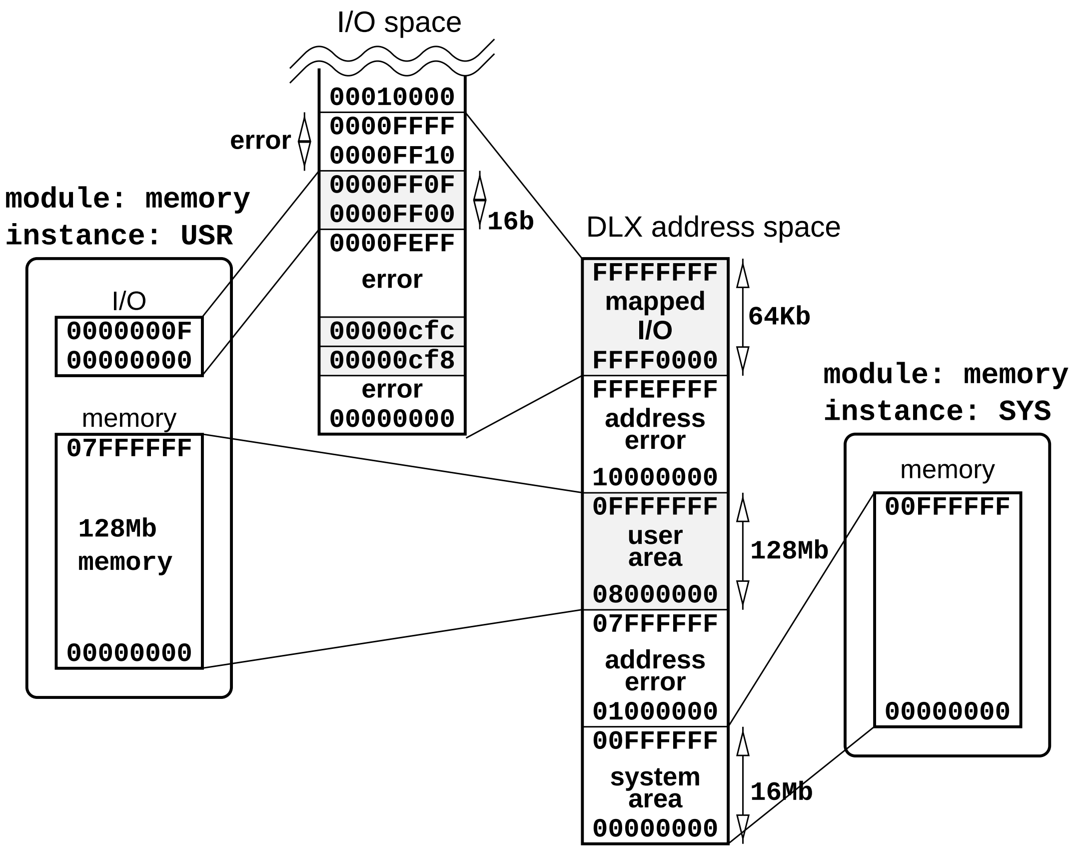

図 6: system0 のメモリ, I/O マップ

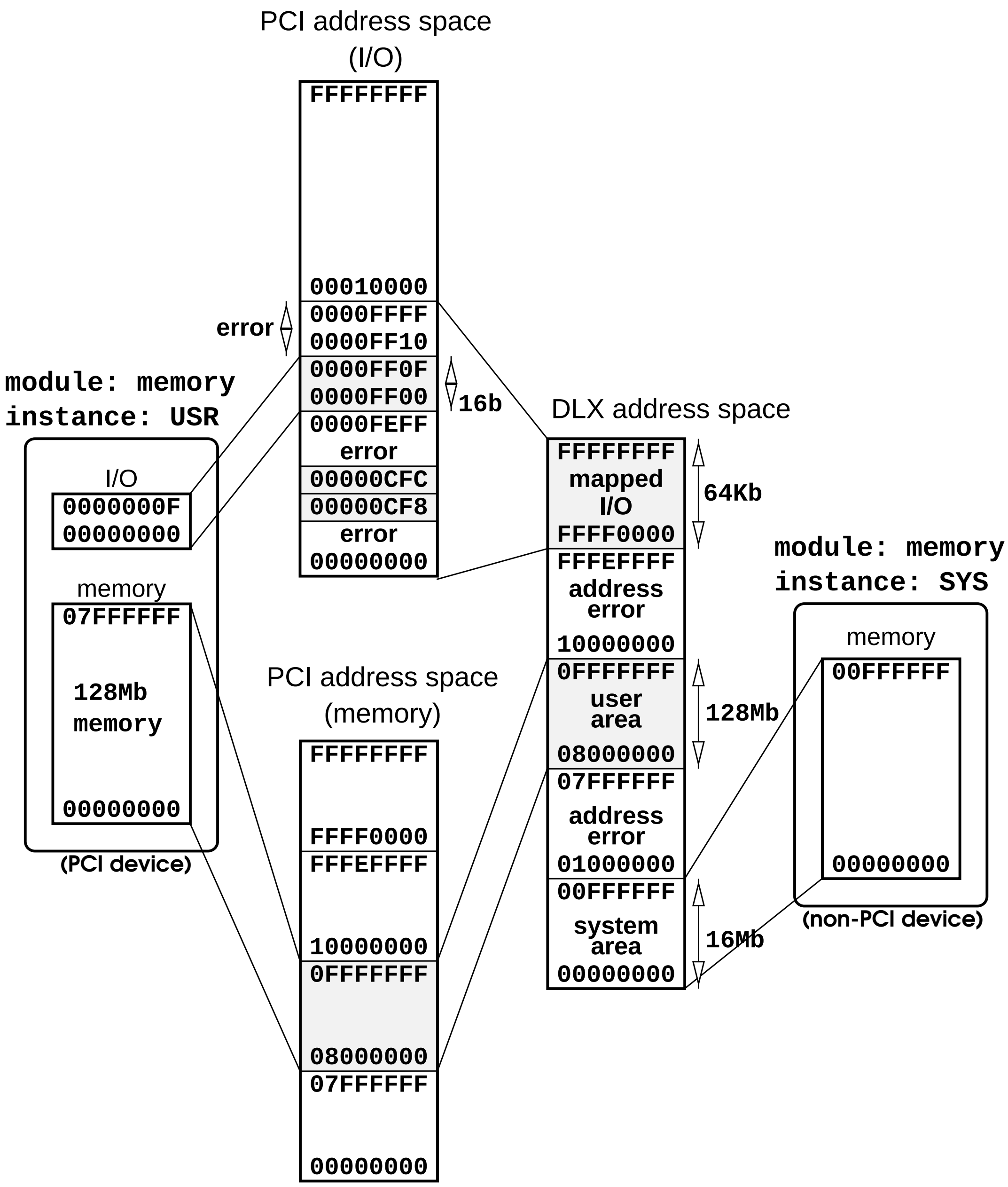

図 7: `system1`, `system2` のメモリ, I/O マップ

C　リスト

C.1　system0 のシミュレーション・ログ

```
#--- simulation system0.sim fact.bin :clock 496 ---
#-- DLX Info.
!dlPc    ! PC:08000044 rst(0)put(0)(in(zzzzzzzz))add(0)inc(1)get(1)(out(08000044))
!dlMem   ! inst:read(1)        ->iok(1):[adr(08000044)         ](data(0bfffffc))
!dlMem   ! data:read(0)write(0)->dok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dlPipe  ! runSV(active(1)state:run)
!dlPipe  !                  |     |     |    MGFU**..... |    MGFU**..... |    MGFU**..... |
!dlPipe  ! active/wait(tag):|1/0(_)|1/0(_)|1/0(00000000000)|0/0(10001100000)|0/0(01011101000)|
!dlLatch ! ID :[ir   00000020]
!dlLatch ! EX :[opd1 00000000 opd2 fffffffc opd3 00000000 op 000000]
!dlLatch ! MEM:[mar  08002208 data 0000eeee]
!dlLatch ! WB :[d    0000eeee]
