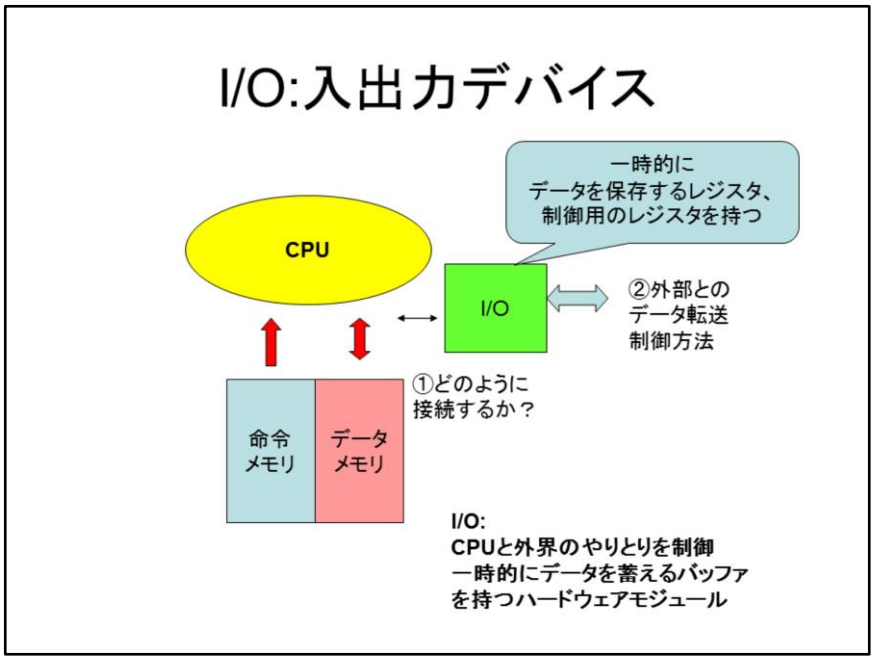
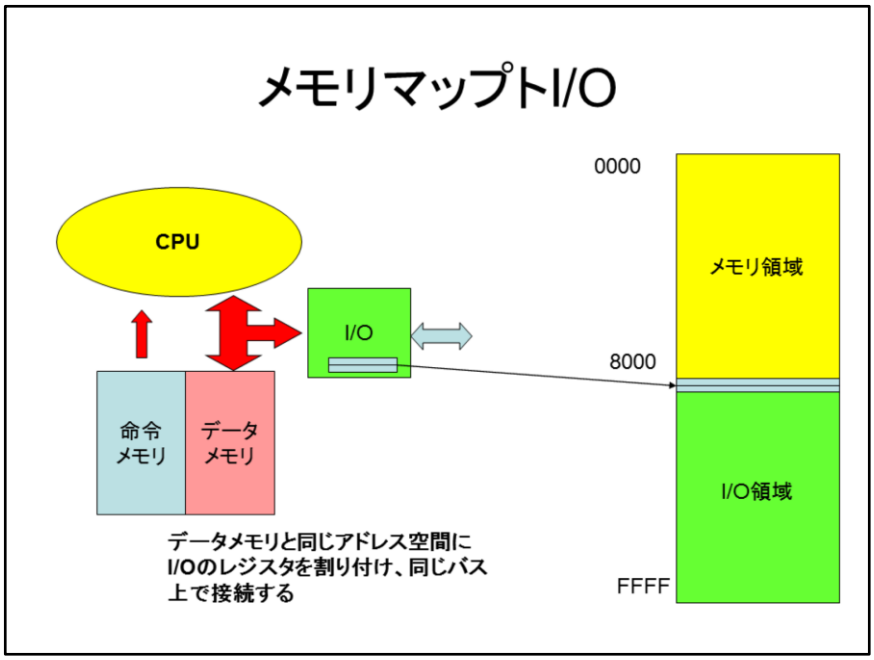
コンピュータアーキテクチャ
第5回　入出力

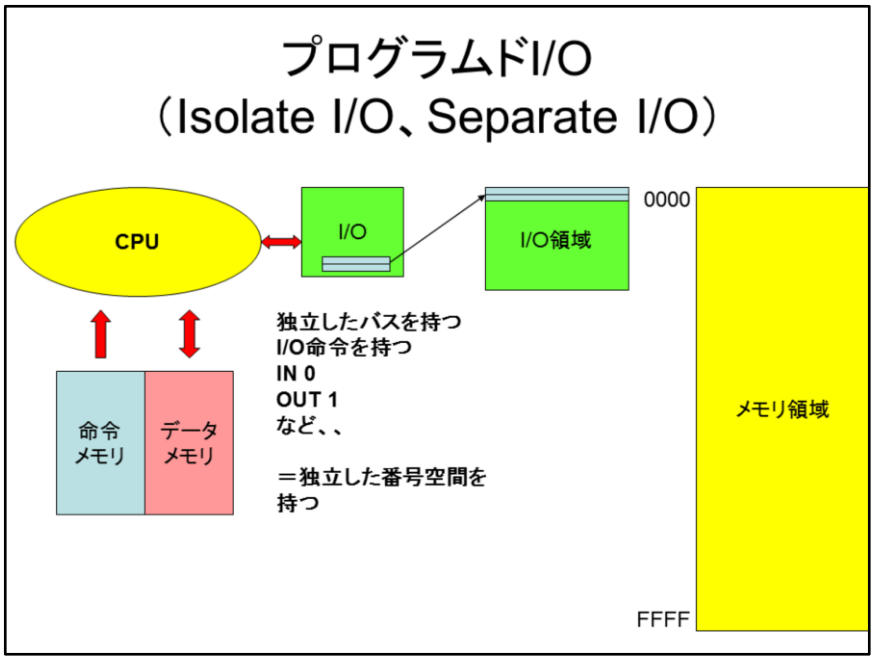
天野　hunga@am.ics.keio.ac.jp



CPUが外部とデータをやり取りするための装置をI/Oと呼びます。データをやりとりするため、一時的にデータを蓄えておくレジスタを持っています。これをバッファと呼ぶ場合があります。I/Oは繋ぐ対象によって動作が様々なので授業で扱うのが難しいです。しかし、どのI／Oも
①まずCPUと接続しなければならず、②外部とデータ転送を行わなければならないです。なので、この2点について押さえておこうと思います。後はあなたの扱うI／O毎に個別の勉強するしかないです。



まずCPUからI／Oを扱う方法としてメモリマップトI／OとプログラムドI／Oがあります。メモリマップトI／Oでは、データメモリと同じアドレス空間にI／Oのレジスタを割り当て、メモリを読み書きするのと同じLD,ST命令を使ってレジスタを読み書きします。どんなCPUにでも接続でき、I／Oのために特殊な命令やバスを設ける必要がないという利点がある一方、I／Oのレジスタの領域は比較的小さいので、使われない領域が無駄になりやすい点が問題です。

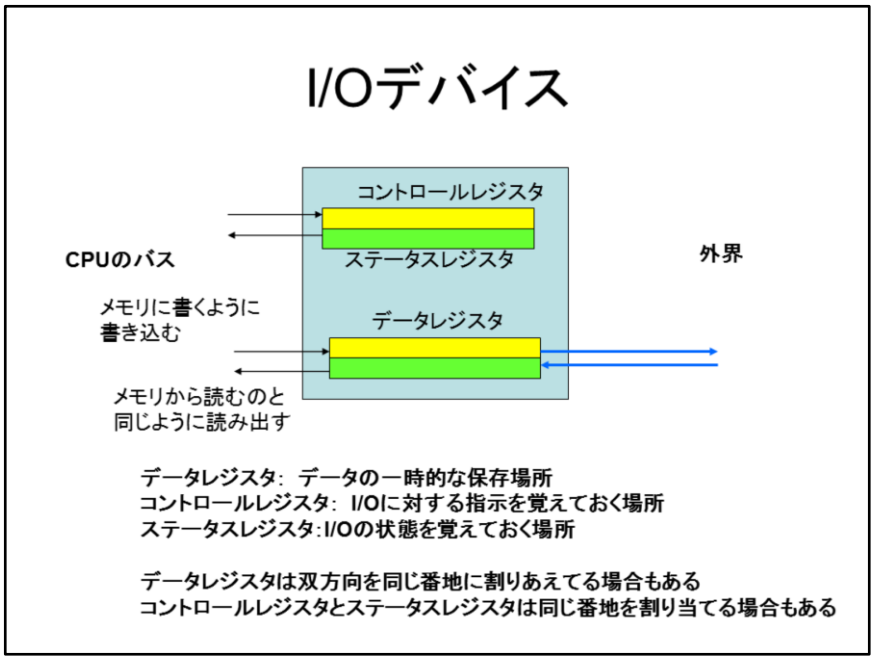


これに対してプログラムドI／Oは、IN命令、OUT命令など専用の命令でI／Oに読み書きをします。IN 0, OUT 1などI／O番号を指定しますが、これはすなわちメモリから独立したアドレス（番地）空間を持っているということに他ならないです。この方式は、バスもメモリと独立しているIsolate I/Oとバスは共通で番地空間だけ分けるSeparate I/Oを分けて考える人も居ますが、あまりこだわらない人も居ます。そもそもプログラムドI／Oという名前もさほど一般的ではありません（でも正式にはこう言うらしいです）。この方法は、独立した番号の空間があるので、メモリ領域を無駄遣いしない利点がありますが、専用命令が必要です。ちなみにバスが独立しているものは、メモリのアクセスとI/Oのアクセスを同時に行なうことができます。

メモリマップトI/O vs.プログラムドI/O

- メモリマップトI/O
  - I/Oとメモリを同一の命令、アドレス空間で扱える
  - 命令の種類が減る
  - ハードウェアが減る
  - どのような構造のCPUでも使える
- プログラムドI/O
  - メモリアドレス空間中にI/Oの虫食いができない
  - I/O中にメモリアクセスが可能
- Intel 86系はプログラムドI/O用の命令を持っていても実際にはメモリマップトI/Oで動いている

