マイクロプロセッサ特論13回
コントロールハザード

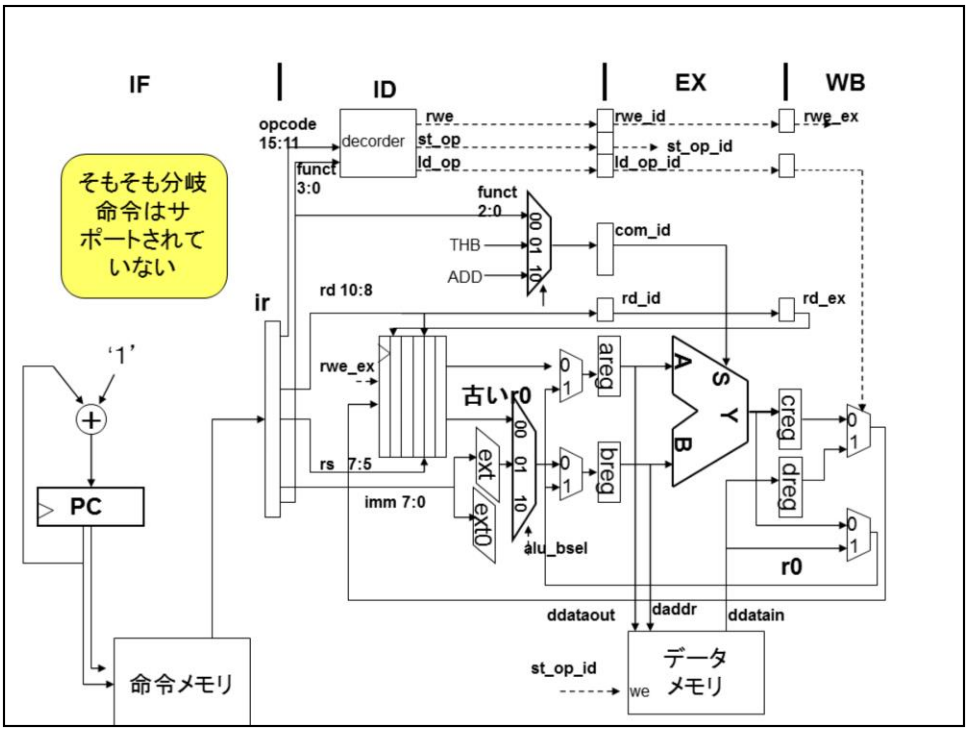
慶應大学
天野英晴

今日は前回の続きと、最後のまとめをやります。

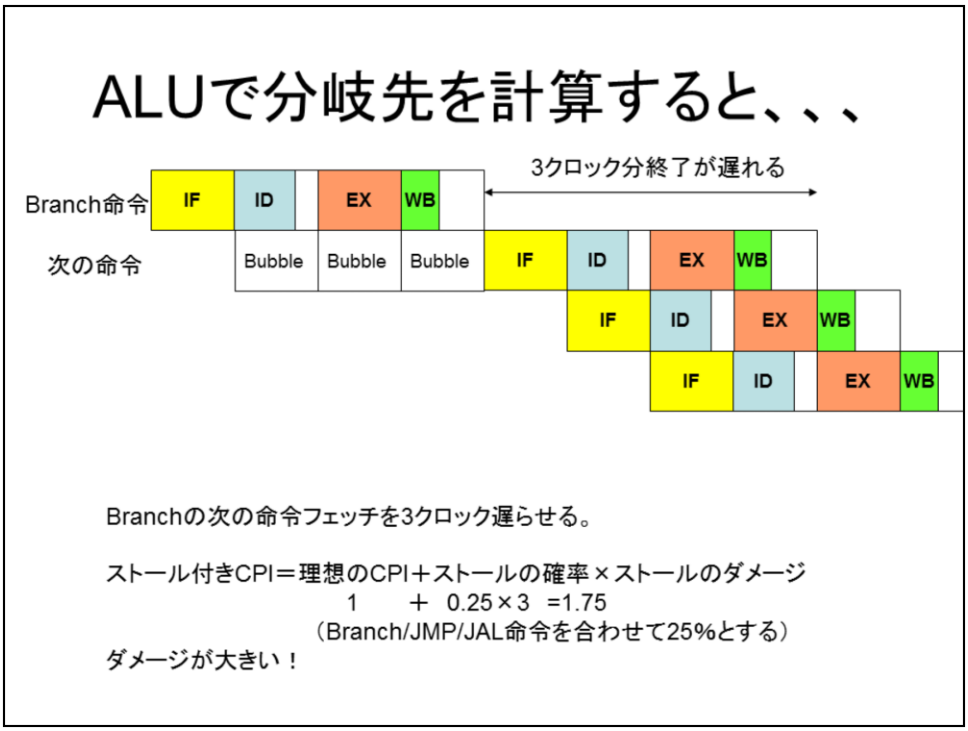
パイプラインハザードとは？

- パイプラインがうまく流れなくなる危険、障害のこと
  - 構造ハザード
    - 資源が競合して片方のステージしか使えない場合に生じる
  - データハザード
    - データの依存性により生じる
    - 先に進んだ命令の結果を後の命令が利用するため、その結果がレジスタに書かれるまで、読むことができない
  - コントロールハザード
    - 分岐命令が原因で、次に実行する命令の確定ができない
- パイプラインストール
  - ハザードが原因による性能の低下
  - パイプライン処理は理想的に動くとCPIが1
    - ストールによりCPIが大きくなってしまう

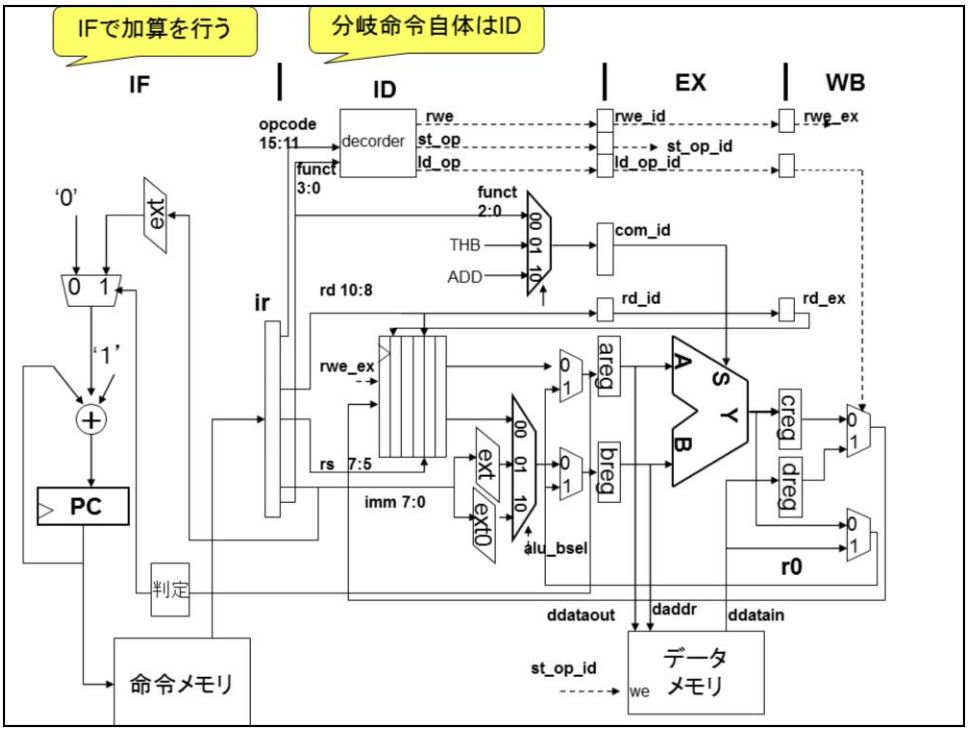
前回まで構造ハザード、データハザードを紹介しました。三つ目がコントロールハザードです。これは分岐命令が原因です。元々パイプラインは次に実行する命令が分かっているから流れるのであって、分岐命令があって次に実行する命令がどちらになるかが分からない場合にストールが起きるのは当然と言えます。



