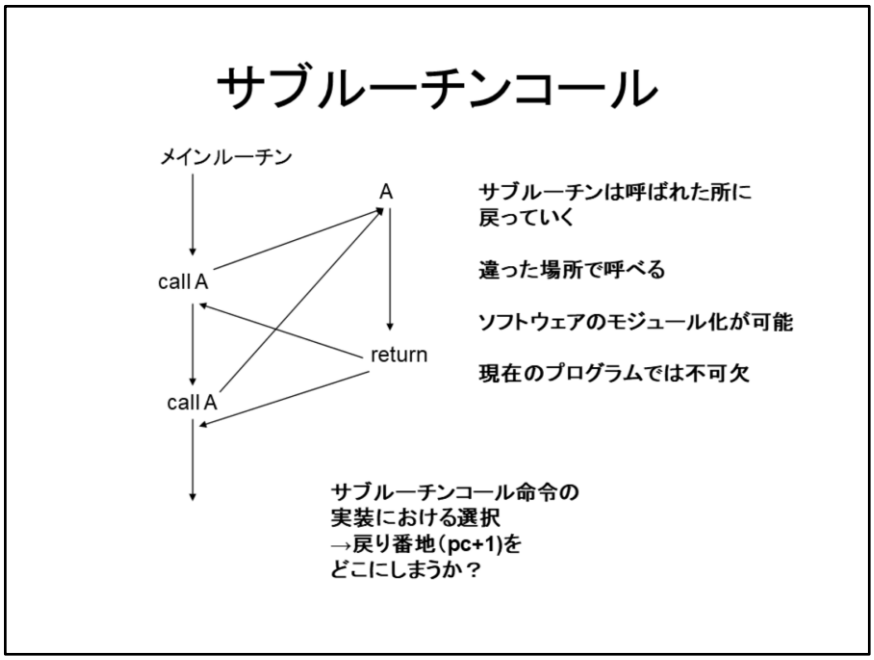
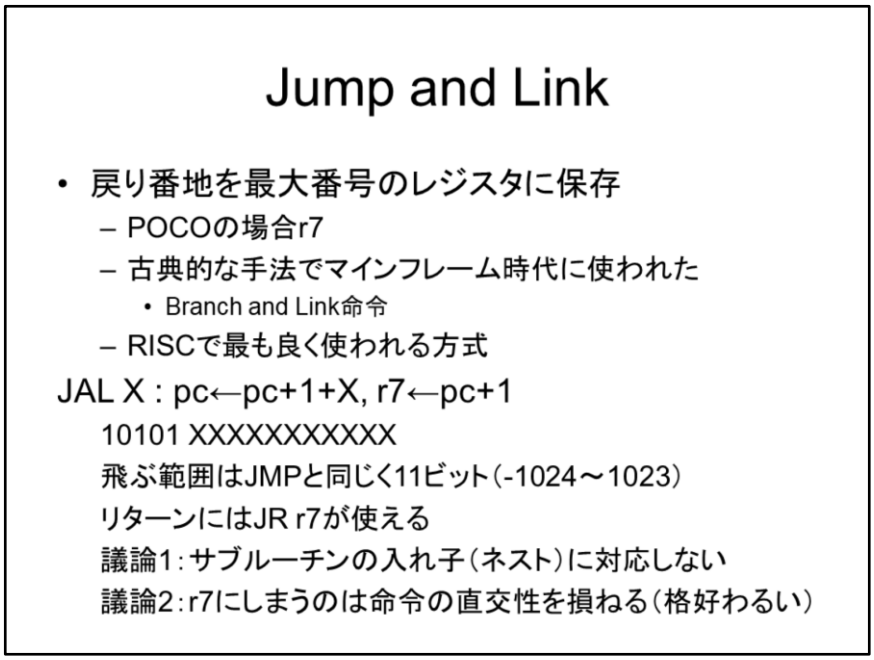
マイクロプロセッサ特論　第8回
サブルーチンコールとスタック
テキストp85-90

慶應大学情報工学科
天野英晴



分岐命令、ジャンプ命令と違ってサブルーチンコール命令は、呼ばれた所（次の命令）に戻ってくる点が特徴です。図の例ではAを呼び出して、リターン命令実行時にコール命令の次に戻ります。Aは色々なところで使えるため、ソフトウェアのモジュール化が可能です。この考え方は現在のプログラムでは不可欠です。問題は、戻り番地（すなわちコール命令実行時のPC+1）をどこにしまっておくか？という点です。



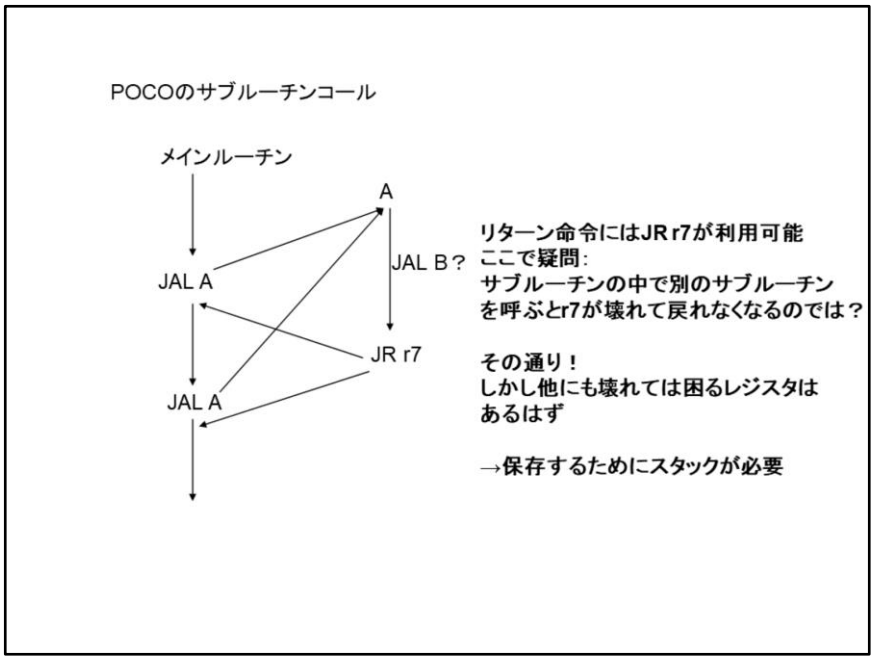
古典的な方法は、戻り番地を最大番号の汎用レジスタに保存しておく方法です。これをJump and Link（Branch and Link)と呼びます。元々メインフレーム時代に開発された由緒ただしい方法で、RISCでもよく使われます。POCOではr7に格納します。飛ぶ範囲は広い方が良いので、JMP命令と同じくJ型とし、オプコード以外は全て飛び先に使い、PC+1を起点として−1024から1023の範囲で飛んでいきます。この方法では戻り番地（PC＋１）はr7に格納されているため、リターン命令はJR　r7となります。さて、この方法には二つ議論すべき点があります。一つは、サブルーチンの中でサブルーチンを呼ぶ（サブルーチンの入れ子と呼びます）と戻り番地のr7が破壊されてしまう点です。もう一つはr7にしまうことにより、r7が他の汎用レジスタと違った役目を持つことになり、命令の直交性（Orthogonarity)を悪くしてしまう点です。この点については後で検討します。

2乗を計算する例

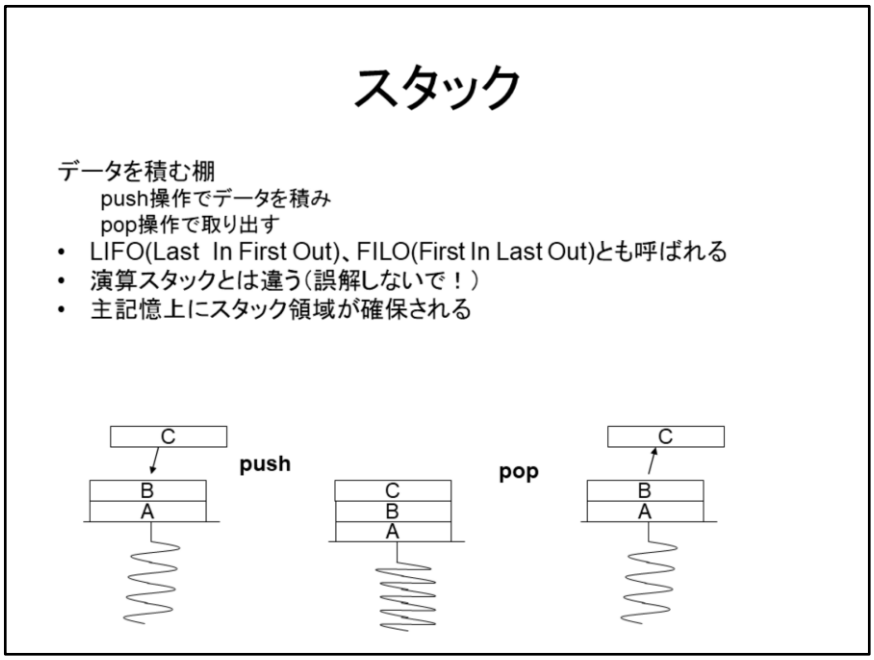
```
        LDIU r0,#0
        LD r1,(r0)
        MV r2,r1     r1とr2に同じ値をセットする
        JAL mult
end:    JMP end

// Subroutine Mult  r3 ←  r1 × r2 ここでr2は破壊される
mult:   LDIU r3,#0
loop: ADD r3,r1
        ADDI r2,#-1
        BNZ r2, loop
        JR r7            ←  r7には戻り番地が入っている
```