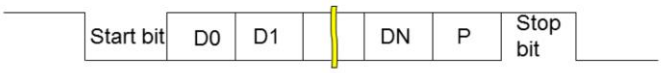
両者の特徴をまとめてみましょう。どのようなCPUでも使えるメモリマップトI/Oが最近は標準的に使われます。個別的なバスは標準バスに対応できないためです。このためIntel 86系のCPUはプログラムドI／O用の命令を持っていても実際にはメモリマップトI／Oで動いている場合がほとんどです。プログラムドI／Oは主として信号処理用プロセッサDSPや制御用のマイクロコントローラで用いられ、複数のバスを使ってI／O間で高速なデータ転送を行えるものもあります。



I／Oデバイスは、データレジスタ、コントロールレジスタ、ステータスレジスタの三種類のレジスタを持っています。データレジスタはCPUが読み書きする場合、一時的にデータを保存しておく場所です。入力用のデータレジスタと出力用のデータレジスタは同じ番地に割り当てられており、書き込むと出力され、読み出すと入力される場合もあります。ステータスレジスタは、I／Oの状態を覚えておく場所で、データが到着したり、データを送り終わったりしたことを示す情報を持っています。基本的には読み出し専用です。コントロールレジスタはI／Oに対する指示を記憶する場所です。Ethernetコントローラなどの動作が複雑なI／Oでは、多数のコマンドレジスタを持っています。基本的には書き込み専用です。このため、ステータスレジスタと同じ番地を割り当て、読むとステータスレジスタ、書くとコマンドレジスタとして使われる場合もあります。

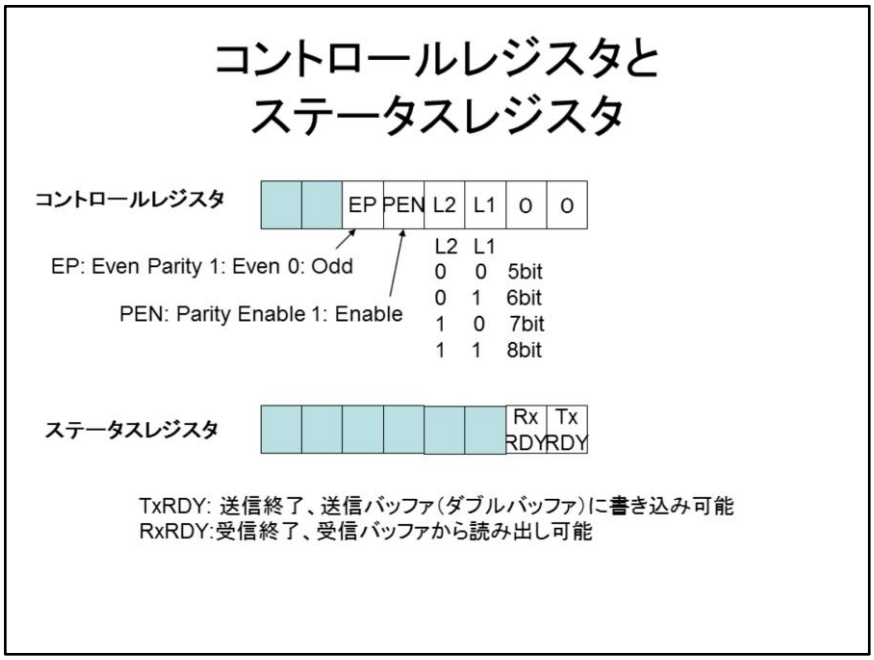
古典的なI／OとしてUART 8251を紹介しましょう。このI／Oは、パラレル／シリアル変換用で、CPUからのデータを直列に、すなわち時間的に順番に出力し、直列に入力されたデータをCPUの並列データとして受け取ります。5bit-8bit単位のデータを10Mbps(bit per second)という遅い転送レートで送ります。昔、データを音に変換して電話で交信するモデムという装置が使われたときの規格であるRS232C用に作られました。RS232Cは、モデムがなくなった後も簡単な端末用のインタフェースとして使われ、今でもその簡便さから遅くも良いインタフェースとして用いられます。8251は今でもFPGA内でIP(Intellectual Property: 出来合いの回路）として使われます。

シリアル転送フォーマット

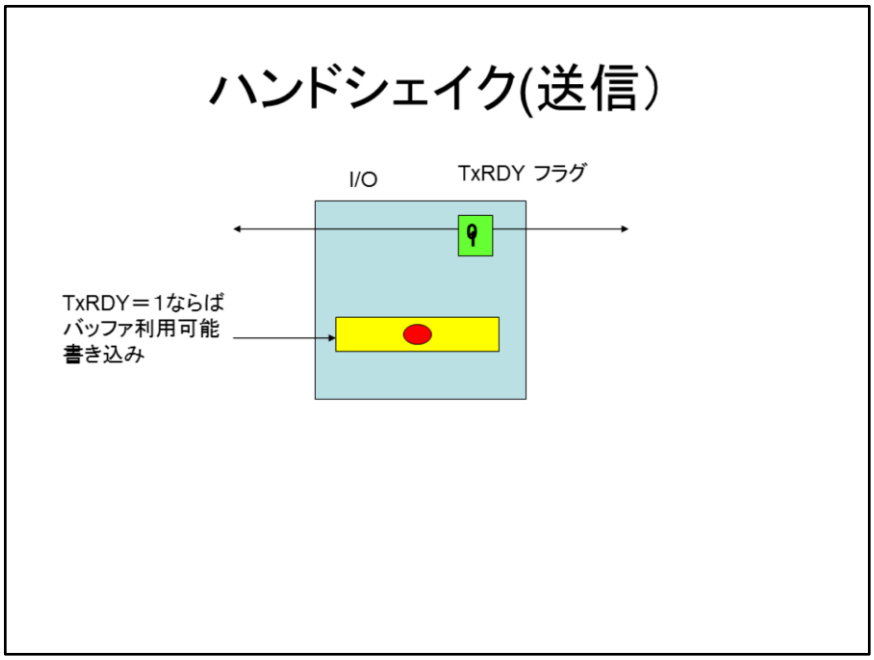


Start bit: 0にして開始を示す
D0-DN: データ、5ビットから8ビットまで選択可能
P:パリティビット 偶数/奇数の選択可能
Stop bit: 1にして終了を示す。長さを選択可能

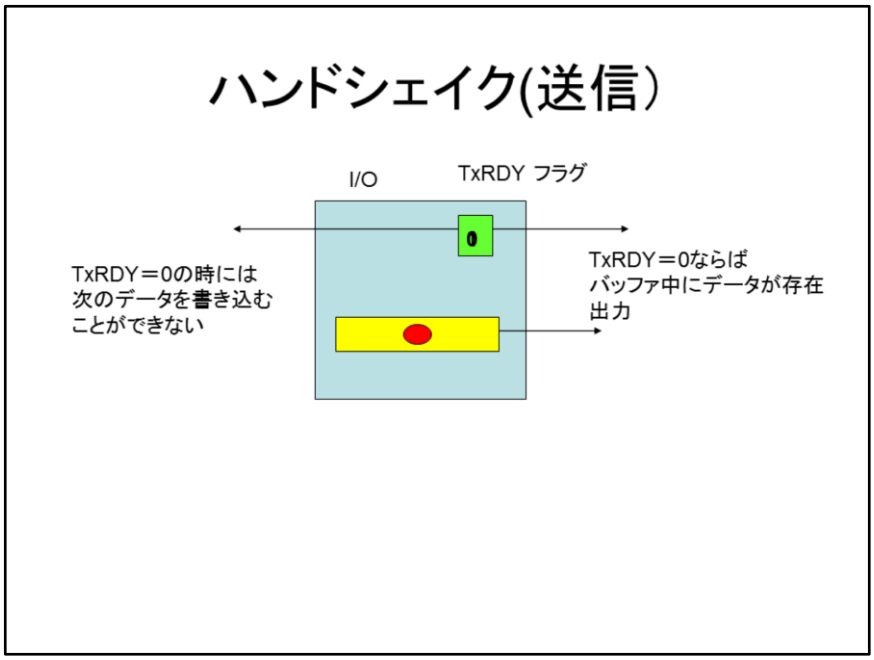
シリアル転送はまずStart bitとして一定の時間0を送ります。次に順に5ビットから8ビットのデータを転送し、最後にパリティを送った後にストップビットを一定の時間1にして送ります。パリティは偶数パリティ、奇数パリティを選択可能で、ストップビットも長さを選択可能です。



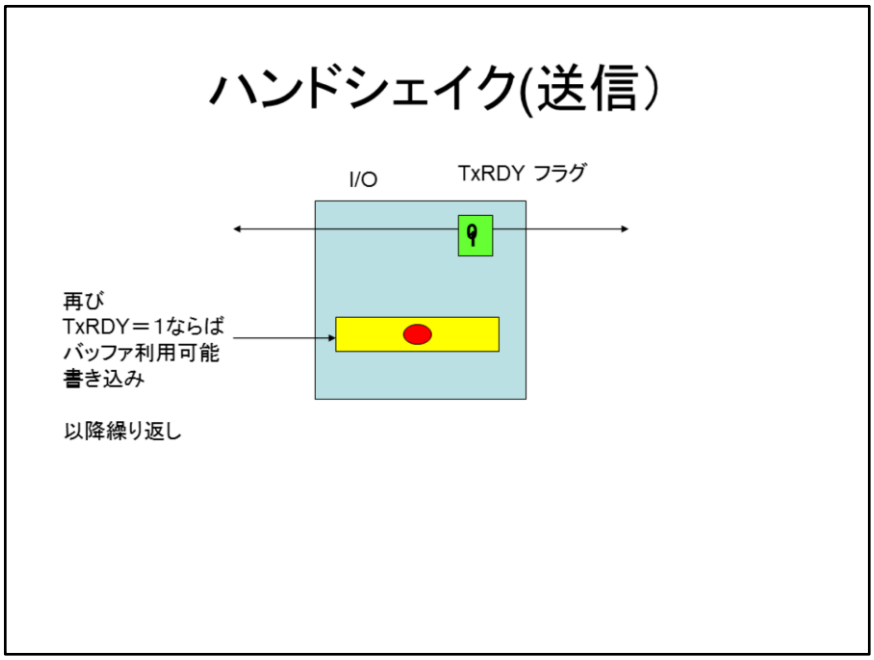
この8251のコントロールレジスタとステータスレジスタを紹介します。コントロールレジスタは、パリティの選択、データ長の選択を行ないます。パリティ(Parity)とは、1ビットの誤り検出符号で、データ内の1の個数を偶数または奇数にそろえることにより、1ビットエラーの検出を行ないます。偶数パリティを例に取って説明します。データ内の１の個数が偶数ならばParity bitを0とし、奇数ならばParity bitを1とします。Parity bitを含めた全体のデータの1の個数は常に1になるので、1の数が奇数のデータを受け取ったら誤りがあったことに気づくことができます。8251ではPEN=1でパリティを使う設定にし、EP=1ならば偶数パリティ、0ならば奇数パリティに設定できます。ステータスレジスタは、フラグを含みます。ここでは0ビット目がTxRDYでここが1ならば、送信用のバッファが空いていて書き込み可能であることを示します。1bit目はRxRDYでここが1ならば、受信が終わって、受信バッファ内に有効なデータがあることを示します。これらはハンドシェイクに使います。



