計算機構成
割り込みとDMA

天野　hunga@am.ics.keio.ac.jp

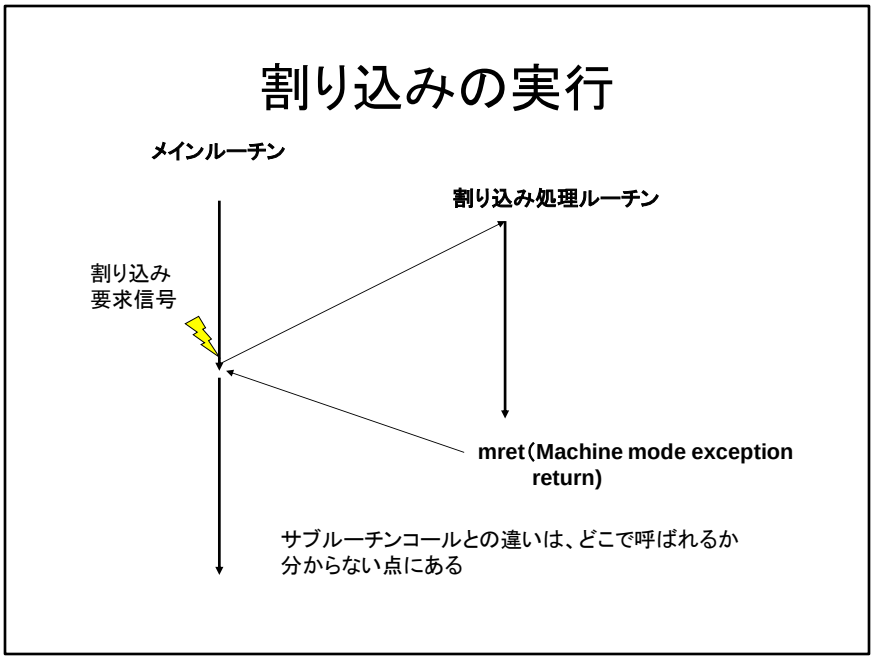
今回は入出力の続きです。

割り込み（Interrupt)とは？

- I/O側からCPUに対して割り込みを要求
- CPUはこれを受け付けると自動的にPCを割り込み処理ルーチンの先頭に変更
- 戻り番地はどこかに保存
- 割り込み処理ルーチンを実行、終了後リターン命令で元のルーチンに戻る

→ 元のルーチンは割り込みが起きたことがなかったかのように実行しなければならない

さて、今までのI/Oは、常にCPUの方からI/Oの状態を見に行かなければならないです。このため、I/Oを行うと、他の処理ができなくなってしまいます。逆に何か処理をはじめると、定期的にI/Oを見に行かねばならず、大変です。考えてみると、I／OがCPUに何か要求があっても、CPUが読みに行くまではそれを知らせる手段はありません。CPUの基本はとことん自分中心にできているのです。
これでは困るので、I/O側からCPUに対して要求を出す機構が生まれました。これが割り込み（Interrupt)です。I/OはCPUに対して割り込み要求を出します。CPUはこれを受け付けると、自動的にPCが割り込み処理ルーチンの先頭に変更されます。この際、戻り番地はどこかに保存しておきます。割り込み処理ルーチンを実行し、終了後リターン命令で元のルーチンに戻ります。この際、元のルーチンは割り込みが起きなかったかのように実行されなければならないです。



割り込みの実行は、サブルーチンコールと少し似ています。メインルーチンを実行中、割り込みを許可にする命令を実行した後、割り込み要求信号が入ると、割り込み処理ルーチンの先頭に飛びます。ここから始まる割り込み処理ルーチンを実行し、最後にmret（Machine mode exception return)命令を実行すると、メインルーチンに戻って、処理を再開します。動作は似ているのですが、サブルーチンコールと違って、どこで割り込みが掛かるかわからない点にあります。

割り込みの実現方法

- 割り込み処理のとび先番地
  - 固定番地　MIPSでは0x8000_0010
  - レジスタに設定　RISC-VではmtvecレジスタⅠ
    - 割り込み処理ルーチンの先頭でどのI/Oが要求を出したか調べる必要がある
    - ここでは0x80000000番地に初期化してある。
  - テーブル引き
    - I/O毎に飛び先を変えることができる
- 戻り番地の格納手法
  - 特殊なレジスタ、RISC-VではmepcⅠという特殊なレジスタに格納する。原因はmcauseレジスタに格納
  - ハードウェアスタックを持っているマシンはそれを使う

