計算機構成　第6回
RISCの構成とVerilog記述
テキスト第4章

慶應大学
天野英晴

前回教育用のRISC POCOを導入しました。今日はそのVerilog記述を紹介します。まず、この復習をやっておきましょう。

メモリの読み書き

- レジスタ間接指定
  LD r0,(r1)　r1の中身の番地のデータを読み出してr0に転送
  ST r0,(r1)　r1の中身の番地に、r0を書き込む
- 実効アドレス（実際に読み書きされるアドレス）＝レジスタの内容
- 他にもアドレッシングモード（実効アドレスを決める方法）は色々あるがPOCOはレジスタ間接指定しか持って居ない
  - アキュムレータマシンの際のLD　0は、直接指定と呼ぶ
  - しかしこれはPOCOでは持っていない

最も重要な点は、メモリの読み書きで、レジスタ間接指定の理解です。これはポインタと一緒なので、間違えないように修得してください。

基本演算命令

- レジスタ同士でしか演算はできない
  - ADD r1,r2　　r1←r1+r2
  - SUB r1,r2　　r1←r1-r2
  - AND r1,r2　　r1←r1 AND r2
  - OR r1, r2　　r1←r1 OR r2
  - SL r1　　r1<<1
  - SR r1　　r1>>1
  - MV r1,r2　　r1 ← r2 単純な移動
  - NOP　　何もしない（NoOperation）

RISCなので、基本の演算はレジスタ同士でしかできません。MVはレジスタ間のデータ移動なので気をつけてください。

イミーディエイト命令

- 命令コード中の数字(直値）がそのまま演算に使われる

  LDI　r1,#1　　r1←1

  ADDI r1,#5　　r1←r1+5

  – 直値は8ビット符号付　→　演算時は16ビットに符号拡張（sign extension)される

  – 符号無し命令

    LDIU　r1,#200　r1←200

    ADDIU r1,#0xf0　r1←r1+0xf0

    (ADDI r1,#0xf0ならばr1←r1-16)

イミーディエイト命令は、命令コード中の数字がそのまま演算に使われます。符号付の命令は直値が符号拡張されます。LDI,ADDIがこれに当たります。符号無し命令は頭に0を補うのもので、LDIU、ADDIUがこれに当たります。

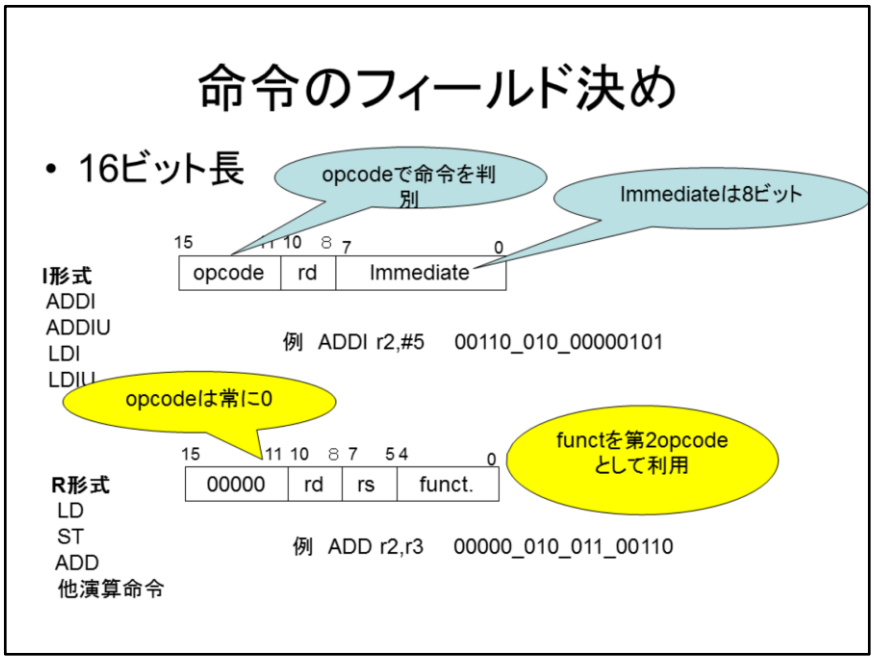
プログラム例

- 0番地の内容と1番地の内容を加算して2番地に格納せよ

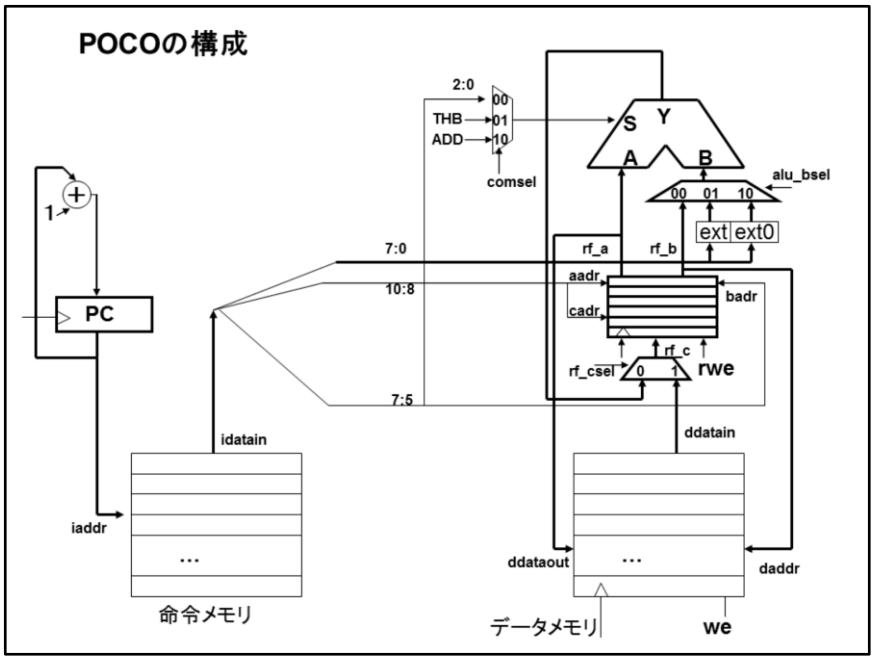
```
LDI  r0,#0        // r0: ポインタ
LD r1,(r0)        // 0番地から読み出し
ADDI r0,#1        // ポインタを先に進める LDI r0,#1でもいい
LD r2,(r0)        // 1番地から読み出し
ADD r1,r2         // 加算
ADDI r0,#1        // ポインタを先に進める LDI r0,#2でもいい
ST r1,(r0)        // 結果を2番地にしまう
```

