計算機構成　第10回
POCOの性能評価と論理合成
テキスト7章

情報工学科
天野英晴

今回は今まで設計してきたPOCOの性能、コスト、消費電力を評価しましょう。今回、一部でHennessy&PattersonのComputer Architecture，Patterson&HennessyのComputer Organizationのスライドを使っています。感謝致します。これは、教育用に公開されているもので、違法にパクったわけではありません。

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。

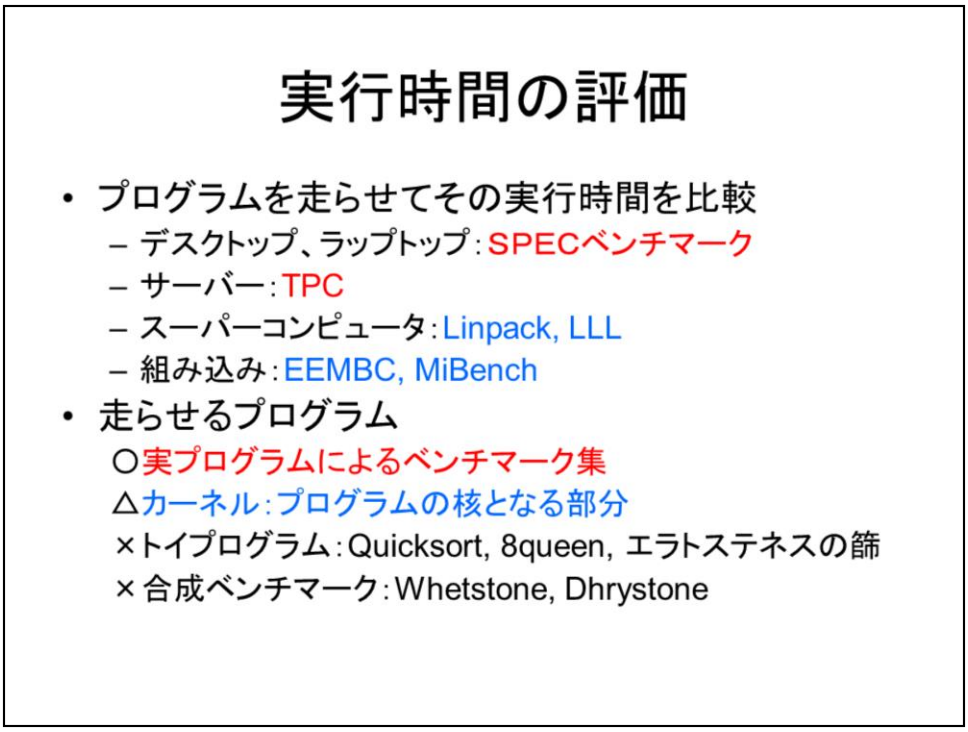
性能の比較

- 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

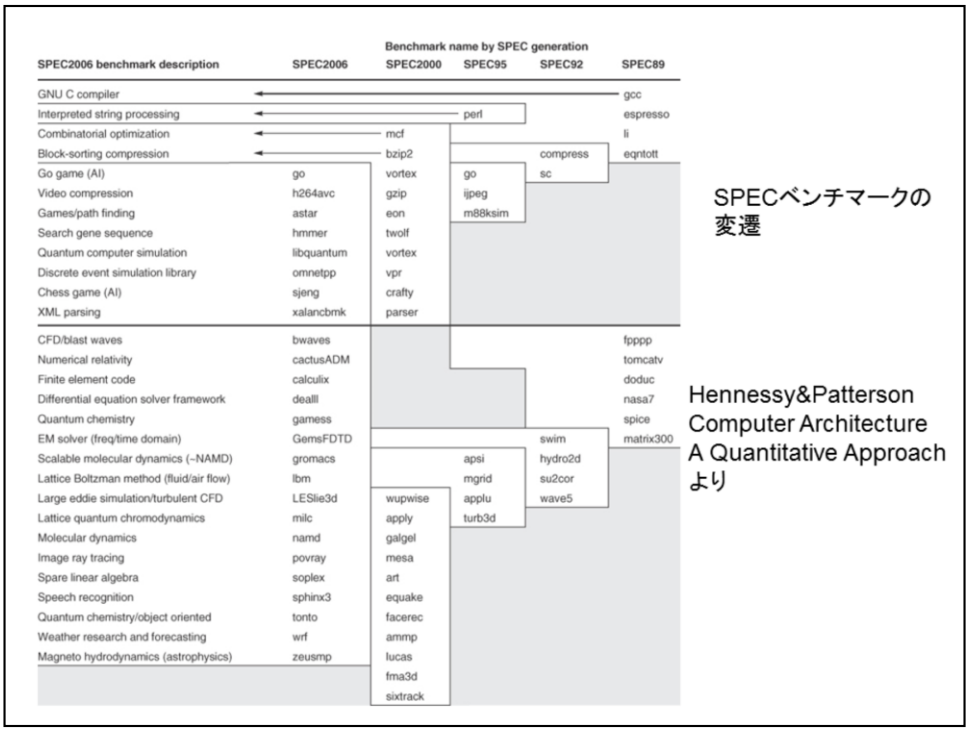
$$\frac{\text{CPU Aの性能}}{\text{CPU Bの性能}} = \frac{\text{CPU Bの実行時間}}{\text{CPU Aの実行時間}}$$

×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。

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

  ○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

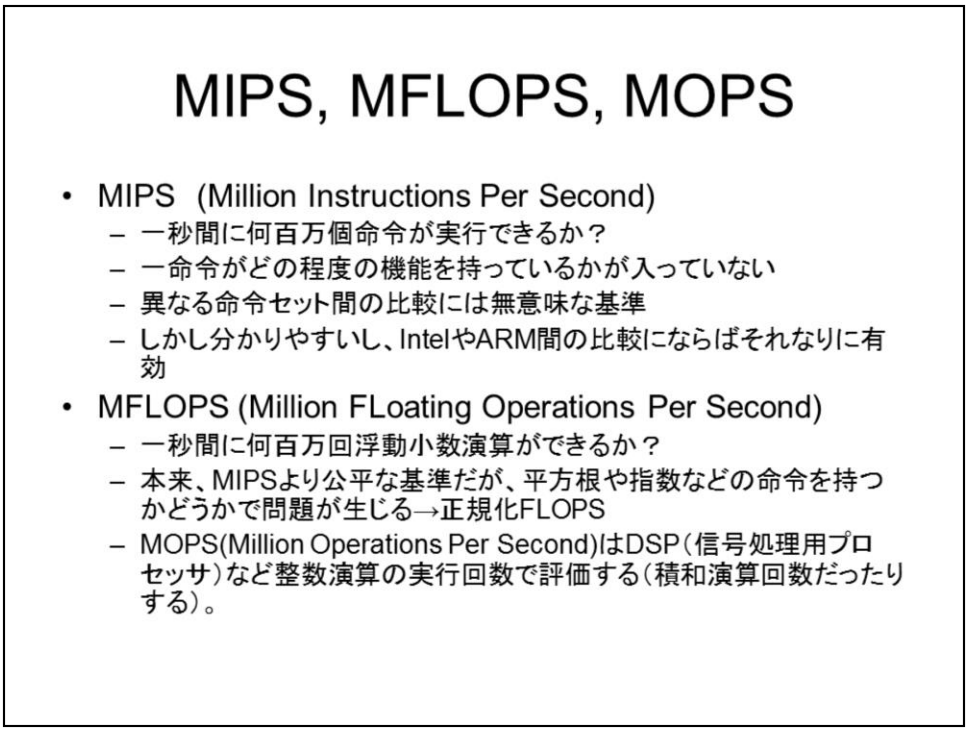
ではベンチマーク集を使って評価をしたとしましょう。複数のプログラムからなるベンチマークの実行時間をどのようにまとめればよいのでしょう？それぞれのベンチマークの実行時間を同じにすることはできません。単純に実行時間の平均を取ると、実行時間の長いプログラムのウェイトが大きいことになってしまいます。しかし、ベンチマークの実行時間は入力データとの関係で決まり、それが長いからと言って全体に対する影響力が大きいとはいえません。そこで、まず、皆が持っているマシンを基準マシンとし、評価するマシンとの相対値を取ります。多数のプログラムを実行した場合、この相対値の相乗平均（幾何平均）を取ります。この方法は、プログラ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。

Description	Name	Instruction Count $\times 10^9$	CPI	Clock cycle time (seconds $\times 10^{-9}$)	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

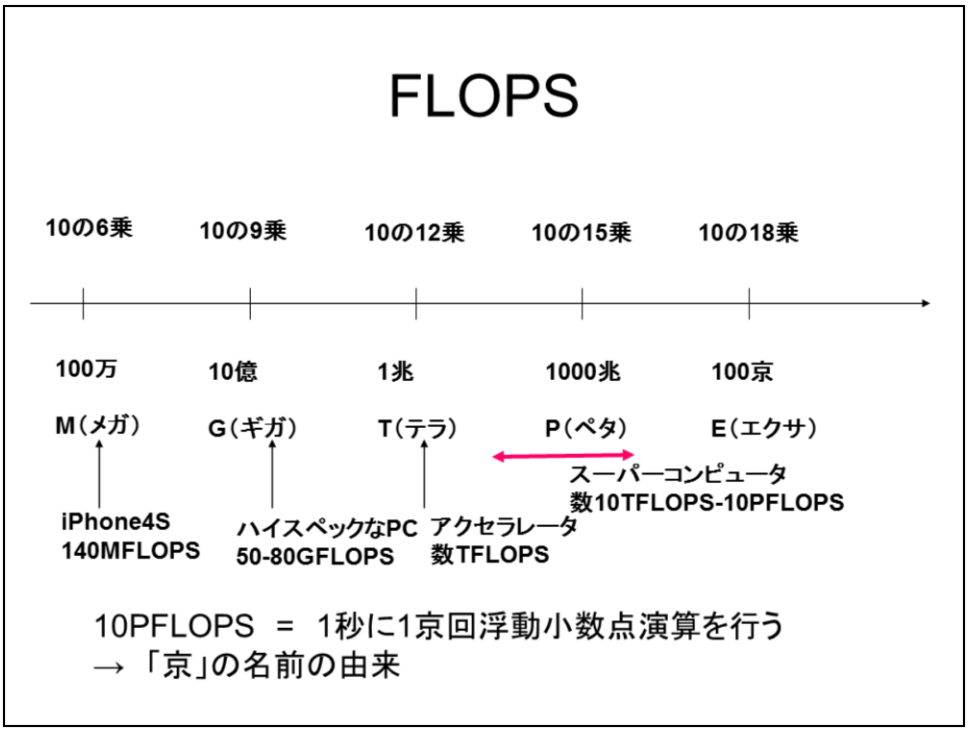
FIGURE 1.20 SPECINTC2006 benchmarks running on AMD Opteron X4 model 2356 (Barcelona). As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply 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。