割り込みが行ったときの飛び先はどのように設定すれば良いでしょうか？最も簡単な方法は固定番地にしておくことです。MIPSではこの方法をとり、0x80000010番地に飛ばすことにしています。一方、RISC-VではmtvecレジスタⅠに飛び先番地を設定しておき、それを使います。この方法では、だれが割り込みを掛けたかわからないので、割り込み処理ルーチンの先頭でどのI/Oが出したのかを知るためにステータスレジスタを調べる必要があります。信号処理用プロセッサやマイクロコントローラの中には、割り込み要求信号によって、違った飛び先に飛べるようにしてあるもの、飛び先をテーブル引きにして自由に選べるようにしているものもあります。I/Oが少なければ、それぞれの処理ルーチンの先頭に直接飛んでいくことができます。では戻り番地はどこにしまっておけばよいでしょうか？RISCではこのための特殊なレジスタを持たせており、RISC-VではmepcⅠ(Machine mode exception pc)に戻り番地をしまっておきます。割り込みが掛かった原因もmcauseレジスタという専用レジスタにしまいます。サブルーチンコールにハードウェアスタックを用いるCPUでは、これを戻り番地の格納にも使います。

割り込みからの復帰

- mret: Machine-mode Exception Return

  0011000 00010 00000 000 00000 1110011

  pc ← mepcにし、割り込みを許可する

  (mie:machine-mode interrupt enable)をセットして割りこみを有効にする
- 例外レジスタのアクセスには、csrs(Control and Status Register Read and Set)など用意されている

割り込みからの復帰はmret命令で行います。実は割り込みは例外処理という大きな枠組みの中の一つとして位置づけられています。例外レジスタのアクセスには各種命令が用意されているが、動作が複雑なのでここでの説明は省略します。

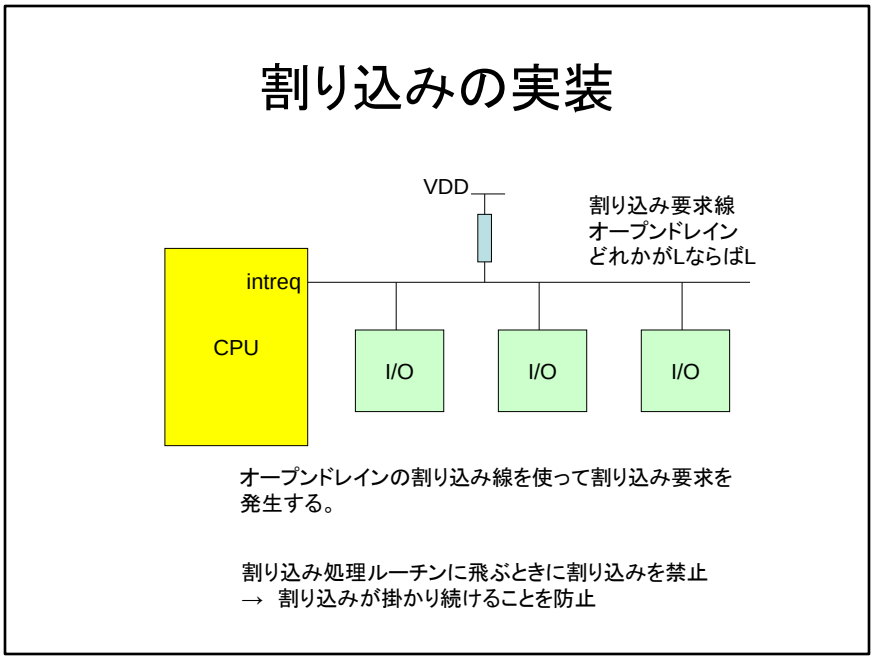
RISC-Vの割り込みの実際

- 入出力割込み＋例外処理（Exception)が16種類ある
  - アクセス・フォールト例外：アクセスタイプが合わない場合、ROMにストアした場合など
  - ブレークポイント例外：ebreak実行、またはデバッグのトリガにアドレス、データが一致した場合
  - 環境呼び出し例外：ecall命令を実行した時
  - 不正命令例外：無効なオプコードをデコードした時
  - 非整列化アドレス例外：ワード境界が合わない時
  - ページフォルト
  - 特権状態違反　など
- mcauseレジスタに原因が格納される

