第0章　はじめに

この講義ではRISCプロセッサのためのコード最適化 — 特にループの**ソフトウェア・パイプライン化** — を修得することを目的としている。

コンパイラはプログラム開発に必須のツールであり、高水準言語のソースプログラムを機械語コードへ自動変換する。機械語コードが元のソースプログラムの意図する通りに動作すべきことは当然であるが、コードはできる限り高速に動作すべきである。この講義ではそこに焦点を当てて様々な高速化手法を紹介する。普段、私たちが利用するコンパイラが私たちの目の届かない最深部で何を行っているのか（行おうとしているのか）を知る一助としてほしい。

0.1　RISCとは

IT用語辞典（http://e-words.jp/）では、**RISC**（Reduced Instruction Set Computer）を以下のように説明している。

RISC ... マイクロプロセッサの設計様式の一つ。個々の命令を簡略化することによりパイプライン処理（並行して複数の命令を処理する方式）の効率を高め、処理性能の向上をはかっている。ワークステーション用のCPUにはこの型のプロセッサが多い。Sun Microsystems社のSPARCやDEC社（現在はCompaq Computer社の一部門）のAlpha、IBM社とMotorola社のPowerPCなどが有名。

この文章中の**パイプライン処理**とは、命令が連続的に実行され、しかも先に実行を開始した命令の終了を待たずに後続の命令の実行を開始する方法[1]と説明することもできる。この特徴を可能な限り活かして、プログラム（アセンブリコード）の高速実行を実現するのがこの授業の目的である。

なお、RISCに対立する語はCISC（Complex Instruction Set Computer）だが、同辞典では以下のように説明している。参考にされたい。

CISC ... マイクロプロセッサの設計様式の一つ。個々の命令を高級言語に近づけ、複雑な処理を実行できるようにすることで処理能力の向上をはかっている。パソコン用のCPUとしてあわせて9割以上のシェアを持つIntel社のx86シリーズとその互換プロセッサはCISCと見

[1] これを置いてきぼり処理ともノンブロック型実行とも呼ぶ。

なされている。しかしプロセッサの内部構造は RISC そのものになっており、最新の RISC が旧式の CISC の衣をまとっているような構造である。

0.2　RISC プロセッサの実行形態

さて、最近の RISC プロセッサはその実行形態によってさらに以下の二つのカテゴリに分類できる。

out-of-order 型実行 ... 依存関係のない複数の命令を、それらのプログラム（コード）中での出現順序にとらわれず、次々と実行し、高速化する方式。実行順序をプロセッサが自動判定するため、ハードウェアは複雑になりがちである。しかしコンパイラ（コード最適化器）にかかる負担が小さい。

in-order 型実行 ... out-of-order でない方式。命令をプログラム（コード）中での出現順序に厳密に従い、実行する。ハードウェアを簡素化できるが、実行効率を挙げるにはコンパイラによるコード最適化が必須である。

この授業はコード最適化を目的としているため、後者の in-order 型プロセッサを仮定する。これ以降、実行形態を議論することはないが、一応、上記事項を念頭において進めていきたい。

第1章　RISCプロセッサの構成要素

以下では、この授業で想定するプロセッサの仕様を定める。0章で紹介したRISCプロセッサを具体化する訳である。内容が十分理解できない場合には2章または3章まで読んだ後に戻ってくればよいだろう。

1.1　レジスタ

この授業で対象とするRISCプロセッサは以下のようなレジスタ（register, CPU内の高速記憶領域）を持つと仮定する。

汎用レジスタ （general register、general purpose register） ... ひとつの汎用レジスタは64bit長の整数データを保持[1]し、プロセッサはそのような汎用レジスタを128個持つと仮定する。以下、GRと総称する。個々のレジスタを`r0`、`r1`、`...`、`r127`で表す。

浮動小数点レジスタ （floating-point register） ... ひとつの浮動小数点レジスタは64bit長の浮動小数点データ（すなわちdoube型データ）を保持し、プロセッサはそのような浮動小数点レジスタを128個持つと仮定する。以下、FRと総称する。個々のレジスタを`f0`、`f1`、`...`、`f127`で表す。

その他にもプロセッサには以下のようなレジスタが暗黙に仮定される。しかし、当面これらを使用することはない。参考程度に憶えておけばよい。

命令レジスタ （instruction register） ... プロセッサが実行している命令のメモリ・アドレスを保持している。

条件レジスタ （condition register）または**述語レジスタ**（predicate register） ... 比較命令の実行結果を保持し、条件付き分岐命令で参照する。述語レジスタは講義の後半で用いる。

アプリケーション・レジスタ （application register） ... 現在時刻、システムのconfiguration、モニタリングに関連する情報を格納している。

表 1.1: この講義で用いる命令の一覧

項番	命令形式	概略
1 (a)	`ldfd f`i` = [r`j`]`	浮動小数点数のロード
	`ldfd f`i` = [r`j`],8`	同上、rjの自動インクリメント
1 (b)	`stfd [r`j`] = f`i	浮動小数点数のストア
	`stfd [r`j`],8 = f`i	同上、rjの自動インクリメント
2 (a)	`fadd f`i` = f`j`,f`k	浮動小数点加算
2 (b)	`fsub f`i` = f`j`,f`k	浮動小数点減算
2 (c)	`fmpy f`i` = f`j`,f`k	浮動小数点乗算
3 (a)	`add r`i` = r`j`,r`k	整数加算
	`add r`i` = r`j`,`*imm*	定数との整数加算
3 (b)	`sub r`i` = r`j`,r`k	整数減算
	`sub r`i` = r`j`,`*imm*	定数との整数減算
3 (c)	`mpy r`i` = r`j`,r`k	整数乗算
	`mpy r`i` = r`j`,`*imm*	定数との整数乗算
4 (a)	`cmp.`*rel* `r`i`,r`j	整数の比較
	`cmp.`*rel* `r`i`,`*imm2*	整数と定数の比較
	`cmp.`*rel* *imm1*`,r`j	整数と定数の比較
4 (b)	`fcmp.`*rel* `f`i`,f`j	浮動小数点数の比較
4 (c)	`br.cond` *label*	条件付き分岐
4 (d)	`br` *label*	無条件分岐
5 (a)	`nop`	何もしない

1.2　命令の種類と概要

この授業で使用するRISCプロセッサの**命令**（instruction）を表1.1にまとめた。これらはIntel社IA-64プロセッサの命令セットを参考にしており、以下にそれらの詳しい説明を述べる。

1. **メモリ・アクセス**

   (a) **浮動小数点データのロード**

   - `ldfd f`i` = [r`j`]`
   - `ldfd f`i` = [r`j`],8`

   GR `r`j が示すメモリ・アドレスから8バイトのメモリ領域の内容を、FR `f`i に転送する。2番目の形式では転送処理の起動直後に `r`j の値を8増やす。この命令形式は配列演算などのメモリを連続アクセスする場合に有効である。なお、この授業で扱う浮動小数点データは全て `double` 型（8byte）であると仮定する。命令名 `ldfd` は、load float double に由来する。

