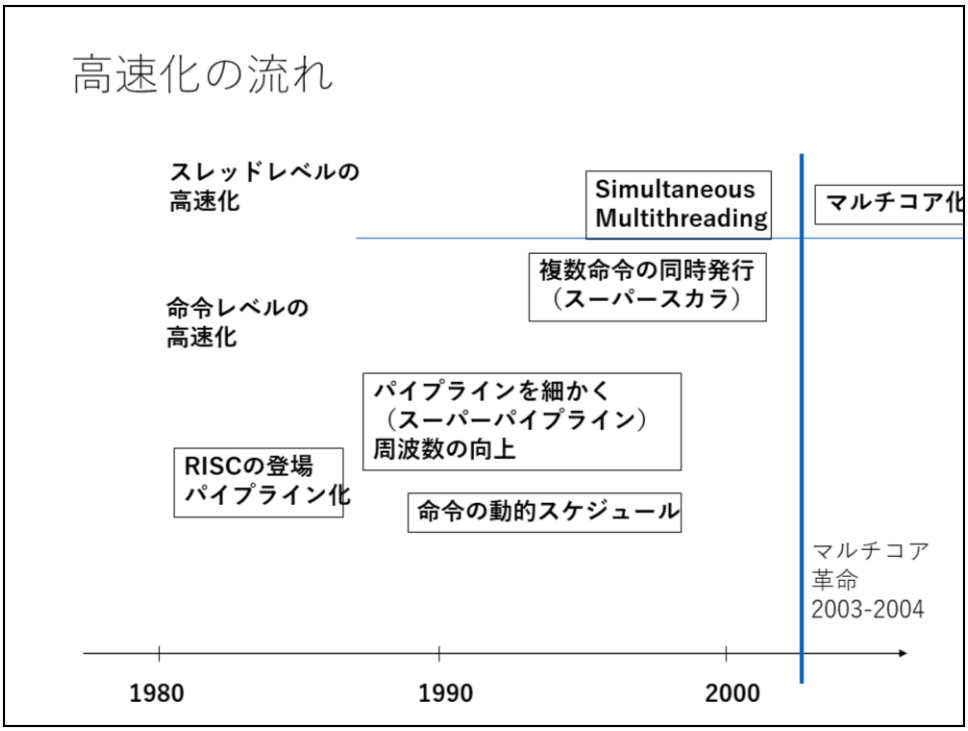
コンピュータアーキテクチャB
さらなる高速化

天野　hunga@am.ics.keio.ac.jp

最後にCPUのさらなる高速化の話を紹介します。



前回までに紹介したパイプライン化は1980年代のCPUの性能向上を後押ししました。しかし、90年代になってこれが限界に達すると、様々な命令レベル高速化技術が発達しました。ポイントは、今のバイナリをそのまま高速化することにありました。

CPUの高速化手法

- パイプラインの段数を増やしていく
  - →動作周波数を上げる
  - スーパーパイプラインと呼ぶ場合もある
  - メモリの遅延を回避する
- 複数命令の同時発行
  - スーパースカラ
  - VLIW
- 命令の静的スケジューリング
  - ループアンローリング
- 後は雰囲気のみ
  - 命令の動的スケジューリング
  - 投機的実行（Speculative Execution）
  - 分岐予測

ここで紹介する手法は、VLIWと静的スケジューリングを除いては、配布されたコードをそのまま高速化するための手法です。現在の高性能のCPUはこれらを全て利用し、さらにマルチコア化がなされています。

MIPS R4000の8ステージパイプライン

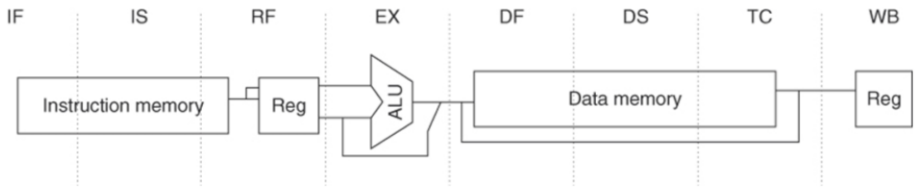


Figure C.41 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction 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.

パイプラインの段数を増やせば、その分動作周波数を上げることができます。ここでポイントは、できるだけ処理時間の長いステージを分割することです。実際にプロセッサを実装すると、処理時間の長いステージはキャッシュのアクセスであることがわかります。MIPSのR3000の次に登場したR4000では、これを分割して8段構成にして動作周波数を上げました。

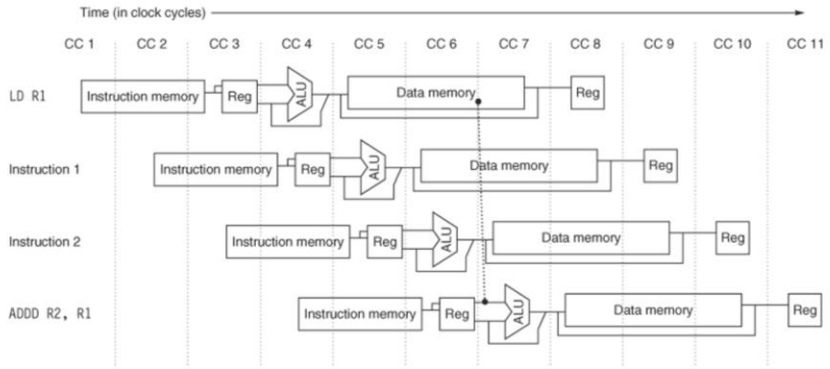


Figure C.42 The structure of the R4000 integer pipeline leads to a 2-cycle load delay. A 2-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