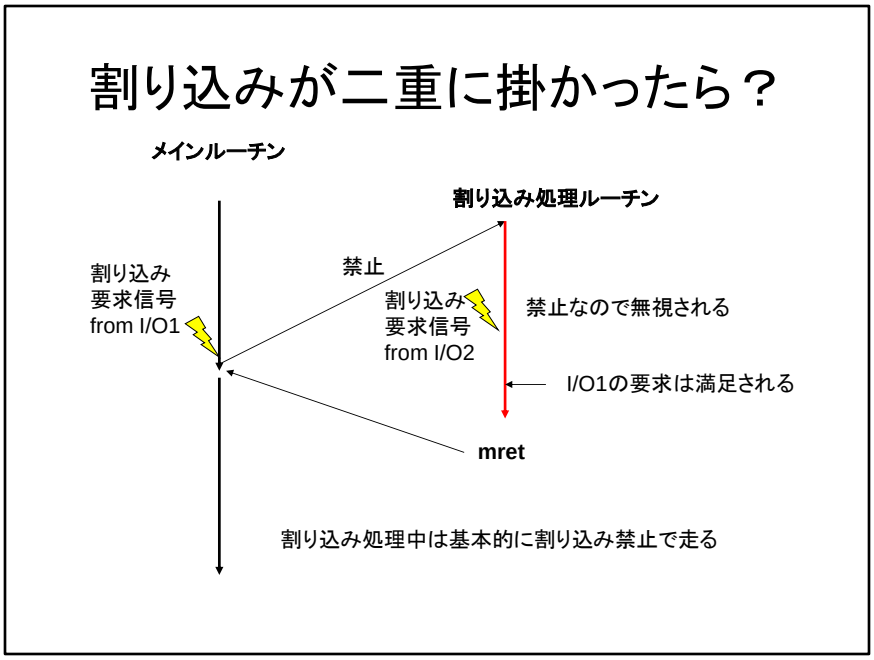
RISC-Vは、外部からの割込み、タイマー割り込み、ソフトウェア割込みなど各種割込みの他に16種類ある例外処理が一緒にまとめられています。例外の中では、先に紹介したページフォルトも含まれています。どのような割込みが起きたかはmcauseを見て判別します。

例題

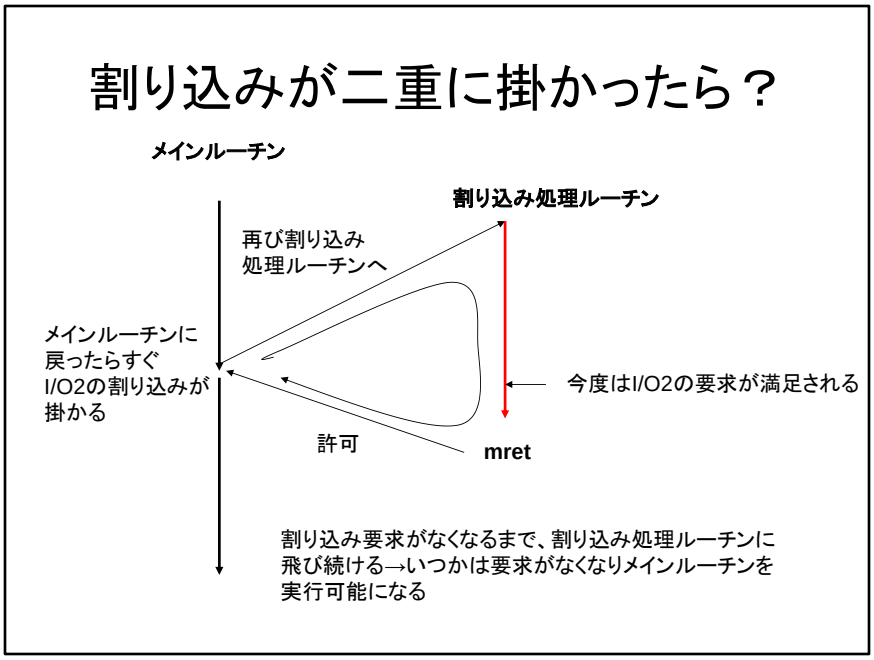
- フィボナッチ数列を計算（fibo.asm)しながら、入出力の処理（test_int.asm)を行う割込みプログラムを実行する。
- make test → iverilog実行
- make fibo → フィボナッチ数列（メインプログラム）のアセンブル
- make test_int →入出力処理（割込みプログラム）のアセンブル
- ./test | moreで実行の様子を観測しよう



