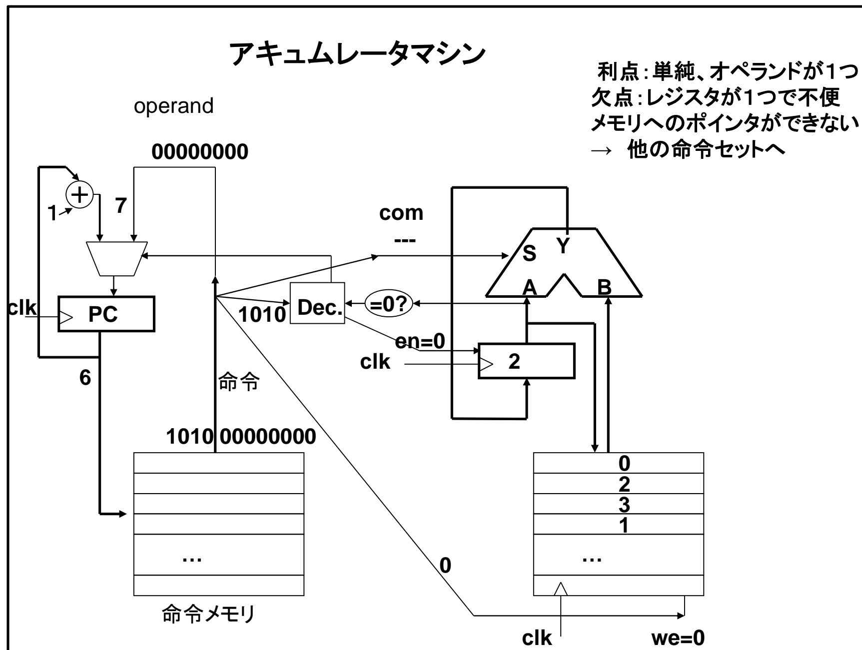
マイクロプロセッサ特論　第5回
RISCの命令セットアーキテクチャ
テキスト第4章

情報工学科

天野英晴



アキュムレータマシンは、オペランドが1つしかなく、構造も簡単ですが、欠点があります。このままではメモリに対するポインタができないので、配列もポインタも実現できません。このため、アキュムレータマシンとはいってもアキュムレータだけではなく、メモリに対するポインタ用のレジスタを持っていました。これをインデックスレジスタと呼びます。インデックスレジスタはメモリに対するポインタ、アキュムレータは計算と用途が違っています。でも使っているとインデックスレジスタでも計算をやらせたくなるので、ハードウェアに余裕が出るにつれ、同じ機能を持ったレジスタを2本持たせるようになりました。さらにレジスタの本数が増えて行き、アキュムレータマシンは複数レジスタを持つ汎用レジスタマシン（専用レジスタマシン）へと変わって行き、自然消滅しました。今回は、より広い視点でコンピュータの命令セットを紹介します。

命令セットアーキテクチャ(Instruction Set Architecture: ISA)とは？

- ソフトウェアとハードウェアのインタフェース
  - プログラムはISAを対象にすれば個々のハードウェアは気にしなくてもいい
  - ハードウェアはISAが動けば共通のプログラムが動く
  - IBM360開発時に明確になった概念
    - それまでは開発したマシン毎にソフトウェアを作っていた
    - 様々な性能、価格のモデルが同じISAを共通できた

    → IBMのメインフレームでの覇権を確立した
- IntelのIA32、ARM、SPARC、MIPSなどが長期間に渡って拡張され、利用されている

では命令セットアーキテクチャ：ISAとは何でしょう？この概念はIBM360開発時に明確になった概念です。コンピュータの草創期、開発したマシン毎に命令セットを決め、それに合わせてソフトウェアを作っていました。しかし、コンピュータの用途が広がり、色々な性能、コストの製品が幅広く要求されるようになると、ソフトウェアの共通化が必要になりました。そこで、IBMは一連の製品の命令セットを統一し、ソフトウェアとハードウェアのインタフェースとしてのISAを明確に定義しました。プログラマは、ハードウェアの詳細を気にすること無しにISAを対象にコンパイラ、OSを作り、ハードウェア設計者は、ISAの仕様を満足するように、要求性能、コストが違う様々な製品を作れば、全ての製品で同じソフトウェアが動作しました。IBM360は様々なモデルを長期間にわたって供給し、これによりメインフレームでの覇権を確立しました。以降、この考え方は全てのコンピュータに受け継がれ、Intelのx86(IA32)、ARM,SPARC,MIPS等様々なISAが長期間にわたって拡張され、利用されています。

IBM360

IBM360と、ISAの概念を固めた一人であるアムダールさんの画像はこんな感じです。

ISAの分類

- オペランド数による分類
  - 0:スタックマシン

    PUSH 0
    PUSH 1
    ADD
    POP 2

    - 演算スタックを使う方法
    - B5000、HP9000などの名機があったが80年代に絶滅→スタックを使うとパイプライン化、複数命令発行ができない
  - 1:アキュムレータマシン

    LD 0
    ADD 1
    ST 2

    - EDSAC、EDVACなど黎明期のマシン
    - 6800、6502など黎明期のマイクロプロセッサ
    - 当初からインデックスレジスタは必要としていた
    - レジスタが増えて汎用(専用)レジスタマシンに進化し、消滅
  - 2, 3:汎用(専用)レジスタマシン

