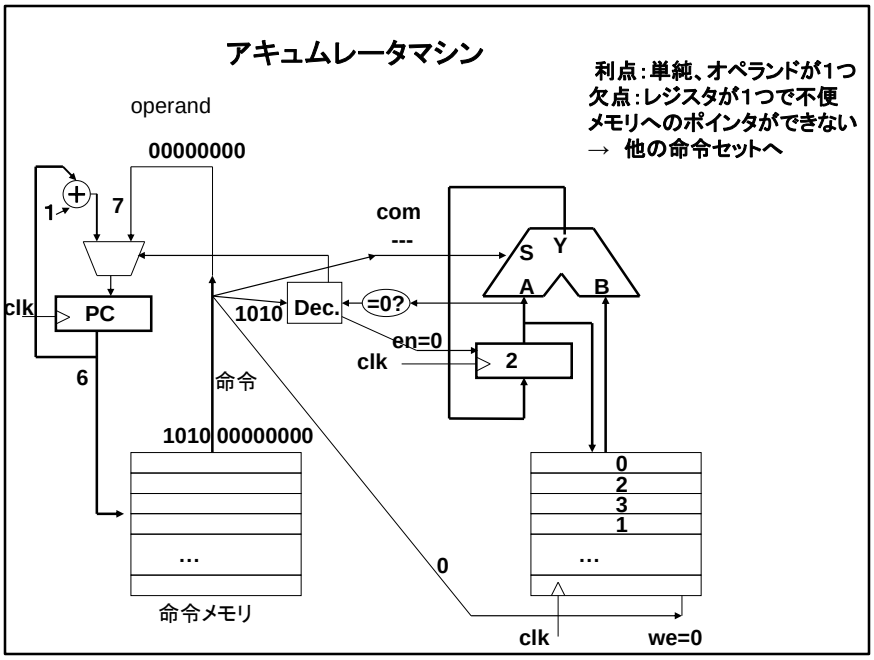
# 計算機構成　第5回
# RISCの命令セットアーキテクチャ

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
```

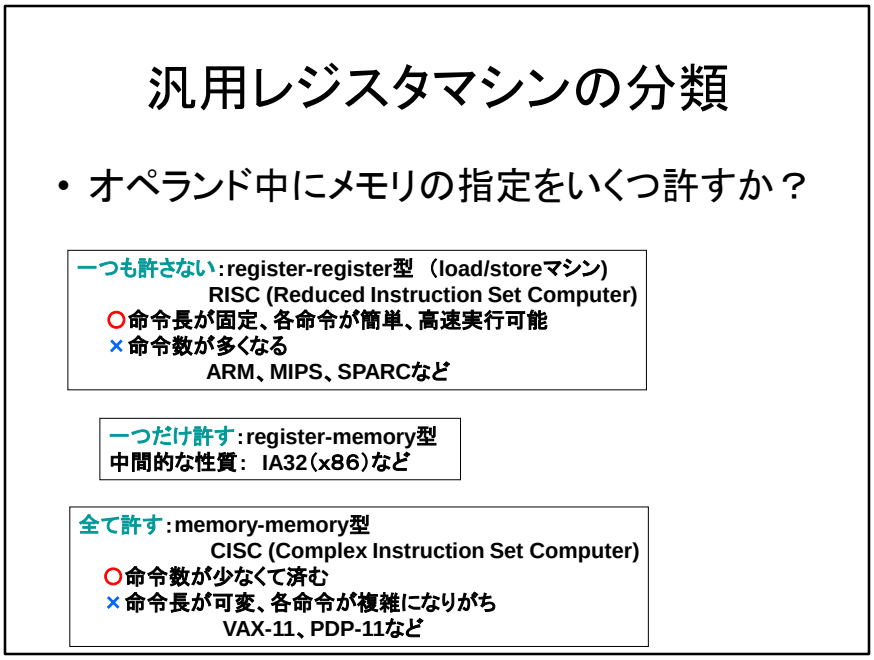
    - EDSAC、EDVACなど黎明期のマシン
    - 6800、6502など黎明期のマイクロプロセッサ
    - 当初からインデックスレジスタは必要としていた
    - レジスタが増えて汎用(専用)レジスタマシンに進化し、消滅
  - 2, 3:汎用(専用)レジスタマシン

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

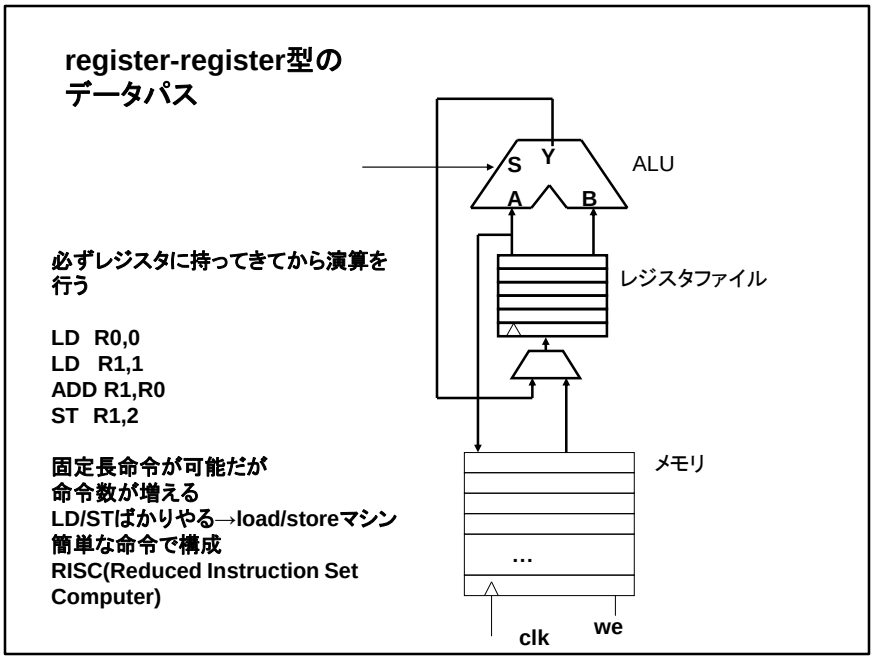
    - **現在のマシンは全てここに分類される**

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

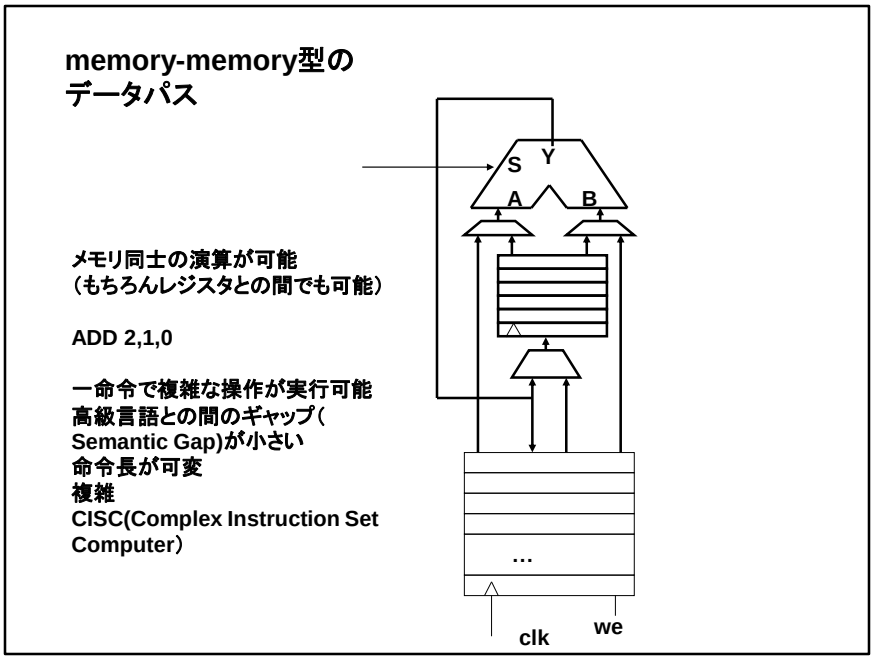
ンドを2つまたは3つ持つ汎用レジスタマシンです。専用レジスタマシンは、汎用レジスタマシンの特殊なもので、レジスタの用途に制限があるものです。これも半導体の集積度が向上すると、不便な制限をなくする方向に進化し、ほぼ現在は消滅しています。複数持つレジスタをここではR1,R2,…と表します。



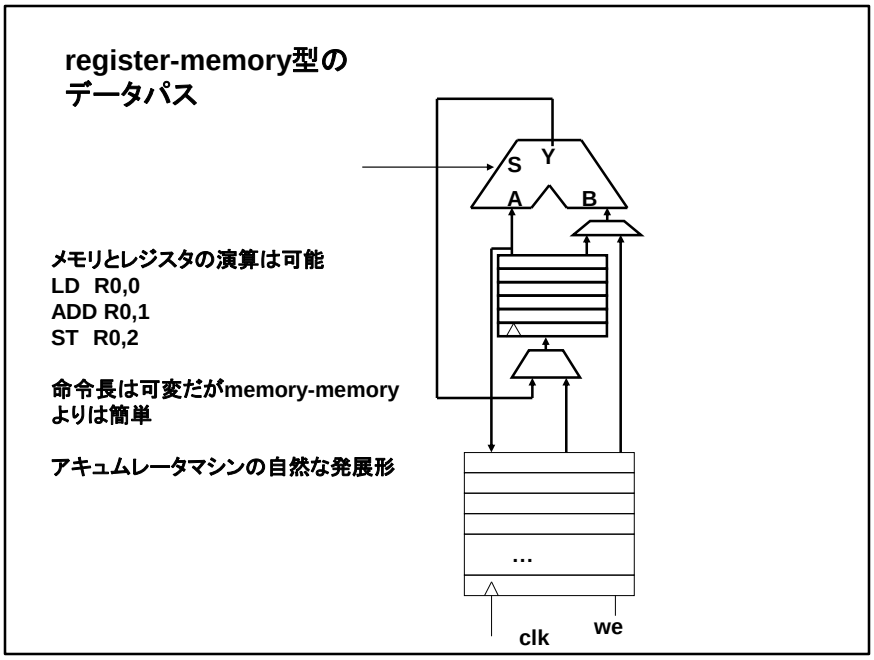

結局のところ、汎用レジスタマシン以外残っていないとすると、汎用レジスタマシンをさらに分類するにはどのような方法があるでしょうか？複数あるオペランドのうちにメモリの指定をいくつ許すか？という視点で分類するのが普通です。汎用レジスタマシンは複数レジスタを持つので、当然複数あるオペランドにはそのレジスタ名を指定することができます。ADD R1,R2あるいはADD R1,R2,R3などのようにです。分類のポイントは、オペランドにいくつメモリを指定できるか？という点です。一つも許さない、一つだけ許す、全て許す、の３つに分けます。一つも許さない方式は、必ずレジスタ同士で演算が行われますので、register-register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



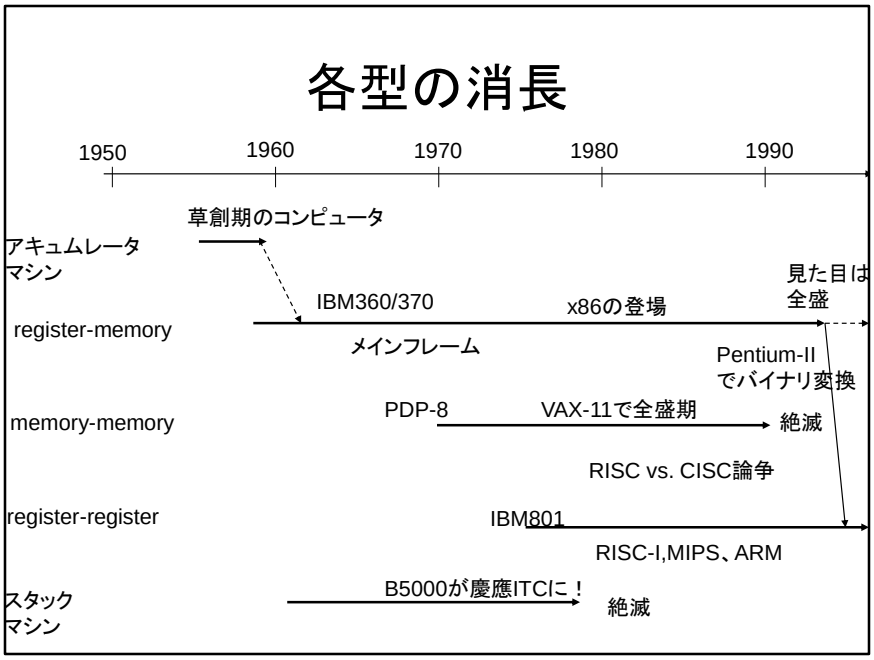

register-register型は、演算を行う場合必ずレジスタに持ってくる方法です。このデータパスの概念図を示します。汎用レジスタが持つ複数のレジスタR1,R2…は、レジスタファイルと呼ばれるレジスタの集合体の形で実装されます。これは後ほど詳しく紹介します。ここでは単純にレジスタがここに集まっていて、レジスタの値を自由に読み書きできると考えてください。register-register型はこのレジスタファイルから二つのレジスタを読み出し、これをALUの両方に入力して、答はレジスタファイルに書き込みます。メモリとのやりとりはロード、ストア命令で行います。ちなみに、台形印は以前紹介したマルチプレクサです。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。

