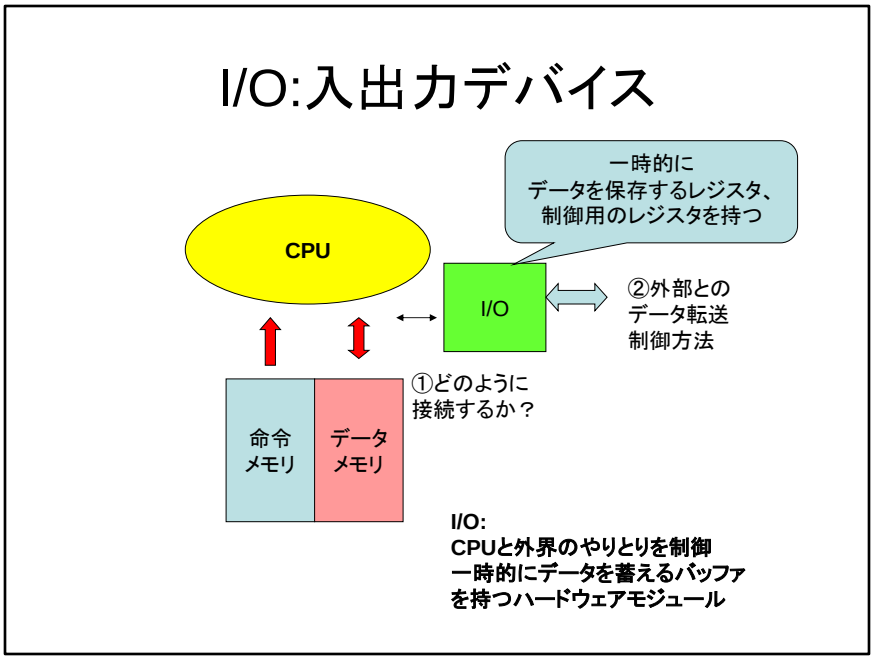
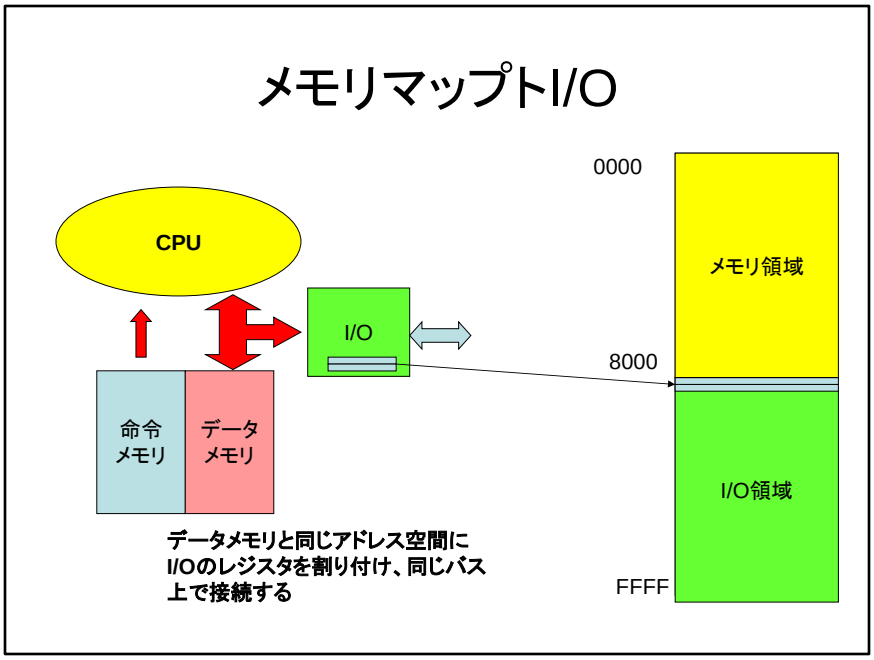
計算機構成同演習
入出力