[1] よって、いわゆる64bitプロセッサである。

(b) **浮動小数点データのストア**

- `stfd [r`j`] = f`i
- `stfd [r`j`],8 = f`i

FR `f`i の内容を GR `r`j が示すメモリ・アドレスから 8 バイトのメモリ領域に転送する。２番目の形式は転送処理の起動直後に `r`j の値を 8 増やす。これは配列演算などのメモリを連続アクセスする場合に有効である。命令名 `stfd` は、store float double に由来する。

2. **浮動小数点演算**

(a) **浮動小数点加算**

- `fadd f`i `= f`j`,f`k

FR `f`j と `f`k の和を求め、結果を FR `f`i に代入する。命令名 `fadd` は、float add に由来する。

(b) **浮動小数点減算**

- `fsub f`i `= f`j`,f`k

FR `f`j から `f`k を減じ、結果を FR `f`i に代入する。命令名 `fsub` は、float subtract に由来する。

(c) **浮動小数点乗算**

- `fmpy f`i `= f`j`,f`k

FR `f`j と `f`k の積を求め、結果を FR `f`i に代入する。命令名 `fadd` は、float multiply に由来する。

なお、この授業では除算は対象としない。除算は加減乗算を組み合わせて近似計算する方法が一般的であるが、演算コストが高く、特別な扱いが必要となることが多いからである。

3. **整数演算**

(a) **整数加算**

- `add r`i `= r`j`,r`k
- `add r`i `= r`j`,`imm

GR `r`j と `r`k （または定数 imm）の和を求め、結果を GR `r`i に代入する。レジスタは 64bit であり、64bit 演算である。

(b) **整数減算**

- `sub r`i `= r`j`,r`k
- `sub r`i `= r`j`,`*imm*

GR `r`j と `r`k （または定数 *imm*）を減じ、結果を GR `r`i に代入する。

(c) **整数乗算**

- `mpy r`i `= r`j`,r`k
- `mpy r`i `= r`j`,`*imm*

GR `r`j と `r`k （または定数 *imm*）の積を求め、結果を GR `r`i に代入する。64bit どうしの整数乗算結果は一般に 128bit 長になるが、簡単のため、ここでは結果は 64bit 長を超えないものと仮定する（仮に 64bit を超えてもオーバーフローを無視する)。

4. **比較演算と分岐**

(a) **整数の比較**

- `cmp.`*rel* `r`i`,r`j
- `cmp.`*rel* `r`i`,`*imm2*
- `cmp.`*rel* *imm1*`,r`j

GR `ri` （または定数 *imm1*）と GR `r`j （または定数 *imm2*）を関係演算 *rel* に関して比較する。*rel* の種類は、

`==`、　`!=`、　`>`、　`<`、　`>=`、　`<=`

の 6 種類である。命令名 `cmp` は、compare に由来する。

(b) **浮動小数点数の比較**

- `fcmp.`*rel* `f`i`,f`j

FR `f`i と `f`j を関係演算 *rel* に関して比較する。*rel* の種類は 4.(a) と同じである。命令名 `fcmp` は、float compare に由来する。

(c) **条件付き分岐命令**

- `br.cond` *label*

直前の `cmp`、`fcmp` の結果が真ならば、制御はラベル（英数字からなる名前）*label* の位置へ分岐する。さもなくば、制御はそのまま直後の命令へ移る [2]。なお、分岐先ラベルの設定は、コード中に

```
label:
```

[2] これを fall-through（下へ通り抜けるの意味）などとも呼ぶ

と、ラベル名 `label` とコロン : を書いて指定する。命令名 `br.cond` は、branch conditional に由来する。

(d) **無条件分岐命令**

- `br` *`label`*

ラベル *label* の位置へ分岐する。

5. **その他の命令**

(a) **何もしない命令**

- `nop`

この命令は、レジスタ、メモリに何の変化も与えない。通常、`nop` を実行する価値はほとんどないが、この授業では後に述べるプロセッサ・ストールを明示するために使用する。命令名 `nop` は、no operation に由来する。

上の命令セットの特徴のひとつは、メモリにアクセスする命令がロード/ストア命令のみに限定されていることである。その他の命令ではレジスタのみを演算対象としていることに注意してほしい。このような特徴を持つ命令セット（および CPU）を特に**ロードストア・アーキテクチャ**と呼ぶ。一般に、メモリ・アクセスの速度はレジスタ・アクセスの速度に比べ 10 倍以上遅く、近年その差は開く一方である。そこでメモリ・アクセスを行う命令を制限し、逆に演算はレジスタ間に制限することで、演算速度の低下を防ぐのが目的である。このため、データを保持する十分な数のレジスタが必要となるが、通常のほとんどの演算は 16 個程度のレジスタがあれば足りると考えてよい[3]。

1.3　命令の実行モデル

ここではプロセッサの実行過程をいくつかの用語と共に定める。

実行と発行:　プロセッサは命令を**実行**（execute）する。その実行を開始することを、特に**発行**（issue）と呼ぶ。実行と発行は同じ意味で使用される場合もあるが、厳密には区別されるべきである。

命令並列度:　プロセッサは**マシーン・サイクル**（Machine Cycle、以下では MC と表記する）の刻みにあわせて次々と命令を発行していくが、同時に実行できる命令の最大数を**命令並列度**（degree of instruction parallelism）と呼ぶ。しかしこれはあくまでも「可能な最大数」であって、以下に

[3] しかしレジスタ数は多ければ多いほどよい。レジスタはグローバル・データを保持するためにも利用できる。後に述べるソフトウェアパイプライン化ではレジスタの個数分だけ高度な最適化が可能になる。

述べるように、命令間の依存関係などの理由によって、その数の命令を発行できない場合も多々ある。また多くのプロセッサでは、同時に発行できる命令の種類に制限がある（二つのロード命令は同時に発行できない、など）。しかし、この授業では簡単のため、与えられた命令並列度の範囲内で任意の命令の組み合わせが同時に発行できるとする。より多くの命令を同時発行し、トータルとしてプログラムの実行速度を上げることがコード最適化の目的のひとつでもある。命令並列度が 2 以上のプロセッサを**スーパースカーラー**（super scalar）と呼ぶ。

ソースとターゲット:　命令は、通常、あるデータをレジスタから読み込み、あるレジスタに結果を書き出す。この読み込むレジスタのことを**ソース**（source）と呼び、書き出すレジスタのことを**ターゲット**（target）と呼ぶ。または入力レジスタ、出力レジスタと呼ぶ。たとえば、浮動小数点加算:

```
fadd f1 = f2,f3
```

では `f2` と `f3` はこの命令のソースレジスタ、`f1` はターゲットレジスタである。またロード命令:

```
ldfd f1 = [r1]
```

では `r1` はこの命令のソースレジスタ、`f1` はターゲットレジスタである。`r1` に格納されているデータは読み込むデータそのものではないが、命令を実行する際の入力情報であるから、ソースとなる。ストア命令:

```
stfd [r1] = f1
```