この場合、問題になるのがデータハザードです。ロード命令の後は2クロック待たないとその結果は使えなくなります。

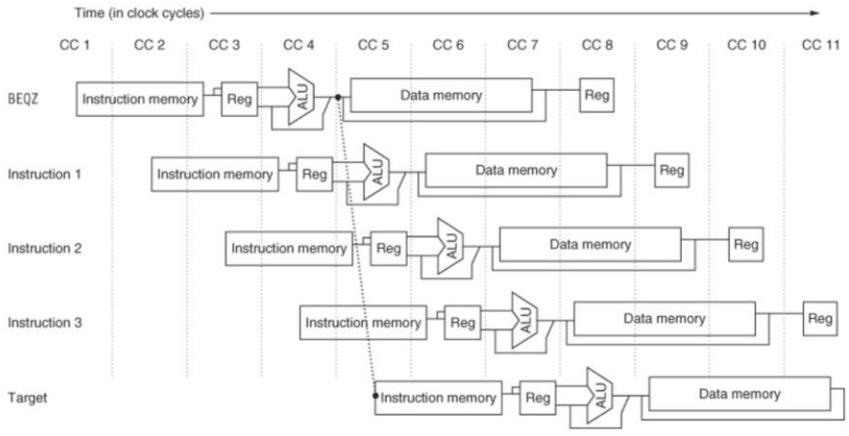
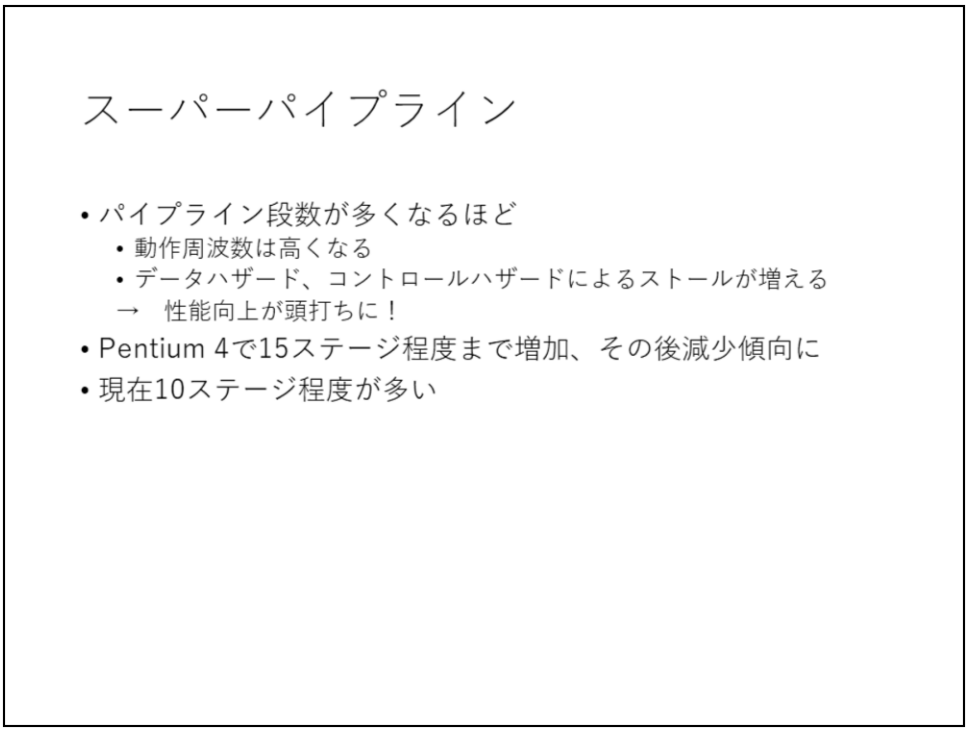
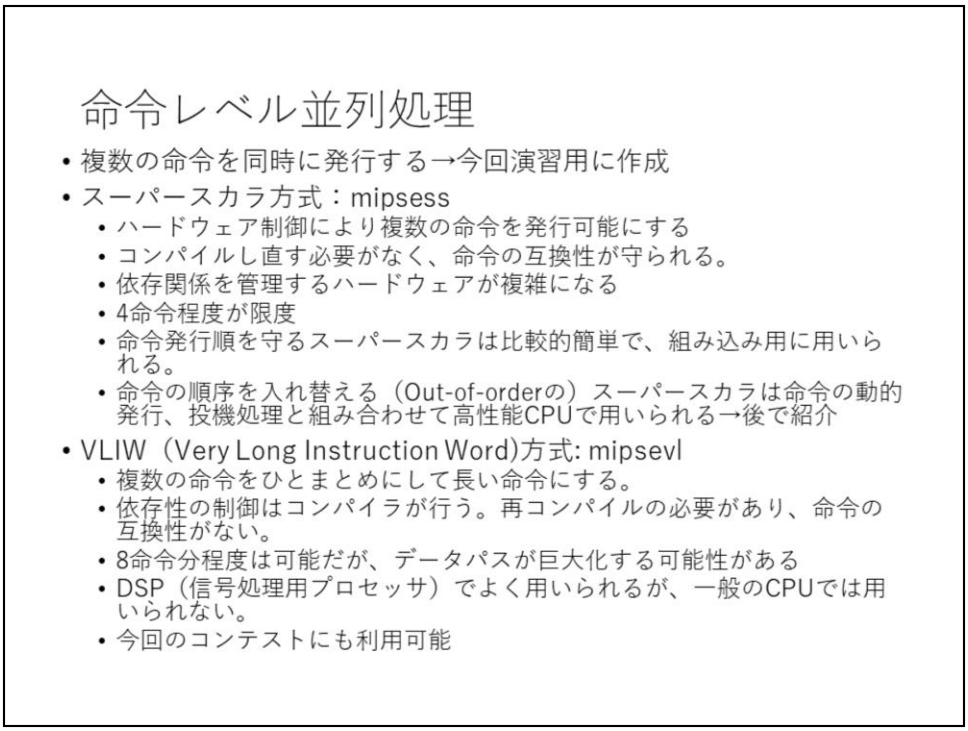


Figure C.44 The basic branch delay is 3 cycles, since the condition evaluation is performed during EX.

分岐についても3クロック待たないと飛び先まで飛ぶことができません。遅延分岐を実装すると、遅延スロットが3クロック分になります。こうなると埋めるのが大変になります。



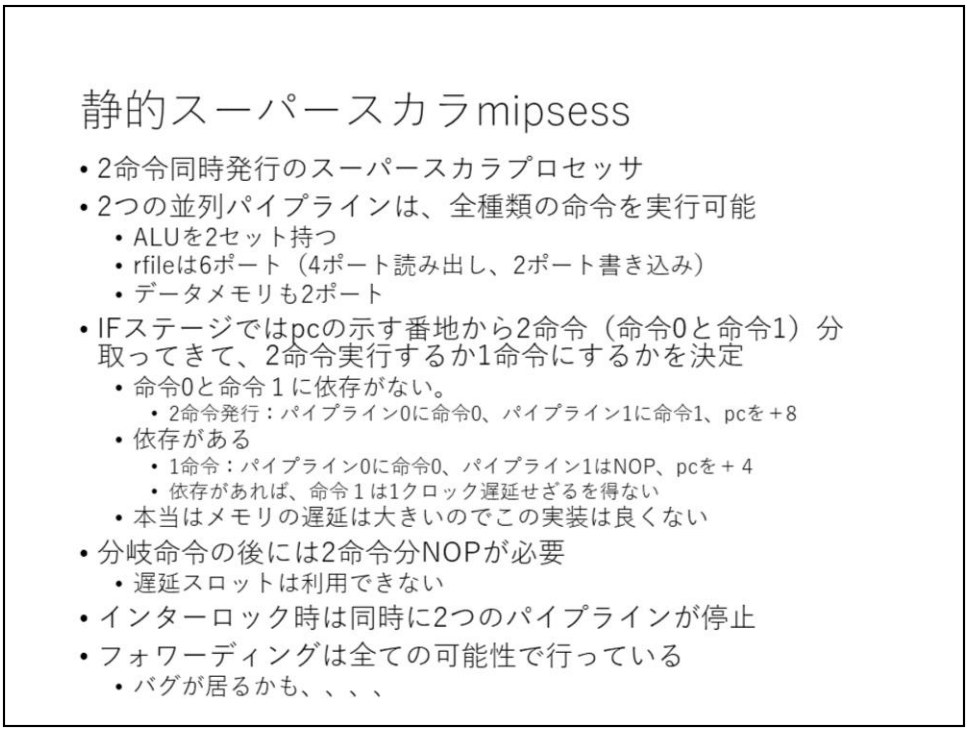
基本的なパイプラインの段数である5段を越えてパイプラインを分割することをスーパーパイプラインと呼びます。確かに分割すればするほど動作周波数は高くなりますが、データハザード、コントロールハザードによるストールが増え、性能向上は頭打ちになります。かつてPentium 4では15ステージのパイプラインを使っていましたが、その後はやや減少傾向にあり、現在は10ステージ程度が多くなっています。



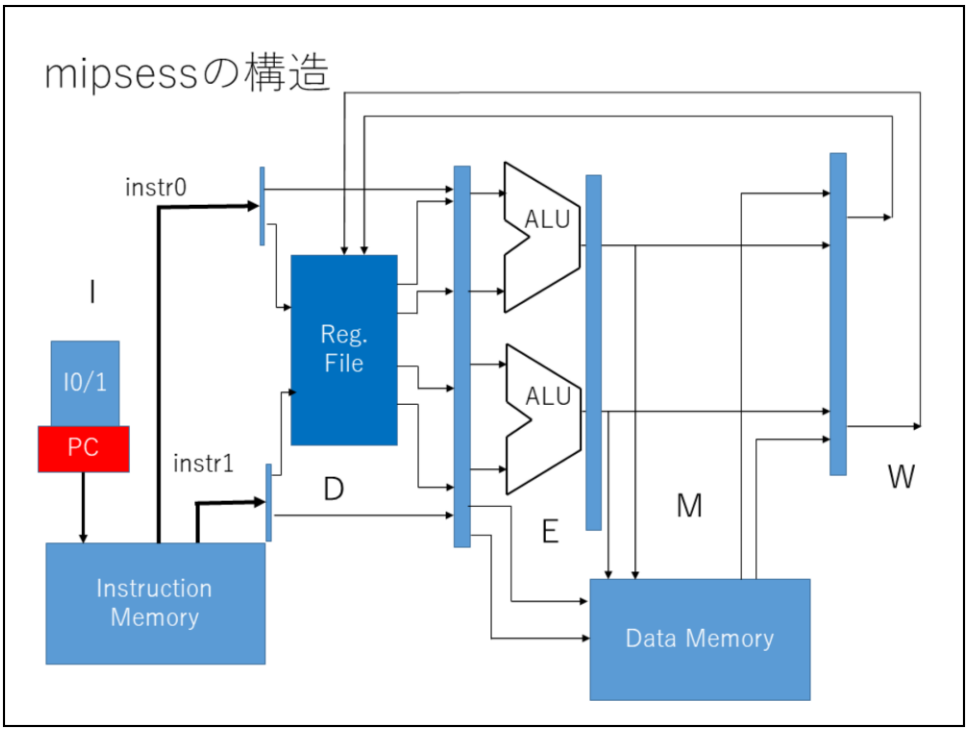
PCが指し示す命令だけでなく、その次の命令、次の次の命令を同時に発行させたらばどうでしょう？うまく同時に動くことができれば、高速化が実現できるはずです。ハードウェアの管理でこれを実現する方法をスーパースカラと呼びます。この方法は、命令の依存関係を管理するためのハードウェアが複雑になるので、4命令同時発行くらいが多いです。しかし、今までの機械語をコンパイルしなおさないで高速化できるので、ビジネスモデルに良く合うことから、現在のほとんどの高性能CPUで使われています。

