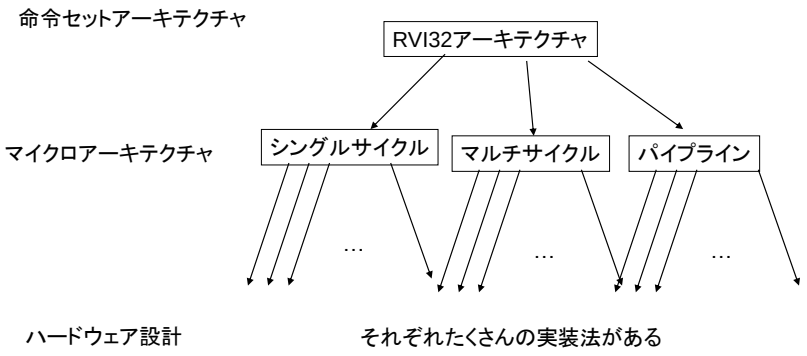
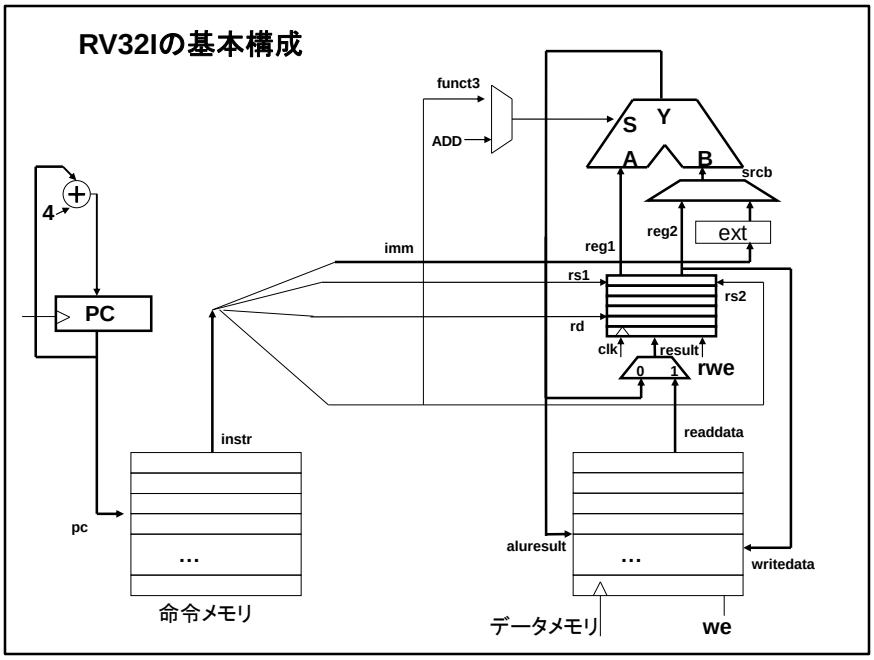
計算機構成　第6回
RV32Iのマイクロアーキテクチャ

天野　hunga@am.ics.keio.ac.jp

マイクロアーキテクチャ

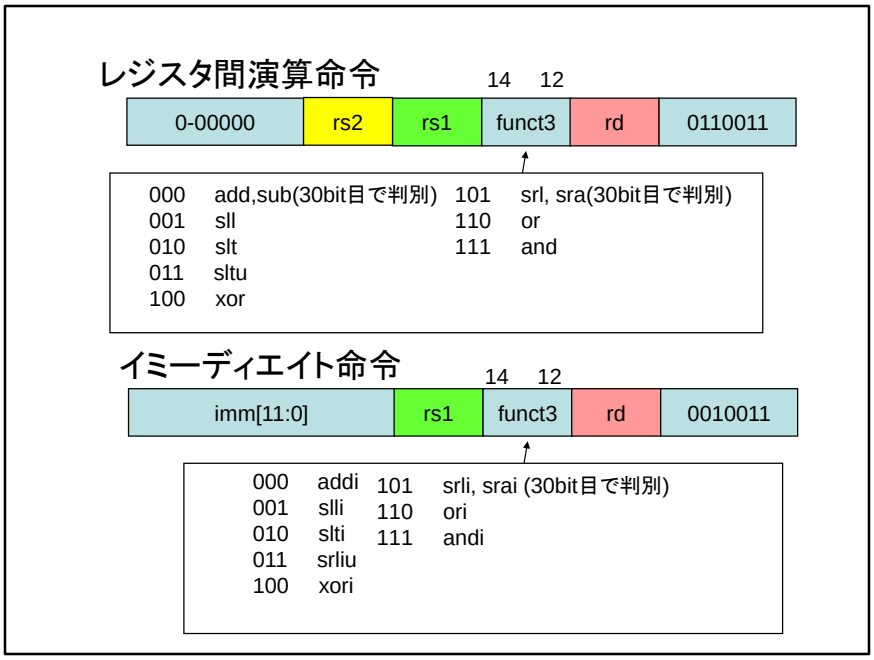


同じ命令セットでも様々な実装法があります。どのようにCPUを実現するかを決めるのがマイクロアーキテクチャです。ここでは、RV32Iのマイクロアーキテクチャを紹介します。まずは、一番簡単なシングルサイクル実装を紹介しましょう。



では、RV32Iの構成をざっと説明します。PC周辺は詳細化していないので、これだと先に進むだけになります。主要なモジュールはALUとレジスタファイルです。
ALUのA入力はレジスタファイルのAポートに直接繋ぎますが、Bポートは直値を命令から持ってくるために、マルチプレクサが入っています。
RV32Iは、イミーディエイト命令に、符号付きと符号無しを持っているので、命令の直値部分を符号拡張、ゼロ拡張をした結果と、レジスタファイルからのB入力をマルチプレクサで切り替えられるようにします。ここで、extは符号拡張用のハードウェアで、これはMSBを複製したり、ゼロを入れたりすると共に配線を入れ替えたります。中身はさほどハードウェア量の多いものではないです。
ALUのコマンドは、アキュムレータマシン同様、3ビットのfunctフィールドを入れて、レジスタ同士の命令、イミーディエイト命令がそのまま実行できるようになっています。ディスプレースメントを加算するなどの役割のため、A入力、B入力の加算を行うADDをfunct3と切り替えて入れられるようにしています。ALUのY出力は、マルチプレクサを経由してレジスタファイルのCポートのデータ入力に繋ぎ、計算結果を書き込めるようにします。一方で、ディスプレースメントとの加算結果をデータメモリのアドレスに接続します。
レジスタファイルのポートAのアドレスと、ポートBのアドレスには、命令コードのrs1, rs2に相当する部分を入れます。Cポートのアドレスには、rdに相当する部分を入れます。

データ入力にはBポートの出力を入れます。これでlw命令、sw命令を実行します。読み出したデータは、マルチプレクサを経由してレジスタファイルに入れます。
ではまず主要な部品であるALUとレジスタファイルを紹介します。

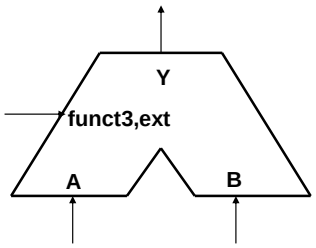


RV32Iの命令コードの特徴は、レジスタの位置が完全に揃っていること、6ビット目から0ビット目までの7ビットのopcodeで基本的な命令を識別し、14ビット目から12ビット目までで大分類の中の細かい種類を識別できる（30ビット目も補助的に使います）ことです。これを利用して、イミーディエイト命令、レジスタ間演算命令の両方で、同じ演算は同じfunct3を使っています。このため普通はこれを利用してALUを設計します。

ALUの拡張

15:12ビット目（funct3）と30ビット目（ext)
で演算を選択するように命令コードができている

funct3	ext	Y
0	0	A+B
0	1	A-B
1	-	A<<B
2	-	slt
3	-	sltu
4	-	A^B
5	0	A>>B
5	1	A>>>B
6	-	A|B
7	-	A&B