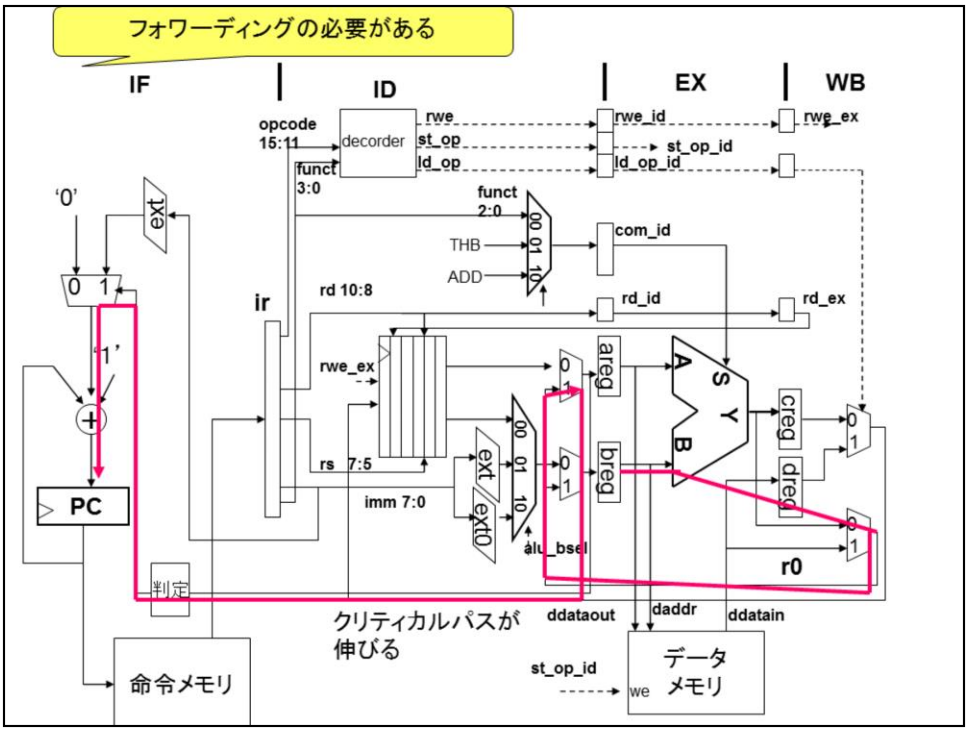
実を言うと、今まで分岐命令をパイプラインに組み込んでいませんでした。PCは毎クロック1づつ増やしていったため、パイプラインはスムーズに流れたわけです。しかし、実際にはそうは行きません。ではALUで飛び先を計算するとどうなるでしょう？この場合、飛び先が決まるのには3クロック掛かってしまいます。



次の命令を取ってこれるのは、飛び先のアドレスが決まった次のサイクルからになってしまいます。この場合は3クロック分バブルが入ってしまい、ストールのダメージは3クロックになります。分岐命令がフェッチされる確率はプログラムに大きく依存しますが、それなりの割合になることが多いです。Branch,JMP,JALの割合を合わせて25%と考えると、理想CPIの1が1．75になります。これは相当大きなダメージであると言えます。

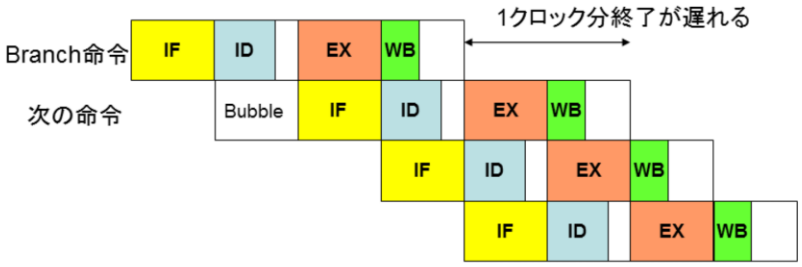


ダメージを減らすためには、分岐命令が成立するかどうか、成立する場合どこに飛ぶかを早めに判別する必要があります。命令がirに入った後、すなわちIDステージで分岐命令かどうかは分かります。このステージでレジスタファイルからレジスタが読み出され、分岐の条件がチェックされます。ir中にある飛び先をIFステージに送って加算を行い、分岐が成立する場合にはpcの内容を更新します。



分岐命令は、レジスタの内容で分岐するかどうかを判定します。このレジスタは直前の命令で更新される可能性があります。データハザードを避けるためには、直前の演算結果をフォワーディングする部分について判定が必要になります。この判定の結果で飛び先をpcに書き込むかどうかが決まるので、かなり長大なパスができてしまいます。これが、この方法の問題点です。