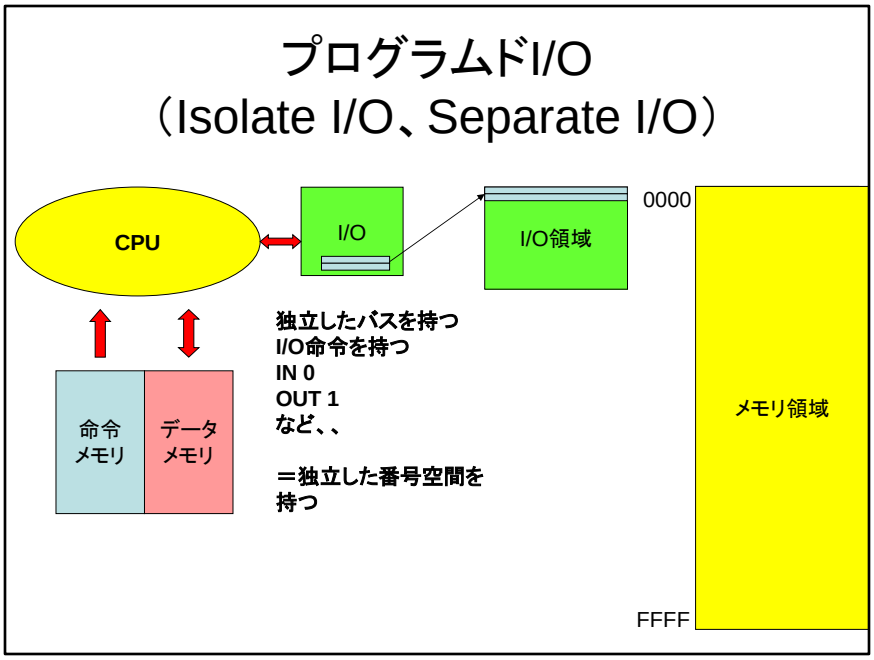
天野　hunga@am.ics.keio.ac.jp



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。

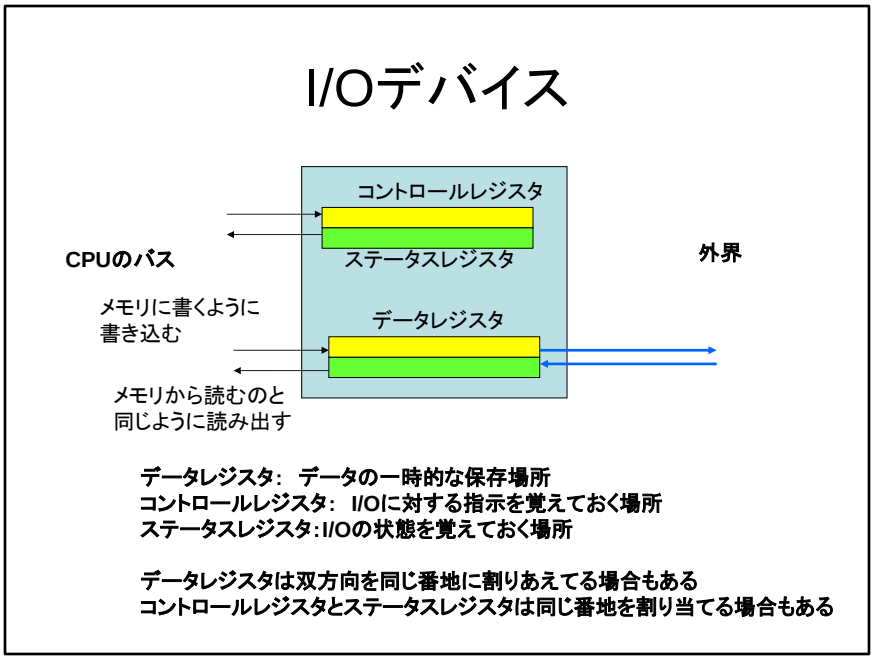


これに対してプログラムドI／Oは、IN命令、OUT命令など専用の命令でI／Oに読み書きをします。IN 0, OUT 1などI／O番号を指定しますが、これはすなわちメモリから独立したアドレス（番地）空間を持っているということに他ならないです。この方式は、バスもメモリと独立しているIsolate I/Oとバスは共通で番地空間だけ分けるSeparate I/Oを分けて考える人も居ますが、あまりこだわらない人も居ます。そもそもプログラムドI／Oという名前もさほど一般的ではありません（でも正式にはこう言うらしいです）。この方法は、独立した番号の空間があるので、メモリ領域を無駄遣いしない利点がありますが、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



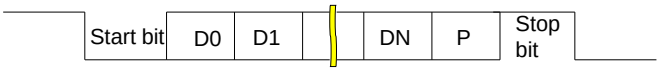
I／Oデバイスは、データレジスタ、コントロールレジスタ、ステータスレジスタの三種類のレジスタを持っています。データレジスタはCPUが読み書きする場合、一時的にデータを保存しておく場所です。入力用のデータレジスタと出力用のデータレジスタは同じ番地に割り当てられており、書き込むと出力され、読み出すと入力される場合もあります。ステータスレジスタは、I／Oの状態を覚えておく場所で、データが到着したり、データを送り終わったりしたことを示す情報を持っています。基本的には読み出し専用です。コントロールレジスタはI／Oに対する指示を記憶する場所です。Ethernetコントローラなどの動作が複雑なI／Oでは、多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