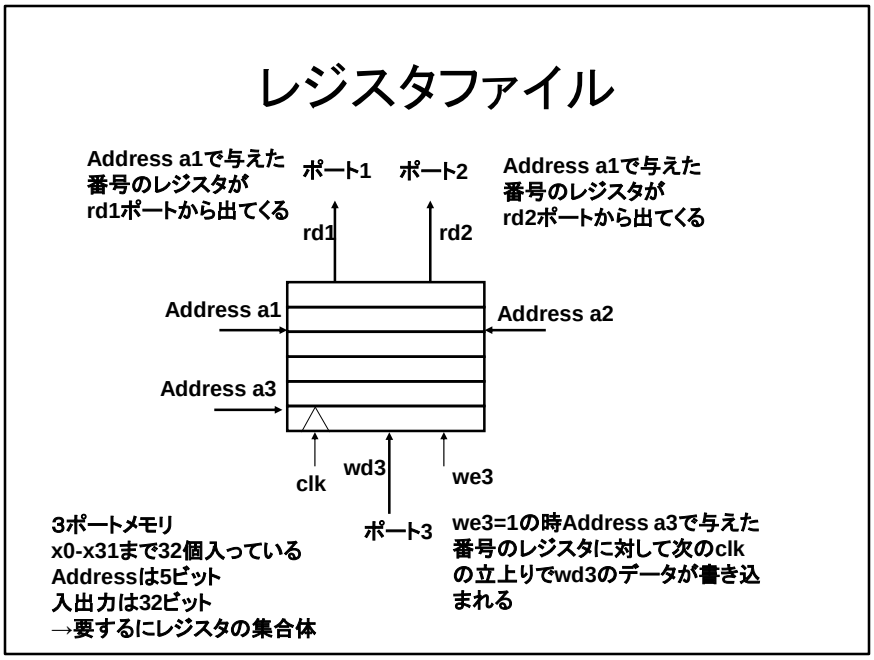
slt (set less than) sltu(set less than unsigned)
比較命令で、小さければ1、そうでなければ0を出力
→後に説明

この点を利用した、ALUの構成を示します。14:12ビット目のfunct3をそのまま演算の選択に利用するこができます。アキュムレータマシンでなかった機能にSet less thanという比較命令があります。この操作はA<Bの時Yから1、そうでないときに0を出します。符号付数と考えて比較する場合と、符号無数として比較する場合があります。

```
module alu (
  input [31:0] a, b,      input [2:0] s,
  input ext,addcom,
  output [31:0] y );
  wire [4:0] shamt;   wire signed [31:0] sa, sb, sy, slt;
  wire [31:0] sltu;   wire [31:0] yy;
  assign sa = $signed(a);    assign sb = $signed(b);
  assign shamt = b[4:0];
  assign sy = sa >>> shamt;
  assign sltu = a <b ? 1:0;
  assign slt = sa <sb ? 1:0;
  assign y = addcom ?  a+b: yy;
  assign yy = s==`ALU_ADD & ext ? a-b:
              s==`ALU_ADD & ~ext ? a+b:
              s==`ALU_XOR ? a ^ b:
              s==`ALU_OR ? a | b:
              s==`ALU_AND ? a & b:
              s==`ALU_SLT ? slt:
              s==`ALU_SLTU ? sltu:
              s==`ALU_SLL ? a << shamt:
              s==`ALU_SRL & ~ext ? a >> shamt:
              s==`ALU_SRL & ext ? sy: 0;
endmodule
```

符号付の比較、符号付の右シフトが必要。このため、wire signed 宣言と、符号付き変換組み込み関数$signedが必要やや面倒になる。ここではあまり深く突っ込まない

ALUの記述はここに示すようにやや複雑になります。これは、符号付の右シフトである算術シフトが必要になることと、符号付の比較が必要になることです。これをVerilogの基本分法で書くと非常に面倒になるため、符号付数宣言wire signedと符号付数変換関数$signedを使いますが、ここではあまり突っ込まないことにします。



次にレジスタファイルについて具体的に紹介します。レジスタファイルはレジスタの集合体で、小規模なマルチポートメモリと考えられます。RV32Iで使うレジスタファイルは、三つのポートを持っています。それぞれが独立のアドレスを持ち、rd1,rd2ポートは読み出しポート、wd3ポートは書き込みポートです。RV32Iではレジスタは32個なのでそれぞれのアドレスは5ビットです。アドレスa1が00011ならばrd1ポートからx3が読み出され、アドレスa2が00010ならばrd2ポートからx2が読み出されます。一方、書き込みは、wd3ポートにデータを与えて、アドレスa3に書き込むレジスタの番号を与え、we3を1として、クロックが立ち上がったときに行われます。we3はwrite enableでこれを1にしたときだけに書き込まれます。この辺はメモリと同じです。レジスタファイルはこの授業ではレジスタの集合体として論理合成してしまいますが、場合によってはメモリ同様に出来合いの回路（IP:Intellectual Property)で実現します。

レジスタファイルの記述(rfile.v)

```
`include "def.h"
module rfile (
 input clk,
 input [`REG_W-1:0] a1, a2, a3,
 output [`DATA_W-1:0] rd1, rd2,
 input [`DATA_W-1:0] wd3,
 input we3);

        reg [`DATA_W-1:0] rf[0:`REG-1];
        assign rd1 = |a1 == 0 ? 0: rf[a1];
        assign rd2 = |a2 == 0 ? 0: rf[a2];
        always @(posedge clk)
                if(we3) rf[a3] <= wd3;

endmodule
```