では、`r1` と `f1` は共にソースレジスタである。この場合、出力先はメモリであるため、ターゲットレジスタはない。以下のロード命令:

```
ldfd f1 = [r1],8
```

では `r1` はソースであるが、同時にターゲットでもある。何故ならば、この命令では浮動小数点データのロードと同時に `r1` の値を 8 増やす（更新された値が格納される）からである。

レーテンシ:　命令は、それが「発行」されてから実行が開始され、そして一定時間後に「終了」する。この、発行されてから終了するまでにかかる時間を**レーテンシ**（latency）と呼ぶ。単位はマシーン・サイクル数（MC 数）である。たとえば市販の多くのプロセッサの演算命令のレーテンシは 1〜5MC 程度である。レーテンシはプロセッサの状態、CPU 内のデータの流れ方などに依存して変動する場合がある。一般に演算命令のレーテンシの変動は高々1〜2MC 程度である。しかしロード命令のレーテンシだけは例外的に大きく、データがキャッシュ・メモリに載っているか否か、同一のメモリ・バンクへアクセスが集中していないか否か等に依存し、100MC 幅で変動することもある。

しかしこの授業では簡単のために、全てのデータはキャッシュ・メモリに載っていると仮定し、ロード命令を含め、全ての命令のレーテンシは変動しない（すなわち定数である）と仮定する。

データ依存:　ある命令 I1 のターゲット・レジスタが、後続の命令 I2 のソース・レジスタまたはターゲット・レジスタであるとき、I2 は I1 に**依存**（depend）すると言う。たとえば加算結果をメモリへストアする以下の二つの命令の並びの場合:

```
fadd f1 = f2,f3
stfd [r4] = f1
```

ストア命令は加算命令に依存している。この例題のような依存を特に Read After Write 依存（レジスタへの書き込み後に同じレジスタから読み出す依存、RAW 依存と略す）と呼ぶ。例題の場合、加算命令が f1 へ演算結果を書き込んだ後にストア命令が f1 の値を読んでいる。

また二つの命令が同じターゲットへ書き込む場合:

```
fadd f1 = f2,f3
fsub f1 = f4,f5
```

後者の減算命令は前者の加算命令に依存している。これを特に Write After Write 依存（レジスタへの書き込み後に同じレジスタへ書き込む依存、WAW 依存と略す）と呼ぶ。

以下の例:

```
stfd [r2] = f1
fadd f1 = f2,f3
```

では、f1 に関して加算命令はストア命令に依存している。これを特に Write After Read 依存（レジスタからの読み込み後に同じレジスタへ書き込む依存、WAR 依存と略す）と呼ぶ。つまり、上の例では f1 のデータをメモリにストアする前に（あるいはストアしている間に）加算結果を f1 に書き込んではいけない。

コード最適化において命令間の依存を考慮することは非常に重要である。何故ならば、依存のある命令は同時に実行できないからである。上には RAW、WAW、WAR の 3 種類の依存を述べたが、この中で RAW 依存は本来の計算の性質上避けることができないが、WAW 依存と WAR 依存はコードを工夫することで解消できる。たとえば上の WAW 依存の例の場合、以下のように、減算命令のターゲット・レジスタ番号を f6 へ変更すれば依存が解消される。

```
fadd f1 = f2,f3
fsub f6 = f4,f5
```

1 そして、この二つの命令は並列実行可能となる。このとき、必要とされるレジスタ数は増加する
2 が、それは避けようがない。この場合、空間効率（必要レジスタ数）を犠牲にして時間効率（実行
3 速度）を優先した訳である。

4 **補足:**　上ではレジスタに関して RAW/WAW/WAR 依存を述べたが、一般にこれらの依存はメモ
5 リに関しても考えることができる。メモリに関する依存は、レジスタに関する場合よりも解析が難
6 しいことが知られている。たとえば、以下のコード:

```
7          stfd [r1] = f1
8          stfd [r2] = f2
```

9 では、`r1` と `r2` が同じアドレス値を保持する場合には WAW 依存となる。しかし、`r1` と `r2` が同じ
10 値を持つか否かを静的に解析するには、それらの値がどのように作られたか、コードを逆に辿る必
11 要がある。しかも辿ったからといって答えが得られるとは限らない。解析対象とするメモリアドレ
12 スが、C 言語で言う配列部に相当する場合にはソースプログラムの情報を用いることで比較的容易
13 に解析できるが、ポインタに相当する場合には解析は非常に困難と考えられており、最適化のネッ
14 クになる。

15 **プロセッサ・ストール**　命令が実行される様子を詳細に検討しよう。命令 `I1` と `I2` が、以下のよ
16 うに、この順番に実行されると仮定する。

```
17          I1
18          I2
```

19 まず、命令 `I1` の実行が開始されると、`I1` のターゲット・レジスタにロック（lock）が掛かり、他
20 の命令からこのレジスタへアクセスすることができなくなる。そのロックは、`I1` の実行が終了し、
21 演算結果がそのターゲット・レジスタに書き込まれた直後にはずれる。ロックが掛かっている状態
22 で `I1` に依存する命令 `I2` がそのレジスタの値を読み込もうとしたり、書き込もうとした場合には
23 `I2` の実行はそこで待たされる。待っている間は後続の命令は発行できない [4] ため、プログラムの
24 実行効率が落ちる。これを**ストール** [5]（stall、立ち往生）という。1 章で述べた out-of-order 型プロ
25 セッサではストールを回避するハードウェア機構を持っているが、それも万能ではないため、コー
26 ド最適化ではストールしないようなコードを作ることが重要である。この授業の論点はまさにここ
27 にある。

[4] この授業では in-order 型プロセッサを仮定しているためである。out-of-order 型プロセッサでは、依存のない命令を自動的に探し出し、命令の並び順に関わらず、それを発行する。これが out-of-ordr（順序不問）と呼ばれる理由である。そのため in-order 型よりもプログラムの実行効率が総じて高くなる。しかし 0 章で触れたように、out-of-order 型のプロセッサは高機能であるため、in-order 型に比べ、LSI チップの構造が複雑になる。逆に言えば、in-order 型は命令依存解析部の構造が out-of-order 型よりも簡単なため、プロセッサのクロック数を上げたり、あるいは別の機能を付加することが可能となる。

[5] 自動車で使う「エンスト」という語はエンジン・ストールの略である。

なお、ロックが掛るのはターゲット・レジスタだけであり、ソース・レジスタにはロックは掛らない。というのも、ソース・レジスタの値は命令実行開始直後にプロセッサ内に別に用意されているバッファへコピーされる[6]。コピーされた後に別の命令がそのレジスタにアクセスしても何の問題も生じないからである。

パイプライン実行　ある命令が発行され、その命令の実行が終了する前に、後続の命令を発行できる機構を**命令パイプライン**と呼ぶ。命令の実行は、少なくとも

- フェッチ（命令をメモリ/キャッシュから読む込むこと）、
- デコード（命令を解釈し、プロセッサ内の演算器などの各部位へ指令を与えること）、
- 実行（所定の演算を行い、演算結果を得ること）

の 3 ステップ[7] からなる。各命令はプロセッサ内でこの 3 ステップに従って順に処理されていく。

今、以下のように、命令 I1、I2、I3、... が順に発行されていくことを考える。

```
I1
I2
I3
...
```

命令パイプラインでは、3 ステップを次々と実行していく（下図参照）。

タイミング	命令 I1	命令 I2	命令 I3	...
1	フェッチ			
2	デコード	フェッチ		
3	実行	デコード	フェッチ	
4		実行	デコード	...
...			...	...