一方、複数の命令をひとまとめにして長い命令として扱う方法をVLIW（Very Long Instruction Word)方式は、依存性の管理をコンパイラで行うため、制御回路は簡単になり、8命令同時発行くらいまで比較的簡単に実装できます。しかし、再コンパイルする必要があることから汎用CPUのビジネスモデルに載らず、主としてDSP(信号処理プロセッサ）で使われます。

これらの技術はコンテストで使うこともできます。



mipsessは、命令の順序を入れ替えない静的なスーパースカラプロセッサです。パイプラインを2本持ち、それぞれは全ての種類の命令実行が可能です。このため、ALUは2セット、レジスタファイルは4ポート読み出し、2ポート書き込み、データメモリも2ポートに拡張してあります。まずIFステージでPCの示す番地から2命令分（命令0、命令1）取って来て、2命令実行可能か、1命令だけにするかを判断します。2命令同時実行可能なのは命令0と1の間に依存関係がない場合で、この時は命令0をパイプライン0に、命令1をパイプライン1にいれて、PCを＋8します。依存があれば、パイプライン0に命令0を入れて、パイプライン1にはNOPを入れ、PCを＋4します。本来、命令メモリの遅延が大きく、あまり難しい判断をさせるとクリティカルパスになってしまうIFでこんなことをやるのは良くないのですが、この場合、ま、しょうがないでしょう。分岐命令の後は、2つのパイプラインに同時にNOPが入ります。面倒を避けるため、遅延分岐は使いません。インターロックは2つのパイプラインを同時にストップし、フォワーディングは全てのケースにおいて行っています。

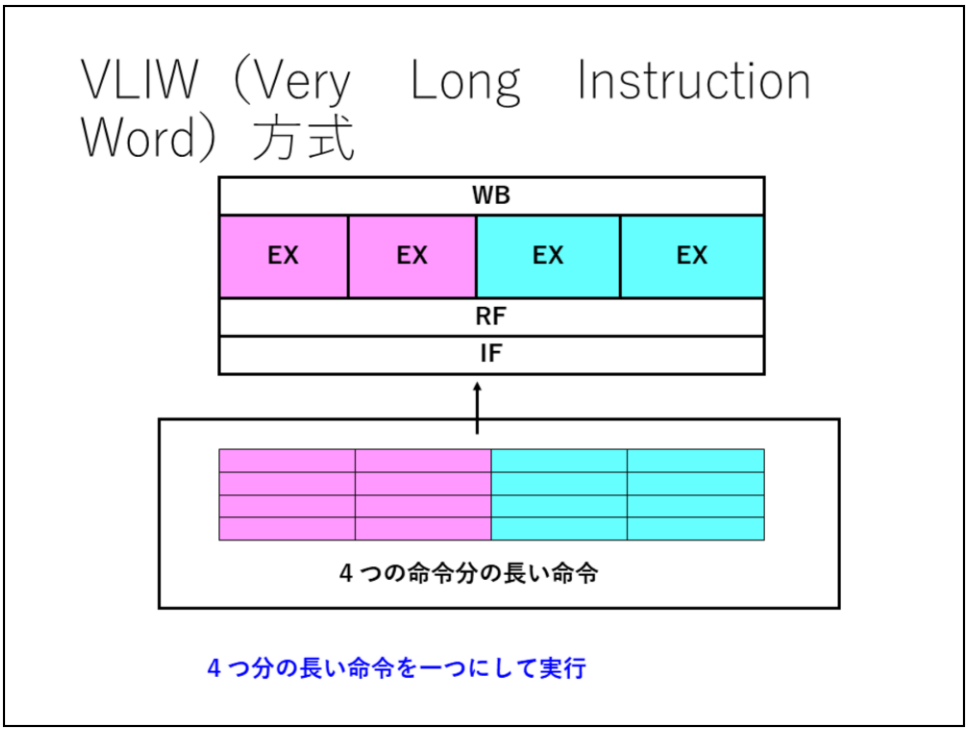


mipsessのブロック図を示します。パイプライン0，1がレジスタファイル、メモリを共有しています。静的スーパースカラで2命令同時発行なので、比較的構造は簡単です。

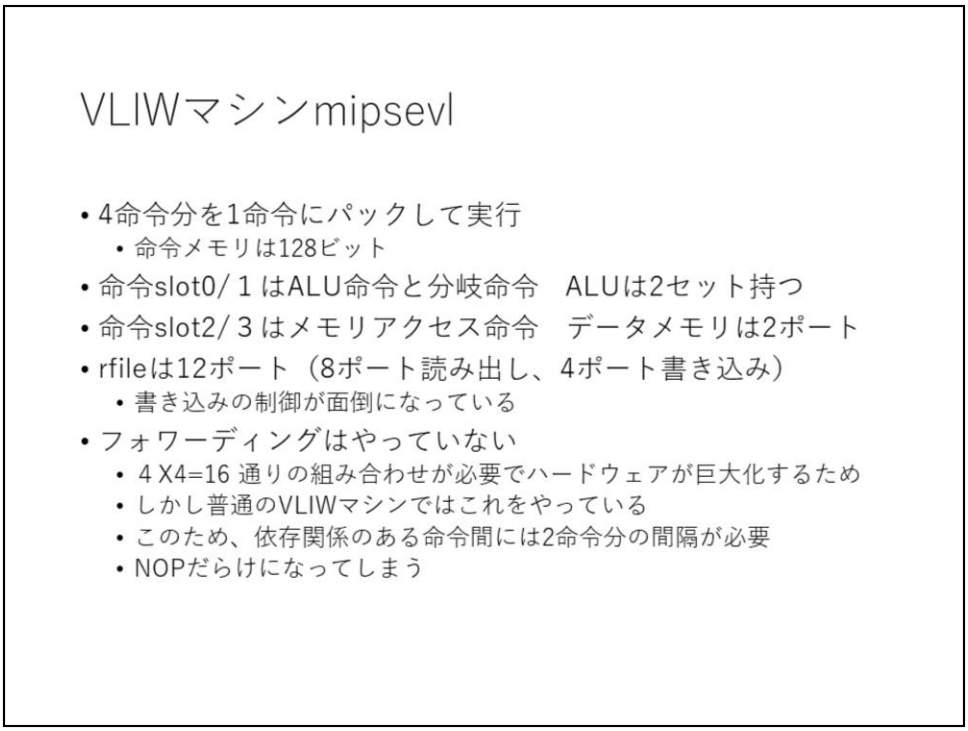
例題：　mipsessを動かしてみよう

- ディレクトリはmipsess
- 掛け算のプログラムmult.asmと
- 0番地から9番地までの10.個の数の総和を求めるaddarray.asm が用意されている。
- makeでiverilogを実行してくれるので、
- ./asm.pl mult.asm –o imem.datで今まで同様にアセンブルする
- ./a.outで実行する。7fff番地に答を書いて終了する。
- 答、count: 実行したクロック数、stall:ストールしたクロック数、2slot:2命令同時発行したクロック数を表示してくれる。
- 実行して様子を見よう。