    LD R0,0
    ADD R1,1
    ST R0,2

    - **現在のマシンは全てここに分類される**

ISAは、代表的な演算命令(例えば加算命令など)のオペランド数により分類されます。オペランド数が0なのはスタックマシンです。スタックマシンは、全ての命令を演算スタックで行います。スタックは後で解説するように棚であり、最初に積んだものが最後に出てきます。棚に積む作業をPUSH、棚から取り出す作業をPOPと呼びます。演算は棚の一番上のデータとその次のデータの間で行われ、結果は棚の一番上に積まれます。0番地のデータと1番地のデータを加算する場合、順番にデータをスタックにPUSHして、加算します。答えは棚の一番上に積まれるので、これをPOPして計算が終わりです。演算命令にはオペランドがないのでオペランド数は0です。スタックマシンは70年代に流行り、B5000、HP9000などの名機が生まれました。しかし、演算スタックを利用することで、高速化手法が使い難い欠点があり、性能を上げることが難しく、80年代に絶滅しました。
アキュムレータマシンは前回までに紹介した通りで、演算命令に対してメモリのアドレスをオペランドとして取ります。計算の相手は常にアキュムレータなので、一つだけ指定すれば良いのです。この方式は、EDSAC, EDVACなどの黎明期のマシン、6800、6805などの初期のマイクロプロセッサに使われましたが、半導体の集積度の向上により、レジスタが増えることで汎用レジスタマシンに進化し、発展的に消滅しました。
残ったのは、複数(4－32)のレジスタを持ち、これを指定するためにオペランドを2つまたは3つ持つ汎用レジスタマシンです。専用レジスタマシンは、汎用レジスタマシン

の特殊なもので、レジスタの用途に制限があるものです。これも半導体の集積度が向上すると、不便な制限をなくする方向に進化し、ほぼ現在は消滅しています。複数持つレジスタをここではR1,R2,…と表します。

汎用レジスタマシンの分類

- オペランド中にメモリの指定をいくつ許すか？

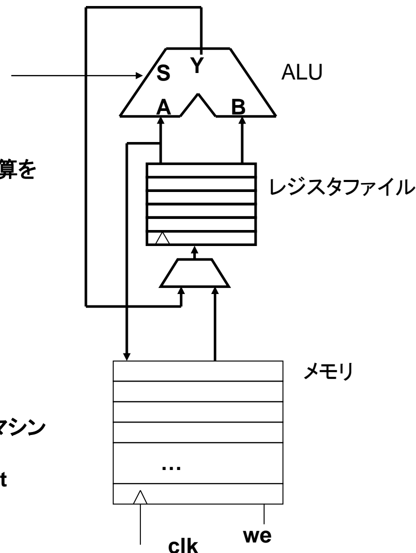
一つも許さない:register-register型 （load/storeマシン)
RISC (Reduced Instruction Set Computer)
〇命令長が固定、各命令が簡単、高速実行可能
×命令数が多くなる
ARM、MIPS、SPARCなど

一つだけ許す:register-memory型
中間的な性質： IA32（x86）など

全て許す:memory-memory型
CISC (Complex Instruction Set Computer)
〇命令数が少なくて済む
×命令長が可変、各命令が複雑になりがち
VAX-11、PDP-11など

結局のところ、汎用レジスタマシン以外残っていないとすると、汎用レジスタマシンをさらに分類するにはどのような方法があるでしょうか？複数あるオペランドのうちにメモリの指定をいくつ許すか？という視点で分類するのが普通です。汎用レジスタマシンは複数レジスタを持つので、当然複数あるオペランドにはそのレジスタ名を指定することができます。ADD R1,R2あるいはADD R1,R2,R3などのようにです。分類のポイントは、オペランドにいくつメモリを指定できるか？という点です。一つも許さない、一つだけ許す、全て許す、の3つに分けます。一つも許さない方式は、必ずレジスタ同士で演算が行われますので、register-register型と呼びます。1つだけ許す方法は、register-memory型、全てを許す方法はmemory-memory型と呼びます。これを順に解説していきます。