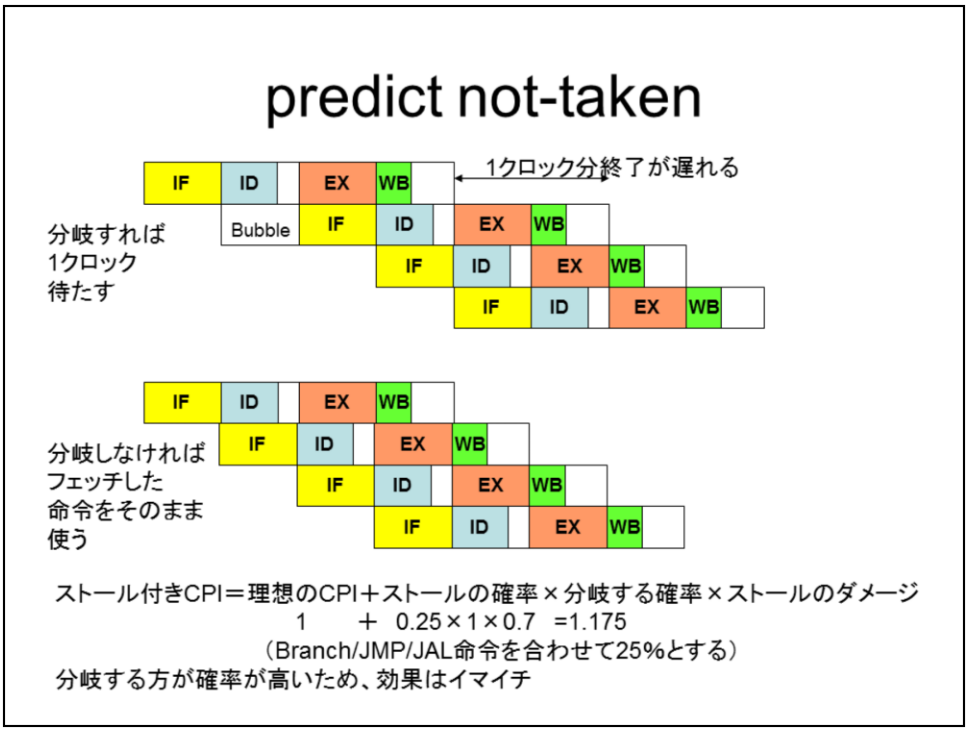
IDステージで分岐先を計算すると、、、



Branchの次の命令フェッチを1クロック遅らせる。

ストール付きCPI＝理想のCPI＋ストールの確率×ストールのダメージ
1　＋　0.25×1　＝　1.25
（Branch/JMP/JAL命令を合わせて25%とする）
これ以上はどうにもならない

先のスライドの方法を使っても、分岐命令はIDステージで実行されることになるため、分岐命令の直後の命令は実行していいのかどうか分かりません。真面目にこれに対処するためには分岐命令の後に1クロックストールさせて、次の命令を1クロック遅らせてフェッチする必要があります。このようにすると、CPIが1から1．25に増えます。分岐命令の判定をこれ以上早く行なうことができないことを考えると、この性能低下は避けられないように思います。しかし、単純な方法が二つ考えられます。

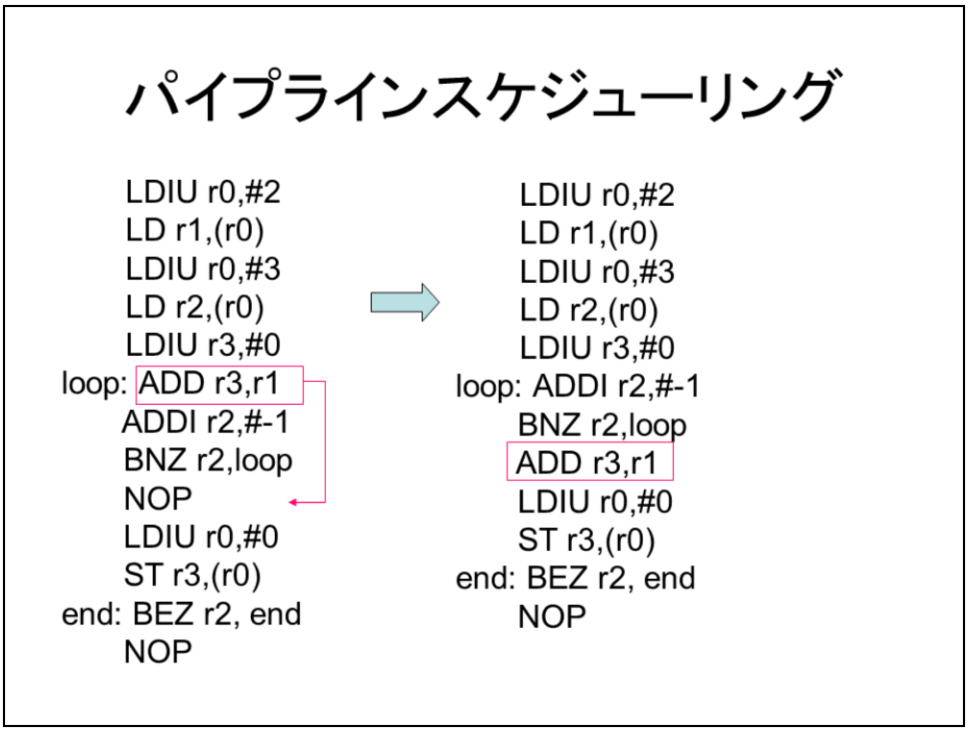


一つの方法はpredict not-takenと呼び、分岐命令が成立した場合は1クロック待たす一方、分岐しなければフェッチした命令を捨てないで流してやる方法です。この方法を使うと、ストールするのは分岐命令が成立する場合のみになります。では、分岐が成立する確率はどの程度になるでしょう？実はこれは結構高く、ここでは7割と見積もりました。なぜ高いかと言うと、コンピュータはループを繰り返し実行しますが、ループにつき1回かならずアドレスが後方に戻っているはずだからです。したがってこの方法の効果はイマイチと言えます。

遅延分岐

- 分岐命令の次の命令（遅延スロット）をパイプラインに入れてしまう。
  - 遅延スロットの命令は必ず実行される
    →POCOの場合は遅延スロットは1
  - つまり、遅延の効き目が遅い
  - 有効な命令を入れてやる必要がある
  - → パイプラインスケジューリング
  - どうしても埋まらなければNOPを埋める

もう一つの方法は遅延分岐と言って、成立しようがしまいが、次の命令をパイプラインに入れて実行してしまう方法です。すなわち、分岐命令の直後の命令（これを分岐スロット内の命令と呼びます）は必ず実行されます。このことは言葉を変えると、分岐は1命令分効き目が遅いと考えることができます。このためこの方法を遅延分岐と呼びます。分岐スロットに有効な命令を入れてやることで、オーバーヘッドはなくなります。命令の順番を入れ替えて遅延スロットに有効な命令を入れることをパイプラインスケジューリングと呼びます。どうしても埋まらない場合はNOP命令で埋めてやります。



2番地の内容と3番地の内容の掛け算を行なうプログラムで遅延スロットの埋め方を説明しましょう。元のプログラムは左のようにBNZの後にはNOPを入れます。NOPはバブルと同じなので、このプログラムはループ1回につき1クロック分オーバーヘッドが掛かることが分かります。ところがここにADD r3,r1を移動したらどうなるでしょう。一見右のプログラムにおいてADD r3,r1はループの外に出ているようですが、BNZは遅延分岐なので、直後のADD命令は必ず実行されます。結果として右と左のプログラムは同じ結果を出力します。しかし、ループ内にNOPが入らない右のプログラムの方がループを回る回数に相当するクロック数分速いことになります。p3kaiの下の例題プログラムを使ってちゃんと答えが出ることを確認しましょう。