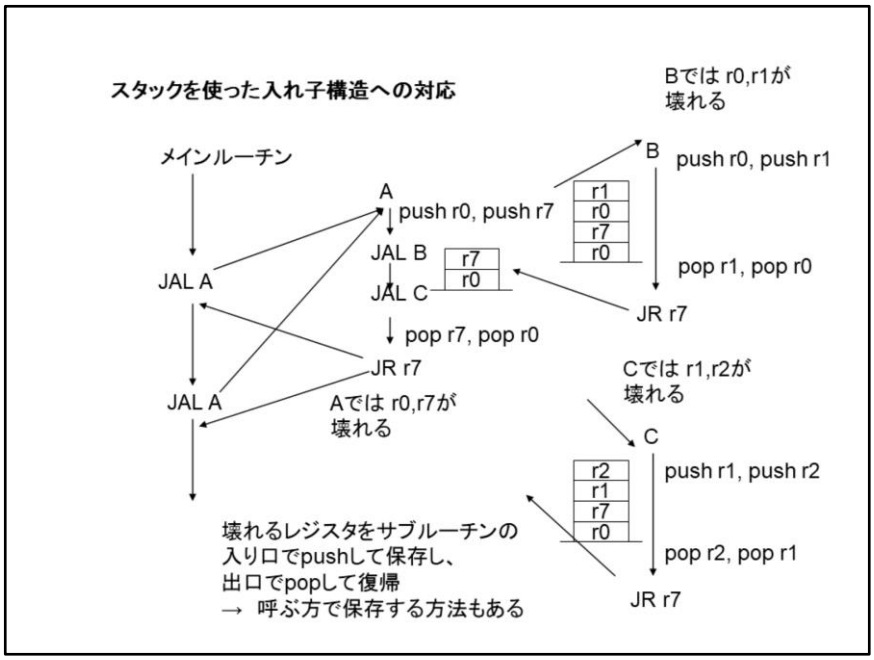
では、2乗を計算する例を紹介しましょう。この例ではサブルーチンとしてmultを定義します。このサブルーチンは、r1の値とr2の値を掛け算して、答えをr3に返します。ここではr2は破壊されてしまいます。まず、メインルーチンではr0を0にセットして、0番地の内容をr1にロードします。r1の内容をr2にコピーしてJAL multを実行するとサブルーチンが実行され、0番地の内容の自乗が計算され、答えがr3に返ります。



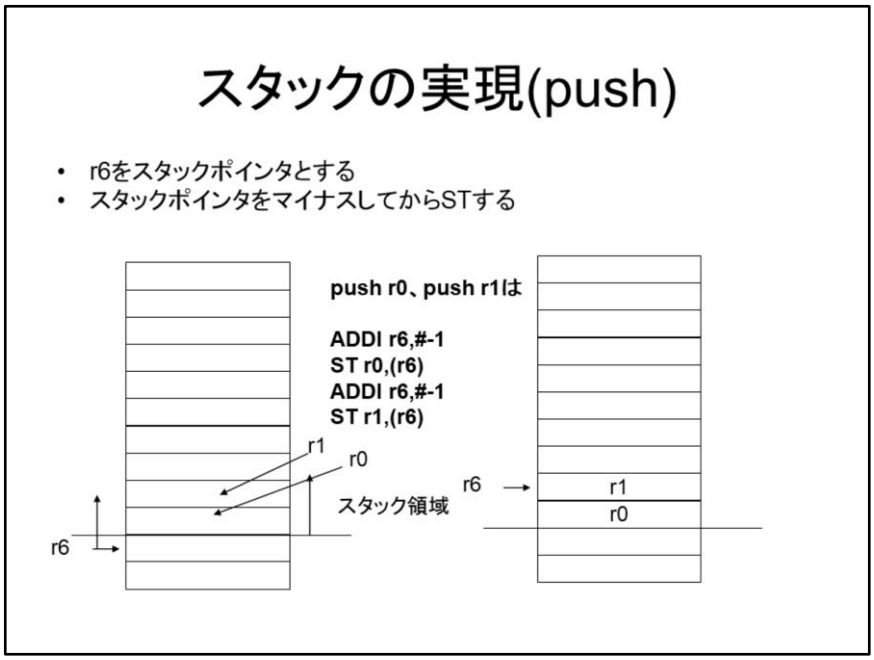
このJALというやり方は、サブルーチンの入れ子に対応しません。サブルーチンAの中で別のサブルーチンBを呼ぶと、r7の内容は破壊されてしまうため、サブルーチンAの最後にJR r7を実行しても、呼ばれた元に戻ることができません。これでは困るではないか、と思うかもしれませんが、実はサブルーチンを呼んだ時のレジスタの破壊は、r7以外にも問題になります。メインルーチンとサブルーチンA,サブルーチンAとサブルーチンBで同じレジスタを使うと、サブルーチンから戻ってきたときに中身が破壊されて、実行が継続できなくなってしまいます。すなわち、r7だけではなく、サブルーチン内でのレジスタの破壊はサブルーチンコール自体の本質的問題なのです。これを解決するにはスタックというデータ構造が必要になります。



スタックとは、データを積む棚です。この棚にデータを積む操作をpush、棚から取り出す操作をpopと呼びます。先に積んだものが後から取り出されることからLIFO (Last In First Out)と呼びます。逆に考えると、後に積んだものが先に取り出されるのでFILO（First In Last Out)と呼ぶ場合もあります。この積んだ逆順に取り出すことのできる性質からサブルーチンコール時にレジスタを退避するのに適しています。
以前紹介したスタックマシンで利用した演算スタックは、演算用の特殊なメモリですが、サブルーチンコールのレジスタの退避用のスタックは主記憶上に確保するのが普通です。スタックは棚ですが、ばねがついているイメージがあります。データを積むときは押し込むイメージからpushと呼び、取り出すときは、飛び出すイメージからpopと呼ばれます。



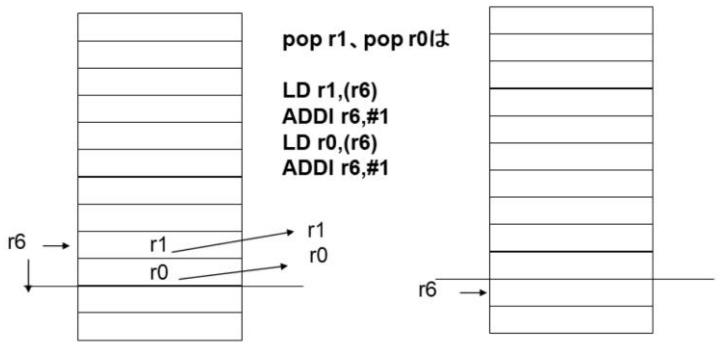
スタックを使ってレジスタを退避する様子を示します。この例では、サブルーチンの入り口で、中で使って壊れるレジスタを退避し、リターンする直前に復帰する方法を示します。これはコーリーサーブと呼びます。逆に呼ぶ側で、壊れて困るレジスタをスタックに積んでからサブルーチンを呼び出す方法（コーラーセーブ）もあります。サブルーチンAではr0を使います。中で別のサブルーチンを呼ぶのでr7も退避します。サブルーチンBを呼んだ際にr0、r1を退避します。この2つのレジスタはサブルーチンAを呼んだ際のr0、r7の上に積まれます。サブルーチンBの中でさらに別のサブルーチンを呼ぶ場合、さらにこの上に積み重なります。サブルーチンからリターンする直前に、pushしたのと逆順にpopします。そのようにすると、スタックの内容は呼ばれた時と同じになります。さらに別のサブルーチンCを呼んだ場合、サブルーチンの入れ子になった場合も同様に対処できます。再帰呼び出し（リカーシブコール）を行った場合も、スタックの容量が許す限り、スタックにレジスタを積み続けることができます。（再帰呼び出しのプログラムにバグがあるとセグメンテーションフォルトになるのは、スタックが溢れてしまうためです）