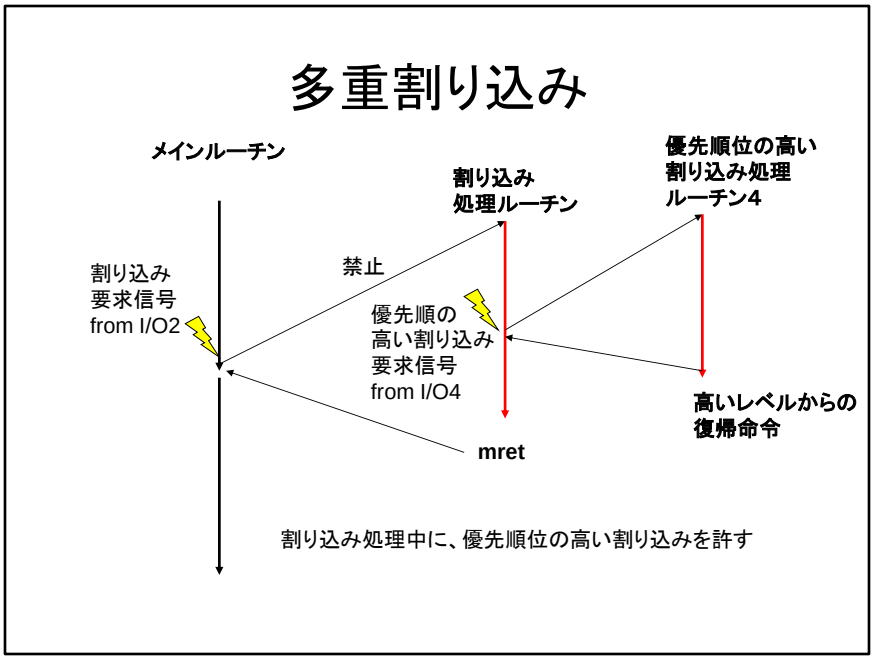
割り込みを実装するために、CPUは割り込み要求入力（ここではintreq）を持っています。割り込み要求線はオープンドレインといってどれかがLならば全体がLレベルになる信号線を使います。I/Oのどれか（複数でも）が要求を出すと、intreq=Lとなって割り込みが掛かります。割り込みが掛かり続けてCPUの処理が先に進まなくなることを防ぐため、割り込み処理ルーチンに飛ぶと同時に割り込みは禁止なり、割り込み処理ルーチンは禁止のままで走ります。割り込み処理ルーチンないで別の割り込みを受け付けるのを多重割り込みと呼びます。



一般的に、割り込み処理ルーチンに飛ぶと同時に割り込みは禁止となります。そうでないと、割り込みが掛かり続けて先に処理が進まなくなるためです。基本的に割り込み処理ルーチンは禁止状態で走って、そのどこかでI/Oの要求は満足されて、割り込み要求はリセットされます。mret命令でメインルーチンに戻ったときには、すでに要求は出ていないので、CPUはメインルーチンの先を続けることができます。



では、割り込み処理中に別の割り込み要求が掛かったらどうなるでしょうか？通常は、割り込み処理ルーチンは最初の割り込みの処理が終わったら、メインルーチンに戻ります。しかし、別の割り込み要求が掛かり続けているので、メインルーチンに戻って割り込み許可になった瞬間に再び割り込み処理ルーチンに飛ばされます。割り込み処理ルーチン中で、次の割り込み要求に対する処理を行い、再びメインルーチンに戻ります。いくつ割り込みが同時に掛かっても、この動作を繰り返して順番に処理していけば、そのうち要求がなくなりメインルーチンが実行可能になります。



