CPUの性能評価と高速化

慶應義塾大学
天野

シングルサイクル版vs.
マルチサイクル版

- CPUのマイクロアーキテクチャは
  性能、コスト（面積）、消費電力で評価する。
- ここでは性能とコスト（ハードウェア量）を簡単に評価する。
- 本格的な評価は論理合成をやった後

では、次にシングルサイクル版とマルチサイクル版のどちらのマイクロアーキテクチャが有利なのかを評価しましょう。ここでは性能とハードウェア量を簡単に見積もって比較しましょう。本格的な評価は論理合成をやった後で、多分来年のコンピュータアーキテクチャになると思います。

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
→ 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

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、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。

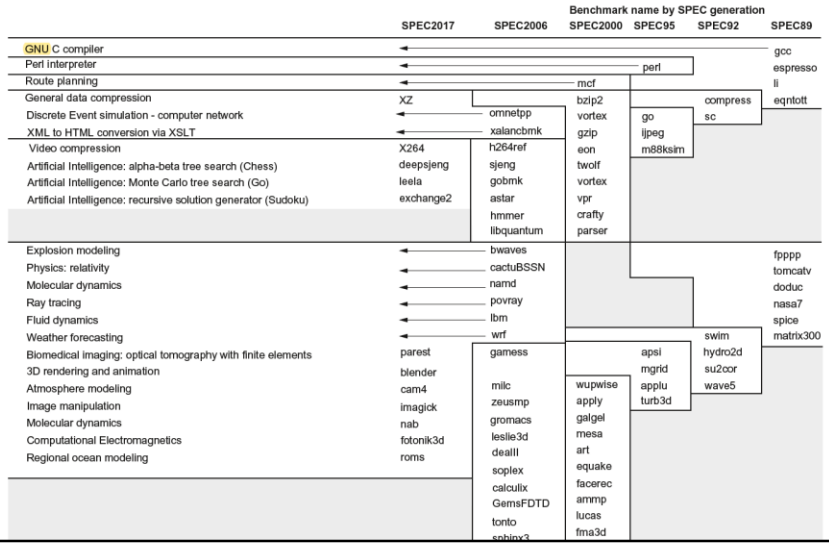
実行時間の評価

- プログラムを走らせてその実行時間を比較
  - デスクトップ、ラップトップ：ＳＰＥＣベンチマーク
  - サーバー：TPC
  - スーパーコンピュータ：Linpack, LLL
  - 組み込み：EEMBC, MiBench
- 走らせるプログラム
  - ○実プログラムによるベンチマーク集
  - △カーネル：プログラムの核となる部分
  - ×トイプログラム：Quicksort, 8queen, エラトステネスの篩
  - ×合成ベンチマーク：Whetstone, Dhrystone

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。

SPECベンチマークの変遷



	Benchmark name by SPEC generation					
	SPEC2017	SPEC2006	SPEC2000	SPEC95	SPEC92	SPEC89
GNU C compiler						gcc
Perl interpreter				perl		espresso
Route planning			mcf			li
General data compression	XZ		bzip2		compress	eqntott
Discrete Event simulation - computer network		omnetpp	vortex	go	sc	
XML to HTML conversion via XSLT		xalancbmk	gzip	ijpeg		
Video compression	X264	h264ref	eon	m88ksim		
Artificial Intelligence: alpha-beta tree search (Chess)	deepsjeng	sjeng	twolf			
Artificial Intelligence: Monte Carlo tree search (Go)	leela	gobmk	vortex			
Artificial Intelligence: recursive solution generator (Sudoku)	exchange2	astar	vpr			
		hmmer	crafty			
		libquantum	parser			
Explosion modeling		bwaves				fpppp
Physics: relativity		cactuBSSN				tomcatv
Molecular dynamics		namd				doduc
Ray tracing		povray				nasa7
Fluid dynamics		lbm				spice
Weather forecasting		wrf			swim	matrix300
Biomedical imaging: optical tomography with finite elements	parest	gamess		apsi	hydro2d	
3D rendering and animation	blender			mgrid	su2cor	
Atmosphere modeling	cam4	milc	wupwise	applu	wave5	
Image manipulation	imagick	zeusmp	apply	turb3d		
Molecular dynamics	nab	gromacs	galgel			
Computational Electromagnetics	fotonik3d	leslie3d	mesa			
Regional ocean modeling	roms	dealII	art			
		soplex	equake			
		calculix	facerec			
		GemsFDTD	ammp			
		tonto	lucas			
		sphinx3	fma3d			

評価のまとめ方、報告の仕方

- 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扱うか？
  - 基準マシンを決めて相対値を取る
  - 複数のプログラムに対しては相乗平均を取る