スタックをメモリ上に実現するには、スタックポインタを使います。ここではr6をスタックポインタの役割に使います。スタックポインタは、スタック領域の一番上の番地＋１の所に初期化します。push操作は、まずスタックポインタを減らし、空いた領域にレジスタを書き込みます。スタック領域はメモリ上の番地が減る方向に伸びていきます。この図はpush r0, push r1を順に実行した様子を示します。

スタックの実現(pop)

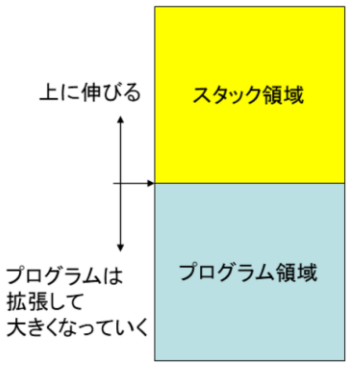
- スタック領域はメモリの番地の小さい方に伸びる→昔からの習慣
- pop操作は、LDしてからスタックポインタを＋する。



逆にpop操作はまずスタックポインタの指し示す番地から取り出して、スタックポインタを増やします。図はpop r1, pop r0を順に実行した様子を示します。スタックポインタは最初の位置に戻ります。

スタックが上に伸びる理由

- 昔からの習慣
- プログラムがアドレスの上に向かって伸びるのとぶつからないようにするため



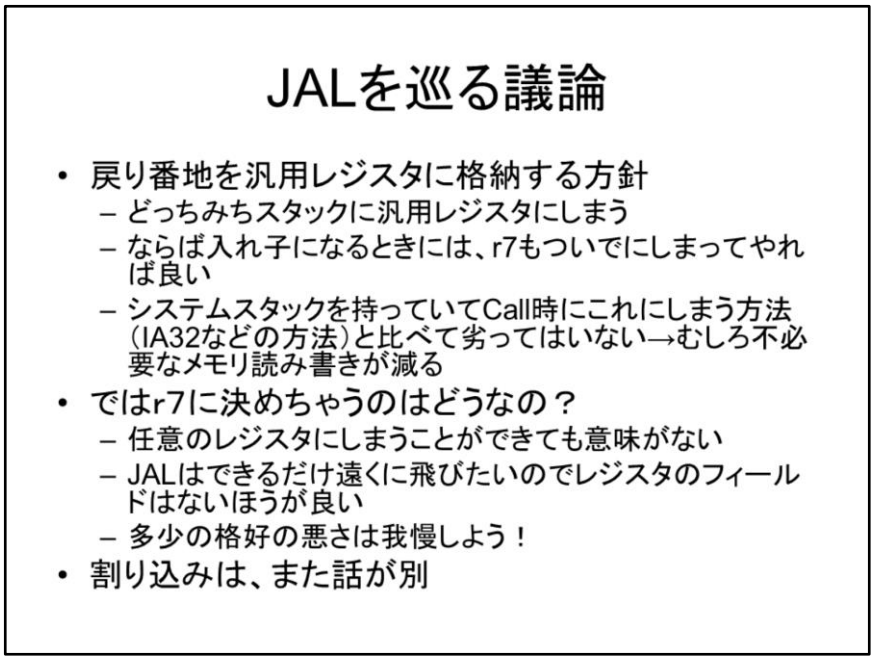
スタックは番地の小さくなる方向に（この図では上に向かって）伸びるのが普通ですが、これは昔からの習慣に基づいています。昔はある場所に境界を引いて、それより小さい番地はスタック領域とし、大きな番地はプログラム領域としました。これはプログラムというのは開発を進めると大きくなるので、番地が増える方向に伸びます。スタックを番地の増える方向に成長させると方向が同じになり、成長したスタックがプログラム領域を食い荒らす可能性があります。そこで、スタックは番地が小さい方向に伸ばす習慣ができました。もちろん領域を食いつぶしてしまえば、どちらの方向でもトラブルになります。

掛け算用サブルーチン

r2を破壊しないサブルーチンコール
r6はメインルーチンで初期化する必要がある

```
// Subroutine Mult  r3 ← r1 × r2
mult:   ADDI r6,#-1  この2行でr2をスタックにpush
        ST r2,(r6)     r2の値がスタックに退避される
        LDIU r3,#0
loop: ADD r3,r1
        ADDI r2,#-1
        BNZ r2, loop
        LD r2,(r6)    この2行でr2をスタックからpop
        ADDI r6,#1  r2の値がスタックから復帰する
        JR r7          それからリターン
```

ではどのレジスタを保存すればよいでしょうか？もちろん答えを返すレジスタであるr3は保存しません。r2は入力の値を渡すレジスタですが、これをスタックに退避することで、r1同様、メインルーチンで値を使うことができます。



JALは戻り番地を汎用レジスタr7にしまうため、サブルーチンが入れ子になると壊れてしまいます。しかい、どっちみち他の汎用レジスタだってメインルーチンとサブルーチンで両方使う場合は、壊れるのでスタックにしまう必要があります。サブルーチンの入れ子になる場合はこれと一緒にr7もしまっていまえばいい、という考え方です。これはリーゾナブルだと思います。Intelのx86（IA32)などでは、システムスタックというシステムで管理するスタックを持っていてサブルーチンコール時に戻り番地をこれに自動的にpushする方法を取ります。これはサブルーチンの入れ子に対応可能ですが、入れ子でない場合も強制的にスタックに積んでしまうので、無駄なメモリアクセスが増えると言えます。
次にr7に決めてしまっていますが、これはどうでしょう？JALは出来る限り遠くに飛びたいです。これは、サブルーチンは、ライブラリとして、ユーザープログラムとは別の番地に置かれる場合が多いためです。戻り番地をしまうレジスタはどっちみちどこかに決めてしまいます。ならば、これをr7に決めてしまい、残りの全てのビットを飛び先を決めるアドレスに充てた方が良いです。このことにより命令の直交性が低下します。直交性とは、ある操作をレジスタ番号や命令の種類に関係なしに施すことができるかどうかを示す性質です。直交性の高い命令は美しい命令になります。r7のみ戻り番地をしまえることにより直交性は低下しますが、この場合実害はないので、多くのRISCはこの方法を使っています。

JAL命令のVerilog記述

- まず例によってJAL命令をデコードする

```
assign jal_op = (opcode == `JAL_OP);
```

- pcをr7に保存する部分

```
assign rf_c = ld_op ? ddatin: jal_op ? pc+1: alu_y;   //
        JAL命令ならばPC+1をレジスタファイルの入力に与える
assign rwe = ld_op | alu_op…| jal_op;        //
        レジスタファイルに書き込むためにrweを1にする
```

では、JAL命令を付け加えてみましょう。JAL命令は他の命令とやることがかなり違うので、改造箇所が多いです。まず他の命令と同じく命令デコード信号を作ります。これがjal_opです。いうまでもなく、def.hにJAL_OPを定義（10101)しておく必要があります。次にpcをr7に保存しなければいけないので、レジスタファイルの入力rf_cを変更する必要があります。jal_opの時はpc+1を入れるようにし、rweも1になるようにします。