ではちょっとmipsessを動かして様子を見ましょう。



VLIW（Very Long Instruction Word)型のコンピュータでは、スーパースカラ型における制御回路の複雑さを避けるため、命令の依存性を満足して命令を発行するタイミングをコンパイラが検出して、これらをくっつけた長い命令の形にして実行します。命令のそれぞれのフィールドは、通常の一命令分に相当しますが、これらを独立に動かすことはできません。何もやることがない場合、そのフィールドはNOPで埋めます。



mipseのVLIW版であるmipsevlを作りました。これは4命令分を1つにパックして実行するため、命令長は128bitになります。それぞれの命令に相当する部分をスロットと呼びます。スロット0・1はALU命令と分岐命令、2・3は、メモリアクセス命令を実行可能です。このため、データメモリは2ポートに、レジスタファイルは8ポート同時読み出し、4ポート書き込みの12ポートに拡張しています。このため、書き込みの制御が面倒になっています。今回の実装は手抜きでフォワーディングをやっていません。このため依存関係のある命令間には2命令分の間隔が必要になります。普通のVLIWはインターロックが起きないようにスケジュールはしますが、フォワーディングはやっています。

mipsevlでのaddarray.asmの実行

	slot0	slot1	slot2	slot3
	addi $1,$0,0	add $2,$0,$0	nop	nop
	addi $3,$0,8	nop	nop	nop
	nop	nop	nop	nop
lp:	addi $1,$1,4	nop	lw $4,0($1)	nop
	addi $3,$3,-1	nop	nop	nop
	nop	nop	nop	nop
	add $2,$2,$4	nop	nop	nop
	bne $3,$0,lp	nop	nop	nop
	nop	nop	nop	nop
	nop	nop	sw $2,0x7fff($0)	nop
end:	beq $0,$0,end	nop	nop	nop
	nop	nop	nop	nop

ファイル中では1スロット1行になっているので注意！

命令中に並んだ数の総和を求めるaddarray.asmをmipsevlで実行する様子を示します。依存性のある命令の間隔を2つ空けなければならないため、スロットはあまり埋まって居ません。

例題：　mipsevlを動かしてみよう

- ディレクトリはmipsevl
- 掛け算のプログラムmult.asmと
- 0番地から9番地までの10個の数の総和を求めるaddarray.asmが用意されている。
- makeでiverilogを実行してくれるので、
- ./asm.pl mult.asm –o imem.datで今まで同様にアセンブルする
- 命令4行分が一つの命令になる。
  - 最初の二つはALUまたは分岐命令
  - 次の二つはLWかSW命令
  - 依存のある命令間には2つ命令の間隔を置く
  - このルールを守らないとうまく実行ができない！
- ./a.outで実行する。7fff番地に答を書いて終了する。
- 答、count: 実行したクロック数、stall:ストールしたクロック数、instc:NOP以外の命令数を表示してくれる。
- 実行して様子を見よう。

では、このプログラムを実行してみましょう。あまりこれでは性能が上がってないことがわかります。

ループアンローリング

- addarray.asm

```
        addi $1,$0,0
        add $2,$0,$0
        addi $3,$0,8
lp:  lw $4,0($1)
        add $2,$2,$4
        addi $1,$1,4
        addi $3,$3,-1
        bne $3,$0,lp
        nop
        nop
        sw $2,0x7fff($0)
end:    beq $0,$0,end
        nop
        nop
```

- addarray_lu.asm

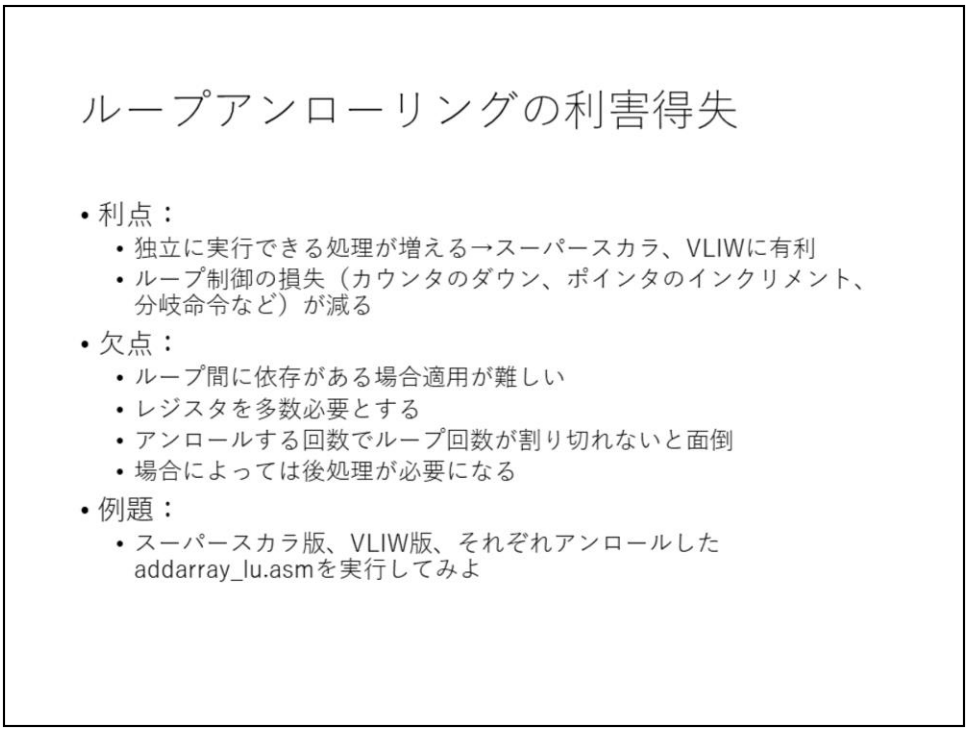
```
        addi $1,$0,0
        add $2,$0,$0
        add $6,$0,$0
        addi $3,$0,8
lp:     lw $4,0($1)
        lw $5,4($1)     //2つ分lw
        add $2,$2,$4
        add $6,$6,$5  //2つ分加算
        addi $3,$3,-2  //カウンタは2減らす
        addi $1,$1,8   //ポインタは8進める
        bne $3,$0,lp
        nop
        nop
        add $2,$2,$6   //この分は損する
        sw $2,0x7fff($0)
end:    beq $0,$0,end
        nop
        nop
```

複数命令を同時発行するプロセッサの性能を上げるためには、あらかじめ細工をして、依存関係を持たないで実行できる命令の数を増やしてやるのが効果的です。このために良く用いられるテクニックがループアンローリングです。このテクニックは通常ループ1回で1つ実行する処理を、ループを複数個展開して実行することで、依存性のない命令を増やすテクニックです。左が普通のaddarrayで1回のループで配列一個分を足し算します。これをアンロールしたのが右のコードで、1回分のループで、二つのデータを持ってきて加算します。積算するレジスタは2つ用意し、カウンタは2減らし、ポインタは8進めます。

mipsevlでのループアンローリング

	slot0	slot1	slot2	slot3
	addi $1,$0,0	add $2,$0,$0	nop	nop
	addi $3,$0,8	add $6,$0,$0	nop	nop
	nop	nop	nop	nop
lp:	addi $1,$1,8	nop	lw $4,0($1)	lw $5,4($1)
	addi $3,$3,-2	nop	nop	nop
	nop	nop	nop	nop
	add $2,$2,$4	add $6,$6,$5	nop	nop
	bne $3,$0,lp	nop	nop	nop
	nop	nop	nop	nop
	add $2,$2,$6	nop	nop	nop
	nop	nop	nop	nop
	nop	nop	nop	nop
	nop	nop	sw $2,0x7fff($0)	nop
end:	beq $0,$0,end	nop	nop	nop
	nop	nop	nop	nop