  〇プログラムの実行時間、基準マシンに依らない一貫性のある結果が得られる

  ×非線形が入る
- 結果の報告
  - 再現性があるように
  - ハードウェア：　動作周波数、キャッシュ容量、主記憶容量、ディスク容量など
  - ソフトウェア：OSの種類、バージョン、コンパイラの種類、オプションなど

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。

性能比較の一例　AMD　A10-6800K　vs. Intel Xeon ES-2690

Benchmarks	Sun Ultra Enterprise 2 time (seconds)	AMD A10-6800K time (seconds)	SPEC 2006Cint ratio	Intel Xeon E5-2690 time (seconds)	SPEC 2006Cint ratio	AMD/Intel times (seconds)	Intel/AMD SPEC ratios
perlbench	9770	401	24.36	261	37.43	1.54	1.54
bzip2	9650	505	19.11	422	22.87	1.20	1.20
gcc	8050	490	16.43	227	35.46	2.16	2.16
mcf	9120	249	36.63	153	59.61	1.63	1.63
gobmk	10,490	418	25.10	382	27.46	1.09	1.09
hmmer	9330	182	51.26	120	77.75	1.52	1.52
sjeng	12,100	517	23.40	383	31.59	1.35	1.35
libquantum	20,720	84	246.08	3	7295.77	29.65	29.65
h264ref	22,130	611	36.22	425	52.07	1.44	1.44
omnetpp	6250	313	19.97	153	40.85	2.05	2.05
astar	7020	303	23.17	209	33.59	1.45	1.45
xalancbmk	6900	215	32.09	98	70.41	2.19	2.19
Geometric mean			31.91		63.72	2.00	2.00

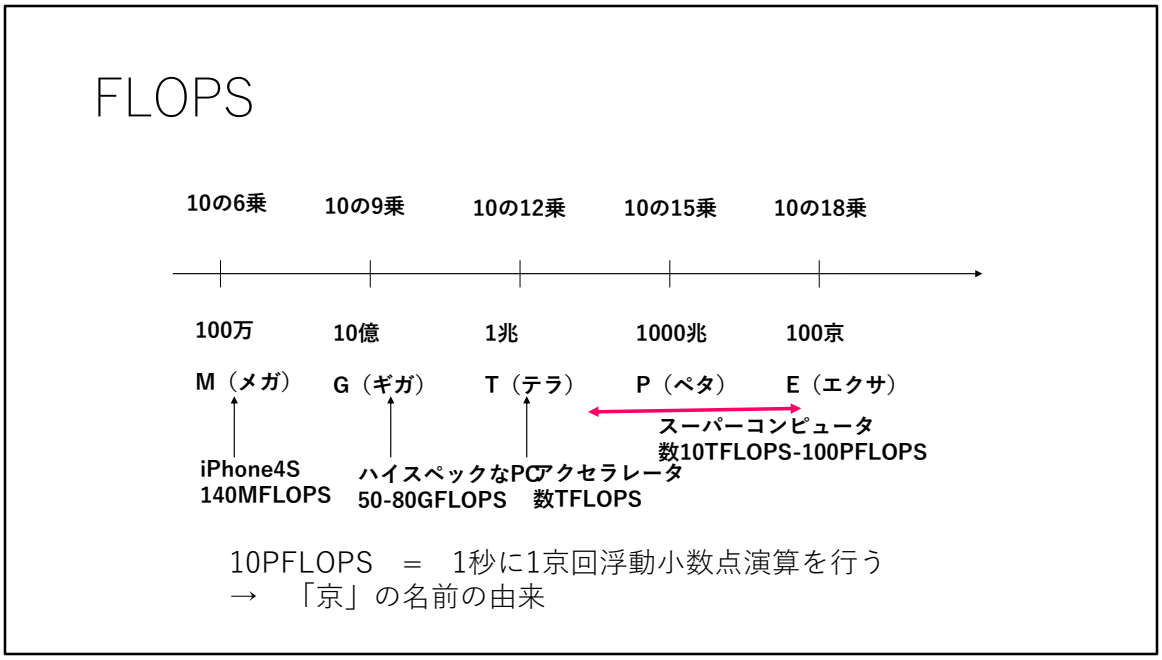
Figure 1.19 SPEC2006Cint execution times (in seconds) for the Sun Ultra 5—the reference computer of SPEC2006—and execution times and SPECRatios for the AMD A10 and Intel Xeon E5-2690. The final two columns show the ratios of execution times and SPEC ratios. This figure demonstrates the irrelevance of the reference computer in relative performance. The ratio of the execution times is identical to the ratio of the SPEC ratios, and the ratio of the geometric means (63.7231.91/20.86 = 2.00) is identical to the geometric mean of the ratios (2.00). Section 1.11 discusses libquantum,

SPEC2006Cintを用いたAMDのA10とIntel Xeon の比較です。Sun Ultraを基準としたSPECratioの比は、実行時間の比と等しくなります。全体のGeometric mean(相乗平均）も示されています。