register-register型の
データパス

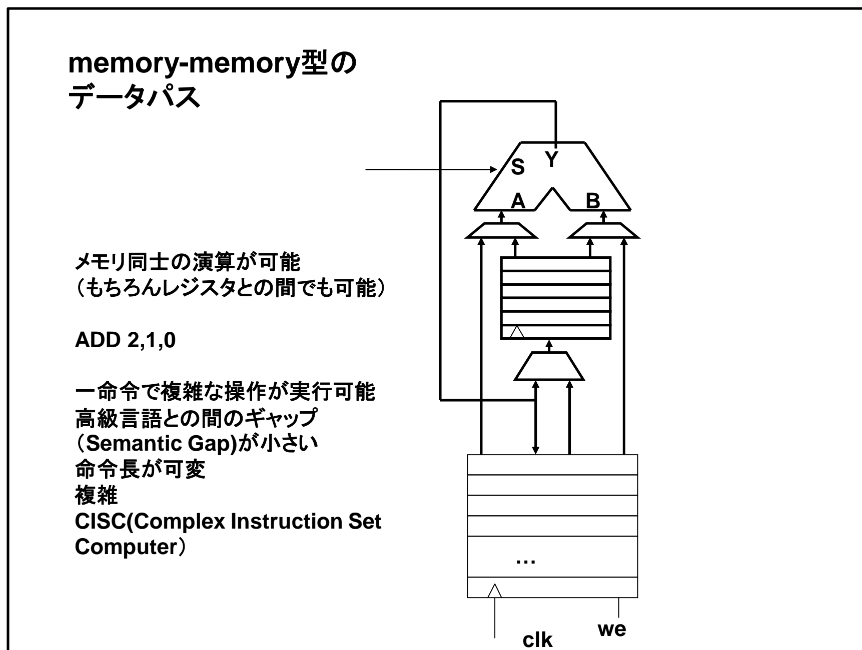


必ずレジスタに持ってきてから演算を行う

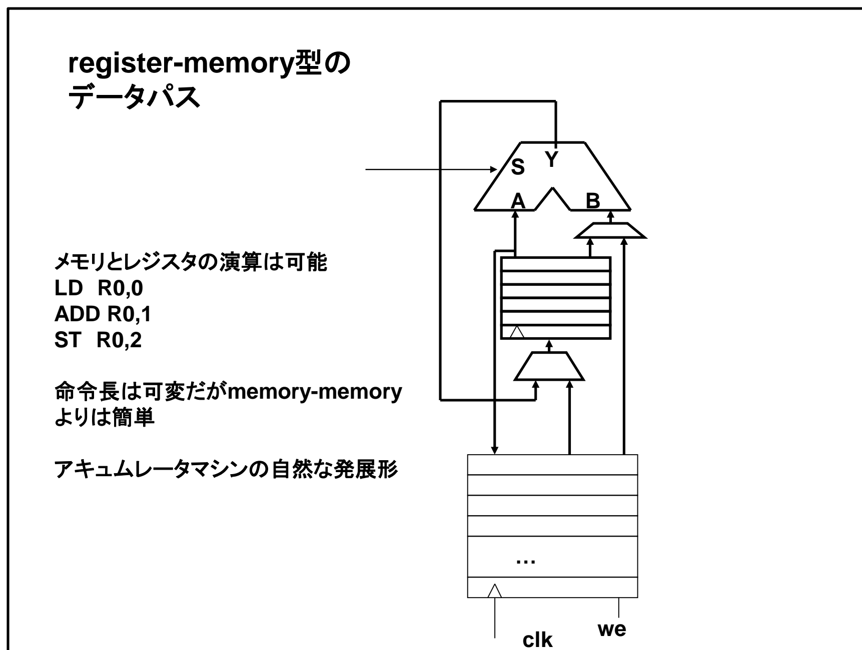
LD R0,0
LD R1,1
ADD R1,R0
ST R1,2

固定長命令が可能だが
命令数が増える
LD/STばかりやる→load/storeマシン
簡単な命令で構成
RISC(Reduced Instruction Set Computer)

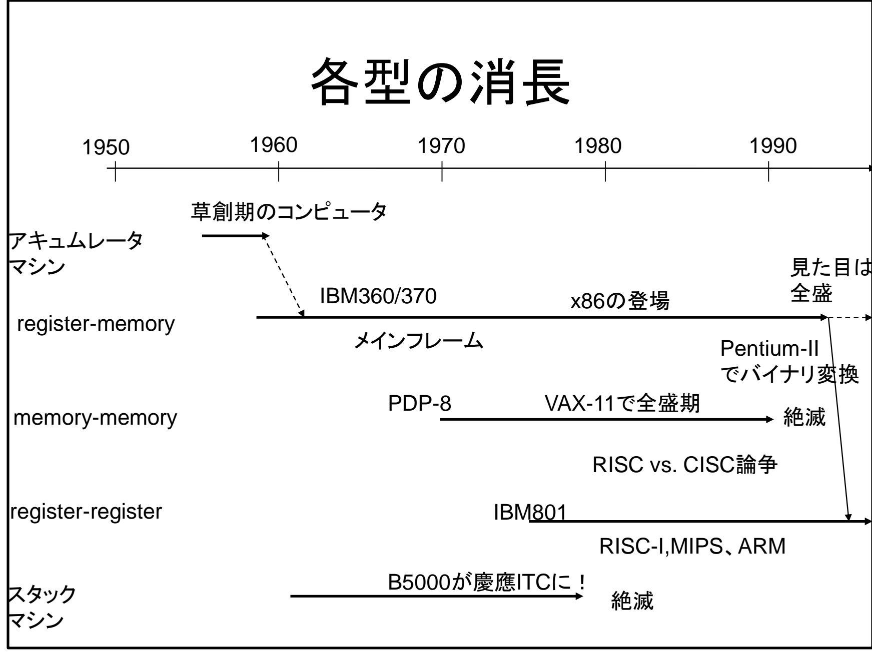
register-register型は、演算を行う場合必ずレジスタに持ってくる方法です。このデータパスの概念図を示します。汎用レジスタが持つ複数のレジスタR1,R2…は、レジスタファイルと呼ばれるレジスタの集合体の形で実装されます。これは後ほど詳しく紹介します。ここでは単純にレジスタがここに集まっていて、レジスタの値を自由に読み書きできると考えてください。register-register型はこのレジスタファイルから二つのレジスタを読み出し、これをALUの両方に入力して、答はレジスタファイルに書き込みます。メモリとのやりとりはロード、ストア命令で行います。ちなみに、台形印は以前紹介したマルチプレクサです。擬似コードを見ていただくと分かるように、ロードとストアを繰り返すことからload-storeマシン（アーキテクチャ）とも呼ばれます。演算はレジスタ同士でしか行わないことから、命令が固定長で可能です。簡単な命令で、命令数も少なくて済むことからReduced Instruction Set Computer RISCとも呼ばれます。



これと対照的なのがmemory-memory型です。この方式はメモリ同士の計算を行って、結果をメモリに戻すことができます。このため、0番地、1番地の中身を足して2番地にしまう操作がたった一つの命令で済みます。この方式は、アセンブリ表記と高級言語とのギャップ（Semantic Gap)が小さい方法であると呼ばれました。ちなみにもちろんレジスタとメモリ、レジスタ同士の演算も可能ですが、メモリ同士の演算が可能であるため、演算はメモリ同士でやり勝ちです。一つの命令の機能が高いので、CISC(Complex Instruction Set Computer)とも呼ばれます。一方で、命令長は可変でなければならず、様々なサイズの命令を扱う必要があります。DEC社のPDP-11、VAX-11がこのタイプの代表です。



register-register型とmemory-memory型の中間的な性質を持つのが、register-memory型です。これは、メモリとレジスタの演算が可能ですが、メモリ同士の演算はできません。また、答えはレジスタに書き込むのが普通です。register-register型よりは、命令数が少なくて済みますが、memory-memory型よりは多く必要です。一方、命令長は可変でなければならないですが、memory-memory型よりはバラエティが少なくて済みます。このタイプはレジスタが1個ならばアキュムレータと同じです。すなわち、アキュムレータの数が増えたものと考えられ、歴史的には最も早い時期に発達しました。IBM360,370、Intelのx86アーキテクチャはこのタイプに属します。