## RISC-V　RV32I

- RISC-I,IIの流れの5番目に相当
- RISC-I,IIの問題点が取り除かれている
- ハードウェア実装を良く考えた命令構成
- RV32Iは最も基本的な32ビットISA
  - モジュール構成
    - RV32M：乗除算付き、RV64I: 64bit拡張、RV32V:ベクトル拡張、RV32C：組み込み用短命令化
- 参考文献
  - RISC V 原典　　日経BP
  - ヘネシー&パターソン　コンピュータアーキテクチャ　SiB

では、本格的なRISCアーキテクチャRISC-Vを紹介します。RISC-Vは、RISCの元祖の一つRISC-I,IIの流れを汲み、その欠点が解消されています。最も基本的な32ビットアーキテクチャであるRV32Iに、様々なモジュールを取り付けて拡張するモジュールアーキテクチャになっています。ここでは、まずこの基本構成を解説します。

# RV32Iの基本アーキテクチャ

- 32bitのregister-register型
  - メモリのアドレス、データ共に32bit(4M×8ビット）
    - 最初の実装では命令、データが分離している
  - メモリはバイトアドレッシング→ 8ビット単位でアドレス
- 32ビットの汎用レジスタを32本 （x0-x31) 持つ。ただし、x0は常に0
- 3オペランド命令

  add rd,rs1,rs2 x[rd] ← x[rs1]+x[rs2]

  左：destination operand

  右2つ：source operand

  (IBM/Intel方式）

RV32Iは32ビットのレジスターレジスタ型のアーキテクチャです。命令メモリ、データメモリ共に、アドレス、データは32ビットです。メモリの番地付は8ビット単位であり、32ビットは4つ分番地を占有します。これをバイトアドレッシングと呼び、標準的な方法です。レジスタも32ビットで、32本持っています。ここではこれをx0..x31という形で表します。ここで、x0は常に0であり、このレジスタへの書き込みは意味を持ちません。この設定はちょっと奇妙な気がしますが、大変役に立つのでほとんどすべてのRISCで使われています。RISC-Vの命令は3オペランドを持ちます。add rd,rs1,rs2と書くとx[rs1]+x[rs2]の答がx[rd]に入ります。ディスティネーションレジストとソースレジスタは完全に分離することができます。もちろんこれらに同じレジスタ番号を指定してもいいです。ディスティネーションレジスタを左に書くIBM/Intel方式を取ります。

# メモリの読み書き

- ディスプレースメント付きレジスタ間接指定
  - レジスタの中身とカッコの前に付けた数字を足した番地をアクセス！

  lw , rd,rs1,100 (lw rd,100(rs1) ) rs1の中身に100を足した番地のデータを読み出してrdに転送

  sw rs2,rs1,40 (sw rs2,40(rs1) )の中身に40を出した番地にrs2を書き込む

  ディスプレースメントは12ビットの符号付き数

  →符号拡張されて、加算される。