MIPS, MFLOPS, MOPS

- MIPS （Million Instructions Per Second）
  - 一秒間に何百万個命令が実行できるか？
  - 一命令がどの程度の機能を持っているかが入っていない
  - 異なる命令セット間の比較には無意味な基準
  - しかし分かりやすいし、IntelやARM間の比較にならばそれなりに有効
- MFLOPS (Million FLoating Operations Per Second)
  - 一秒間に何百万回浮動小数演算ができるか？
  - 本来、MIPSより公平な基準だが、平方根や指数などの命令を持つかどうかで問題が生じる→正規化FLOPS
  - MOPS(Million Operations Per Second)はDSP（信号処理用プロセッサ）など整数演算の実行回数で評価する（積和演算回数だったりする）。

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。かつて世界一を奪取した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。現在開発中の富岳はこの100倍の性能を目指しています。現在の世界最速のコンピュータはアメリカのSummitで、148PFLOPS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

- CPUのコスト＝半導体のコスト
- 半導体のコストは、
  - ダイのコスト
    - 1枚のウエハから取れるダイの個数
    - ダイの歩留まり（良品の割合）
    - ダイ面積の3乗～4乗になる
  - テストのコスト
    - テスト容易化設計で減らすことができる
  - パッケージのコスト
    - ピン数、放熱性能によって違う
    - セラミックパッケージはかなり高価
  - 最近は設計費とマスク代などのNRE（Non-Recurrent Engineering)コストが増大

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。

Trends in Cost

半導体のコスト

- **半導体のコストの式**

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

- Bose-Einstein **式**:

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^N$$

- Defects per unit area = 0.016-0.057 defects per square cm (2010)
- N = process-complexity factor = 11.5-15.5 (40 nm, 2010)

半導体のコストを計算する式を示しましょう。ウェーハから取れる数が増え、歩留まりが高くなればコストは下がります。それぞれの係数は、ここに示すような数値なので、これを代入すれば大体のコストの比を求めることができます。

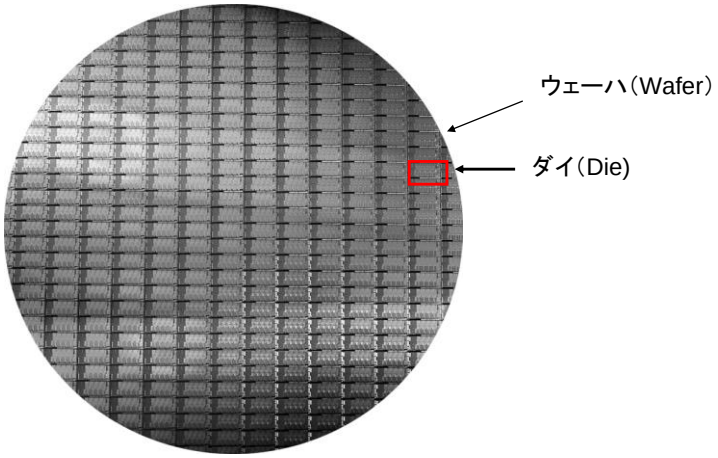


Figure 1.15 This 300 mm wafer contains 280 full Sandy Bridge dies, each 20.7 by 10.5 mm in a 32 nm process. (Sandy Bridge is Intel's successor to Nehalem used in the Core i7.) At 216 mm2, the formula for dies per wafer estimates 282. (Courtesy Intel.)

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。

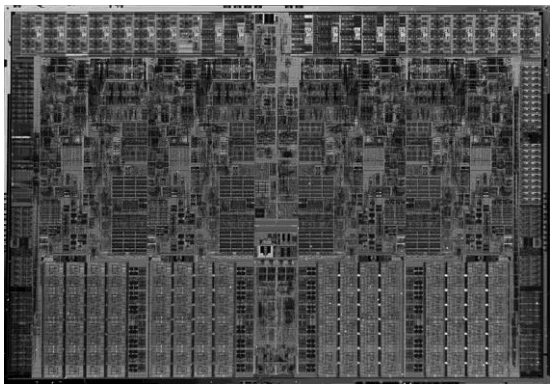


Figure 1.13 Photograph of an Intel Core i7 microprocessor die, which is evaluated in Chapters 2 through 5. The dimensions are 18.9 mm by 13.6 mm (257 mm2) in a 45 nm process. (Courtesy Intel.)