!dlGr0   ! read1(1)[#(00)](data(0000)) read2(1)[#(00)](data(0000)) write(0)[#(zz)](data(zzzz))
!dlGr1   ! r0:00000000 at:00000000 v0:00375f00 v1:0000000a a0:00000001 a1:00000000 a2:00000000 a3:00000000
!dlGr2   ! t0:0000eeee t1:00000004 t2:ffffff00 t3:00000000 t4:00000000 t5:00000000 t6:00000000 t7:00000148
!dlGr3   ! s0:00000000 s1:00000000 s2:00000000 s3:00000000 s4:00000000 s5:00000000 s6:00000000 s7:00000000
!dlGr4   ! t8:00000147 t9:ffffff00 k0:00000000 k1:00000000 gp:08002200 sp:0800ffb0 fp:0800ffc0 ra:08000028
#-- iAccess (SYS) Info.
!iaccSYS ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccSYS ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccSYS ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- iAccess (USR) Info.
!iaccUSR ! Adr(00000044|zzzzzzzz) Be(1111|zzzz) Data(zzzzzzzz|0bfffffc)
!iaccUSR ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccUSR ! ReadReq: M(1) ML(0) I(0)     ->done(1) WriteReq: M(0) I(0)     ->done(0)
#-- fact.bin Info.