例：UART　8251

- パラレル/シリアル変換を行うI/O
- モデム・端末インタフェースRS232C用
  - 5bit-8bit単位でシリアル転送を行う
  - ボーレートはさほど高くない（最大でも10Mbps)
- 現在でもIPとしてFPGA内で良く利用される

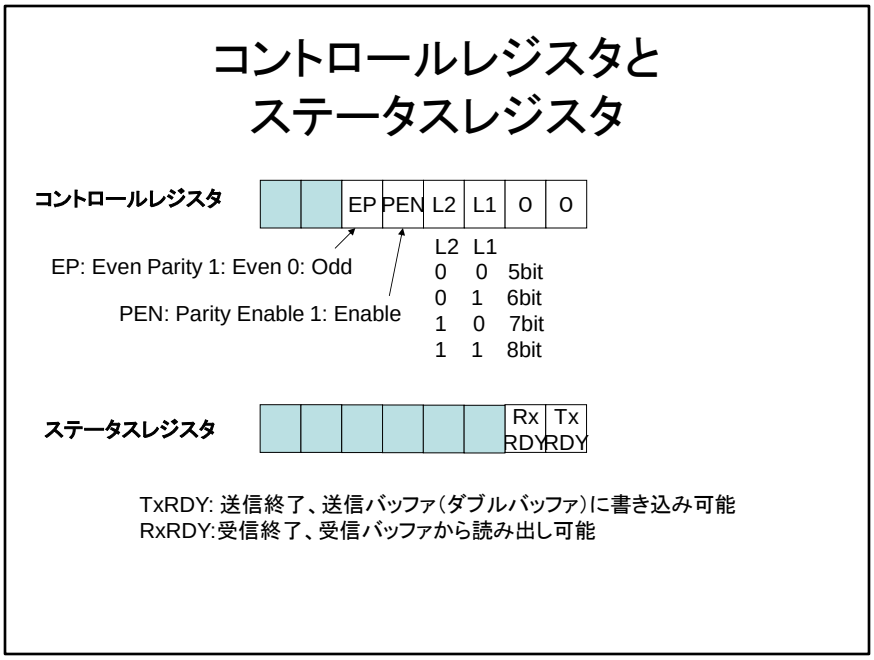
古典的なI／OとしてUART 8251を紹介しましょう。このI／Oは、パラレル／シリアル変換用で、CPUからのデータを直列に、すなわち時間的に順番に出力し、直列に入力されたデータをCPUの並列データとして受け取ります。5bit-8bit単位のデータを10Mbps(bit per second)という遅い転送レートで送ります。昔、データを音に変換して電話で交信するモデムという装置が使われたときの規格であるRS232C用に作られました。RS232Cは、モデムがなくなった後も簡単な端末用のインタフェースとして使われ、今でもその簡便さから遅くも良いインタフェースとして用いられます。8251は今でもFPGA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



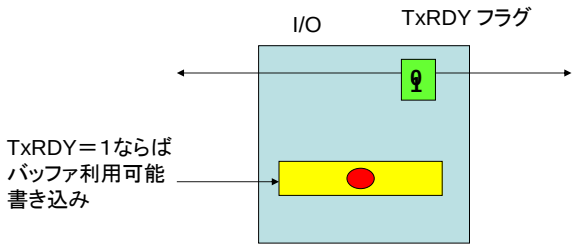
Start bit：　0にして開始を示す
D0-DN：　データ、5ビットから8ビットまで選択可能
P:パリティビット　偶数/奇数の選択可能
Stop bit:　1にして終了を示す。長さを選択可能