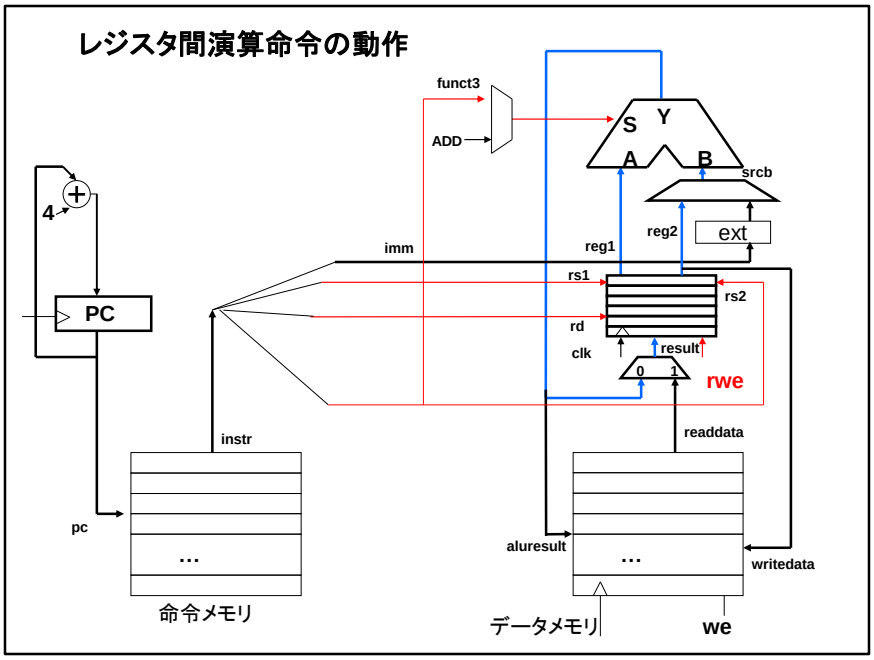
2read/1writeの3ポートメモリ

32個あるのでメモリ宣言している（このためgtkwaveで見れない）

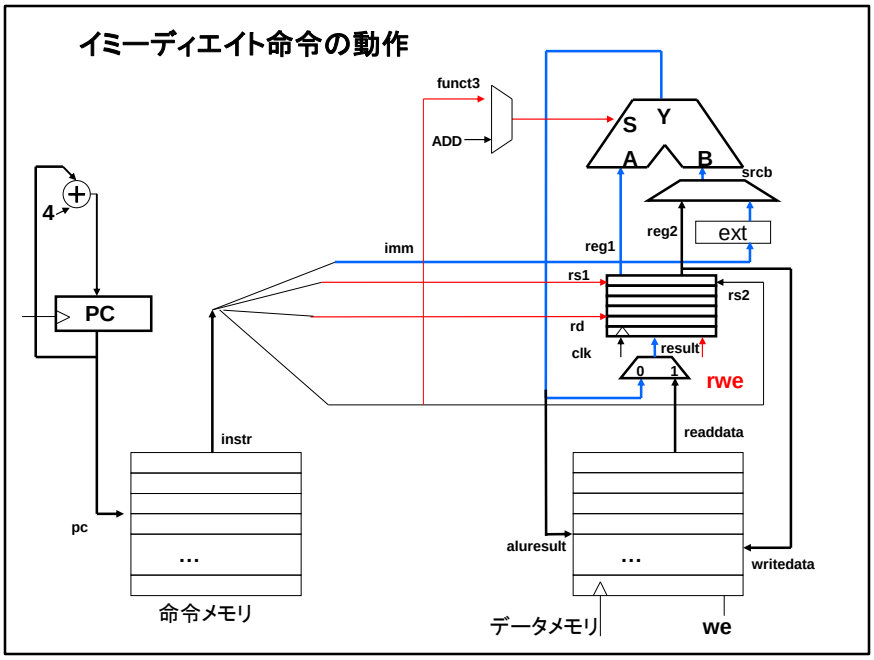
0の場合は常に0

clkの立ち上がりに同期して書き込み

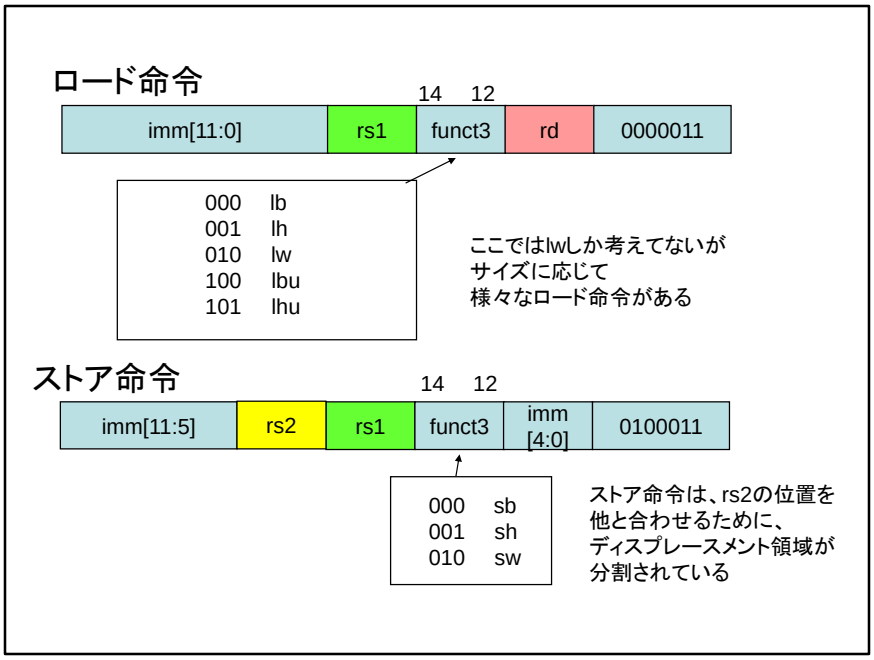
では次にレジスタファイルの記述を示します。レジスタファイルは、32本のレジスタを蓄えておくところで、2read/1writeの3ポートメモリとして記述してあります。ポート名は図に準拠してあります（ただし小文字です）。ポート1，2が読み出し、ポート3は書き込みです。rd1,rd2からはa1,a2で選んだ番号のレジスタが読み出されます。レジスタ0からは常に0が読まれるようになっています。ここではクロックと関係しないので、条件選択文が使われています。書き込みはwe3が1の時だけ、wd3の値をa3で選んだ番号のレジスタに書き込みます。ここではメモリ記述を使いますので、gtkwaveで中身を見ることができません。



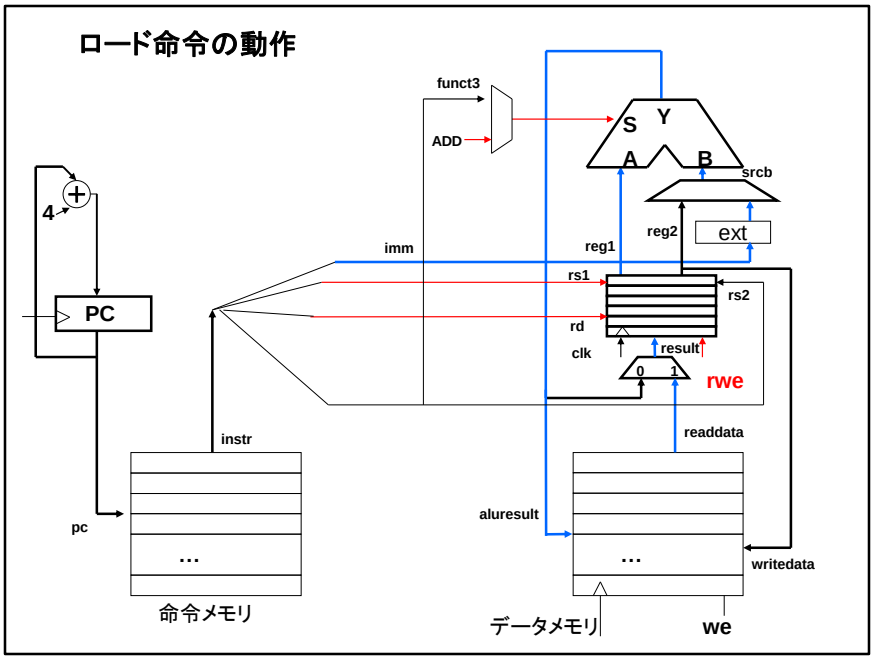
レジスタ間演算命令の動作は図のようになります。プログラムカウンタに従って命令メモリからフェッチ（取ってきた）した命令は、rs1, rs2, rdがそれぞれのレジスタファイルのアドレスに送られます。reg1, reg2二つのポートから出力された値を片方は直接ALUのA入力へ、もう片方は、B入力に入れます。funct3によって演算が選択されて、答えはレジスタファイルへ書き込まれます。このためにwre=1にする必要があります。



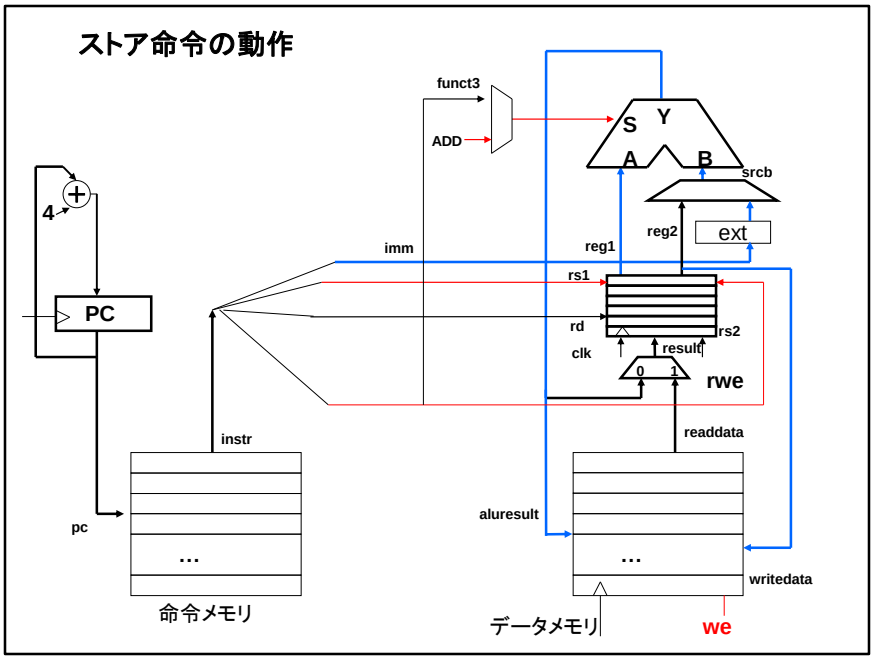
イミーディエイト命令の場合、rs1, rdのみがレジスタファイルに送られ、reg1のポートから出力された値を直接ALUのA入力へ入れます。もう片方は、命令コード中の12ビットのイミーディエイトを符号拡張してB入力に入れます。レジスタ間演算命令と同じでfunct3によって演算が選択されて、答えはレジスタファイルへ書き込まれます。このためにwre=1にする必要がある点も同じです。さて、この時、rs2にはイミーディエイトの一部が送られ、意味のないレジスタの値がreg2から出力されるのですが、これはマルチプレクサで選ばれないため、意味のある動作をしません。このような動きは省略して考えます。