コンピュータの草創期は、利用可能なハードウェア量が少なかったため、アキュムレータマシンが使われました。しかし、60年代になり、トランジスタ、ICが利用可能になると、アキュムレータの数を増やしたregister-memoryマシンが登場しました。スタックマシンは早い時期に登場し一時期かなり使われましたが、80年代に入ると、コンピュータの速度向上に付いて行けずに絶滅しました。70年代に、ミニコンピュータを中心にmemory-memoryマシンが登場し、VAX-11により全盛期を迎えました。一方、この型に対するアンチテーゼとしてregister-register型が登場し、80年代に渡ってRISC vs. CISCの論争が繰り広げられました。この頃、最初の授業で紹介した一回目の革命が起こり、コンピュータの中心はメインフレームからマイクロプロセッサを使ったPCに移りました。この急激な性能向上手法はregister-register型が中心になって開発されました。この型は単純なのでパイプライン処理、命令の同時発行、コンパイラによる最適化がやり易かったのです。複雑な命令を持つため、この性能向上の技術を取り入れることができず、memory-memory型は90年代のはじめに絶滅しました。90年代のはじめ、残ったのはIntelのx86アーキテクチャを中心とした古典的なregister-memory型とregister-register型だけになりました。しかし90年代になってもマイクロプロセッサの性能向上と高速化技術の発達は留まることを知らず、Intelですらregister-memory型を守ることが困難になりました。しかし、IntelはIBM PCを始めとしてWindowsが走るPCのCPUとして高いシェアを持っており、この時点で命令セットアーキテクチャを変えることはこの市場を失う可能性がありました。そこで、Intelは、

今までと互換性のある命令を実行時にregister-register型に変換するバイナリ変換という技術を利用することで、互換性を守りつつ、register-register型向きの高速化技術を利用することを可能としました。このため、Pentium-Pro(II)以降、IntelのCPUは見た目はregister-memory型、中身はregister-register型となって今日に至っています。
register-register型の中でも様々な種類が表れて、それぞれ使われましたが、現在、最もよく用いられているのはARMという命令セットアーキテクチャです。ラップトップPCなどではIntelのx86アーキテクチャ（実は中身はregister-register型）、スマフォ、タブレット、組み込みなどではARMです。すなわち、実質的には全てはregister-register型になっています。

RISC vs. CISC

- 70年代の後半memory-memory型は全盛期を迎える
  - DEC PDP-11,VAX-11がミニコンピュータ、スーパーミニコンピュータとして広く用いられる
  - 機能の高い命令を実行することで命令数を減らす
  - 周波数が上げ難い、命令当たりの実行クロック数も大きい
- 全く反対の概念のregister-register型＝RISCが登場
  - 機能が低いが単純、固定長の命令を、高い周波数で実行する。命令当たりの実行クロック数も小さい
  - Berkeley大D.A.Patterson（RISC I・II→SPARC)、Stanford大J.L.Hennessy（MIPS)らが中心になってRISCの優位性を主張
  - memory-memory型（register-memory型も）をCISCと呼んで両者のISAの違いを明瞭にした

70年代の終わりごろ、memory-memory型は全盛期を迎えました。この方式は、DEC社のPDP-11、VAX-11で使われ、ミニコンピュータ、スーパーミニコンピュータとして数多くの大学、企業の研究所で導入されました。この時代はメインフレームが主に使われていたのですが、これは高価なため、多人数で共同利用したため、OSの研究や科学技術計算を長期間実行するには適しませんでした。そこで広まったのがミニコンピュータ、スーパーミニコンピュータで、メインフレームよりも低価格で、小型でした。現在のLinuxの前身となったUNIXなど数多くのOS、システムソフトウェアがミニコンピュータ上で開発されました。この型のコンピュータは、一つ一つの命令の機能を高めることで、機械語の機能をなるべく高級言語に近づける、という思想で作られました。このため、一つのプログラムで実行される命令数が少ないという利点がある一方、複雑な命令の実装が難しく、周波数が上げ難い欠点がありました。ここで、全く反対の概念のISAの作り方が登場しました。この方法はIBM801で試された方法ですが、本格的に考え方を広めたのは、Berkeley大のPatterson、Stanford大のHennessyらでした。彼らは、register-register型の固定長の命令を持ったISAをRISCと呼び、memory-memory型（register-memory型も）のISAをこれに対するCISCと呼んで、RISCはCISCよりも優れていると主張しました。80年代の前半に、RISC対CISCの論争が繰り広げられたのですが、80年代の後半には勝負が付いてしまいました。RISCの単純な命令の方が、実装が簡単で様々な性能向上技術を使うことができたため、性能価格比でCISCを圧倒しました。このためCISCは90年代のはじめには絶滅しました。

register-memory型で性能向上を目指していたIntelもついに内部実行形式にRISCを使うことにしたため、90年代の終わりには実質的に全てのコンピュータはRISCになってしまいました。RISCの命令セットを理解してなぜこれがコンピュータを制覇したのかを知ることがこの授業の主題の一つです。

D.A.Patterson

RISC-Iの開発者

J.L.Hennessy

MIPSの開発者

RISCを推進したPattersonとHennessyはこんな感じです。

教育用RISC　POCO

- 16bitのregister-register型
  - 命令メモリ、データメモリのアドレス、データ共に16bit(64K×16ビット）
- レジスタ8本 （r0-r7)
- 2オペランド命令

  ADD　r0,r1　r0 ← r0+r1

  左:destination operand

  右:source operand

  (IBM/Intel方式）
- 前身のPICOをさらに簡単化
  - とにかく実装が楽になるように

さて、今最も使われているregister-register型命令セットはARMです。しかしARMはやや癖があり、教育用としては適していません。そこで、ここでは非常に簡単ですが、RISCの基本は学べるPOCOという命令セットを使います。POCOは16bitのRISC（register-register型）で、命令メモリ、データメモリのアドレス、データ共に16ビットで、64K×16ビットのアドレス空間を持ちます。レジスタも16ビットを8本（r0-r7)を持っています。POCOは2オペランド命令で、ADD r0,r1のように書きます。これはr0+r1の結果をr0に入れるという意味です。二つのオペランドのうち、左はdestination operandと呼び、結果が入ります。右はsource operandといい、入力データを蓄えます。左にdestination operandを持ってくる方式はIntel/IBM方式と呼ばれ、現在の多くの命令セットで使われています。（しかし全てではないです。日立のSHシリーズなどは逆です）。POCOは教育用の命令セットアーキテクチャで、前身のPICOをさらに簡単化しており、複数の選択肢があった場合は実装が楽な方を選んでいます。しかし、安直に設計しすぎてRISCの本質的な部分は失わないように工夫しています。今日はPOCOの命令セットアーキテクチャを紹介します。

