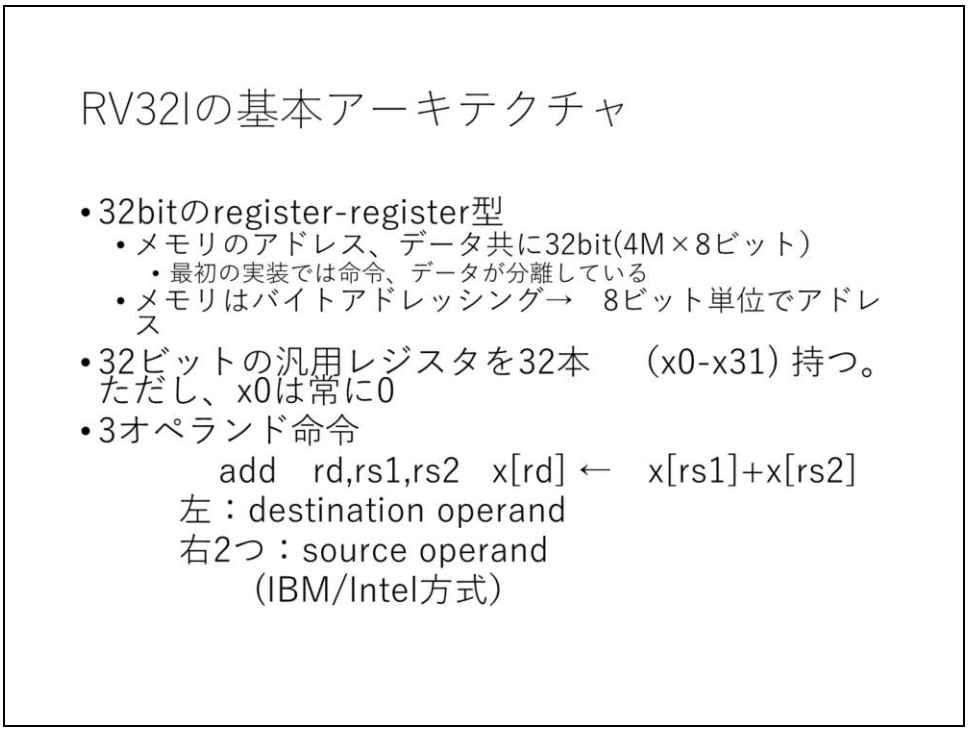
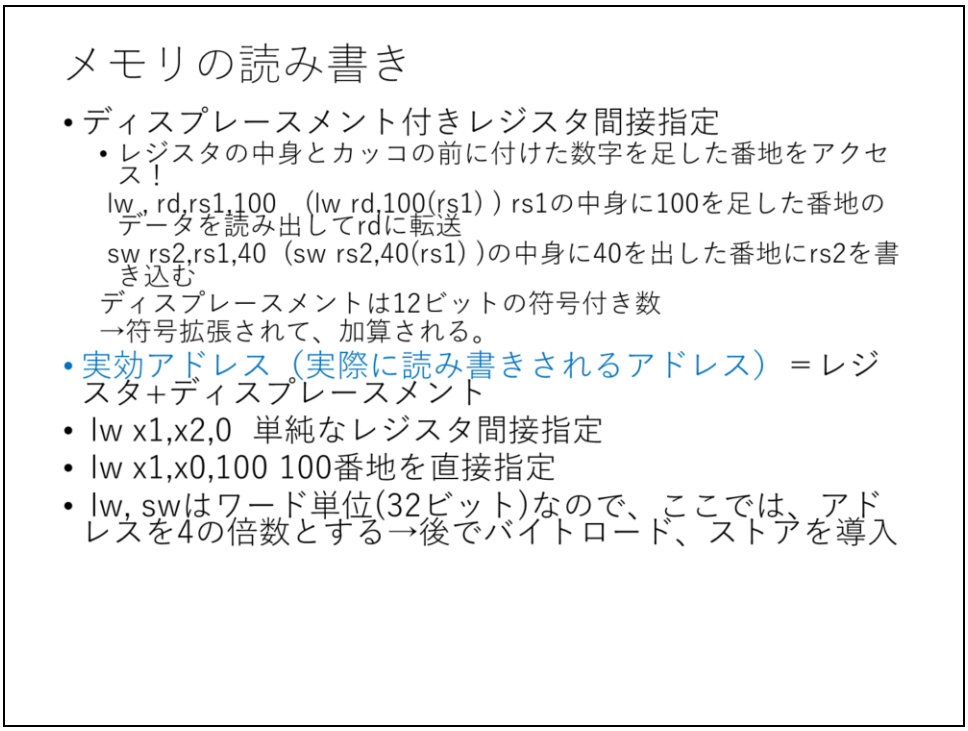
32ビットRISCと高速化技法

天野　hunga@am.ics.keio.ac.jp

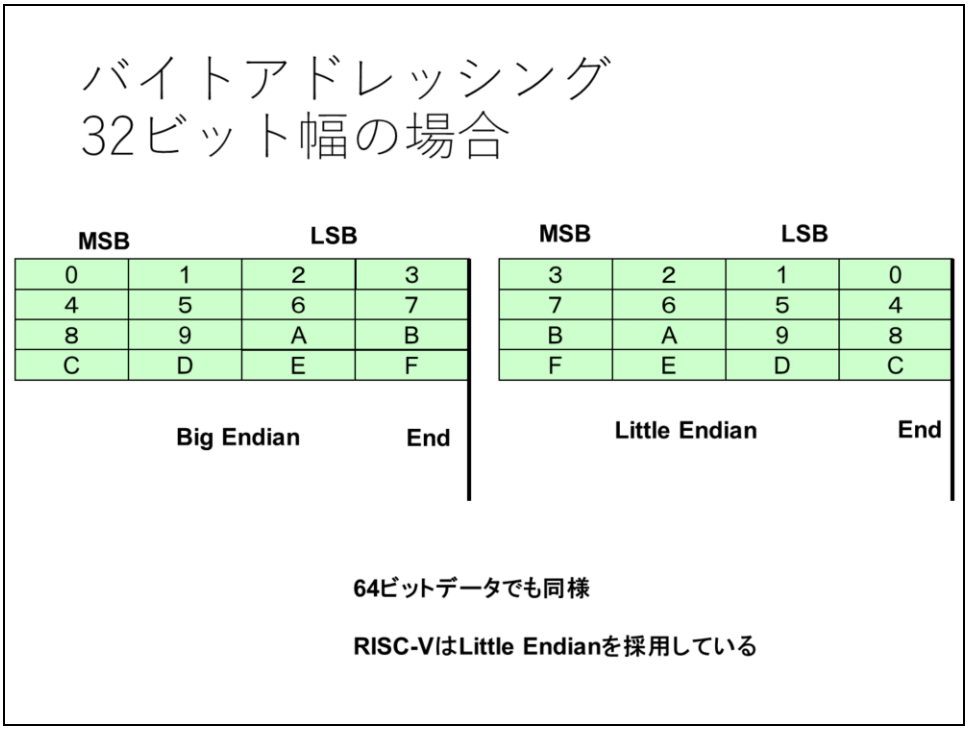
最後にCPUのさらなる高速化の話を紹介します。今までは16ビットRISCを紹介してきましたが、ここで32ビットRISCの代表としてRISC-Vを紹介します。



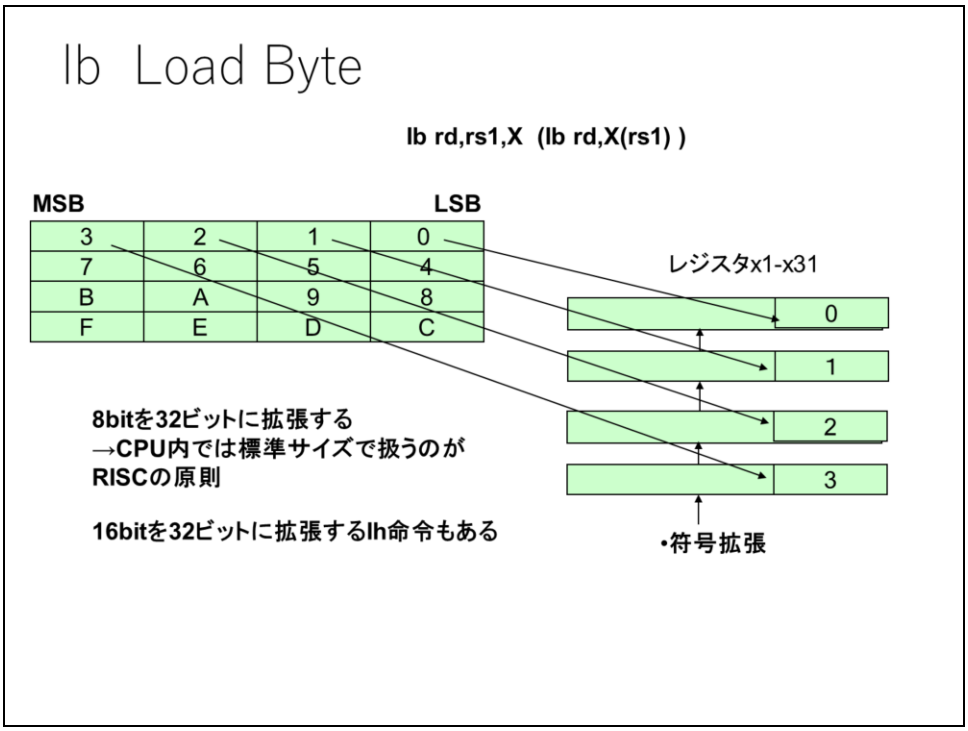
RV32Iは32ビットのレジスターレジスタ型のアーキテクチャです。命令メモリ、データメモリ共に、アドレス、データは32ビットです。メモリの番地付は8ビット単位であり、32ビットは4つ分番地を占有します。これをバイトアドレッシングと呼び、標準的な方法です。レジスタも32ビットで、32本持っています。ここではこれをx0..x31という形で表します。ここで、x0は常に0であり、このレジスタへの書き込みは意味を持ちません。この設定はちょっと奇妙な気がしますが、大変役に立つのでほとんどすべてのRISCで使われています。RISC-Vの命令は3オペランドを持ちます。add rd,rs1,rs2と書くとx[rs1]+x[rs2]の答がx[rd]に入ります。ディスティネーションレジストとソースレジスタは完全に分離することができます。もちろんこれらに同じレジスタ番号を指定してもいいです。ディスティネーションレジスタを左に書くIBM/Intel方式を取ります。