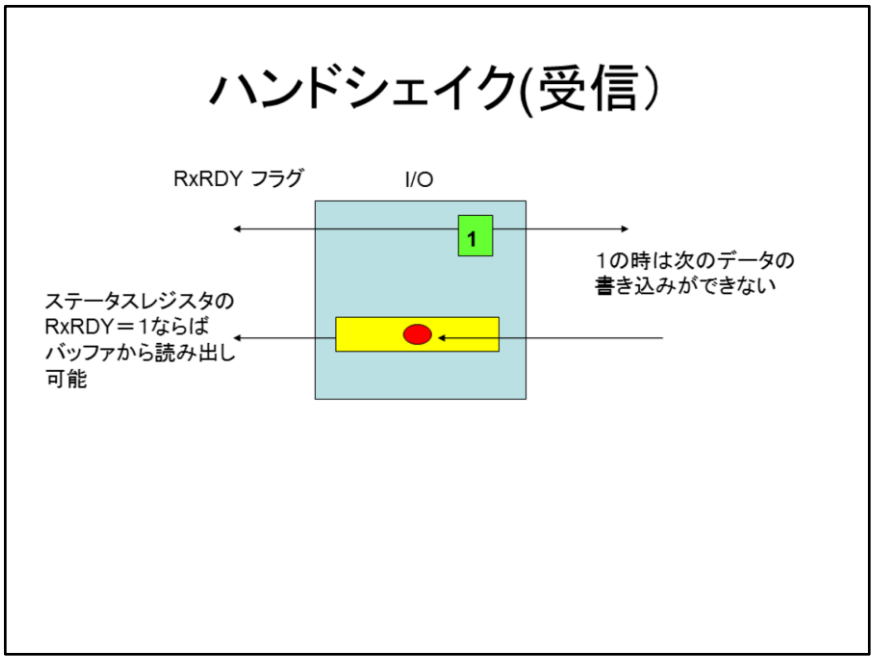
ハンドシェイクとは、送信側と受信側が同期を取って取りこぼしなくデータを転送するのに使われる方法です。送信を例に取って示しましょう。CPUがステータスレジスタを読んで、TxRDYが1ならば、バッファが空いているので、ここにデータを書き込みます。すると、TxRDYが0になります（正確にはダブルバッファなので動きが多少違うのですが、）。TxRDYのようにバッファの状態を示す1ビットの情報をフラグ（旗）と呼びます。分岐命令の条件を示すフラグと機能は同じです。



CPUはTxRDY=0の時は、バッファ内のデータはまだ転送が終わっていないので、待ち状態になります。外部にあるモデム装置（あるいは端末）は、TXRDY=0でデータが書き込まれたことが分かり、受信状態になりシリアルにデータが転送されます。



データの転送が終わると、TxRDY=1になります。CPUはこれを検出して次のデータを書き込みます。このようにして書き潰しを起こすことなく、データを転送することができます。このような同期操作をフラグを使ったハンドシェイク（握手）と呼びます。



受信の時はこの逆になります。外部のモデム装置はRxRDYが0であることを確認してバッファにデータを書き込みます。CPUはRxRDYが1になると、データが受信されたことが分かるので、バッファからデータを読み出します。この操作でフラグは0になりますので、モデム装置は次のデータを書き込みます。

フラグを使ってハザードを防ぐ

- RAWハザード：Read After Writeハザード
  - 値が書き込まれるのを待って読み込む
  - 二重読みを防ぐ
- WARハザード：Write After Readハザード
  - 値を読む前に書きこまれることがないようにする
  - 書き潰しを防ぐ
- WAWハザード：Write After Writeハザード
  - これも書き潰しだが、WARがなければ普通大丈夫

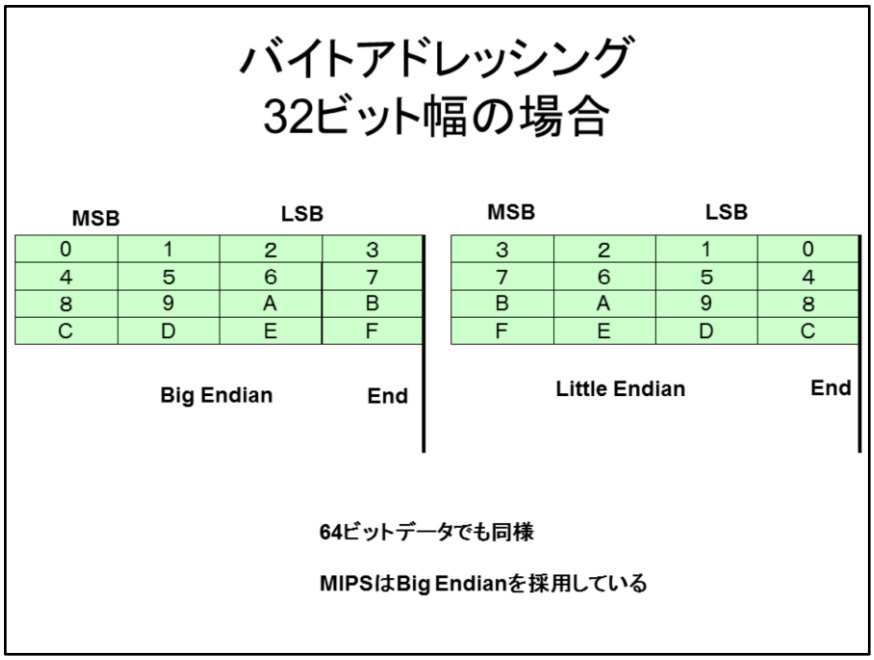
フラグを使うことで、読み出し、書き込み間の障害（ハザード）を防ぐことができます。一般的にハザードは3種類あります。Read After Write(RAW)ハザードは、ちゃんと値を書き込まれるのを待って読み出すことで、これがおきると同じデータを間違って複数回読んでしまう問題が起きます。一方、Write After Read (WAR)ハザードは、読む前に書いてしまう問題で、これは書き潰しを起こしてデータが消失してしまう問題です。Write After Write (WAW)ハザードは書き込む順番が狂う問題で、単純な入出力では考えなくてもいいです。これらのハザードは、パイプライン処理でも起こるのでその際にまた解説します。

メモリのアドレッシング

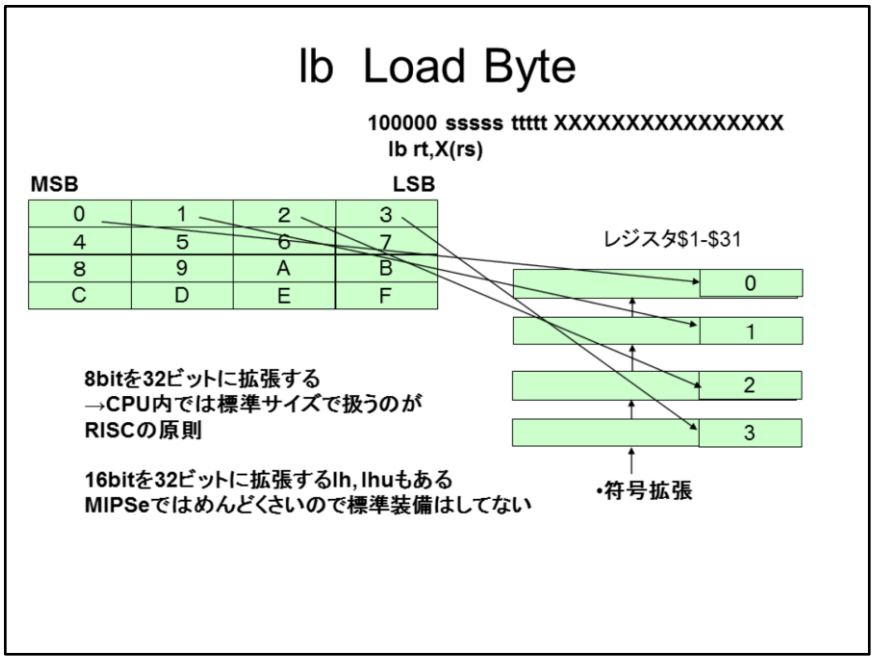
- I/Oデータは8ビット・1ビットなど短い幅が多い
- データ幅が一律32ビットでは効率が悪い
- ASCIIなど文字コードは短い幅が多い
  man asciiと打ち込んでASCIIコードを見てみよう