しかし、割り込み処理プログラムが長時間にわたると、割り込みが長期間受け付けられない可能性が出てきます。I/Oの処理を頻繁に扱うプロセッサの中にはこれでは困る場合も生じます。そこで、割り込み処理中に割り込みを許す機構を装備しているプロセッサもあらわれました。この方法では、割り込みに優先順位を設けます。メインルーチンの優先順位は0とし、自分より高い優先順位の割り込みを受け付け、割り込み処理プログラムに飛んだ瞬間に割り込みレベルを上げてやります。この例では割り込みレベル2の要求を実行中に割り込みレベル4の要求があった場合を示します。この場合、優先順位が高いので割り込みが掛かってレベル4の割り込み処理プログラムに飛んでいきます。この時自分より低い（例えばレベル１）の割り込みが掛かっても応答しません。高いレベルの割り込みが終了したら元のレベルに戻ります。この方式ではレベルを管理するスタックと、それぞれのレベルでの戻り番地を管理するレジスタが必要になります。

多重割り込みは必要か？

- 複雑になるため、多くのCPUでは装備していない
- 割り込み処理時間が長いと緊急事態に対処できない
  - 多くのCPUでは、通常の割り込みと、禁止できないノンマスカブル割り込みで対処
  - PC退避用の専用レジスタ、リターン用専用命令が必要
  - 割り込み処理中でもNMIは受け付ける
  - NMIは緊急事態のみ
  - RISC-Vではマシンモードとユーザーモードを持つが、多重割込みをサポートしない

多くのCPUでは多重割り込み機構を持ちません。このため、割り込み処理の時間が長いと、この方法では緊急事態に対処することができません。そこで、多くのCPUでは、通常の割り込みと、禁止できない優先順位の高い割り込み（ノンマスカブル割り込み）の2種類を持っています。ノンマスカブル割り込み（NMI)は、割り込み処理実行中でもその要求が受け付けられ、別の番地に飛びます。この機構を実現するにはNMI専用のPC退避用レジスタ、リターン命令が必要になります。RISC-Vの場合は、基本的には多重割込みはサポートしていません。しかし、割込みのモードが、マシンモードとユーザーモードが分かれており、割込みという優先順位の高い割込みを持ちます。

割り込み対応用verilogコード

```
module rv32i(
input clk, rst_n,
input intrq,
… );
….
reg mstatus ;
reg [`DATA_W-1:0] mepc, mtvec;
…..
assign instrt = (intrq & mstatus) ? `NOP : instr;
```

要求intrq
mstatusは1で許可
mepcはpc保存用
mtvecは飛び先保持

割り込みが掛かったらそのサイクルは命令を実行しない

割り込みの実装はパイプライン化を行うと頭痛の種になりますが、現在の状況では比較的簡単です。入力信号として割り込み要求intrqを加え、状態として割り込み許可かどうかを示すintenを設けます。また、PC保存用にepcを設けます。割り込みが掛かったら、そのサイクルは実装せず、NOPとし、次のPCから命令をフェッチさせます。

pc周辺

```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if(intrq & mstatus) pc <= mtvec;
  else if(mret_op) pc <= mepc;
  else if ( jal_op) | ….
```

要求があり許可されていればpcは割り込みプログラムの先頭番地へ

mret割り込み復帰命令でmepcから復帰

PC周辺は、要求があって許可されていれば、PCは割り込みプログラムの先頭番地(mtvecの中の番地)に飛ぶようにします。さらにmret命令によっていmtvecの中身をPCに戻すようにします。