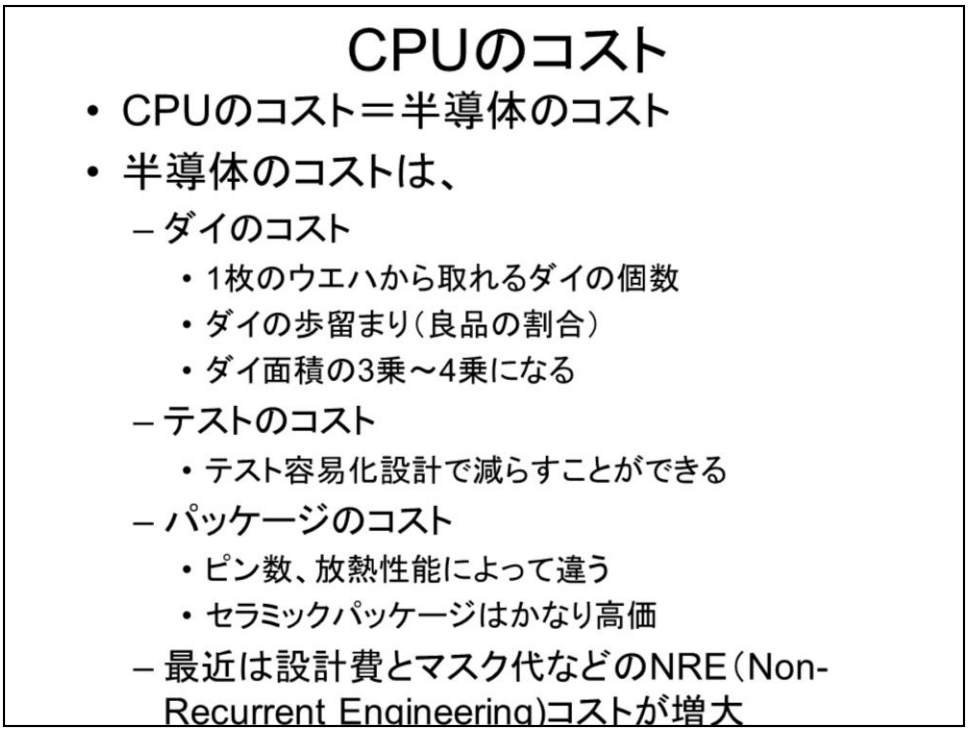


では、性能の単位として一般的に使われているものを紹介しておきましょう。MIPS（Million Instructions Per Second)は1秒間に何百万個命令を実行できるか、という尺度です。最近の高性能プロセッサではGIPS(Giga Instruction Per Second:1秒間に何十億個命令が実行できるか）が使われます。この尺度は実行する命令がどのようなものであるかを度外視しているため、違った命令セットの間で比較するのは意味がないです。しかし、同一の命令セットアーキテクチャ、例えばIntelのマシン間、ARMのマシン間で比較するならば、それなりに実感にあった値となります。

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。

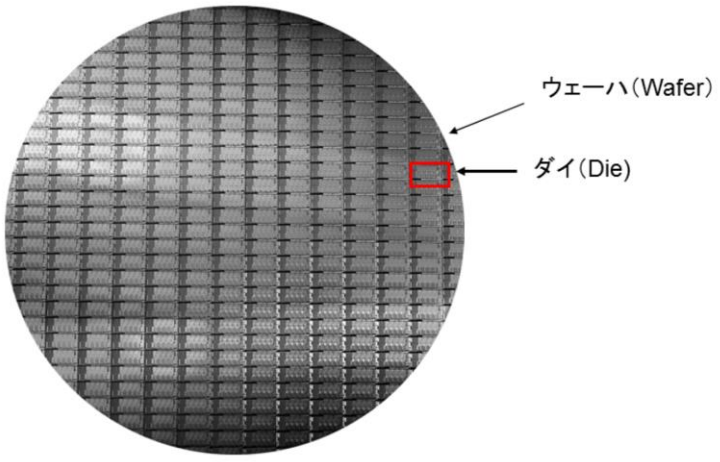
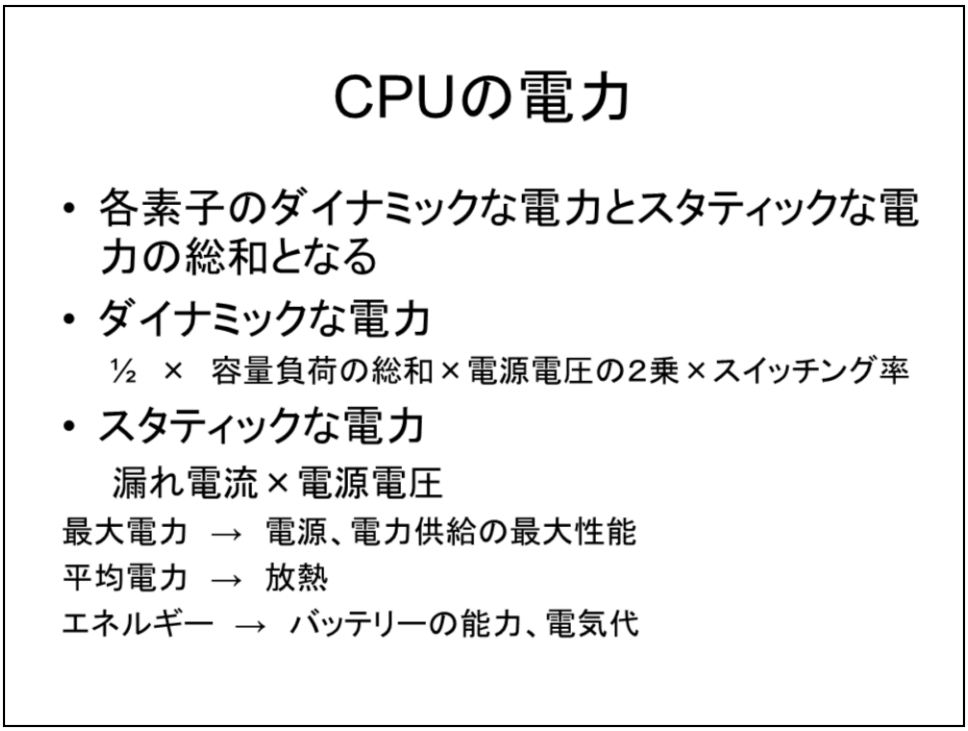


Figure 1.15 This 300 mm wafer contains 280 full Sandy Bridge dies, each 20.7 by 10.5 mm in a 32 nm process. (Sandy Bridge is Intel's successor to Nehalem used in the Core i7.) At 216 mm2, the formula for dies per wafer estimates 282. (Courtesy Intel.)

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。

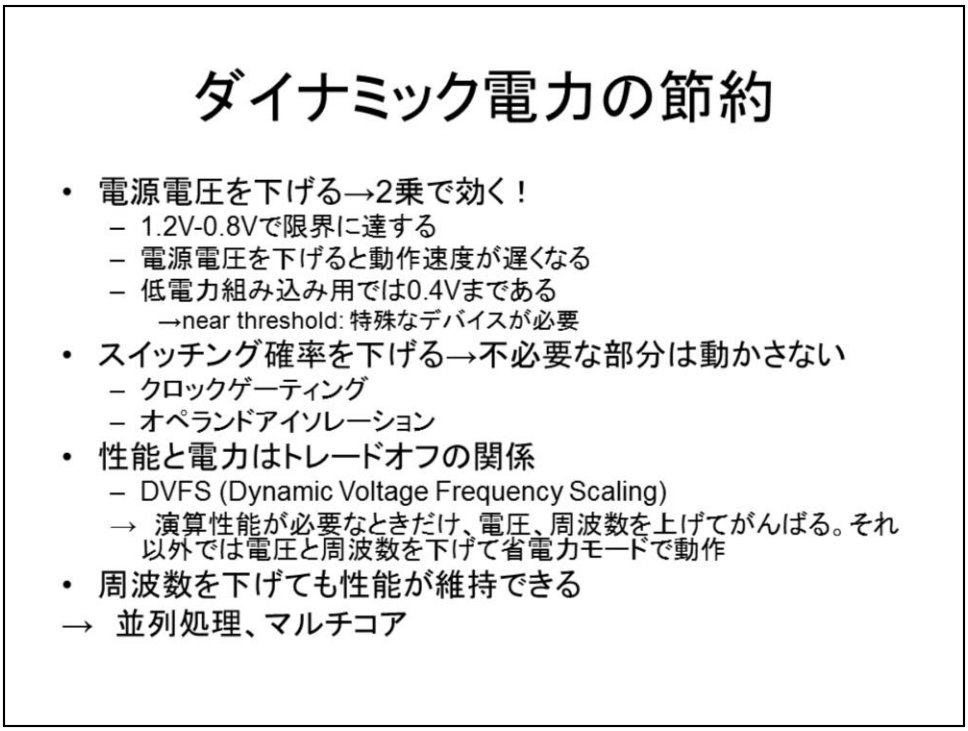


最後にCPUの消費電力に関してまとめておきましょう。CPUは半導体の各ゲートのダイナミック（動的）な電力とスタティック（静的、漏れ）な電力の総和になります。ダイナミックな電力は、CMOSを構成するトランジスタがON/OFFする時に流れる貫通電流（Internal Power)と、負荷となる容量を充放電するスイッチング電力（Switching Power)に分けられますが、貫通電流はトランジスタ内部に等価的な容量を想定して考え、これを負荷容量に含めて考えます。そうすると、1／2×容量負荷の総和×電源電圧の2乗×スイッチング率で求められます。容量負荷の総和は、あまり多数の出力を繋ぎ過ぎない（ファン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。