LDIの代わりにLDIU、ADDIの代わりにADDIUでも良い

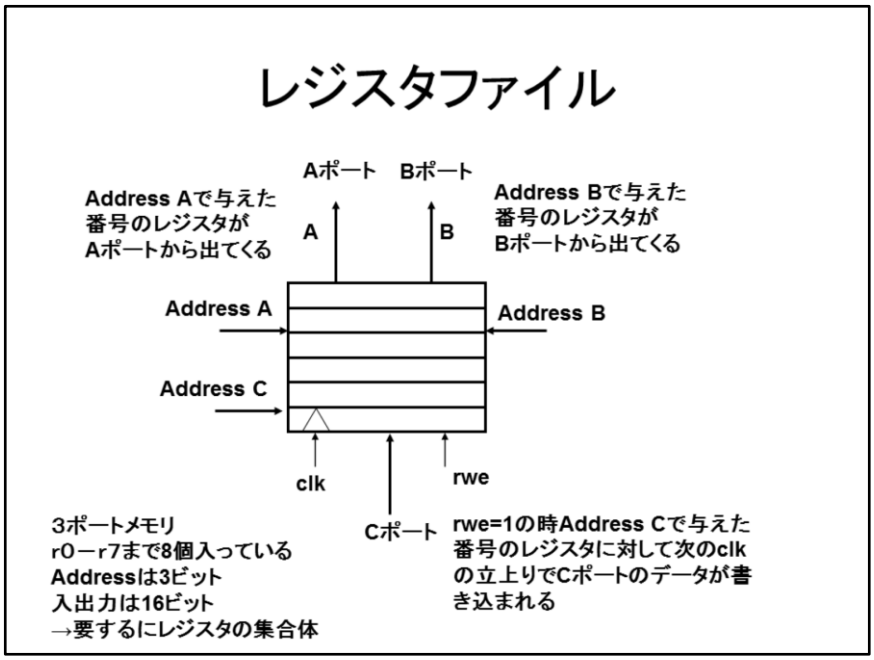
このようなプログラムでは、いちいち数字をレジスタに入れるのが面倒ですが、ステップ数が多くなるのを厭わなければ問題なくできます。



命令のフィールドを決める方法としてはI形式とR形式があります。I形式は8ビットの直値を伴うもので、R形式はopcodeは全て0で、funct形式を利用して命令を識別します。



POCOの構造はこのような図になります。これを今回はVerilogで記述していきます。



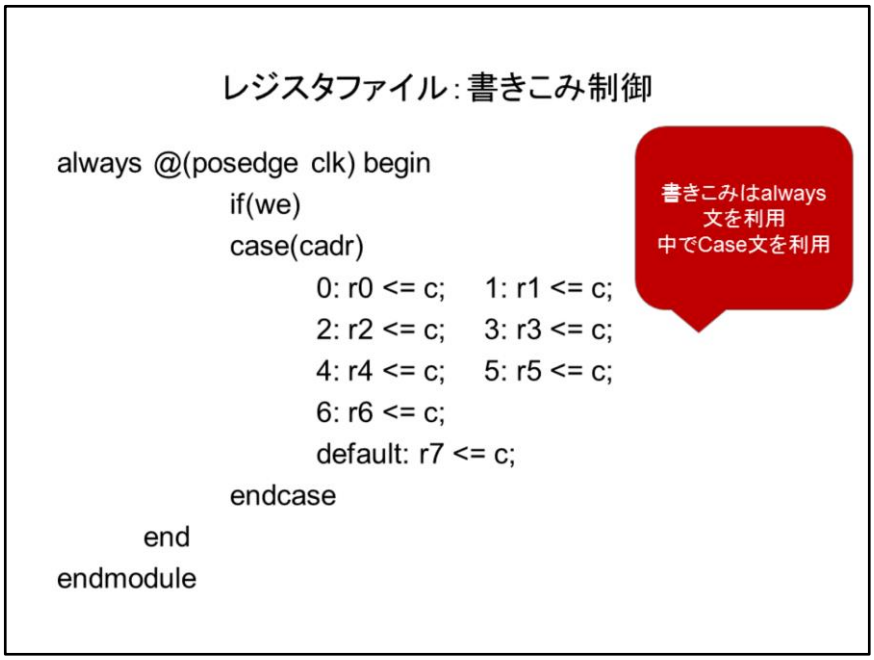
ではまずレジスタファイルの記述から始めましょう。A,Bポートは読み出し専門、Cポートは書き込み用です。レジスタファイルも、メモリ同様、出来合いの回路であるIP(Intellectural Property)で実装する場合がありますが、ここではレジスタ数が少ないので、合成の対象としたいと思います。ここで使うレジスタファイルは、読み出し専用のAポート、Bポートと書き込み専用のCポートを持つ2R1Wの3ポートメモリです。読み出しはアドレスに対応したデータがA,Bポートから直接表れます。書き込みについてはrwe=1の時にCポートに置いた入力データをクロックの立ち上がりに同期してアドレスCに対して書き込みます。