遅延分岐のPOCOのVerilog記述

```
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n) pc <= 0;
  else if(pcsel)
    pc <= pc + {{8{imm[7]}},imm};
  else
    pc <= pc + 1;
end
…
assign pcsel = (bez_op & fwda == 16'b0) | (bnz_op &
   fwda != 16'b0) ;
```

では、このためのVerilog記述を紹介します。irにとってこられた命令が分岐命令かどうか、これが成立するかを調べます。成立すればpcselが1になります。条件を調べるレジスタはフォワードされた結果を使わなければならないのでfwdaを使うことに注意してください。後はpcselを1の時だけpcを飛び先に設定します。この記述はJMP、JR、JALを含んでいませんが、これは演習に譲ることにします。

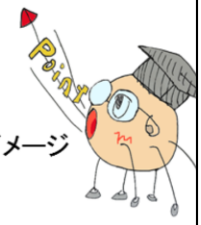
本日のまとめ

- コントロールハザードは、分岐命令によっておきる
- 分岐命令をIDステージで終わらせてしまう
  - IFステージに専用加算器が必要
  - クリティカルパスが延びる
- それでも1命令分バブルが生じる
  - predict not-taken
    - 分岐が成立したときだけストールさせる
  - 遅延分岐
    - 分岐命令直後の命令をパイプラインに入れて実行してしまう
    - 効き目が遅い分岐と考える
    - 直後の命令に有効な命令を埋められればオーバーヘッドはなくなる
    - うまく埋められなければNOPを埋める

理想CPI＋分岐命令の確率×遅延スロットが埋まらない確率×ダメージ

1＋　0．25×0．2×1＝1．05

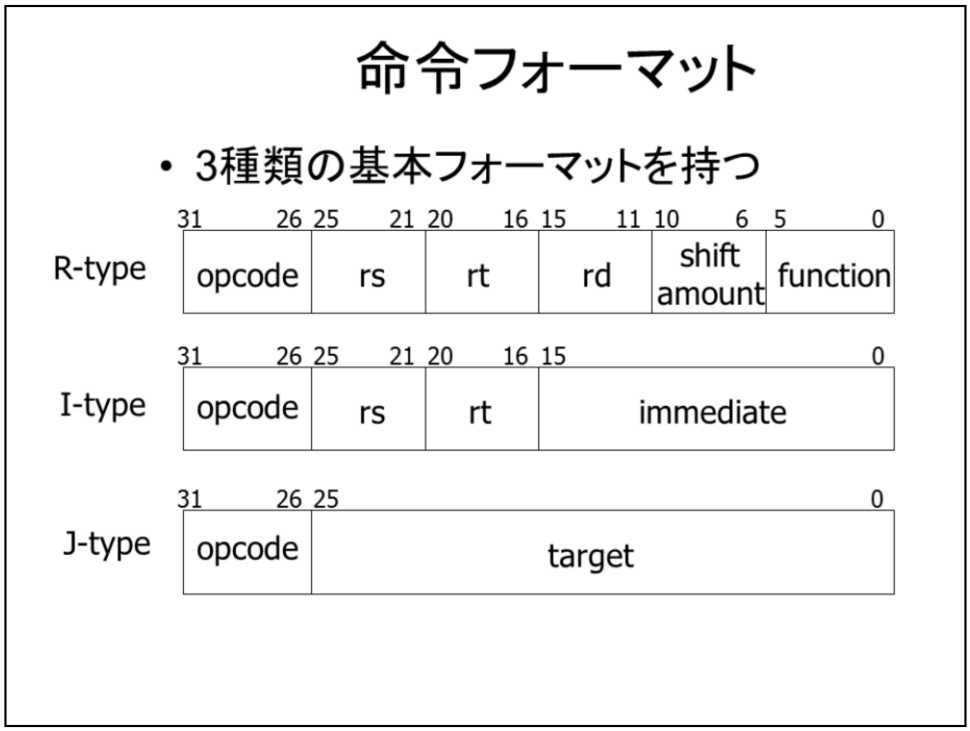
たいした性能低下にはならない



インフォ丸が教えてくれる今日のまとめです。

MIPSの命令セット

- 32ビットレジスタが32個ある
- 3オペランド方式
- LD，STはディスプレースメント付き
  - ADD　R3，R1，R2
  - ADDI　R3，R1，＃1
  - LD　R3，　10（R1）
  - ST　R1，　20（R2）
  - BEQ　R1，R2，　loop



・op(opcode):命令の種類を表すオペコードフィールド。

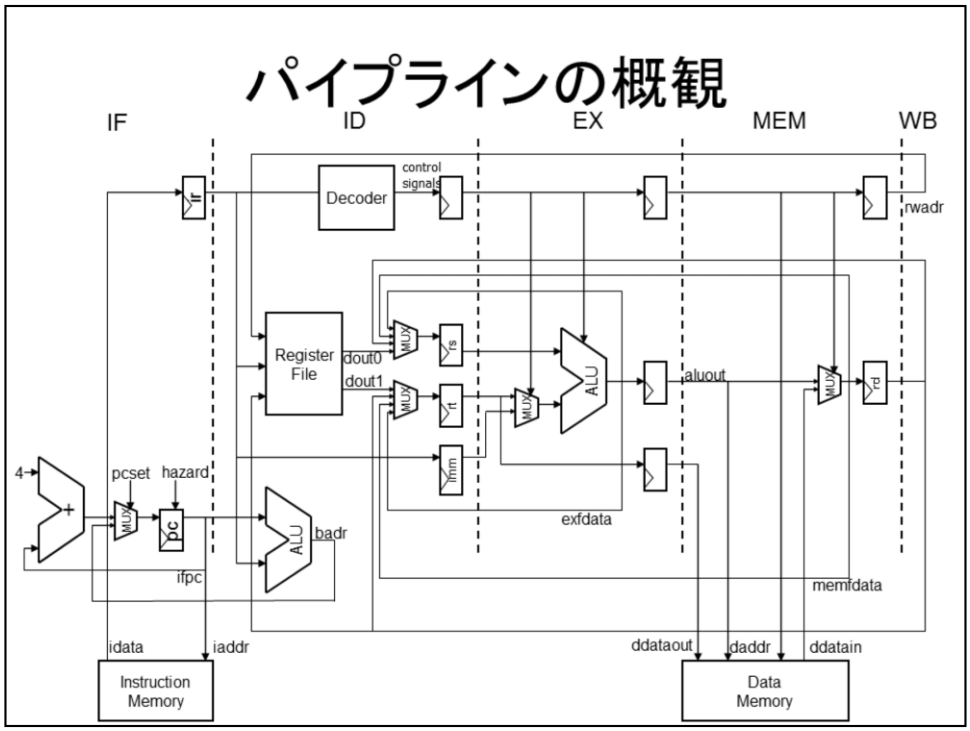
・FUNC(functional code):opフィールドで表現しきれない場合に補助オペコードとしてopフィールドを拡張する形で用いるフィールド。PICO-16ではopフィールドで2オペランドの算術論理演算命令を示し、ADDやSUBといった演算の種類をFUNCフィールドで特定する。