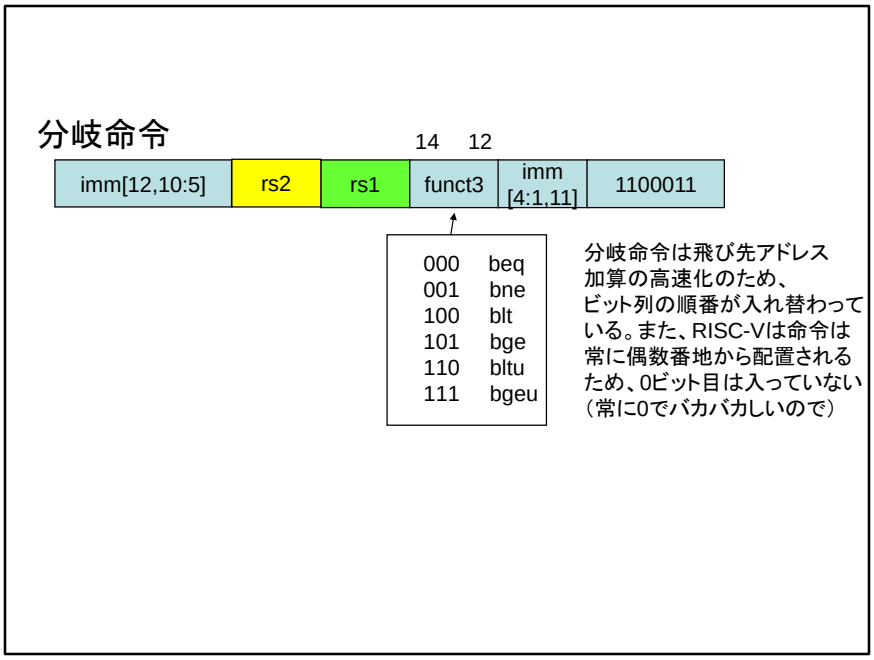
次にロード命令とストア命令の動作を考えましょう。ロード命令の構成は、イミーディエイト命令と同じで、opcodeで識別します。実はlw以外にもlb,lh,lbu,lhuなどのロード命令があるのですが、ここではlwのみを考えます。ストア命令は、rs2のフィールドを他の命令と揃えたため、ディスプレースメントを示すフィールドが左右に分離されています。



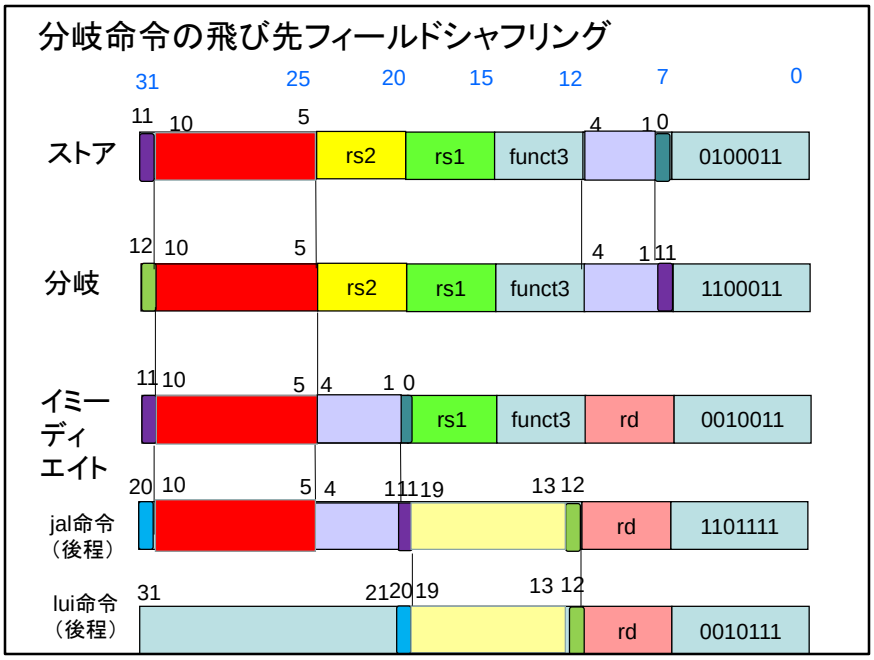
ロード命令の場合も、rs1, rdのみがレジスタファイルに送られ、reg1のポートから出力された値を直接ALUのA入力へ入れます。もう片方は、命令コード中の12ビットのイミーディエイトを符号拡張してB入力に入れます。これがディスプレースメント値に相当します。ロードの場合はアドレス計算を行うために、ALUには加算の指示を与えます。この場合、答えは、メモリのアドレスになりますが、これは既に接続されていますので、メモリからはロードすべきデータが読み出されてreaddataに表れます。これをレジスタファイルの入力マルチプレクサで選択して、レジスタファイルへ書き込まれます。このためにwre=1にする必要があります。



ストア命令の場合も、rs1, rs2がレジスタファイルに送られ、reg1のポートからアドレス、reg2のポートから書き込むデータが読み出されます。アドレスの方は、ALUのAポートに送られ、命令コード中のイミーディエイトと加算されます。ここで、今までのロード命令、イミーディエイト命令と違って、イミーディエイトの値が分割されているため、くっつける必要があります。このため、この符号拡張回路extはややごちゃごちゃするため、ここでは省略してあります。Bポートから読み出された書き込みデータはwritedataとしてメモリのデータ入力ポートに送られます。ここでwe=1としてデータを書き込みます。ストア命令ではレジスタファイルには値を書きこまないので、rwe=0としなければならないです。

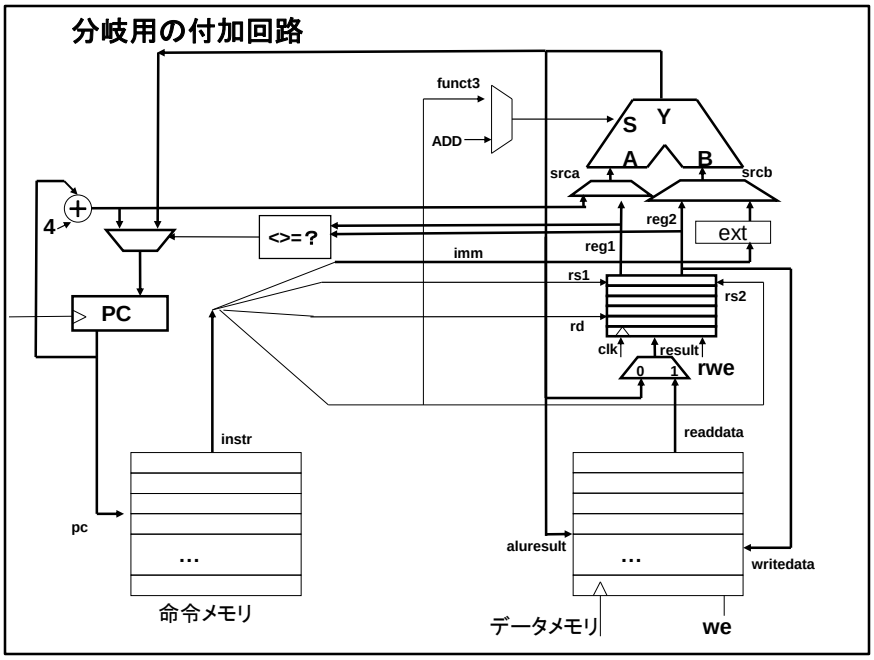


分岐命令もrs2の位置を揃えるため、飛び先を示すイミーデエイト領域が分離されていますが、さらに飛び先番地の高速化のためにビット列の順序が入れ替わっています。また、RISC-Vの命令は必ず偶数番地から配置されるため、最下位ビットは命令コードに含まれないです。このため、格納するビットは、他の命令と全然互換性がありません。さて、この分岐命令は、これまでのデータパスでは実現できず、付加回路を付け加える必要があるのですが、他の命令と違って分岐命令は、一つの命令で二つやることがあります。①イミーディエイト領域をPCと加算して飛び先を計算する。②レジスタの大小、等しいかどうかの関係を調べて分岐が成立するかどうかを決定する。今回の実装では、ALUで①を行い、別の回路を設けて②をやることにします。しかしこれは逆もアリです。