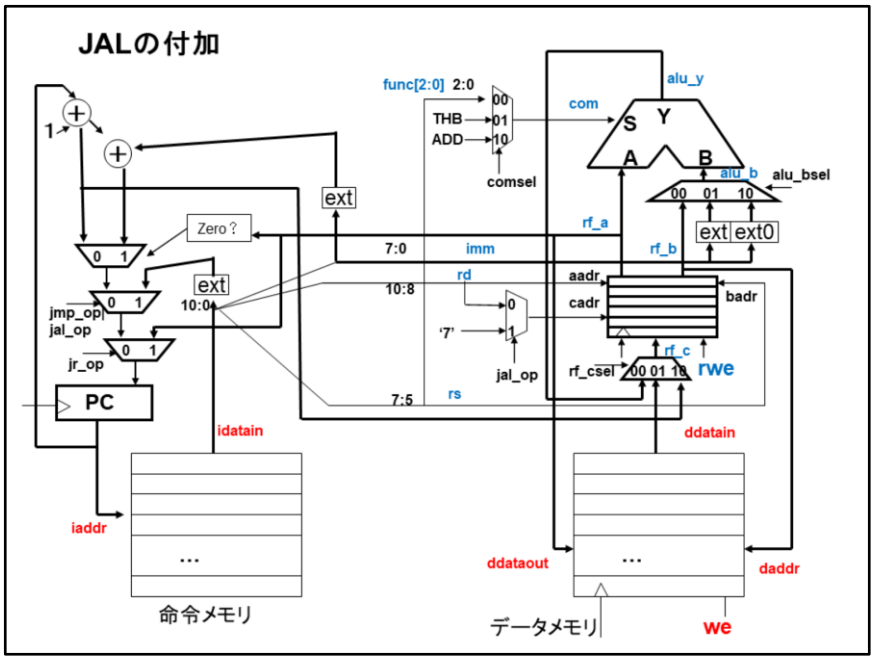
r7へのpc+1の書きこみ

今まで、レジスタファイルのAポートとCポートは同じくディスティネーションレジスタの番号(rd)を入れていた。しかしJAL命令では番号7を強制的に入れる必要がある。そこで、新しくcadrという信号名を設ける。

```
wire [`REG_W-1:0] cadr;
assign cadr = jal_op? 3'b111 : rd; //
        JALのときだけ7それ以外は今まで通りrd

rfile  rfile_1(.clk(clk), .a(rf_a), .aadr(rd),
               .b(rf_b), .badr(rs), .cadr(cadr), .we(rwe);
```

次にレジスタファイルのCポートのアドレスについて変更が必要です。これまではレジスタファイルのAポートとCポートは同じディスティネーションレジスタ番号rdを入れていました。これはPOCOの演算のrdがソースとディスティネーションを兼ねるためです。ところがJAL命令に関してはr7にpc+1を書き込むという特殊な操作が必要になります。そこで、Cポートのアドレスcadrという信号名を定義し、jal_opのときのみr7を示すために‘7’を入れてやります。それ以外の命令では今まで通りrdを入れます。このcadrをレジスタファイルのCポートアドレスに接続します。



この変更によって、POCOの構造は図のように変わります。Cアドレスのところに新しいマルチプレクサが必要です。レジスタファイルのCポート入力のマルチプレクサも拡張しPC+1を引っ張ってきて入れてやります。

PC周辺

飛び方はJMPと同じ

```
always @(posedge clk or negedge rst_n)
begin
    if(!rst_n) pc <= 0;
    else if ((bez_op & rf_a == 16'b0) | (bnz_op & rf_a != 16'b0) |
          (bpl_op & ~rf_a[15]) | (bmi_op & rf_a[15]))
           pc<=pc+{{8imm[7]}},imm}+1;
    else if (jmp_op| jal_op)
           pc <= pc +{{5{idatain[10]}},idatain[10:0]}}+1;
    else if(jr_op)
           pc <= rf_a;
    else
           pc<=pc+1;
end
```

PC周辺も若干変更します。JALはJMPと同じくJ型の命令なので、オプコード以外の11ビットを全て飛び先指定に使います。したがってJMP命令と同じ符号拡張の記述を使います。

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
JR rd	pc ← rd	00000ddd---01010

今までに出てきたR型の命令です。前回と同じです。

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
BEZ rd,X	if(rd=0) pc←pc+X+1	10000dddXXXXXXXX
BNZ rd,X	if(rd≠0) pc←pc+X+1	10001dddXXXXXXXX
BPL rd,X	if(rd>=0) pc←pc+X+1	10010dddXXXXXXXX
BMI rd,X	if(rd<0) pc←pc+X+1	10011dddXXXXXXXX

今まで出てきたI型の命令です。これも前回と同じです。

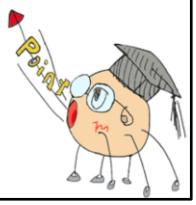
J型命令一覧

JMP #X	pc←pc+X+1	10100XXXXXXXXXX
JAL #X	pc←pc+X+1, r7←pc+1	10101XXXXXXXXXX

今回新しくJAL命令が加わっています。

ここまでのまとめ

- サブルーチンコール命令JAL 飛び先
  - 飛び方はJMPと同じで11ビットの範囲で飛べる
  - 戻り番地はr7に格納される
- リターン命令はJR r7
- 壊していけないレジスタはスタックに退避
  - r6スタックポインタとする
  - pushはADDI r6,#-1 , ST rx,(r6)
  - popはLD rx,(r6), ADDI r6,#1
  - サブルーチンの入れ子になる場合はr7を退避



インフォ丸が教えてくれる今日のまとめです。

POCOの評価

- CPUの評価尺度
  - 性能
    - 実行命令数、CPI（Clock cycles Per Instruction)、動作周波数で決まる
  - コスト
    - チップ上の面積、利用ゲート、メモリ数で決まる
  - 電力
    - ダイナミック電力とスタティック電力で決まる
- 性能対コスト、性能対電力にはトレードオフ（片方を立てると片方が立たない関係がある。

ここまでで、POCOは一応完成です。ではこれをどのように評価するのでしょう。CPUの評価尺度には性能、コスト、電力があり、性能と他の2つにはトレードオフ（片方を立てればもう片方が立たない）関係があります。必要な状況に応じてこれをいかにバランスするかが設計上重要です。

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
→ POCOでは1だが通常のCPUでは命令毎に異なる

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

まず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、POCOでは全部1ですが、普通のCPU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。現在、POCOは全ての命令を1クロックで実行するため、この問題については実感がわかないと思うので、後ほどマルチサイクルをやってから検討しましょう。