これはIntel Core i7のダイ写真です。非常に面積が大きいことが分かります。パターンを見ると4つのコアがわかります。

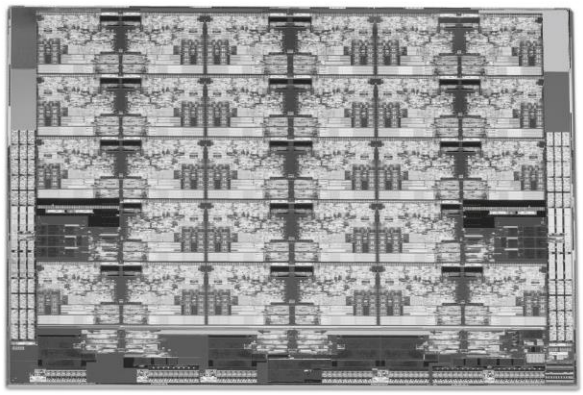


Figure 1.14 Photograph of an Intel Skylake microprocessor die, which is evaluated in Chapter 4.

これは、より最近のIntel　Skylakeのチップ写真です。コア数が増加しているのがわかります。

CPUの電力

- 各素子のダイナミックな電力とスタティックな電力の総和となる
- ダイナミックな電力

  ½　×　容量負荷の総和×電源電圧の２乗×スイッチング率
- スタティックな電力

  漏れ電流×電源電圧

最大電力　→　電源、電力供給の最大性能

平均電力　→　放熱

エネルギー　→　バッテリーの能力、電気代

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。

ダイナミック電力の節約

- 電源電圧を下げる→2乗で効く！
  - 1.2V-0.8Vで限界に達する
  - 電源電圧を下げると動作速度が遅くなる
  - 低電力組み込み用では0.4Vまである
    →near threshold: 特殊なデバイスが必要
- スイッチング確率を下げる→不必要な部分は動かさない
  - クロックゲーティング
  - オペランドアイソレーション
- 性能と電力はトレードオフの関係
  - DVFS (Dynamic Voltage Frequency Scaling)

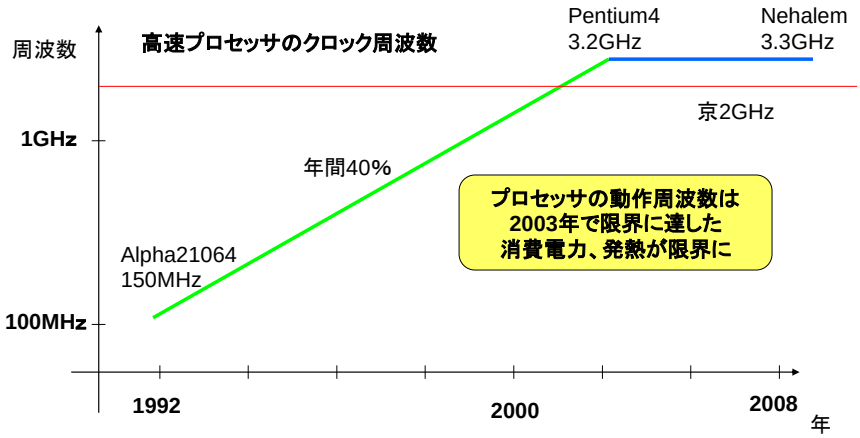
  →　演算性能が必要なときだけ、電圧、周波数を上げてがんばる。
  それ以外では電圧と周波数を下げて省電力モードで動作
- 周波数を下げても性能が維持できる

→　並列処理、マルチコア

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

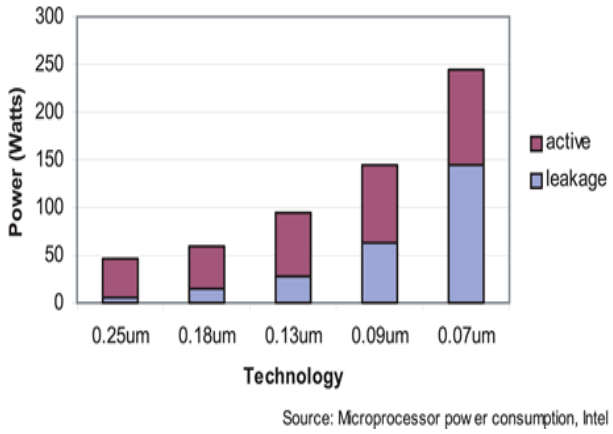
（Dynamic Voltage Frequency Scalling)と呼び、皆さんの使うPCに普通に使われています。
マルチコア、メニーコアを使う並列処理も、周波数を上げずに性能を維持する方法のひとつです。