・Rd(destination register):算術論理演算の計算結果やメモリからロードしてきたデータといった処理の結果を格納するレジスタ番号を指定するフィールド。PICO-16の演算ではRdの内容を1つのソースデータとしてRsの内容とADD等の演算を行い、Rdの内容を書きつぶして結果を書き込む。

・Rs(source register):算術論理演算のソースデータを格納しているレジスタ番号を指定するフィールド。PICO-16では1つしか指定できないので2オペランド命令であるが、多くの32ビットアーキテクチャではRsを2つとRdを指定できる。

・Immediate:命令に直接値を埋め込むのに使用するフィールド。PICO-16では算術論理演算のソースやレジスタに直接immediateの値をロードする際と、条件分岐命令の相対分岐オフセットに用いる。

・Offset:immediateと同様に命令に直接値を埋め込むが、offsetフィールドはPC(プログラムカウンタ)のオフセットを指定するのに用いる。相対分岐命令で大きなプログラムの範囲を指定できるように、多くの命令セットにおいてoffsetフィールドができるだけ長くとれるように工夫してある。



分岐(Branch)命令の処理はIFステージとRFステージで行う。条件分岐命令では条件の判別をどこで行うかによって分岐先のアドレスをフェッチするか続くアドレスをフェッチするかが決定するまでのクロックサイクル数が変化する。

分岐命令によって分岐先に飛ぶ前に、分岐命令の次の命令を必ず実行すると定める方法がある。この方法を用いると分岐先が決定するまで待たねばならないクロックサイクルに分岐命令の次の命令を実行して無駄をなくすことができる。その続く命令を実行するスロットをDelay Slotと呼ぶ。

今回のパイプラインではRFステージで条件を読み出すのと並列に分岐先を計算してIFステージに送り、IFステージでは条件の真偽が判別されしだいそれに従って続きのpcと計算された分岐先のpcを選択して命令フェッチを行う。この構造をとるとDelay Slotを1クロックサイクル必要とするが、構造は単純となる。Delay Slotが存在するとプログラムのコード自体をそのDelay Slot数に対応させて書き換える必要がある。

MIPS R4000の8ステージパイプライン

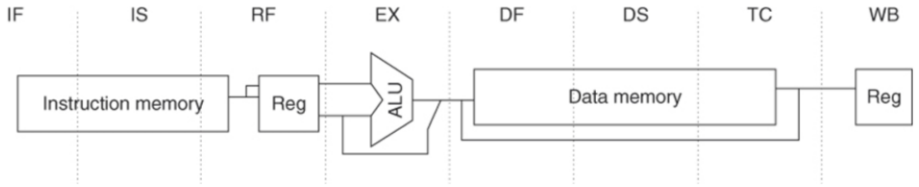


Figure C.41 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, since we cannot write the data into the register until we know whether the cache access was a hit or not.

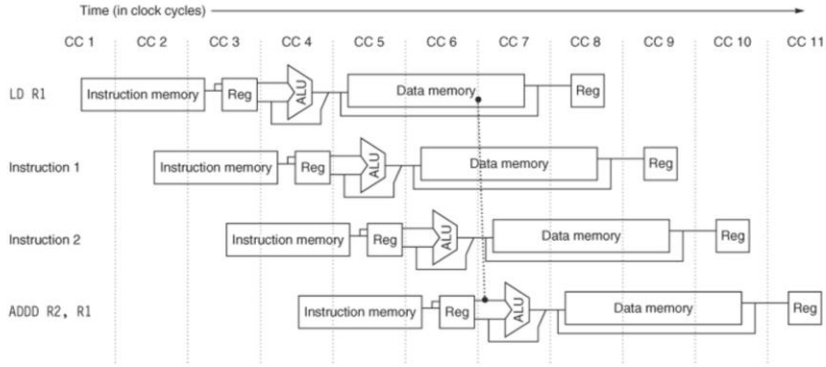


Figure C.42 The structure of the R4000 integer pipeline leads to a 2-cycle load delay. A 2-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

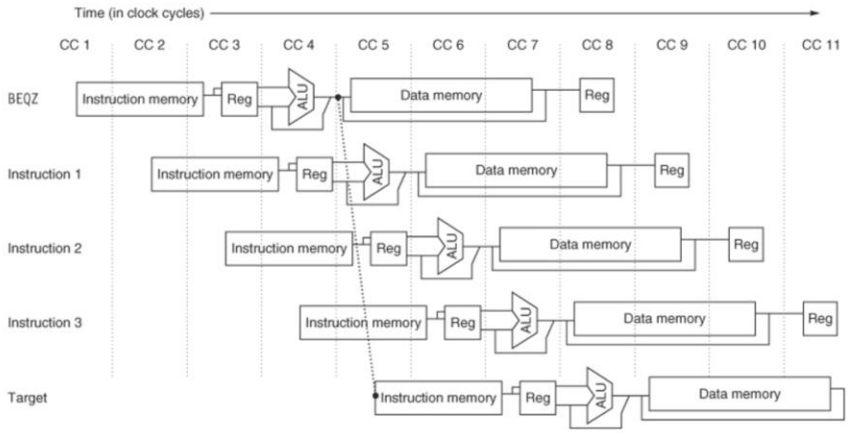
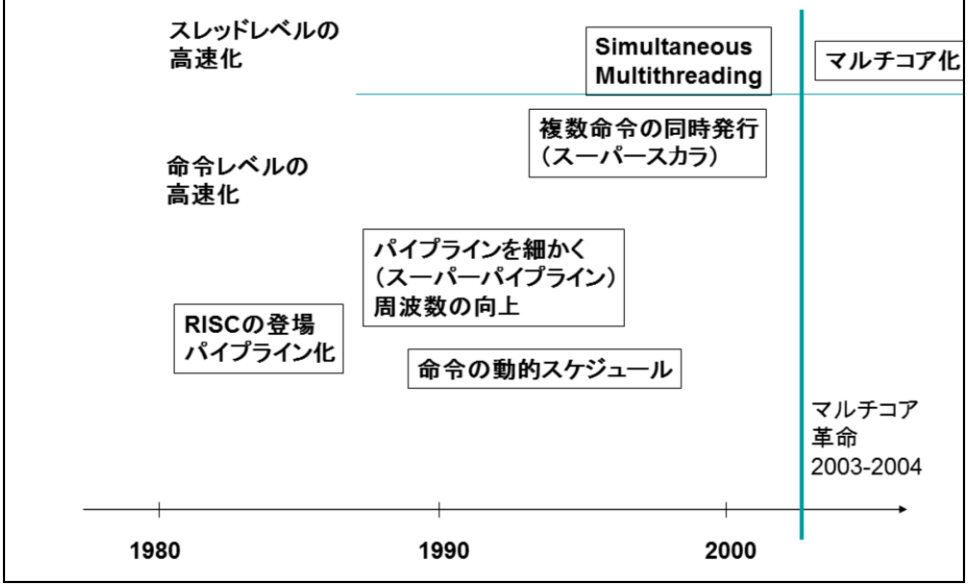


Figure C.44 The basic branch delay is 3 cycles, since the condition evaluation is performed during EX.