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

では、まずメモリの読み書きの方法を紹介します。POCOはregister-register型なので、メモリはロード、ストア命令以外では読み書きできません。ここで問題になるのは、実際に読み書きするアドレス（実効アドレスと呼びます。これはEffective Addressの訳で実行アドレスではないのでご注意を）をどのように決めるか？ということです。実効アドレスを決める方法をアドレッシングモードと呼びます。POCOではレジスタを指定して、その中身のアドレスが実効アドレスになるレジスタ間接指定を使います。ここではアドレスを示すレジスタを（　）を付けてしまします。（この記法は一般的に使われます）LD r0,(r1)では、r1が示すアドレスの中身が読み出されてr0に入ります。逆にST r0,(r1)は、r0の中身が、r1の示すアドレスに書き込まれます。例えばr0が100ならば100番地に書き込まれるのです。この方法はレジスタをポインタとして使うことができ、配列、リンクトリスト、スタックなど様々なデータ構造が実現できます。一方で、メモリを読み書きするために、まずアドレスをレジスタに入れておかないといけないので、この点が少々面倒です。さて、アキュムレータマシンの際使ったLD 0など、直接メモリのアドレスを書くアドレッシングモードを直接指定と呼びます。しかし、これはアドレスが長くなるのでPOCOではもっていません。

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)

基本演算命令は、レジスタ同士でしかできません。ここでは、アキュムレータマシンと同じALUを持っていると考え、全く同じ命令を想定します。一つ新しい命令はMV(move)命令で、レジスタ間でデータを移動（といっても元の値が消えるわけではないので、コピーです）します。この命令とLD、STを誤解しないでください。MVはあくまでもレジスタ間のデータの移動で、LD,STはレジスタとメモリ間のデータの移動です。

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)

アキュムレータマシンで導入したイミーディエイト命令は、POCOでは必須です。なにせレジスタに値を入れないとメモリの読み書きができないもんで。ここではLDIとADDIの2種類を定義します。LDI r1,#1はr1に1が入ります。LD r1,(r0)とくれぐれも間違えないでください。LD命令はメモリから値を持ってきますが、LDIは命令コード中の値がそのまま入ります。本来＃は不要ですが、強調のために付けることにします。ADDI r1,#5はr1に5を加える命令です。直値は8ビットの符号付数で、マイナスの値も扱えます。なので、ADDI r1,#-1でr1から1を引くことができます。このためSUBIという命令は設けていません。さて、ここで問題があります。レジスタは16ビット幅ですが、直値は8ビットです。このため16ビット幅に揃えるために符号拡張（sign extension)を行います。これは上位8ビットを符号ビットで埋める方法で、正の数の場合は0が負の数の場合は1を埋めます。このため、－1などの負の数は16ビットに拡張しても同じ値になります。これはビット幅を増やすための標準的な方法です。この方法は符号なし数を扱う場合、時に不便なので、正負を無視して上位8ビットを0で埋める符号無し命令も用意しておきます。これをゼロ拡張と呼びます。これがLDIU（LDI Unsigned）とADDIU（ADDI Unsigned)です。ADDI r1,#0xf0は、符号ビットが１なので、ADDI R1,#-16と解釈されますが、ADDIU r1,#0xf0はr1にそのまま0xf0が加算されます。

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

では、0番地の内容と1番地の内容を加算して2番地にしまうプログラムはどうなるでしょう？一例を示します。メモリの読み書きを行うのにレジスタに値を入れるのが面倒ですが、これを厭わなければ問題ないと思います。