CPUのクロック周波数の向上



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。

スタティック電力（リーク電力）の節約



Source: Microprocessor pow er consumption, Intel

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。

スタティック電力（リーク電力）の削減

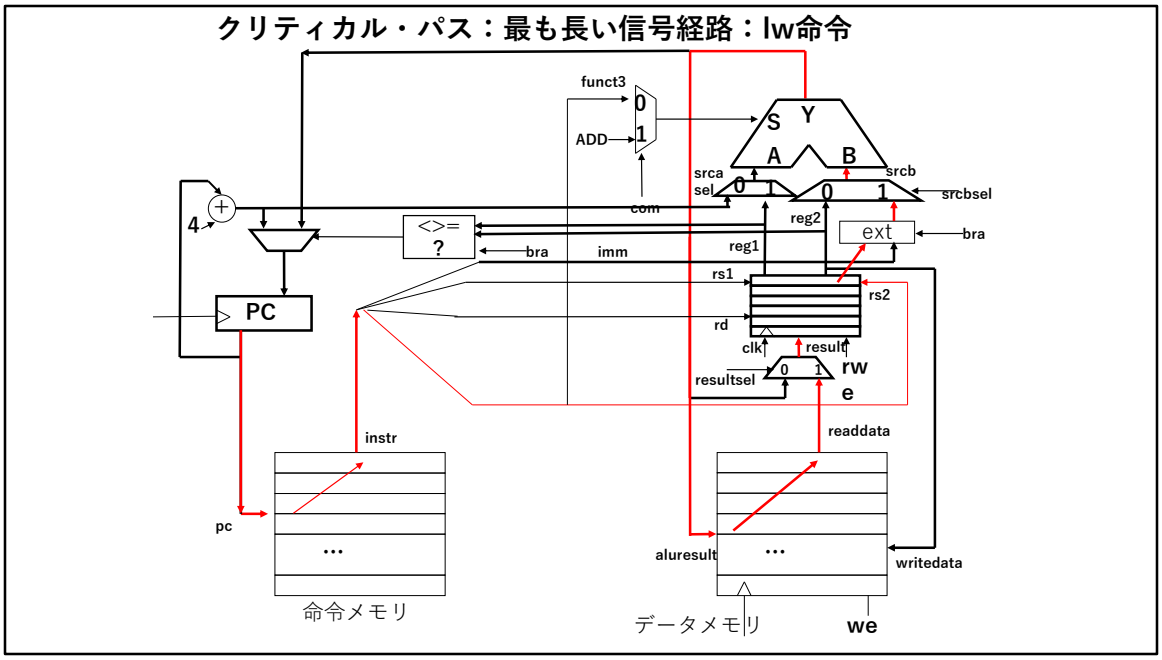
- リーク電流は、動作しなくても流れる
  →バッテリー駆動では致命的
- リーク電流は、スレッショルドレベルが低いと大きくなる
  - 超高速CPU
  - 超低電圧システム
- パワーゲーティング
  - スレッショルドレベルの高いトランジスタをスイッチに使って電源を切断
- Dual Vth
  - スレッショルドレベルの違うトランジスタを併用する
  - クリティカルパスには高速トランジスタを利用
- ボディバイアス制御
  - スレッショルドレベルを変化させる

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。

シングルサイクル版vs.
マルチサイクル版

- では実際にどうすれば良いのか？
- 性能
  - クリティカルパスにより動作周波数を見積もる
  - CPIはプログラムを実行して測定する
- コスト
  - 本来は論理合成して評価する
  - 今回は構成要素から見積もる
- 消費電力
  - 本来は論理合成して評価する
  - 今回は構成要素から見積もる→今回これは一番怪しい。。。

では、次にシングルサイクル版とマルチサイクル版のどちらのマイクロアーキテクチャが有利なのかを評価しましょう。ここでは性能とハードウェア量を簡単に見積もって比較しましょう。本格的な評価は論理合成をやった後で、多分来年のコンピュータアーキテクチャになると思います。