レジスタファイル：rfile.v

```
`include "def.h"
module rfile (
  input clk,
  input [`REG_W-1:0] aadr, badr, cadr,
  output [`DATA_W-1:0] a, b,
  Input [`DATA_W-1:0] c,      input we);
   reg [`DATA_W-1:0] r0, r1, r2, r3, r4, r5, r6, r7;
   assign a = aadr == 0 ? r0:  aadr == 1 ? r1:
              aadr == 2 ? r2:  aadr == 3 ? r3:
              aadr == 4 ? r4:  aadr == 5 ? r5:
              aadr == 6 ? r6: r7;
     assign b = badr == 0 ? r0:  badr == 1 ? r1:
              badr == 2 ? r2:    badr == 3 ? r3:
              badr == 4 ? r4:    badr == 5 ? r5:
              badr == 6 ? r6: r7;
```

読み出しは組み合わせ回路マルチプレクサ文を利用

レジスタファイルのVerilog記述を示します。ここでは8つのレジスタを別々に宣言しています。これはメモリの形で宣言しても良いのですが、これをやるとgtkwaveで観測できなくなります（メモリの中身は普通vcdファイルに保存しません。これをやるとファイル量が巨大化するためです）。別々に宣言しておけば、見ることができて便利です。
さて、読み出しについては条件選択文を使っています。アドレスに対応したレジスタをそれぞれのポートに出力します。



次に書き込みです。ここでは、レジスタへの書き込みなので、always文を使います。リセットはしたいところですが、レジスタは一種のメモリなので、ここはしないことにします。書き込みイネーブルwe=1の時書き込みます。ここで、どのレジスタに書き込むのかを選ぶのにcase文を使っています。case文はC言語のswitch文と似ていて、（ ）内の値に応じた処理を行います。どれも成り立たない場合はdefault以下の文が実行されます。条件選択文と違ってdefaultは省略可能ですが、きちんと記述することをお勧めします。

Case文

```
case (信号、レジスタ)
  値1:  文1
  値2:  文2
    …
  default: 文n
endcase
```

( )内に書いた信号、レジスタの値が
値1ならば文1、値2ならば文2、、、どれでも
なければ文nが実行される

Case文の文法をまとめておきます。それぞれの文はbegin .. endの書き方を使って複数の文を書くことができます。case文の終わりはendcaseです。

Case文の制約

- if文同様、always文の中にだけ書くことができる。
- このため文に書けるのはレジスタへの値の書きこみのみ
  - なぜか？レジスタは値が記憶できるので、全ての条件が尽くされなくても中身が確定するから
- C言語のswitch文に似ている。(Pascalのcase文に準拠している)

case文は大変便利な書き方なので、色々な場所で使いたいのですが、if文と同じくalways文の中だけに使うことができます。すなわち、この文の左辺は、レジスタのみで、文中にはレジスタへの値の書き込みだけです。なぜこのような制約があるのでしょうか？if文同様、レジスタは値が記憶できるので、全ての条件が尽くされなくても、中身が確定するためです。繰り返しますが、レジスタへの値の書き込みは、通常のプログラム言語における変数への代入と大変似ているのです。

注意！

レジスタファイルはメモリ構文を使っても書ける

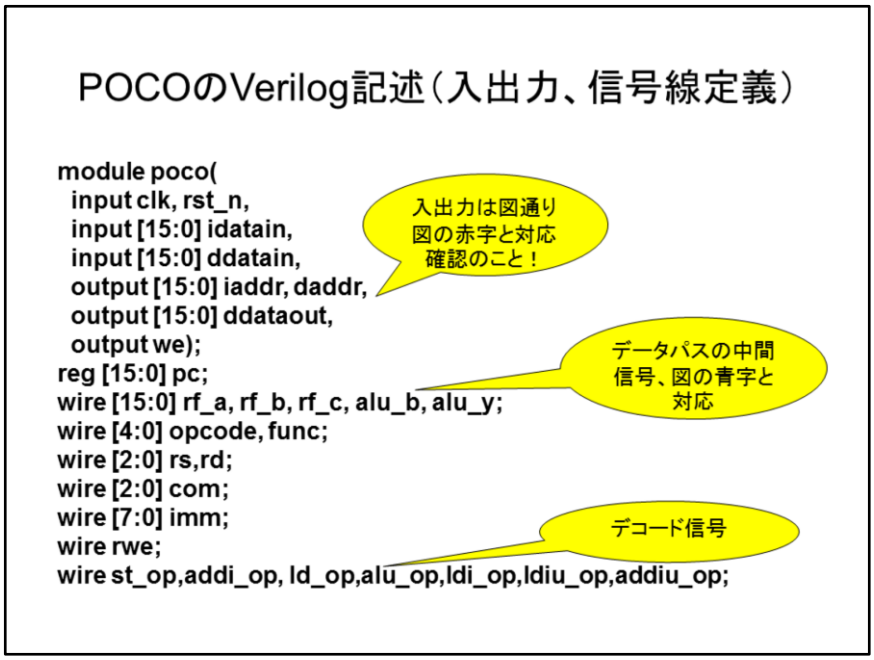
reg [15:0]　r [0:7]