N          [08002200]: 00 00 00 0a
FACTN      [08002204]: 00 37 5f 00
END        [08002208]: 00 00 ee ee
TIME       [0800220c]: 00 00 01 48
ACCESS     [08002210]: 00 00 01 47
stack10    [0800ff00]: uu uu uu uu uu uu uu uu 08 00 ff 20 08 00 00 bc
stack09    [0800ff10]: 00 00 00 01 00 00 00 01 08 00 ff 30 08 00 00 bc
stack08    [0800ff20]: 00 00 00 02 00 00 00 02 08 00 ff 40 08 00 00 bc
stack07    [0800ff30]: 00 00 00 03 00 00 00 06 08 00 ff 50 08 00 00 bc
stack06    [0800ff40]: 00 00 00 04 00 00 00 18 08 00 ff 60 08 00 00 bc
stack05    [0800ff50]: 00 00 00 05 00 00 00 78 08 00 ff 70 08 00 00 bc
stack04    [0800ff60]: 00 00 00 06 00 00 02 d0 08 00 ff 80 08 00 00 bc
stack03    [0800ff70]: 00 00 00 07 00 00 13 b0 08 00 ff 90 08 00 00 bc
stack02    [0800ff80]: 00 00 00 08 00 00 9d 80 08 00 ff a0 08 00 00 bc
stack01    [0800ff90]: 00 00 00 09 00 05 89 80 08 00 ff b0 08 00 00 64
stack00    [0800ffa0]: 00 00 00 0a 00 37 5f 00 08 00 ff c0 08 00 00 28
```

C.2 system1 のシミュレーション・ログ

```
#--- simulation system1.sim fact.bin :clock 623 ---
#-- DLX Info.
!dlPc    ! PC:08000048 rst(0)put(1)(in(08000044))add(0)inc(1)get(1)(out(08000048))
!dlMem   ! inst:read(1)        ->iok(1):[adr(08000048)         ](data(00000020))
!dlMem   ! data:read(0)write(0)->dok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dlPipe  ! runSV(active(1)state:run)
!dlPipe  !                 |      |      |    MGFU**..... |    MGFU**..... |    MGFU**..... |
!dlPipe  ! active/wait(tag):|1/0(_)|1/0(_)|0/0(10001100000)|0/0(10001100000)|0/0(01011101000)|
!dlLatch ! ID :[ir   0bfffffc]
!dlLatch ! EX :[opd1 08002200 opd2 00000008 opd3 0000eeee op 000000]
!dlLatch ! MEM:[mar  08002208 data 0000eeee]
!dlLatch ! WB :[d    0000eeee]
!dlGr0   ! read1(0)[#(zz)](data(zzzz)) read2(0)[#(zz)](data(zzzz)) write(0)[#(zz)](data(zzzz))
!dlGr1   ! r0:00000000 at:00000000 v0:00375f00 v1:0000000a a0:00000001 a1:00000000 a2:00000000 a3:00000000
!dlGr2   ! t0:0000eeee t1:00000004 t2:ffffff00 t3:00000000 t4:00000000 t5:00000000 t6:00000000 t7:00000177
!dlGr3   ! s0:00000000 s1:00000000 s2:00000000 s3:00000000 s4:00000000 s5:00000000 s6:00000000 s7:00000000
!dlGr4   ! t8:000000ab t9:ffffff00 k0:00000000 k1:00000000 gp:08002200 sp:0800ffb0 fp:0800ffc0 ra:08000028
!dcCache0! inst:read(1)        hit(1)->iok(1):[adr(08000048)         ](data(00000020))
!dcCache0! data:read(0)write(0)hit(0)->dok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dcCache1! load(0)loadLine(0)        ->lok(0):[adr(zzzzzzzz)         ](data(zzzzzzzz))
!dcCache1! store(0)                  ->sok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dcCache1! loading(0) to line(4), position(0+0)
!dcCache2! 0:1:0800ff:uuuuuuuu uuuuuuuu 0800ff20 080000bc 1:1:0800ff:00000001 00000001 0800ff30 080000bc
!dcCache2! 2:1:080000:af20000c 0c000024 00000020 8f2f0000 3:1:080000:8f38000c af8f000c af980010 3408eeee
!dcCache3! 4:1:080000:af880008 0bfffffc 00000020 2bbd0010 5:1:0800ff:00000005 00000078 0800ff70 080000bc
!dcCache3! 6:1:080000:0c00001c 00000020 af820004 8fbf000c 7:1:080000:8fbe0008 23bd0010 4be00000 00000020
!dcCache4! 8:1:0800ff:00000008 00009d80 0800ffa0 080000bc 9:1:0800ff:00000009 00058980 0800ffb0 08000064
!dcCache4! a:1:0800ff:0000000a 00375f00 0800ffc0 08000028 b:1:080000:00000020 28840001 0ffffc4 00000020
!dcCache5! c:1:080000:8fc30000 00000020 00431018 afc20004 d:1:080000:8fbf000c 8fbe0008 23bd0010 4be00000
!dcCache5! e:1:080000:00000020 uuuuuuuu uuuuuuuu uuuuuuuu f:0:uuuuuu:uuuuuuuu uuuuuuuu uuuuuuuu uuuuuuuu
!iaccCPU ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccCPU ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccCPU ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- iAccess (SYS) Info.