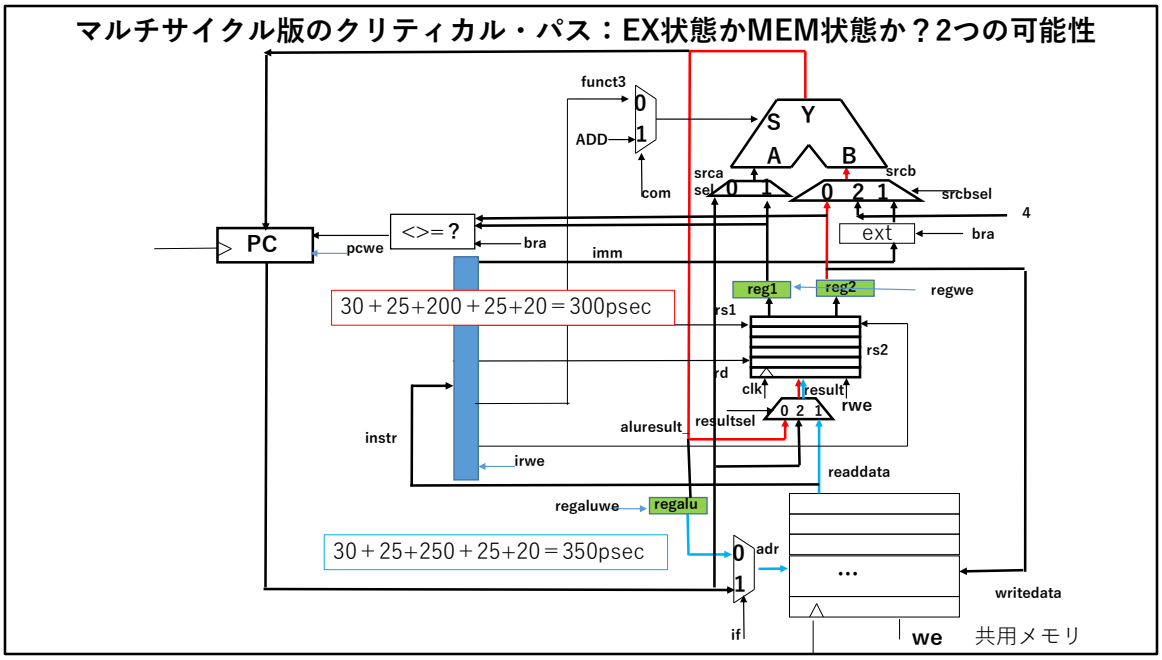
分岐命令用に拡張したRV32Iの構成を示します。RV32Iの分岐命令にはやることが二つあります。レジスタの比較（大小も含め）と、飛び先アドレスの計算です。どちらかにALUを使い、どちらかに専用の回路を設ける必要があります。ここでは、ALUで飛び先を計算することにします。飛び先アドレスを計算するために、PC＋４をALUのAポートに入れるために、マルチプレクサを付けます。B入力のextも、分岐命令のイミーディエイトデータをまとめるために、構造を変更する必要がありますが、この図には表れていません。符号拡張され0を補ったデータがB入力から入ると考えてください。読み出してきた二つのレジスタは、専用の比較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

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、1.05GH z で動作することがわかります。



マルチサイクル版のクリティカルパスには二つ可能性があります。EXEC状態で、ALUを利用する場合（赤いパス）と、MEM状態でメモリから読み出しを行う場合（青いパス）です。ここではメモリの遅延が大きいため、後者の方（青い線）が大きくなり、350psecになります。

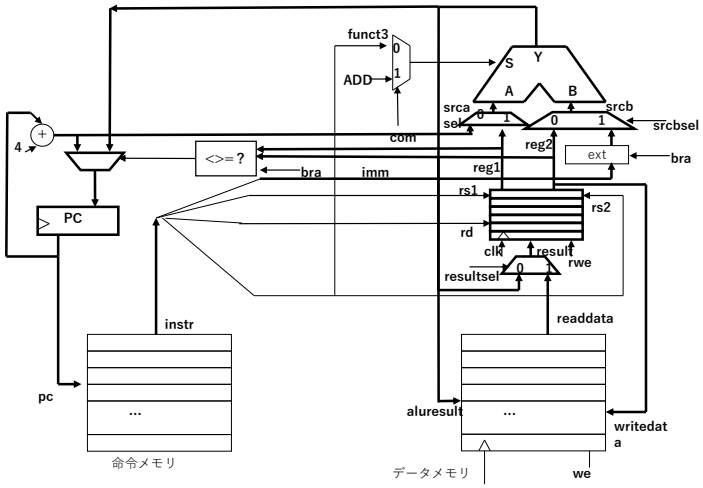
マルチサイクル版性能解析

- クリティカルパス：今回の仮定では
  - ALU：tpcq+tmux+tALU+tmux+tsetup
    30+25+200+25+20＝300ps
  - メモリ：tpcq+tmux+tmem+tsetup
    30+25+250+25+20=350ps　（2.86GHｚ）
- 平均CPI
  - multの実行結果から3.29
- 350×3.29＝1151.5
- これはシングルサイクルの950より長い(つまり遅い)
- なぜだろう？

この二つのパスを先ほどの値を入れて検討するとこのようになります。マルチサイクル版は、メモリアクセス命令ではCPI=4、それ以外ではCPI=3となります。multの評価結果から平均CPUは3.29になりました。シングルサイクルと性能を比較すると完敗です。これは、一命令あたりのクロックサイクル数が増えた割には、遅延時間が減っていないためです。クリティカルパスをきっちり4等分するのは不可能に近く、今回の実装はそこそこがんばっている方です。マルチサイクル版は、性能面ではシングルサイクルに勝てない場合が多いです。