では、アンローリングした結果をスケジュールしてみましょう。だいぶ、nopのスロットが減ったことがわかります。今は2つ分のアンロールを示しましたが、もっと多くのループをアンロール（展開）することで、依存性のない命令を増やすことができます。

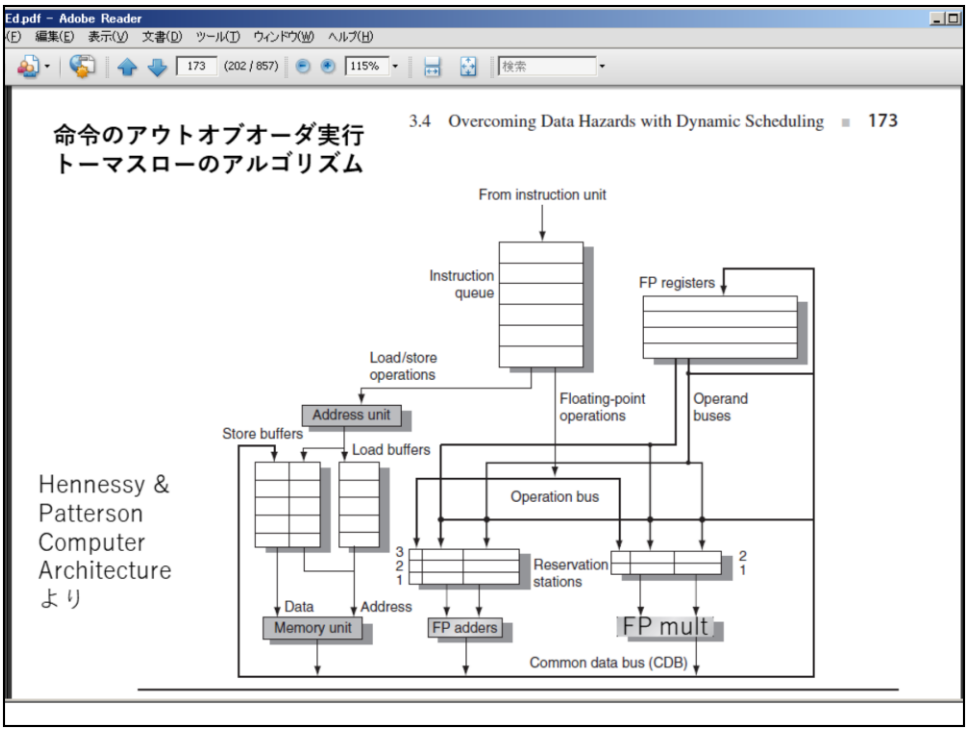


ループアンローリングは独立に実行できる命令が増えるため、スーパースカラ、VLIWの両方に効果があります。それだけではなく、カウンタやポインタの制御などスープ制御自体の損失を減らすことができます。しかし、ループ間に依存がある場合には適用が難しいですし、レジスタ数を多く必要となります。アンロール回数でループ回数が割り切れなかったり、そもそもループ回数が動的に決まる場合、面倒な前処理や後処理が必要となる問題もあります。最後にこのテクニックはコンパイルのし直しが必要なので、配布したコードをそのまま速くする、というビジネスモデルにはマッチしないです。とはいえ、ループ構造が単純な信号処理、画像処理などでは非常に有効なので、良く用いられます。

命令の動的スケジューリング Tomasuloのアルゴリズム

- 実行可能な命令が、依存性により実行できない命令を追い越す →Out of Order実行
- リザベーションステーションを使って逆依存、名前依存を自動的に解決
- 先に発行した命令を後で発行した命令が追い越して実行することができる
  - Out-of-order実行
- 命令管理表
  - Op: 実行する演算
  - Qj,Qk:ソースオペランドを作るリザベーションステーション、Vj,Vkに値が入っていれば不要
  - Vj,Vk:ソースオペランドの値
  - A:　メモリアドレス、ロード・ストア命令用
  - Busy：　利用中かどうか？
- レジスタ管理表
  - Qi:レジスタに結果をしまう演算を行うリザベーションステーションの番号
- パイプラインは、Issue, Execute, Write resultの3段
  - その前に命令はFetchされて命令キューに入っていると仮定

パイプラインが長くなって、依存性がある命令による損失が大きくなる場合、依存性のない命令が依存性により実行できない命令を追い越して先に実行することができれば性能向上を果たすことができます。このように実行順序を動的に変更して動かす実行方式のことをOut-of-order実行と呼び、与えられた順番で命令を実行するin-order実行と区別します。Out of Order実行を行うための代表的な方法がTomasuloのアルゴリズムです。このアルゴリズムはリザベーションステーションを使って独立に依存性の判断を行うことで、様々な命令間の依存を解決します。



ヘネパタのテキストに載っているTomasuloのアルゴリズムの図です。浮動小数点加減算器(FP adder)、浮動小数点乗除算器(FP mult)の前にあるのがリザベーションステーションです。命令キューの中の命令は、リザベーションステーションの中に蓄えられ、CDB(共通データバス)を見て、自分の使うデータが送られていた場合にそれを自分のフィールドに取り込みます。両方のフィールドに有効な値がセットされた時に演算は、実行可能になり、リザベーションステーションの内容は演算器に送られ、演算が実行され、答はCDBに送られます。

Tomasuloのアルゴリズム

- スーパースカラと組み合わせると命令数が増えて有効
- 分岐が成立するかどうかがはっきりするまで次の命令の発行ができない

→ 分岐予測

→ 投機的実行

Tomasuloのアルゴリズムは、スーパースカラ方式と組み合わせると、複数の同時に発行される命令をリザベーションステーションに割り当てるように拡張すれば、効率が改善されます。したがって両者の組み合わせは高速プロセッサの定番となっています。一方、Tomasuloのアルゴリズムには限界もあります。実際のプログラムでは分岐命令があってこれが成立するかどうか分からないため、発行できる命令の数が制限されることです。そこで2つの方法が研究されました。一つは分岐するかどうかをできるだけ正確に分岐する分岐予測であり、もう一つはこの予測に従ってどんどん命令を発行してしまい、予測がはずれたらやり直す投機的実行（Speculative Execution)です。