!iaccSYS ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccSYS ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccSYS ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- iAccess (USR) Info.
!iaccUSR ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccUSR ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccUSR ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- Config. Reg.(CPU) info.
!cr0DCR  ! D(4567) V(bbbb) C(00)(00)(00) R(01) BIST(0000) H.T.(00) L.T.(00)
!cr0DCR  ! MAX_LAT(04) MIN_GNT(00) Int.Pin(00) Int.Line(00)
!cr1DCR  ! B.A.R. 0:00000001 1:00000000 2:00000000 3:00000000 4:00000000 5:00000000 ROM:00000000
!cr1DCR  ! Status:(0000000000000000) Command:(0000000101000100) Cache Line Size(04)
#-- Config. Reg.(USR) info.
!cr0UCR  ! D(0123) V(bbbb) C(00)(00)(05) R(01) BIST(0000) H.T.(00) L.T.(00)
!cr0UCR  ! MAX_LAT(04) MIN_GNT(00) Int.Pin(00) Int.Line(00)
!cr1UCR  ! B.A.R. 0:0000ff01 1:08000000 2:00000000 3:00000000 4:00000000 5:00000000 ROM:00000000
!cr1UCR  ! Status:(0000000000000000) Command:(0000000000000011) Cache Line Size(04)
#-- fact.bin Info.
N         [08002200]: 00 00 00 0a
FACTN     [08002204]: 00 37 5f 00
END       [08002208]: 00 00 ee ee
TIME      [0800220c]: 00 00 01 77
ACCESS    [08002210]: 00 00 00 ab
stack10   [0800ff00]: uu uu uu uu uu uu uu uu 08 00 ff 20 08 00 00 bc
stack09   [0800ff10]: 00 00 00 01 00 00 00 01 08 00 ff 30 08 00 00 bc
stack08   [0800ff20]: 00 00 00 02 00 00 00 02 08 00 ff 40 08 00 00 bc
stack07   [0800ff30]: 00 00 00 03 00 00 00 06 08 00 ff 50 08 00 00 bc
stack06   [0800ff40]: 00 00 00 04 00 00 00 18 08 00 ff 60 08 00 00 bc
stack05   [0800ff50]: 00 00 00 05 00 00 00 78 08 00 ff 70 08 00 00 bc
stack04   [0800ff60]: 00 00 00 06 00 00 02 d0 08 00 ff 80 08 00 00 bc
stack03   [0800ff70]: 00 00 00 07 00 00 13 b0 08 00 ff 90 08 00 00 bc
stack02   [0800ff80]: 00 00 00 08 00 00 9d 80 08 00 ff a0 08 00 00 bc
stack01   [0800ff90]: 00 00 00 09 00 05 89 80 08 00 ff b0 08 00 00 64
stack00   [0800ffa0]: 00 00 00 0a 00 37 5f 00 08 00 ff c0 08 00 00 28
```

C.3 system2 のシミュレーション・ログ

```
#--- simulation system2.sim fact.bin :clock 1007 ---
#-- DLXC Info.