実習用ディレクトリで シミュレーションせよ

iverilog test_poco.v poco.v rfile.v alu.v

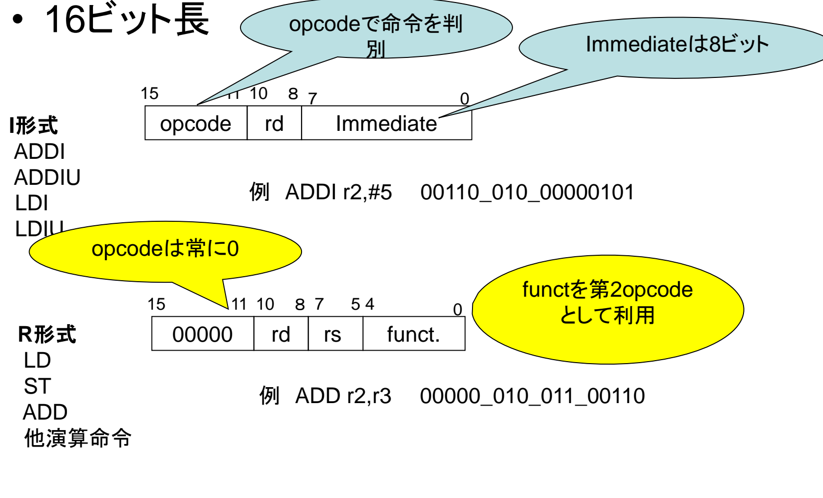
imem.datの中に先のプログラムが入っている

./a.out | more または

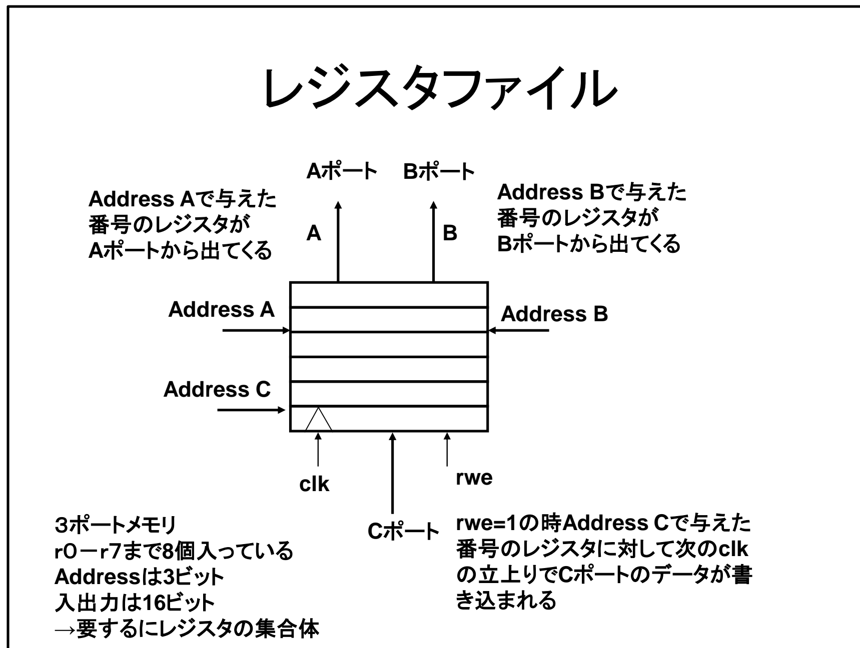
./a.out > tmp で実行結果を良く観察すること

次ページからのパワポと照らし合わせると理解しやすい

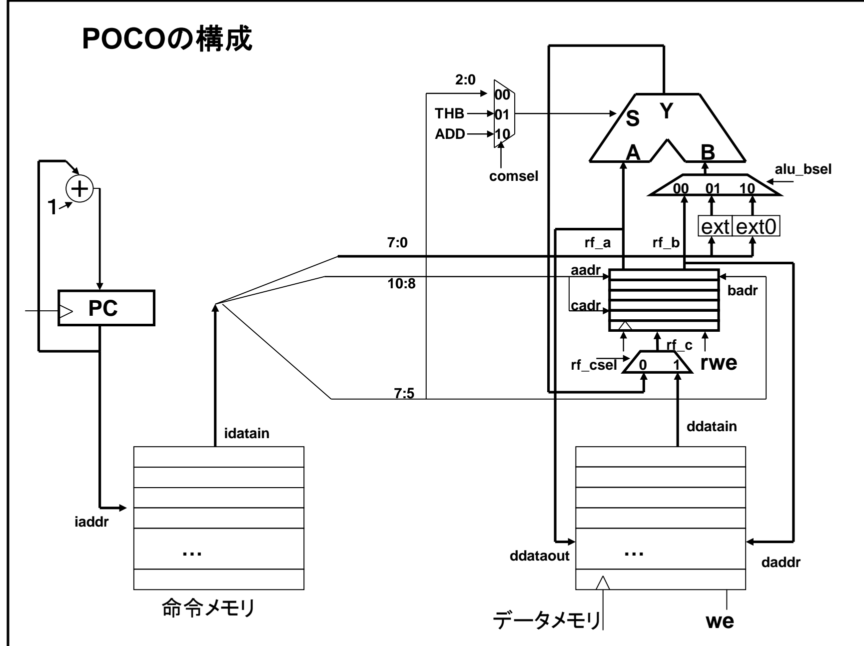
では、次は実習用のディレクトリでシミュレーションをして結果を確認しましょう。とりあえずレジスタの値が変化していく様子を見てください。



さて、ここでPOCOの命令の機械語構成を示します。POCOはregister-register型で16ビットの固定長命令を持ちます。命令の操作内容を示すopcodeは5ビットとします。これで32種類の命令が実装可能です。今まで紹介したPOCOの命令のうち、ADDI,ADDIUなどのイミーディエイト命令は8ビットのイミーディエイトとレジスタを1つ指定する必要があります。レジスタは8個なので3ビットで指定できます。このため、opcode, レジスタ番号、イミーディエイトの順に並べて、16ビットになります。これをI型と呼びます。I型の命令はopcodeで種類を識別します。ADDI r2,#5の例を示します。次にADD,SUBなどの基本演算命令とLD,ST命令はレジスタを二つ指定します。opcodeとrd、rsを並べると11ビットになり5ビット余ります。この余ったフィールドをfunctフィールドと呼び、これを補助的なopcodeとして扱います。POCOといえども32種類しか命令を指定できないのではちょっと少なすぎます。そこで、2つのレジスタを指定してイミーディエイトが不要な命令の場合、opcodeを常に0とし、このfunctフィールドで命令を識別するようにします。これをR型と呼びます。ADD r2、r3の例を示します。



