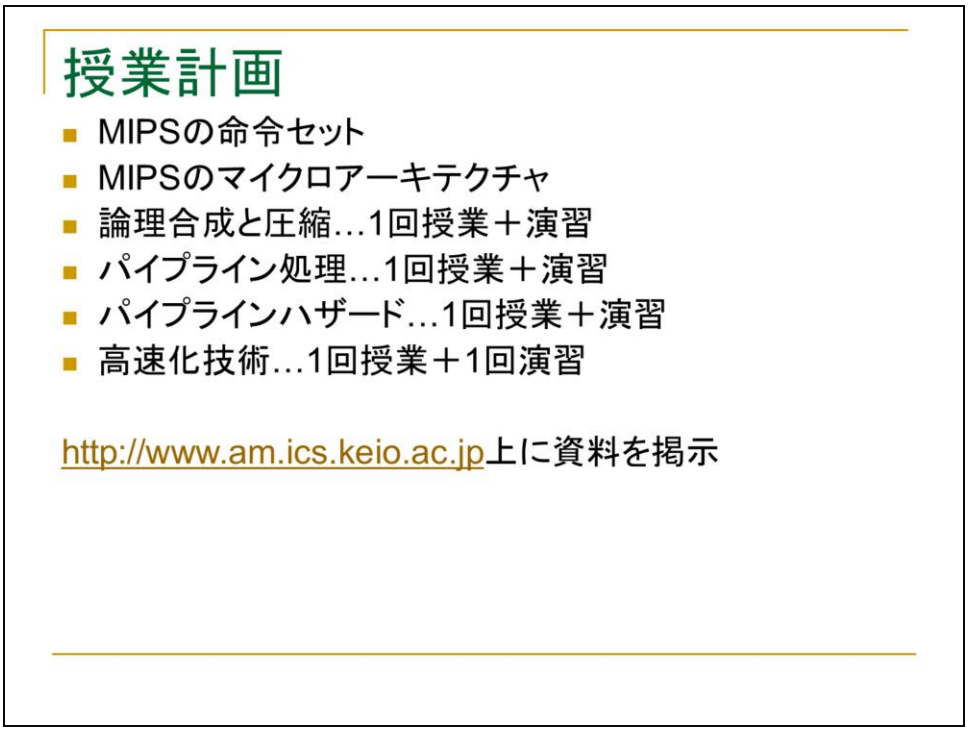
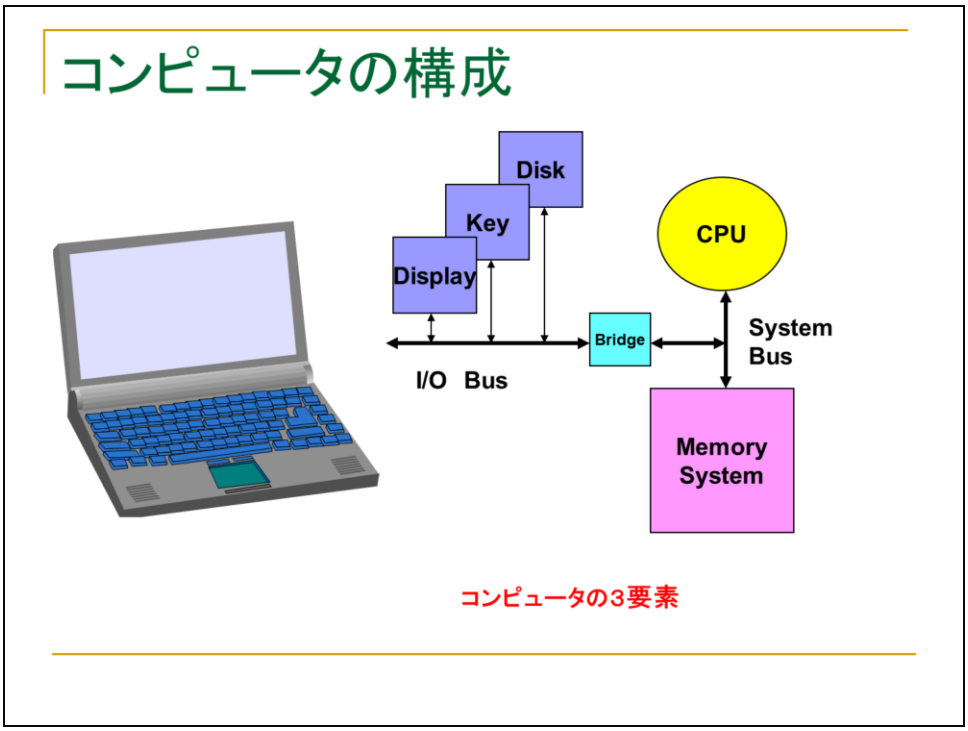
# コンピュータアーキテクチャ B
# 第1回 MIPS命令セット