　r[0], r[1],…r[7]として扱う

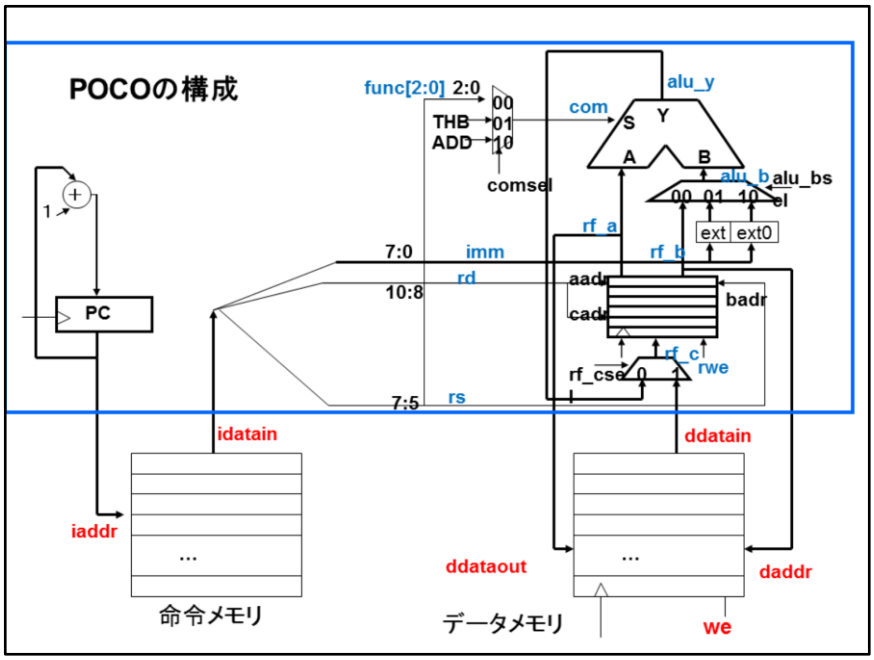
しかし、そうするとgtkwaveで中身が見れなくなる。

このためこの授業ではわざわざ分けて書いたが、通常はメモリ構文を使うのが良い

ちなみに、レジスタはメモリ構文を使ってもかけます。この場合、r[0],r[1]…という書き方をします。実はこの記法の方が普通なのですが、gtkwaveで中身が見れないと困るので、ここでは敢えてレジスタとして分けて宣言しています。

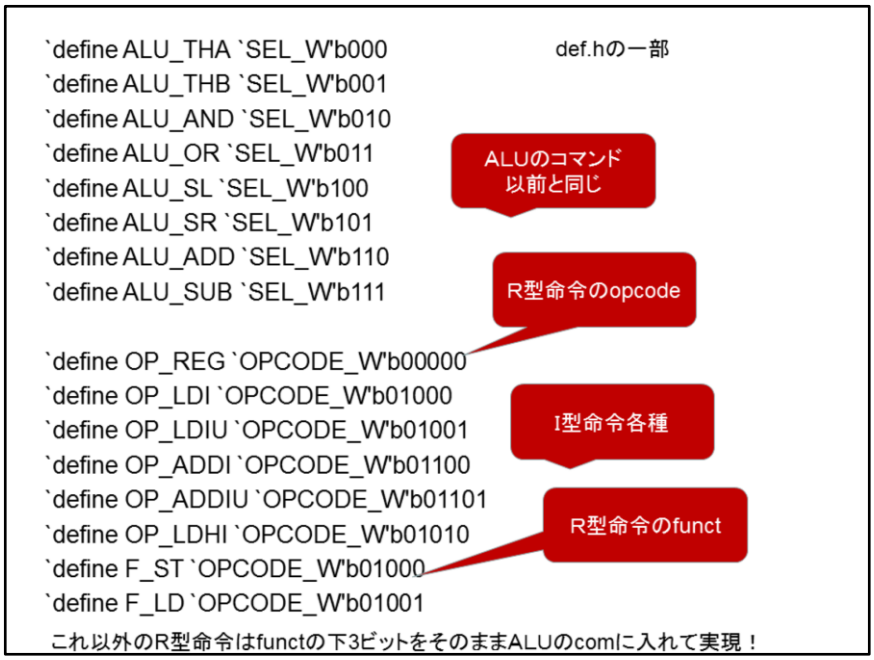


では、POCOのVerilog記述を見て行きましょう。アキュムレータマシン同様、命令メモリとデータメモリはCPUの範囲に入れません。入力はおなじみのclk, rst_n、命令メモリからの入力はidatain、命令メモリのアドレスはiaddrとしています。データメモリからの入力はddatain,書き込み用の出力はddataout、アドレスはdaddrです。書き込みイネーブル信号はweでこれが1の時のclkの立ち上がりで、ddataoutに出力した値がメモリに書き込まれます。アキュムレータマシン同様pcをレジスタで宣言します。次にデータパスの中間信号を定義します。これは図の青字と対応させてください。やはりアキュムレータマシン同様、デコード信号を定義します。これは対応する命令がフェッチされたときに1になるようにします。