投機的実行のための
改造
→リオーダバッファを
付加し、結果を一時格納

コミットされたときに
レジスタに書き込み

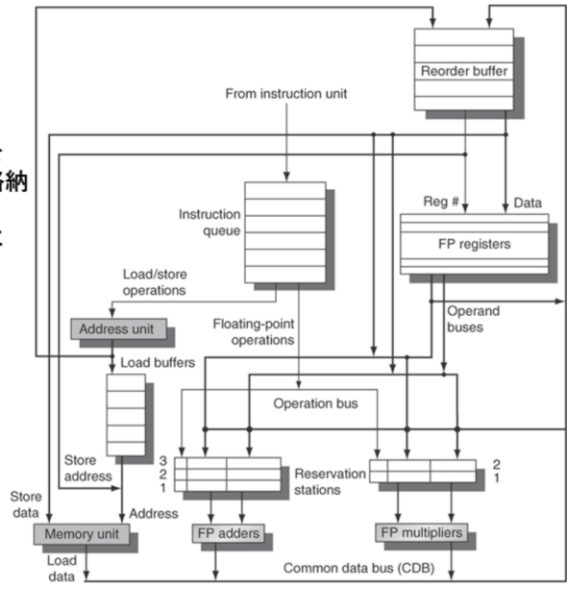


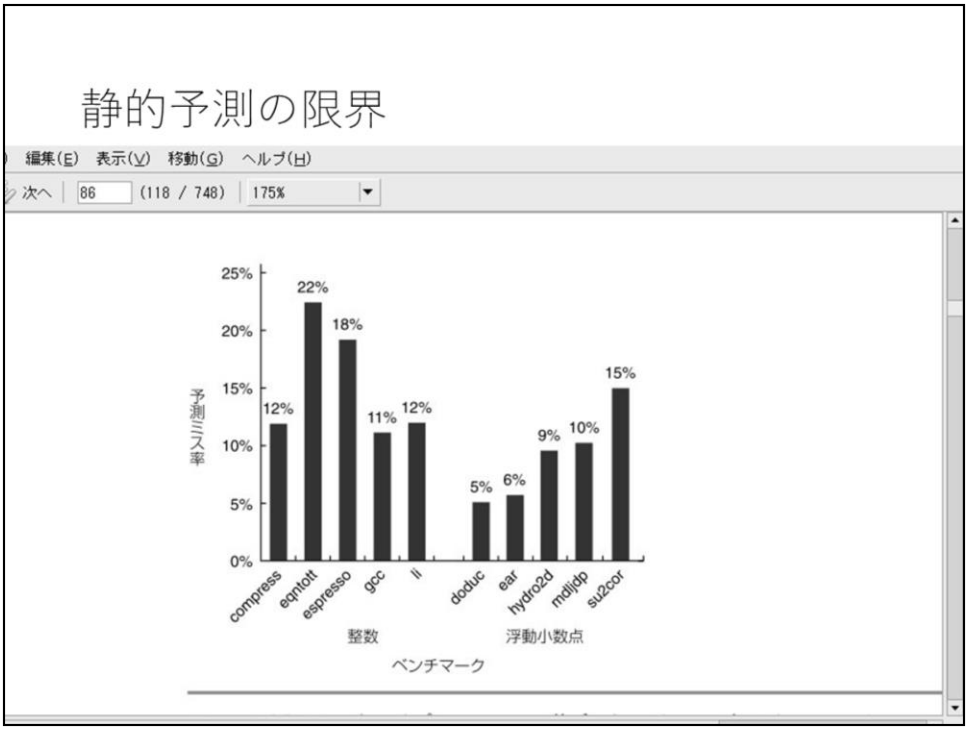
Figure 3.11 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.6 on page 173, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

投機的実行をサポートするために拡張したTomasuloのアルゴリズムの図です。実行結果はレジスタに格納される前にリオーダーバッファという所（右上）に一時的にしまわれ、分岐予測の結果が確定すると、レジスタに格納されます。分岐予測がはずれると、その内容はクリアされ、その時点から処理のやり直しになります。この仕組みは実は以前紹介した例外処理の対策にもなることから多くのプロセッサで用いられています。

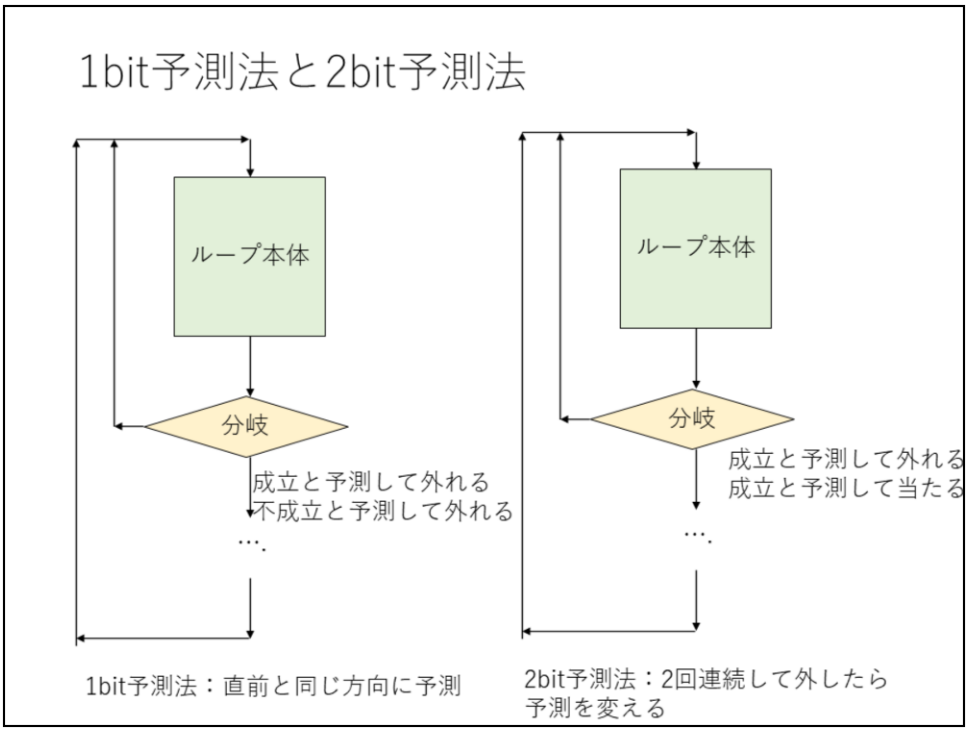
分岐予測

- 一番簡単な予測
  - 全ての分岐を飛ぶと予測すること
    →6-7割位当たる
- 静的予測
  - コンパイラによって行う分岐予測
  - プロファイル（仮に実行して様子を見る）を用いると一定の精度が得られるが限界がある
- 動的予測
  - ローカルなもの（その分岐の過去の結果を使う）
  - グローバルなもの（ある分岐の前後の分岐を見る）
  - トーナメント法：両方考えて当たる方を使う

投機的実行を行う場合、分岐予測がはずれると処理のやり直しになり、ダメージが大きいです。したがって正確な予測を行うことは重要です。最も簡単な予測は全てを「飛ぶ」方に賭けるもので、これでも6-7割当たります。しかし、分岐予測は95%以上は当てたいのでこれでは全然ダメです。分岐予測にはコンパイラによって行う静的なものと動的なものがあります。静的分岐は、プロファイルなどで精度を高めることはできますが、一定の限界があり、動的予測の初期値に使われます。動的予測にはローカルなものグローバルなもの、両者を合わせたものがあります。

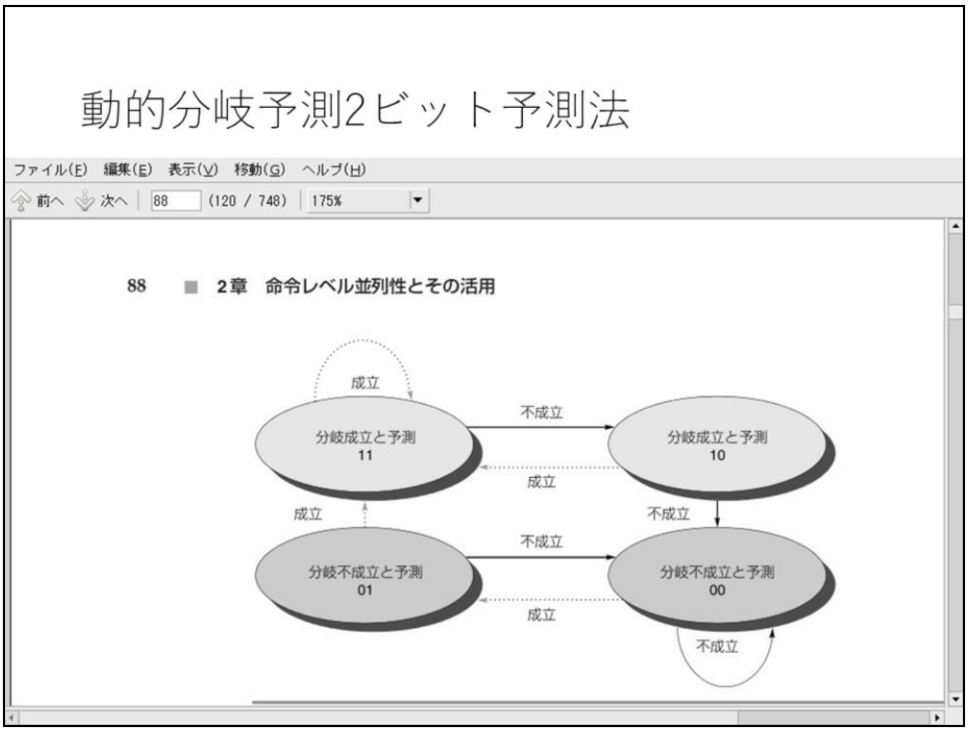


この図は静的予測の精度を示しています。この程度が限界です。



最も簡単な動的分岐予測法は1ビット予測法です。これは同じ分岐に関して、それが直前に成立したら、成立と予測し、不成立ならば不成立と予測する方法です。しかし、この方法は、2回続けて外す傾向があります。図のような2重ループを考えます。内側のループを回っている間、成立と予測して当たり続けますが、ループを飛び出す時は不成立なので、外れることになります。再び内側のループに飛び込んできた際に不成立と予測すると、こんどはループを回るので、再び外してしまいます。2bit予測法はこの改良版で、2bit持たせておいて、2回続けて外した時に予測を変えます。これだとループから飛び出す時は外れますが、次に飛び込んだ時には外れません。