- 実効アドレス（実際に読み書きされるアドレス）＝レジスタ+ディスプレースメント
- lw x1,x2,0 単純なレジスタ間接指定
- lw x1,x0,100 100番地を直接指定
- lw, swはワード単位(32ビット)なので、ここでは、アドレスを4の倍数とする→後でバイトロード、ストアを導入

メモリの読み書きは、ディスプレースメント付きレジスタ間接指定を使って読み書きする番地を指定します。この方法は、lw x1,x2,100のように記述し、カッコの前に書いた数字（ディスプレースメントあるいはオフセット）にレジスタx2の中身が足された番地が実効アドレスになります。例えばx2に50が入っていたとすると、150番地が読み出されてx1に入ります。lwはload wordで32ビットのデータを読み出します。後で詳しく説明しますが、メモリの番地は8ビット単位についているので、実効アドレスは4の倍数でなければならないです。ディスプレースメントは、12ビットの符号付き数で、レジスタ中の32ビットの数に対して符号拡張されてから加算されます。したがって、マイナスの数を書くこともできます。sw(store word)はこの逆で、sw x1,x2,40 (sw x1,40(x2) )と書けば、レジスタx1の内容（32ビットのデータ）を、レジスタx2の中身に40足した番地に書き込みます。この方式は、ディスプレースメントに0を指定すれば、単純なレジスタ間接指定となり、レジスタをx0にすれば、ディスプレースメントで示した番地がそのまま指定できる直接指定方式になります。

# レジスタ間演算命令

- レジスタ同士でしか演算はできない
  - add rd,rs1,rs2　　x[rd←x[rs1]+x[rs2]
  - sub rd,rs1,rs2　　x[rd]←x[rs1]-x[rs2]
  - and rd,rs1,rs2　　x[rd]←x[rs1]&x[rs2]
  - or rd,rs1,rs2　　x[rd]←x[rs1]| x[rs2]
  - xor rd,rs1,rs2　　x[rd] ← x[rs1] ^x[rs2]
- 加減算、加減算、シフト、セット演算

RISC-Vは、レジスターレジスタアーキテクチャなので、演算にはメモリを指定することができず、一度レジスタに持ってくる必要があります。3オペランド方式なので、レジスタの書きつぶしを心配しなくて良くて便利です。

# イミーディエイト命令

- 命令コード中の数字(直値imm)がそのまま演算に使われる
- 直値は12ビットの符号付き(ディスプレースメントと同じ)

  addi rd,rs1,imm　x[rd]←x[rs1]+imm

  addi x3,x4,5　x3←x4+5

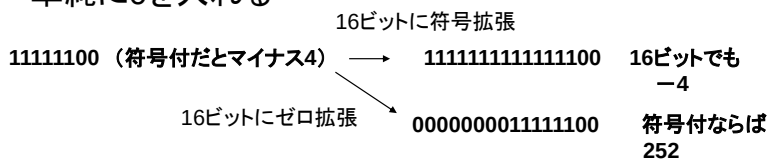
  頭に0xを付けると16進数としてアセンブラが扱ってくれる

- 12ビットを32ビットに拡張して演算する。
  - 算術演算、論理演算:いずれも符号拡張
  - subiは存在しない→マイナスの数を足せばよい

イミーディエイト命令では、命令コード中の数字（直値:imm)がそのまま演算に使われます。RISC-Vの場合、直値は12ビットで、算術演算命令、論理演算命令の両方で符号拡張されます。すなわちaddi rd,rs1,immを実行すると、immが32ビットに符号拡張され、32ビットのレジスタと加算されます。