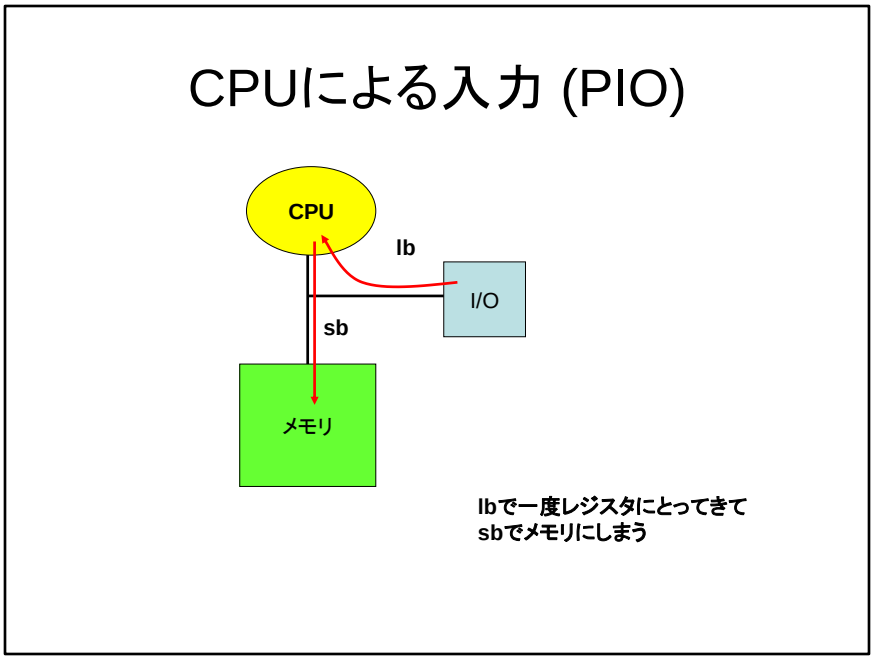
割り込み許可フラグとmepc

```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) mstatus <= 1;
  else if(intrq & mstatus) mstatus <= 0;
  else if(mret_op ) mstatus <= 1;
end
always @(posedge clk or negedge rst_n)
  if(!rst_n) mepc <= 0;
  else if(intrq & mstatus) mepc <= pc;
```

割り込み許可フラグの制御

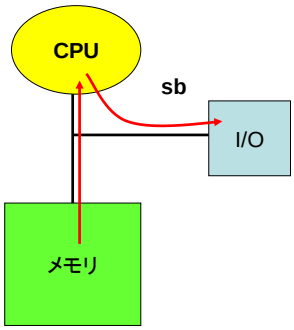
割り込みが掛かるとそのときのpcがmepcに保存される

割り込み許可フラグと、mepcレジスタの制御用のコードを加えます。ここではmstatusは1ビットしか実装していませんが、本来、モードに依存した情報を多数持っています。



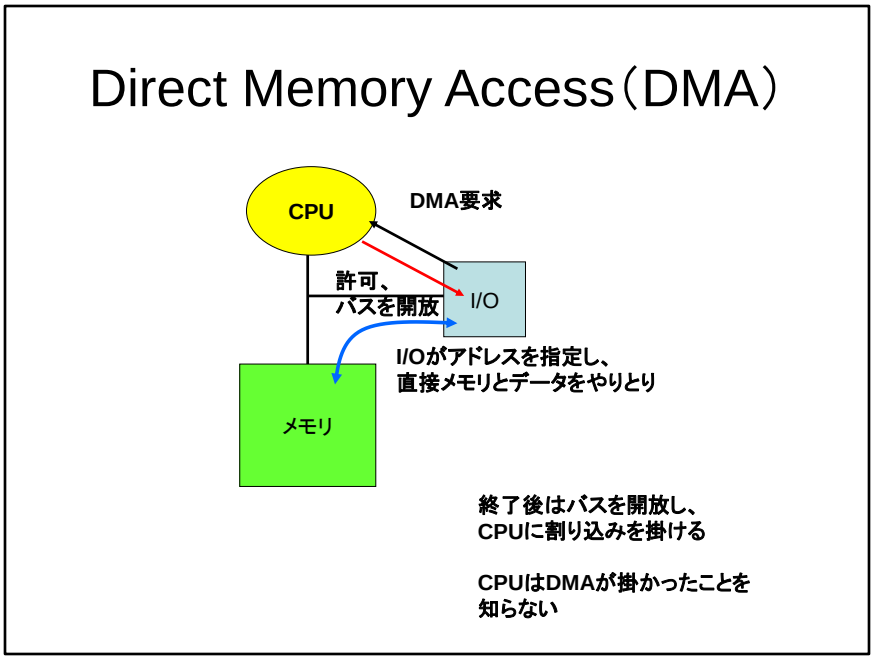
では最後にI/Oに関するもう一つの問題点を取り上げます。I/Oから入力する場合、lbで一度CPUのレジスタに取ってきて、sbでメモリにしまいます。これは2回コピー作業を行っていることに相当します。このように完全にプログラムで制御するIOをPIOと呼ぶことがあります。