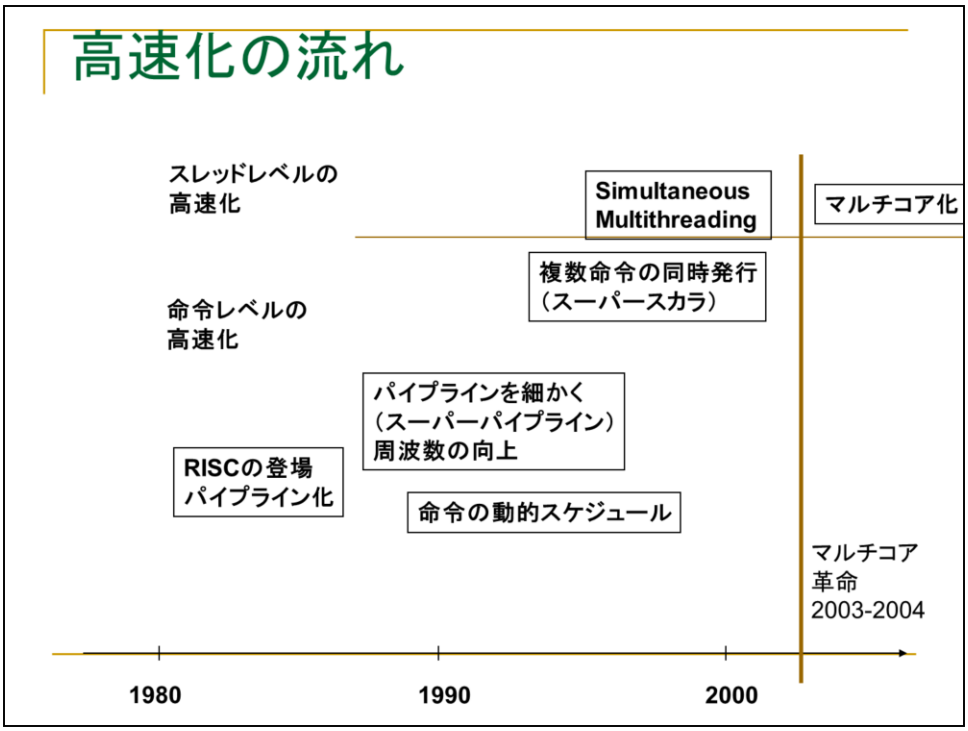
天野 hunga@am.ics.keio.ac.jp




この授業は、計算機構成同演習の後を受けてコンピュータアーキテクチャのやや高度な内容を紹介します。計算機構成同演習を受けていないと、HDLの部分が分からないと思うので、かなり補習が必要ですので、お申し出ください。




コンピュータを構成する3つの要素のうち、計算機構成ではCPUの基本的な設計、メモリシステム、I/Oを紹介しました。ここでは残った部分、すなわちCPUの高度な内容について紹介します。