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。



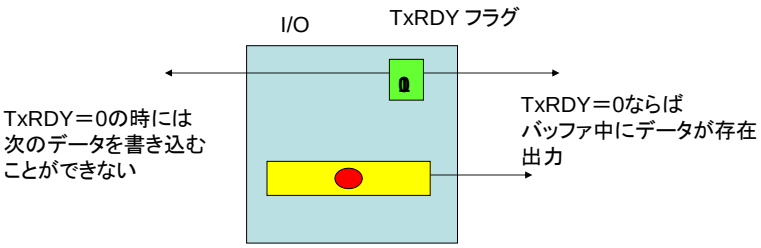
この8251のコントロールレジスタとステータスレジスタを紹介します。コントロールレジスタは、パリティの選択、データ長の選択を行ないます。パリティ（Parity)とは、1ビットの誤り検出符号で、データ内の1の個数を偶数または奇数にそろえることにより、1ビットエラーの検出を行ないます。偶数パリティを例に取って説明します。データ内の１の個数が偶数ならばParity bitを0とし、奇数ならばParity bitを1とします。Parity bitを含めた全体のデータの1の個数は常に1になるので、1の数が奇数のデータを受け取ったら誤りがあったことに気づくことができます。8251ではPEN=1でパリティを使う設定にし、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(送信）



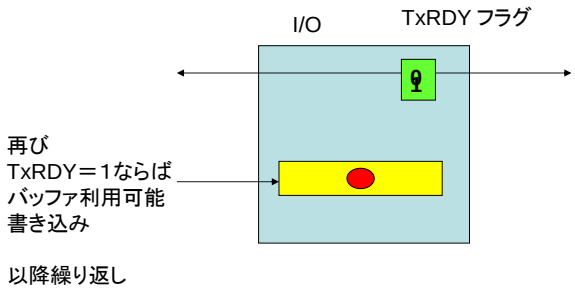
ハンドシェイクとは、送信側と受信側が同期を取って取りこぼしなくデータを転送するのに使われる方法です。送信を例に取って示しましょう。CPUがステータスレジスタを読んで、TxRDYが1ならば、バッファが空いているので、ここにデータを書き込みます。すると、TxRDYが0になります（正確にはダブルバッファなので動きが多少違うのですが、）。TxRDYのようにバッファの状態を示す1ビットの情報をフラグ（旗）と呼びます。分岐命令の条件を示すフラグと機能は同じです。

ハンドシェイク(送信）



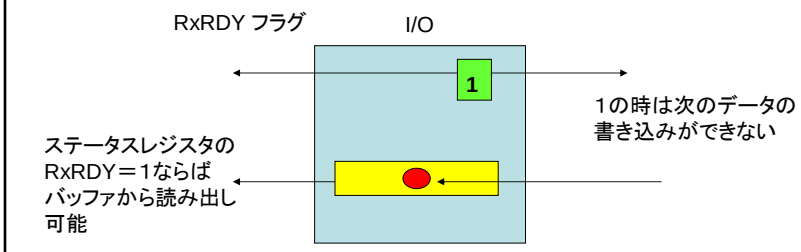
CPUはTxRDY=0の時は、バッファ内のデータはまだ転送が終わっていないので、待ち状態になります。外部にあるモデム装置（あるいは端末）は、TXRDY=0でデータが書き込まれたことが分かり、受信状態になりシリアルにデータが転送されます。

ハンドシェイク(送信）



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。

ハンドシェイク(受信）



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

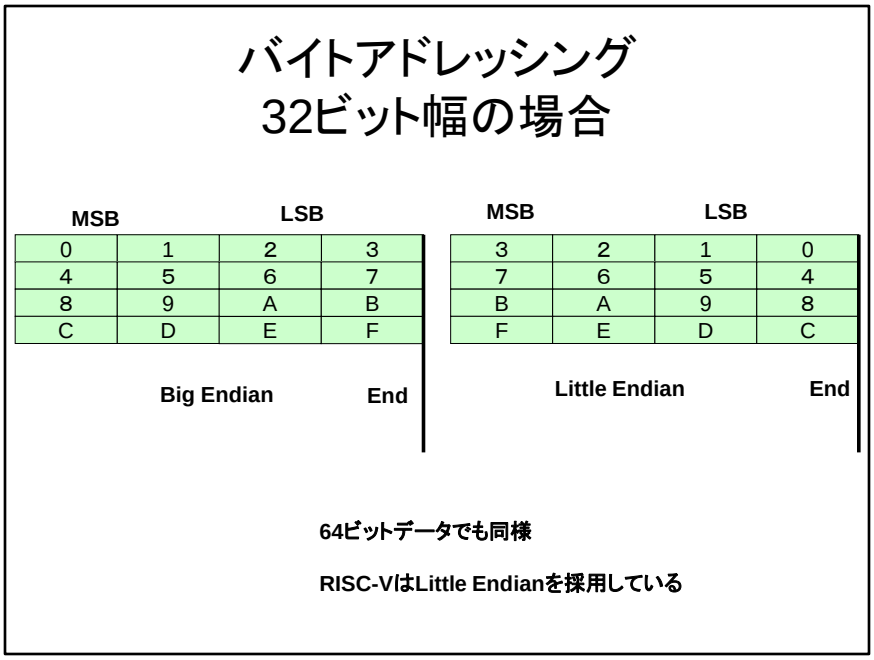
フラグを使うことで、読み出し、書き込み間の障害（ハザード）を防ぐことができます。一般的にハザードは3種類あります。Read After Write(RAW)ハザードは、ちゃんと値を書き込まれるのを待って読み出すことで、これがおきると同じデータを間違って複数回読んでしまう問題が起きます。一方、Write After Read (WAR)ハザードは、読む前に書いてしまう問題で、これは書き潰しを起こしてデータが消失してしまう問題です。Write After Write (WAW)ハザードは書き込む順番が狂う問題で、単純な入出力では考えなくてもいいです。これらのハザードは、パイプライン処理でも起こるの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