!dlPc     ! PC:08000048 rst(0)put(1)(in(08000044))add(0)inc(1)get(1)(out(08000048))
!dlMem    ! inst:read(1)        ->iok(1):[adr(08000048)         ](data(00000020))
!dlMem    ! data:read(0)write(0)->dok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dlPipe   ! runSV(active(1)state:run)
!dlPipe   !                  |      |      |    MGFU**..... |    MGFU**..... |    MGFU**..... |
!dlPipe   ! active/wait(tag):|1/0(_)|1/0(_)|1/0(00000000000)|1/0(00000000000)|0/0(01011101000)|
!dlLatch ! ID :[ir   0bfffffc]
!dlLatch ! EX :[opd1 00000000 opd2 00000000 opd3 00000000 op 000000]
!dlLatch ! MEM:[mar  fffffffc data 00000000]
!dlLatch ! WB :[d    0000eeee]
!dlGr0    ! read1(0)[#(zz)](data(zzzz)) read2(0)[#(zz)](data(zzzz)) write(0)[#(zz)](data(zzzz))
!dlGr1    ! r0:00000000 at:00000000 v0:00375f00 v1:0000000a a0:00000001 a1:00000000 a2:00000000 a3:00000000
!dlGr2    ! t0:0000eeee t1:00000004 t2:ffffff00 t3:00000000 t4:00000000 t5:00000000 t6:00000000 t7:0000029a
!dlGr3    ! s0:00000000 s1:00000000 s2:00000000 s3:00000000 s4:00000000 s5:00000000 s6:00000000 s7:00000000
!dlGr4    ! t8:000000ab t9:ffffff00 k0:00000000 k1:00000000 gp:08002200 sp:0800ffb0 fp:0800ffc0 ra:08000028
!dcCache0! inst:read(1)        hit(1)->iok(1):[adr(08000048)         ](data(00000020))
!dcCache0! data:read(0)write(0)hit(0)->dok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dcCache1! load(0)loadLine(0)        ->lok(0):[adr(zzzzzzzz)         ](data(zzzzzzzz))
!dcCache1! store(0)                  ->sok(0):[adr(zzzzzzzz):be(zzzz)](data(zzzzzzzz))
!dcCache1! loading(0) to line(4), position(0+0)
!dcCache2! 0:1:0800ff:uuuuuuuu uuuuuuuu 0800ff20 080000bc 1:1:0800ff:00000001 00000001 0800ff30 080000bc
!dcCache2! 2:1:080000:af20000c 0c000024 00000020 8f2f0000 3:1:080000:8f38000c af8f000c af980010 3408eeee
!dcCache3! 4:1:080000:af880008 0bfffffc 00000020 2bbd0010 5:1:0800ff:00000005 00000078 0800ff70 080000bc
!dcCache3! 6:1:080000:0c00001c 00000020 af820004 8fbf000c 7:1:080000:8fbe0008 23bd0010 4be00000 00000020
!dcCache4! 8:1:0800ff:00000008 00009d80 0800ffa0 080000bc 9:1:0800ff:00000009 00058980 0800ffb0 08000064
!dcCache4! a:1:0800ff:0000000a 00375f00 0800ffc0 08000028 b:1:080000:00000020 28840001 0ffffc4 00000020
!dcCache5! c:1:080000:8fc30000 00000020 00431018 afc20004 d:1:080000:8fbf000c 8fbe0008 23bd0010 4be00000
!dcCache5! e:1:080000:00000020 uuuuuuuu uuuuuuuu uuuuuuuu f:0:uuuuuu:uuuuuuuu uuuuuuuu uuuuuuuu uuuuuuuu
!iaccCPU ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccCPU ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccCPU ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- iAccess (SYS) Info.
!iaccSYS ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccSYS ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccSYS ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- iAccess (USR) Info.
!iaccUSR ! Adr(zzzzzzzz|zzzzzzzz) Be(zzzz|zzzz) Data(zzzzzzzz|zzzzzzzz)
!iaccUSR ! Read   : M(0) ML(0) I(0) C(0)->done(0) Write   : M(0) I(0) C(0)->done(0)
!iaccUSR ! ReadReq: M(0) ML(0) I(0)     ->done(0) WriteReq: M(0) I(0)     ->done(0)
#-- Config. Reg.(CPU) info.
!cr0DLX  ! D(4567) V(bbbb) C(00)(00)(00) R(01) BIST(0000) H.T.(00) L.T.(00)
!cr0DLX  ! MAX_LAT(04) MIN_GNT(00) Int.Pin(00) Int.Line(00)
!cr1DLX  ! B.A.R. 0:00000001 1:00000000 2:00000000 3:00000000 4:00000000 5:00000000 ROM:00000000
!cr1DLX  ! Status:(0000000000000000) Command:(0000000101000100) Cache Line Size(04)
#-- Config. Reg.(USR) info.
!cr0USR  ! D(0123) V(bbbb) C(00)(00)(05) R(01) BIST(0000) H.T.(00) L.T.(00)
!cr0USR  ! MAX_LAT(04) MIN_GNT(00) Int.Pin(00) Int.Line(00)
!cr1USR  ! B.A.R. 0:0000ff01 1:08000000 2:00000000 3:00000000 4:00000000 5:00000000 ROM:00000000
!cr1USR  ! Status:(0000000000000000) Command:(0000000000000011) Cache Line Size(04)
#-- Config. Reg.(BRI) info.