# 符号拡張とゼロ拡張

- 短いサイズに入れたデータの桁数を増やす場合：RISC-Vでは12ビット→32ビット
  - ここでは8ビット→16ビットで解説
- 符号を考えて増やすのが符号拡張
  - 一番上の符号ビットを複製して必要な桁数に拡張する
- 考えないのがゼロ拡張
  - 単純に0を入れる




符号拡張（Sign Extension)は、符号を考えて、データの桁数を増やしてやります。RISC-Vでは12ビットの数を32ビットに拡張しますが、長すぎるので、ここでは8ビット→16ビットで説明します。8ビットの最上位の符号ビットを8ビット分上位に補い、数を16ビットに引き伸ばしてやります。これは、コンピュータのあらゆる場所で使います。符号を考えない場合、0を埋めればよく、これをゼロ拡張と呼びます。

# RV32Iの基本演算

- RV32Iでの基本演算は、整数演算、論理演算、シフト
- レジスタ間演算命令、イミーディエイト命令は統一されている（subは別：funct3も同じ）
- 乗算と除算はRV32Mで定義される
- 浮動小数点数はRV32F、RV32Dで定義される

RISC-Vでの基本的な演算は、整数演算、論理演算、シフトです。これは、レジスタ間演算命令とイミーディエイト命令で統一されています。乗算と除算はRV32Mで定義され、浮動小数点数は、RV32F、RV32Dで定義されます。

# 論理演算

- 論理積（and, andi）　&
  - 0&0=0, 1&0=0, 0&1=0, 1&1=1
  - レジスタの対応するビット間の演算となる
    - 例）1011 & 1101 = 1001
    - 1を検出するマスク操作に良く用いる
- 論理和（or, ori）　|
  - 0|0=0, 1|0=1, 0|1=1, 1|1=1
    - 例）1001 | 1101 = 1101
- 排他的論理和（xor, xori）　^
  - 0^0=0, 1^0=1, 0^1=1, 1^1=0
    - 例）1001 ^ 1101 = 0100

論理積andはここでは＆で表し、二つの入力が共に１の時だけ出力が１になります。多桁の2進数の場合、それぞれの桁の論理積をとります。この操作は、マスク操作といってある特定の桁が１かどうか判断するのに使います。これに対して論理和orはここでは｜で表し、どちらか片方の入力が１の時に出力が１になります。andと同様、多桁の場合、対応するビットの間のorになります。xorは、排他的論理和で、入力が一致しない時に１を出力します。

# シフト（論理シフト）

- 左シフト（Shift Left Logical:sll,slli命令)　<<
  - 指定ビット数分左にずらす　2倍、4倍、8倍、、、
  - ずれた分、右（LSB:Least Significant Bit)には0を詰める
    11101010<<1 = 11010100
    11101010<<5 = 01000000
- 右シフト(Shift Right Logical：srl,srli命令)　>>
  - 指定ビット数分右にずらす　½、¼、1/8、、、、
  - ずれた分、左（MSB:Most Significant Bit)には0を詰める