CPUによる出力(PIO)



lbで一度レジスタにとってきて
sbでI/Oのデータレジスタへ出力

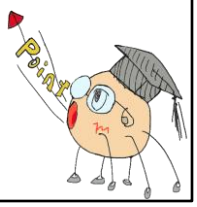
逆にI/Oから出力する場合はどうでしょう？lbでメモリからCPUのレジスタにデータを取ってきて、そこからI/Oにデータを出力します。2回のコピーが必要です。これは、CPUが常にメモリとI/Oを結ぶバスの利用権を握り続けているためです。



DMA(Direct Memory Access）は、この問題を解決するための方法です。まずI/OからDMA要求を出します。CPUは受付可能であれば、DMAを許可し、バスの利用権を開放します。I/OはCPUに代わってバスを使って、直接メモリとの間でデータを交換します。DMAは一度にデータを転送するブロック転送（バースト転送）に向いていて、大規模なデータを高速に転送するのに使われます。終了後はバスを開放し、CPUに利用権を返します。CPUのプログラムはDMAが掛かったことを知らないので、必要があれば割り込みを掛けて知らせてやる必要があります。

本日のまとめ

- 割り込みはI/Oから要求してCPUの動きを変える。
  - プログラムは割り込み処理ルーチンに自動的に飛んでいき、リターン命令で戻ってくる
  - メインルーチンは、あたかも割り込みがなかったかのように実行を続ける。
- DMAはI/Oとメモリ間で直接データをやりとりする。



インフォ丸が教えてくれる今日のまとめです。今回はいろいろな言葉が出たので意味は理解しましょう。

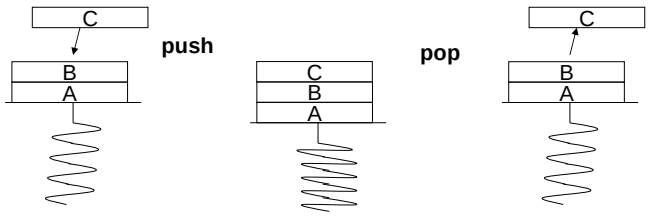
演習1

- 割り込み処理ルーチンではメインルーチンで利用するレジスタを破壊しないために利用する全てのレジスタをスタックに保存しなければならない。
  - fibo.asmではフィボナッチ数列を計算している
  - ex_int.asmでは入出力を行う割込みプログラムでtest_int.asmと同じだがfibo.asmとレジスタが重なるため割込みが掛かるとfiboで使っているレジスタが破壊される
  - この両者がきちんと動作するために、ex_int.asmにレジスタ保存と復帰のコードを付け加えよ
  - スタックポインタはx2で、fibo.asm内で設定済