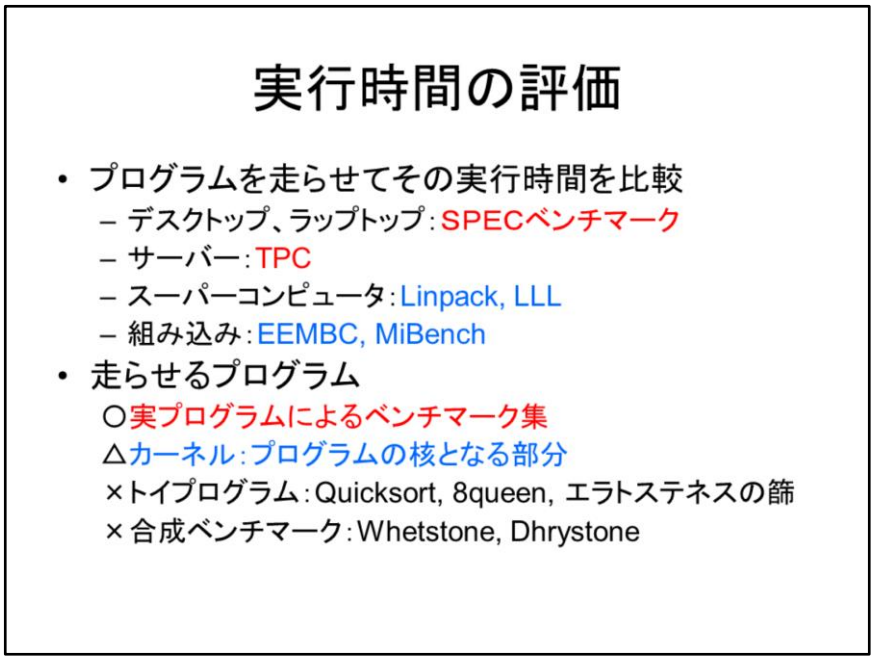
性能の比較

- CPU A 10秒で実行
- CPU B 12秒で実行
- Aの性能はBの性能の1.2倍
  遅い方の性能（速い方の実行時間）を基準にする

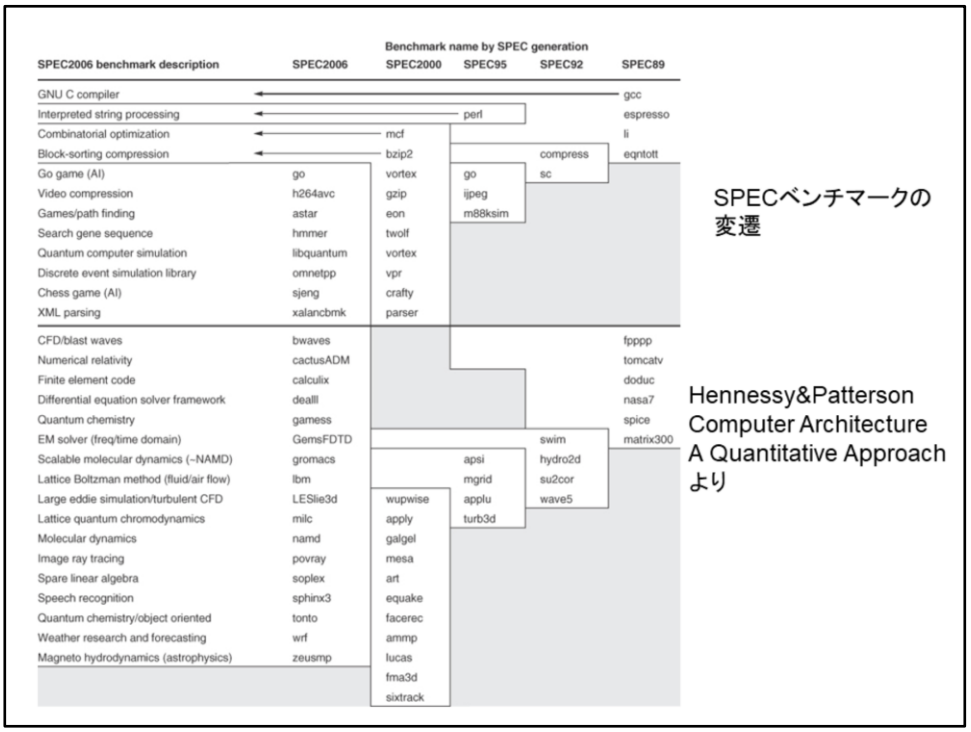
$$\frac{\text{CPU Aの性能}}{\text{CPU Bの性能}} = \frac{\text{CPU Bの実行時間}}{\text{CPU Aの実行時間}}$$

×BはAの1.2倍遅い この言い方は避ける

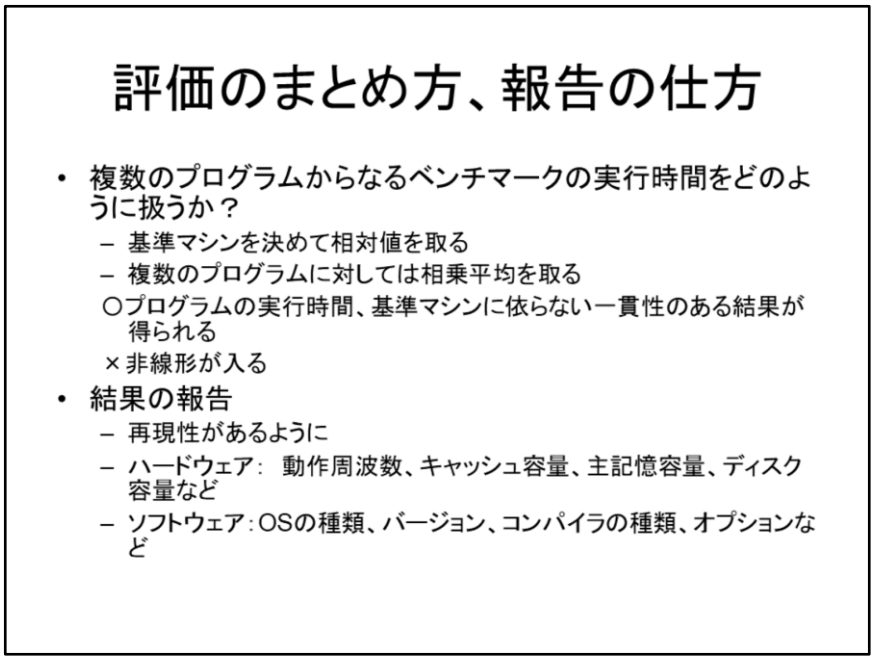
では次に性能の比較方法について検討します。CPU Aはあるプログラムを10秒で実行し、Bは同じプログラムを12秒で実行します。AはBの何倍速いでしょう？この場合、Bの性能を基準とします。Bの性能はBの実行時間の逆数、Aの性能はAの実行時間の逆数なんで分子と分母が入れ替わり、Bの実行時間をAの実行時間で割った値となります。これは12／10で1.2倍になります。ではBはAの何倍遅いのでしょうか？この考え方は基準が入れ替わってしまうため混乱を招きます。このため、コンピュータの性能比較では常に遅い方の性能（つまり速い方の実行時間）を基準に取ってで、（速い方）は（遅い方）のX倍という言い方をします。



では、実行時間はどのように評価すればよいでしょうか？もちろんプログラムを走らせればいいのですが、ではどのようなプログラムを走らせればいいのでしょう？最も良く使われているのは、現実のアプリケーションプログラムを一定の入力データのセットと組み合わせたベンチマーク集です。ベンチマーク集（ベンチマークスーツ）は複数のプログラムからできていて、良く使われるのがデスクトップやラップトップの業界で使われるSPECベンチマークです。SPECベンチマークはCコンパイラ、CAD、人工知能のプログラムから成る非数値系のSPECintと、流体力学、構造解析、量子力学などの数値計算のプログラムから成るSPECfpがあります。実プログラムから出来ているため、リアルな評価ができる利点がありますが、評価するのに手間が掛かり、プログラムが複雑なので具体的な性能の分析がやり難いです。このため、スーパーコンピュータや組み込みシステムでは、実際のプログラムの中で良く使われる部分を抜き出したカーネル（プログラム核）を用いる場合があります。スーパーコンピュータのランキングに使うLinpackなどのがこの例です。Quicksort、８－Queenなどの簡単なプログラム（トイプログラム）は、コンパイラがまだ出来ていない計算機の評価に使われる場合もありますが、一般的なプログラムとは違った特殊な動きをするため、あまり良い方法ではないです。また、様々なプログラムの挙動を一つに詰め込んだ合成ベンチマーク、Whetstone,Dhrystoneは、今でも組み込み分野では使われることがありますが、やはりプログラムの挙動が一般的ではない点、ベンチマークのみに通じる最適化を施しやすい点などから、やはり良い方法ではないといわれています。



このスライドはSPECベンチマークの変遷を示しています。真ん中の線より上がSPECint、下がSPECfpです。ベンチマークは、対象とするマシンの性能向上やメモリの増大に対応して、一定の期間毎に入れ替えが行われています。



ではベンチマーク集を使って評価をしたとしましょう。複数のプログラムからなるベンチマークの実行時間をどのようにまとめればよいのでしょう？それぞれのベンチマークの実行時間を同じにすることはできません。単純に実行時間の平均を取ると、実行時間の長いプログラムのウェイトが大きいことになってしまいます。しかし、ベンチマークの実行時間は入力データとの関係で決まり、それが長いからと言って全体に対する影響力が大きいとはいえません。そこで、まず、皆が持っているマシンを基準マシンとし、評価するマシンとの相対値を取ります。多数のプログラムを実行した場合、この相対値の相乗平均（幾何平均）を取ります。この方法は、プログラムの実行時間が違っても、基準マシンが変わっても、皆が同じものを使えば、一貫性のある結果がでます。ただし、これで得られた結果は、あくまで目安に過ぎません。相乗平均には非線形性があるので、ベンチマークのプログラムを組み合わせて走らせた場合に平均的にこの数値分速くなることはないのです。

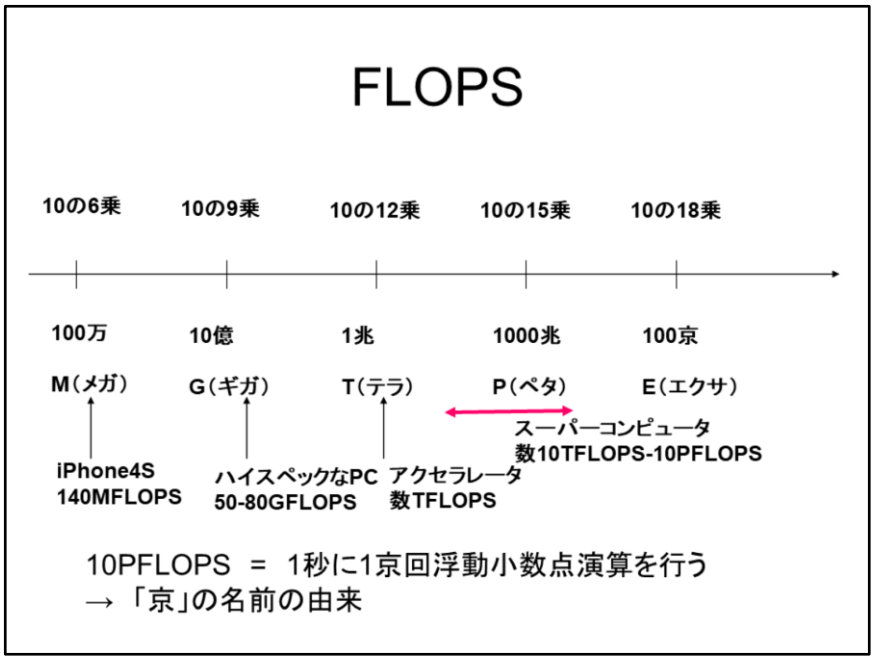
次に結果の報告については、他の人が同じマシンを使って同じプログラムを走らせた場合、同じ結果が出るように、つまり再現性があるように、ハードウェアの諸元をはじめ、ソフトウェアについてもOSの種類、バージョン、コンパイラの種類、オプションなどを報告するように心がけてください。