    11101010>>1 = 01110101
    11101010>>5 = 00000111

シフト操作はビットを左右にずらす操作です。左方向にずらす左シフト（Shift Left)は、C言語と同じく＜＜という演算子で、ずらす桁数を＜＜の後に記して表します。最も下の桁（Least Significant Bit:LSB)のずらした分には0を詰めるのが普通で、これを論理シフトと呼びます。論理左シフトは、数を2倍、4倍、8倍にすることに相当します。
一方、右方向にずらす右シフト（Shift Right)は、最も上の桁（Most Significant Bit:MSB)のずれた分には0を詰めるので、元の数を１／２、１／４、1／８…にすることに相当します。

# シフト（算術シフト）

- 右シフト(Shift Right Arithmetic:sra, srai命令）
  - 指定ビット数分右にずらす
  - ずれた分、左（MSB:Most Significant Bit)には符号ビットを詰める
  - 負の数を右シフトして(1/2、1/4、、、)も負の数の属性を保持する
  - Verilogでは>>> と表記するが、符号付数wire singedを使わなければならない

    11101010>>>1 = 11110101

    11101010>>>5 = 11111111

    01101010>>>5= 00000011

  算術左シフトは存在しない

シフト操作の中で論理右シフトは、ずらした隙間に0を詰めるため、ずらしたことにより符号ビットが0になってしまいます。負の数を右シフトさせても負の数の属性を維持するためには、ずらした隙間には符号ビットが１の時は１、0の時は0を詰める必要があります。これを行うのが算術シフトです。ちなみに算術左シフトは存在せず、論理左シフトと同じとする計算機もあります。

## 比較命令slt, slti, sltu, sltiu (set less than)

- slt rd,rs1,rs2　if(x[rs1]<x[rs2]) x[rd] ←1 else x[rd] ←0
- slti rd,rs1,imm if(x[rs1]<imm) x[rd] ←1 else x[rd] ←0
- unsignedはレジスタの値が符号無として、比較する。

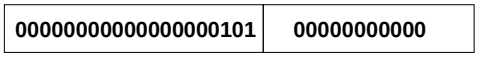
大小比較を行う命令として、RV32IではSlt (set less than)とslti(set less than immediate)と呼ぶ比較命令を使います。この命令は2つのレジスタ、あるいはレジスタとイミーディエイトを比較して、その結果をレジスタに格納します。slt rd,rs1,rs2を実行するとx[rs1]<x[rs2]の場合は、x[rd]に1を、そうでなければ0をセットします。slti rd,rs1,immはx[rs1]<immの時はx[rd]に1を、そうでなければ0をセットします。RV32Iは分岐命令に大小判定機能があるので、この命令は、主として複数のワードに跨った数を演算する時、桁上げを移動したりする場合に使われます。unsigned命令は、レジスタの中身が符号無と考えて比較します。