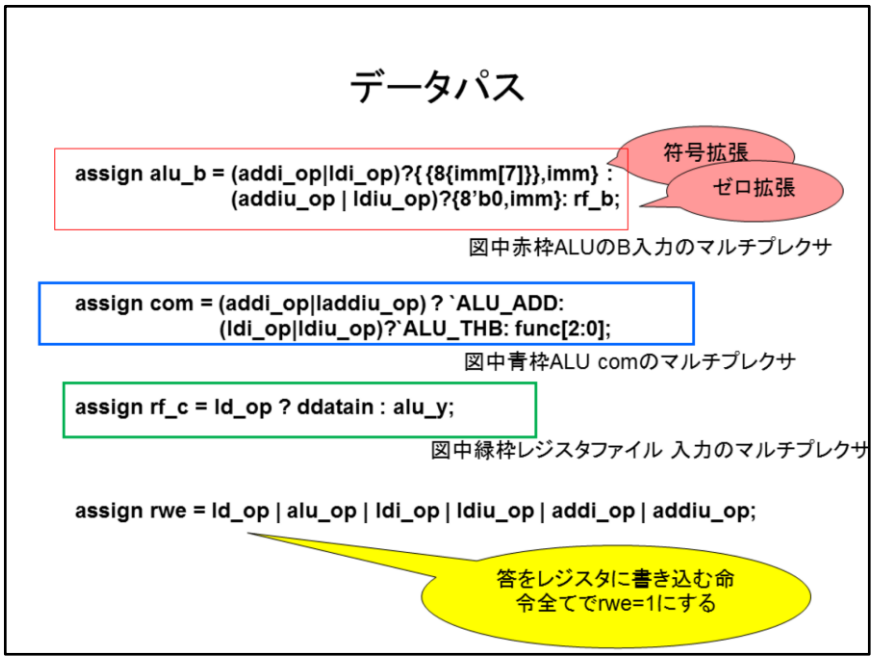


POCOの構成を図に示します。信号線の名前をVerilog記述と対応してください。

まず、ddataoutにはST命令で値を書き込むポートなので、レジスタファイルのAポートrf_aをそのまま繋ぎます。一方、アドレスは、レジスタファイルのBポートからの出力rf_bをつなぎます。これでレジスタ間接指定ができます。命令メモリのアドレスにはpcをそのままつなぎます。読んで来た命令idatainはopcode,rd,rs,funcの並びに分解されます。一方、下位8ビットはイミーディエイトが入る場所なんでimmとして8ビットを切り出します。次にst_opからaddiu_opまで、それぞれの命令に対応した信号線がHレベルになります。ここで、R型の場合、funcで命令を識別しているのが分かります。ちなみにalu_opは、ALU命令全体をカバーします。これはfuncの上位2ビットが00であることにより識別します。



定義が並んでいるdef.hの内容を示しましょう。I型命令はopcode、R型命令はfuncの4ビットと比較します。R型命令のopcodeは0000です。



ではデータパスの記述を確認します。ここで、alu_bはALUのB入力に入れるためのデータを選択します。ADDIとLDIでは符号拡張、ADDIUとLDIUではゼロ拡張した結果を使います。これらの命令以外ならば、レジスタ間演算命令なので、レジスタファイルのBポートからの出力を使います。
次にALUのcomは、ADDIとADDIUの時は加算命令(110)、LDIとLDIUではB入力のスルー（OO1）が入るようにします。それ以外の場合はfuncの下位3ビットが入り、それぞれの演算を行います。
rf_cはレジスタファイルの入力データポートで、LD命令の時はメモリからのデータ入力ddatainが入り、その他の場合は、ALUの出力が入ります。
rweはレジスタへの書き込み信号なので、レジスタに書き込む命令のORをとります。