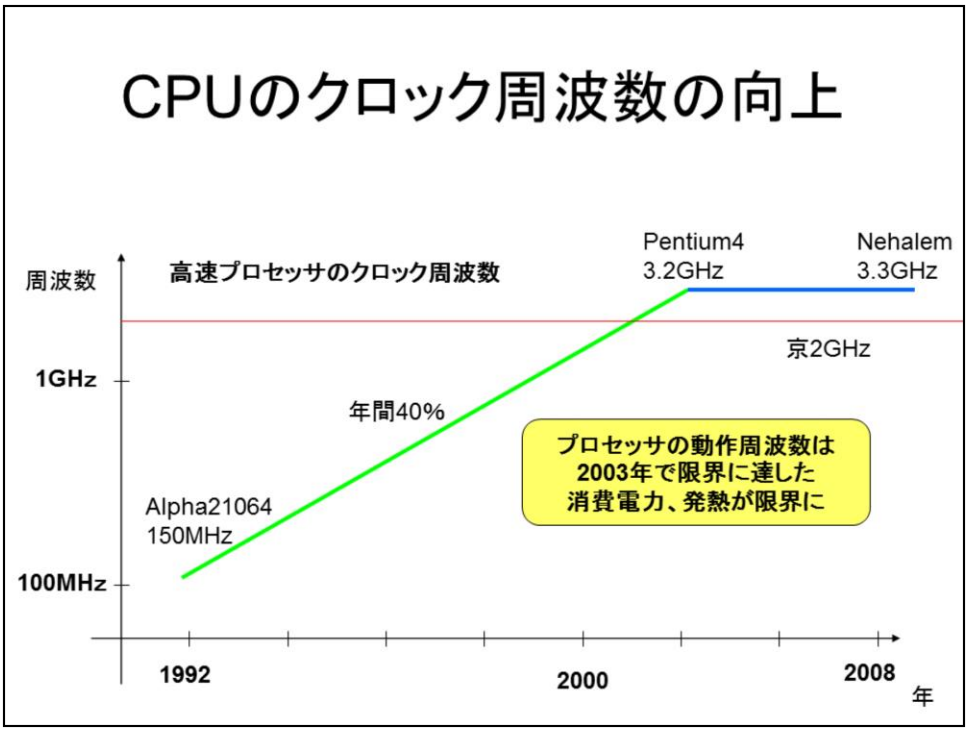


ダイナミックな電力を減らすにはどうすれば良いでしょう？一番簡単な方法は、電源を下げることです。このため、ディジタル回路の電源電圧はこの30年間で5Vから1Vくらいまで下がりました。しかし、0.8Vくらいで限界に達してしまいました。同じトランジスタで、電源電圧を下げると動作速度が遅くなります。現在、低電力用のICでは0.4V程度で動きます。これらはニアスレッショルド（near threshold)といってスレッショルドに近い電源電圧で動かします。この場合、ダイナミックな電力を極端に減らすことができるのですが、動作速度は落ちます。

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。

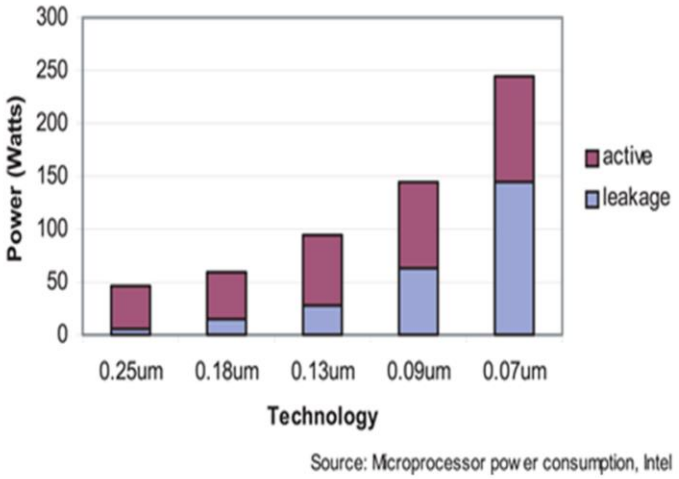
電圧、周波数を上げれば動作速度は上がりますが、電力が増えます。つまり性能と電力はトレードオフの関係にあります。では性能が必要なときだけ、電圧を上げ、周波数を上げて性能を上げてやり、さほど必要としないときはこれらを落として電力を節約することができます。このような手法はDVFS（Dynamic Voltage Frequency Scalling)と呼び、皆さんの使うPCに普通に使われています。

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。

スタティック電力（リーク電力）の節約



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

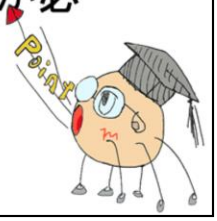
- リーク電流は、動作しなくても流れる
  →バッテリー駆動では致命的
- リーク電流は、スレッショルドレベルが低いと大きくなる
  - 超高速CPU
  - 超低電圧システム
- パワーゲーティング
  - スレッショルドレベルの高いトランジスタをスイッチに使って電源を切断
- Dual Vth
  - スレッショルドレベルの違うトランジスタを併用する
  - クリティカルパスには高速トランジスタを利用
- ボディバイアス制御
  - スレッショルドレベルを変化させる

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。

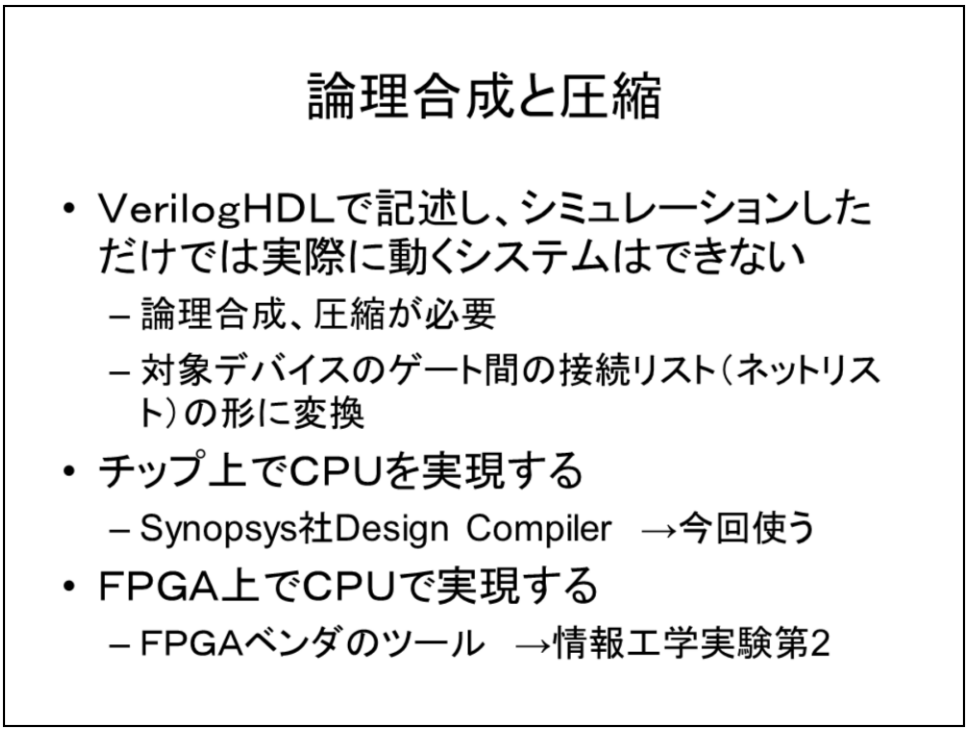
これを押さえるには、スレッショルドレベルが高いトランジスタをスイッチに使って、使っていない場合に電源を切ってしまうパワーゲーティングという手法が一般的に使われます。また、スレッショルドレベルの違うトランジスタを用意しておき、クリティカルパスには高速で漏れ電流の大きなトランジスタを使い、そうでないところには遅いけれど漏れ電流の小さなトランジスタを使うDual Vthという手法も一般的です。最近のトランジスタの中には、ボディ(サブストレート)に電圧を掛けることで、スレッショルドを変えることができるものがあり、これを使って漏れ電流と性能のバランスを使い方に応じて制御する方法も使われます。