メモリのアドレッシング

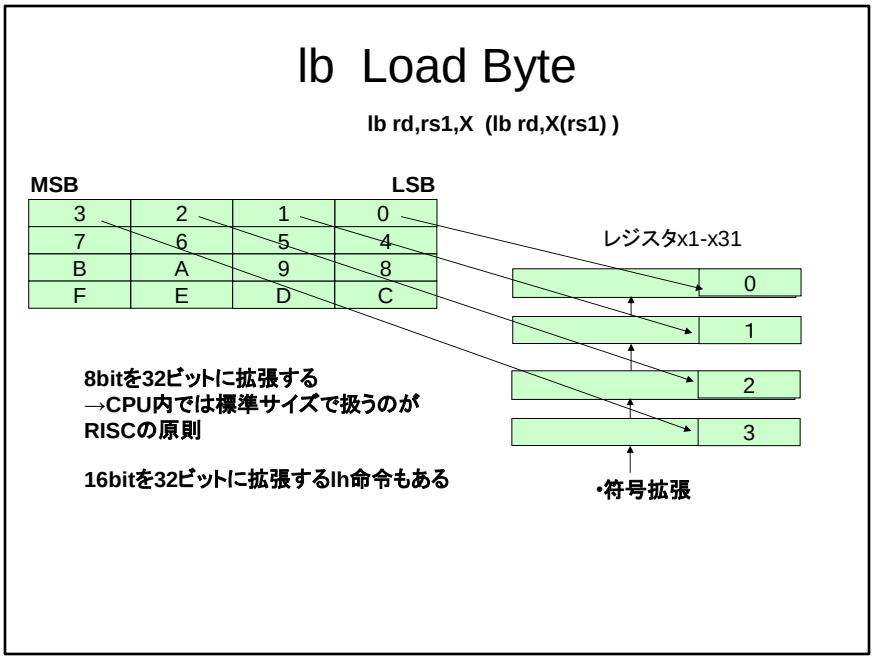
- I/Oデータは8ビット・1ビットなど短い幅が多い
- データ幅が一律32ビットでは効率が悪い
- ASCIIなど文字コードは短い幅が多い
  man asciiと打ち込んでASCIIコードを見てみよう

→ 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

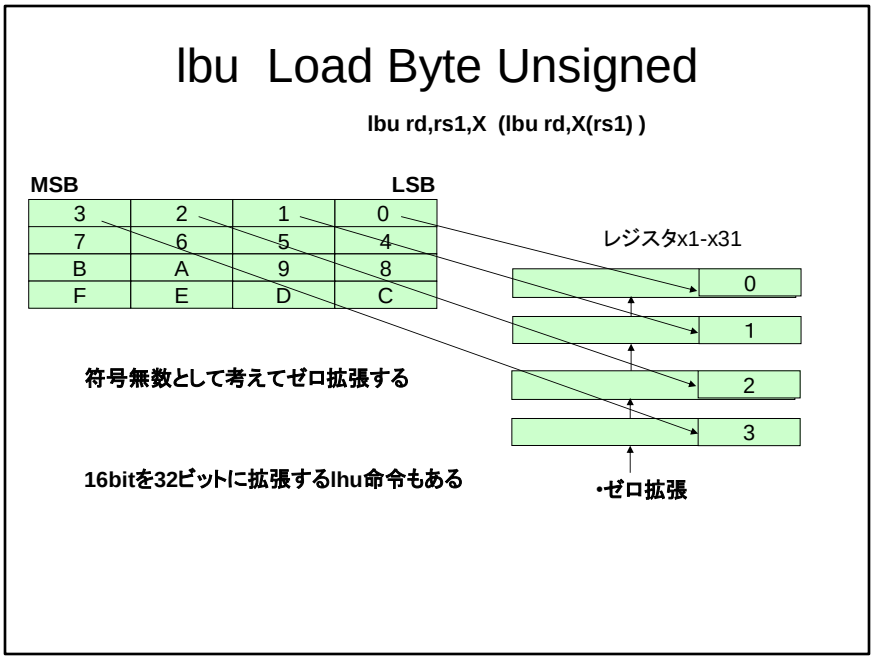
さて、ここまででI／Oについて紹介しましたが、いくつか問題があります。I／Oデータは8ビットのASCIIコードや数ビットのフラグが多いので32ビットデータのうちの一部しか使っていません。これはもったいないので、実は今のコンピュータは8ビット単位のバイトアドレッシングを用いています。これについて紹介します。次に、ビジーウェイト中CPUは無駄にループを回っていて馬鹿みたいです。この問題を解決するための手法として割り込みを紹介します。