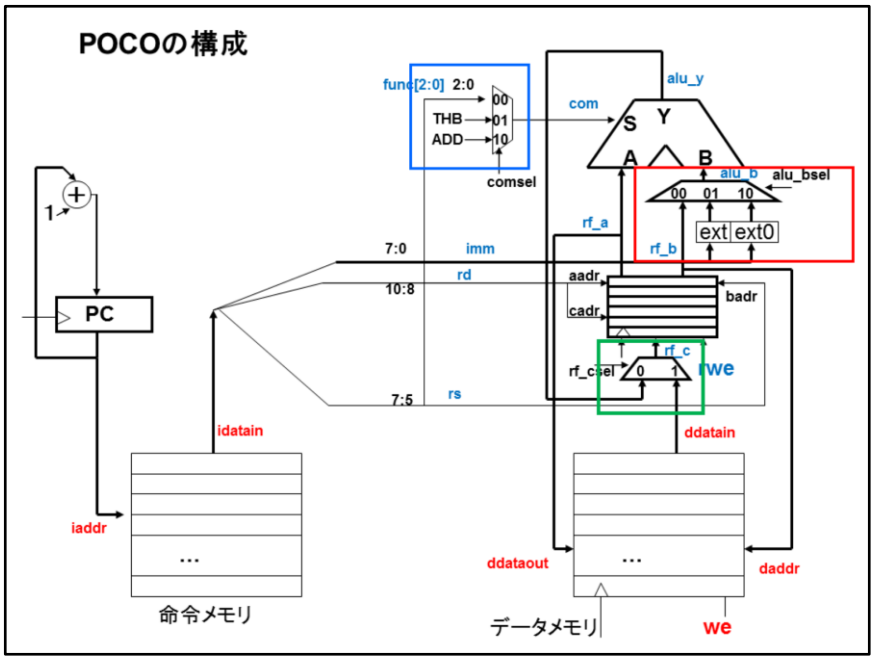
符号拡張とゼロ拡張

n{x}はxをn回繰り返して並べることを意味する

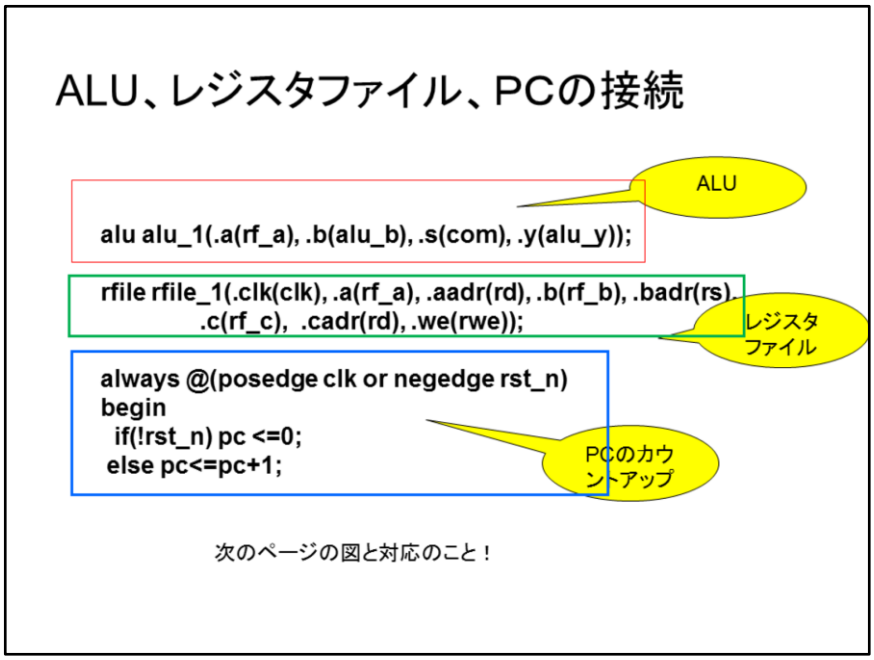
{ {8{imm[7]}},imm} → 符号ビットを8ビット並べ、immと連結→ 符号拡張

{8'b0,imm} → 0を8個とimmを連結→ ゼロ拡張

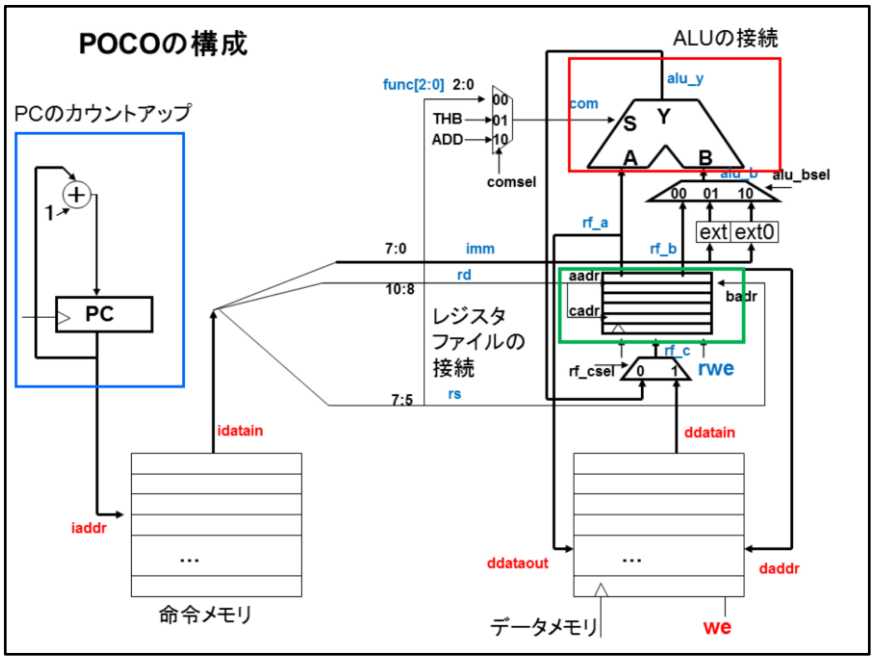
符号拡張とゼロ拡張の書き方を復習しましょう。符号ビットを8個並べて元の数とくっつけます。



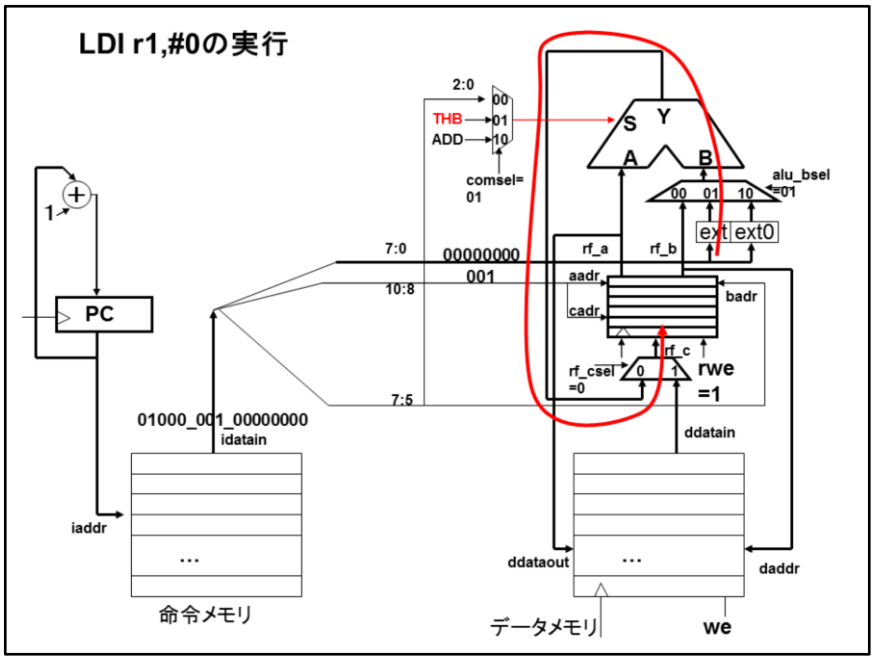
では、POCOの構成と、それぞれの記述の対応を考えましょう。青い四角、赤い四角、緑の四角がそれぞれの条件選択文と対応しています。2ページ前のスライドと対応を考えてください。