ここまでのまとめ

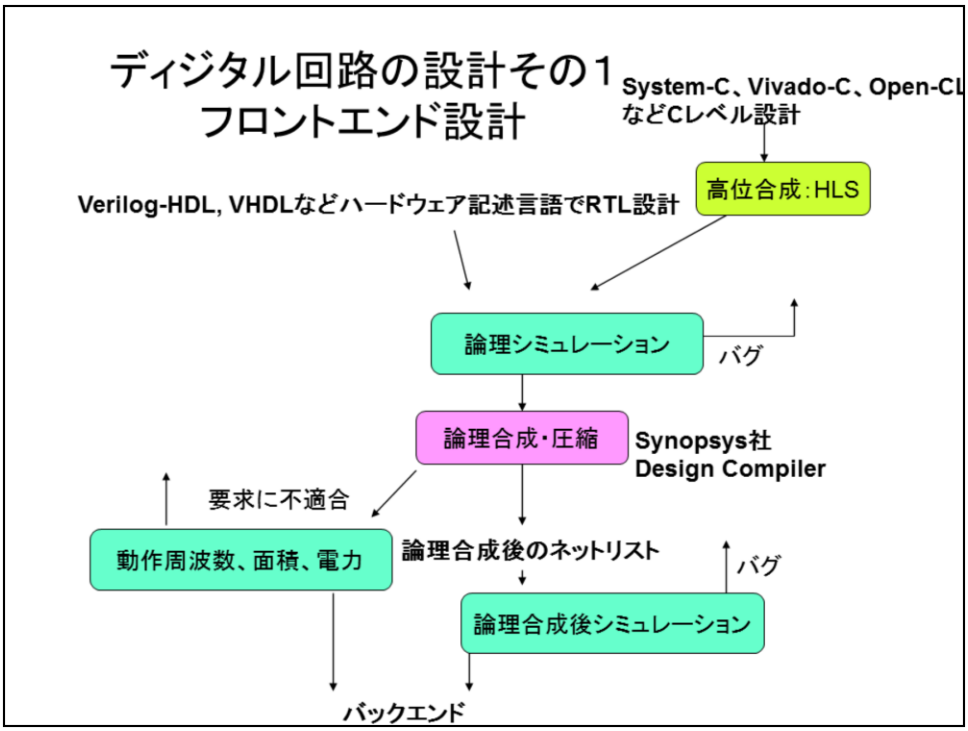
- POCOではCPIは1
- 性能は動作周波数で決まる
- コストは面積＝ゲート数で決まる
- 電力は動作周波数、ゲート数、スイッチング率で決まる
- 上記を評価するには論理合成・圧縮が必要！



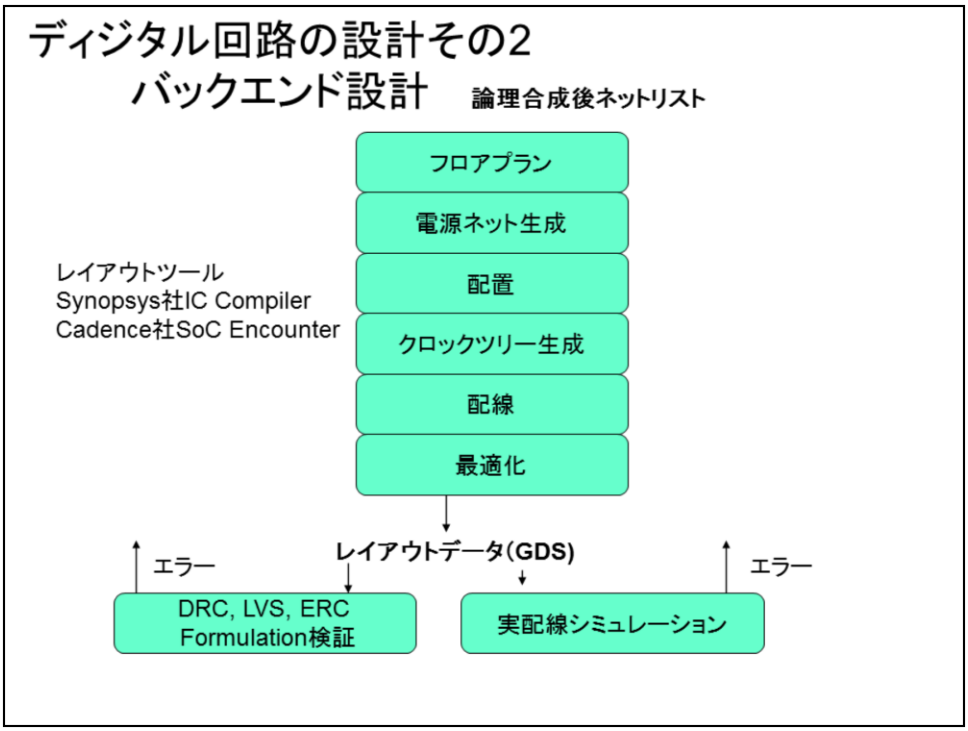
さて、ここまでのまとめを示します。ではPOCOの性能、コスト、電力を評価するにはどうすれば良いか？というと論理合成、圧縮を行って、ゲートレベルに落としてやる必要があります。これからこの方法を紹介します。



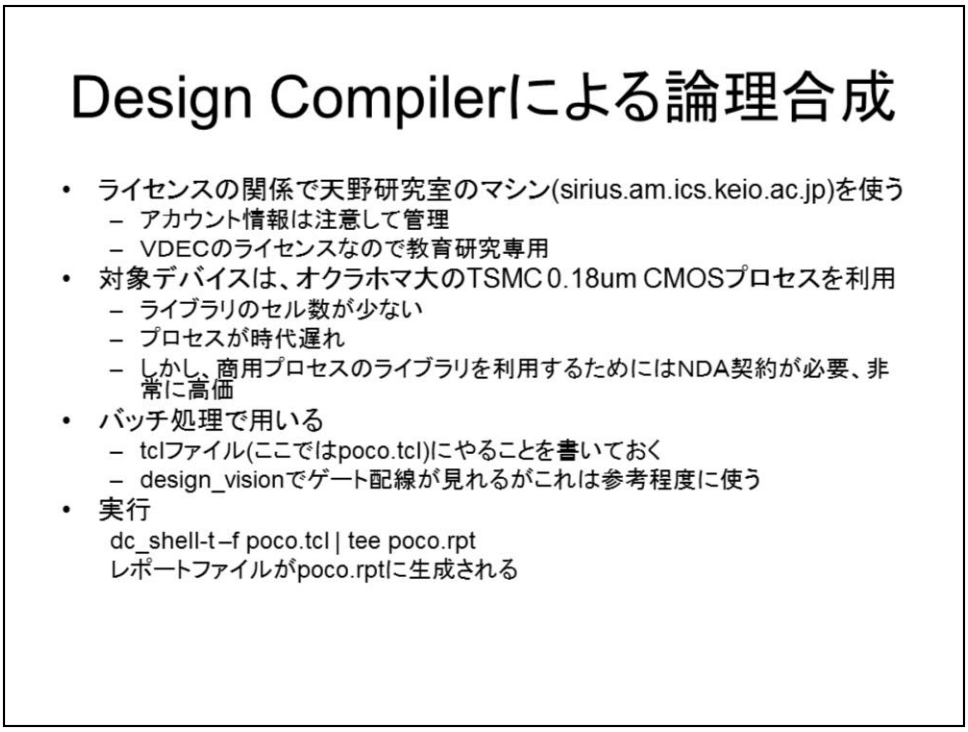
今までVerilogHDLでPOCOを記述してシミュレーションして動作を確認してきました。しかし、これだけでは実際に動くシステムはできません。CPUの入った集積回路を作る場合も、書き換え可能なLSIであるFPGA上で実現する場合も、論理合成、圧縮を行って、対象とするデバイスのゲート間の接続リスト（ネットリスト）に変換する必要があります。今回は、Synopsys社のDesign Compilerを用いて、実際の集積回路上にチップを実装する方法を紹介します。実際には書き換え可能なLSIであるFPGA（Field Programmable Gate Array)上で実現する場合が多いですが、これは来年の情報工学実験第2で実際に行います。



ディジタル集積回路の設計は、まずフロントエンド設計を行います。まずハードウェア記述言語でRTL設計を行います。今までやってきたように論理シミュレーションを行って、動作を確認しながら設計を進めます。設計が一段落したら、論理合成・圧縮を行います。CADがRTL記述を解析し、論理ゲート間の接続図の形に変換します。これと同時に動作周波数、面積、電力を見積もります。これが要求を満足しなければ、最初に戻って設計をやり直します。要求を満足すれば、ネットリストをシミュレーションして動作を確認します。基本的にCADで生成されたネットリストのシミュレーションは元のRTLのシミュレーションと一致するはずですが、記述のやり方が悪いと意図通りの動作を行わない場合があります。もしも問題が見つかったら設計をやり直します。最近はSystem-C、Vivado-C、Open-CLなどのCレベル設計が特にFPGAを対象として広まっています。このC言語の記述は高位合成（High Level Synthesis:HLS)によりRTL記述に変換されます。これ以降は同じ流れになります。



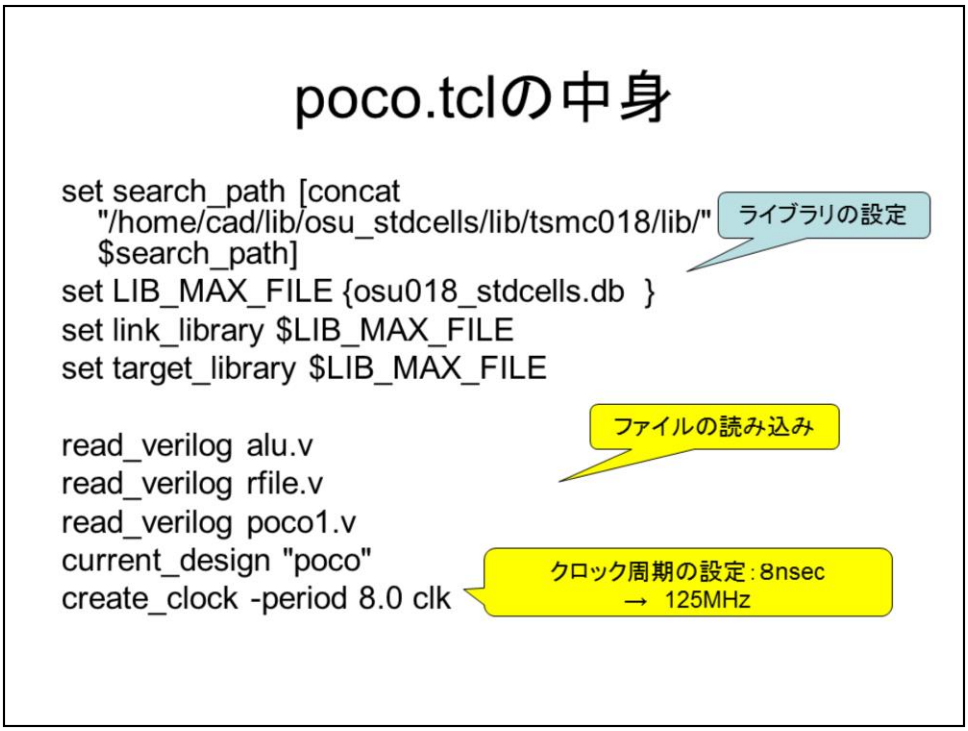
フロントエンド設計により、ネットリストが固まったら、次の段階はバックエンド設計です。これは、4年生のVLSI設計論で実際に演習をしながら学んで行きます。