Description	Name	Instruction Count × 10⁹	CPI	Clock cycle time (seconds × 10⁹)	Execution Time (seconds)	Reference Time (seconds)	SPEC Ratio
Interpreted string processing	perl	2,118	0.75	0.4	637	9,770	15.3
Block-sorting compression	bzip2	2,389	0.85	0.4	817	9,650	11.8
GNU C compiler	gcc	1,050	1.72	0.4	724	8,050	11.1
Combinatorial optimization	mcf	336	10.00	0.4	1,345	9,120	6.8
Go game (AI)	go	1,658	1.09	0.4	721	10,490	14.6
Search gene sequence	hmmer	2,783	0.80	0.4	890	9,330	10.5
Chess game (AI)	sjeng	2,176	0.96	0.4	837	12,100	14.5
Quantum computer simulation	libquantum	1,623	1.61	0.4	1,047	20,720	19.8
Video compression	h264avc	3,102	0.80	0.4	993	22,130	22.3
Discrete event simulation library	omnetpp	587	2.94	0.4	690	6,250	9.1
Games/path finding	astar	1,082	1.79	0.4	773	7,020	9.1
XML parsing	xalancbmk	1,058	2.70	0.4	1,143	6,900	6.0
Geometric Mean							11.7

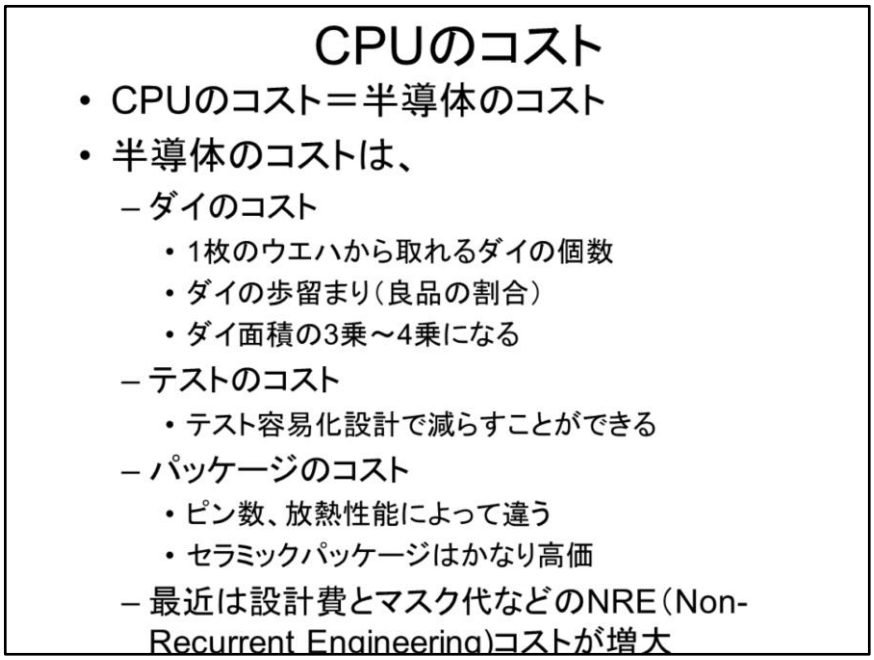
FIGURE 1.20 SPECINTC2006 benchmarks running on AMD Opteron X4 model 2356 (Barcelona). As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios. Figure 5.40 on page 542 shows that mcf, libquantum, omnetpp, and xalancbmk have relatively high CPIs because they have high cache miss rates.

この表はSPECINT2006ベンチマークをOpteran X4モデル2356というマシンで走らせた場合の性能を示しています。ここで、SPECratio（一番右の欄）が基準マシンとの性能（実行時間の逆数）の比ですがプログラムの種類によって非常に違っていることがわかります。最後にGeometric Mean（相乗平均）が示されています。

では、性能の単位として一般的に使われているものを紹介しておきましょう。MIPS（Million Instructions Per Second)は1秒間に何百万個命令を実行できるか、という尺度です。最近の高性能プロセッサではGIPS(Giga Instruction Per Second:1秒間に何十億個命令が実行できるか）が使われます。この尺度は実行する命令がどのようなものであるかを度外視しているため、違った命令セットの間で比較するのは意味がないです。しかし、同一の命令セットアーキテクチャ、例えばIntelのマシン間、ARMのマシン間で比較するならば、それなりに実感にあった値となります。
一方、MFLOPS（Million Floating Operations Per Second)は1秒間に何百万回浮動小数演算ができるかを示す性能指標です。計算機の命令セットが違っても、あるアルゴリズムを実行するのに必要な演算回数は同じなので、基本的にはこの指標はMIPSより命令セットの依存性が少ないです。しかし、乗算、除算、加算を同じ1回として数えていいのか、平方根や指数演算はどう数えるか、など問題があり、これらに重みを与えた正規化FLOPSが使われます。この正規化FLOPSは重みの与え方が公平ではないという批判はあるものの、科学技術計算が目的のスーパーコンピュータなどでは、一般的な性能指標として広く使われます。同じように信号処理用プロセッサでは、整数演算（特に積和演算）の実行回数で評価するMOPS（Million Operations Per Second)が用いられます。



この図は、それぞれの計算機がどの程度のFLOPS値を持つかを示しています。iPhone4Sは大体140MFLOPSあると言われています。現在のハイスペックなPCは50－80GFLOPS、GPUなどのアクセラレータ（一定の種類の演算の性能を上げる演算ユニット）は、TFLOPSを実現します。スーパーコンピュータはさらに3桁上のPFLOPSクラスの性能を実現します。日本最速のスーパーコンピュータ「京」は10PFLOPSを実現します。これは1秒間に1京回の演算を実現することに相当し、「京」の名前の由来になっています。現在の世界最速のコンピュータは中国の天河2で、30PFLOPSの性能を持ちます。しかし、これはLinpackという行列計算のカーネルを使って測定した数値であり、現実的なプログラムがこの速度で動くわけではありません。



CPUのコストは、半導体のコストによって決まります。半導体のコストは、ダイ（次のページの図）のコスト＋テストのコスト＋パッケージのコストで計算できます。ダイのコストは、ウエーハ1枚のコストを（1枚のウエーハ（次のページの図）から取れるダイの個数とダイの歩留まり（良品の割合）の積）で割ったものになります。ダイの面積が増えるほど、1枚当たりから取れる数が減ります。また、ダイの歩留まりは、半導体の欠損がどの程度発生するかによって決まるのですが、やはり面積が大きくなるほど悪くなります。ざっくり考えてダイのコストはダイの面積の3乗から4乗になると言われています。しかし、一部に欠損があっても動作するように設計する冗長設計によって、歩留まりは改善することができます。半導体は高額なテスターを使って、正しく動作するかチェックします。このコストも馬鹿にならない位大きくなります。これはテスト容易化設計で、テスト工程を簡単にすることで減らすことができます。さらにパッケージのコストも掛かります。これは、ピン数の多く放熱特性に優れたセラミックパッケージを使う場合、高額になります。電力とピン数を削減してプラスチックの安いパッケージにすることができれば削減ができます。最近の新しい半導体プロセス技術を使うと、設計費、IP代、マスク代などのNRE（Non-Recurrent Engineering)コスト、すなわち一回だけ掛かる製造費が非常に大きくなっています。