→ 通常のCPUは8ビット（バイト）単位にアドレスが振られている：バイトアドレッシング

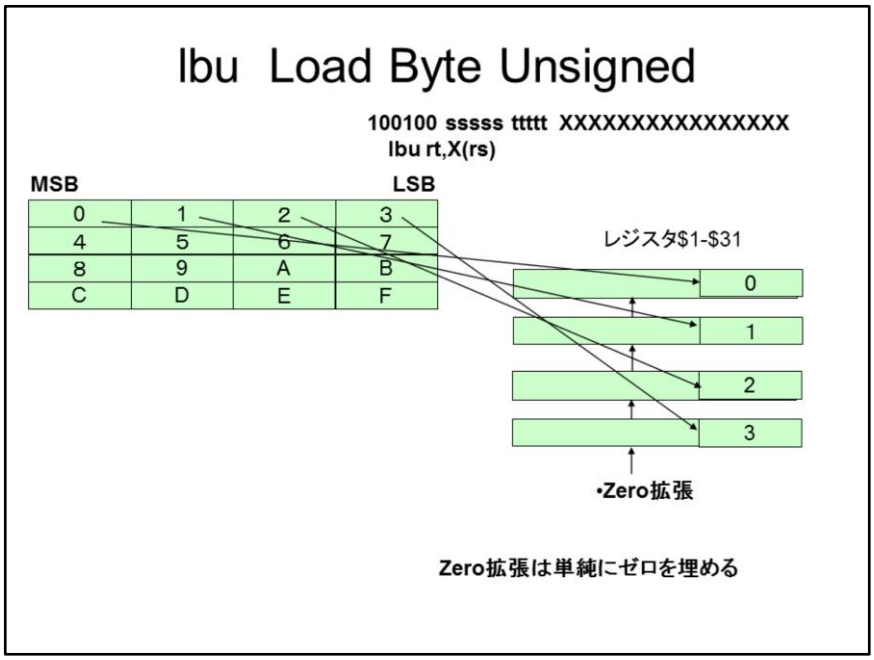
さて、ここまででI／Oについて紹介しましたが、いくつか問題があります。I／Oデータは8ビットのASCIIコードや数ビットのフラグが多いので32ビットデータのうちの一部しか使っていません。これはもったいないので、実は今のコンピュータは8ビット単位のバイトアドレッシングを用いています。これについて紹介します。次に、ビジーウェイト中CPUは無駄にループを回っていて馬鹿みたいです。この問題を解決するための手法として割り込みを紹介します。



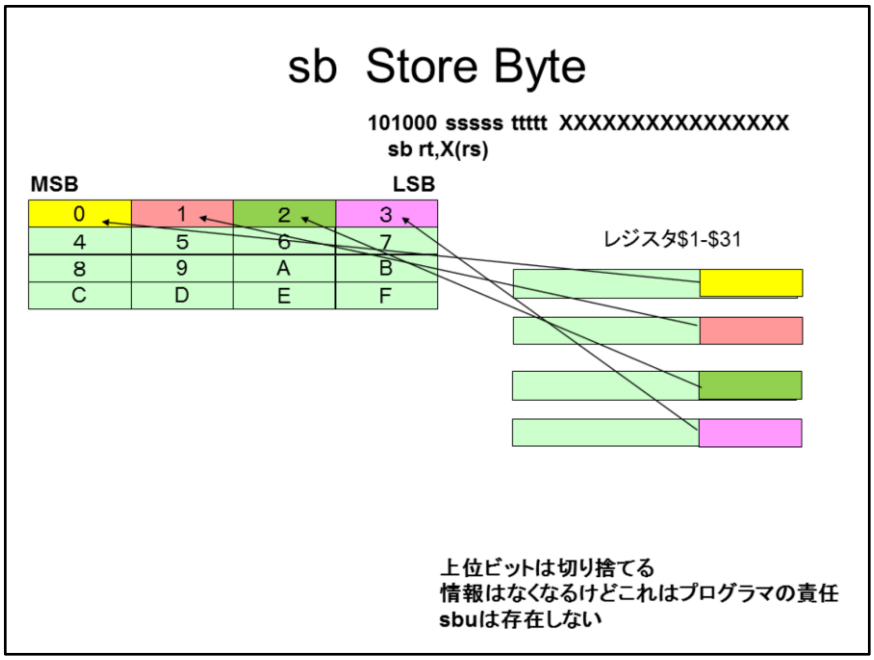
I／Oデータなどを扱う上での無駄を防ぐために、最近のCPUは32ビットではなく8ビット単位で番地が振られています。すなわち、32ビットは番地4つ分に当たります。ここで、桁の大きい方から0，1と振っていく方法と小さい方から0，1と振っていく方法の二つが考えられます。前者をBig Endian,後者をLittle Endianと呼びます。ここでは以降、Big Endianで番号を振ることにします。この振り方は統一が取れておらず、コンピュータ間でデータを交換する際にトラブルの元となります。最近のCPUは電源投入時の指定でどちらの方法を取ることもできるものが多いです。MSB側から0を振っていくと、大きい方で端（LSB: end)に達することからBig Endian, LSB側から0を振っていくと小さい方で端に達することからLittle Endianという名前になっていることが分かります。Endianとは奇妙な英語ですが、これはこの言葉が、ガリバー旅行記の卵の細い方から割る派(Little Endian)と太い方から割る派(Big Endian)で内乱が起きる話に由来するためです。



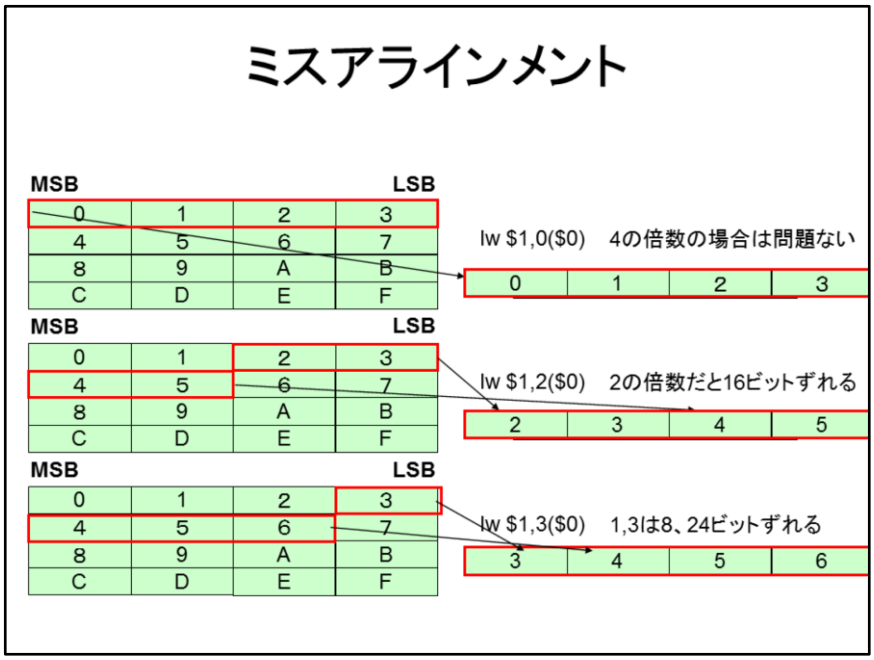
では、バイトデータを扱う命令を定義しましょう。lbは指定されたアドレスの8ビットを読み出して、レジスタの下位8bitに置きます。上位24bitは符号拡張されます。RISCではCPUの内部ではデータのサイズを統一してしまうのが普通です。この場合も読み出す際に32ビットに拡張します。lbはlwと同じR型で定義します。ちなみにMIPSでは16ビットデータを読み出して32ビットに拡張するlh(Load Halfword)もあります。最近の文字コードは16ビットが多いので、案外16ビットデータは利用価値が高いためです。ただし、ここでは実装が面倒なので省略してあります。



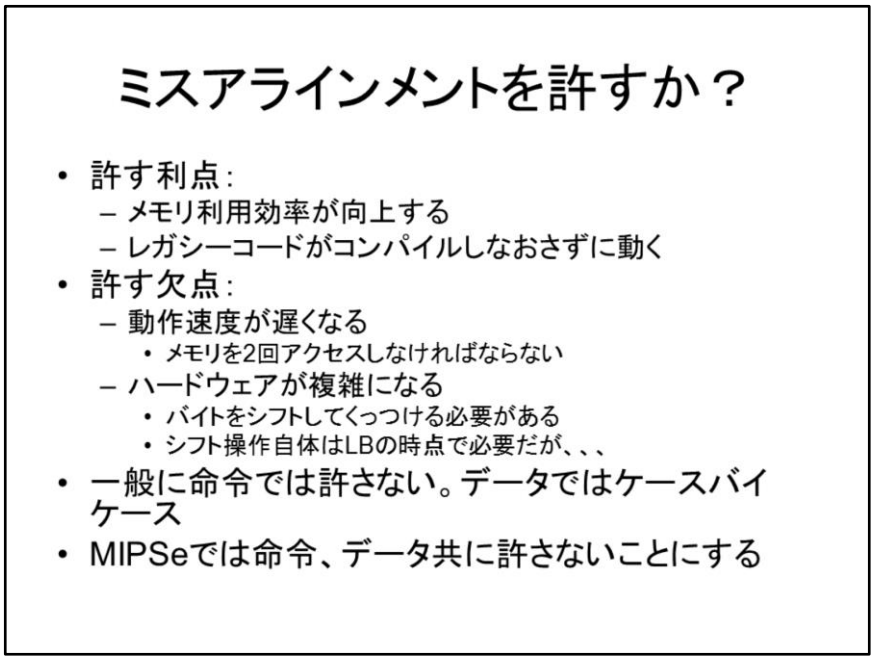
I／Oは符号無しのデータを扱うことも多いので、ゼロ拡張の8bit読み出し命令も用意しておくのが普通です。これがlbu(Load Byte Unsigned)で、上位24ビットには0が入ります。MIPSには、16ビットのLoad命令lhu(load half word unsigned)もありますが、ここでは省略します。