動的分岐予測2ビット予測法



2bit予測法は図のような状態遷移で実現します。2回連続して予測をはずすと、違った予測に切り替えます。3bit用いることで、3回連続してはずすと、違った予測に切り替えるように拡張することができます。より一般的にn-bitをカウンタとして用いて、一定の閾値で判断を切り替えるように拡張することもできます。これをn-bit予測法と呼びます。

予測と共に飛び先も入れておく

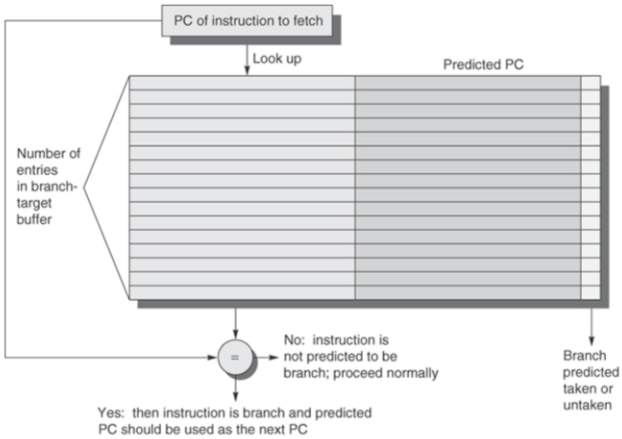
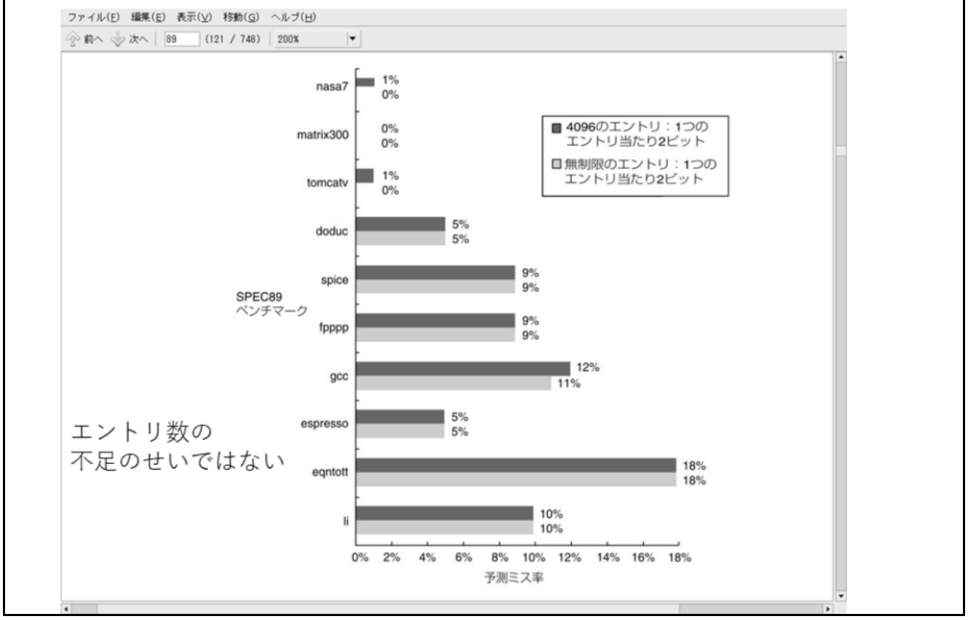


Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

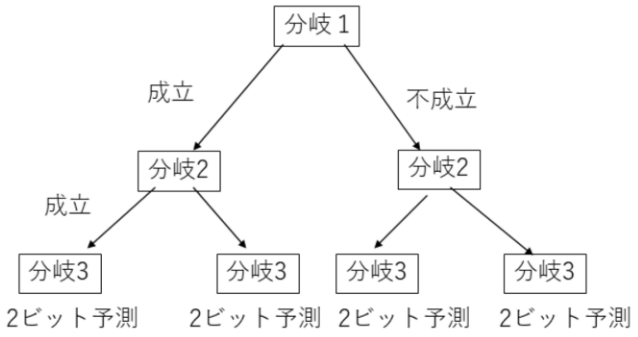
2bit予測法を使うためには、分岐命令を識別して以前の結果を保存しておく必要があります。これは、分岐予測バッファと呼ぶキャッシュに似た機構で実現します。分岐命令の置かれているアドレスの一部をインデックスとして検索します。この図では分岐予測の予測bitに付け加えて、成立した場合の飛び先アドレスも記憶しています。一度計算したら変わらないのでその結果を再利用している点が賢いです。

2ビット予測法のミス率



この図は2bit予測法のミス率、つまり外れた割合を示します。分岐予測バッファの容量が足りないと、分岐命令が競合してしまい間違った分岐の結果で予測をする場合があります。しかし、容量が無限に大きくしてもミス率が減らないことから、この影響はわずかであることがわかります。