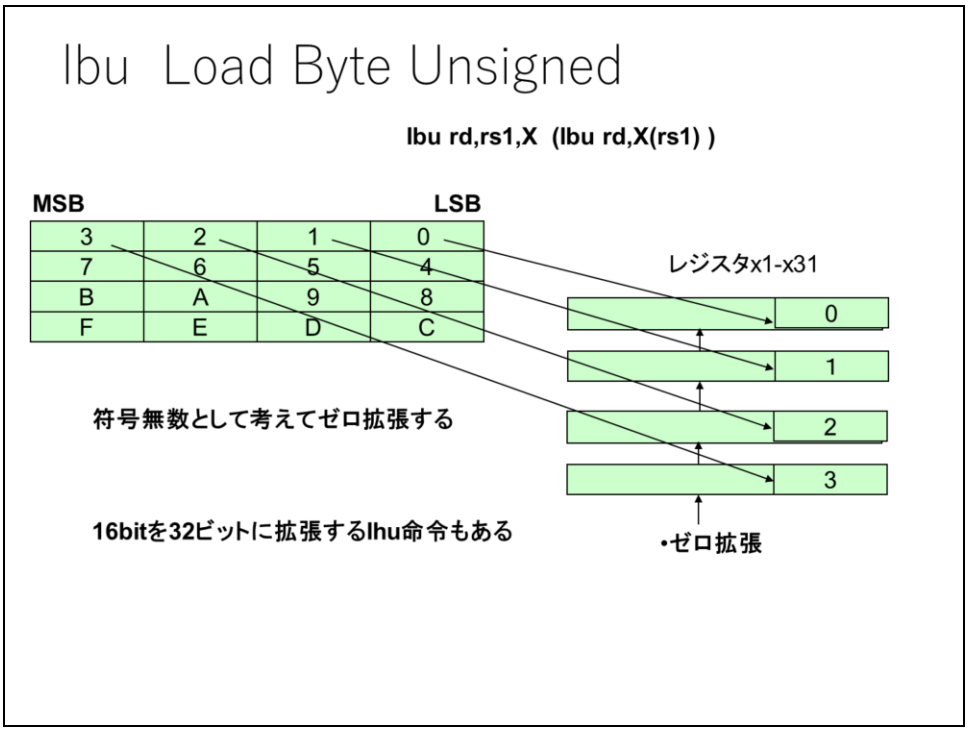
メモリの読み書きは、ディスプレースメント付きレジスタ間接指定を使って読み書きする番地を指定します。この方法は、lw x1,x2,100のように記述し、カッコの前に書いた数字（ディスプレースメントあるいはオフセット）にレジスタx2の中身が足された番地が実効アドレスになります。例えばx2に50が入っていたとすると、150番地が読み出されてx1に入ります。lwはload wordで32ビットのデータを読み出します。後で詳しく説明しますが、メモリの番地は8ビット単位についているので、実効アドレスは4の倍数でなければならないです。ディスプレースメントは、12ビットの符号付き数で、レジスタ中の32ビットの数に対して符号拡張されてから加算されます。したがって、マイナスの数を書くこともできます。sw(store word)はこの逆で、sw x1,x2,40 (sw x1,40(x2))と書けば、レジスタx1の内容（32ビットのデータ）を、レジスタx2の中身に40足した番地に書き込みます。この方式は、ディスプレースメントに0を指定すれば、単純なレジスタ間接指定となり、レジスタをx0にすれば、ディスプレースメントで示した番地がそのまま指定できる直接指定方式になります。



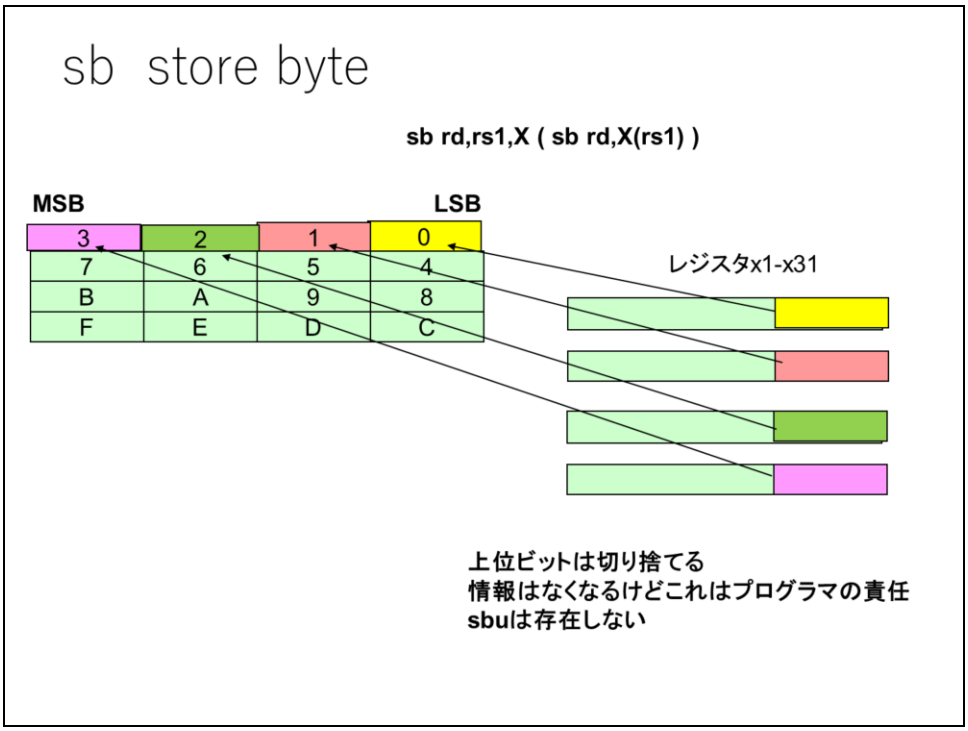
I／Oデータなどを扱う上での無駄を防ぐために、最近のCPUは32ビットではなく8ビット単位で番地が振られています。すなわち、32ビットは番地4つ分に当たります。ここで、桁の大きい方から0, 1と振っていく方法と小さい方から0, 1と振っていく方法の二つが考えられます。前者をBig Endian,後者をLittle Endianと呼びます。ここでは以降、Big Endianで番号を振ることにします。この振り方は統一が取れておらず、コンピュータ間でデータを交換する際にトラブルの元となります。最近のCPUは電源投入時の指定でどちらの方法を取ることもできるものが多いです。MSB側から0を振っていくと、大きい方で端（LSB: end)に達することからBig Endian, LSB側から0を振っていくと小さい方で端に達することからLittle Endianという名前になっていることが分かります。Endianとは奇妙な英語ですが、これはこの言葉が、ガリバー旅行記の卵の細い方から割る派(Little Endian)と太い方から割る派(Big Endian)で内乱が起きる話に由来するためです。



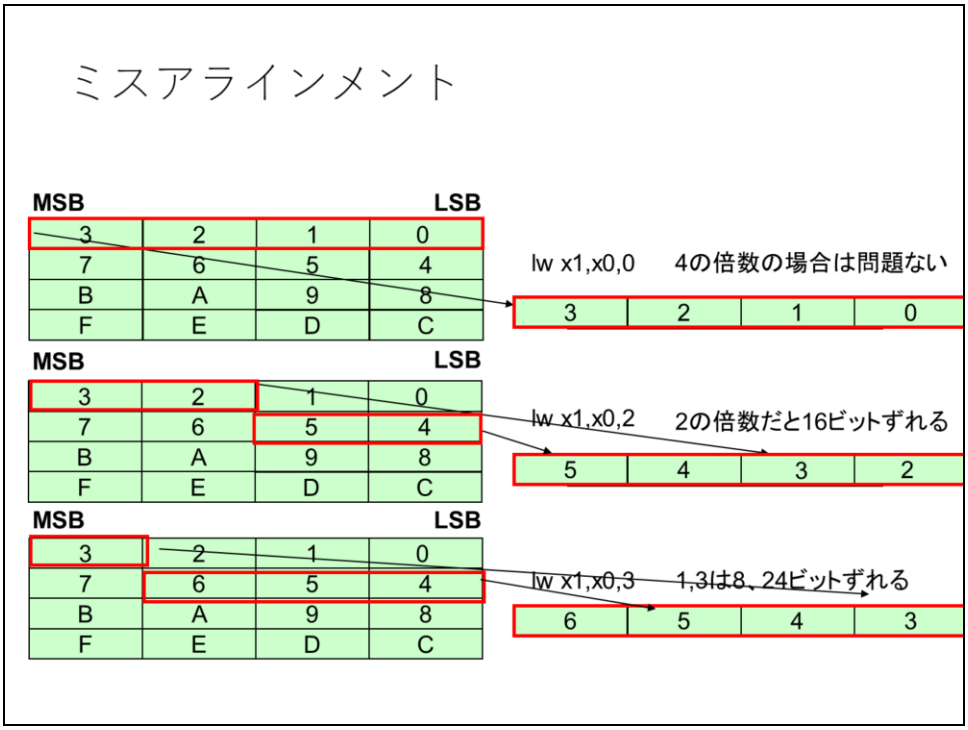
では、バイトデータを扱う命令を定義しましょう。lbは指定されたアドレスの8ビットを読み出して、レジスタの下位8bitに置きます。上位24bitは符号拡張されます。RISCではCPUの内部ではデータのサイズを統一してしまうのが普通です。この場合も読み出す際に32ビットに拡張します。lbはlwと同じR型で定義します。ちなみにRV32Iでは16ビットデータを読み出して32ビットに拡張するlh(Load Halfword)もあります。最近の文字コードは16ビットが多いので、案外16ビットデータは利用価値が高いためです。ただし、ここでは実装が面倒なので省略してあります。



I／Oは符号無しのデータを扱うことも多いので、ゼロ拡張の8bit読み出し命令も用意しておくのが普通です。これがlbu(Load Byte Unsigned)で、上位24ビットには0が入ります。RV32Iには、16ビットのLoad命令lhu(load half word unsigned)もあります。



逆にレジスタ中の値を8ビット単位でメモリに書き込む命令がsb(store Byte)です。この命令はレジスタの下位8ビットを指定されたメモリの番地に書き込みます。上位24ビットは無視されます。上位の情報がなくなっても困らないようにするのはプログラマの責任です。ちなみにsbはデータのサイズが減る方向なので、sbuとかを作る必要はありません。16bit用のstore halfwordは存在します。