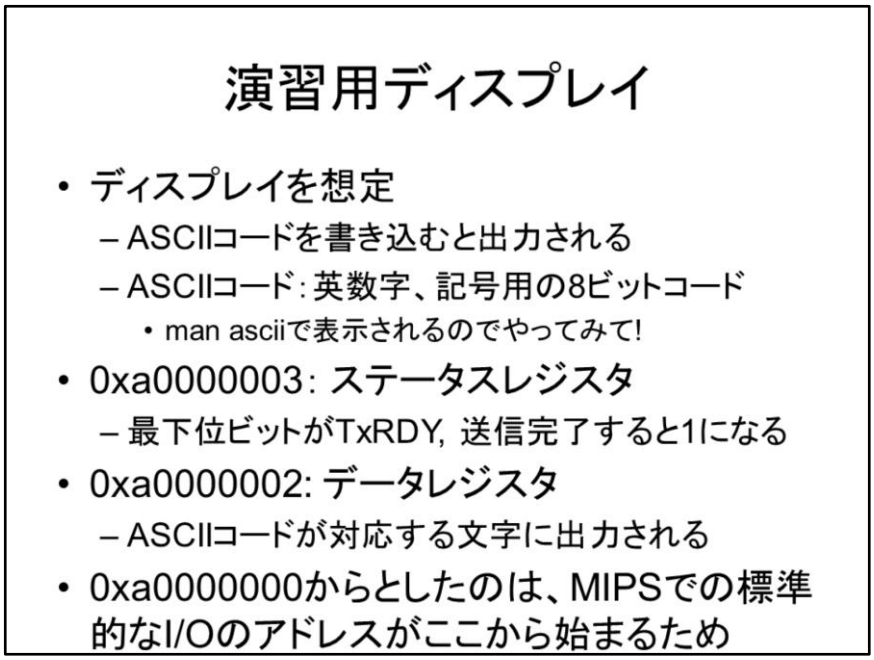
逆にレジスタ中の値を8ビット単位でメモリに書き込む命令がsb(Store Byte)です。この命令はレジスタの下位8ビットを指定されたメモリの番地に書き込みます。上位24ビットは無視されます。上位の情報がなくなっても困らないようにするのはプログラマの責任です。ちなみにsbはデータのサイズが減る方向なので、sbuとかを作る必要はありません。



バイトアドレッシングのメモリをlw,sw命令で扱った場合、4の倍数の番地から読めば問題はありません。メモリ中のバイトの順番でそのままデータがレジスタに読み込まれます。しかし奇数番地から読んだらどうなるでしょう。この例では1番地の8ビットを上位にもってきて2番地の8ビットを下位にもってきてくっつける必要があります。このように16ビット、32ビット、64ビットのデータがバイトアドレッシングの境界にうまく整列していない問題をミスアラインメントと呼びます。



ミスアラインメントを許すかどうかは難しい問題です。命令コードでは許さないのが普通です。データについては悩ましいです。ミスアラインメントは、32ビットのデータをアクセスするのにメモリを2回に分けてアクセスするため、性能面では不利です。メモリ周辺のハードウェアも複雑になります。利点はメモリ利用効率が向上することですが、ミスアラインメントを許すことによるメモリの容量の効率化は効果がさほど大きくないです。ミスアラインメントを許すのは、主として、既に普及してしまったコード（レガシーコード）で、ミスアラインメントを許す状態でコンパイルしてしまったものをコンパイルしなおさなくても動くようにするため、という要求に基づく場合が多いようです。このため、データについては許す場合と許さない場合があります。ここでは両方共許さないことにします。



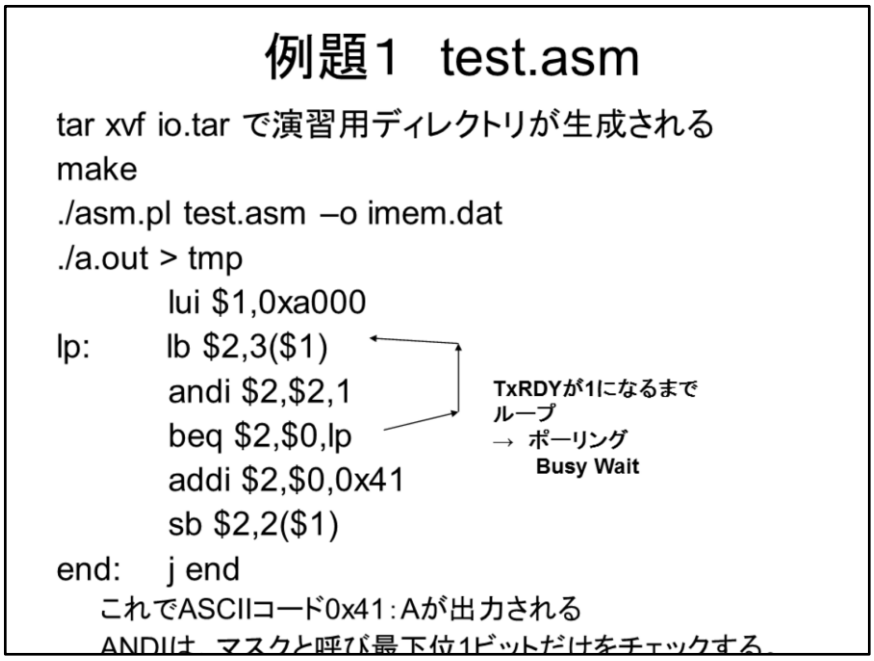
では、ここで演習用にディスプレイを想定します。このディスプレイは、ASCIIコードを書き込むとその文字が出力されます。ASCIIコードとは、英数字、記号用の8ビットの文字コードで、国際的に広く使われています。Linux端末ではman asciiで表示されます。ここでは、簡単のため、データレジスタを0xa0000002番地、ステータスレジスタを0xa0000003番地に割り当てます。ディスプレイに対してデータは、シリアルに転送され、表示には一定の時間が掛かります。TxRDYがこの番地の最下位ビットに割り当てられており、送信完了で1になります。

ASCIIコード

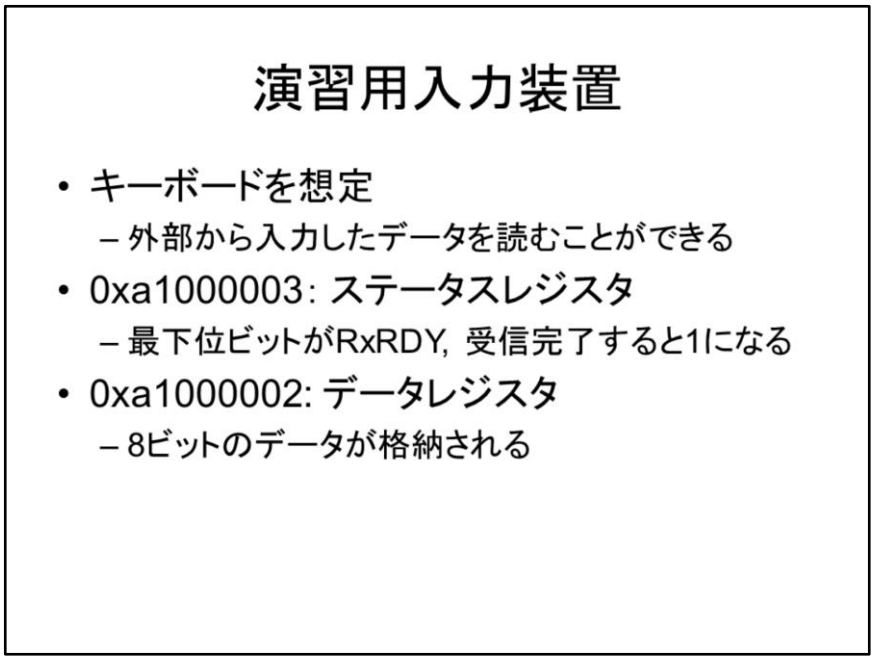
一部のみ、対応する数は16進数で示す

30	0	38	8	40	@	48	H	50	P	58	X
31	1	39	9	41	A	49	I	51	Q	59	Y
32	2	3A	:	42	B	4A	J	52	R	5A	Z
33	3	3B	;	43	C	4B	K	53	S	5B	[
34	4	3C	<	44	D	4C	L	54	T	5C	¥
35	5	3D	=	45	E	4D	M	55	U	5D	]
36	6	3E	>	46	F	4E	N	56	V	5E	^
37	7	3F	?	47	G	4F	O	57	W	5F	_

文字コードはコンピュータ上で文字を表すコードです。昔から使われているコードがASCIIコードで7ビット（8ビット）を使って英数、特殊文字を表現します。ここではその一部を示します。ここではman asciiで表示するのがいいでしょう。



この例題プログラムでは、r0にlui命令を使って0xa0000000を入れてやります。ディスプレースメントの機能を使ってデータを読んで（lb）は、0だったらLoopに戻る（beq）動作を繰り返します。ちなみにandiは最下位1ビットのみを判定に使うために他のビットを無視するための操作であり、これをマスクと呼びます。この操作でTxRDYが1になるまで待ってやります。このループから抜け出したということはTxRDYが1なので、データを書き込むことができます。ここでは‘A’の文字をディスプレイに出力するために0x41をデータレジスタに書き込みます。このように、フラグが1になるまでループしてチェックを繰り返す操作をビジーウエイト（Busy Wait)あるいはポーリング(Polling)と呼びます。