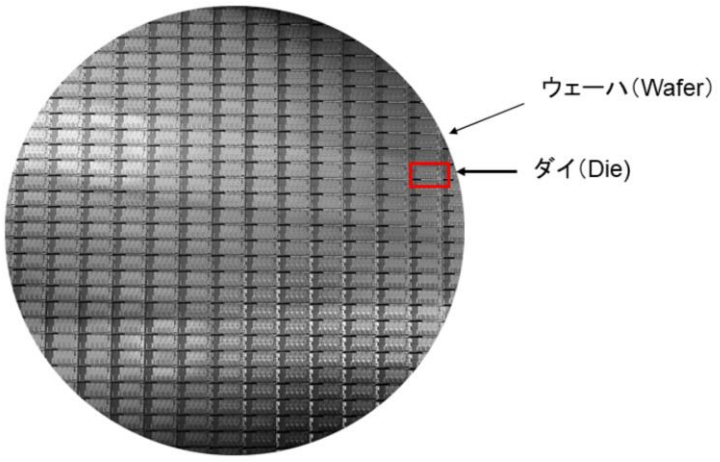
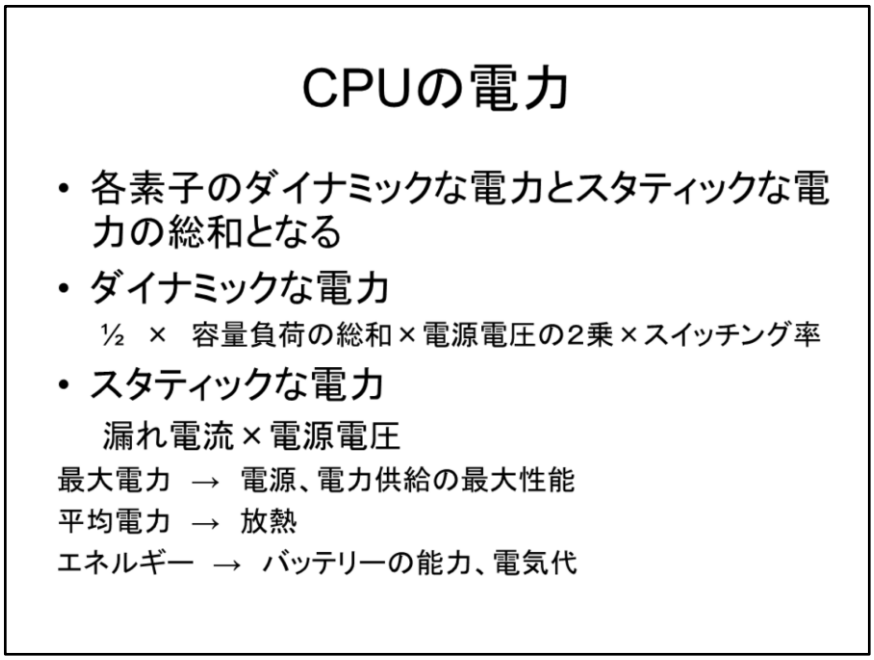


Figure 1.15 This 300 mm wafer contains 280 full Sandy Bridge dies, each 20.7 by 10.5 mm in a 32 nm process. (Sandy Bridge is Intel's successor to Nehalem used in the Core i7.) At 216 mm2, the formula for dies per wafer estimates 282. (Courtesy Intel.)

この図はIntelのCore i7(Sandy Bridge）のウェーハ写真です。直径30センチの円盤上に長方形のダイが並んでいます。これを切り離して、パッケージに組み込んで半導体チップができます。周辺部の模様が欠けているダイはもちろん使えません。ウェーハは半導体の製造工程上、どうしても30センチ程度の円盤になるので、ダイの面積が増えると、搭載できる個数が減ってしまうことがわかります。

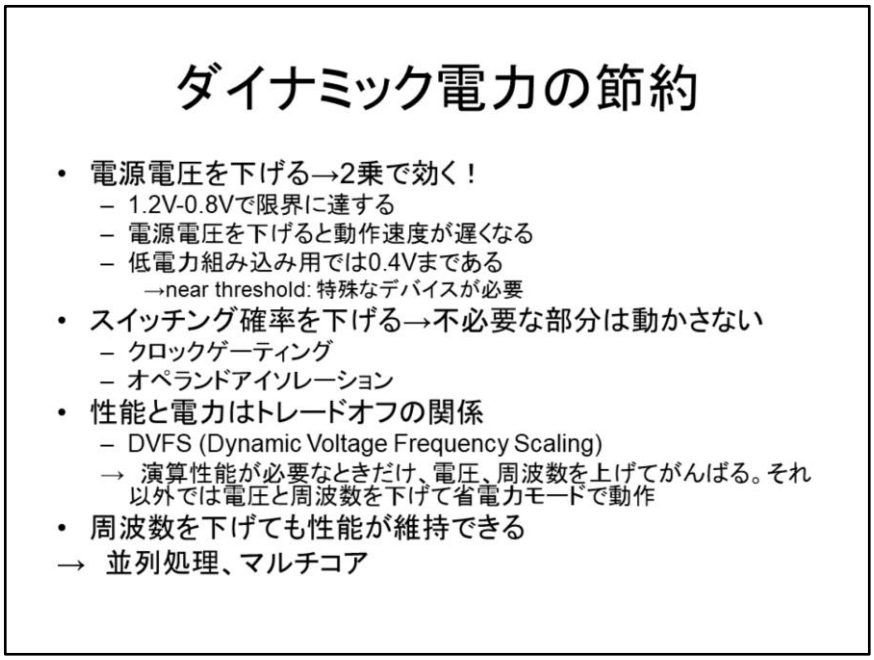


最後にCPUの消費電力に関してまとめておきましょう。CPUは半導体の各ゲートのダイナミック（動的）な電力とスタティック（静的、漏れ）な電力の総和になります。ダイナミックな電力は、CMOSを構成するトランジスタがON/OFFする時に流れる貫通電流（Internal Power)と、負荷となる容量を充放電するスイッチング電力（Switching Power)に分けられますが、貫通電流はトランジスタ内部に等価的な容量を想定して考え、これを負荷容量に含めて考えます。そうすると、1／2×容量負荷の総和×電源電圧の2乗×スイッチング率で求められます。容量負荷の総和は、あまり多数の出力を繋ぎ過ぎない（ファンアウトを取り過ぎない）などの設計上の工夫で減らすことはできますが、プロセス技術で大体決まってしまいます。電源電圧が2乗で効くことに注目してください。これが一つの理由となり、電源電圧は30年前に標準であった5Vから1V以下まで下がりました。コンピュータの設計上重要なのはスイッチング率です。スイッチング率は周波数で決まります。すなわち高速に動かすと電力が増えます。

スタティックな電力は、CMOSトランジスタのソースードレイン間、ゲートーソース間の漏れ電流によって生じます。最近プロセス技術が進んでトランジスタのサイズが小さくなるにつれてその割合が増えてきました。スタティックな電力は動かなくても消費されるので、バッテリー駆動の製品ではとりわけ重要です。

電力には最大電力、平均電力、エネルギーがあります。最大電力は瞬間的に消費される最大の電力で、電源の供給能力の最大性能を決めます。平均電力は平均的

に消費される電力で、放熱性能がこれに対応できなければならないです。またエネルギーは一定のプログラムを実行するのに必要な時間に電力をかけたもので、バッテリーの能力や電気代に影響します。エネルギーは、時間に比例するので、高速に実行して早く終わらせてしまえば小さくなります。しかし、高速に実行すると電力は増えるので、両者を良く考える必要があります。

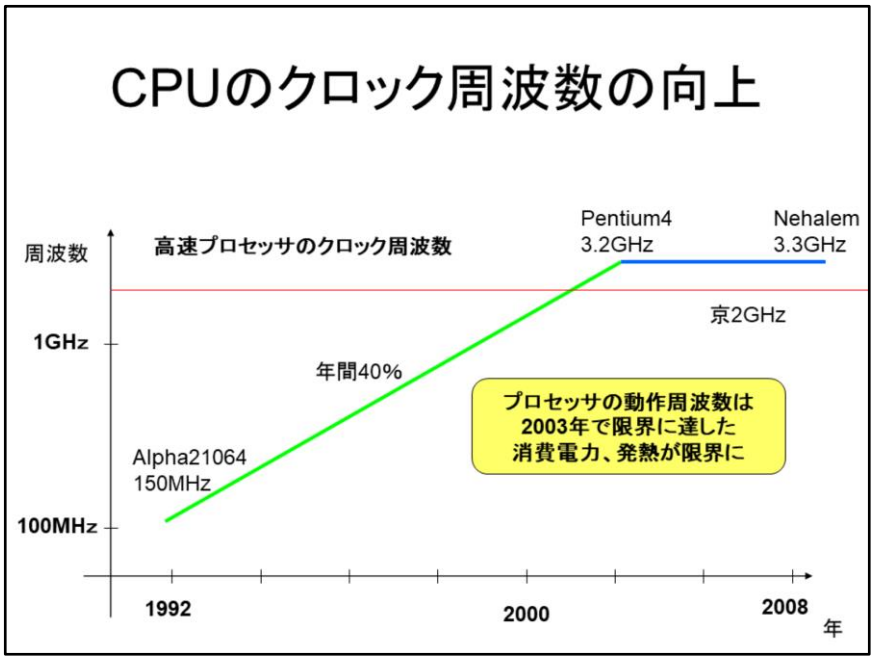


ダイナミックな電力を減らすにはどうすれば良いでしょう？一番簡単な方法は、電源を下げることです。このため、ディジタル回路の電源電圧はこの30年間で5Vから1Vくらいまで下がりました。しかし、0.8Vくらいで限界に達してしまいました。同じトランジスタで、電源電圧を下げると動作速度が遅くなります。現在、低電力用のICでは0.4V程度で動きます。これらはニアスレッショルド（near threshold)といってスレッショルドに近い電源電圧で動かします。この場合、ダイナミックな電力を極端に減らすことができるのですが、動作速度は落ちます。