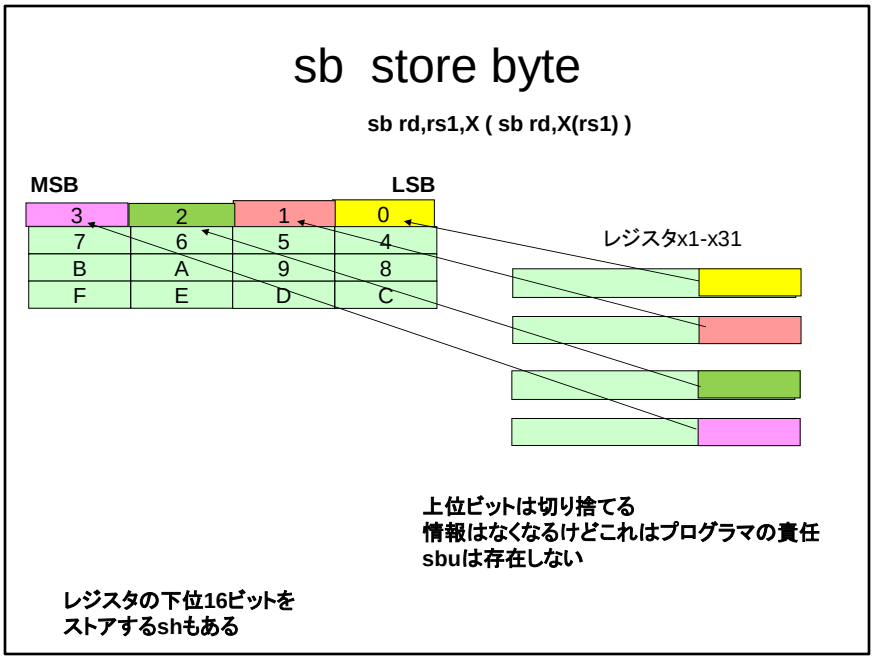
I／Oデータなどを扱う上での無駄を防ぐために、最近のCPUは32ビットではなく8ビット単位で番地が振られています。すなわち、32ビットは番地4つ分に当たります。ここで、桁の大きい方から0，1と振っていく方法と小さい方から0，1と振っていく方法の二つが考えられます。前者をBig Endian,後者をLittle Endianと呼びます。ここでは以降、Big Endianで番号を振ることにします。この振り方は統一が取れておらず、コンピュータ間でデータを交換する際にトラブルの元となります。最近のCPUは電源投入時の指定でどちらの方法を取ることもできるものが多いです。MSB側から0を振っていくと、大きい方で端（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。