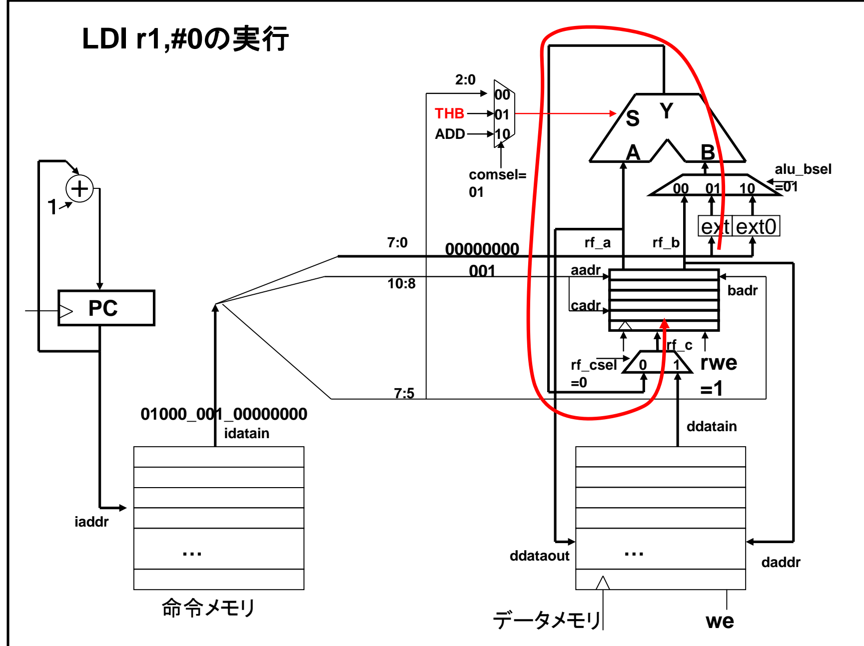
次にレジスタファイルについて具体的に紹介します。レジスタファイルはレジスタの集合体で、小規模なマルチポートメモリと考えられます。POCOで使うレジスタファイルは、A,B,Cの三つのポートを持っています。それぞれが独立のアドレスを持ち、A,Bポートは読み出しポート、Cポートは書き込みポートです。POCOではレジスタは8個なのでそれぞれのアドレスは3ビットです。アドレスAが011ならばAポートからr3が読み出され、アドレスBが010ならばBポートからr2が読み出されます。一方、書き込みは、Cポートにデータを与えて、アドレスCに書き込むレジスタの番号を与え、rweを1として、クロックが立ち上がったときに行われます。rweはregister file 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（後の授業でやります）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。



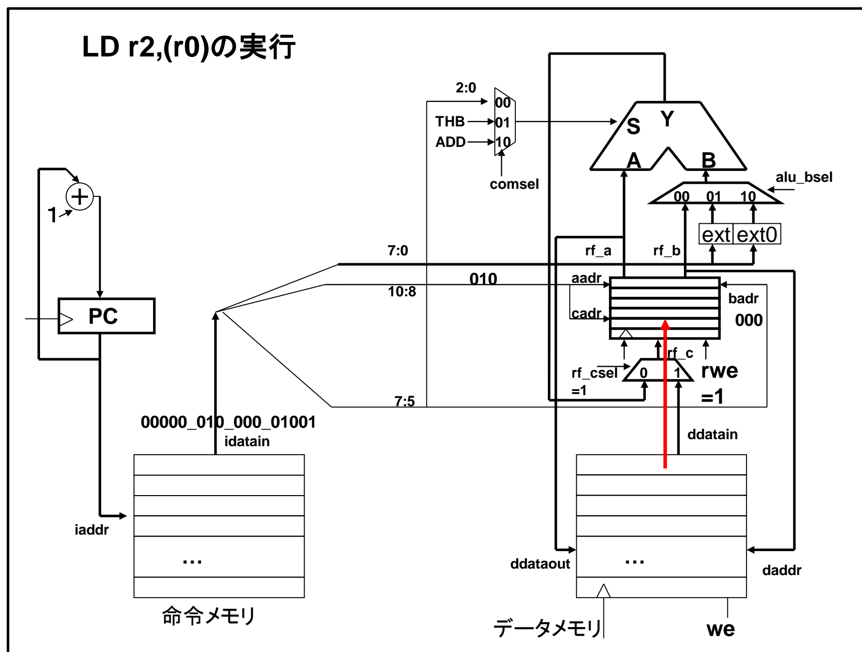
では、POCOの構成をざっと説明します。詳しいVerilog記述は来週紹介します。PC周辺はアキュムレータマシンと同じですが、アキュムレータの代わりにレジスタファイルを使います。

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。POCOは、符号付きと符号無しの命令を持っているので、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、ext0はゼロ拡張用のハードウェアですが、これはMSBを複製したり、ゼロを入れたりすれば良いので簡単です。ALUのコマンドは、アキュムレータマシン同様、functフィールドの下3ビットを入れて、R型の命令がそのまま実行できるようにします。LDI、ADDIなどを実行できるように、B入力をそのままY出力に出すTHBと、A入力、B入力の加算を行うADDを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。

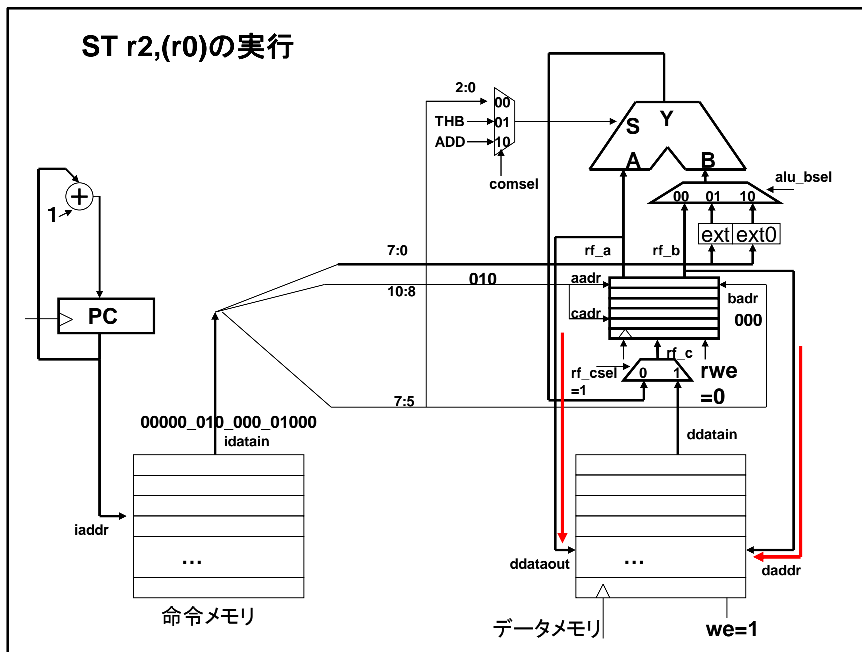
レジスタファイルのポートAのアドレスと、ポートBのアドレスには、命令コードのrd, rsに相当する部分を入れます。Cポートのアドレスには、rdに相当する部分を入れます。メモリに関してはアドレスにはレジスタファイルのBポート、データ入力にはAポートの出力を入れます。これでLD命令、ST命令を実行します。読み出したデータは、マルチプレクサを経由してレジスタファイルに入れます。