高速化の流れ


スレッドレベルの
高速化
Simultaneous
Multithreading
マルチコア化
複数命令の同時発行
（スーパースカラ）
命令レベルの
高速化
パイプラインを細かく
（スーパーパイプライン）
周波数の向上
RISCの登場
パイプライン化
命令の動的スケジュール
マルチコア
革命
2003-2004
1980
1990
2000

浮動小数演算

- MIPSの浮動小数演算は独立性が高い
  →コプロセッサCP1で扱う
- ADD.S F0,F1,F2　　単精度（32ビット）
- ADD.D F0,F2,F4　　倍精度（64ビット）

倍精度の場合はF0,F1をセットにして64ビットとして扱う

- ADD、SUB、MUL、DIV命令がある
- MUL、DIVは整数演算でも特殊
  - R3000では専用レジスタを用いる
- L.S、L.D、S.S、S.D　ロード・ストア命令
- 演算結果はフラグに記憶
- GPRとFPR間の転送命令がある

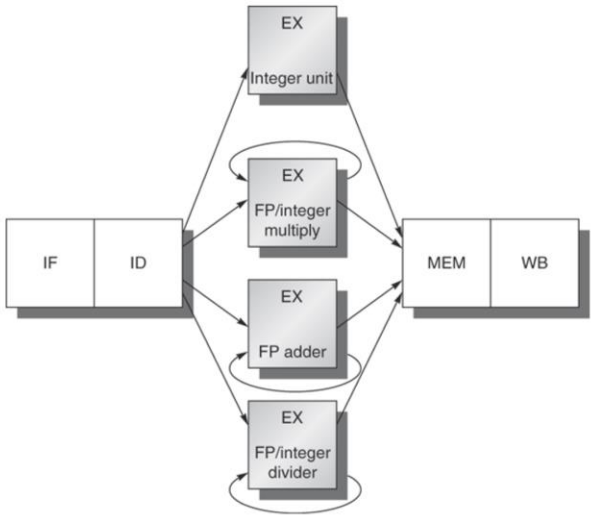


Figure C.33 The MIPS pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

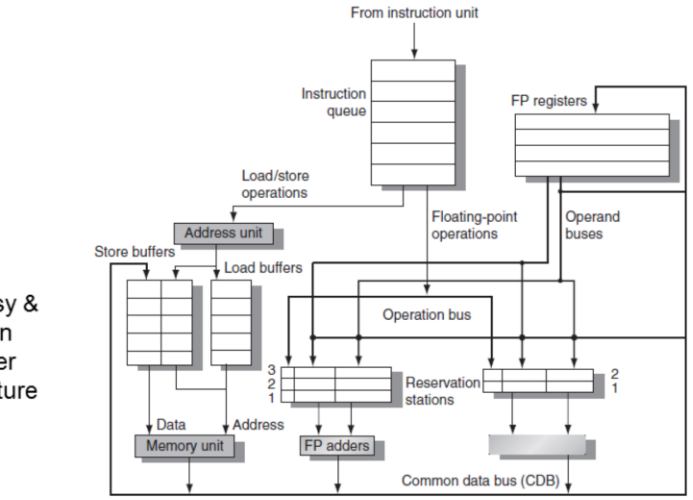
Ed.pdf – Adobe Reader

(F) 編集(E) 表示(V) 文書(D) ツール(T) ウィンドウ(W) ヘルプ(H)

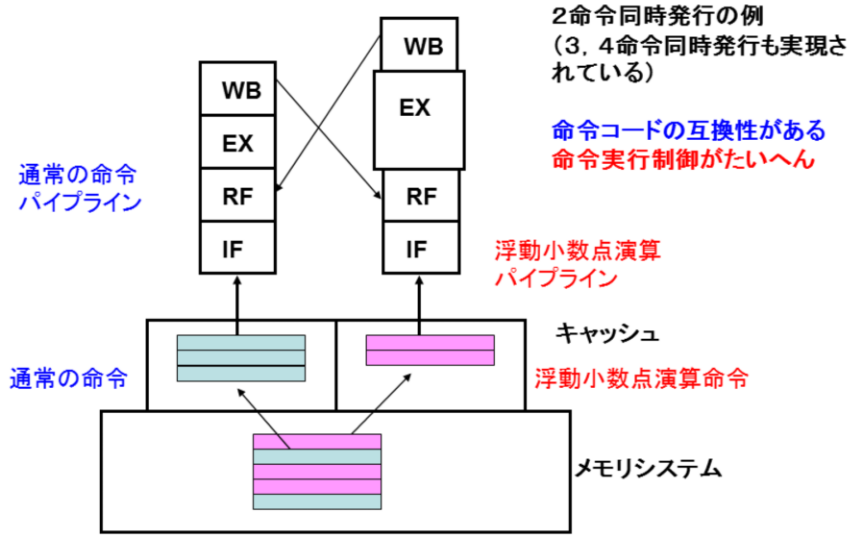
173 (202 / 857) 115% 検索

命令のアウトオブオーダ実行
トーマスローのアルゴリズム

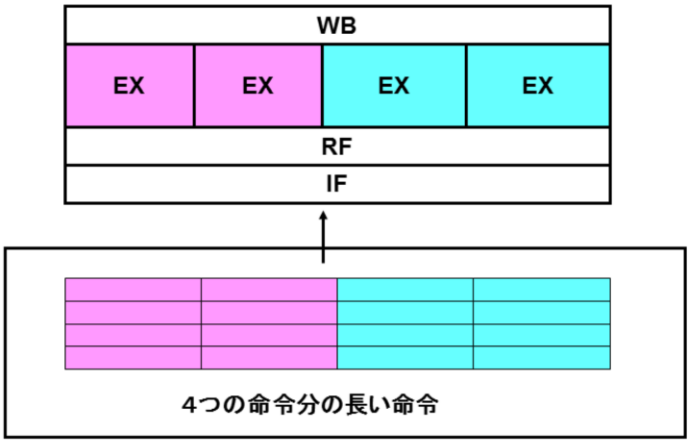
Hennessy & Patterson Computer Architecture より