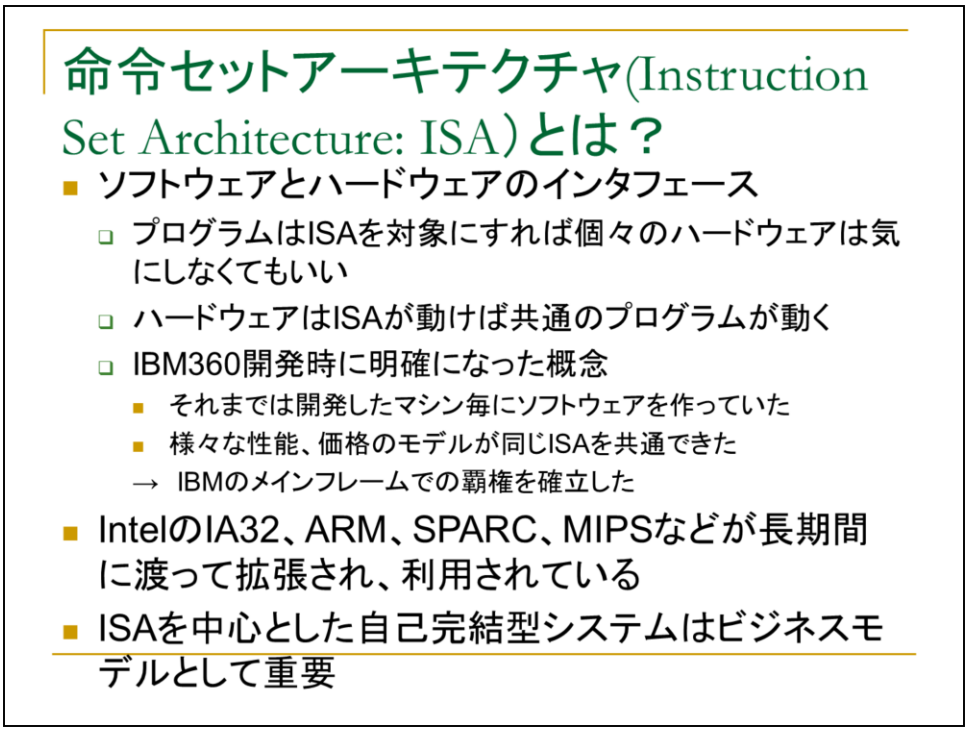



この授業ではCPUの高速化を学びます。

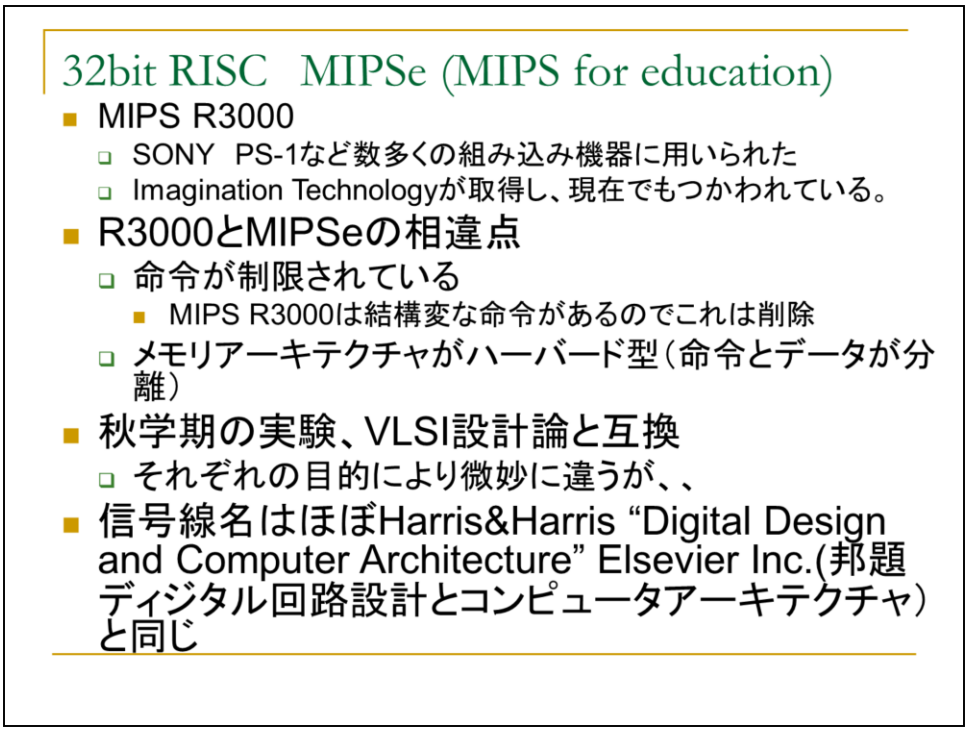
# 授業資料と評価

- 授業資料
  - http://www.am.ics.keio.ac.jp/
    - 授業・教科書→コンピュータアーキテクチャのページに掲示
    - コンピュータアーキテクチャAと共通
  - Web上の掲示に注意！
- 評価
  - 演習　50%
  - 設計コンテスト　50%
    - 未提出者は成績が付かない
    - 上位入賞者は演習に関わらずA