つまり I1 の実行終了を待たずに I2 の実行を開始する。このように、命令の実行を複数ステップに分解し、先行する命令の実行終了を待たずに、次々と命令を発行していくことで短時間当りの命令発行数を増やすことができ、結果として、プロセッサの実行性能が向上する。

[6] 十分なバッファを有しない、機能の低い低価格のプロセッサの命令や平方根や除算などの複雑な命令では、ソース・レジスタをロックするものもある。その場合、命令実行終了までそのレジスタにはアクセスできない。またストア命令が連続して発行された場合には、プロセッサからメモリへのデータ転送路が混雑し、バッファが詰まる場合がある。その場合、ストア命令がストールすることがある。余談であるが、高価なワークステーションと低価格のパソコンの機能の違いのひとつに、プロセッサとメモリの間の転送速度、転送容量の差がある。如何にプロセッサが高速であっても、転送速度が遅いと、そこがネックになって性能が出ない。

[7] 3 ステップというのは説明のため単純化したステップ数であり、最近のプロセッサでは 10 ステップ以上のものが普通である。

第2章　代入文のコード生成例

前章のような設定の下、簡単な例題コード（命令の列）を解説する。なお、これ以降、「プログラム」と「コード」という語を明瞭に使い分け、それぞれ以下の意味とする。

プログラム ... コンパイル前の C 言語風のソース・プログラム。

コード ... アセンブリ・コード (機械語コード)。ソース・プログラムからコンパイルして得られたものであり、1 章のプロセッサで直接実行可能な命令の列。

2.1　代入文のコード生成

まず、図 2.1 のC言語の代入文に相当するコードは図 2.2 の通りである。ただしプロセッサの命令並列度は 1 とし、変数 x、y は double 型であり、x と y のメモリ・アドレス値は r1、r2 に事前に格納されていると仮定する [1]。ここに命令のレーテンシは

命令	レーテンシ（MC）
ldfd	3
stfd	1
fadd/fmpy	2

と仮定する。図 2.2 のコードには nop が現れるが、nop は実行結果には関係しないから、これを図 2.3 のように書いてもよい。しかし、コードを図 2.2 のように書いても図 2.3 のように書いてもプログラムの実行には 4MC を要するから、この授業ではプロセッサのストールする様子を明示するために敢えて nop を加えて表記すると約束する。コード中の nop を見ればプロセッサのストールの状況が分かる。nop の多いコードはそれだけ実行効率の悪いコードと言ってよい。

次に図 2.4 の代入文に相当するコードは図 2.5 の通りである。ただし double 型変数 z のメモリ・アドレス値は r3 に事前に格納されていると仮定する。このコードの 5〜7 行目は

```
5        fadd f1 = f1,f2
6        nop
7        stfd [r1] = f1
```

[1]通常、全ての変数値はメモリ上に格納する。よって変数にアクセスするには、そのアドレスを知っていることが大前提となる。 この授業では、必要なアドレスは全て GR に事前に格納されているものと仮定して話を進めていく。

```
1        x = y;
```

図 2.1: ひとつの代入文

```
1        ldfd f1 = [r2]
2        nop
3        nop
4        stfd [r1] = f1
```

図 2.2: 図 2.1 のコード（&x=r1、&y=r2）

```
1        ldfd f1 = [r2]
2        stfd [r1] = f1
```

図 2.3: 図 2.2 のコードから nop を除いた場合

としてもよい。後者の方が使用する FR の個数が 1 個少ないため、より優れたコードと言えるが、この講義テキストで想定するプロセッサには 100 個以上のレジスタがあると仮定するから、使用レジスタ数を気を使う必要はない。この講義テキストのコード最適化の章では命令毎に異なるターゲットレジスタを用いると約束する。使用レジスタ数を少なくする技術 — これを一般に**レジスタ割付**（register allocation）と呼ぶ — については、コード最適化の後に述べる。

図 2.6 は 2 行の代入文からなるプログラムである。これに対応するコードは図 2.7 の通りである。変数 `x`、`y`、`z` のアドレスは上と同じレジスタにあらかじめ格納されているものとし、変数 `a` のメモリ・アドレスは `r4` に事前に格納されていると仮定する。図 2.6 のコードは図 2.7 が唯一ではない。図 2.8 のコードでも同じ計算ができる。しかも 1MC 分だけ高速である。図 2.7 のコードでは 1 行目から 7 行目までが代入文 `a = y+z;` に対応し、8 行目以降が代入文 `x = a*a+z;` に対応している。それに対して図 2.8 では 7 行目と 8 行目を入れ替えている。

もうひとつプログラム例を示す。図 2.9 のプログラムには同じ式 `y+z` が二つの代入文に現れている。このような式を共通部分式（common subexpression）と呼ぶ。共通部分式は 1 回だけ計算すれば十分であり、同じ計算を 2 回繰り返すことは無駄である。それはよく知られたコード最適化技法のひとつである。

問題　共通部分式に注意して、図 2.9 に対応するコードを作成せよ。命令のレーテンシ、変数のアドレスはここまでと同じとする。

```
1        x = y+z
```

図 2.4: 加算を含む代入文

```
1        ldfd f1 = [r2]
2        ldfd f2 = [r3]
3        nop
4        nop
5        fadd f3 = f1,f2
6        nop
7        stfd [r1] = f3
```

図 2.5: 図 2.4 のコード（&z=r3）

```
1        a = y+z;
2        x = a*a+z;
```

図 2.6: 複数の代入文からなるプログラム

```
1        ldfd f1 = [r2]
2        ldfd f2 = [r3]
3        nop
4        nop
5        fadd f3 = f1,f2
6        nop
7        stfd [r4] = f3
8        fmpy f4 = f3,f3
9        nop
10       fadd f5 = f4,f2
11       nop
12       stfd [r1] = f5
```

図 2.7: 図 2.6 のコード（&a=r4）

2.2　命令の並列実行

命令並列度が 2 であるようなプロセッサを考える。このプロセッサでは最大 2 個の命令を同時に発行できるが、その 2 個は以下のように 1 行にセミコロン”;”で区切って表記するものと約束する。

```
inst1;          inst2
```

そうすると図 2.6 のプログラムに対応するコードは図 2.10 のようになる。このコードには多数の nop が現れているが、そのことは、単純に言えば、このコードについてプロセッサが効率的に動作しないことを意味する。