スーパスカラ方式

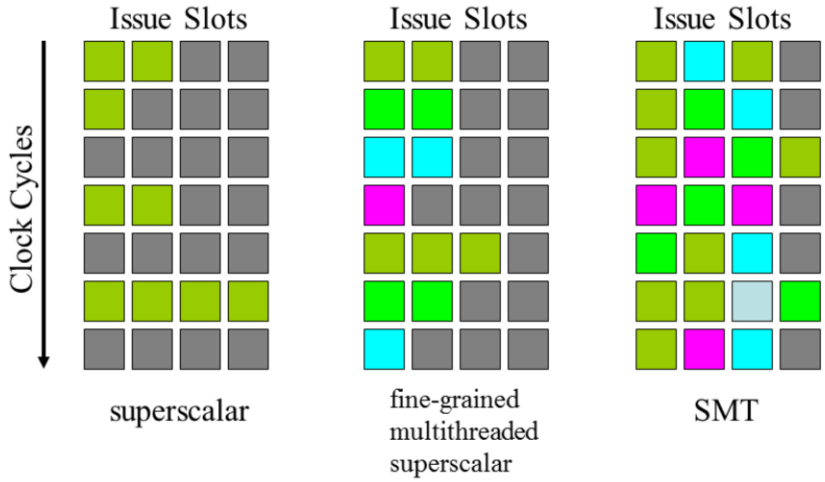

2命令同時発行の例
（3，4命令同時発行も実現されている）
命令コードの互換性がある
命令実行制御がたいへん
通常の命令
パイプライン
WB
EX
RF
IF
WB
EX
RF
IF
浮動小数点演算
パイプライン
キャッシュ
通常の命令
浮動小数点演算命令
メモリシステム

VLIW（Very Long Instruction Word）方式



実行の制御はコンパイラがあらかじめ行うので制御は簡単
コードの互換性がない

マルチスレッドとSMT(Simultaneous Multi-Threading)



マルチコア、メニーコア

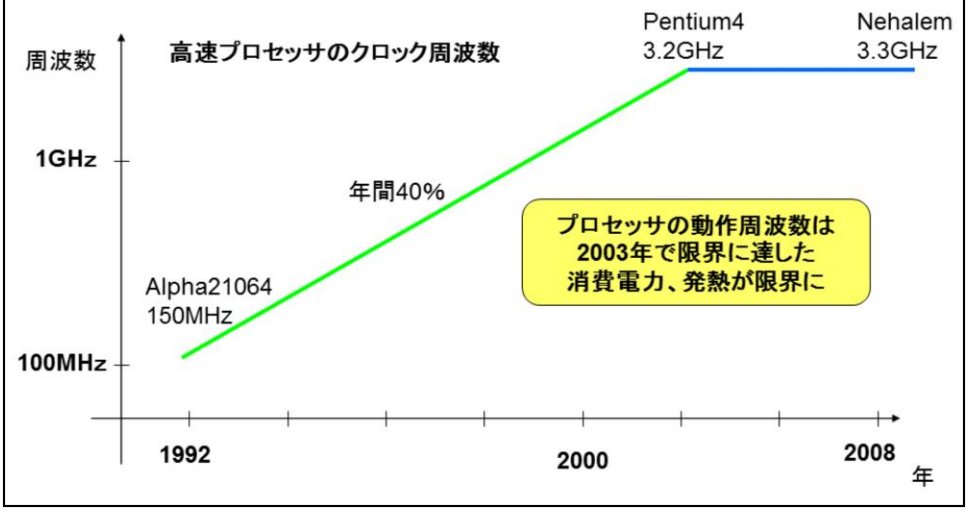
- 動作周波数の向上が限界に達する
  - 消費電力の増大、発熱の限界
  - 半導体プロセスの速度向上が配線遅延により限界に達する
- 命令レベル並列処理が限界に達する
- メモリのスピードとのギャップが埋まらない

→ マルチコア、メニーコアの急速な発達

マルチコア革命 2003-2004年

- プログラマが並列化しないと単一プログラムの性能が上がらない

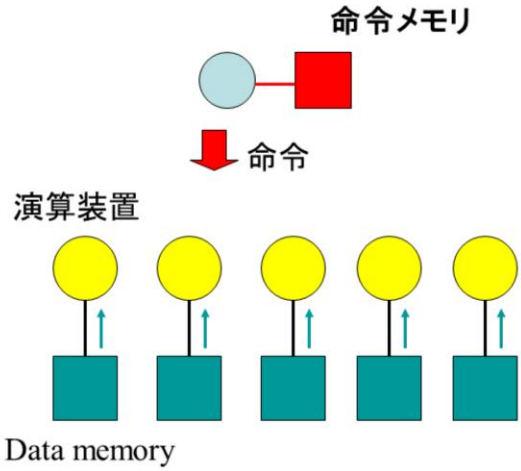
クロック周波数の向上


周波数
高速プロセッサのクロック周波数
Pentium4
3.2GHz
Nehalem
3.3GHz
1GHz
年間40%
プロセッサの動作周波数は
2003年で限界に達した
消費電力、発熱が限界に
Alpha21064
150MHz
100MHz
1992
2000
2008
年

Flynnの分類

- 命令流(Instruction Stream)の数：M(Multiple)/S(Single)
- データ流（Data Stream）の数：M/S
  - SISD
    - ユニプロセッサ（スーパスカラ、VLIWも入る）
  - MISD：存在しない（Analog Computer）
  - SIMD
  - MIMD