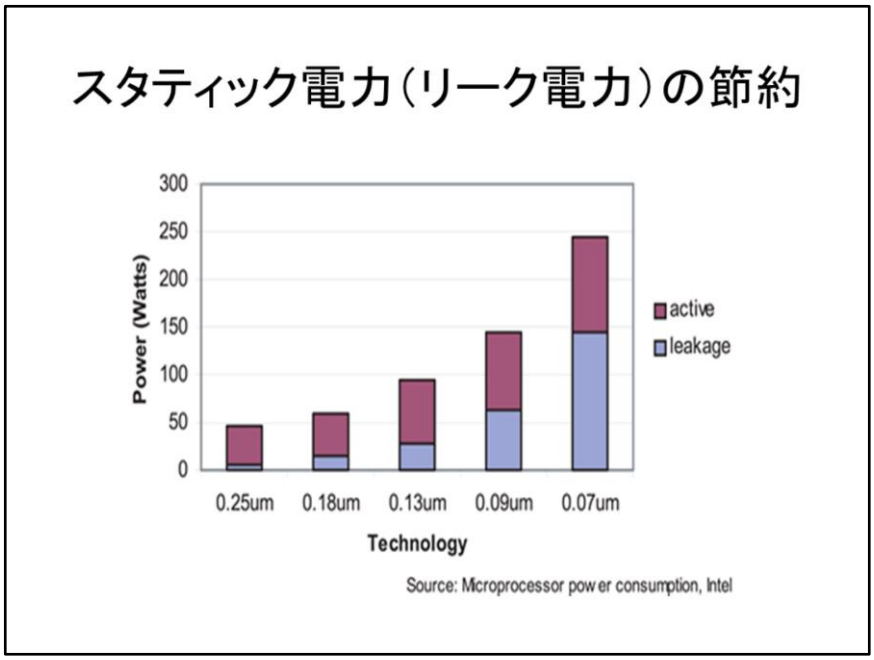
一方、スイッチング確率を下げるためには周波数を下げればよいのですが、これではもちろん動作速度が落ちます。不必要なスイッチを減らすことにより、動作速度を落とさず電力を落とせる可能性があります。クロックゲーティングは、使わないレジスタ等のクロックを止めてしまいます。またオペランドアイソレーションは演算に用いないデータの変化をなくすテクニックです。POCOは両方ともやってないので、LD命令やST命令でALUを使ってないときも動いています。これをとめることで電力を節約できます。

電圧、周波数を上げれば動作速度は上がりますが、電力が増えます。つまり性能と電力はトレードオフの関係にあります。では性能が必要なときだけ、電圧を上げ、周波数を上げて性能を上げてやり、さほど必要としないときはこれらを落として電力を節約することができます。このような手法はDVFS（Dynamic Voltage Frequency Scalling)と呼び、皆さんの使うPCに普通に使われています。

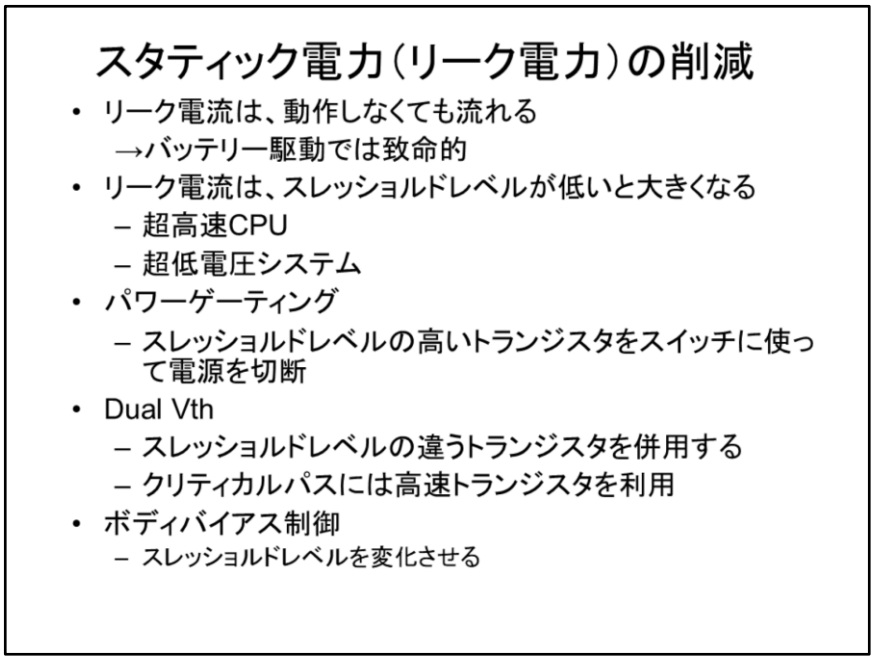
マルチコア、メニーコアを使う並列処理も、周波数を上げずに性能を維持する方法のひとつです。



単一プロセッサの周波数は1990年代には年間40%のペースで増強されました。消費電力は周波数に比例するため、CPUの消費する電力はどんどん増えて行き、2003年前後に放熱の限界に達してしまいました。これより先は、周波数を増やすよりも、チップ内のCPUのコア数を増やすマルチコアがコンピュータの主流となりました。



電子回路の時間に紹介しましたが、CMOS回路は片方のトランジスタがONの時はペアのトランジスタがOFFになっていて、回路が切れています。しかし、OFFのトランジスタスタも、ソースードレイン、ゲートーソース間に一定の電流が漏れてしまいます。スタティック電力はこれが原因で生じるので、リーク電力とも呼ばれます。リーク電力は、プロセス技術が進んでトランジスタが微細に作られれば作られるほど増える傾向にあり、最近のプロセスでは電力のうちの大きな要因を占めます。この図は10年ほど前にIntelが示した予測で、現状はこれほど増えてはいないものの、場合によってはかなりの割合を占めます。

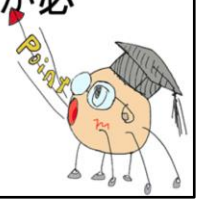


リーク電流は、動作しなくても流れます。したがってスイッチング率を減らす手法は使えません。止めておいても電源を入れておくと、電力を消耗してしまうため、バッテリー駆動の製品では致命的です。また、リーク電流はトランジスタのスレッショルドレベルが低いと大きくなります。スレッショルドレベルが低いトランジスタは高速に動作するため、高速CPUでは大きなリーク電流が流れます。また、ダイナミック電力を減らすために電源電圧を下げた際にも動作するためにはスレッショルドレベルが低くなければなりません。このため、電源電圧が低い領域で動作する超低電圧CPUでも漏れ電流の割合が大きくなります。

これを押さえるには、スレッショルドレベルが高いトランジスタをスイッチに使って、使っていない場合に電源を切ってしまうパワーゲーティングという手法が一般的に使われます。また、スレッショルドレベルの違うトランジスタを用意しておき、クリティカルパスには高速で漏れ電流の大きなトランジスタを使い、そうでないところには遅いけれど漏れ電流の小さなトランジスタを使うDual Vthという手法も一般的です。最近のトランジスタの中には、ボディ（サブストレート）に電圧を掛けることで、スレッショルドを変えることができるものがあり、これを使って漏れ電流と性能のバランスを使い方に応じて制御する方法も使われます。

ここまでのまとめ

- POCOではCPIは1
- 性能は動作周波数で決まる
- コストは面積＝ゲート数で決まる
- 電力は動作周波数、ゲート数、スイッチング率で決まる
- 上記を評価するには論理合成・圧縮が必要！



さて、ここまでのまとめを示します。ではPOCOの性能、コスト、電力を評価するにはどうすれば良いか？というと論理合成、圧縮を行って、ゲートレベルに落としてやる必要があります。この方法は来週紹介します。

演習

multを利用して0番地の数の3乗を計算せよ
提出物はsanjo.asm

演習は、2乗の例を拡張し、multというサブルーチンを2回呼んでください。