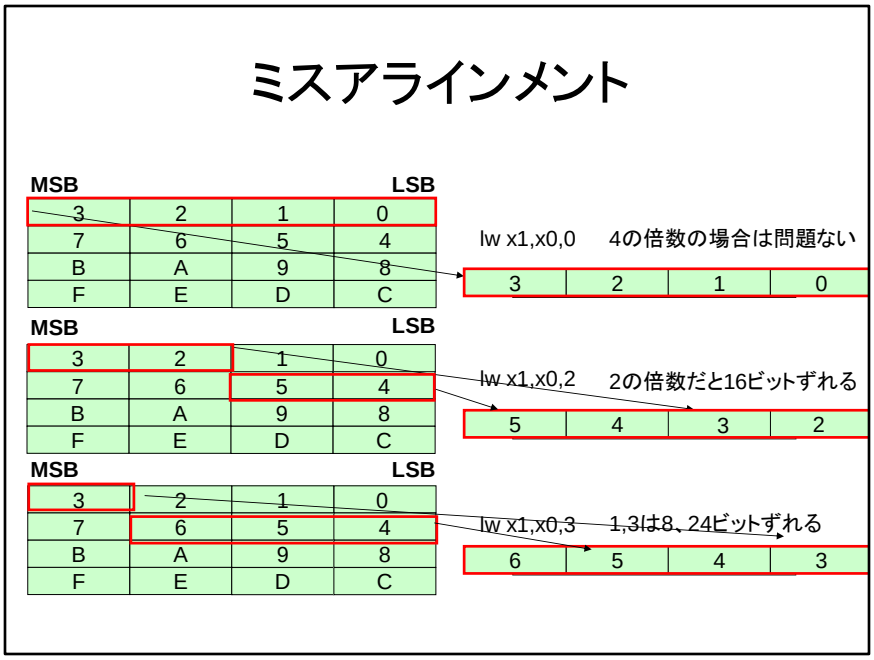
では、バイトデータを扱う命令を定義しましょう。lbは指定されたアドレスの8ビットを読み出して、レジスタの下位8bitに置きます。上位24bitは符号拡張されます。RISCではCPUの内部ではデータのサイズを統一してしまうのが普通です。この場合も読み出す際に32ビットに拡張します。lbはlwと同じR型で定義します。ちなみにRV32Iでは16ビットデータを読み出して32ビットに拡張するlh(Load Halfword)もあります。最近の文字コードは16ビットが多いので、案外16ビットデータは利用価値が高いためです。ただし、ここでは実装が面倒なので省略してあります。



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。RV32Iには、16ビットのLoad命令lhu(load half word unsigned)もあります。



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。16bit用のstore halfwordは存在します。



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？

- 許す利点:
  - メモリ利用効率が向上する
  - レガシーコードがコンパイルしなおさずに動く
- 許す欠点:
  - 動作速度が遅くなる
    - 実装によってはメモリを2回アクセスしなければならない
  - ハードウェアが複雑になる
    - バイトをシフトしてくっつける必要がある
    - シフト操作自体はlbの時点で必要だが、、、
- 一般に命令では許さない。データではケースバイケース
- RISC-Vでは許している

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。RISC-Vではデータについてはミスアラインメントを許しています。

例題のディスプレイ

- ディスプレイを想定
  - ASCIIコードを書き込むと出力される
  - ASCIIコード：英数字、記号用の8ビットコード
    - man asciiで表示されるのでやってみて!
- 0xc0000003：ステータスレジスタ
  - 最下位ビットがTxRDY, 送信完了すると1になる
- 0xc0000002: データレジスタ
  - ASCIIコードが対応する文字に出力される
- 0xc0000000からとしたのは、RISC-Vのメモリ領域の外側に当たるため

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0xc0000002番地、ステータスレジスタを0xc0000003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

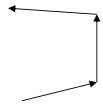
一部のみ、対応する数は16進数で示す

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。

例題1 otst.asm

```
        lui x1,0xc0000
lp:     lb x2,x1,3
        andi x2,x2,1
        beq x2,x0,lp
        addi x2,x0,0x41
        sb x2,x1,2
end:    beq x0,x0, end
```



TxRDYが1になるまで
ループ
→ ポーリング
Busy Wait

これでASCIIコード0x41：Aが出力される
ANDIは、マスクと呼び最下位1ビットだけをチェックする。
出力までに遅延があるのでecallは使っていない

この例題プログラムでは、r0にlui命令を使って0xc0000000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。

例題の入力装置

- キーボードを想定
  - 外部から入力したデータを読むことができる
- 0xc1000003: ステータスレジスタ
  - 最下位ビットがRxRDY, 受信完了すると1になる
- 0xc1000002: データレジスタ
  - 8ビットのデータが格納される

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。Oc1000003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0xc1000002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　iotst.asm

```
        lui x1,0xc1000
lpin:   lb x2,x1,3
        beq x2,x0,lpin      RxRDYをBusy Wait
        lb x2,x1,2          ここでデータ読み込み
        lui x1,0xc0000
lop:    lb x3,x1,3          TxRDYをBusy Wait
        beq x3,x0,lpo
        sb x2,x1,2          ここでデータ出力
        ecall x0,x0,x0
```