コストの計算：シングルサイクル版

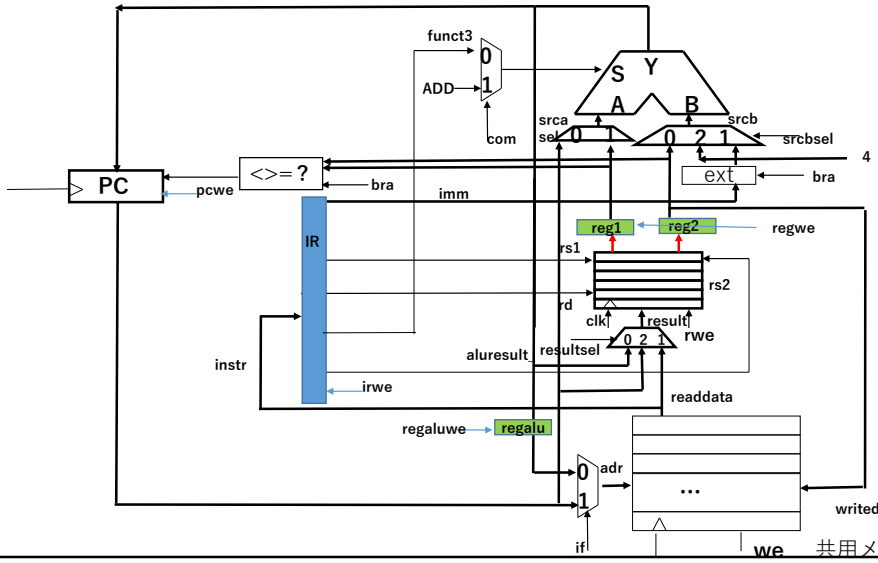
モジュール	個数
メモリ	2
レジスタファイル	1
ALU	1
加算器	1
マルチプレクサ	5
レジスタ	1
比較器	1



シングルサイクル版のコストを表に示します。

コストの計算：マルチサイクル版

モジュール	個数
メモリ	1
レジスタファイル	1
ALU	1
加算器	0
マルチプレクサ	5
レジスタ	5
比較器	1



一方、マルチサイクル版は、メモリが一つで済み、加算器がなくなった一方、レジスタが増えています。とはいえ、レジスタのハードウェア量はさほど大きくないことを考えると、コスト的にはかなり有利と言えると思います。ただし、このコストには制御回路のFSMのは含まれていないので注意は必要です。

半導体面積と電力の見積

- オクラホマ州立大学0.18μm実験用プロセスを利用
  - 昔のプロセスだが、相対値が問題なので、、、

	面積(μm²)	電力（mW）	シングルサイクル面積	マルチサイクル面積	シングルサイクル電力	マルチサイクル電力
レジスタファイル	226728	13.89＊0.49/10	226728	226728	0.68	0.68
32ビットレジスタ	7592	0.635＊0.44/10	7592	37960	0.0279	0.1395
マルチプレクサ	2278	0.925*0.97/10	11390	11390	0.0899	0.0899
ALU	50992	22.4*7.32/10	50992	50992	16.4	16.4
加算器	3808	1.53*6.68/10	3808	0	1.02	0
比較器	2687	1.02*3.65/10	2687	2687	0.037	0.037
総計			303197 0.55㎜角	329757 0.57㎜角	18.246ｍW 10MHｚ動作時	17.34ｍW 10MHｚ動作時

結果として、、、

- メモリのコストが入っていないのでマルチサイクル版がほぼすべての点で不利。
- 電力は小さいが、本当は周波数が高くなった分増えてしまう。

結果は、マルチサイクル版がどの項目も不利になってしまいます。電力はやや減っていますが、これは100MHｚ動作時に換算しているためで、マルチサイクル版はより高速に動かさないとシングルサイクル版と競うほど性能が出ないので、実際にはもっと消費します。ただしこの評価は、メモリのコスト、電力が反映されない点で不公平です。

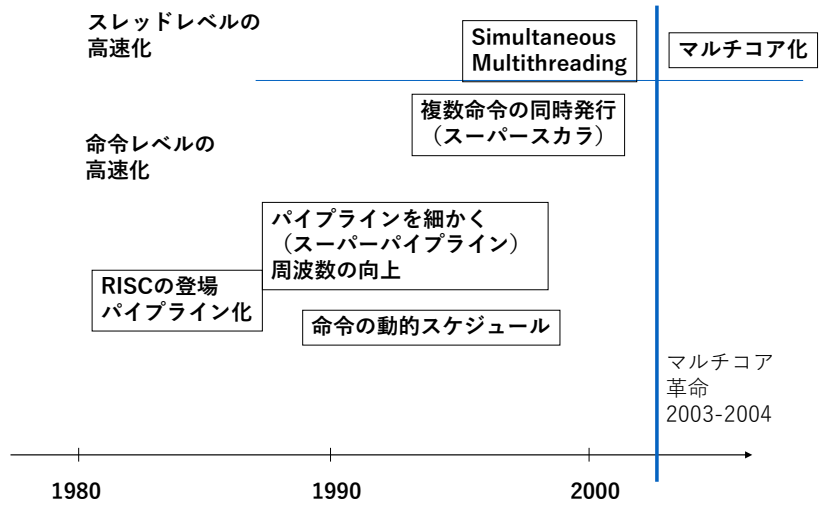
Principles

CPUをどのように高速化するか？

- 並列性を利用せよ
  - e.g. マルチプロセッサ, ディスク, メモリバンク, パイプライン処理,複数処理ユニット
- 局所性の原則
  - データと命令の再利用
- 共通の場合に注目せよ
  - Amdahl's Law