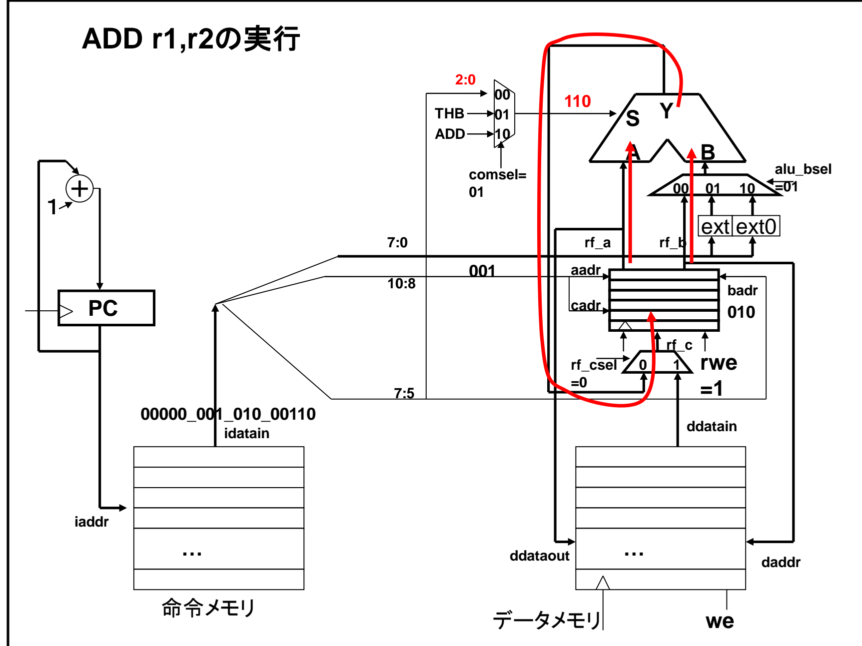
では、各種命令の実行の様子を見ましょう。LDI r1、#0のコードは01000_001_00000000です。このうち直値フィールドの00000000は符号拡張されてB入力に入ります。ALUの演算にはTHBを入れてやり、B入力がそのままYに出力されるようにします。レジスタファイルの入力マルチプレクサを切り替え（rf_csel＝0）、ALUのY出力をレジスタファイルのC入力ポートに入れます。レジスタファイルのCポートのアドレスには001が入っているので、rwe=1にしてやることで、r1に0が書き込まれます。



次はLD r2,(r0)の場合を見て行きましょう。これはR型で00000_010_000_01001になります。レジスタファイルのBポートのアドレスはrsのフィールドが入るので000になります。なのでr0がポートBから読み出されます。これは直接メモリのアドレスに入っているので、メモリのデータからr0の中身に示すアドレスからデータが読み出されます。（r0が100なら100番地からです）マルチプレクサを制御し(rf_csel=1)、読んできたデータをCポートの入力に入れてやります。ここで、Cポートのアドレスにはrdフィールドに当たる010が入っています。rwe=1にすることで読んできた値がr2に書き込まれます。



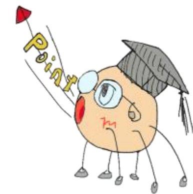
ST r2,(r0)は、LDとほとんど同じです。Aポートのアドレスが010になっていることからr2の内容がAポートから読み出されます。これがデータメモリの入力データに接続されています。実はこれはLD命令でも起きていたのですが、データは使われませんでした。しかし、ST命令では、we=1として、Bポートから読み出したアドレス（つまりはr0の中身）に、このデータが書き込まれます。



最後に通常の演算命令ADD r1,r2をチェックしましょう。この命令はR型でfunctの下位3ビットをそのままALUのコマンド入力に入れてやります。この場合はALUの加算操作を示す110が入ります。Aポートにはrdの値001が、Bポートにはrsの値010が入ります。これにより、r1とr2の値がそれぞれAポート、Bポートから読み出され、加算結果がYから出力されます。レジスタファイル入力用マルチプレクサを制御(rf_csel=0)してやり、rwe=1にして結果をレジスタファイルに書き込みます。Cポートのアドレスには Aポートと同じ値でる001が入っているので、答えはr1に書き込まれます。

本日のまとめ

- 命令セットアーキテクチャ(ISA)はソフトウェアとハードウェアのインタフェースである
  - きちんと定義すれば、ソフトウェア、ハードウェアを独立に開発できる
  - IBM360以降様々なISAが疲れている。
- ISAの分類は、オペランド数、オペランドの中でいくつメモリを指定できるかで行う
  - 汎用レジスタマシンで、オペランド中でのメモリ指定を許さないregister-register型＝RISCが現在のISAの主流
  - IntelのCPUは見かけはregister-memory型だが、register-register型に変換して実行する
- 16ビットRISC　POCO
  - メモリアクセスはレジスタ間接指定
  - イミーディエイト命令は符号拡張と、ゼロ拡張がある。



インフォ丸が教えてくれる今日のまとめです。

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

今回習ったR型命令をまとめます。新しく増えたのはMVです。LDとSTは他の命令と違っていますが、これはfunctの一番上のビットが１であるところで判断できます。

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型命令の一覧です。LDHIは来週紹介します。

演習

- 演習６－２：0番地にA、1番地にBがある場合、（A+B) OR (A-B)をプログラムして実行せよ。答えは2番地に入れろ。

  提出物　imem.dat
- imem.datの名前を変える場合、test_poco.vを書き換える必要があるので注意！
- A+B=1101　A-B=0011　1111

では演習をやってみましょう。今日は命令メモリの初期化を行うために、imem.datを書き換えればよいです。ちなみにこのファイル名はテストベンチ（test_poco.v）に記されているので、imem.datの名前を変える場合は、これに合わせてtest_poco.v中の名前を変えてやる必要があります。