!cr0BRI  ! D(89ab) V(bbbb) C(00)(00)(06) R(01) BIST(0000) H.T.(00) L.T.(00)
!cr0BRI  ! MAX_LAT(04) MIN_GNT(00) Int.Pin(00) Int.Line(00)
!cr1BRI  ! B.A.R. 0:00000000 1:00000000 2:00000000 3:00000000 4:00000000 5:00000000 ROM:00000000
!cr1BRI  ! Status:(0000000000000000) Command:(0000000000000100) Cache Line Size(04)
#-- PCI(BUS) Info.
!pb0BUS   ! val.: RST#(1) AD(ffffffff) C/BE#(1111)
!pb0BUS   ! val.:                 PAR(0) FRAME#(1) IRDY#(1) TRDY#(1) STOP#(1) DEVSEL#(1) PERR#(1)
!pb1BUS   ! enb1: AD(0) C/BE#(0) PAR(1) FRAME#(0) IRDY#(1) TRDY#(0) STOP#(0) DEVSEL#(0) PERR#(0)
!pb2BUS   ! enb2: AD(0) C/BE#(0) PAR(0) FRAME#(0) IRDY#(0) TRDY#(0) STOP#(0) DEVSEL#(0) PERR#(0)
!pb3BUS   ! enb3: AD(0) C/BE#(0) PAR(0) FRAME#(0) IRDY#(0) TRDY#(0) STOP#(0) DEVSEL#(0) PERR#(0)
#-- PCI(DLX) Info.
!PCI0DLX  ! RST#(1) AD(0|ffffffff|zzzzzzzz) C/BE#(0|1111|zzzz) PAR(1|0|0) PERR#(0|1|z) SERR(0)
!PCI0DLX  ! FRAME#(0|1|z) IRDY#(1|1|1) TRDY#(0|1|z) STOP#(0|1|z) IDSEL(0) DEVSEL#(0|1|z) REQ#(1) GNT#(1)
#-- PCI(USR) Info.
!PCI0USR  ! RST#(1) AD(0|ffffffff|zzzzzzzz) C/BE#(0|1111|zzzz) PAR(0|0|z) PERR#(0|1|z) SERR(0)
!PCI0USR  ! FRAME#(0|1|z) IRDY#(0|1|z) TRDY#(0|1|z) STOP#(0|1|z) IDSEL(0) DEVSEL#(0|1|z) REQ#(1) GNT#(1)
#-- PCI(BRI) Info.
!PCI0BRI  ! RST#(1) AD(0|ffffffff|zzzzzzzz) C/BE#(0|1111|zzzz) PAR(0|0|z) PERR#(0|1|z) SERR(0)
!PCI0BRI  ! FRAME#(0|1|z) IRDY#(0|1|z) TRDY#(0|1|z) STOP#(0|1|z) IDSEL(0) DEVSEL#(0|1|z) REQ#(1) GNT#(1)
#-- fact.bin Info.
N          [08002200]: 00 00 00 0a
FACTN      [08002204]: 00 37 5f 00
END        [08002208]: 00 00 ee ee
TIME       [0800220c]: 00 00 02 9a
ACCESS     [08002210]: 00 00 00 ab
stack10    [0800ff00]: uu uu uu uu uu uu uu uu 08 00 ff 20 08 00 00 bc
stack09    [0800ff10]: 00 00 00 01 00 00 00 01 08 00 ff 30 08 00 00 bc
stack08    [0800ff20]: 00 00 00 02 00 00 00 02 08 00 ff 40 08 00 00 bc
stack07    [0800ff30]: 00 00 00 03 00 00 00 06 08 00 ff 50 08 00 00 bc
stack06    [0800ff40]: 00 00 00 04 00 00 00 18 08 00 ff 60 08 00 00 bc
stack05    [0800ff50]: 00 00 00 05 00 00 00 78 08 00 ff 70 08 00 00 bc
stack04    [0800ff60]: 00 00 00 06 00 00 02 d0 08 00 ff 80 08 00 00 bc
stack03    [0800ff70]: 00 00 00 07 00 00 13 b0 08 00 ff 90 08 00 00 bc
stack02    [0800ff80]: 00 00 00 08 00 00 9d 80 08 00 ff a0 08 00 00 bc
stack01    [0800ff90]: 00 00 00 09 00 05 89 80 08 00 ff b0 08 00 00 64
```

```
88  stack00   [0800ffa0]: 00 00 00 0a 00 37 5f 00 08 00 ff c0 08 00 00 28
```

C.4 例題プログラム fact.asm

```
# fact.asm --- calculate 10!(factorial) with recursive func. call.
# by K.Nagami