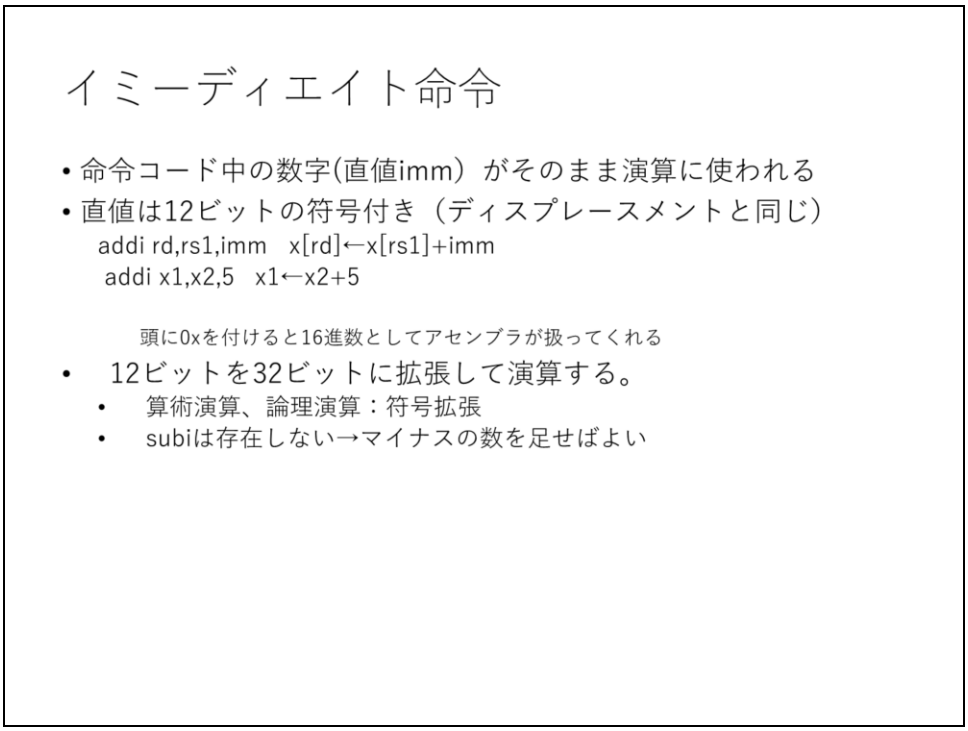


バイトアドレッシングのメモリをlw,sw命令で扱った場合、4の倍数の番地から読めば問題はありません。メモリ中のバイトの順番でそのままデータがレジスタに読み込まれます。しかし奇数番地から読んだらどうなるでしょう。この例では1番地の8ビットを上位にもってきて2番地の8ビットを下位にもってきてくっつける必要があります。このように16ビット、32ビット、64ビットのデータがバイトアドレッシングの境界にうまく整列していない問題をミスアラインメントと呼びます。

レジスタ間演算命令

- レジスタ同士でしか演算はできない
  - add rd,rs1,rs2　　x[rd←x[rs1]+x[rs2]
  - sub rd,rs1,rs2　　x[rd]←x[rs1]-x[rs2]
  - and rd,rs1,rs2　　x[rd]←x[rs1]&x[rs2]
  - or rd,rs1,rs2　　x[rd]←x[rs1]| x[rs2]
  - xor rd,rs1,rs2　　x[rd] ← x[rs1] ^x[rs2]
- 　加減算、加減算、シフト、セット演算

RISC-Vは、レジスターレジスタアーキテクチャなので、演算にはメモリを指定することができず、一度レジスタに持ってくる必要があります。3オペランド方式なので、レジスタの書きつぶしを心配しなくて良くて便利です。



イミーディエイト命令では、命令コード中の数字（直値:imm)がそのまま演算に使われます。RISC-Vの場合、直値は12ビットで、算術演算命令、論理演算命令の両方で符号拡張されます。すなわちaddi rd,rs1,immを実行すると、immが32ビットに符号拡張され、32ビットのレジスタと加算されます。

比較命令slt, slti, sltu, sltiu (set less than)

- slt rd,rs1,rs2　if(x[rs1]<x[rs1]) x[rd] ←1 else x[rd] ←0
- slti rd,rs1,imm if(x[rs1]<imm) x[rd] ←1 else x[rd] ←0
- unsignedはレジスタの値が符号無として、比較する。

大小比較を行う命令として、RV32Iではslt (set less than)とslti(set less than immediate)と呼ぶ比較命令を使います。この命令は2つのレジスタ、あるいはレジスタとイミーディエイトを比較して、その結果をレジスタに格納します。slt rd,rs1,rs2を実行するとx[rs]<x[rt]の場合は、x[rd]に1を、そうでなければ0をセットします。slti rd,rs1,immはx[rs1]<immの時はx[rd]に1を、そうでなければ0をセットします。RV32Iは分岐命令に大小判定機能があるので、この命令は、主として複数のワードに跨った数を演算する時、桁上げを移動したりする場合に使われます。unsigned命令は、レジスタの中身が符号無と考えて比較します。

RV32Iの条件分岐命令

PC相対指定

target ← （符号拡張）offset
beq rs1,rs2,offset： if(x[rs1]==x[rs2])
PC←PC+target
PCは命令コードの置かれたアドレス＋４
命令コード中には、offsetの0ビット目は含まれていない
bne rs1,rs2,offset if(x[rs1]≠x[rs2])
PC←PC+target

0と比較する場合はx0を使えば良い

RV32Iの条件分岐命令は、プログラムカウンタ相対指定で飛び先を指定します。飛ぶ番地の起点は、分岐命令の置かれた番地＋4で、飛び先は、飛び越す命令の数(offset)で表します。RISC-Vの場合、それぞれの命令は2バイト（RV32C）か4バイトなので、飛び先の番地は偶数しかありえないです。このためoffsetでは最下位ビットは省略して12ビット分を持たせています。飛び先を求めるには、0を補って、符号拡張して足してやります。

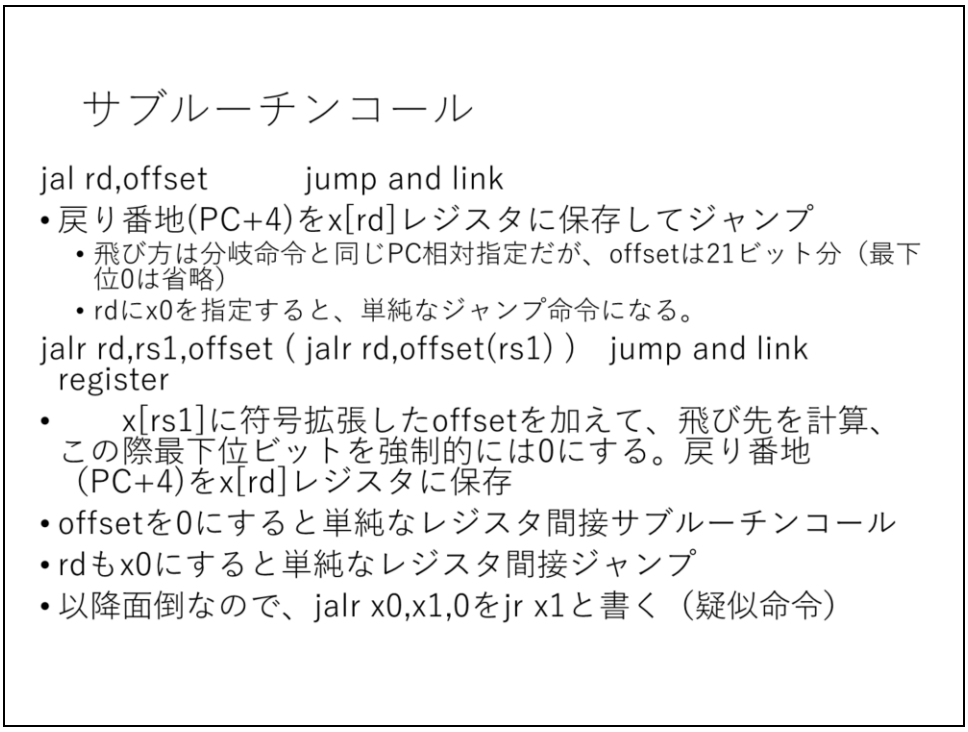
飛ぶかどうかの判断は、2つのレジスタを指定して、それが等しい時に分岐するbeq（branch equal)と等しくない時に分岐するbne(branch not equal)の二つが用意されています。0かどうかを判断に使いたいときは、一つのレジスタをx0にすればOKです。

大小比較を含んだ分岐

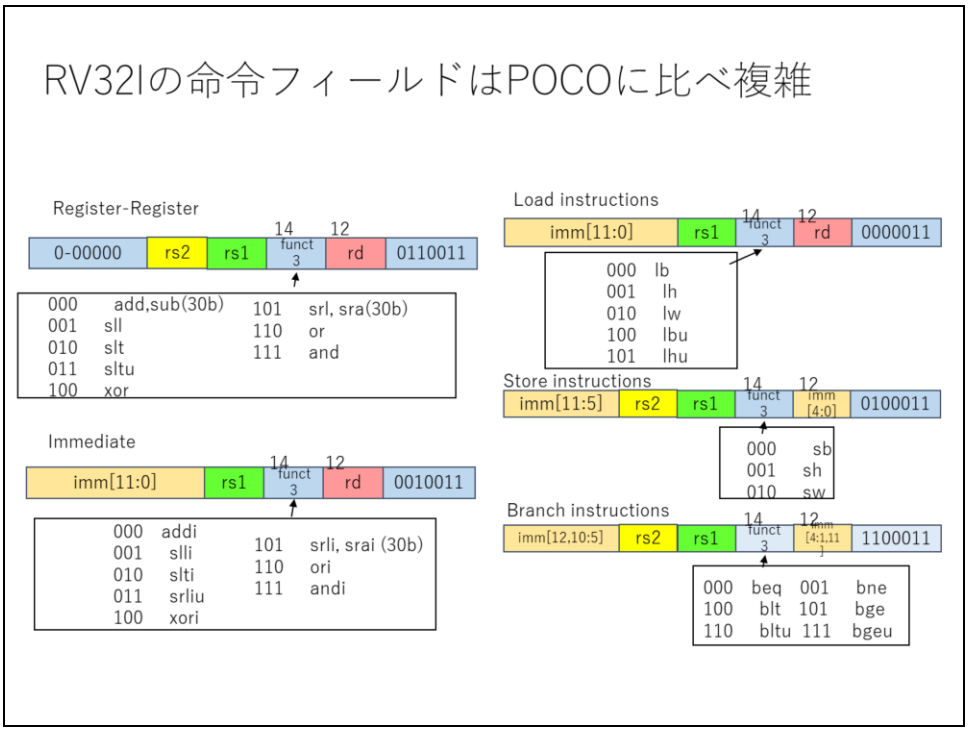
branch less than
blt rs1,rs2,offset： if(x[rs1]<x[rs2])
PC←PC+target
branch greater equal
bge rs1,rs2,offset if(x[rs1]>=x[rs2])
PC←PC+target
bltu, bgeuはunsighned命令で、レジスタの内容を符号無数と考えて比較

b le, bgtなどは存在しないが、レジスタの順番を入れ替えて実現できる→疑似命令

RV32Iはレジスタの大小を比較して分岐するblt (branch less than)とbge(branch greater equal)を持っています。飛び方はbeq, bneと同じです。レジスタの中身を符号無と考えて大小比較を行うunsigned命令bltu, blgeuも用意されています。ble, bgtは存在しませんが、レジスタの順番を入れ替えて実現でき、疑似命令としては用意されています。