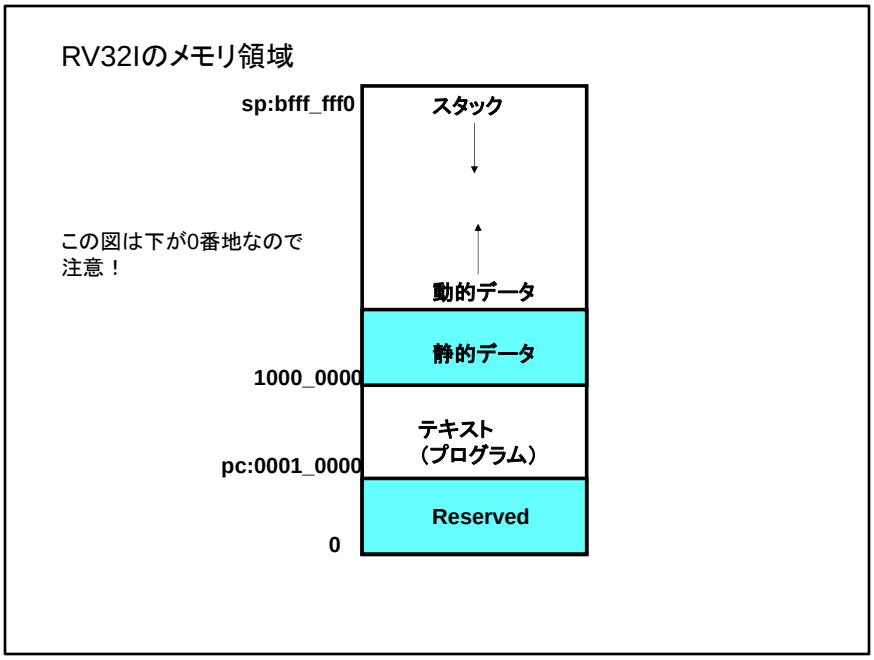
以下のページは、ヒントです。

スタック

データを積む棚
　push操作でデータを積み
　pop操作で取り出す
- LIFO(Last　In Fi$st Out)、FILO(Fi$st In Last Out)とも呼ばれる
- 演算スタックとは違う（誤解しないで！）
- 主記憶上にスタック領域が確保される

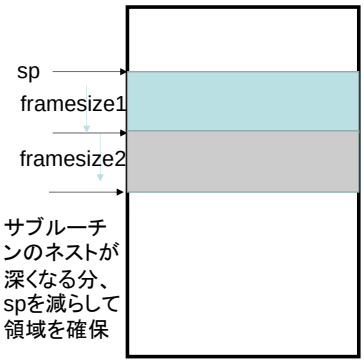


スタックとは、データを積む棚です。この棚にデータを積む操作をpush、棚から取り出す操作をpopと呼びます。先に積んだものが後から取り出されることからLIFO (Last In Fi$st Out)と呼びます。逆に考えると、後に積んだものが先に取り出されるのでFILO（Fi$st In Last Out)と呼ぶ場合もあります。この積んだ逆順に取り出すことのできる性質からサブルーチンコール時にレジスタを退避するのに適しています。
以前紹介したスタックマシンで利用した演算スタックは、演算用の特殊なメモリですが、サブルーチンコールのレジスタの退避用のスタックは主記憶上に確保するのが普通です。スタックは棚ですが、ばねがついているイメージがあります。データを積むときは押し込むイメージからpushと呼び、取り出すときは、飛び出すイメージからpopと呼ばれます。



RV32Iのメモリ領域を示します。この図は下が0番地で上に行くほど番地が増えるのでご注意ください。プログラムは0001_0000から貼り付けます。PCもリセットすると0001_0000になるように設定します。（ここでは0番地にしてしまいます。）1000_0000からは静的なデータを割り付け、スタックはbfff_ffffから番地の下の方に伸ばしていきます。（ここでは1000番地に設定してあります）

RV32Iにおけるスタックの実現




```
addi  sp,sp, -framesize
sw ra, sp,framesize-4
sw x18,sp,framesize-8
sw x19,sp,framesize-12

….サブルーチン本体

lw  ra,sp,framesize-4
lw x18,sp,framesize-8
lw x19,sp,framesize-12
addi sp,sp, framesize
jr ra
```

実際にスタックを作る場合の方法を示します。サブルーチンに入った時に、まずそのサブルーチンで必要なレジスタを数からフレームのサイズを計算し、その分スタックポインタを減らします。そしてその領域内にレジスタを格納して行きます。ここではディスプレースメントが威力を発揮します。サブルーチンの実行が終わったら、レジスタを復帰し、スタックポインタをフレームサイズ分だけ加えます。最後にjr (jalr)でメインルーチンに戻ります。この方法は、サブルーチン内で別のサブルーチンを呼ぶとその分spが下に下がり、リターンすると上がります。

実際に数値を入れた例

```
addi  x2,x2, -16   // framesize=16
sw x1, x2,12
sw x18,x2,8
sw x19,x2,4

….サブルーチン本体

lw  x1,x2,12
lw x18,x2,8
lw x19,x2,4
addi x2,x2,16
jr x1

x1: ra戻り値   x2: spスタックポインタ
jr x1はjalr x1,x0,0 (疑似命令）
```

実際に数値を入れた例を示します。スタックポインタはx2,戻り番地のレジスタにはx1を使います。ここではx18、x19、x1を保存/復帰しています。