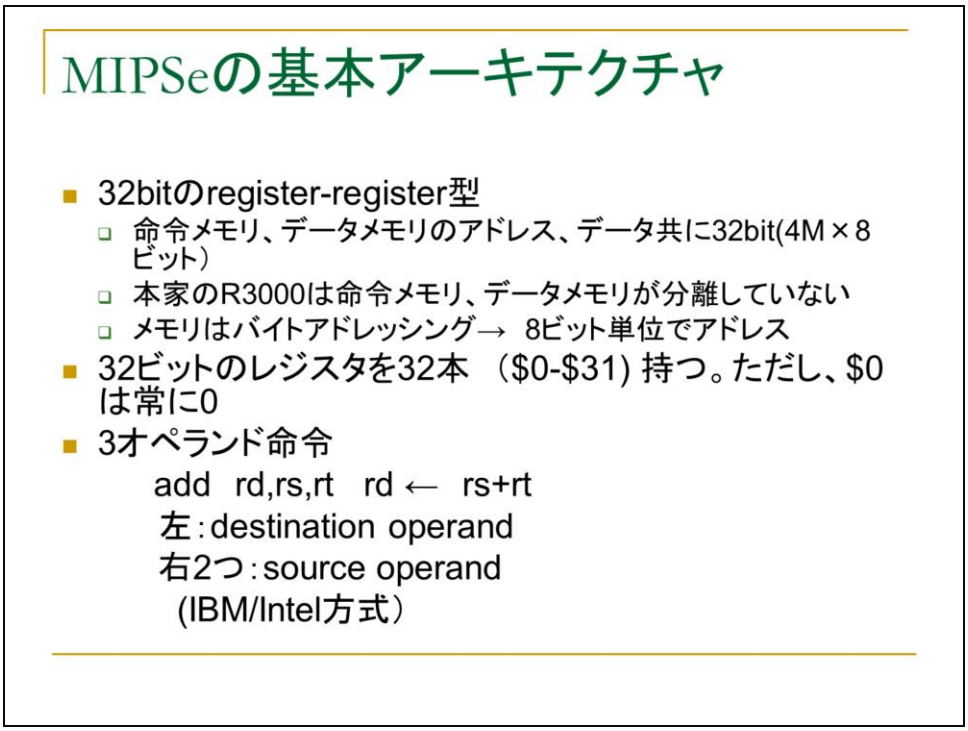
授業の方法と評価を示します。この授業は最後の設計コンテストが50%を占めます。




では命令セットアーキテクチャ:ISAを復習しましょう。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。ISAを維持することは、コンパイラ、オペレーティングシステム、デバッガ、システムプログラムなどの一式を維持することに繋がり、ビジネスとしてはこれが非常に重要です。Intelは古臭く、かつ技術的な合理性に乏しいx86 ISAを維持することにより、PCでの覇権を保ち続けました。ARMは半導体チップではなく、ソフトウェア環境を含めたIP(Intellectual Property)を提供するビジネスモデルにより、スマートフォン、タブレットからスーパコンピュータまでを制しつつあります。




では、このうちのRISCアーキテクチャMIPSの命令セットを紹介します。MIPSは1980年代にStanford大学で開発され、後に商用化されたRISCで、このうちR3000は、SONYのPlayStation-1など様々な組み込み機器に使われました。最近、MIPS社はImagination Technology社に買収された後独立し、組み込みシステムなFPGAに使われています。ここでは、MIPS R3000のサブセットであるMIPSeを使います。これはR3000のサブセットで、利用可能な命令が制限されています。R3000はそれなりに沢山命令を持っていて、中には変な命令もあって、フル実装は結構大変です。MIPSeはこのうち演習に必要で重要な命令のみを選んで実装しています。また、MIPSeは演習が楽になるように命令メモリとデータメモリが分かれたハーバード型になっています。同様なMIPS R3000は秋学期の実験やVLSI設計論でも使われます。構造はほぼ同じですが、目的に応じて微妙に構造が違っています。今回のMIPSeは信号線名がHarris&HarrisのDigital Design and Computer Architectureと同じになっています。この本はテキストとして世界中で使われており、翻訳も出てます（天野も訳者の一人です）。




MIPSe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。POCOとの違いは、メモリの番地付が8ビット単位であり、32ビットは4つ分番地を占有することです。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＄0－＄31という形で表します。ここで、＄0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。MIPSeの命令はPOCOと違って3オペランドを持ちます。add rd,rs,rtと書くとrs+rtの答が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。POCO同様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
  lw $1, 100($2)　$2の中身に100を足した番地のデータを読み出して$1に転送
  sw $1,40($2)　$2の中身に40を出した番地に$1を書き込む
  ディスプレースメントは16ビットの符号付き数
- 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 $1,0($2)　単純なレジスタ間接指定
- lw $1,100($0) 100番地を直接指定
- lw, swはワード単位(32ビット)なので、アドレスは4の倍数でなければならない→授業の後の方で議論