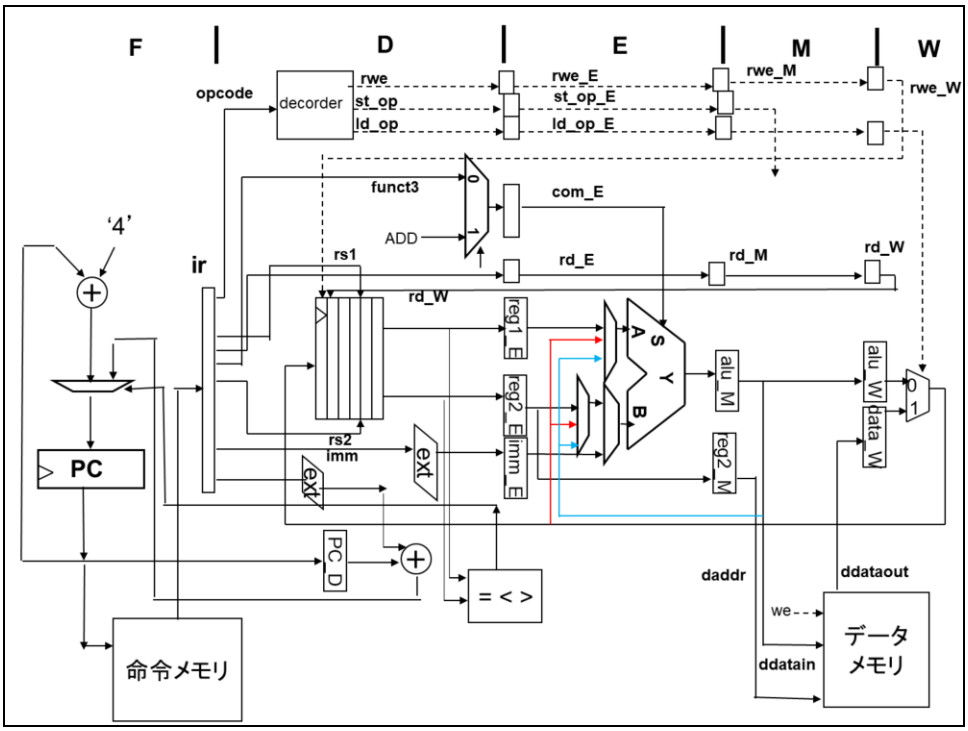
jal(jump and link)はサブルーチンコールで、戻り番地（PC+4)をrdに入れます。飛び方は分岐命令と同じPC相対指定ですが、遠くに飛べるようにoffsetは21ビット分あり最下位の0は命令中では省略されています。戻り番地を保存する必要のない無条件ジャンプにするためにはrdにx0を指定します。beq x0,x0,offsetでも無条件ジャンプになりますが、こちらの方が遠くに飛べます。jalr(jump and link register)は、x[rs1]に12ビットのoffsetを加算したものが飛び先になります。この時、最下位ビットが1になると困るので、強制的に0にします。同時にx[rd]に戻り番地を格納します。この命令は、offsetを0にすると単純なレジスタ間接サブルーチンコールになり、rdをx0にすると単純なレジスタ間接ジャンプになります。以降、面倒なので、jalr x0,x1,0をjr x0（jump register)と書きます。このように、実際は他の命令に置き換えて実行される命令を、あたかも存在するかのように扱う方法を疑似命令と呼びます。疑似命令はアセンブラで置き換えられます。



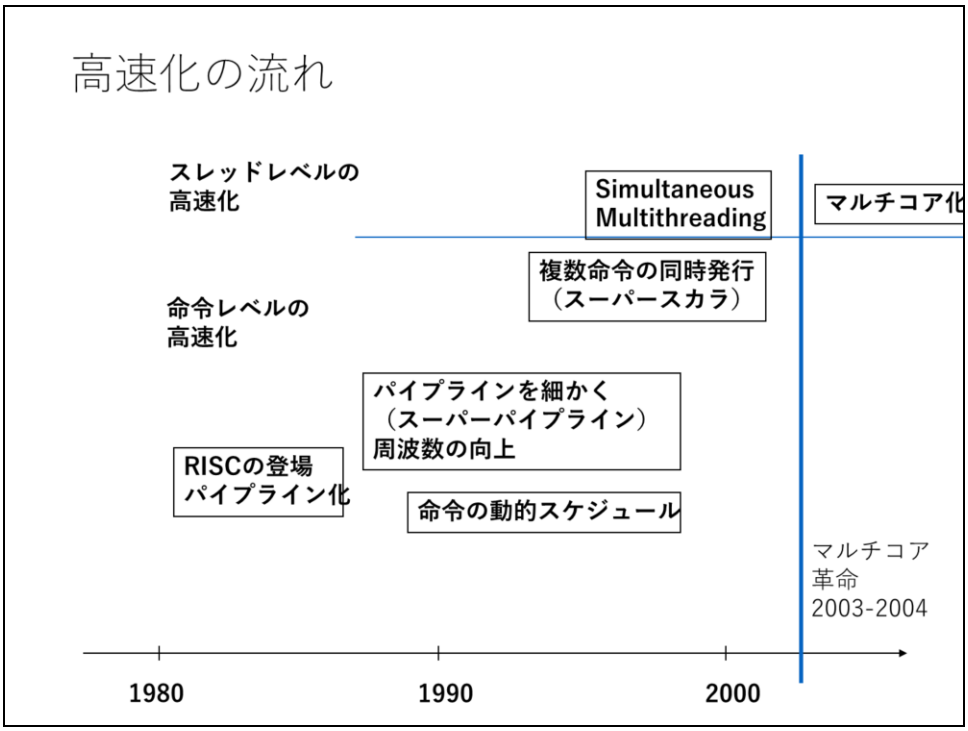
RV32Iの命令フィールドは、POCOに比べて複雑です。ここにはメジャーな5種類を示しますが、これに加えてあまり使わないのが1つあり、全部で6種類あります。しかし、それぞれのフィールドは揃っており、規則性は高いです。

iverilogによるシミュレーション

- 演習資料を取ってくる
  - wget http://www.am.ics.keio.ac.jp/chou_aki/base.tar
  - tar xvf base.tar
  - cd base
- プログラムのアセンブルshapaはRISC-V用にも使えるが、Pythonのプログラムで後処理をしている。
- mult.asmをアセンブルするには？
  - make mult → imem.datが生成される
- シミュレーションの実行は？
  - make
  - ./test
  - 掛け算のプログラムをやってみよう
  - POCOに比べてdisplacementがあるので楽だ



パイプラインは、F、D,E,M、Wの5段になります。ディスプレースメントをEで計算し、Mでメモリアクセスをする構成です。このため、メモリからロードする値に対するフォワーディングは間に合わず、1クロックサイクルストールさせます。同様に、直前の演算の結果を反映して分岐するとクリティカルパスが長くなりすぎるので、1クロックストールさせます。

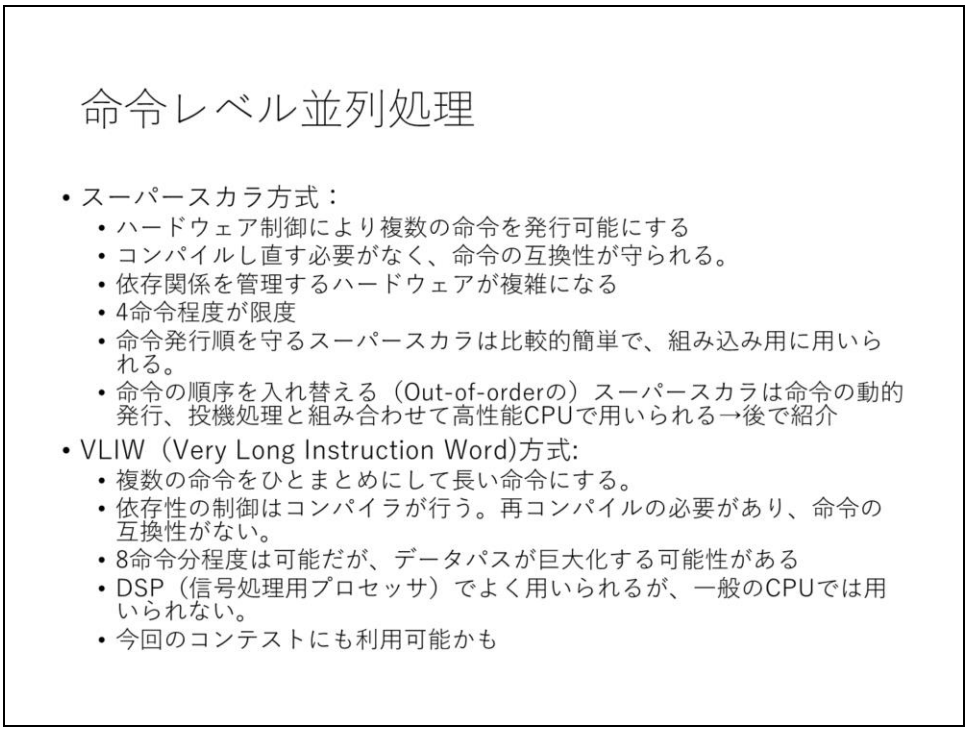


今まで紹介したパイプライン化は1980年代のCPUの性能向上を後押ししました。しかし、90年代になってこれが限界に達すると、様々な命令レベル高速化技術が発達しました。ポイントは、今のバイナリをそのまま高速化することにありました。

CPUの高速化手法

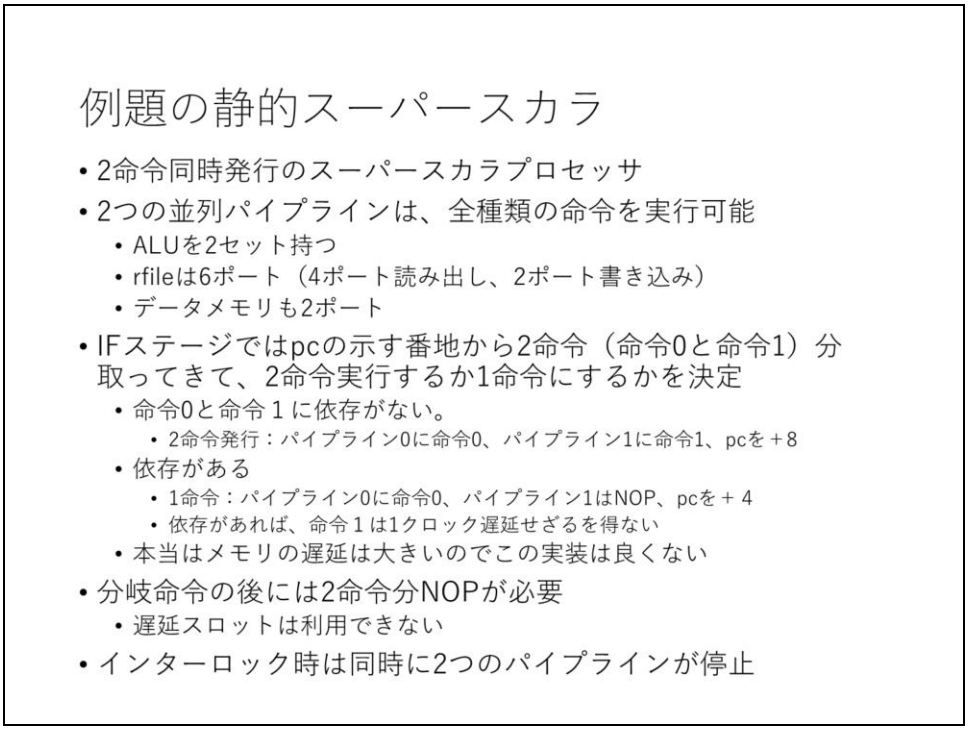
- パイプラインの段数を増やしていく
  →動作周波数を上げる
  スーパーパイプラインと呼ぶ場合もある
  メモリの遅延を回避する
- 複数命令の同時発行
  - スーパースカラ
  - VLIW
- 命令の静的スケジューリング
  - ループアンローリング
- 後は雰囲気のみ
  - 命令の動的スケジューリング
  - 投機的実行（Speculative Execution）
  - 分岐予測