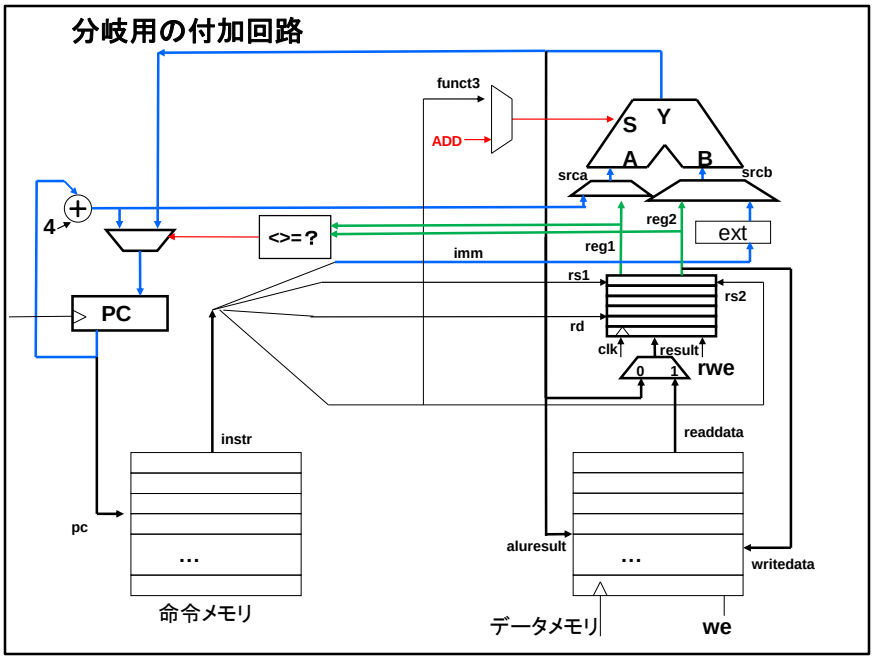


分岐命令の飛び先を示すフィールドはシャッフルされていますが、これによりストア命令、イミーデエイト命令と12ビット目、11ビット目を除いてビット位置が同じになっています。

これを利用すると、イミーディエイトフィールドのマルチプレクサが簡単化されます。最も迫害されているのはビット11で、分岐では7、ストア、イミーディエイトでは31、jal(後に出てくる）では20に割り当てられています。これは3入力マルチプレクサが必要です。しかし、他のビットは2入力で済みます。MSBに常に符号ビットが入るという制約の下では、これ以上の方法は多分ないと思います。良く考えられています。



分岐命令用に拡張したRV32Iの構成を示します。RV32Iの分岐命令にはやることが二つあります。レジスタの比較（大小も含め）と、飛び先アドレスの計算です。どちらかにALUを使い、どちらかに専用の回路を設ける必要があります。ここでは、ALUで飛び先を計算することにします。飛び先アドレスを計算するために、PC＋４をALUのAポートに入れるために、マルチプレクサを付けます。B入力のextも、分岐命令のイミーディエイトデータをまとめるために、構造を変更する必要がありますが、この図には表れていません。符号拡張され0を補ったデータがB入力から入ると考えてください。読み出してきた二つのレジスタは、専用の比較器に入力し、大小、等値関係を判定します。分岐命令の成立条件に適合したら、PCの直前のマルチプレクサを切り替えて、飛び先を設定します。

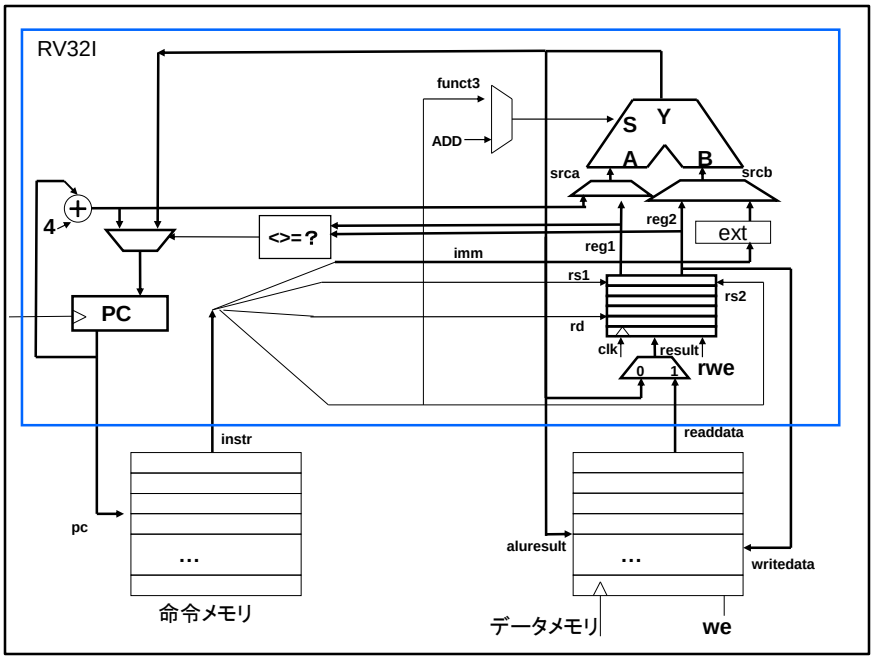


分岐命令のデータの流れを示します。青色が飛び先計算を示します。演算は加算を指定します。緑色のデータパスは、分岐の反転用のデータの流れです。読み出してきた二つのレジスタは専用の比較回路で比較され、結果によって飛び先を設定するかどうかを決めます。

Verilog記述：入出力

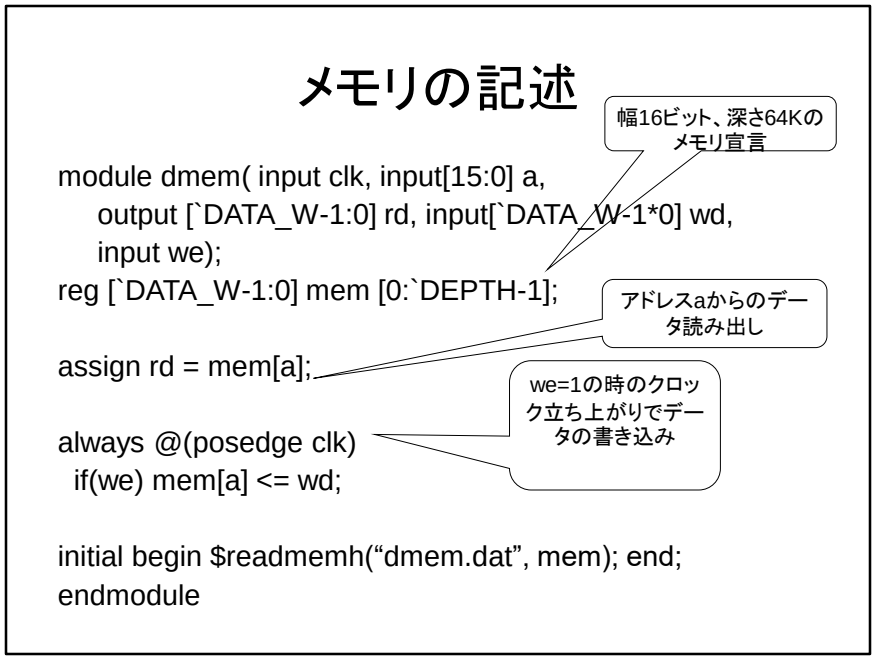
```
module rv32i(
input clk, rst_n,
input [`DATA_W-1:0] instr,
input [`DATA_W-1:0] readdata,
output reg [`DATA_W-1:0] pc,
output [`DATA_W-1:0] aluresult,
output [`DATA_W-1:0] writedata,
output we);
```

では、このRV32IがVerilogでどのように記述されるか見てみましょう。簡単のためいくつかの命令が付いていない版のsimple.tarをダウンロードして解凍してください。まず、入出力について確認しましょう。入出力は、クロック、リセット、命令メモリの入出力、データメモリの入出力からできています。