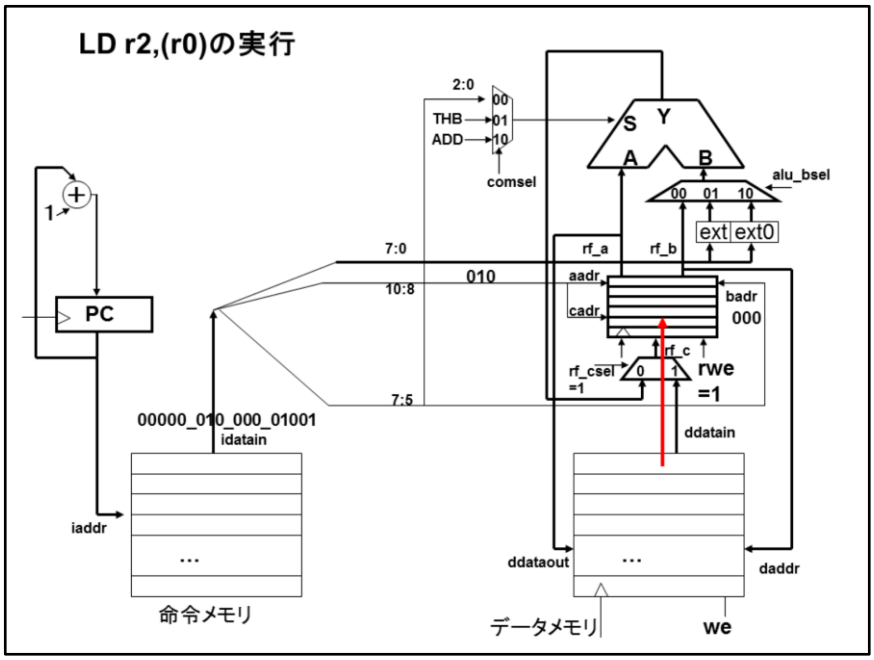
ALU、レジスタファイルを実体化している部分は、今まで定義した信号名を使います。always文中ではPCのカウントアップをしています。これは、アキュムレータマシンと同じです。



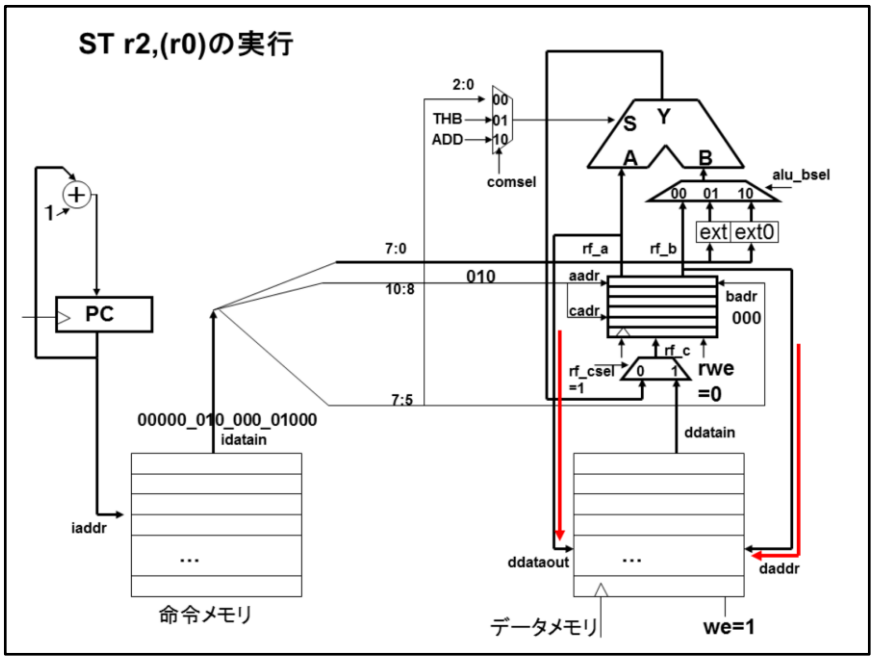
赤い四角と緑の四角を見てください。これが前のページの記述と対応します。



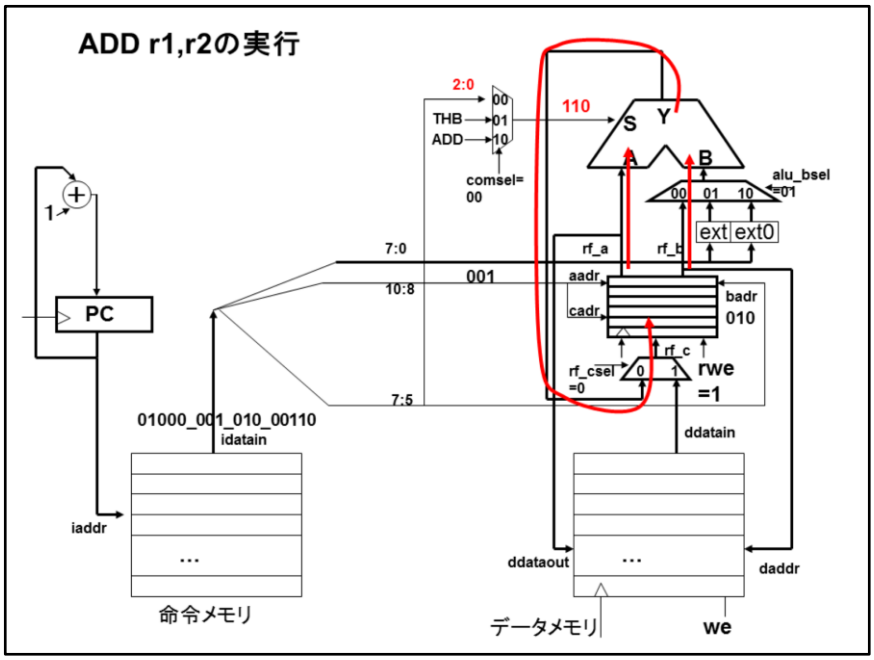
では、どのように各部が動くか、見てみましょう。LDI r1,#0では、命令の下位8ビットを符号拡張します。このため、alu_bは01にします。comは、comselを01にしてTHBを入れてやります。この0をrf_cselを0にしてレジスタファイルのC入力に入れます。rwe=1にして書き込みます。