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。

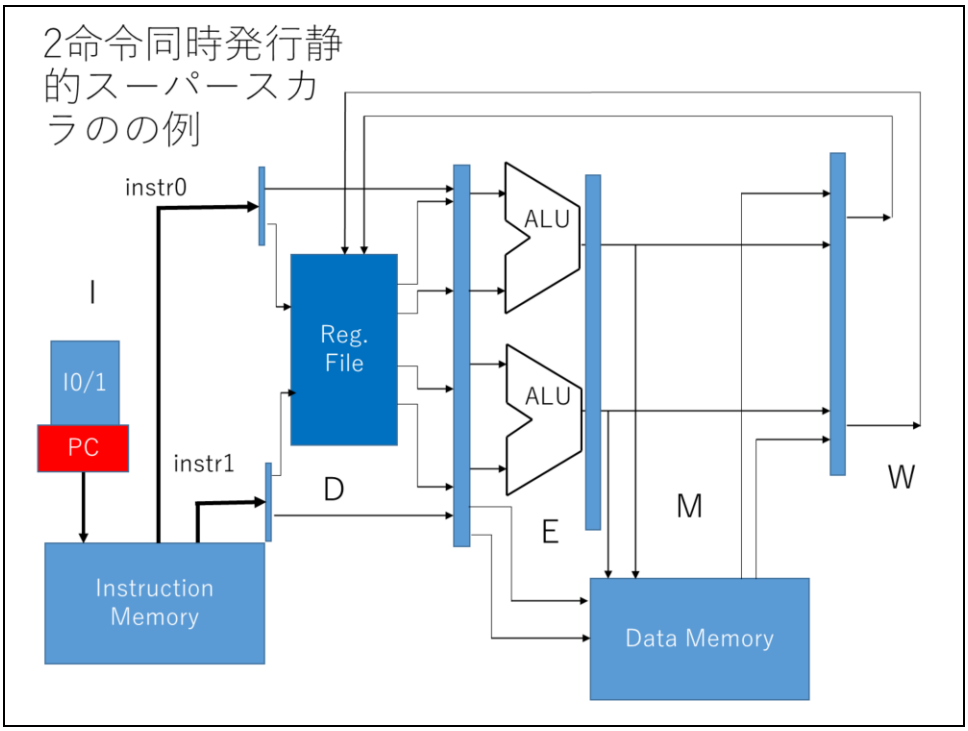


PCが指し示す命令だけでなく、その次の命令、次の次の命令を同時に発行させたらばどうでしょう？うまく同時に動くことができれば、高速化が実現できるはずです。ハードウェアの管理でこれを実現する方法をスーパースカラと呼びます。この方法は、命令の依存関係を管理するためのハードウェアが複雑になるので、4命令同時発行くらいが多いです。しかし、今までの機械語をコンパイルしなおさないで高速化できるので、ビジネスモデルに良く合うことから、現在のほとんどの高性能CPUで使われています。

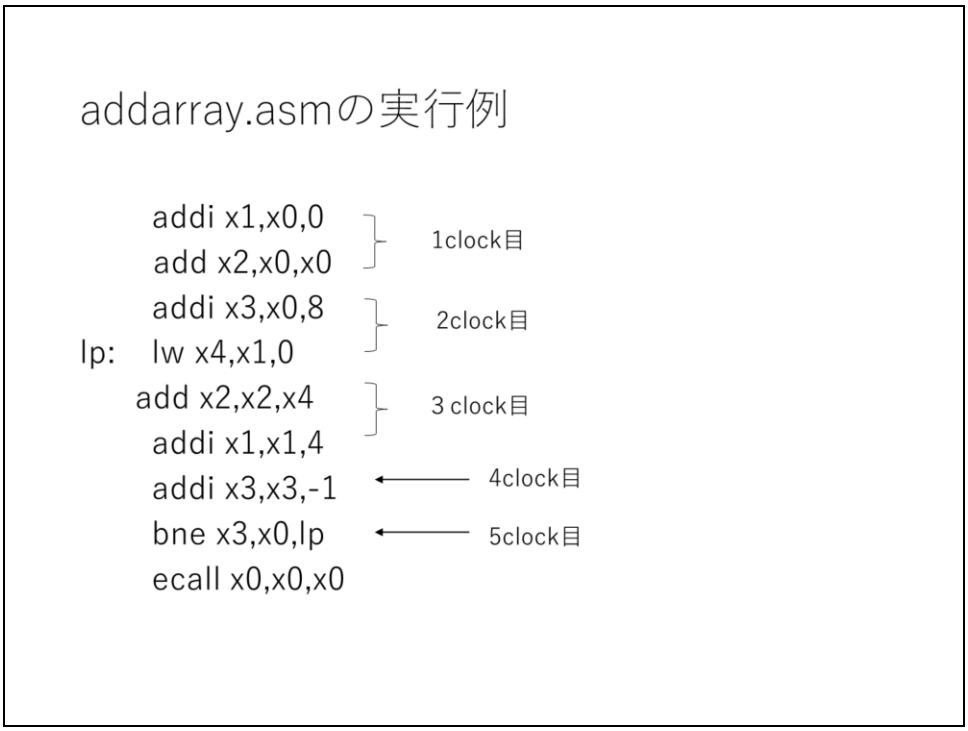
一方、複数の命令をひとまとめにして長い命令として扱う方法をVLIW（Very Long Instruction Word)方式は、依存性の管理をコンパイラで行うため、制御回路は簡単になり、8命令同時発行くらいまで比較的簡単に実装できます。しかし、再コンパイルする必要があることから汎用CPUのビジネスモデルに載らず、主としてDSP(信号処理プロセッサ）で使われます。



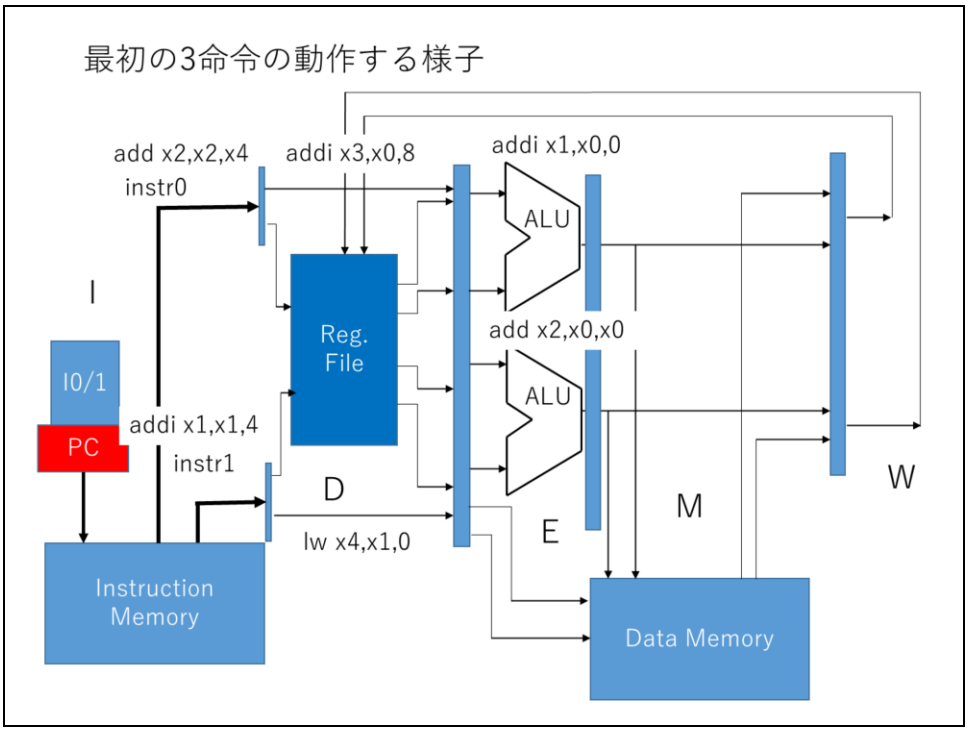
静的なスーパースカラプロセッサは命令をプログラムの順番通り実行するインオーダー（in-order)プロセッサです。ここでは簡単な例を示します。パイプラインを2本持ち、それぞれは全ての種類の命令実行が可能です。このため、ALUは2セット、レジスタファイルは4ポート読み出し、2ポート書き込み、データメモリも2ポートに拡張してあります。まずIFステージでPCの示す番地から2命令分（命令0、命令1）取って来て、2命令実行可能か、1命令だけにするかを判断します。2命令同時実行可能なのは命令0と1の間に依存関係がない場合で、この時は命令0をパイプライン0に、命令1をパイプライン1にいれて、PCを＋8します。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。