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 $1,100($2)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＄2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＄2に50が入っていたとすると、150番地が読み出されて＄1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、16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 $1,40($2)と書けば、レジスタ$1の内容（32ビットのデータ）を、レジスタ＄2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＄0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,rs,rt　　rd←rs+rt
  - sub rd,rs,rt　　rd←rs+rt
  - and rd,rs,rt　　rd←rs+rt
  - or rd,rs,rt　　rd←rs+rt
- 例　add $1,$2,$3　$1←$2+$3

MIPSはRISCで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。ただしPOCOと違って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16ビットの符号付き(ディスプレースメントと同じ)

addi rt,rs,imm　rt←rs+imm

addi $1,$2,5　$1←$2+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

イミーディエイト命令は、POCOと同様、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。MIPSの場合、直値は16ビットで、常に符号拡張されます。すなわちaddi rt,rs,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。演習用のアセンブラでは頭に0xを付けると16進数として扱ってくれます。

さて、POCOではゼロ拡張するイミーディエイト命令がありましたが、MIPSでは原則としてすべてのイミーディエイト命令は符号拡張をします。その代わりにメモリとのやり取りでゼロ拡張を行う命令を設けているので実際的には問題はありません。

## 例題1

- 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review.tarをダウンロード
  - ./asm.pl test.asm –o imem.dat
  - make
  - ./a.out | more
- 結果の波形表示
  - gtkwave mipse.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。test.asm中にアセンブラのプログラムが入っていて、./asm.pl test.asm –o imem.dat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。./a.out | 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 mipse.vcdで見ることができます。この辺、忘れてしまったと思うので使ってみて思い出しましょう。

## MIPSの条件分岐命令

PC相対指定

target ←（符号拡張）{offset,00}
beq rs,rt,offset: if(rs==rt) PC←PC+target
ここでPCはbeqのアドレス＋4になっているので注意
bne rs,rt,offset if(rs≠rt) PC←PC+target

0と比較する場合は$0を使えば良い

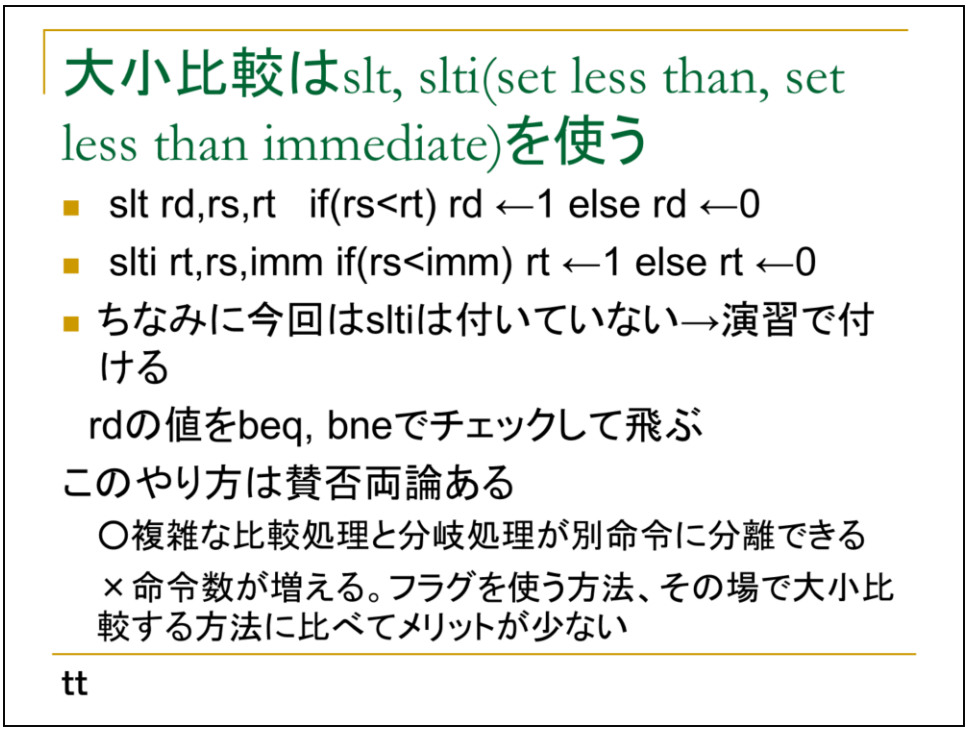
MIPSの条件分岐命令もPOCO同様で、プログラムカウンタ相対指定で飛び先を指定します。飛ぶ番地の起点は、分岐命令の置かれた番地＋4で、飛び先は、飛び越す命令の数(offset)で表します。MIPSの場合、それぞれの命令は4バイトなので、飛び先の番地に変換するには下位に00を付ける必要があります。飛び越す命令数offsetは16bitですので、18ビットで示される番地空間（±128K分）飛ぶことができます。