# These 4 lines are only for the definition of memory map.
        .data 0x00000000 = " "[0]
sys:    .space 0
        .data 0x08000000 = " "[0]
user:   .space 0
        .data 0xffff0000 = " "[0]
io:     .space 0

#--- I/O port ---
        .data io + 0x0000ff00 = " "[0]
        .align 0
IOcnt:  .space 0                        # valid I/O port area
Ccnt:   .space 0                        # clock counter port
        .space 4
Rcnt:   .space 0                        # read access counter port
        .space 4
Wcnt:   .space 0                        # write access counter port
        .space 4
RWcnt:  .space 0                        # access counter port
        .space 4

#--- Boot Segment - Execution starts here. ---
        .text sys = "/SYS/M00"[0]

        lhi $a0, (USR_CF >> 16)         #
        . $a0 |= (USR_CF & 0xffff)      # $a0 = USR_CF
        . call @config                  # configuration
        . nop                           # slot

        lhi $a0, (CPU_CF >> 16)         #
        . $a0 |= (CPU_CF & 0xffff)      # $a0 = CPU_CF
        . call @config                  # configuration
        . nop                           # slot

        lhi $t9, (@program >> 16)       #
        . $t9 |= (@program & 0xffff)    # $t9 = @program
        . goto $t9                      # goto @program
        . nop                           # slot


@config:
        # $a0: start address of configuration data.
        lhi $t9, (io >> 16)
        . $t9 |= (io & 0xffff)          # $t9 = io
        . $t8 = $0 - 1                  # $t8 = 0xffffffff
@loop:
        . $t0 = (%w)$a0[0x0]            # CONFIG_ADDRESS
        . $t1 = (%w)$a0[0x4]            # Write Data
        . $t2 = (%w)$a0[0x8]            # Write Mask
        . $t3 = (%w)$a0[0xc]            # End mark

        . $t9[0x0cf8] = $t0             # write to CONFIG_ADDRESS
        . $t4 = (%w)$t9[0x0cfc]         # read register value
        . $t1 &= $t2                    # mask write data
        . $t2 ^= $t8                    # negate $t2
        . $t4 &= $t2                    # mask read data
        . $t1 |= $t4                    # merge write data and read data
        . $t9[0x0cfc] = $t1             # write the value to register

        bnez $t3, @loop                 # loop back if continued
```

```
        . $a0 [+]= (@@C_END - @@C_ORG)  # valid slot (next entry)
        . return                        # else return
        . nop                           # slot

#--- Configuration Data ---
        .data sys + 0x10000 = "/SYS/M00"[0x10000]
        # For USR Memory Configuration
USR_CF: .space 0
@@C_ORG:.space 0
        #     enable  .......   bus#   dev.#  func.# reg.#
        .word (1<<31)|(0<<24)|(0<<16)|(1<<11)|(0<<8)|(0x04) # [status] [command]
        #                         +- fast back-to-back enable
        #                         |+- SERR# enable
        #                         ||+- wait cycle control
        #                         |||+- parity error response
        #                         ||||+- VGA palette snoop
        #                         |||||+- memory write and invalidate enable
        #                         ||||||+- special cycles
        #                         |||||||+- bus master
        #                         ||||||||+- memory space
        #       status      ......|||||||||+- I/O space
        .half 0x00000000, 0b0000000000000011 # value
        .half 0xffffffff, 0b0000001111111111 # mask
        .word 1 # continue
@@C_END:.space 0

        #     enable  .......   bus#   dev.#  func.# reg.#
        .word (1<<31)|(0<<24)|(0<<16)|(1<<11)|(0<<8)|(0x0c) # [] [] [] [cache line size]
        #                       size
        .byte 0x00, 0x00, 0x00, 0x04 # value
        .byte 0x00, 0x00, 0x00, 0xff # mask
        .word 1 # continue

        #     enable  .......   bus#   dev.#  func.# reg.#
        .word (1<<31)|(0<<24)|(0<<16)|(1<<11)|(0<<8)|(0x10) # [I/O base address]
        .word 0x0000ff00 # value
        .word 0xffffffff # mask
        .word 1 # continue

        #     enable  .......   bus#   dev.#  func.# reg.#
        .word (1<<31)|(0<<24)|(0<<16)|(1<<11)|(0<<8)|(0x14) # [Mem. base address]
        .word 0x08000000 # value
        .word 0xffffffff # mask
        .word 0 # end

        # For DLX CPU Configuration
CPU_CF: .space 0
        #     enable  .......   bus#   dev.#  func.# reg.#
        .word (1<<31)|(0<<24)|(0<<16)|(0<<11)|(0<<8)|(0x04) # [status] [command]
        #                         +- fast back-to-back enable
        #                         |+- SERR# enable
        #                         ||+- wait cycle control
        #                         |||+- parity error response