命令並列度が 3 以上の場合も同様に考えることができる。

問題　図 2.11 のプログラムを命令並列度 2 でコード化せよ。命令のレーテンシ、変数のアドレスは上と同じとする。

```
1        ldfd f1 = [r2]
2        ldfd f2 = [r3]
3        nop
4        nop
5        fadd f3 = f1,f2
6        nop
7        fmpy f4 = f3,f3
8        stfd [r4] = f3
9        fadd f5 = f4,f2
10       nop
11       stfd [r1] = f5
```

図 2.8: 図 2.7 のコードよりも 1MC だけ高速なコード

```
1        a = y*(y+z);
2        y = y+z;
```

図 2.9: 共通部分式を持つプログラム

```
1        ldfd f1 = [r2];    ldfd f2 = [r3]
2        nop;               nop
3        nop;               nop
4        fadd f3 = f1,f2;   nop
5        nop;               nop
6        stfd [r4] = f3;    fmpy f4 = f3,f3
7        nop;               nop
8        fadd f5 = f4,f2;   nop
9        nop;               nop
10       stfd [r1] = f5;    nop
```

図 2.10: 図 2.6 のコード（命令並列度 2 の場合）

2.3　条件文のコード生成

図 2.12 は条件分岐を含むＣプログラムである。これに対応するコードは図 2.13 である。ただし、命令のレーテンシは以下のように仮定する。

命令	レーテンシ（MC）
ldfd	3
stfd	1
fadd/fmpy	2
fcmp	2

命令並列度は 1 とし、浮動小数点定数 0.0 は f0 に事前に格納されていると仮定する。変数 x、y、

```
1          x =(y+z)*(y+a)+z;
```

図 2.11: 複数の演算を含む代入文

```
1        if(y > 0.0){
2            x = z;
3        }else{
4            x = y*y;
5        }
```

図 2.12: 条件文を含むプログラム

```
1         ldfd f1 = [r2]
2         nop
3         nop
4         fcmp.<= f1,f0
5         nop
6         br.cond L1
7         ldfd f2 = [r3]
8         nop
9         nop
10        stfd [r1] = f2
11        br L2
12 L1:    fmpy f3 = f1,f1
13        nop
14        stfd [r1] = f3
15 L2:
```

図 2.13: 図 2.12 のコード

z のアドレスを格納している GR は前節までと同じとする。ここに、4 行目で比較演算を行い、6 行目で比較結果に基づく分岐を行っている。

第3章　基本ブロック

分岐を含まないプログラム片を**基本ブロック**（basic block）と呼ぶ。たとえば図 2.4 は基本ブロックである。また複数の代入文からなるプログラム片（図 2.6、図 2.9）も基本ブロックである。

しかし図 2.12 のプログラムは条件付き分岐を含むから基本ブロックではない。図 3.1 も基本ブロックではない。ただしループの本体は基本ブロックである。図 3.2 も基本ブロックではない。しかもループの本体も基本ブロックではない。

まず、この授業で対象とするのは、図 2.4、2.6、2.9 のような基本ブロックである。基本ブロックの最適化を通して RISC プロセッサおけるコード最適化の基本技術を学ぶ。

次に、図 3.1 のような、ループの本体が基本ブロックであるようなループ・プログラムを論じる。この話がこの授業のメインであり、この授業の内容の 50%以上を占める。

そして最後に図 3.2 のような、ループの本体に条件文が含まれるようなプログラムを論じる。

```
1       for(i = 0; i < 100; i++){
2           x[i] = z[i];
3           y[i] = a+z[i];
4       }
```

図 3.1: for ループ

```
1       for(i = 0; i < 100; i++){
2           if(y > 0.0){
3               x = z;
4           }else{
5               x = y*y;
6           }
7       }
```

図 3.2: 条件文を含む for ループ

第4章　リスト・スケジューリング

2章でも触れたが、プログラムを高速実行させるためには命令の配置（実行順序）を工夫せねばならない。この最適配置の手続きを**命令スケジューリング**（instruction scheduling）と呼ぶ。しかし、最適な（実行時間が最小になる）コードを生成する命令スケジューリング問題は一般にNP困難（NP hard）である（全ての配置を試してみる以外にどのような配置が最速であるか、知ることができない）ため、最適な配置には指数オーダーの計算が必要である。よって、命令数がごく少ない場合を除けば最適解を求めることはあきらめざるを得ず、準最適解を近似的に求めることしかできない。そのような近似解法のひとつとして、以下に紹介する**リスト・スケジューリング**（list scheduling）が広く使用されている。

リスト・スケジューリングを用いる命令スケジューリングは以下の二つのステップからなる。

1. データ依存グラフを作成する。
2. データ依存グラフを基にしたリスト・スケジューリングを行う。

以下に詳細を説明する。

4.1　データ依存グラフ

データ依存グラフ（data dependence graph）とは、命令を頂点（またはノード（node）と呼ぶ）とし、命令間のデータ依存を枝とする有向グラフである。たとえば図4.1のプログラムについては図4.2のようなデータ依存グラフとなる。ただし、プログラム中の定数2.0の値は事前に`f1`に格納されているとし、変数`x`、`y`、`z`のメモリ・アドレスは`r1`、`r2`、`r3`に格納されているとする。このグラフでは、実際に実行すべき命令に対応する頂点は角のない長方形で表した。定数に相当する頂点は角のある長方形で表した（図4.2では`f1`のみが該当）。

さて、頂点n_1から頂点n_2への枝があるとき、n_1をn_2の**先行頂点**（predecessor node）と呼び、n_2をn_1の**後続頂点**（successor node）と呼ぶ。図4.2では、頂点L1が頂点M1の先行頂点である。頂点`A1`は頂点M1の後続頂点である。

頂点n_1から頂点n_2への枝があるとき、n_1に対応する命令のレーテンシをこの枝の**重み**（weight）と定義する。図4.2にはレーテンシを以下のように仮定した重みを付けた。

```
1        z = 2.0*x+y;
```

図 4.1: リストスケジューリングのための例題プログラム（その 1）

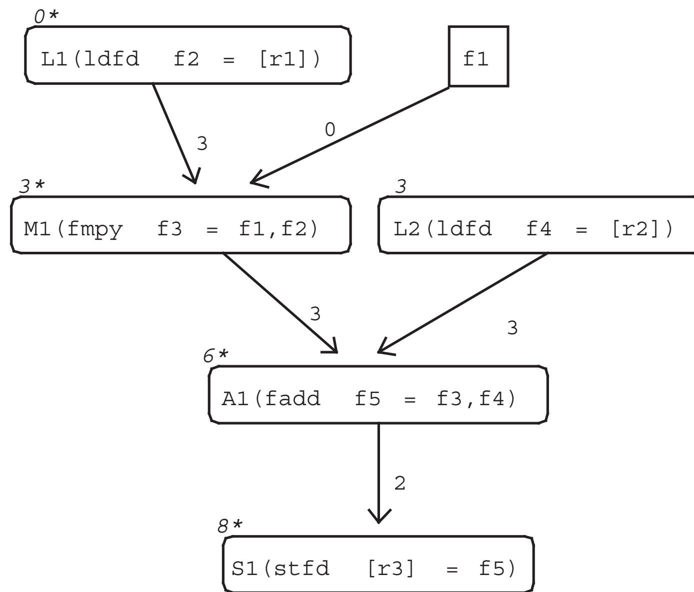

図 4.2: データ依存グラフ（その 1）

命令	レーテンシ（MC）
ldfd	3
stfd	1
fmpy	3
fadd	2

図 4.2 の頂点 f1 から頂点 M1 への枝の重みが 0 となっているのは、f1 の値は事前にレジスタに格納されており、レーテンシが 0 と見なせるからである。