今回利用するのはSynopsys社のDesign Compilerという論理合成用のツールです。このツールはチップ設計用に世界中で使われており、実は非常に高価です。しかし、大学の教育研究用に限って、大規模集積設計教育研究センター（VDEC)により安価で提供されています。したがって、管理の都合上、天野研のマシン上で動かすことになります。このツールは教育研究用の利用以外は禁止されているので、これを使って何かを設計して売り出したりしてはいけません。（これをやりたければ天野研に来れば、正規の形で出来るようにしてあげるので、どうぞおいでください）。実際のチップを設計するためには、チップ上で動くゲート、フリップフロップなどの論理素子の一式が必要です。これをセルライブラリと呼びます。セルライブラリは、遅延時間、面積、消費電力が定義されていて、これを基にCADは論理合成・圧縮を行います。

今回は対象デバイスとしてはオクラホマ大の教育用のセルライブラリを使います。これは、TSMC（台湾の世界的半導体の製造メーカー（ファブ））の0.18μmのセルライブラリをモデルとしており、プロセスとしては古いですがリアルです。実際の商用プロセスのライブラリは非常に高価な上、利用にはNDA（秘密保持契約）を結ぶ必要があり、とても授業では使えません。

Design Compilerは、tclファイル（ここではpoco.tcl)にコマンドを書いておき、これをdc_shellと呼ばれるシェルに入れてやり、一度に実行させます。これをバッチ処理と呼びます。合成後のネットリストを見るツールdesign_visionもありますので、今回はこれも使います。



では、tclファイルの中身を見ながら、そのコマンドを説明します。最初にライブラリの設定を行います。まずライブラリのパスを設定します。osu018_stdcells.dbがライブラリ名です。ここまでは余り意味を考えずにこのまま使ってください。

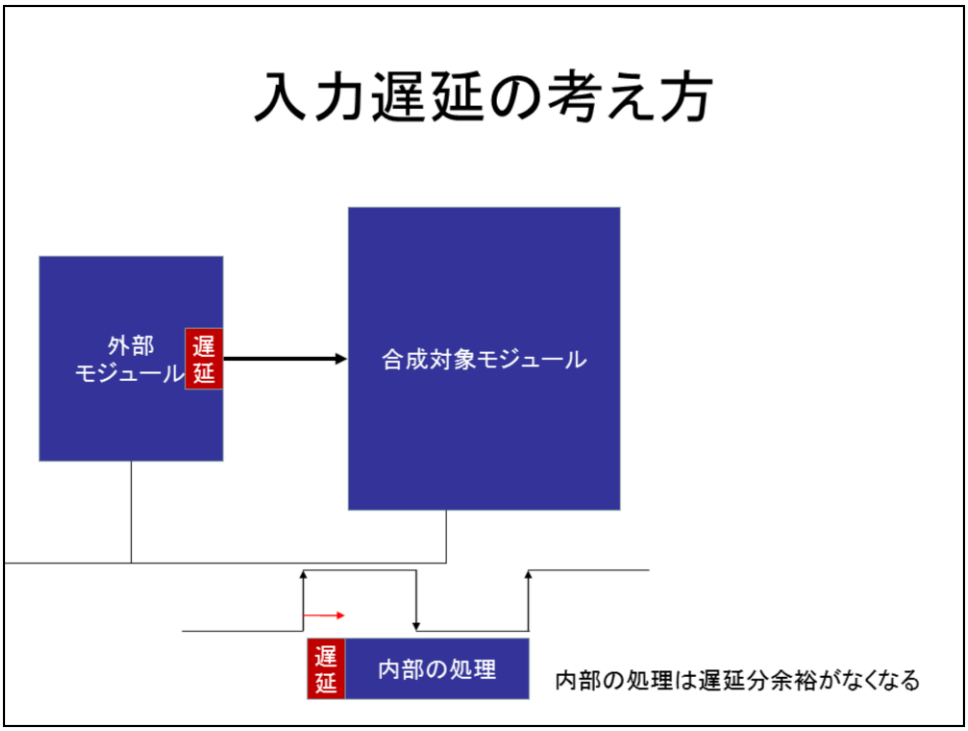
次にread_verilogで、Verilogファイルを読み込みます。忘れずに全てのファイルを読み込んでください。ここで、VerilogHDLの文法エラーがあると読み込みに失敗します。これは後でエラーメッセージを見ると分かります。シミュレーションでうまく行っても合成時にはエラーが出る場合もあるので注意しましょう。current_designで階層のトップを設定します。ここではpocoです。

次にcreate_clock –period 周期 クロック信号名で、クロック周期を設定します。ここでは8nsすなわち125MHzにします。これはあくまで目標周期で、満足できない場合もあります。

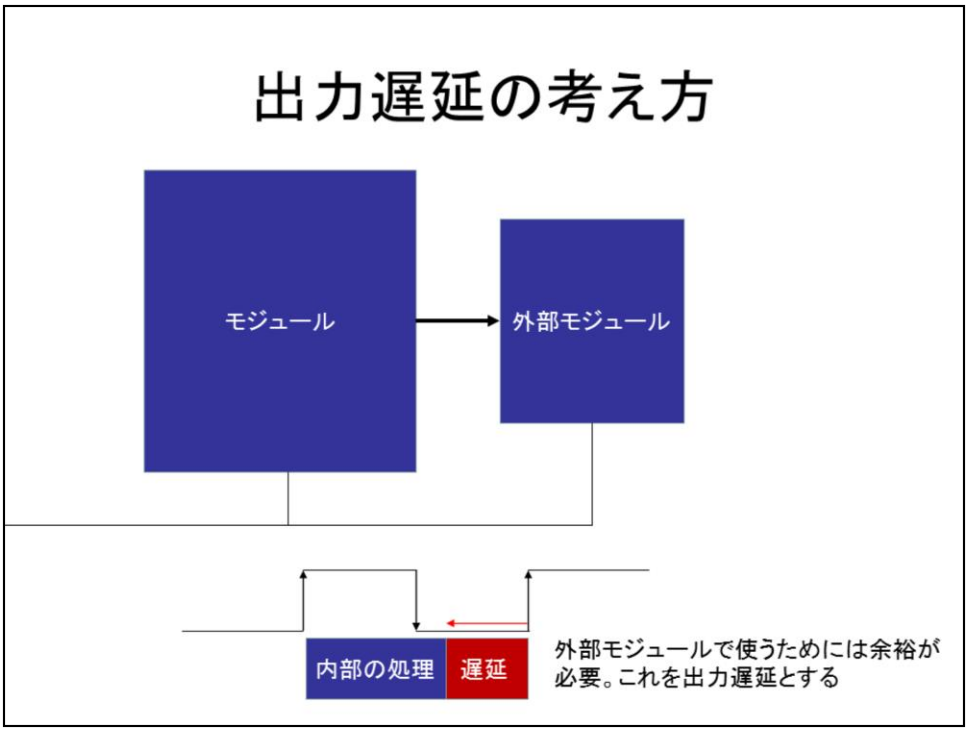
入出力遅延の設定

```
set_input_delay 2.5 -clock clk [find port "idatain*"]
set_input_delay 7.0 -clock clk [find port "ddatain*"]
set_output_delay 7.5 -clock clk [find port "iaddr*"]
set_output_delay 3.0 -clock clk [find port
   "ddataout*"]
set_output_delay 3.0 -clock clk [find port "daddr*"]
set_output_delay 3.0 -clock clk [find port "we"]
```

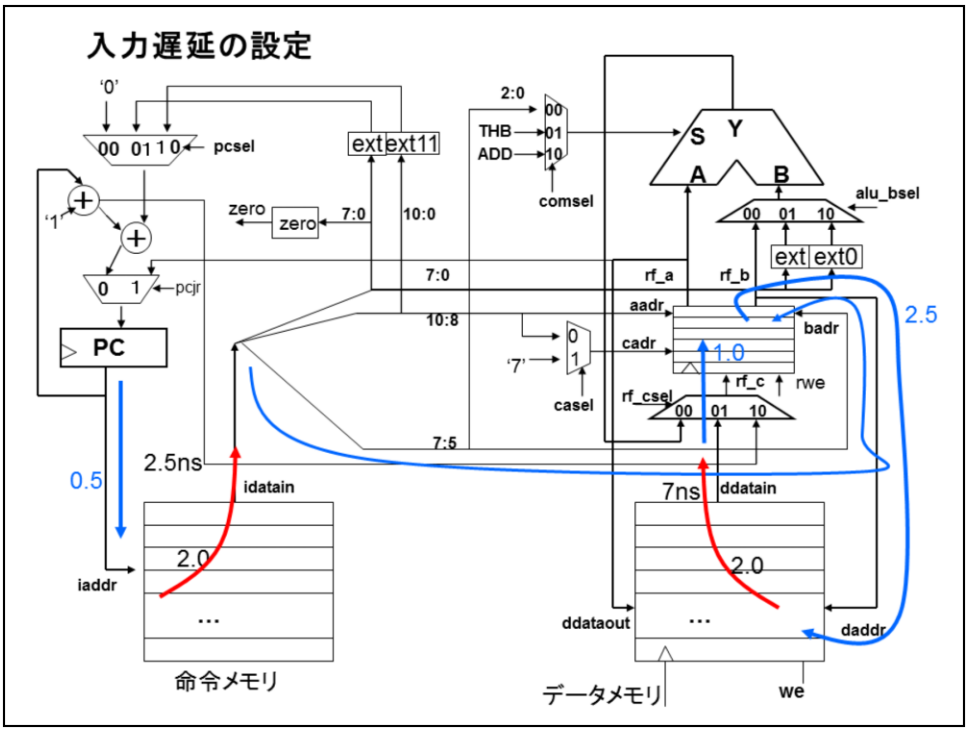
次に入出力の遅延を設定します。1サイクルCPUは、遅延パスが途中で外部メモリを通るので、この設定が面倒です。set_inpit_delayは入力信号の遅延、set_output_delayは出力信号の遅延設定です。それぞれの信号について設定します。