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$0にすればOKです。

## 例題2

- 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。




beq,bneは等しいかどうかの判断しかできません。大小比較を行う場合、POCOではbmi, bplを使って引き算をした結果の正負を判断しました。これに代わってMIPS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,rs,rtを実行するとrs<rtの場合は、rdに1を、そうでなければ0をセットします。slti rt,rs,immはrs<immの時はrtに1を、そうでなければ0をセットします。（ちなみに今日の設計にはsltiが付いておらず、演習の時につけてもらいます。）

このようにして比較結果が1か0になれば、beq, bneでこれをチェックして分岐することができます。しかし、このやり方は賛否両論があります。大小比較は複雑な処理なので、これを分岐処理と別の命令で実行することで、実装が楽になり動作速度が改善されます。一方で、フラグを使う方法や、その場で大小比較をする方法に比べてそのままではメリットが少ないです。実はこの方法はパイプライン処理のスケジュールがしやすいメリットがあります。これは後程説明します。

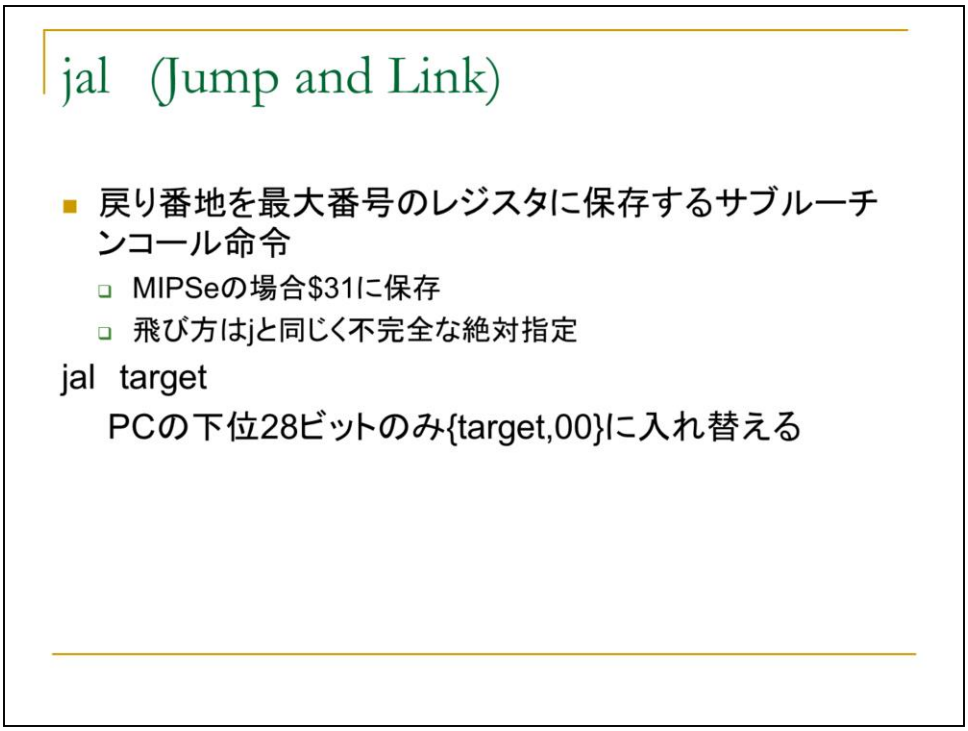
## jとjr

j target　PCの下位28ビットのみ{target,00}に入れ替える
- 絶対指定だが上位4ビットは現在のPCの値になる
- レジスタ指定がない分遠くに飛べる