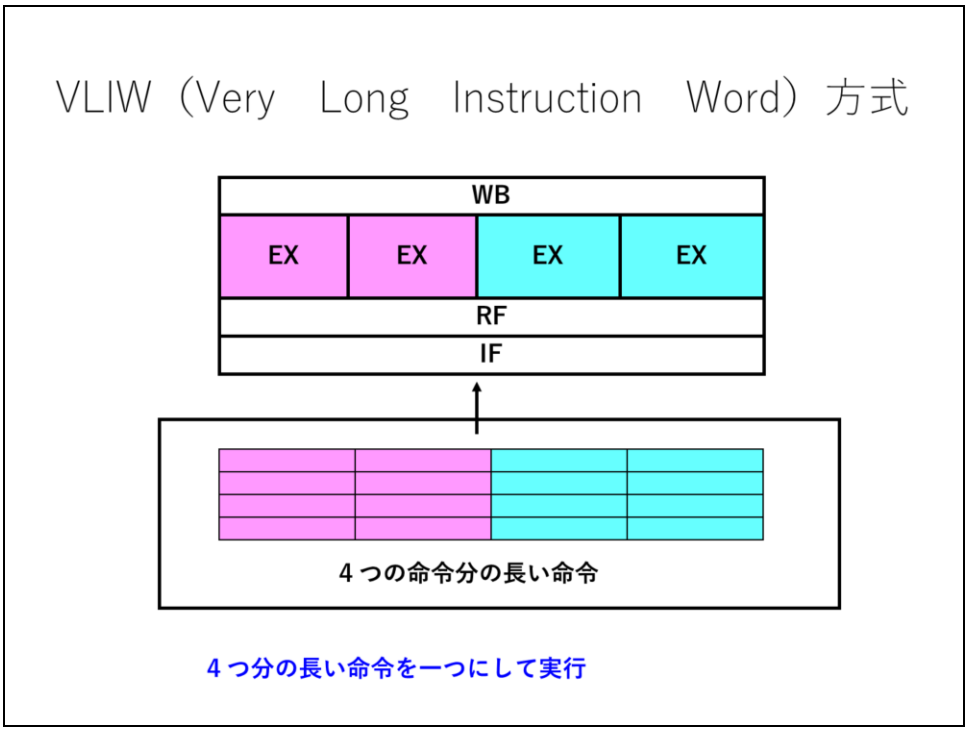
RISC-V静的スーパースカラ版のブロック図を示します。パイプライン0，1がレジスタファイル、メモリを共有しています。静的スーパースカラで2命令同時発行なので、比較的構造は簡単です。



簡単なプログラムがスーパースカラで動作する例を示しましょう。これは、メモリ上に並んでいる8つの数の和を計算する大変単純なプログラムです。最初の辺は問題なく2命令ずつフェッチが行われます。



このプログラムの最初の三命令の動作する様子を示します。この部分は調子良く2命令が同時に実行されますが、実はlwがMに入る際には全体がストールしますし、分岐命令は片方にしか入れません。



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。

例題のVLIWマシン

- 4命令分を1命令にパックして実行
  - 命令メモリは128ビット
- 命令slot0/１はALU命令と分岐命令　ALUは2セット持つ
- 命令slot2/３はメモリアクセス命令　データメモリは2ポート
- rfileは12ポート（8ポート読み出し、4ポート書き込み）
  - 書き込みの制御が面倒になっている
- フォワーディングはやっていない
  - ４X4=16 通りの組み合わせが必要でハードウェアが巨大化するため
  - しかし普通のVLIWマシンではこれをやっている
  - 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
  - NOPだらけになってしまう

RISC-VのVLIW版を紹介します。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。

sum.asm(メモリの内容の加算）の実行

	slot0	slot1	slot2	slot3
	addi x1,x0,0	add x2,x0,x0	nop	nop
	addi x3,x0,8	nop	nop	nop
	nop	nop	nop	nop
lp:	addi x1,x1,4	nop	lw x4,x1,0	nop
	addi x3,x3,-1	nop	nop	nop
	nop	nop	nop	nop
	add x2,x2,x4	nop	nop	nop
	bne x3,x0,lp	nop	nop	nop
	nop	nop	nop	nop
	ecall	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RISC-Vevl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。

ループアンローリング

- addarray.asm

```
        addi x1,x0,0
        add x2,x0,x0
        addi x3,x0,8
lp:  lw x4,x1,0
        add x2,x2,x4
        addi x1,x1,4
        addi x3,x3,-1
        bne x3,x0,lp
        nop
        nop
        sw x2,0x7fff(x0)
end:    beq x0,x0,end
        nop
        nop
```