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。

sb用のverilog記述

書き込む8ビットの位置を番地に応じて決める

```
assign writedata =
        sb_op & aluresult[1]&aluresult[0] ? {rd2[7:0],24'b0} :
        sb_op & aluresult[1]&!aluresult[0] ? {8'b0,rd2[7:0],16'b0} :
        sb_op & !aluresult[1]&aluresult[0] ?  {16'b0,rd2[7:0],8'b0} :
        sb_op & !aluresult[1] &!aluresult[0] ? {24'b0,rd2[7:0]} : rd2;
```

それぞれ8ビットに対応する書き込みビットを4ビット生成

```
assign memwrite = { (sw_op | aluresult[1] & aluresult[0] &sb_op),
                    (sw_op | aluresult[1] & !aluresult[0] &sb_op),
                    (sw_op | !aluresult[1] & aluresult[0] &sb_op),
                    (sw_op | !aluresult[1] & !aluresult[0] &sb_op) };
```

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。

ALU,レジスタファイル

lbもsbも符号拡張は行う

```
assign srcb = imm_op | lw_op | lb_op |jalr_op ? {sext, imm_i}:
                               bra_op ? {sext[18:0],imm_b}:
                               sw_op | sb_op ? {sext, imm_s}:
                               jal_op ? {sext[10:0], imm_j}:
                               reg2;
```

lbもsbもディスプレースメントと加算

```
assign addcom = (lw_op|lb_op|sw_op|sb_op|bra_op|jal_op|jalr_op);
```

lbはデータを書き込む必要がある

```
assign rwe = lw_op | lb_op | alu_op | imm_op | jal_op | jalr_op | lui_op
```

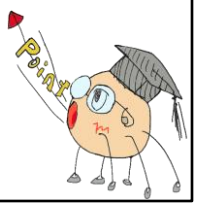
lb用に8bitの位置を移動

```
assign result = (jal_op|jalr_op) ? pcplus4 :
      lui_op ? {instr[31:12],12'b0}:
      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。

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。



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

入力装置から入力した数字をASCIIコードに直して出力するプログラムascii.asmを書け

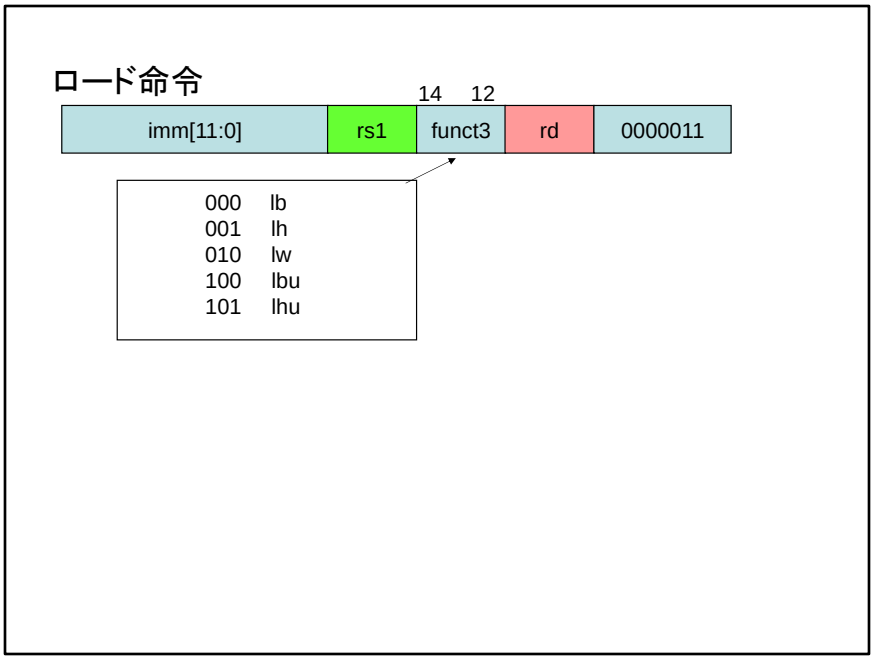
iotst.asmを改造すれば良い

test_rv32it.vをtest_rv32i.vにコピーして使え（元のも取っておいた方がいい）

1、2、3、4が表れるはず

演習2

- lbu を実装せよ　この命令は8ビットのデータをゼロ拡張してレジスタに格納する点以外はlbと同じである
- 命令形式は次のページ
  - 演習用のrv32i.v中には用意してある
- 提出物：改造したrv32i.v
- テストにはlbutst.asmを用いよ
- x2が最初0xffffffaa次に0x000000aaになればOK



ロード命令はイミーディエイト命令と形式が同じです。ストア命令はrs2の位置を他と揃えるために、イミーディエイト値が分割されています。