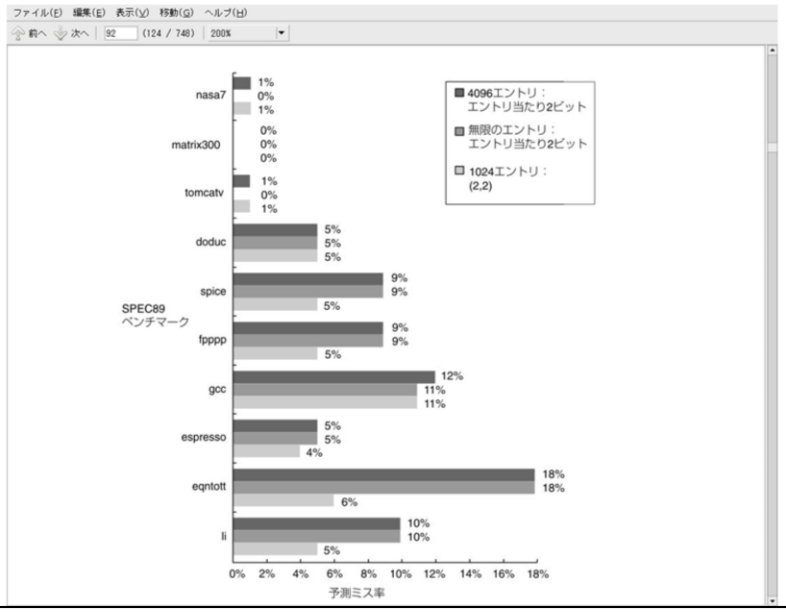
相関分岐予測



3-2予測法　　一般的にm-n予測法がある

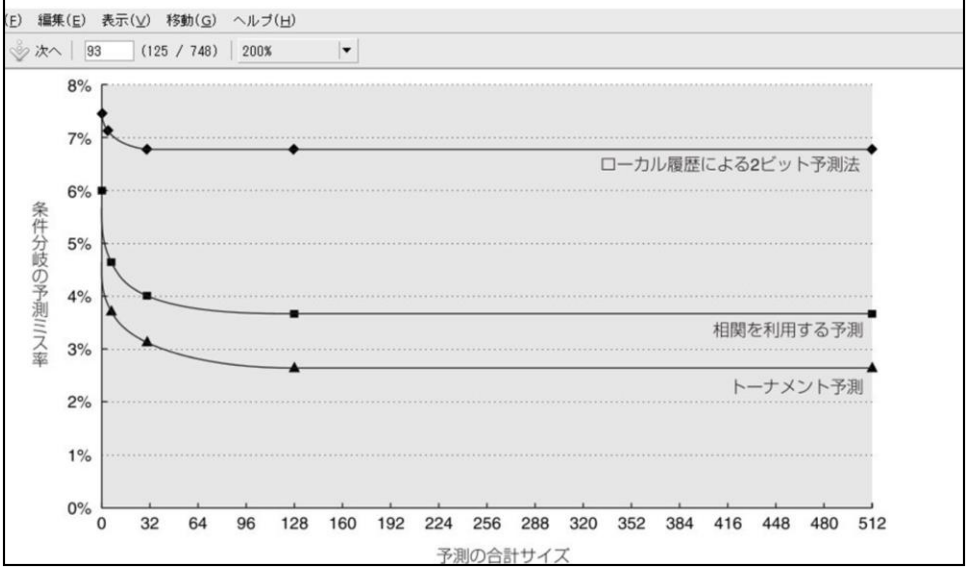
ではどうすれば予測正答率を上げることができるでしょう。分岐が単純なループ構造でない場合、成立か不成立かは、その直前の別の分岐が成立したかどうかに影響を受けます。このため、ある分岐が成立したかどうかをその直前、二つ前の分岐の成立、不成立別に記憶しておけば、その影響を考えた予測ができます。これが相関分岐予測で、2つ前までの相関を調べるには4つのケースについて別々に記憶する必要があります。そのそれぞれにn-bit予測法を使うことができるので、いくつ前の分岐まで調べるか（m)、それぞれ何ビット使うか（n）で、ｍ－ｎ予測法として一般化することができます。

相関分岐法の効果



相関分岐法によりどの程度分岐予測ミスが改善されるかを示しています。プログラムによってはかなり効果があることがわかります。

トーナメント法

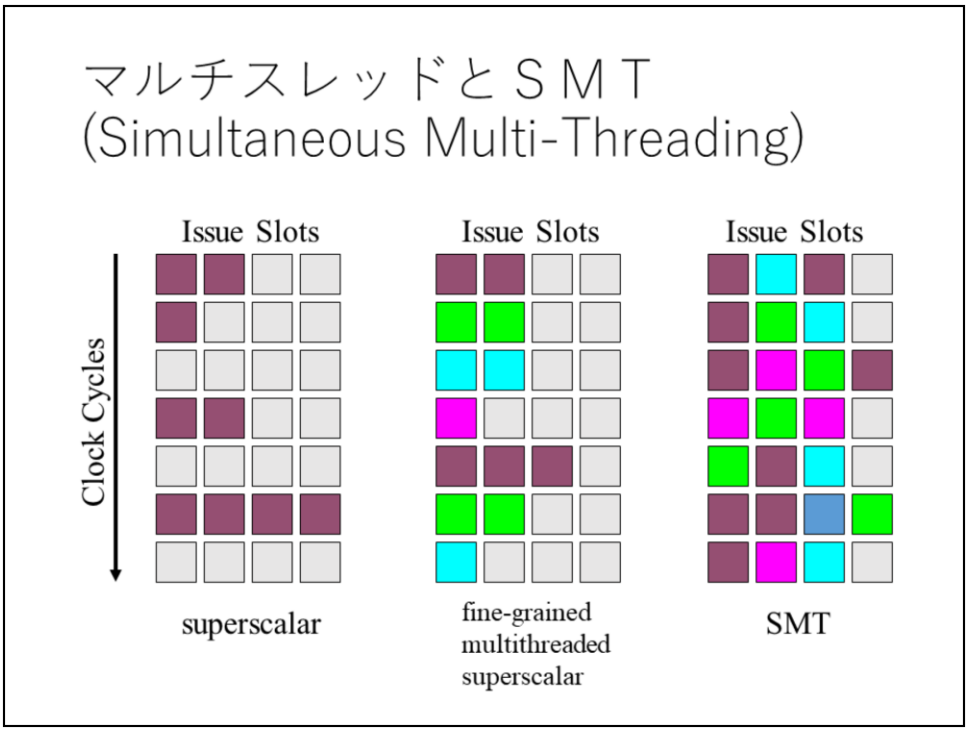


自分の過去の履歴が影響を及ぼすか、相関を利用するとより効果が高いかは、分岐の使われ方によって決まっています。そこで、どちらの予測を使うかを分岐毎に判断する方が、m-n分岐のmとnを増やすよりも効果がありそうです。これがトーナメント分岐で、どちらの分岐が当たるかを調べて、当たる方を利用します。現在の高性能CPUで最もよく使われている方法です。予測の合計が64を越えれば、その予測ミス率は3%を下回ります。

分岐予測　まとめ

- コンパイラによる静的予測は動的分岐予測の初期値として使われる
- 動的分岐予測は現在の高速プロセッサではトーナメント方式が良く用いられる
- まだ新しい方法が提案されたりするが、研究としてはだいたい終わっている

分岐予測をまとめます。まだ分岐予測の研究が終わったわけではないですが、方法としてはほぼ出尽くしています。ニューラルネットワークを使って学習させて、予測率を上げたりする試みもあるのですが、下手をすると本体のCPUよりもハードウェアが大きくなってしまいます。



スーパースカラ型のプロセッサの資源の利用率の低さを改善するために使われるもう一つの方法は空いている時間帯に別のスロットを動かすことです。この場合、スレッドとは別のPCを持つ別のプログラムなので、独立に実行することで資源を効率的に利用することができます。1クロック（あるいは数クロック）で高速でスレッドを切り替える方法を細粒度（fine-grained)マルチスレッディングと呼びます。真ん中の図は、毎クロック4つのスレッドを順に切り替える方法を示します。自分の番が回ってくる頃には依存関係のある先行命令の実行は終わっているため、利用率は上がります。しかし、この方法では自分の順番が4回に1回しか回ってこないため、一つのスレッドの実行時間は伸びてしまいます。そこで、順番を気にせずにどのタイミングでもそれぞれのスレッドを動作できるようにしたのがSMT（同時マルチスレッディング）です。この方法では資源の利用率は向上し、一つのスレッドの実行時間もさほど大きくなりません。しかし、制御が複雑で、キャッシュのヒット率が落ちるなどの問題点があります。この方式をIntelはハイパースレッディングと呼び、2程度の小規模なものを各世代のCPUで使っています。

単一CPUの高速化手法のまとめ

- 現在の高速プロセッサはパイプラインのステージ数は10程度が多く、ほぼ限界に達している。
- 現在の高速プロセッサは命令の動的スケジューリングを複数命令同時発行と組み合わせて用いている
- 投機実行は、例外処理にも利用できるので、多くのプロセッサが装備している
- 分岐予測は上記の高速化を支える上で重要であり、現在、ほぼ手法が出尽くしている。

今までに紹介した単一CPUの高速化手法をまとめてみましょう。これらはマルチコア化をされたプロセッサにおいても、それぞれのコアで使われています。

マルチコア時代へ

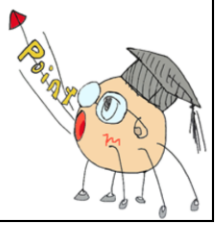
- クロック周波数向上の限界
  - 消費電力
  - 放熱
- ILP（命令レベル並列性）の限界
- メモリの壁

→コンピュータアーキテクチャAの内容へ

さて、このような単一CPUの高性能化技術も2003年ごろに、ほぼ行き詰りました。それからはマルチコアの時代になります。これはコンピュータアーキテクチャAの領域です。

本日のまとめ

- 複数命令の同時発行
  →ループアンローリングを行って並列性を高めると有効
  - スーパースカラ
    - In-orderなもの
    - Out-of-orderなもの
  - VLIW
- マルチスレッディング
- SIMD命令とアクセラレータ
- マルチコア、メニーコア



本日のまとめです。紹介したいくつかの方法はコンテストにも利用可能です。

演習7

mipsess, mipsevlをそれぞれsiriusにて
dc_shell-t –f mipse.tcl | tee mipse.rpt
で合成し、addarray.asmの実行時間、動作周波数、面積、電力の表を作成せよ。

では、今日の演習です。簡単な問題なんでさっとやって終わりにしましょう。