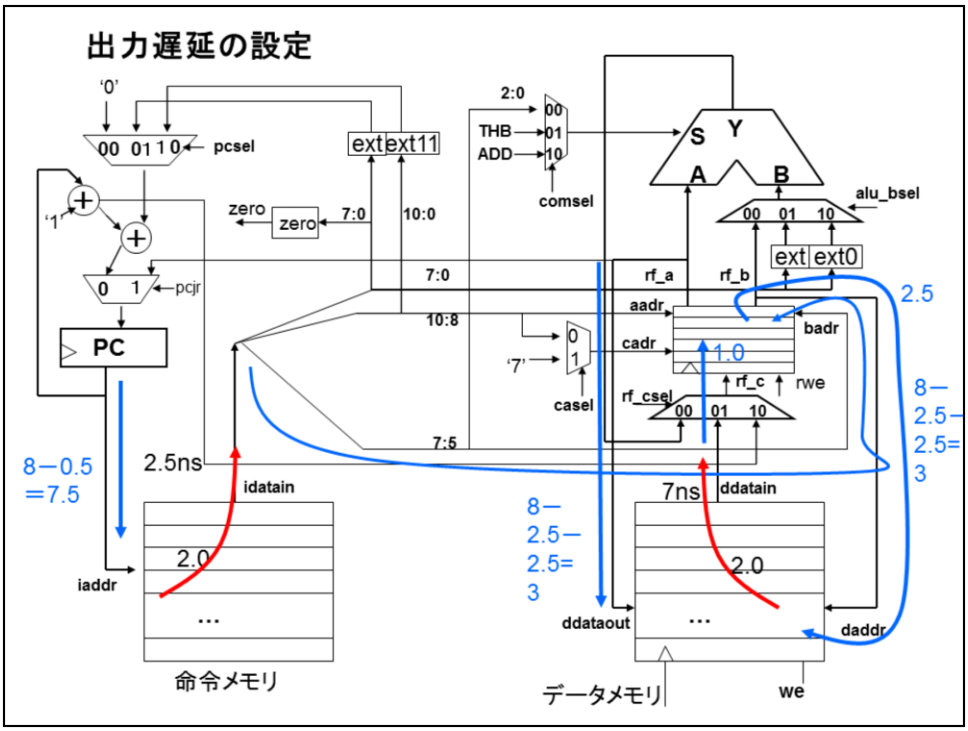
ではまず入力遅延を設定しましょう。外部モジュールと合成対象モジュールは同じクロックで動作していると考えます。入力は外部モジュールから与えられますが、ここには一定の遅延があるはずです。合成対象モジュールは、これに内部の処理の遅延を加えた全体の遅延が、次のクロックの立ち上がりに間に合うように合成しなければなりません。つまり入力遅延が大きいと、内部の動作は厳しくなり、場合によっては設定周期を満足できなくなります。遅延は立ち上がりのクロックを基準として図の赤矢印の方向に設定します。



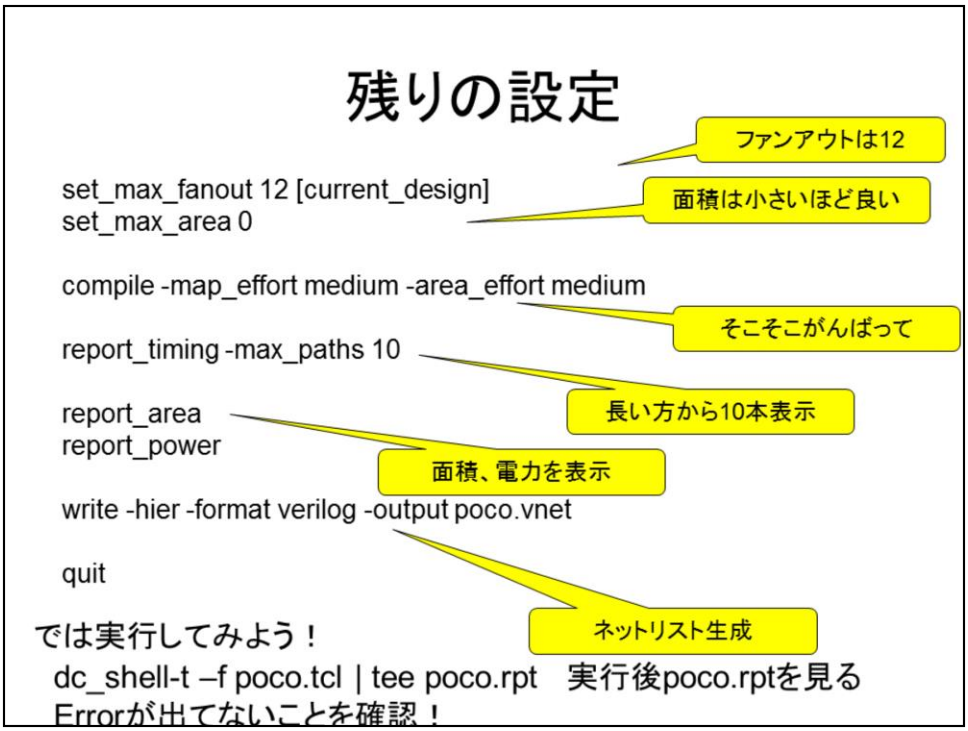
出力遅延は、入力遅延と逆で、モジュールからの出力が外部モジュールで確実に格納されるために余裕を持って出力しなければならず、このための遅延です。この遅延は次のクロックの立ち上がりを基準として、図の赤矢印の方向に考えて設定します。



ではPOCOの入力遅延を考えましょう。先の入力遅延の簡単なモデルとの違いは、入力データidatainはPCからの値がiaddrに出力され、これが命令メモリの遅延を経て戻ってくる点です。ddatainはさらに、idatain上の命令が内部のレジスタファイルの遅延を経てdaddrに表れ、これによりデータメモリを読み出した結果です。すなわち外部のメモリとの間で情報が2回出たり入ったりします。このため、入力遅延は、それぞれの部分の遅延をどうしたいのかを考えて決める必要があります。メモリによる遅延は2.0nsecとします。これはもう決まった値で動かすことができません。図では赤い矢印で示します。一方、POCOの内部の遅延は、論理合成の結果によって変わってきます。逆にここをどの程度の遅延にしたいかを設定してやることになります。ここではPCから命令メモリのアドレスまでは0.5nsec、idatainが入ってからレジスタファイルを読み出してaddrに出力する遅延を2.5nsec、ddatainから入ってきたデータがマルチプレクサを抜けてレジスタファイルに書き込むまでの値を1.0nsecとしています。この数値が論理合成の目標値となります。これを実現するためにidatainの入力遅延は0.5+2.0=2.5nsec、ddatainの遅延は2.5+2.0+2.5=7.0nsecに設定しています。

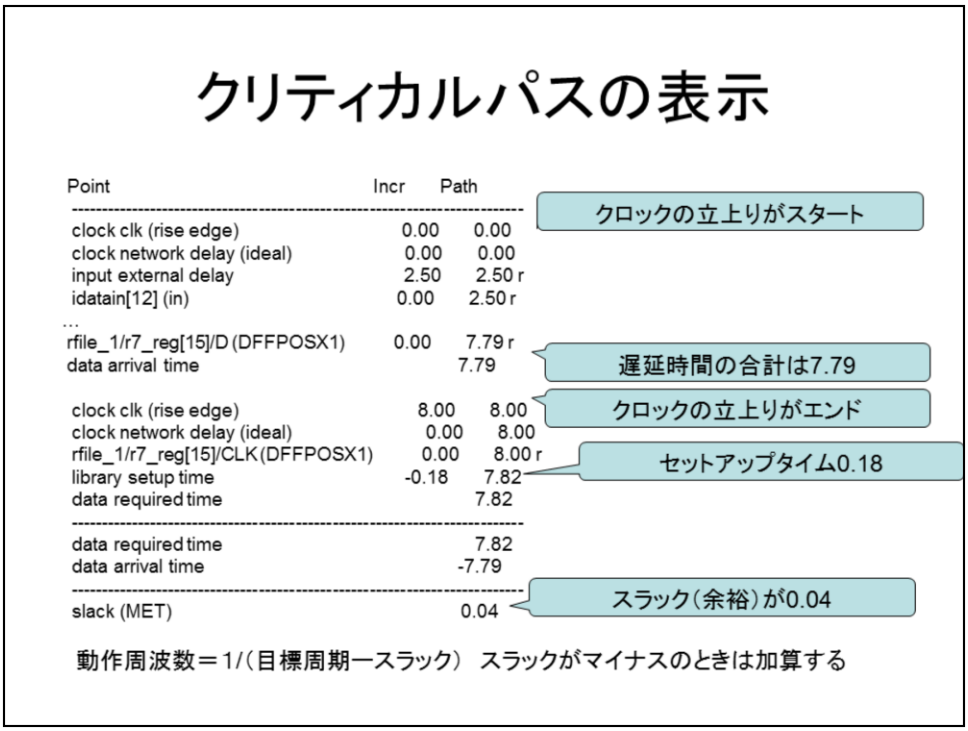


出力遅延は次のクロックの立ち上がりが基準です。iaddrはサイクルの始めに出さないといけないです。この場合命令メモリのアドレスまでを0.5nsecと考えたので、8-0.5＝7.5nsecと設定します。逆から考えると、この後の遅延を全部足して1.0＋2.0＋2.5＋2.0＝7.5として計算してもいいです。(ちょっと大変ですが、、)ddataout, daddr、weは、8-2.5-2.5=3nsecに設定すれば、メモリの遅延2.0nsecを差し引いても、レジスタファイルに書き込むための1nsecを確保できます。逆から考えると2+1＝3nsecと計算してもいいです。これで矛盾のない入力遅延と出力遅延の設定ができました。この遅延の値が矛盾すると、論理合成に無駄が生じる可能性があります。