- addarray_lu.asm

```
        addi x1,x0,0
        add x2,x0,x0
        add x6,x0,x0
        addi x3,x0,8
lp:     lw x4,x1,0
        lw x5,x1,4     //2つ分lw
        add x2,x2,x4
        add x6,x6,x5  //2つ分加算
        addi x3,x3,-2  //カウンタは2減らす
        addi x1,x1,8   //ポインタは8進める
        bne x3,x0,lp
        nop
        nop
        add x2,x2,x6   //この分は損する
        sw x2,0x7fff(x0)
end:    beq x0,x0,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。

VLIW版でのループアンローリング

	slot0	slot1	slot2	slot3
	addi x1,x0,0	add x2,x0,x0	nop	nop
	addi x3,x0,8	add x6,x0,x0	nop	nop
	nop	nop	nop	nop
lp:	addi x1,x1,8	nop	lw x4,x1,0	lw x5,x1,4
	addi x3,x3,-2	nop	nop	nop
	nop	nop	nop	nop
	add x2,x2,x4	add x6,x6,x5	nop	nop
	bne x3,x0,lp	nop	nop	nop
	nop	nop	nop	nop
	add x2,x2,x6	nop	nop	nop
	nop	nop	nop	nop
	ecall	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の利害得失

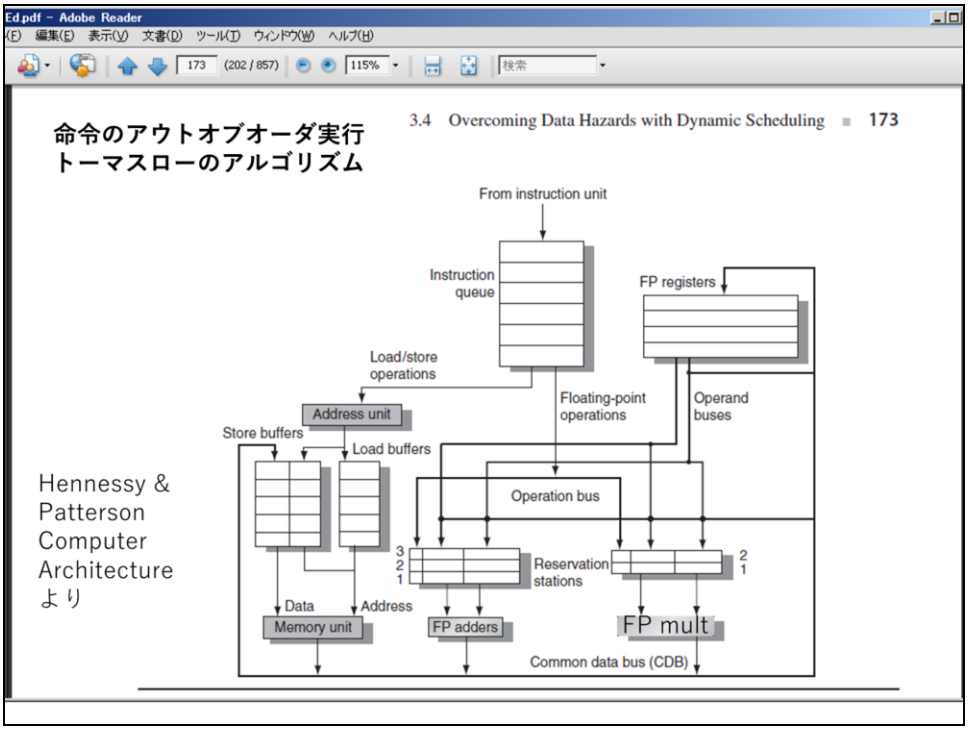
- 利点：
  - 独立に実行できる処理が増える→スーパースカラ、VLIWに有利
  - ループ制御の損失（カウンタのダウン、ポインタのインクリメント、分岐命令など）が減る
- 欠点：
  - ループ間に依存がある場合適用が難しい
  - レジスタを多数必要とする
  - アンロールする回数でループ回数が割り切れないと面倒
  - 場合によっては後処理が必要になる

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

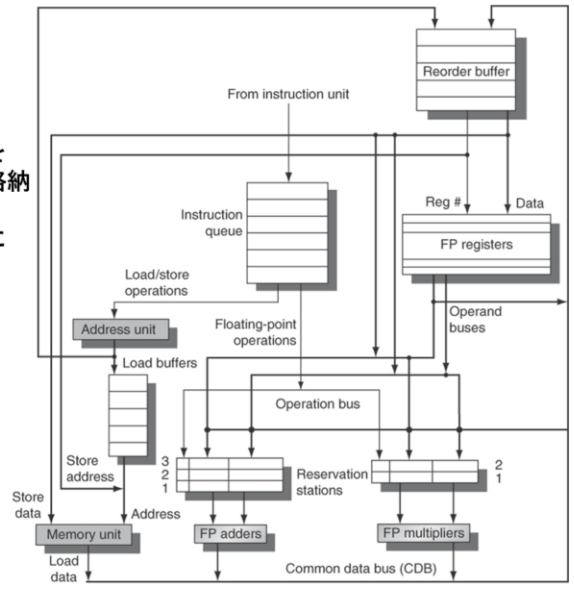


Figure 3.11 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.6 on page 173, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated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

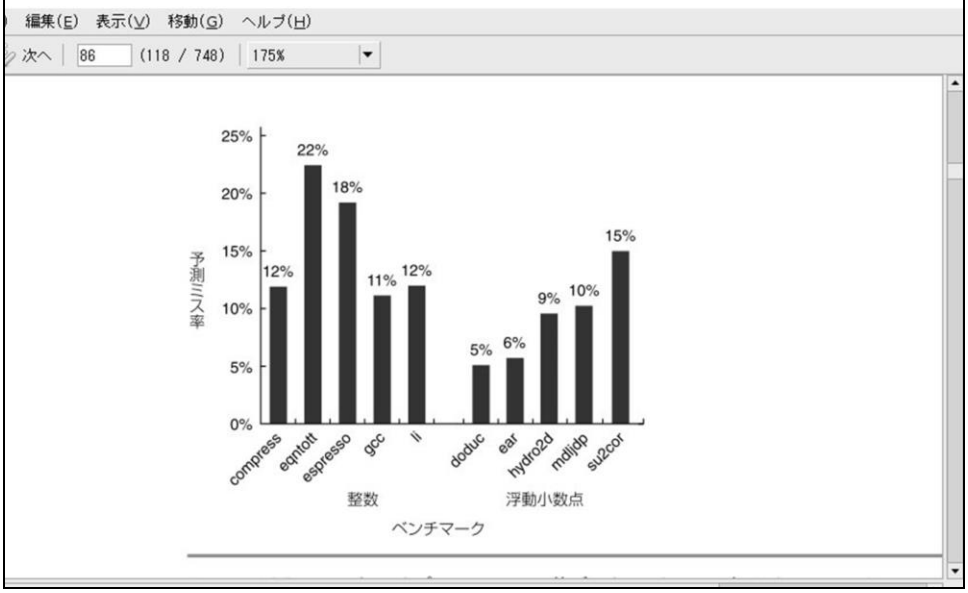
投機的実行をサポートするために拡張したTomasuloのアルゴリズムの図です。実行結果はレジスタに格納される前にリオーダーバッファという所（右上）に一時的にしまわれ、分岐予測の結果が確定すると、レジスタに格納されます。分岐予測がはずれると、その内容はクリアされ、その時点から処理のやり直しになります。この仕組みは実は以前紹介した例外処理の対策にもなることから多くのプロセッサで用いられています。

分岐予測

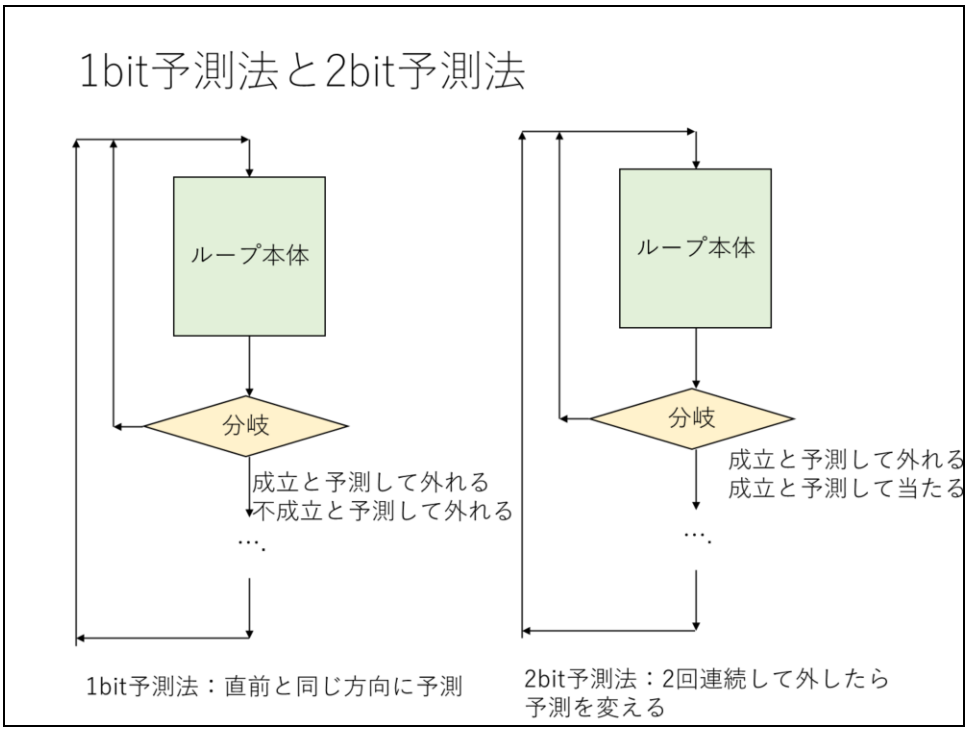
- 一番簡単な予測
  - 全ての分岐を飛ぶと予測すること
    →6-7割位当たる
- 静的予測
  - コンパイラによって行う分岐予測
  - プロファイル（仮に実行して様子を見る）を用いると一定の精度が得られるが限界がある
- 動的予測
  - ローカルなもの（その分岐の過去の結果を使う）
  - グローバルなもの（ある分岐の前後の分岐を見る）
  - トーナメント法：両方考えて当たる方を使う
  - TAGE法：最近話題の手法

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。

静的予測の限界

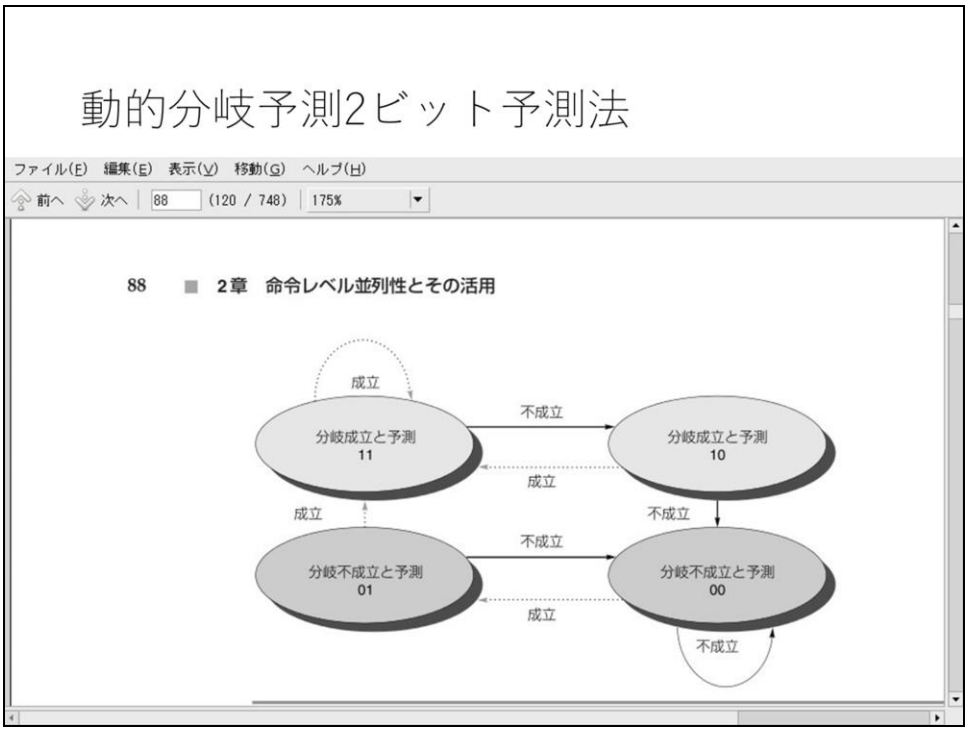


この図は静的予測の精度を示しています。この程度が限界です。



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

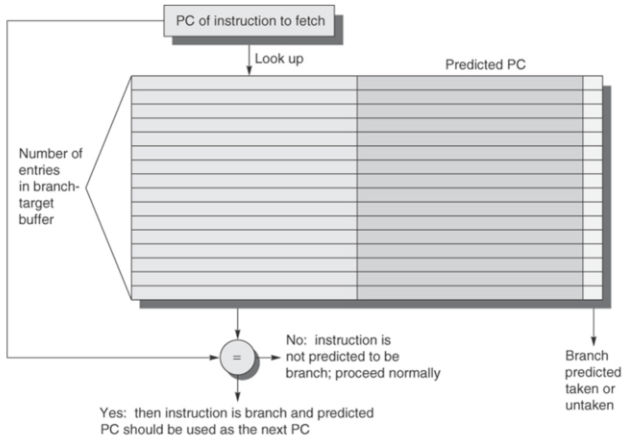