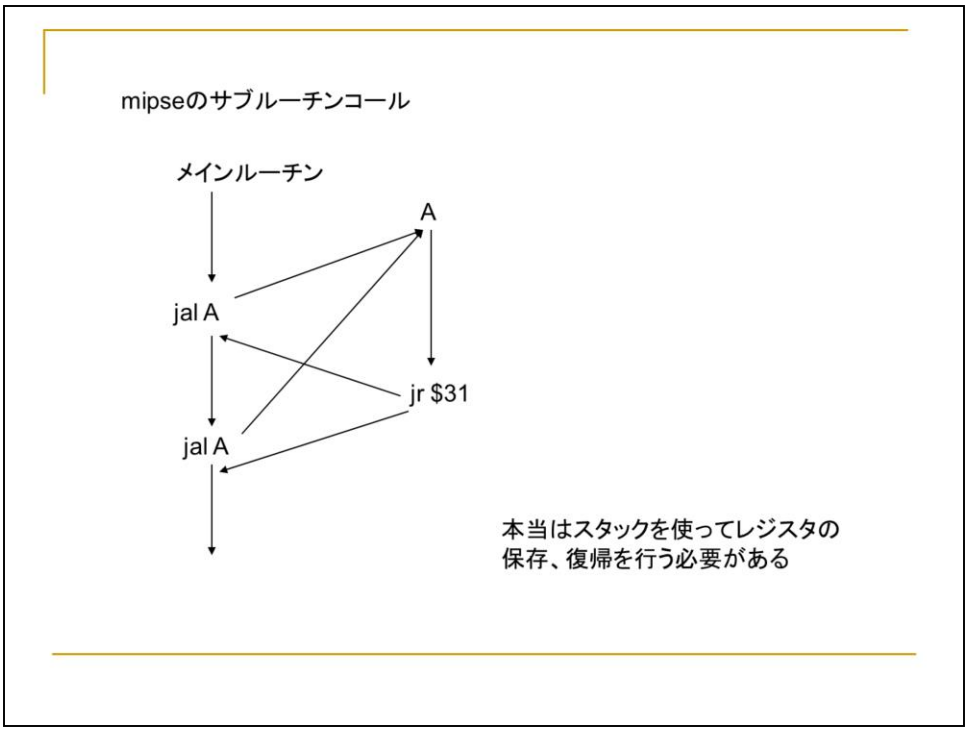
jr rs　　pc←rs　Jump Register
- レジスタ間接ジャンプ
- 絶対指定
- 32ビットのアドレス空間のどこにでも飛べる
- サブルーチンコールのリターンに使える
- テーブルジャンプなどにも使える

j(jump)は遠くに飛ぶ命令です。この命令は、普通の分岐命令が16ビットの範囲であるのに対して、26ビットの範囲で飛び越す命令数を指定できます。すなわち、メモリの番地では28ビットの範囲で飛んでいけます。全体のメモリ空間が32ビットなので、かなり遠くに飛んでいけると言えます。しかし、この命令はPC相対指定ではなく、絶対指定で、28ビットで指定できない上位4ビットは、現在のPCの値が入ります。これは、このジャンプ命令が全体のアドレス空間を16に区切ってその一つから外には出られないことを意味します。これはちょっと変なのではないかと思うのですが、事実上これで十分ということでこのような仕様になっているのだと思います。この特徴を嫌ってこの命令を使わない人もいます。16ビットの範囲で良ければ、beq $0,$0,飛び先を、常にジャンプする命令として使えばよいからです。

jr(jump register)は飛び先の番地をレジスタ間接指定するレジスタ間接ジャンプ命令で、32ビットのレジスタの中身の番地のどこにでも飛べます。ただし、下位2ビットは00で4の倍数にしておかなければならないです。




jal(jump and link)はサブルーチンコールで、考え方はPOCOと同じです。POCOではr7に戻り番地を入れましたが、MIPSの場合レジスタは32本あるので、最大の番号である＄31に入れます。飛び方はjと同じく不完全な絶対指定で、PCの下位28ビットの範囲で指定した番地が入り、上位4ビットには現在のPCの値が入ります。




POCO同様、jalを使うとサブルーチンAをプログラムの様々な場所から呼び、呼ばれたところに戻ることができます。サブルーチンコールのリターンにはjr $31が使われます。実際にサブルーチンコールを使う場合、スタックを使ってレジスタの破壊を防がなければならないです。