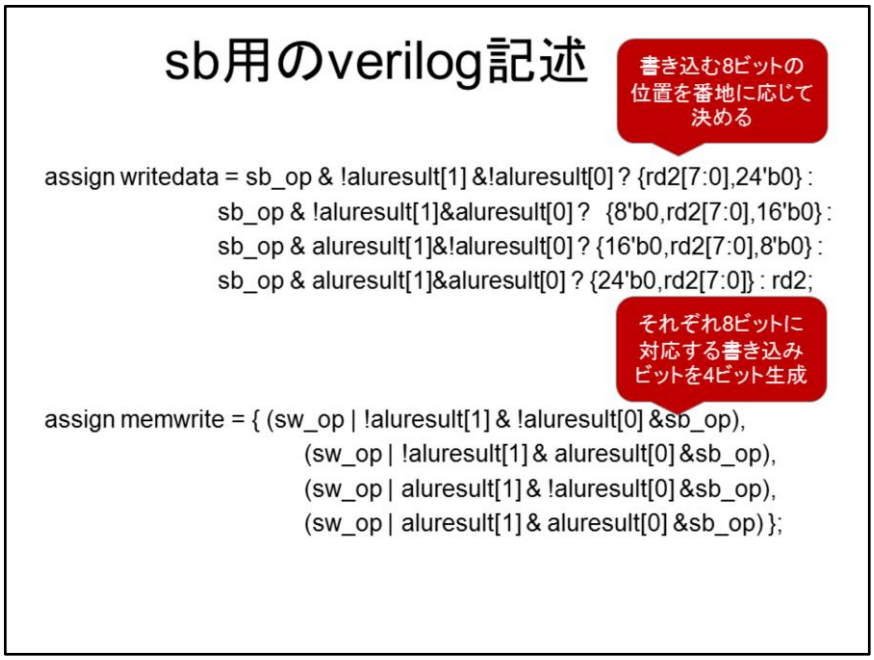
では今度は、演習用に入力装置を想定します。ここでは外部から入力したデータをCPU内部に読み込む装置を考えます。Oa1000003番地にステータスレジスタを割り当てます。この最下位ビットにRxRDYを割り当て、データ受信を完了するとここが1になるとします。このとき0xa1000002番地を読むと8ビットのデータを読むことができます。

例題2　io.asm

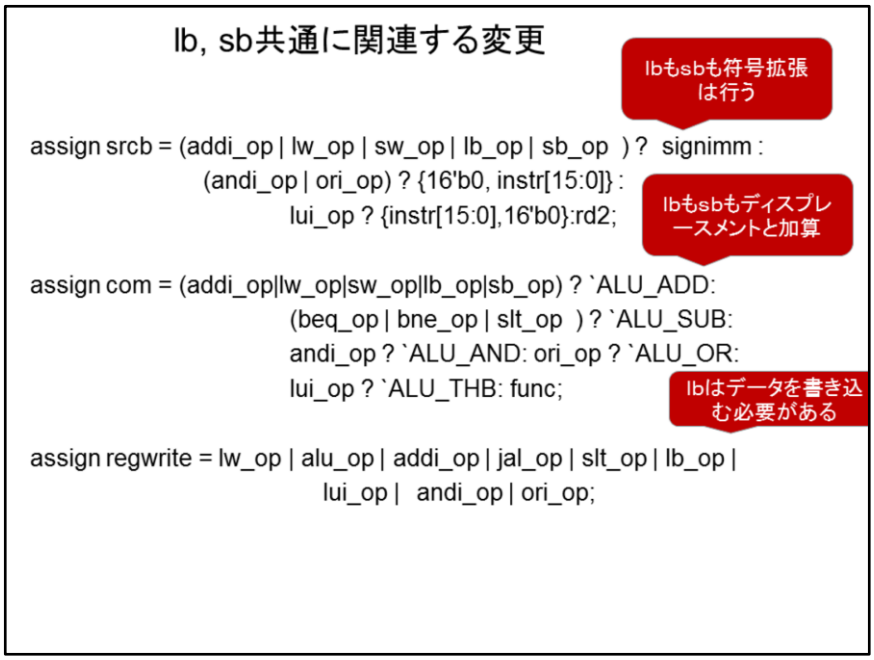
```
        lui $1,0xa100
lpin:   lb $2,3($1)
        beq $2,$0,lpin      RxRDYをBusy Wait
        lb $2,2($1)         ここでデータ読み込み
        lui $1,0xa000
lop:    lb $3,3($1)         TxRDYをBusy Wait
        beq $3,$0,lpo
        sb $2,2($1)         ここでデータ出力
end:    j end
```

入力されたコードがそのまま出力される。
面倒なのでマスクは省略してある。（ここでは実は不要）

では1文字読み出したデータを出力する例題をやってみましょう。読み出すときと出力するときの2回Busy Waitが必要になっています。ここではステータスレジスタの他のビットは0になるため、マスク操作は省略できます。



では今まで導入した命令のVerilog記述を見て行きます。実用的なCPUのメモリ周辺回路はデータの引き回しが必要で面倒です。まず8ビット単位のデータ書き込みを行うため、write enable信号（memwrite)が4ビット必要です。レジスタの下位8ビットから、それぞれの位置にデータを引き回す必要がある点にご注意ください。



他の部分はlb,sbはlw,swと共通です。同様に、符号拡張をしたディスプレースメントとレジスタの値を加算して実効アドレスを求めます。lbはレジスタへの書き込みを伴うので、regwriteも1にする必要があります。

lb用に8bitの位置を移動

```
assign result = slt_op  ? {31'b0,aluresult[31]} :
                jal_op ? pcplus4:
                lw_op  ? readdata :
                lb_op & !aluresult[1]&!aluresult[0]?
                        {{24{readdata[31]}},readdata[31:24]}:
                lb_op & !aluresult[1]& aluresult[0]?
                        {{24{readdata[23]}},readdata[23:16]}:
                lb_op & aluresult[1]& !aluresult[0]?
                        {{24{readdata[15]}},readdata[15:8]}:
                lb_op & aluresult[1]& aluresult[0]?
                        {{24{readdata[7]}},readdata[7:0]}:
                          aluresult;
```

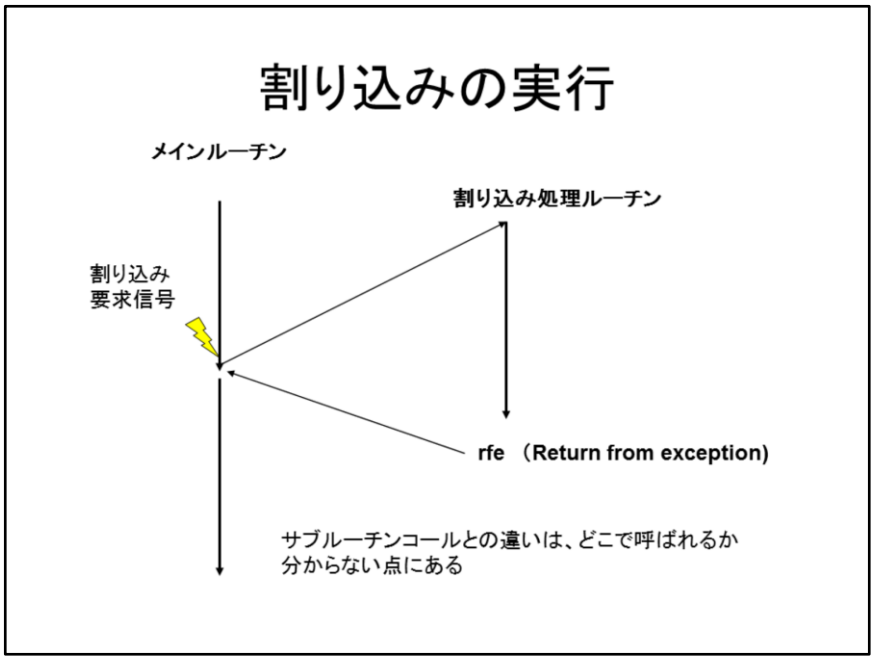
メモリの所定の位置から、アドレスに応じてレジスタの最下位8ビットにデータを移動します。これは単に面倒なだけで、規則的な操作です。

割り込み（Interrupt)

- 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rte（Return from Exception)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
    - 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
  - 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MIPSではepc(exception pc)というレジスタに格納する
  - ハードウェアスタック

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。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、MIPSではepc(Exception pc)に戻り番地をしまっておきます。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

- rte Return from Exception

  010000 0000000000000000 010000

  pc ← epcにし、割り込みを許可する

割り込みからの復帰はrte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。

例外処理：割り込みの一般化

- I/O割り込み
- トラップ/割り出し：ユーザプログラムからシステムプログラムを呼び出す
- ページフォルト：TLBミス
- セグメンテーションフォルト：TLBの保護違反
- ブレークポイント、トレース：デバッグ用
- バスエラー、未定義命令：システムのエラー

- 割り込みと類似の機構で取り扱う→例外処理

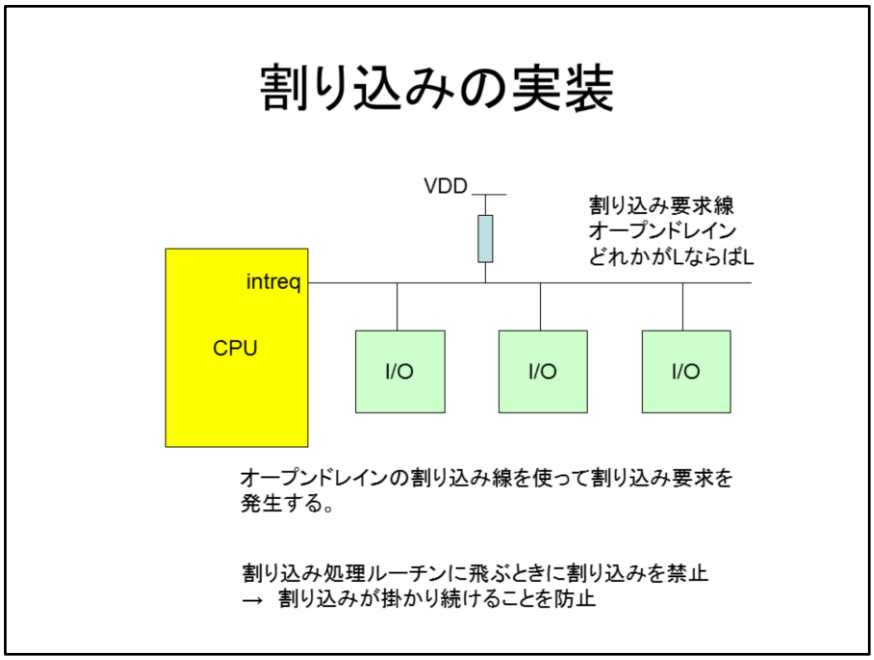
代表的な例外処理を挙げましょう。このうちやっかいなのはページフォルトです。忘れたかもしれませんが、計算機構成の最後の辺にやったTLBがミスヒットした場合、対応するページが主記憶に存在しなかった場合などに起きます。これらは全て割り込みと類似した機構で処理され、一般的に例外処理と呼ばれます。