最大ファンアウトは12に設定し、ひとつの出力に繋がる入力数を12に制限します。次に目標面積に0を設定します。これはムチャか気がしますが、これはあくまで目標であり、面積は小さいほど良いので、通常このように設定します。次はコンパイル（ここでは論理合成、圧縮の最適化）レベルを設定するコマンドで、ここではmediumで中程度にがんばって欲しい設定にしています。これをhighにすると、もっと最適化をがんばってくれますが、コンパイル時間が非常に掛かります。さて、次からはレポート文で出力を制御します。report_timingは最も長いパス、すなわちクリティカルパスを表示します。ここでは長い順に10本表示します。もし最長のだけ表示すれば十分ならばここを1にしてください。次に面積、電力を出力して、最後にはネットリストを生成しておしまいです。ネットリストの名前はpoco.vnetとしています。これも実はVerilogの記述ですが、ゲート間の接続の形に変換されています。

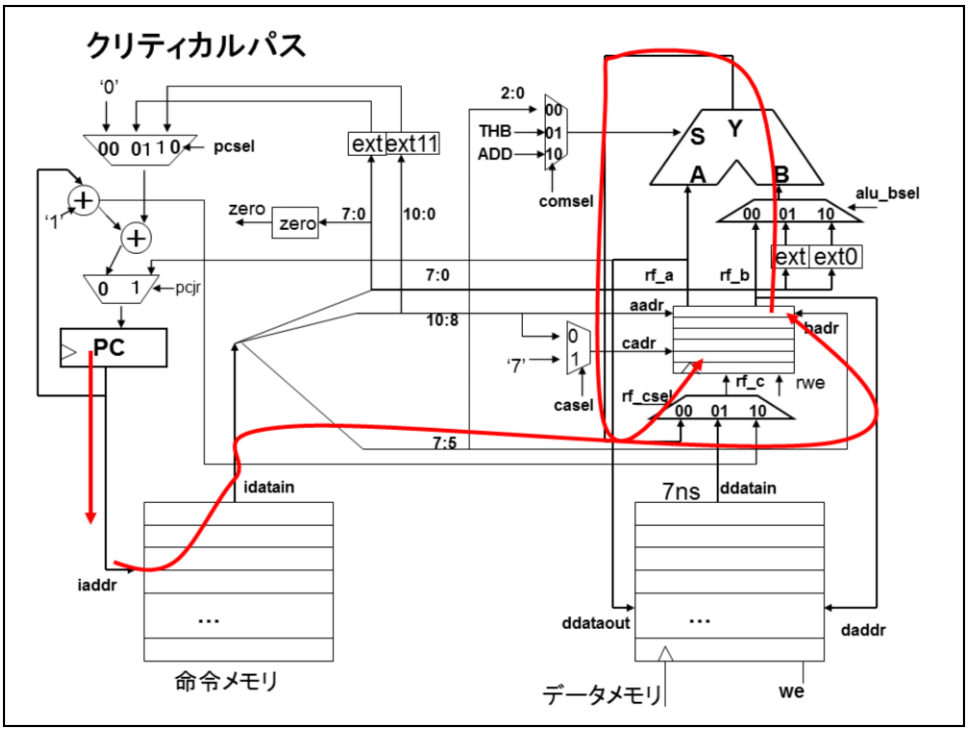
では、dc_shell-t –f poco.tcl | tee poco.tclで実行します。Synopsysの実行には初期設定が必要です。これはこのスライドの最後の方に使い方をまとめているのでここを見てください。



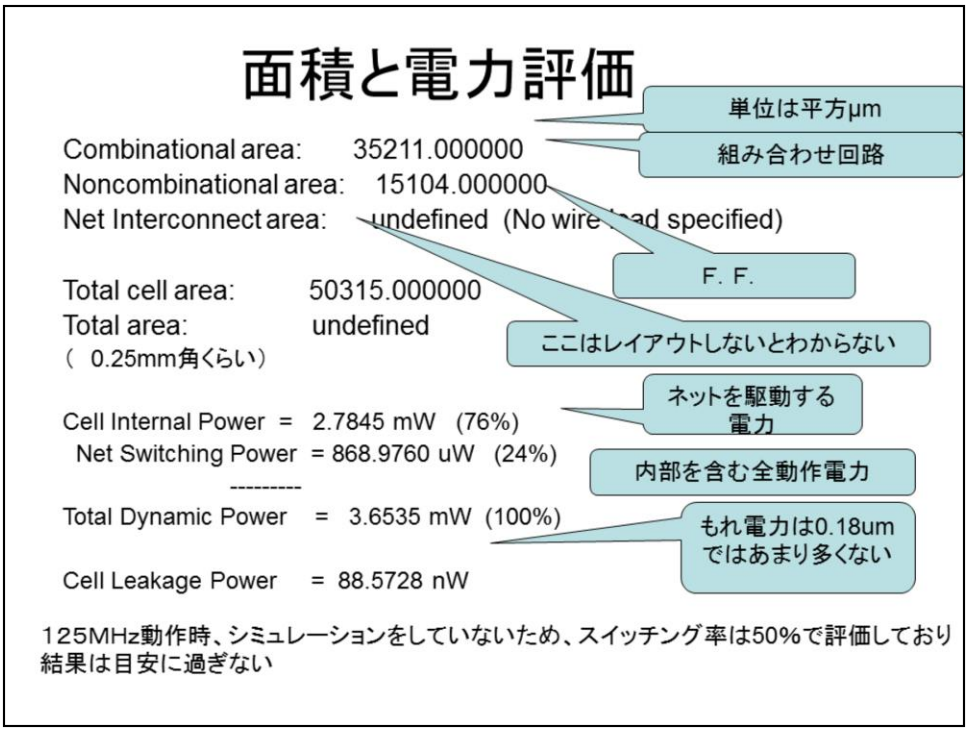
poco.rpt中でCritical Pathの表示を探してください。このスライドのようなレポートが10個出ていると思います。パスは長い順に表示されるので、最初の1本が一番大事です。この表は、クリティカルパスがどのように辿っているか、途中でどのように遅延が増えているかを示しています。スタートポイントはclkの立ち上がりで、idatainに対する入力遅延が2.5nsecがまず加わり、さらにPOCOの内部で順に遅延が積み重なっている様子がわかります。この場合、遅延時間の合計は7.79になっています。これにフリップフロップに値を格納するために必要なセットアップタイム0.18nsecが加わります。これは目標周期を8nsecに設定しているため、0.04nsecの余裕（スラック）が生じていることがわかります。（電卓で計算すると0.03になるはずだが、下の桁まで計算に入れているのかと思う）

この回路の動作周波数は、目標周期ースラックの逆数になります。今回は、125.6MHzです。スラックはマイナスになる場合もあり、この場合は加算されることになり、動作周波数は目標周波数よりも落ちます。

目標周期を短くすると、クリティカルパスを短くしようと論理合成、圧縮をがんばってくれるので、性能が上がる可能性がありますが、その分面積が増えてしまいます。スラックが0前後になるように目標周期を設定するのが多くの場合は良いといわれています。



この場合、クリティカルパスはidatainから入ってレジスタファイルを抜けてALUを通っている様子がわかります。図とクリティカルパスの表示を照合してみてください。



次に面積が表示されます。ここで、単位は平方μmです。つまり、1000000平方μmが1mm角になります。Combinational areaは組み合わせ回路、Noncombinational areaがフリップフロップを表します。Net Interconnect areaは配線のための面積で、これはレイアウトしないと分かりません。セル全体の面積は50315になります。大体セルの充填率つまり詰め込む割合は70%くらいなので、レイアウト全体の面積は71882となり、この平方根は268なので、大体0.25mm角くらいであることが分かります。

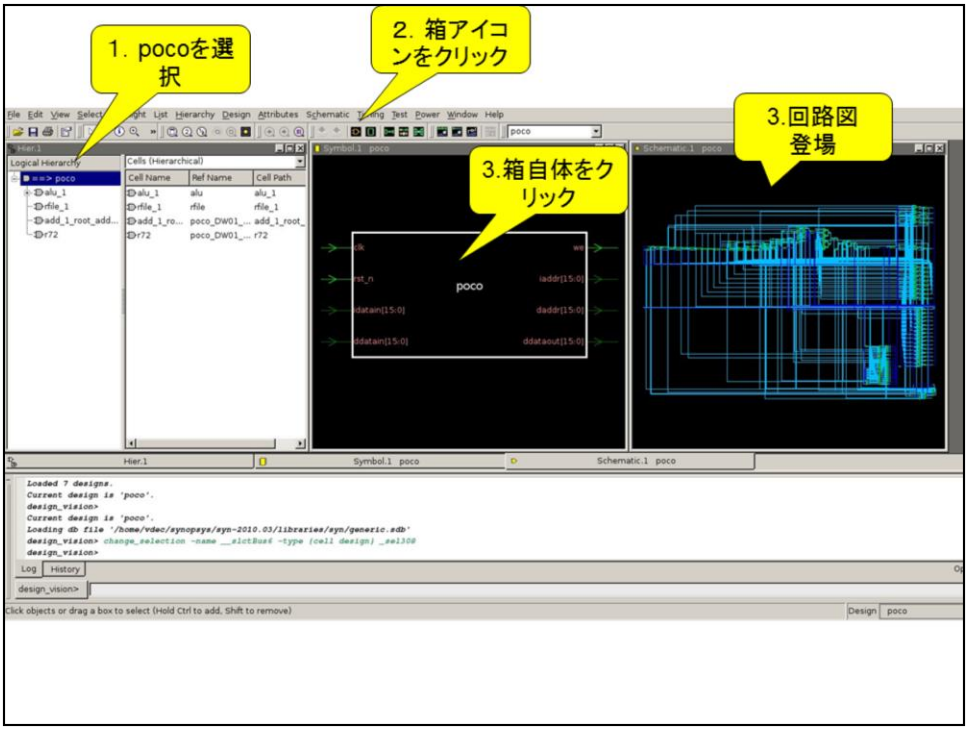
最後は、動作周波数が出てきます。Cell Internal Powerはセルの貫通電力、Net Switching Powerは負荷を駆動するためのスイッチ電力です。この和がダイナミック電力になり、約3.6mWです。これは125MHzでスイッチング電力を50%としたときの見積もりです。正確には合成後のシミュレーションでスイッチ率を出す必要があり、これは目安と思ってください。リーク電力はnWで多くないです。これは0.18μmプロセスで古いためです。