ある頂点 n_1 から頂点 n_2 へのパス（経路）の重みを、そのパス上の枝の重みの総和と定義する。グラフの全てのパスの中で最も大きな重みを持つパスを**クリティカル・パス**（critical path）と呼ぶ。図 4.2 の場合、L1 → M1 → A1 → S1 がクリティカル・パスであり、その重みは 8 である。クリティカル・パスの重みはプロセッサの命令並列度が無限大のときのプログラムの最短実行時間に等

```
1        b = a+3.0;
2        d = b*c+(b+4.0);
3        f = d+e;
```

図 4.3: リストスケジューリングのための例題プログラム（その 2）

しく、基本ブロックのコードの質を評価する際の大きな目安となる。

図 4.1 のプログラムは簡単すぎて、命令スケジューリングの難しさが見えてこない。そこで図 4.3 の例題も検討しよう。ここに定数 3.0、4.0 は既に `f1`、`f2` に格納されているとし、変数 `a`、`b`、`c`、`d`、`e`、`f` のアドレスは `r1`、`r2`、`...`、`r6` に格納されているとする。図 4.4 のクリティカル・パスは `L1` → `A1` → `M1` → `A3` → `A4` → `S3` であり、その重みは 12 である。

なお、データ依存グラフには閉路が存在しないことを注意しておく。閉路は演算が循環することを意味するが、そのような循環計算は基本ブッロク内にはありえない。

4.2　リスト・スケジューリング（その 1）

リスト・スケジューリングは、コード最適化独自の技法ではなく、OR（Operations Research）の一般的な最適化処理技法として知られているものである。マルチ・プロセッサによるタスク処理などにも応用されている。この技法はそのまま命令スケジューリングにも適用可能である。

早速、そのアルゴリズムを命令スケジューリングに応用したアルゴリズムを示す。

1. プロセッサの命令並列度を d とするとき、横に d 列、縦には下に向かって半無限長の空（から）の表を用意する。半無限とは言え、実際にはクリティカル・パスの重みの数倍の行を用意すれば十分である。この表を**リソース予約表**（resource reservation table）と呼ぶ。たとえば、$d = 1$ のときは以下のような表である。

1	
2	
3	
...	

$d = 2$ の場合は以下の通りである。

1		
2		
3		
...		

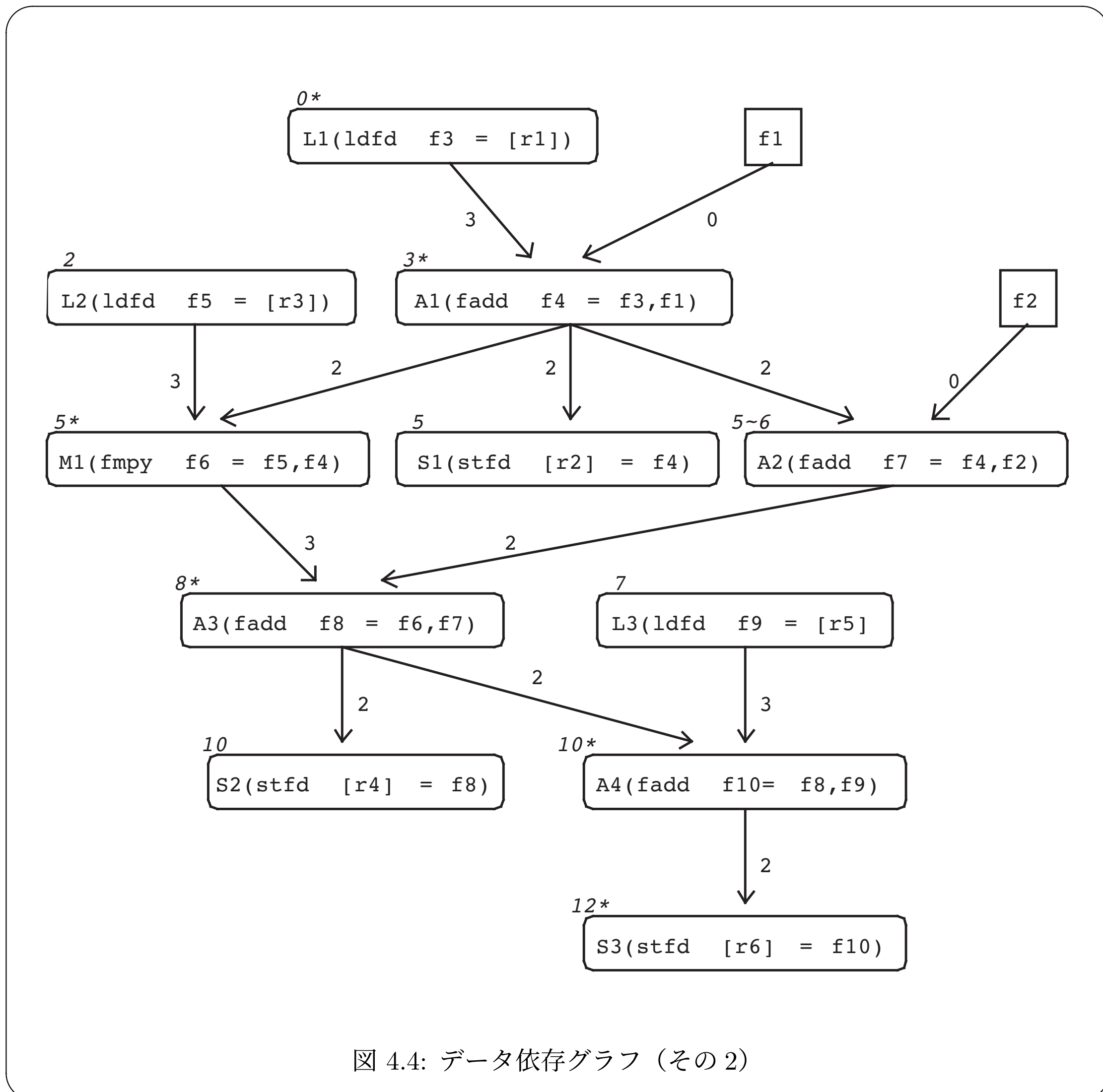

図 4.4: データ依存グラフ（その 2）

ここに最左列の 0、1、2、. . . は、命令を発行する時点（時刻）を表す。表の各エントリは現在のところ空であるが、以下の手順の中で順次、命令を埋め込んでいく。