例題3：割り込みのシミュレーション

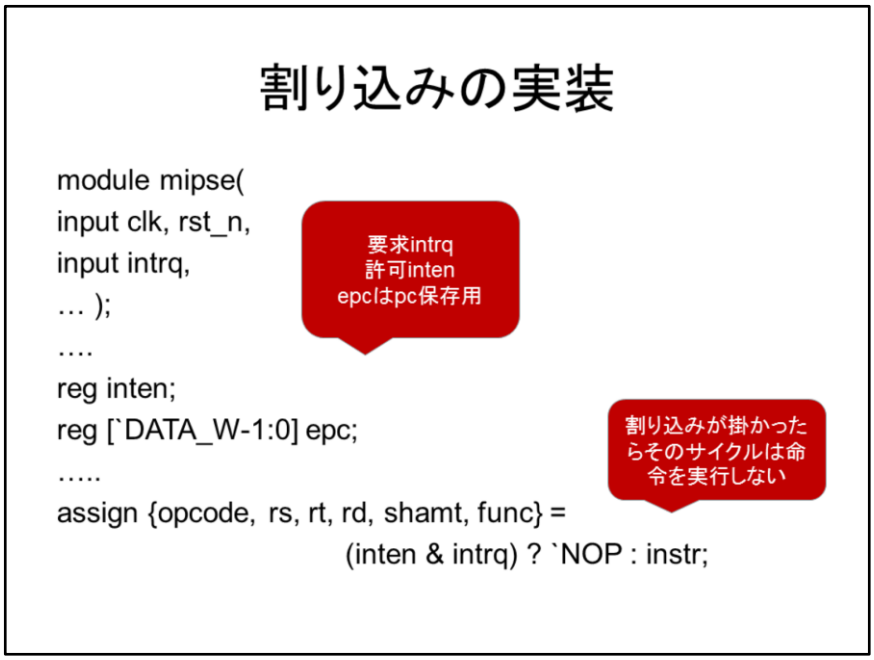
tar xvf int.tarで割り込み演習用ディレクトリが生成される
make
./asm.pl test.asm –o imem.dat
make intmem.dat
./a.out > tmp
でtmpを見てみよう

test.asmは、フィボナッチ数列を計算するメインプログラム
testint.asmは、例題2の1文字入出力プログラム

では割り込みのシミュレーションをやってみましょう。このプログラムでは二つの処理（フィボナッチ数列の計算と、例題2の1文字入力プログラム）を並行して動かしています。割り込みのお蔭でスムーズに処理が分離されます。



割り込みを実装するために、CPUは割り込み要求入力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。



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 & inten) pc <= `INT_AD;
  else if(rfe_op) pc <= epc;
  else if (j_op | jal_op)
          pc      <= {pc[31:28],instr[25:0],2'b0};
  else if (jr_op)
          pc      <= srca;
  else if ((beq_op & zero) | (bne_op & !zero))
    pc <= pcplus4 +{signimm[29:0],2'b0} ;
  else
    pc <= pcplus4;
```

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

rfe割り込み復帰命令でepc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0x80000010番地に飛ぶようにします。さらにrfe命令によっていepcの中身をPCに戻すようにします。

割り込み許可フラグとepc

```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) inten <= 1;
  else if(intrq & inten) inten <= 0;
  else if(rfe_op ) inten <= 1;
end