## 例題3

- 掛け算のサブルーチン（＄1と＄2を掛けて＄3に入れる）を用いて自乗を計算するプログラムを実行せよ
- jijo.asmを実行して結果を確認

では例題を見てみましょう。これはサブルーチンコールの例題です。

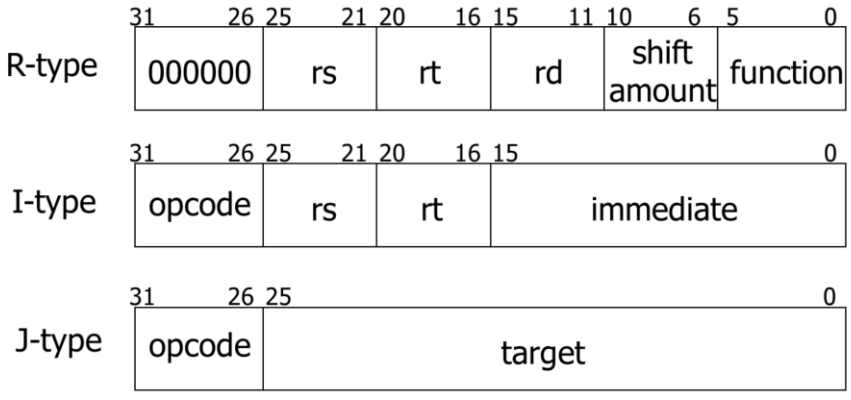
## 便利な3オペランド、$0=0

- MIPSにはレジスタ間移動命令は存在しない
  add $1,$0,$2　$1←$2
- LDI命令も存在しない
  addi $1,$0,100　$1 ← 100
- 16ビットの範囲ならば直接アドレッシングが可能
  lw $1, 0x1000($0)　0x1000番地の中身を＄1に持ってくる
- 0と比較する分岐は$0と比較すれば良い
  beq $1,$0,loop
  常に飛ぶ命令は
  beq $0,$0,target

MIPSは3オペランドが使えますし、$0が常に0です。この特徴を使うといろいろ便利です。例えばMIPSにレジスタ間移動命令は存在しません。レジスタ間の値の移動（コピー）はadd $1,$0,$2を使います。またLDI命令も存在しません。addi $1,$0,100を実行することで＄1を100にすることができます。単純な移動や値のセットに加算を使うのはもったいないのでは？と思うかもしれませんが、実はALUで加算するもの通過するのも1クロックで同じです。また、16ビットの範囲ならば直接アドレッシングも可能ですし、0と比較する分岐は＄0と比較することで実現します。常に飛ぶ命令は、beq $0,$0で実現可能です。

# 命令フォーマット

- 3種類の基本フォーマットを持つ




MIPSも3種類の命令フォーマットを持ちます。opcodeは6ビット、各レジスタは5ビットで示します。functionフィールドを使ってR型の命令を判別します。さらにフィールドが余るので、これはシフト命令のシフトするビット数を指定するフィールドとして使っています。レジスタの位置がアセンブラで書く位置と逆になっている点に注意してください。これはrs,rtをR型とI型で統一するために必要です。rtはI型ではディスティネーション、R型ではソースレジスタとして使われますので注意が必要です。

現在使えるR型命令一覧

| | | |
|---|---|---|
| add rd,rs,rt | rd ←rs+rt | 000000ssssstttttddddd100000 |
| sub rd,rs,rt | rd ←rs-rt | 000000ssssstttttddddd100010 |
| and rd,rs,rt | rd ←rs&rt | 000000ssssstttttddddd100101 |
| or rd,rs,rt | rd ←rs\|rt | 000000ssssstttttddddd101010 |
| slt rd,rs,rt | rs<rt rd←1<br>else rd←0 | 000000ssssstttttddddd101010 |
| jr rs | pc ← rs | 000000sssss0000000000001000 |

では現在使える命令の一覧を示します。

I型命令一覧

| | | |
|---|---|---|
| lw rt,offset(base) | ワードロード | 100011tttttbbbbbb offset |
| sw rt,offset(base) | ワードストア | **101011tttttbbbbbb offset** |
| addi rt,rs,imm | rt←rs+(符号拡張)imm | 001000ttttsssss imm |
| slti rt,rs,imm | rs<(符号拡張)imm rd←1else rd←0 | **001010ttttsssss imm** |
| beq rs,rt, offset | rs=rtで分岐 | **000100tttttssssss offset** |
| bne rs,rt, offset | rs≠rtで分岐 | **000101tttttssssss offset** |

lw, swはディスプレースメントを伴うため、I型になります。

J型命令一覧

| | | |
|---|---|---|
| j offset | 不完全絶対分岐 | 000010 offset |
| jal offset | 不完全絶対指定のサブルーチンコール<br>$31←pc+4 | 000011 offset |

J型はjとjalのみです。

# 演習1

- reivewを用いる
- 0番地から並んでいる8個の数字の総和を求めて0番地に書き込むプログラムsum.asmを書け
- 提出物プログラムsum.asm
- 答38（16進数）が0番地に書かれていればOK

## 演習2

- reviewを用いる
- 0番地の内容をXとしたとき、掛け算のサブルーチンを利用してXの4乗を計算せよ。
- 結果はどこかのレジスタに入れておけばよい
- 提出物プログラムyonjo.asm

5×5×5×5＝625（16進数だと271)になるはず