2. データ依存グラフ G の全ての頂点について、ある基準に基づく優先度をあらかじめ求めておく。

3. グラフ G の中から先行頂点を持たない頂点を全て集め、集合 S とする。これをスケジューリング候補集合と呼ぶ。

4. 候補集合 S の中で最も優先度の高い頂点 n を取り出す。

5. 頂点 n の命令を発行できる最も早い時点を求め、リソース予約表の中の該当する時点（表中の対応する行位置）にその命令を埋める。この際、以下の点に留意する。

(a) プロセッサ・ストールを起こしてはいけない。

(b) 求めた時点に既に d 個の命令が埋まっているときには、そこには n を埋めない。その時点から後方の時点（表の下方）へ向かってエントリが命令で埋まっていない時点を探し、最初に見つかった空の時点に埋める [1]。

6. 頂点 n をグラフ G から取り除く。

7. もし G が空ならば、スケジューリングは終了。さもなくば 3. へ戻る。

上のアルゴリズムを実行するには、「ある基準に基づく優先度」を定めねばならない。この定め方には種々考えられるが、命令スケジューリングにおいて妥当と思われる優先度の定義は以下のようなものである。まず初めに優先度パラメータをグラフ上で以下のように求めていく。

I. 先行頂点の優先度パラメータを p、先行頂点のレーテンシを L とするとき、当該頂点の優先度パラメータは $p+L$ 以上であり、可能な限り小さな値が望ましい。

II. 後続頂点の優先度パラメータを p、当該頂点のレーテンシを L とするとき、当該頂点の優先度パラメータは $p-L$ 以下であり、可能な限り大きな値が望ましい。

III. 優先度パラメータ値の小さな頂点ほど優先度が高い。複数の頂点が同じ優先度パラメータを持つ場合にはクリティカル・パス上の頂点が優先度が高い。

図 4.2、図 4.4 の頂点の左肩に付いている斜体の数値が、この方法によって定めた優先度パラメータである。

上の規則 I.、II. は頂点間の相対的なパラメータ値しか規定しない。そこで、図 4.2、図 4.4 では頂点 `L1` の数値を 0 と設定した。実際、優先度を決めるには相対的な大小関係のみで十分であり、各頂点の絶対値は問題とならない。星印*の付いたものは、その頂点がクリティカル・パス上にあることを表す。

ここで図 4.4 の頂点 `A2` に注目したい。この頂点の優先度パラメータは 5〜6 となっている。これは、規則 I.、II. だけでは優先度パラメータが一意に定まらないためである。このような状況は

- データ依存グラフが木構造でない場合に、
- クリティカル・パス上にない頂点において

発生する可能性がある。規則 I.、II. の中の「望ましい」とはまさにこのような状況を想定した表現である。さらに言えば、このような頂点の優先度パラメータを厳密に定める必要のないことを示している。実際、このような頂点はクリティカル・パス上にないため、その頂点の扱いが実行

[1] 表は半無限長であるから、必ずそのような時点を見つけることができる。

表 4.1: 図 4.2 のスケジューリング過程（命令並列度 1）

MC	1 回目	2 回目	3 回目	4 回目	5 回目
1	L1	L1	L1	L1	L1
2			L2	L2	L2
3					
4		M1	M1	M1	M1
5					
6					
7				A1	A1
8					
9					S1

性能に深刻な影響を与えることは少ない。そこで以下では頂点 A2 の優先度パラメータは平均値 $(5+6)/2 = 5.5$ とする。

　ところで、任意のデータ依存グラフを入力して、上の規則 I.、II. を満たすような優先度パラメータを自動的に計算していくプログラム（アルゴリズム）を作ることは意外に難しい。グラフ上の頂点をどのように辿れば十分か、どのように辿れば効率的か、などを手順化することは意外と骨の折れる仕事である。余裕のある人は検討してみると面白いだろう。

　優先度が求められたならば、リスト・スケジューリングを行おう。まず図 4.2 を例にして考察する。ただし命令並列度 d は 1 とする。以下の表がそのスケジューリング過程である。横の 5 列が、5 個の命令をリソーソ予約表に埋め込んでいく（スケジューリングしていく）過程を表している。

　以下に詳細に説明する。

1 回目　最初のスケジューリング候補集合は {L1,L2} である。ここに L1 と L2 では L1 の優先度が大きい（優先度パラメータ値が小さい）から、頂点 L1 を選択する。L1 を発行できる最も早い時点は最上位行の 1 MC 目であるから、そこに L1 を埋める。そしてグラフから L1 を削除し、L1 から M1 への枝も削除する。これによって M1 の先行頂点はなくなる。

2 回目　次の候補集合は {M1,L2} である。このとき優先度パラメータ値は M1、L2 共に 3 であるが、M1 がクリティカル・パス上にあるため、M1 を選択する。頂点 L1 が 1MC 目で発行されているから、M1 がプロセッサ・ストールを起こさずに発行できる最も早い時点は 4MC（=1MC+3MC）である。よって 4MC 目の行に M1 を埋め、そしてグラフから M1 を削除し、M1 から A1 への枝も削除する。

3 回目　次の候補集合は {L2} である。L2 はもともと先行頂点を持たないから、L1 同様に、L2 が発行可能な最も早い時点は 1MC 目である。しかしそこは既に L1 で埋まっているから、その次

```
1        ldfd f2 = [r1]
2        ldfd f4 = [r2]
3        nop
4        fmpy f3 = f1,f2
5        nop
6        nop
7        fadd f5 = f3,f4
8        nop
9        stfd [r3] = f5
```

図 4.5: 表 4.1 を書き下したコード

の時点の 2MC 目に L2 を埋める。そしてグラフから L2 を削除し、L2 から A1 への枝も削除する。

4 回目 次の候補集合は {A1} である。A1 はもともと M1 と L2 を先行頂点としているから、A1 の発行できる時点はこの二つに依存する。まず M1 は 4MC の時点で発行されるから、それに fmpy のレーテンシ 4MC の加えた 7MC 目が、M1 から定まる最も早い発行時点となる。次に、L2 は 2MC 目に発行されるから、それに ldfd のレーテンシ 3MC の加えた 5MC 目が、L2 から定まる最も早い発行時点となる。この二つの制約から A1 は 7MC 目以降でなければ発行できない。リソース予約表の 7MC 目は空いているから、そこに A1 を埋める。そしてグラフから A1 を削除し、A1 から S1 への枝も削除する。

5 回目 最後の候補集合は {S1} である。fadd のレーテンシは 2MC であるから、S1 は A1 の発行時点から 2MC 以上後に発行せねばならない。そこで、リソース予約表の 9MC（=7MC+2MC）の時点に S1 を埋める。そしてグラフから S1 を削除する。以上でグラフは空になったから、スケジューリングを終了する。