## lui（Load Upper Immediate)

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。

# アセンブラ

- shapaはアセンブラ
  - アセンブリ言語で書かれたプログラムを機械語に変換するソフトウェア
  - ラベルを使えるので、分岐命令などは便利
- rubyで書かれている
  - 簡単なものなので、書き直してくださいませ
- mult.asmをアセンブルするには？
  - make mult → imem.datが生成される
- シミュレーションの実行は？
  - make
  - ./test

今回の実習環境における動作の仕方を示します。Makefileはいい加減な代物なので適当に書き換えてくださいませ。

# 例題1(simple.asm)

- 0番地のメモリの内容と4番地のメモリの内容を加算して答えを8番地に格納せよ
- 演習資料prog.tarをダウンロード
  - make simple
  - make
  - ./test | more
- 結果の波形表示
  - gtkwave rv32i.vcd

では、例題をやってみましょう。prog.tarをダウンロードします。simple.asm中にアセンブラのプログラムが入っていて、make simpleにより機械語に変換が行われて、結果がimem.datに入ります。ここでmakeと打ち込むとVerilogのコンパイルが行われ、シミュレーションの実行イメージが生成されます。./test | moreでシミュレーションが実行され、結果を見ることができます。波形を見たい場合は、gtkwave rv32i.vcdで見ることができます。

## 例題1のプログラム

```
lw x3,x0,0       // x3に0番地の中身を読み出す
lw x4,x0,4     //x4に4番地の中身を読み出す
add x5,x3,x4     //加算して答をx5に
sw x5,x0,8       //2を8番地に格納
```

ポイント：x0は常に0なのでディスプレースメントが直接アドレスになる。

最後の分岐命令はプログラムを止めるため（ダイナミックストップ）

例題１では読み書きする番地が小さいので、x0が常に0であることを利用するとディスプレースメントをそのままアドレスとして使えます。演算に用いるレジスタの番号は好きなように決めることができます。

# RV32Iの条件分岐命令

PC相対指定

target ←（符号拡張）offset
beq rs1,rs2,offset: if(x[rs1]==x[rs2]) PC←PC+target
PCは命令コードの置かれたアドレス＋4
命令コード中には、offsetの0ビット目は含まれていない
bne rs1,rs2,offset if(x[rs1]≠x[rs2]) PC←PC+target

0と比較する場合はx0を使えば良い

RV32Iの条件分岐命令は、プログラムカウンタ相対指定で飛び先を指定します。飛ぶ番地の起点は、分岐命令の置かれた番地＋４で、飛び先は、飛び越す命令の数(offset)で表します。RISC-Vの場合、それぞれの命令は２バイト（RV32C）か4バイトなので、飛び先の番地は偶数しかありえないです。このためoffsetでは最下位ビットは省略して12ビット分を持たせています。飛び先を求めるには、0を補って、符号拡張して足してやります。