always @(posedge clk)
  if(intrq & inten) epc <= pc;
```

割り込み許可フラグの制御

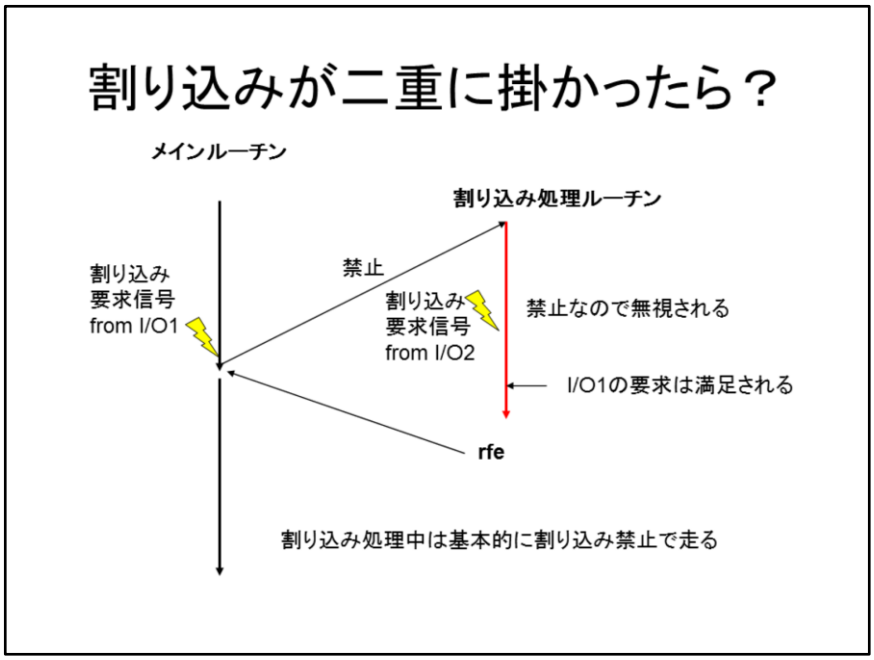
割り込みが掛かるとそのときのpcがepcに保存される

割り込み許可フラグと、epcレジスタの制御用のコードを加えます。

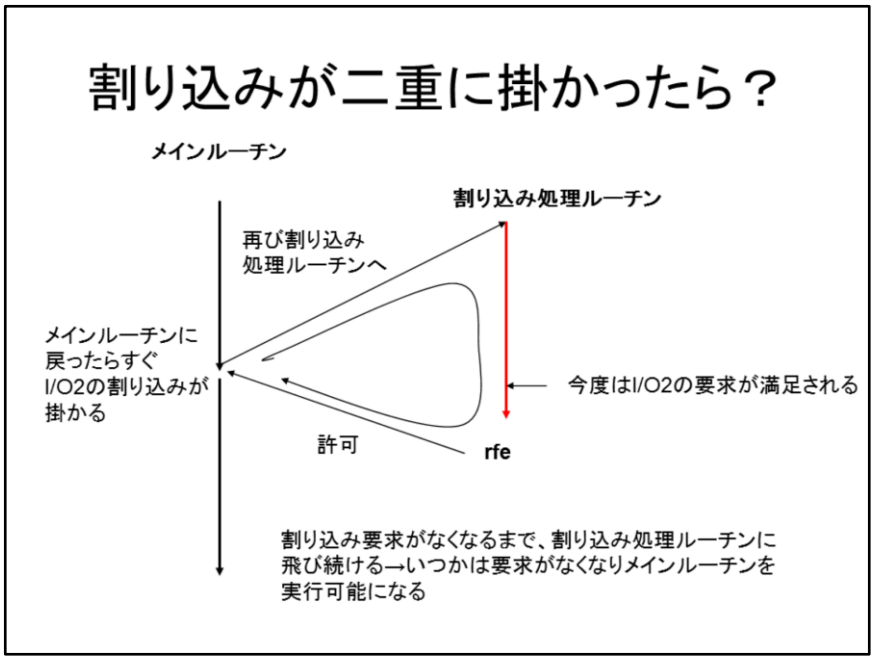
MIPSの割り込みの実際

- 本当はMIPSでは割り込みはコプロセッサcp0が扱いもっと複雑
- 例外処理(Exception)の一つ
  - 未定義命令
  - アドレスエラー
  - 演算の結果のオーバーフロー
  - TLBミス
  - 特権命令違反
- 原因レジスタ、状態、割り込みマスクなど様々な特殊レジスタがある

MIPSの割り込みは本当はもっと複雑で、CPU本体ではなく、コプロセッサcp0が他の例外処理と共に扱います。MIPSの扱う例外処理は、未定義命令の実行、アドレスエラー(ミスアラインメントの制約を破った場合など)、演算結果の桁あふれ、TLBミス、特権命令違反など様々なものがあります。これを扱うためにcp0は原因レジスタ、状態レジスタ、割り込みマスクなど様々なレジスタを持ちます。

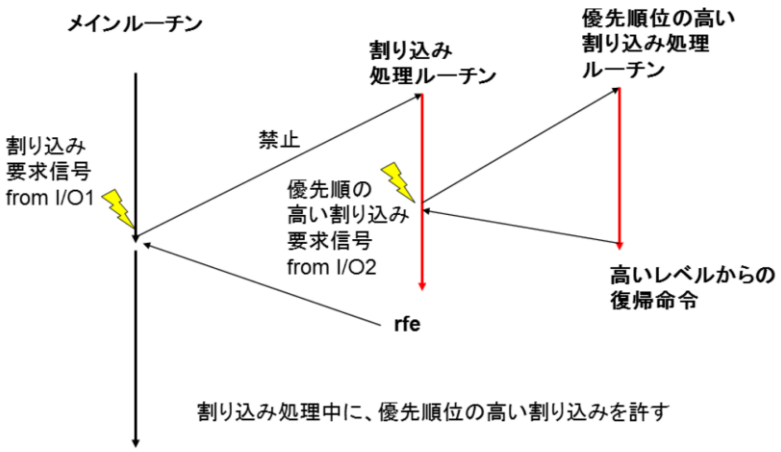


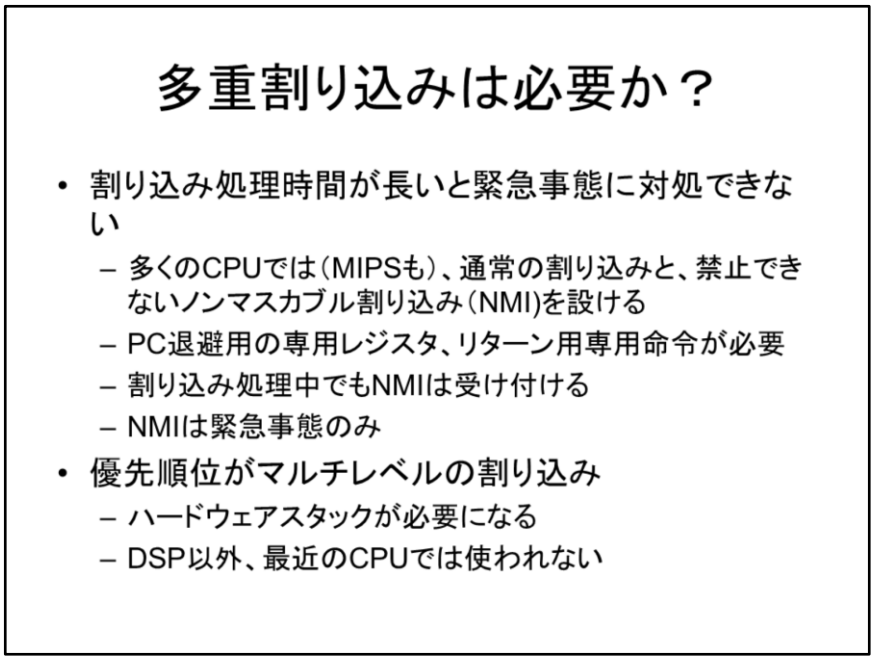
一般的に、割り込み処理ルーチンに飛ぶと同時に割り込みは禁止となります。そうでないと、割り込みが掛かり続けて先に処理が進まなくなるためです。基本的に割り込み処理ルーチンは禁止状態で走って、そのどこかでI/Oの要求は満足されて、割り込み要求はリセットされます。rfe命令でメインルーチンに戻ったときには、すでに要求は出ていないので、CPUはメインルーチンの先を続けることができます。



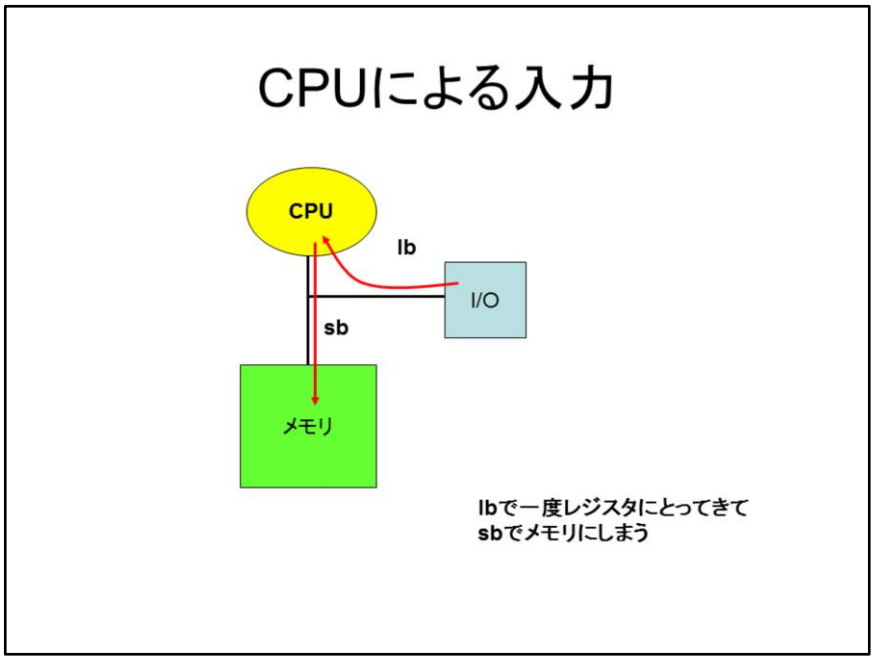
では、割り込み処理中に別の割り込み要求が掛かったらどうなるでしょうか？通常は、割り込み処理ルーチンは最初の割り込みの処理が終わったら、メインルーチンに戻ります。しかし、別の割り込み要求が掛かり続けているので、メインルーチンに戻って割り込み許可になった瞬間に再び割り込み処理ルーチンに飛ばされます。割り込み処理ルーチン中で、次の割り込み要求に対する処理を行い、再びメインルーチンに戻ります。いくつ割り込みが同時に掛かっても、この動作を繰り返して順番に処理していけば、そのうち要求がなくなりメインルーチンが実行可能になります。

多重割り込み

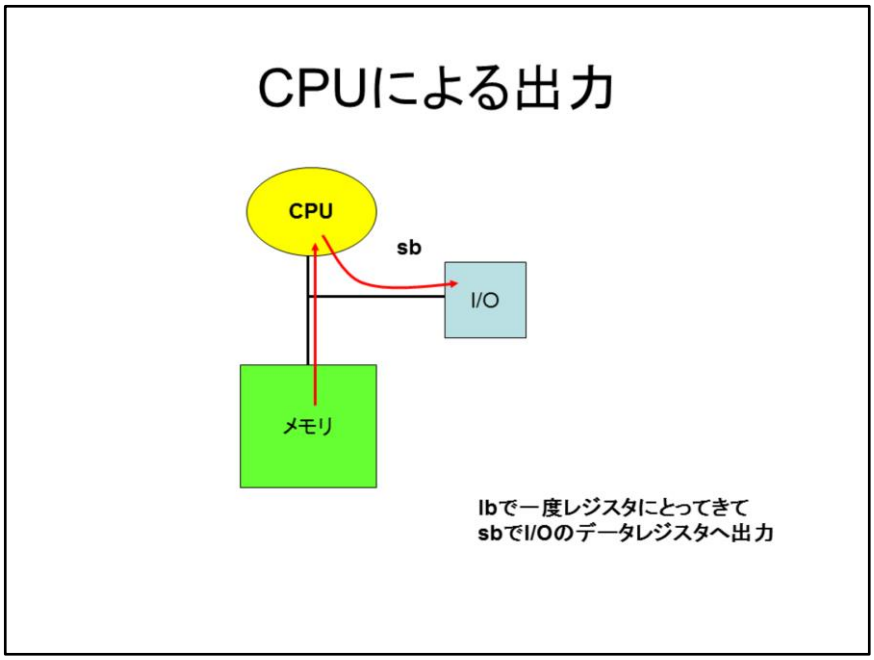

メインルーチン
割り込み
処理ルーチン
優先順位の高い
割り込み処理
ルーチン
割り込み
要求信号
from I/O1
禁止
優先順の
高い割り込み
要求信号
from I/O2
高いレベルからの
復帰命令
rfe
割り込み処理中に、優先順位の高い割り込みを許す



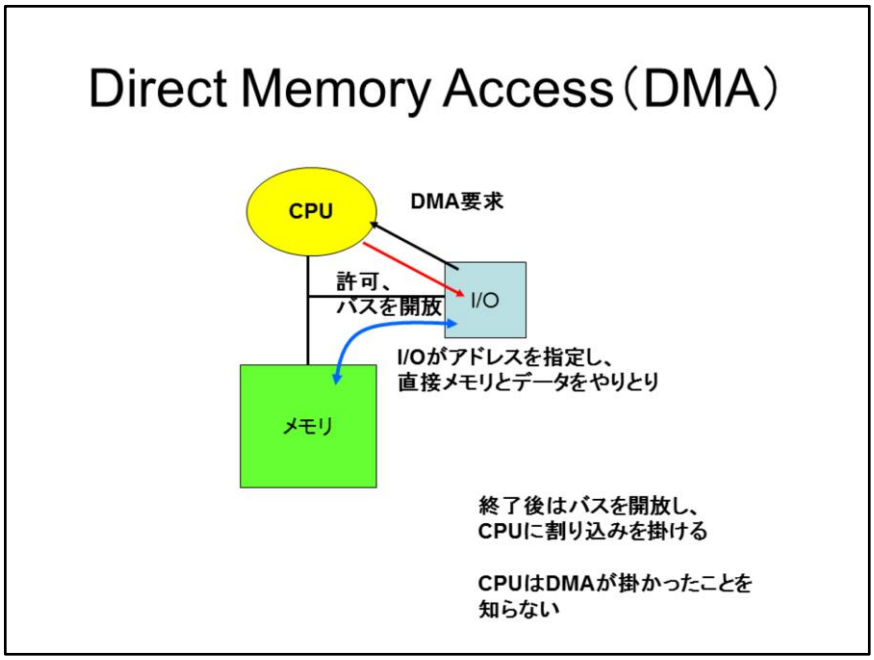
しかし、割り込み処理の時間が長いと、この方法では緊急事態に対処することができません。そこで、MIPSを含む多くのCPUでは、通常の割り込みと、禁止できない優先順位の高い割り込み（ノンマスカブル割り込み）の2種類を持っています。ノンマスカブル割り込み（NMI)は、割り込み処理実行中でもその要求が受け付けられ、別の番地に飛びます。この機構を実現するにはNMI専用のPC退避用レジスタ、リターン命令が必要になります。信号処理用のDSPなどは複数のレベルの優先順位を持つ複雑な割り込み機構を持っていますが、一般的なCPUではあまり使われません。



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。



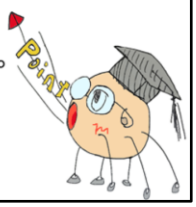
逆にI/Oから出力する場合はどうでしょう？lbでメモリからCPUのレジスタにデータを取ってきて、そこからI/Oにデータを出力します。2回のコピーが必要です。これは、CPUが常にメモリとI/Oを結ぶバスの利用権を握り続けているためです。



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

- I/Oの繋ぎ方は、普通のメモリと同様に接続するメモリマップトI/Oと専用の命令を使うプログラムドI/Oがある
- I/Oはデータレジスタ、ステータスレジスタ、コマンドレジスタを使って、外界とのデータのやりとりをする
- ハンドシェークはフラグを用いて書きつぶしや二重読みを防ぐ
- フラグをチェックするのにはBusy Waitするのが基本だが、CPUが忙しくなってしまう。
- 割り込みはI/Oから要求してCPUの動きを変える
- 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。