一人の命令で皆同じことをする SIMD



半導体チップ内でたくさんの演算装置を動かすには良い方法

アクセラレータ（普通のCPUにくっつけて計算能力を加速する加速装置）の多くはこの方式

安くて高いピーク性能が得られる
→パソコン、ゲーム機と共用

近年、CPUとGPUやCELLといったマルチコアアクセラレータを組み合わせて使うハイブリッドの計算環境が普及しています。

例えばTOP500を見ると、TSUBAMEのNVIDIA GPUはもちろんですが、↑こういったATIのGPUを使ったスパコンも存在します。

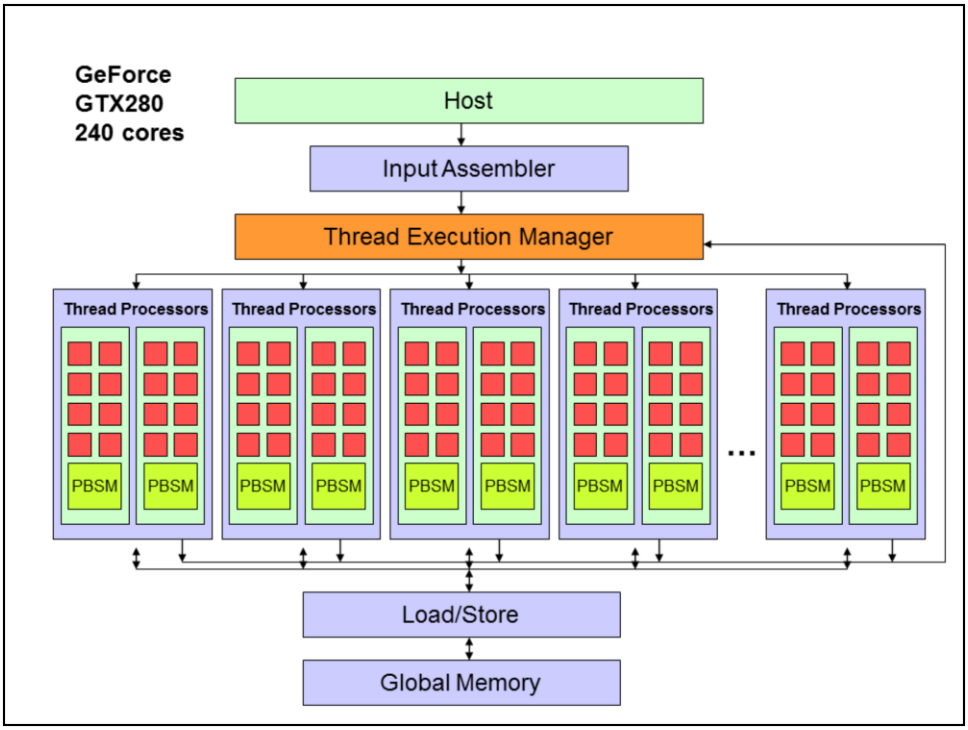
またCell B.E.を搭載したアクセラレータもあります。

これらのアクセラレータを使って高い性能が得られるのですが、アクセラレータごとに異なる環境を用いなければなりませんでした。

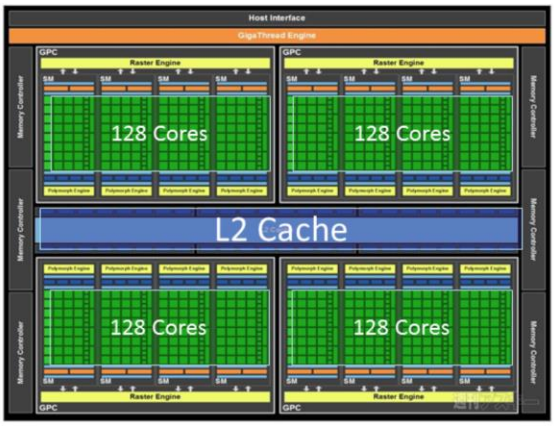
そこで、Open CLという開発環境が登場しました。

OpenCLは、マルチコアプロセッサ向けの共通プログラミング環境で、Open CLにより、

異なるアーキテクチャでもCライクの同一ソースコードで開発が可能になりました。


GeForce
GTX280
240 cores
Host
Input Assembler
Thread Execution Manager
Thread Processors
PBSM
Load/Store
Global Memory

GPU (NVIDIA's GTX580)



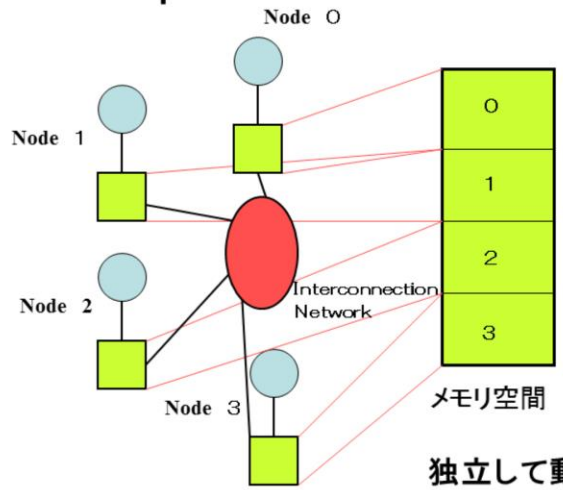
128個のコアは
SIMD動作をする

4つのグループは
独立動作をする

もちろん、このチップを
たくさん使う

512 GPU cores (128 X 4)
768 KB L2 cache
40nm CMOS　550 mm^2

MIMD（Multipe-Instruction Streams/ Multiple-Data Streams)

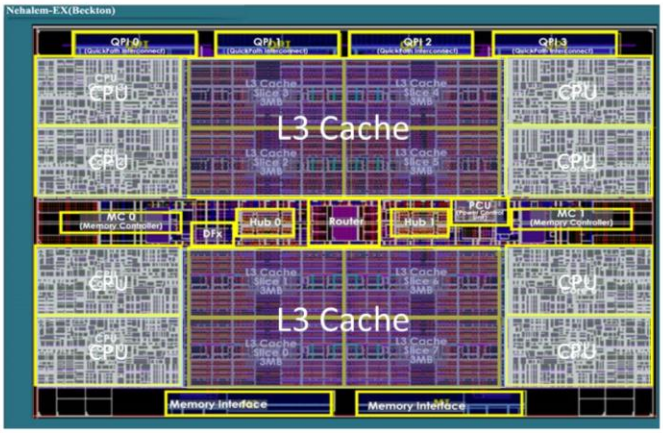


独立して動けるプロセッサを複数使う

MIMD（Multipe-Instruction Streams/ Multiple-Data Streams)の特徴

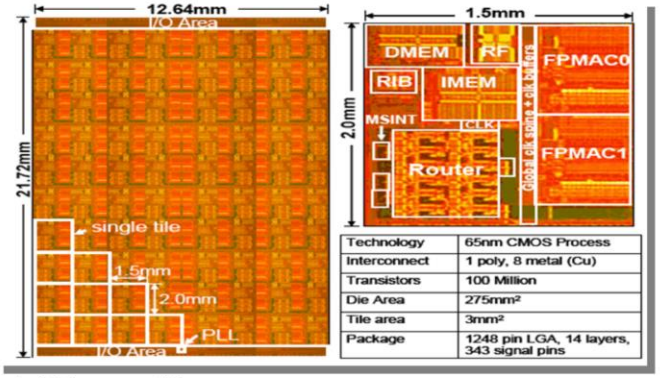
- 自分のプログラムで動けるプロセッサ（コア）を多数使う
  - 同期：足並みを揃える
  - データ交信：共通に使うメモリを持つなど
- 最近のPC用のプロセッサは全部この形を取っている
- 最近はスマートフォン用のCPUもマルチコア化

Multi-Core (Intel's Nehalem-EX)



8 CPU cores
24MB L3 cache
45nm CMOS 600 mm^2

Intel 80-Core Chip



Intel 80-core chip [Vangal,ISSCC'07]

まとめ

- 汎用プロセッサのマルチコア化は現在絶好調進行中
  - 世代が進む毎に2ずつ増えている
  - しかし、コア数をこれ以上増やしても良いことがないかも、、、
- メニーコア
  - GPUなどのアクセラレータの性能向上は進む
  - メニーコアによるクラウドコンピューティング

演習

1. sum.asmをスケジュールし、遅延スロットを埋めて実行せよ。 提出物　sum.asm
2. 遅延分岐付きのpocop.vをiseで合成せよ。提出物　動作周波数、利用LUT数