        #                         ||||+- VGA palette snoop
        #                         |||||+- memory write and invalidate enable
        #                         ||||||+- special cycles
        #                         |||||||+- bus master
        #                         ||||||||+- memory space
        #       status      ......|||||||||+- I/O space
        .half 0x00000000, 0b0000000101000100 # value
        .half 0x11111111, 0b0000001111111111 # mask
        .word 1 # continue

        #     enable  .......   bus#   dev.#  func.# reg.#
        .word (1<<31)|(0<<24)|(0<<16)|(0<<11)|(0<<8)|(0x0c) # [] [] [] [cache line size]
```

```
        #                       size
        .byte 0x00, 0x00, 0x00, 0x04 # value
        .byte 0x00, 0x00, 0x00, 0xff # mask
        .word 0 # end

#--- User Program ---
        #--- code ---
        .text user + 0 = "/USR/M00"[0]
@program:
        lhi $t9, (IOcnt >>16)           #
        . $t9 |= (IOcnt & 0xffff)       # $t9 = IOcnt

        lhi $gp, (DATA >> 16)           #
        . $gp |= (DATA & 0xffff)        # $gp = DATA

        lhi $sp, (ends >> 16)           #
        . $sp |= (ends & 0xffff)        # $sp = ends

        . $fp = $sp + (%s)stack00       # $fp = $sp + sizeof stack00


        . $t9[(%o)Ccnt] = $0            # reset clock counter I/O
        . $t9[(%o)RWcnt] = $0           # reset access counter I/O

        . call main                     # main()
        . nop                           # slot

        # Execution end
        . $t7 = (%w)$t9[(%o)Ccnt]       # get clock count,
        . $t8 = (%w)$t9[(%o)RWcnt]      # get access count,
        . $gp[(%o)TIME] = $t7           #
        . $gp[(%o)ACCESS] = $t8         # and store them

        . $t0 = 0xeeee                  #
        . $gp[(%o)END] = $t0            # *END = 0x00001234

inf:    . goto inf                      # infinite loop
        . nop                           # slot

main:
        # Callee Routine (head)
        . $sp -= (%s)stack00
        . $sp[0x0c] = $ra               # return address
        . $sp[0x08] = $fp               # frame pointer
        . $fp = $sp + (%s)stack00

        . $a0 = (%w)$gp[(%o)N]          #
        . call fact                     # call fact(N)
        . nop                           # slot
        . $gp[(%o)FACTN] = $v0          # *FACTN = fact(N)

        # Callee Routine (tail)
        . $ra = (%w)$sp[0x0c]           # return address
        . $fp = (%w)$sp[0x08]           # frame pointer
        . $sp += (%s)stack00            # pop stack frame
        . return
        . nop                           # slot

fact:
        # Callee Routine (head)
        . $sp -= (%s)stack00            # push stack frame
        . $sp[0x0c] = $ra               # return address
        . $sp[0x08] = $fp               # frame pointer
        . $fp = $sp + (%s)stack00

```

```
        . $fp[0] = $a0              # save argument
        . $t0 = $a0 > 1             #
        bnez $t0, @RECURSION        # if ($a0 > 1) goto L2
        . nop                       # slot
        . $v0 = 1                   # v0 = 1 (fact(1) = 1)
        . goto @TAIL
        . nop                       # slot
@RECURSION:
        . $a0 = (%w)$fp[0]
        . nop                       # load delay slot
        . --$a0                     #
        . call fact                 # fact(N - 1)
        . nop                       # slot
        . $v1 = (%w)$fp[0]          # $v1 = N
        . nop                       # load delay slot
        . $v0 *= $v1                # $v0 = fact(N - 1) * N
@TAIL:
        . $fp[0x04] = $v0           # save return value

        # Callee Routine (tail)
        . $ra = (%w)$sp[0x0c]       # return address
        . $fp = (%w)$sp[0x08]       # frame address
        . $sp += (%s)stack00        # pop stack frame
        . return
        . nop                       # slot

        #--- data ---
        .data user + 0x2200 = "/USR/M00"[0x2200]
DATA:   .space  0                   # DATA <- start address
N:      .word  10                   # the initial argument for fact()
FACTN:  .space  4                   # 10! (0x0037f500) will be stored
END:    .space  4                   # to be 0x0000eeee at the end
TIME:   .space  4                   # clock count
ACCESS: .space  4                   # ACCESS count

        #--- stack ---
        .data user + 0xff00 = "/USR/M00"[0xff00]
stack10:.space 16
stack09:.space 16
stack08:.space 16
stack07:.space 16
stack06:.space 16
stack05:.space 16
stack04:.space 16
stack03:.space 16
stack02:.space 16
stack01:.space 16
stack00:.space 16                   # These are stack frames
ends:   .space 0
#--- End of User Program ---
```