飛ぶかどうかの判断は、2つのレジスタを指定して、それが等しい時に分岐するbeq（branch equal)と等しくない時に分岐するbne(branch not equal)の二つが用意されています。0かどうかを判断に使いたいときは、一つのレジスタをx0にすればOKです。

# 大小比較を含んだ分岐

branch less than

blt rs1,rs2,offset：if(x[rs1]<x[rs2]) PC←PC+target

branch greater equal

bge rs1,rs2,offset　if(x[rs1]>=x[rs2]) PC←PC+target

bltu, bgeuはunsighned命令で、レジスタの内容を符号無数と考えて比較

ble, bgtなどは存在しないが、レジスタの順番を入れ替えて実現できる→疑似命令

RV32Iはレジスタの大小を比較して分岐するblt (branch less than)とbge(branch greater equal)を持っています。飛び方はbeq, bneと同じです。レジスタの中身を符号無と考えて大小比較を行うunsigned命令bltu, blgeuも用意されています。ble, bgtは存在しませんが、レジスタの順番を入れ替えて実現でき、疑似命令としては用意されています。

## 例題2(mult.asm)

- 0番地の内容と4番地の内容を掛け算した答えを8番地に格納せよ
- mult.asmを実行して結果を格納

では例題をもう一つ見てみましょう。今度はアキュムレータマシンでもやった掛け算のプログラムです。mult.asmを先ほどと同じように実行してみてください。

## 例題2のプログラム

```
        lw x3,x0,0          // 0番地の中身をx3へ
        lw x4,x0,4          // 4番地に中身をx4へ
        add x5,x0,x0        // x5を0にする
loop:   add x5,x5,x3        // x5にx3を足す
        addi x4,x4,-1       // x4から1を引く
        bne x4,x0,loop      //0でなければloopへ飛ぶ
        sw x4,x0,8          // 8番地に書き込む
end:    beq x0,x0,end
```

では例題２のプログラムを見て行きましょう。分岐命令は、対応するレジスタを、x0と比較することで0かどうかを比較します。飛び先はラベルを使って指定します。本当は、ここには、１つ先を起点として3命令分を戻すために、3命令×４＝　-12を入れなければならないです。（実際は一番下は省略する必要があります）この辺、ラベルを使えば簡単で計算する必要はなくなります。

## 疑似命令

- j offset　　jal x0,offset
- jr rs　　jalr x0,rs,0
- beqz rs,offset　beq rs,x0,offset
- bnez rs,offset　bne rs,x0,offset
- bgt rs,rt,offset　blt rt,rs,offset