ではどのようにしてCPUの性能を上げれば良いのでしょうか？基本的な原則を三つ紹介します。まず、並列性は演算器レベルからプロセッサレベルまで様々なものがあり、これらは全てのレベルで利用します。局所性の原則は、プログラム格納型計算機の本質的な性質で、特にメモリシステムの所で紹介します。最後の一つは、設計する際に、できるだけ共通に使われる場合に注目して設計する方がうまく行くということです。これはAmdahlの法則で示されます。

高速化の流れ



どのようなテクニックを使って高速化をしていったのかをここで示します。CPUの高速化は様々なレベルで並列性を利用することにより実現してきました。このうち、パイプリアン化と複数命令の同時発行は来年のコンピュータアーキテクチャBで紹介します。また、2003年以降のマルチコアについてはコンピュータアーキテクチャAで紹介します。

Amdahlの法則

高速化の効果はそれが可能な部分の割合によって制限される

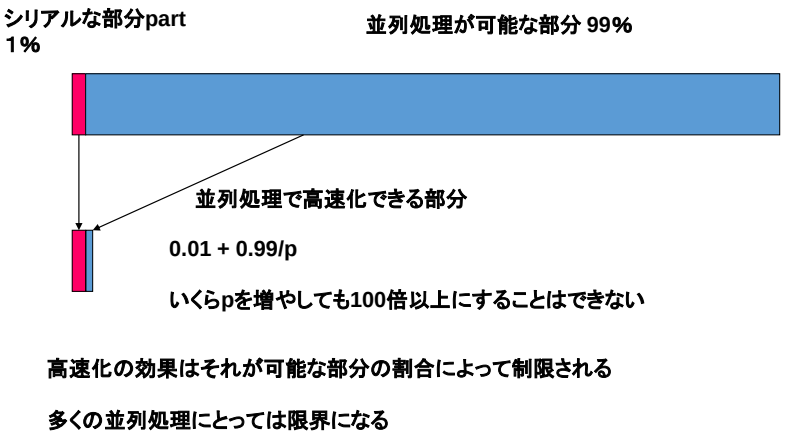
高速化後の時間＝　高速化前の時間×（（1－高速化が効く割り合い）＋
高速化が効く割り合い／スピードアップ）

高速化後の時間／高速化前の時間＝

（1－高速化が効く割り合い）＋高速化が効く割合／スピードアップ

Amdahlの法則は、高速化の効果はそれが可能な部分の割合によって制限される、というもので、ここに示す式で表されます。これは当たり前の法則なのですが、アーキテクチャ設計においては、とかくスピードアップの項に目が行くので、この落とし穴にハマります。いかにスピードアップの割合が高くても、高速化が効く割合が小さいと、その効果は制限されてしまいます。

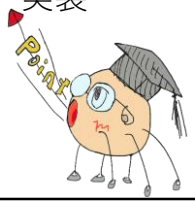
Amdahlの法則



Amdahlの法則は、並列処理の限界を示す式としても使われます。いくら並列処理で性能が高速化されても、逐次的に行わなければならない部分が少しでもあると、その部分によって性能は極端に制限されます。

性能とコストの比較のまとめ

- ISAが同じ場合、性能は、クロック周期とCPIで決まる。
  - クロック周期はクリティカルパスで決まる。
  - CPI（Clock cycles Per Instruction)は、シングルサイクル版は常に１だがマルチサイクル版は動作させるプログラムに依存
  - 性能比較は、遅い方の性能（速い方の実行時間）を基準にする。
- コストは必要モジュール数で評価したが、実装の状況により異なる。



性能とコストの比較の部分をまとめます。

演習1　性能評価

- 授業中のスライドの数値より、やや高速なメモリを利用したことで、tmem=180psecになった。この時、シングルサイクル版とマルチサイクル版の動作周波数をそれぞれ計算せよ。
- 0x400番地から並んでいる8個のデータの総和を求めるプログラムmsum.asmを実行し、シングルサイクル版とマルチサイクル版の実行時間を比較せよ。
- mult.asmの結果と合わせてシングルサイクル版との性能比率を
- 計算せよ。
  XXがYYのZZ倍速いという言い方で示せ。
- テキストファイルを提出のこと

演習２　Amdahlの法則

- メモリアクセス状態とメモリ状態を統合することにより、マルチサイクル実装においてsw命令、lw命令でも3クロックで処理が終わるようにした。
- しかしこのためクリティカルパスが40%伸びてしまった。
- msum.asmを実行する場合、この改造により性能はどうなるか？
- これもテキストファイルで提出せよ