回路Viewer design_vision

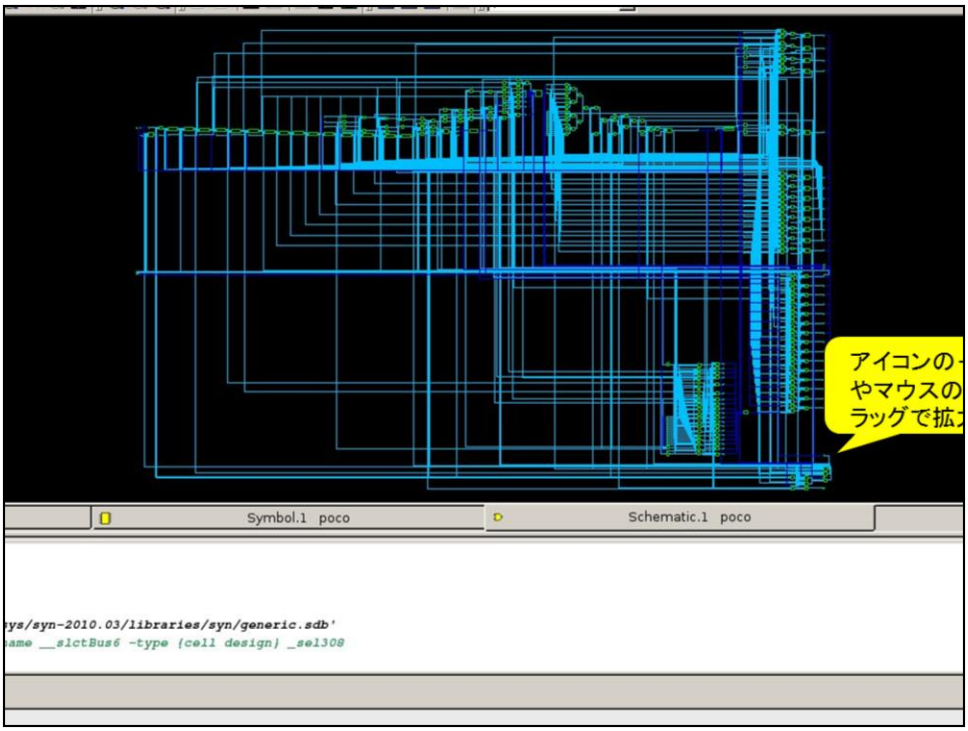
- design_visionはdc_shellのGUIで合成した回路を見たり解析したりするツール
- 実際の回路は巨大すぎるので、設計の現場ではあまり使わない

design_visionで立ち上がるので、
File→Read→poco.ddcを選択して、Openをクリックしてみよう

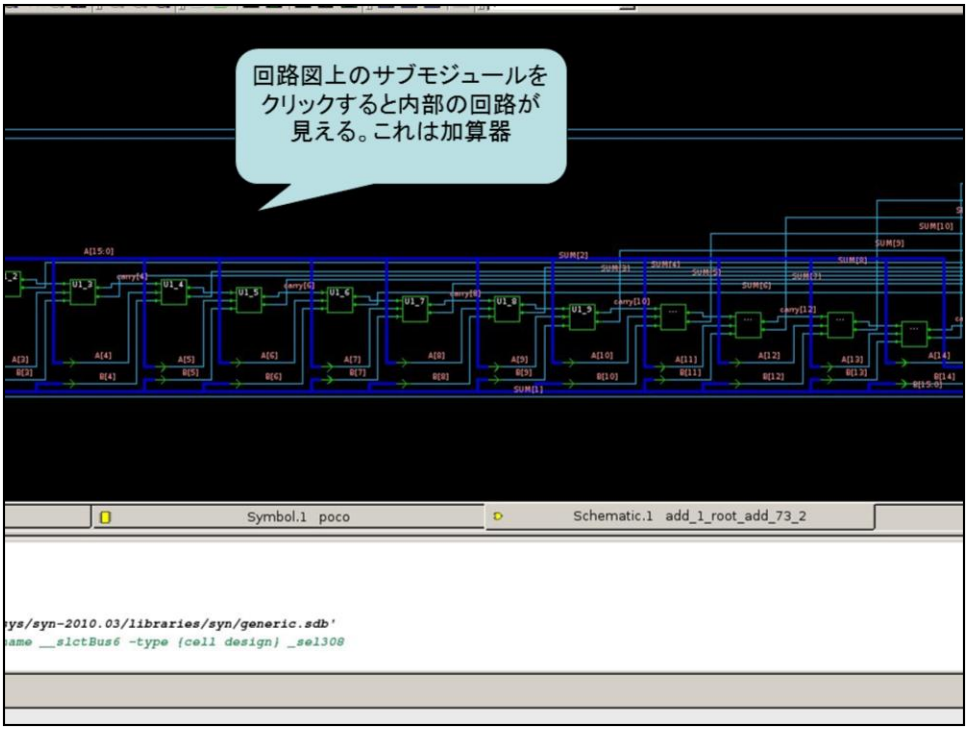
では次にdesign_visionについて解説しましょう。design_visionはdc_shellのGUIで、合成した回路を見たり解析したりするツールです。しかし、実際の設計では巨大すぎ、このツールで見てもわけがわからないので、あまり使われないです。しかしここでは雰囲気を掴むため、使ってみましょう。design_visionと打ち込むと立ち上がるので、File→Read-> poco.ddcを選択してOpenをクリックしてみてください。



スライドの図に従い、pocoを選択、箱アイコンをクリックします。そうすると箱がサブウィンドウに出てくるのでここをクリックすると隣のウインドウに回路図が登場します。



アイコンの＋やマウスのドラッグで拡大します。ゲートのつなぎ方を見てみましょう。



回路中にはゲートだけではなく箱があり、これはサブモジュールになっています。これをクリックすると内部の回路が見えます。これは加算器です。POCOの回路を見てみましょう。

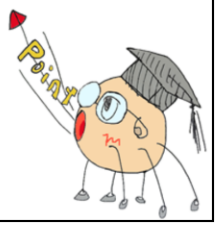
本日のまとめ

- CPUの性能は、実行命令数、CPI、実行周期（周波数）で決まる
  - POCOではCPIは1
- コストは面積で決まる。3乗から4乗に比例する
- 消費電力はダイナミック電力とリーク電力がある
  - ダイナミック電力は電源電圧の2乗、スイッチング率に比例する
  - スタティック電力は電源電圧に比例
- 実行周期、面積、電力は論理合成をして評価
- Design Compilerの実行は
  - ふんが研のマシンにログイン、初期設定の後

dc_shell-t –f poco.tcl | tee poco.rpt

コマンドはpoco.tclに入れておく

poco.rptに合成結果があるのでここを見る



インフォ丸が教えてくれる今日のまとめです。

合成用システムの使い方

計算機構成用
天野

以下は合成用システムの使い方です。やや面倒ですが、高価なCADを使うために仕方ないと思ってください。

ふんが研へのリモートログイン

sirius.am.ics.keio.ac.jpにログインする。

ssh –Y xxxx@sirius.am.ics.keio.ac.jp

xxxxは配布されたアカウント、パスワードを聞いてくるので配布された初期パスワードを打ち込む

- 配布されたアカウントとパスワードでログイン
  - この紙はなくさないで！！！

初めてログインした時だけやること

パスワードの変更

passwd

と打ち込んで紙にあるパスワードを新しいパスワードに変更、これは忘れないこと

環境設定

ln –s /home/vdec/script/.setup_vdec.sh

ln –s /home/vdec/script/.pre_setup_vdec.sh

ln –s /home/vdec/script/.post_setup_vdec.sh

ファイルの転送

ITCのマシン→sirius

ITCのウィンドウで

scp file名 usr名@sirius.am.ics.keio.ac.jp:~/

逆の場合は

scp file名　usr名@ loginXX.user.keio.ac.jp:~/

tar, emacsは普通に利用可能

dc_shellとdesign_visionの実行

- ログイン直後に以下を行う

  source ~/.setup_vdec.sh
- デザインコンパイラの起動

  dc_shell-t –f poco.tcl | tee poco.rpt

  poco.tclを見てErrorが出ていないことを確認！

  実はmakeで上のコマンドが動くことになっている
- 合成後の回路図を見るツールdesign_vision

  design_vision &

演習

1．目標周期を7.8nsにしてPOCOを論理合成して結果を評価せよ。

2．前回のJALRを実装したPOCOを論理合成し、動作周波数、面積、電力を評価せよ。周期は8nsでやってみて、slackが0になるように調整せよ。

3.dice.vを合成し、design_visionでその構造を観察せよ