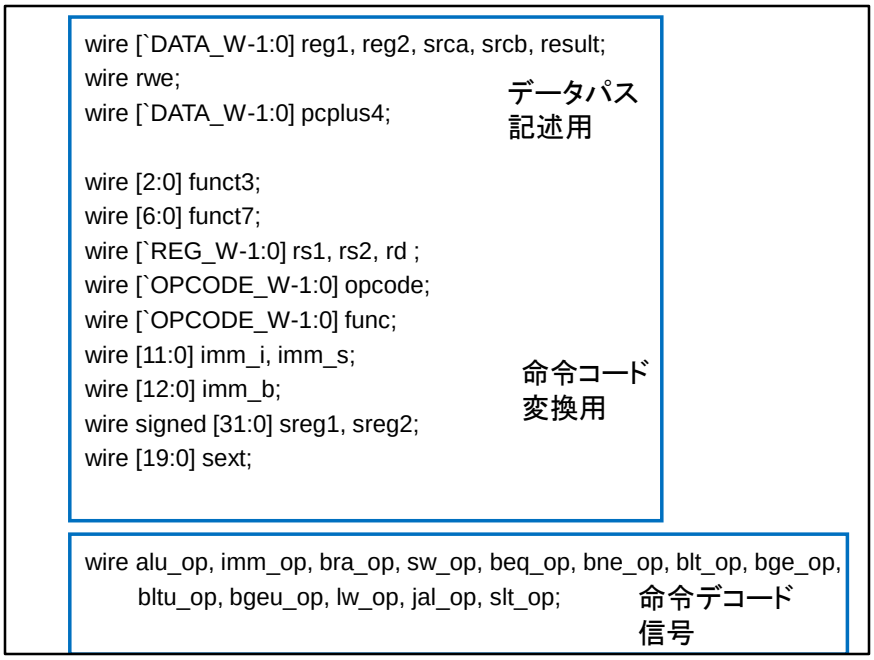


今回の記述も、rv32i.vはこの図の青四角で囲った部分のみです。命令メモリはimem.v, データメモリはdmem.vとして外に出してあります。このインタフェース入出力は前のページに示した通りです。

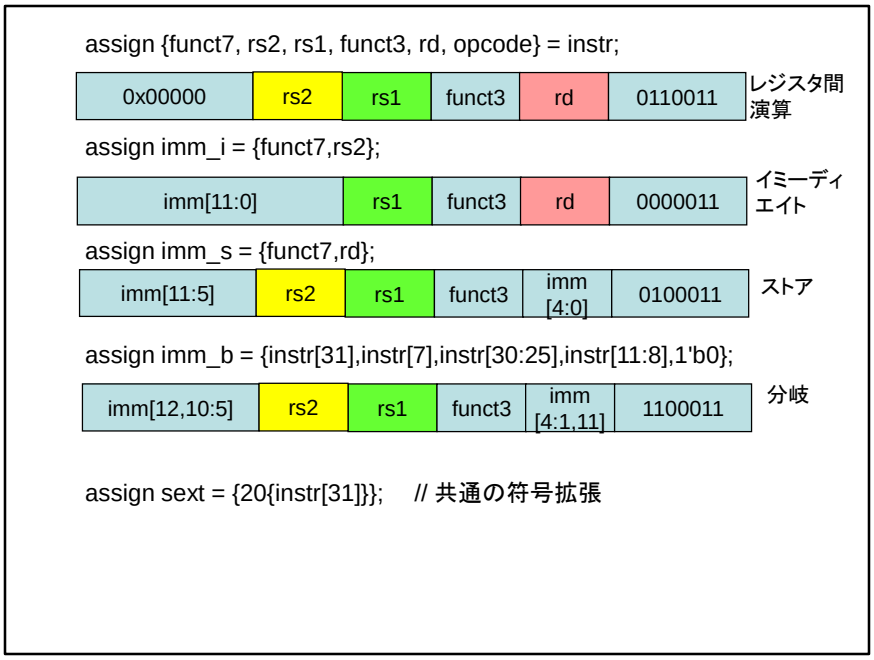
メモリの記述



次はメモリの記述について復習します。今回は命令メモリはimem.vにデータメモリはdmem.vに記述してあり、テストベンチtest_rv32i.vから呼ばれます。ここでは、書き込み機能を含んでいるdmemの方を解説します。メモリの宣言は、reg [MSB:LSB] 「最小アドレス:最大アドレス]で行います。ここでは、16ビットで深さが65536のメモリが宣言されます。最小アドレスは0、最大アドレスは2のn乗にするのが普通です。RV32Iは32ビットのアドレス空間（つまり3G)を持ちますが、演習ではそんなには使わないので、今回は命令、データ共に64Kとしました。このためアドレスは16ビットです。メモリはC言語の配列に似ているので、配列同様[ ]の中に番地を入れて値を取り出します。書き込む場合は、if(we)としてwe=Hの時だけclkに同期して入力を書き込みます。メモリにはファイルに書いてある初期値をあらかじめ設定することが出来ます。このための構文がreadmemh, readmembです。ここでは、dmem.datから16進数で初期値を読み込みます。同様にimem.vも定義されていますが、書き込み機能は持っていません。初期値はimem.datからやはり16進数で入力します。



RISC Vの特徴は命令コードのフィールドが揃っている点で、このことを実現するため、種類が多い点です。これをうまくデコードするために、それぞれ便利なように信号線を設けます。また、どの命令をフェッチしたかを示す命令コード用の信号も設けておきます。



RV32Iの命令は、opcode, rs1,rs2,rdの場所が揃っているので、これは共通にします。補助的なopcodeとして働くfunct3も揃っているのでこれも共通にします。イミーディエイト命令は上位12ビットをイミーディエイトフィールドとして使います。これをimm_iとします。ストア用はrs2の位置を守るためにイミーディエイトが分離されます。これをimm_sに入れます。分岐命令用のオフセット（イミーディエイト）は分離した上に高速化のために位置が入れ替わっています。これは面倒ですが、ビット位置に気を付けてimm_bとします。このimm_bは最下位に0をくっつけるため13ビットになっています。イミーディエイト値を持つ全ての命令で31ビット目は符号フィールドになっています。そこで、ここではこれを32-12=20ビット分符号拡張し、sextとします。これは全てのイミーディエイトにとって上位ビットとして使えます。

RISC-V命令フォーマット

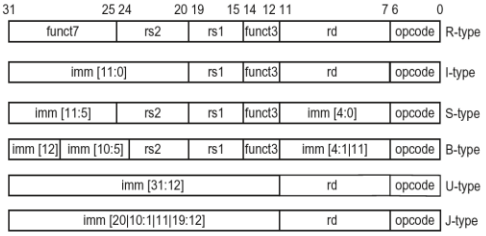


Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

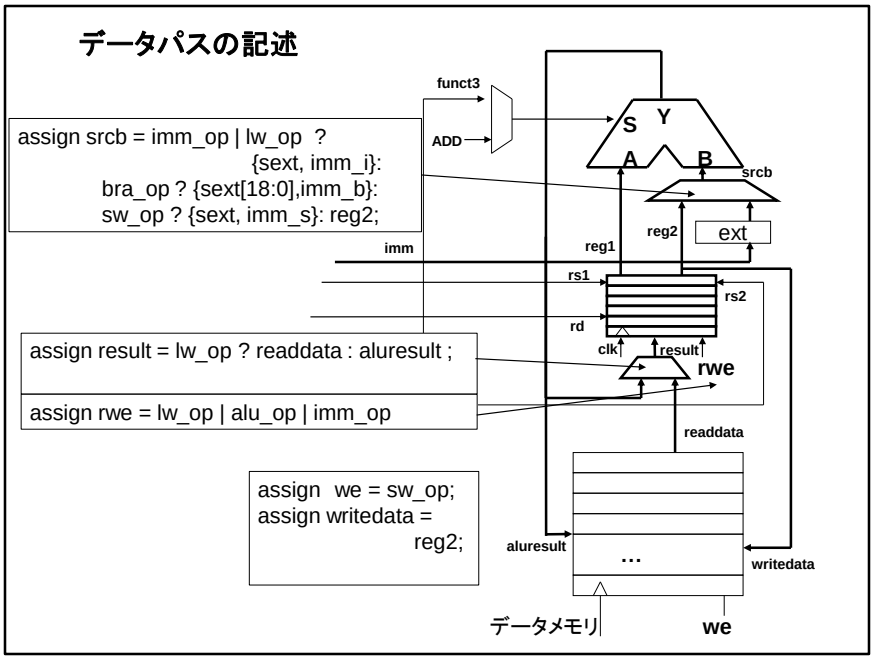
RISC-Vの命令フォーマットは、命令に応じて様々で、R-typeは、通常の演算命令、I-typeはイミーディエイト命令とLoad命令で使われ、イミーディエイトフィールドは12ビットです。イミーディエイトの取り方の違う分岐用にはB-type、Store命令用にS-typeが用意されています。JAL用には遠くまで飛べるJ-type、U-typeは、特殊な命令用です。

```
assign sw_op = (opcode == `OP_STORE) & (funct3 == 3'b010);
assign lw_op = (opcode == `OP_LOAD) & (funct3 == 3'b010);
assign alu_op = (opcode == `OP_REG) ;
assign imm_op = (opcode == `OP_IMM) ;
assign bra_op = (opcode == `OP_BRA) ;

assign beq_op = bra_op & (funct3 == 3'b000);
assign bne_op = bra_op & (funct3 == 3'b001);
assign blt_op = bra_op & (funct3 == 3'b100);
assign bge_op = bra_op & (funct3 == 3'b101);
assign bltu_op = bra_op & (funct3 == 3'b110);
assign bgeu_op = bra_op & (funct3 == 3'b111);

assign ext = alu_op  & funct7[5];
```