LD命令の場合は、rf_bがそのままデータメモリのアドレスに入るので、ddatainから読み出されたデータが入ってくるので、rf_cselを1にしてメモリからのデータをレジスタファイルのC入力にいれてやります。rwe=1にして書き込みます。



ST命令はrf_aをddataoutに出してやり、we=1にしてデータメモリに書き込みます。
daddrには、rf_bが入っており、これはLD命令と同じです。今度はレジスタファイルに書き込まないのでrwe=0にします。



レジスタ同士の演算はどうなるでしょう？この場合、funcコードの下3ビットをALUのSに入れてやります。このためにcomsel=00にします。alu_bselには00を入れてレジスタファイルのBポートの出力を通してやります。計算結果をレジスタファイルに書き込むため、rf_csel=0としてrwe=1とします。

R型命令一覧

NOP		00000------00000
MV rd,rs	rd← rs	00000dddsss00001
AND rd,rs	rd← rd AND rs	00000dddsss00010
OR rd,rs	rd← rd OR rs	00000dddsss00011
SL rd	rd← rd<<1	00000ddd---00100
SR rd	rd← rd>>1	00000ddd---00101
ADD rd,rs	rd← rd + rs	00000dddsss00110
SUB rd,rs	rd← rd - rs	00000dddsss00111
ST rd,(ra)	(ra)← rd	00000dddaaa01000
LD rd,(ra)	rd← (ra)	00000dddaaa01001

R型の命令をまとめます。opcodeは00000でfuncで識別する点に注意ください。funcの上位2ビットが00の場合、下の3ビットをALUのコマンドに入れます。

I型命令一覧

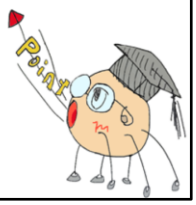
LDI rd,#X	rd← X(符号拡張)	01000dddXXXXXXXX
LDIU rd,rs	rd← X(ゼロ拡張)	01001dddXXXXXXXX
ADDI rd,#X	rd←rd+X(符号拡張)	01100dddXXXXXXXX
ADDIU rd,#X	rd←rd+X(ゼロ拡張)	01101dddXXXXXXXX
LDHI rd,#X	rd←{X,0}	01010dddXXXXXXXX

I型の命令の一覧です。opcodeで命令を識別します。

本日のまとめ

- Verilog　HDLでPOCOを記述する場合、全体の構造と記述の文を対応付ける
- レジスタファイルとALUは別モジュール
- always文の利用はpc周辺
- Verilogの新しい構文はcase文　always文の中だけで使える

```
case (信号、レジスタ)
  値1:　文1
  値2:　文2
    …
  default: 文n
endcase
```



インフォ丸が教えてくれる今日のまとめです。

演習6-1

- LDHI（opcode: 01010)を実装せよ
  - 上位8ビットにImmediateの数字を入れ下位を0にする命令（解説次ページ）
  - 提出物はpoco.v
  - ヒント:
    - op_ldhiを作ってデコードする（def.hに定義してある）
    - ALU B入力のマルチプレクサを拡張
    - rweにも加えるのを忘れずに！
    - test_poco.vのreadmemb(“imem.dat”,imem)を(“ldhitst.dat”,imem)に変更してテスト
    - r0が0x5500になればOK

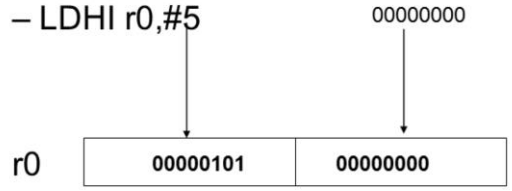
では演習をやってみましょう。LDHIを実装します。この命令は上位8ビットにイミーディエイトの数字を入れ、下位8ビットを0にします。LDI命令では下の8ビットしか入らないので、LDHIは上位8ビットに値を入れる際に便利です。

LDHI（Load High Immediate)

大きな値をレジスタに直値で入れる命令

LDHI（opcode: 01010)

- 上位8ビットにImmediateの数字を入れ下位を0にする
- LDHI r0,#5



今opcodeを01010にします。もちろんこの命令はI型です。この命令はセコイ感じもしますが、便利なので、全てのRISCが持っています。

演習6-2

ORI（opcode:01111)を実装せよ

- ori rd, #XX　0111_ddd_XXXXXXXX
- ori r1, #0x55　r1 ← r1 | 0x0055
- immediateはゼロ拡張せよ
- 提出物は poco.v
- ヒント:
  - ori_opを作ってデコードする（def.hに定義がないので自分でやる）
  - ALU B入力のマルチプレクサを拡張
  - rweにも加えるのを忘れずに！
  - test_poco.vのreadmemb(“imem.dat”,imem)を(“oritst.dat”,imem)に変更してテスト
  - r1が0xfになればOK

次の演習はORIを実装することです。ORIはレジスタとイミーディエイトデータの論理和を計算します。やり方はADDIと同じですが、イミーディエイトはゼロ拡張します。これは論理和などの論理演算を符号付き数に使うことはまずないためです。