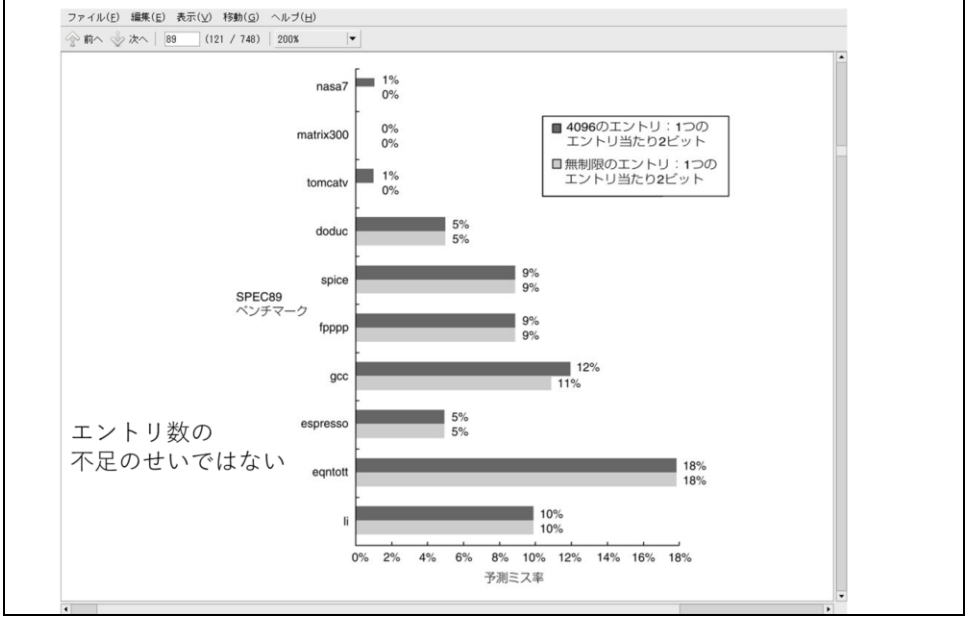


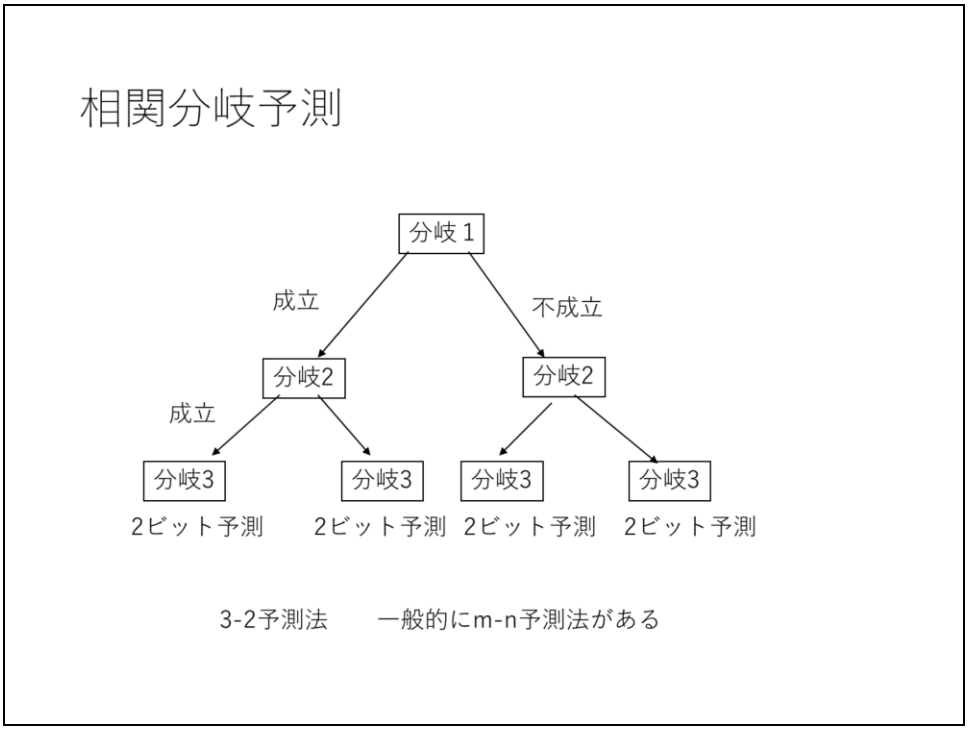
Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and 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

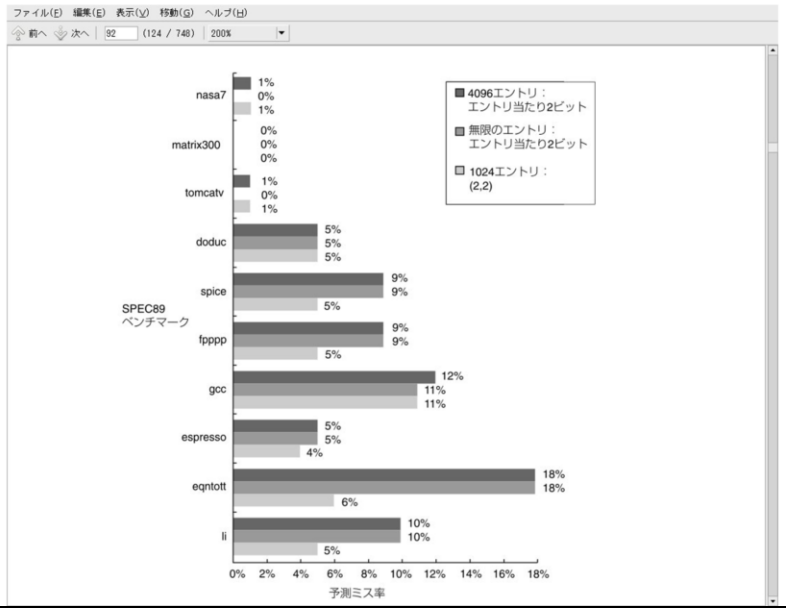


この図は2bit予測法のミス率、つまり外れた割合を示します。分岐予測バッファの容量が足りないと、分岐命令が競合してしまい間違った分岐の結果で予測をする場合があります。しかし、容量が無限に大きくしてもミス率が減らないことから、この影響はわずかであることがわかります。



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。

gshare: 相関分岐予測の簡単で効果的な実装法

186 ■ Chapter Three *Instruction-Level Parallelism and Its Exploitation*

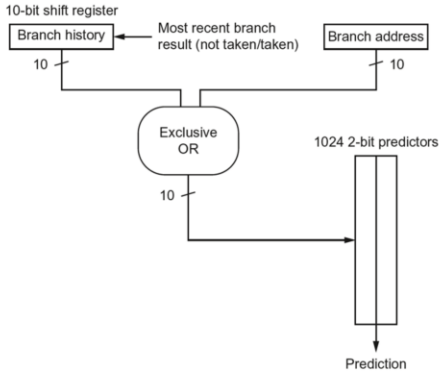


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

215.9 x 260.3 mm

相関分岐予測の簡単で効果的な実装法としてgshareという方式が知られています。この方式は10ビットのシフトレジスタで過去の分岐が飛んだかどうかを表したものと、分岐のアドレスの排他的論理和を取ってもので、2ビット予測器を索引します。この方法は簡単な割に良く当たります。

トーナメント方式

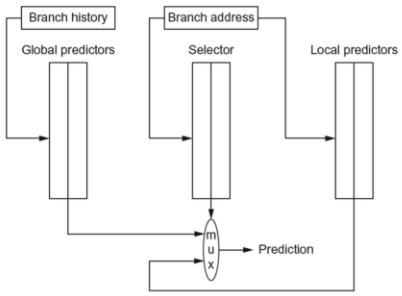
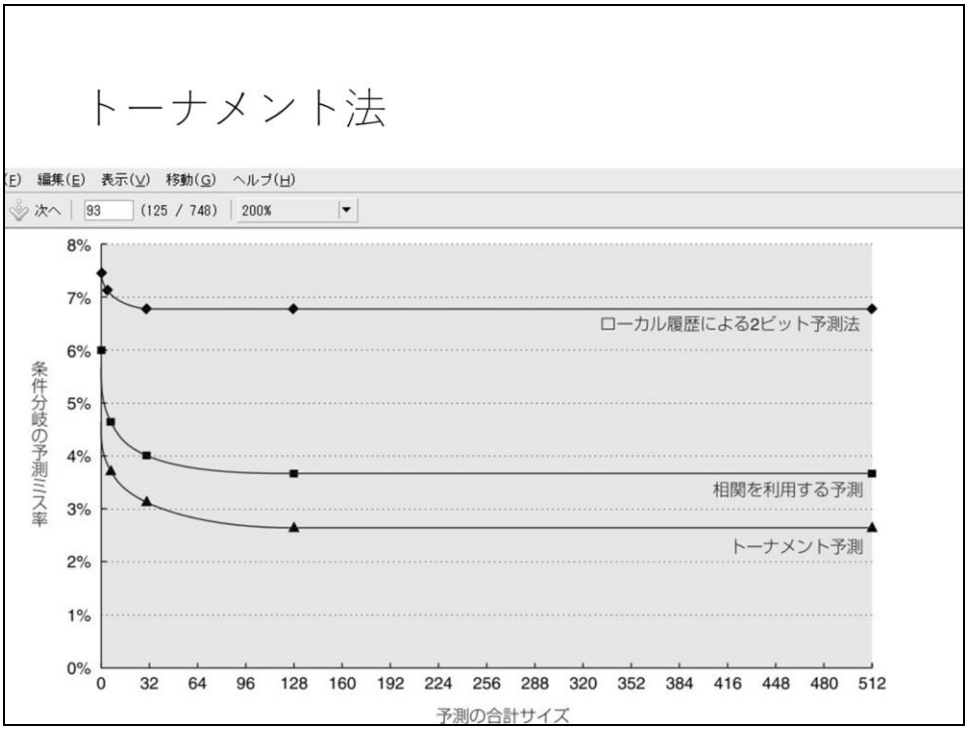


Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case, the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

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使います。図中のSelectorのテーブルに過去どちらが当たったかの履歴が格納されています。



ローカル履歴だけ、相関を利用する予測、トーナメント予測を比較したものです。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。

TAGE(TAgged GEometoric predictor)

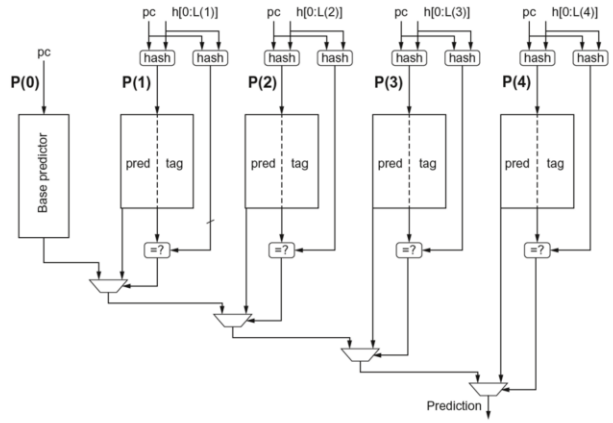


Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

最近、良く当たることで注目されているのがTAGE(TAgged GEometoric predictor: タグ付きハイブリット予測器)で、ハッシュによりテーブルを索引することで容量を減らした複数のテーブルから、一致した最長の分岐履歴に対応する予測器のものを用います。

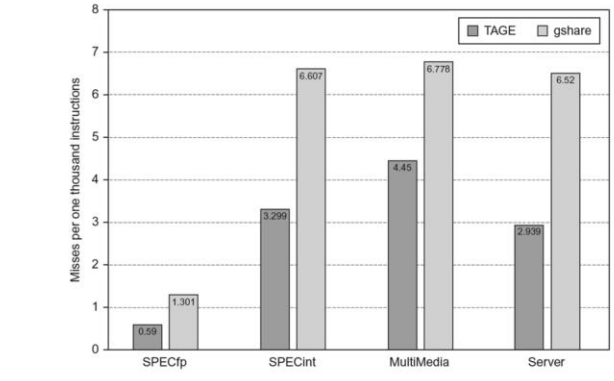


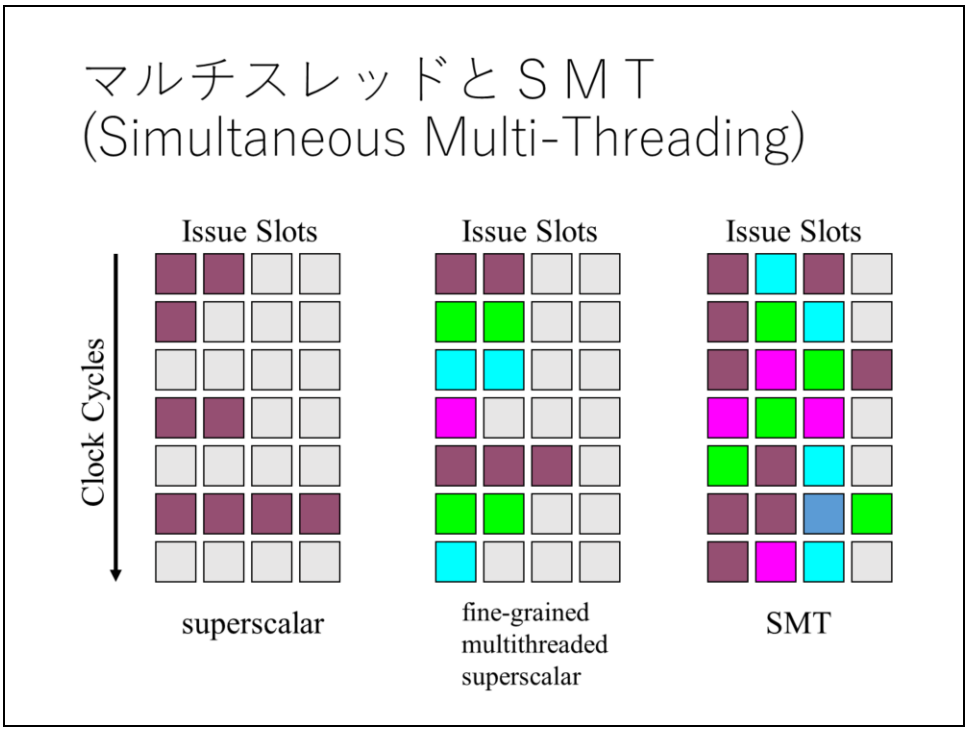
Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

図中にgshareとTAGEのミス率の差を示します。両方の予測器は同じビット数を使うように評価されています。マルチメディア用、サーバー用のかなり当たり難いアプリケーションでも好成績を達成していることがわかります。

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
- TAGEの登場により、ほぼ限界に達したか？

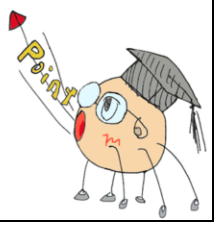
分岐予測をまとめます。まだ分岐予測の研究が終わったわけではないですが、最近のTAGEの登場により、ほぼ性能向上は限界に達したのではないかと言われています。ニューラルネットワークを使って学習させて、予測率を上げたりする試みもあります。



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。

演習

- RISC-Vのシミュレーション環境を用いて、メモリ上の0番地から28番地までの8つの数の総和を求めるプログラムmsum.asmを作れ。答えは16進数で38になるはず。