- ble rs,rt,offset　bge rt,rs,offset
- li rd,imm　　addi rd,x0,imm
- mv rd,rs　　addi rd,rs,imm

RV32Iでは、直接その命令を持っておらず、他の命令により実現する場合がある。このような場合、アセンブラ上では、あたかもその命令が実際に存在するかのように扱う。この命令のことを疑似命令と呼ぶ。疑似命令はいちいち元の命令で書くよりも便利である。

# RISC-V命令フォーマット

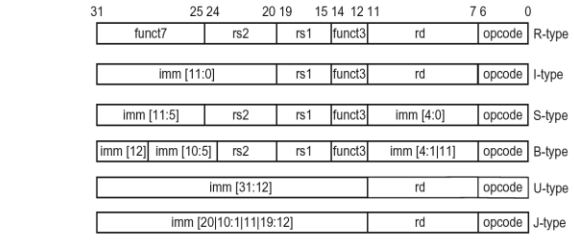
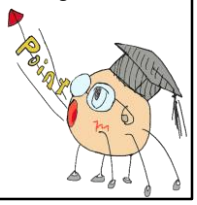



**Figure 1.7 The base RISC-V instruction set architecture formats.** All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the 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。

# 本日のまとめ

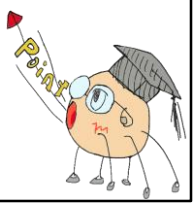
- 命令セットアーキテクチャ（ISA)はソフトウェアとハードウェアのインタフェースである
  - きちんと定義すれば、ソフトウェア、ハードウェアを独立に開発できる
  - IBM360以降様々なISAが疲れている。
- ISAの分類は、オペランド数、オペランドの中でいくつメモリを指定できるかで行う
  - 汎用レジスタマシンで、オペランド中でのメモリ指定を許さないregister-register型＝RISCが現在のISAの主流
  - IntelのCPUは見かけはregister-memory型だが、register-register型に変換して実行する




インフォ丸が教えてくれる今日のまとめです。

# 本日のまとめ

- RV32Iは32ビットのRISC、アドレス、データ共に32ビット
- アドレスはバイトアドレッシング、32ビット命令、データ
- 32ビットのレジスタを32持つ。レジスタ0は常に0
- ディスプレースメント付きレジスタ間接指定でメモリのアドレスを指定
- 3オペランド
- 条件分岐はレジスタ二つを比較、PC相対指定




インフォ丸が教えてくれる今日のまとめです。

# 覚えておくと便利

tarの解凍
tar xvf file.tar

アセンブラ xxx.asmに対して
make xxx

論理シミュレーションiverilog
make

波形ビューアgtkwave
gtkwave mipse.vcd

資料
http://www.am.ics.keio.ac.jp

レポート提出
keio.jp経由で提出のこと

# 演習

- 演習１：0番地にA、4番地にBがある場合、（A+B) OR (A-B)を計算し、答を8番地に入れるプログラムを書け
- 演習2：0番地のデータXが入っている場合、

X+(X-1)+(X-2)+….1+0　を計算し、答を4番地に入れるプログラムを書け

- 演習3：0番地から28番地までの8個のデータの総和を求めてx5に入れるプログラムを書け答えは16進数で38になる

提出物はアセンブラのファイル