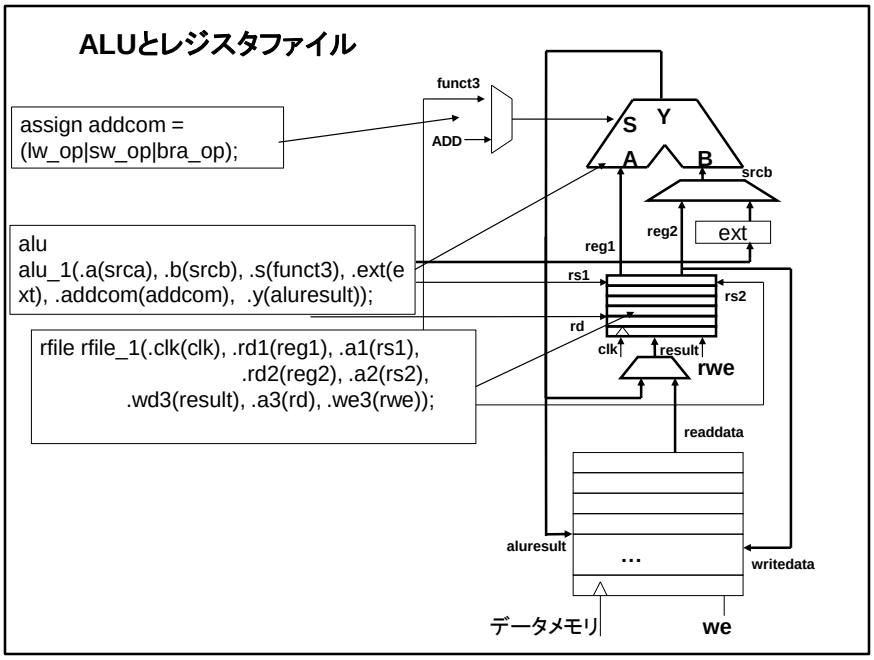
opcodeの
デコード

分岐命令の
デコード

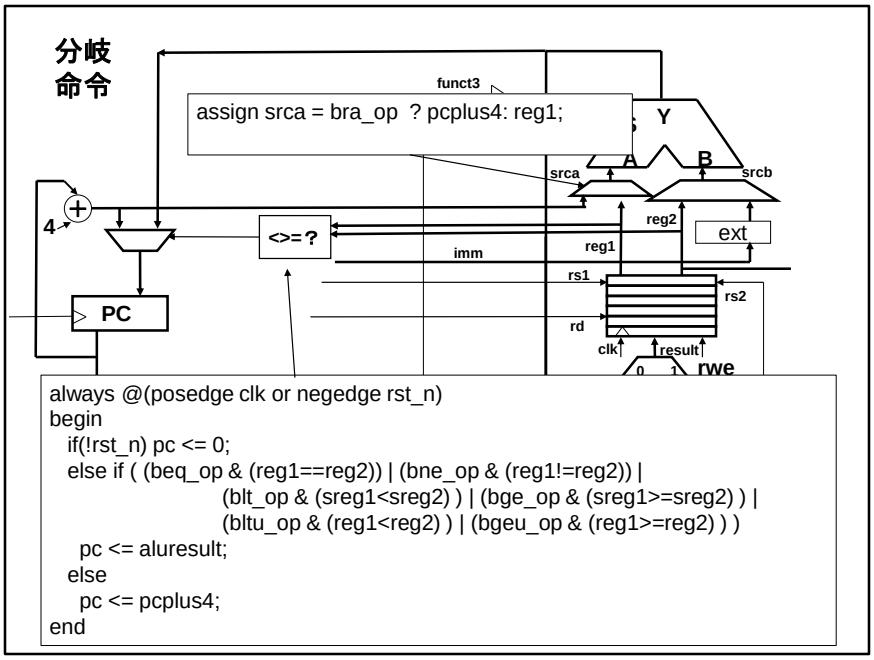
各命令のデコードはopcode部を見てロード、ストア、レジスタ間演算、イミーディエイト、分岐に分類します。このうちロードとストアは現在のところ1種類なので、funct3フィールドまでデコードしてしまいますが、これはあとのためです。分岐命令は細かくデコードします。演算時の拡張ビットはレジスタ間演算命令だけ有効で、上位2ビット目ですのでextというビットを使って示しています。



データパスの記述を示します。ALUのB入力のsrcbは、命令に応じて適切なイミーディエイト値を入れてやり、どれでもなければレジスタのBポートreg2を入れます。レジスタファイルの入力マルチプレクサは、ロード命令ならばメモリから読んできたデータ（readdata)、そうでなければALUの計算結果（aluresult)を選択します。rweは結果を書き込むロード、レジスタ間演算命令、イミーデエイト命令で1になるようにします。メモリの書き込みを制御するweはストア命令の時のみ1、書き込みデータはreg2の値を与えます。



ALUは内部にこの図のマルチプレクサを取り込んでしまっており、addcom=1で加算が行われます。ロード、ストア、分岐命令ではここが1になってメモリアドレスや飛び先アドレスが計算されます。レジスタファイルはrs1,rs2,rdに命令コードの所定の部分がそのまま入ります。



分岐命令の実装にはALUのA入力にpc+4を入れてやる必要があります。このため、srcaという中間信号を設けてマルチプレクサで分岐命令の時にpc+4が入るようにします。結果の判定と分岐の部分は、pcを制御するif文の中に入れます。符号付の比較が必要なので$singed関数で変換したsreg1, sreg2間で比較を行います。分岐が成立するとpcにはALUの演算結果が入るようにしています。

CPUの性能評価式

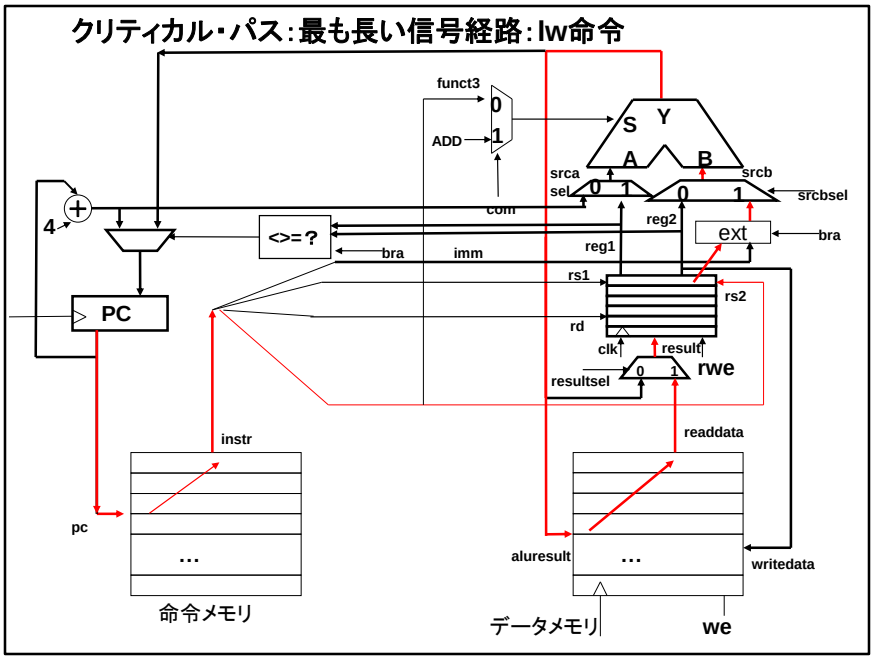
- CPUの性能はプログラム実行時間の逆数

CPU Time＝プログラム実行時のサイクル数×クロック周期
＝命令数×平均CPI×クロック周期
CPI（Clock cycles Per Instruction） 命令当たりのクロック数
→ 1サイクル版では1だがマルチサイクル版では命令によって違ってくる

命令数は実行するプログラム、コンパイラ、命令セットに依存

ここで問題になるのはクロック周期→クリティカルパスによって決まる。

では、次に性能の評価についての一般的な方法を学びます。CPUの性能は、CPUがあるプログラムを実行した際の実行時間の逆数です。実行時間が短い方が性能が高いのでこれは当たり前かと思います。実際のコンピュータではOperating System（OS)が走って実行中にもジョブが切り替わりますが、この影響が入ると困るので、CPUが単一のジョブをOSの介入なしに実行した場合の実行時間（CPU実行時間：CPUTime)を測ります。今まで紹介してきたように、CPUは単一のシステムクロックに同期して動くと考えて良いので、CPU Timeはプログラム実行時のサイクル数×クロック周期で表されます。クロック周期とはクロックが立ち上がってから次に立ち上がるまでの時間で、この逆数がクロック周波数です。プログラム実行時のサイクル数は、実行した命令数×平均CPI（Clock cycles Per Instruction)に分解されます。CPIは一命令が実行するのに要するクロック数で、1サイクル版では全部1ですが、マルチサイクル版では命令毎に違っています。このため、一つのプログラムを動かした場合の平均CPIは、プログラムの種類によって変わります。つまり実行時間の長い命令を多数含んでいるプログラムでは平均CPIは長くなります。もちろんコンパイラにも依存します。



分岐命令用に拡張したRV32Iの構成を示します。RV32Iの分岐命令にはやることが二つあります。レジスタの比較（大小も含め）と、飛び先アドレスの計算です。どちらかにALUを使い、どちらかに専用の回路を設ける必要があります。ここでは、ALUで飛び先を計算することにします。飛び先アドレスを計算するために、PC＋4をALUのAポートに入れるために、マルチプレクサを付けます。B入力のextも、分岐命令のイミーディエイトデータをまとめるために、構造を変更する必要がありますが、この図には表れていません。符号拡張され0を補ったデータがB入力から入ると考えてください。読み出してきた二つのレジスタは、専用の比較器に入力し、大小、等値関係を判定します。分岐命令の成立条件に適合したら、PCの直前のマルチプレクサを切り替えて、飛び先を設定します。

遅延の例

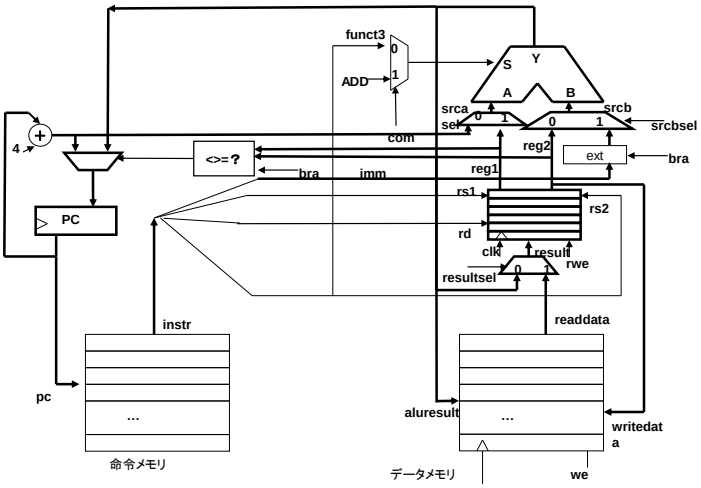
遅延要因	記号	遅延(psec)
レジスタclk→Q	tpcq	30
レジスタセットアップ	tsetup	20
マルチプレクサ	tmux	25
ALU	tALU	200
メモリ読み出し	tmem	250
レジスタファイル読み出し	tRFread	150
レジスタファイルセットアップ	tRFsetup	20

この数値を使うと30+2(250)+150+25+200+25+20=950psec
1.05GHzとなる。

この表は各部の遅延時間の例です。遅延時間はCPUを実装するプロセスによって決まりますが、この値は最近のプロセスとしてリーゾナブルなものです。やはり、メモリの読み出し時間が長いです。ALUは演算機の作り方によりますが、これに次ぐ長さになります。この数値を使うとクリティカルパスは950psecとなり、1.05GHzで動作することがわかります。

コストの計算：シングルサイクル版

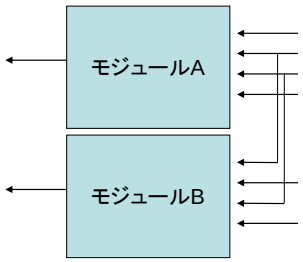
モジュール	個数
メモリ	2
レジスタファイル	1
ALU	1
加算器	1
マルチプレクサ	5
レジスタ	1
比較器	1



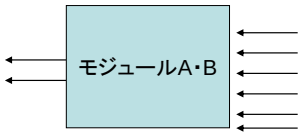
今回は半導体のコストまでは見積もれないので、その前段階となるモジュール数を見積もって見ましょう。シングルサイクル版ではj命令を実装した段階でのデータパスのリソース使用量は表のようになっています。

この授業の記述の特徴

- 出力信号依存の書き方
- 全ての出力を分離して記述している



AとBに共通性があっても
分離して書かなければならない



まとめて書ければ可読性が向上する
（かもしれない）
でもやらない

ここでの授業では、最も基本的な「入門スタイル」を使っています。この詳細はWebをご覧ください。この書き方は全ての出力を分けて書くのでやや見にくいですが、間違いが減ります。

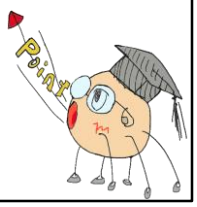
例題

- mult.asmをシミュレーションし、gtkwaveを用いて各信号の動きを確認しよう。

では、mult.asmの実行状況をシミュレーションして、gtkwaveを用いて各信号の動きを見てみましょう。

本日のまとめ

- Verilog HDLの記述を思い出そう！
- 信号線名は図と一致しているので、図を見ながら動作を追って行こう。
- gtkwaveを使って各部の信号を見て行こう。



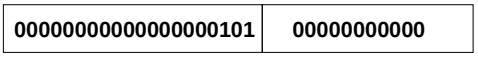
インフォ丸が教えてくれる今日のまとめです。Verilog記述を思い出してくださいませ。

演習１　lui（Load Upper Immediate)の実装

上位20bitに直値を設定する命令

lui rd, imm

- 下位は0にする
- lui x1,5



- x1を0x12345678に設定せよ
- lui x1, 0x12345
- ori x1, 0x678

RISC-Vのイミーディエイト命令は12ビットなので、この範囲を超えると値を入れにくいです。このため、レジスタの上位20ビットにデータを入れる命令が用意されています。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。

演習環境

luiはopcodeが0110111

- def.hにもOP_LUIで登録済
- アセンブラも対応している

lui.asmをアセンブル、実行して、x1が
0x12345678になればOK.
提出物はluiの付いたrv32i.v

ヒント：レジスタファイルの入力側(result)から直接入れてしまうのが楽

演習2

rs1が偶数ならば分岐するオリジナル命令
bev rs1,rs2,X 命令を付け加えよ

funct3=011を割り当てる
cp bev.dat imem.datしてシミュレーションを実行
4番地でループしたら成功
8番地以降まで進んでしまったら失敗
提出物はbevの付いたrv32i.v(luiが付いていてもOK)