さて、表 4.1 の 5 回目の結果をコードとして書き下したものは図 4.5 である。ただし、命令の埋められていないエントリには nop が存在すると考える。図 4.2 のクリティカル・パスの重みは 8 であり、コードは 1MC に実行を開始し、9MC に実行を終了しており、

$$9\text{MC} - 1\text{MC} = 8\text{MC}$$

であるから、このコードは最速のコードである（これよりも高速なコードは原理的に生成できない）。

同様に図 4.4 についてもリスト・スケジューリングを行おう。命令並列度を 1 としてスケジューリングする過程は表 4.2 の通りである。図 4.4 のクリティカル・パスの重みは 12 であり、表 4.2 のコードの実行時間は 14MC − 1MC = 13MC であるから、表 4.2 のコードは最速なコードよりも

表 4.2: 図 4.4 のスケジューリング過程（命令並列度 1）

MC	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1
2		L2	L2	L2	L2	L2	L2	L2	L2	L2	L2
3							L3	L3	L3	L3	L3
4			A1	A1	A1	A1	A1	A1	A1	A1	A1
5											
6				M1	M1	M1	M1	M1	M1	M1	M1
7					S1	S1	S1	S1	S1	S1	S1
8						A2	A2	A2	A2	A2	A2
9											
10								A3	A3	A3	A3
11											
12									A4	A4	A4
13										S2	S2
14											S3

1MC だけ遅いコードかもしれない。実際、別の考察（これについては後述する）によれば表 4.2 のスケジューリング結果は最速ではない。リスト・スケジューリングは近似アルゴリズムであるから、常に最適なスケジューリングができる訳ではないことに注意してほしい。

命令並列度が 2 の場合も検討してみよう。スケジューリングの過程を表 4.3 に示す。ただし、8 回目から 10 回目のステップは省略した。表 4.3 のコードは 1MC〜13MC に配置されており、13MC − 1MC = 12MC はクリティカル・パスの重みの重みに等しいから最速なコードである。具体的にコードを書き下すと図 4.6 の通りである。

ところで表 4.2 （命令並列度 1）では、L3 が必要以上に早い時点（3MC 目）に配置されていることに気づく。L3 の結果が使用される時点は 12MC 目の A4 であるから、L3 は 12 − 3 = 9MC 以前に発行されていれば十分であるにも関わらず、この場合、6MC（=9MC−3MC）も早く発行していることになる。プロセッサ・ストールさえ起こさないならば、早いタイミングで命令を発行してもコードの実行速度には全く影響しないが、レジスタにデータが保持されている期間が無駄に長くなることになる。これは実用上は好ましくない。

同様のことは表 4.3 でも言える。図 4.6 のコード中の f9 は、それをターゲットレジスタとする 2MC 目の命令 L3 （= ldfd f9 = [r5]）が発行されてから、それをソースレジスタとする 11MC 目の命令 A4（= fadd f10 = f8,f9）が発行されるまでの 9MC の間、使用されている。f9 は 9MC − 3MC =6MC の間、無駄に値を保持していることになる。

これを改善する方法を次に述べる。

表 4.3: 図 4.4 のスケジューリング過程（命令並列度 2）

	#1	#2	#3	#4	#5	#6	#7	...	#11
1	L1	L1　L2	L1　L2	L1　L2	L1　L2	L1　L2	L1　L2	...	L1　L2
2							L3	...	L3
3								...	
4			A1	A1	A1	A1	A1	...	A1
5								...	
6				M1	M1　S1	M1　S1	M1　S1	...	M1　S1
7						A2	A2	...	A2
8								...	
9								...	A3
10								...	
11								...	A4　S2
12								...	
13								...	S3

4.3　リスト・スケジューリング（その 2）

改善策の考え方は、命令スケジューリングの順番を 4.2 節の方法とは逆順にすることである。そのために、まず優先度の定義を以下のように変更する。下線部が変更点である。

III'. 優先度パラメータ値の大きな頂点ほど優先度が高い。複数の頂点が同じ優先度パラメータを持つ場合、クリティカル・パス上の頂点ほど優先度が高い。

さらにアルゴリズムを以下のように変える。

1. プロセッサの命令並列度を d とするとき、横に d 列、縦には上に向かって半無限長の空（カラ）のリソース予約表を用意する。たとえば、$d = 1$、$d = 2$ のときの表は以下の通り。

...	
-3	
-2	
-1	

$d = 2$ の場合は以下の通りである。

...		
-3		
-2		
-1		

```
 1       ldfd f3 = [r1];    ldfd f5 = [r3]
 2       ldfd f9 = [r5];    nop
 3       nop;               nop
 4       fadd f4 = f3,f1;   nop
 5       nop;               nop
 6       fmpy f6 = f5,f4;   stfd [r2] = f4
 7       fadd f7 = f4,f2;   nop
 8       nop;               nop
 9       fadd f8 = f6,f7;   nop
10       nop;               nop
11       fadd f10 = f8,f9;  stfd [r4] = f8
12       nop;               nop
13       stfd [r6] = f10;   nop
```

図 4.6: 表 4.3 を書き下したコード

2. データ依存グラフ G の全ての頂点について、ある基準に基づく優先度をあらかじめ求めておく。

3. グラフ G から後続頂点を持たない頂点を全て集め、スケジューリング候補集合 S とする。

4. 候補集合 S の中で最も優先度の高い頂点 n を取り出す。

5. 頂点 n の命令を発行できる最も遅い時点を求め、表中のその時点にその命令を埋める。この際、以下の点に留意する。

   (a) プロセッサ・ストールを起こしてはいけない。

   (b) もしその時点に既に d 個の別の命令が埋まっているときには、そこには n を埋めない。そこから前方の時点（表の上方）へ向かって空いている時点を探し、最初に見つかった空の時点に埋める。

6. 頂点 n をグラフ G から取り除く。

7. もし G が空ならば、スケジューリングは終了。さもなくば 3. へ戻る。

図 4.4 について命令並列度 1 で実際にスケジューリングしよう。

表 4.4 がそのスケジューリング過程である。表 4.2 と比較されたい。表 4.4 ではロード命令 `L3` が必要以上に早い時点で発行される問題は解決されている。しかし逆にストア命令 `S1` が必要以上に遅い時点で発行されるという現象が起きていることに気づく。たとえば `S1` は `A1` に依存するが、`A1` と `S1` の間は $|-10\mathrm{MC} - -4\mathrm{MC}| = 6\mathrm{MC}$ も離れている。`fadd` のレーテンシは 2MC であるから、

表 4.4: 図 4.4 のスケジューリング過程（命令並列度 1）

MC	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
-13											L1
-12											
-11										L2	L2
-10									A1	A1	A1
-9											
-8							M1	M1	M1	M1	M1
-7						A2	A2	A2	A2	A2	A2
-6					L3	L3	L3	L3	L3	L3	L3
-5				A3	A3	A3	A3	A3	A3	A3	A3
-4								S1	S1	S1	S1
-3		A4	A4	A4	A4	A4	A4	A4	A4	A4	A4
-2			S2	S2	S2	S2	S2	S2	S2	S2	S2
-1	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3

6MC − 2MC = 4MC も無駄にレジスタを使用していることになる。4.2 節のアルゴリズムと本節のアルゴリズムではスケジューリングの方向が逆順であるために、ロード命令とストア命令について逆転した現象が起きるのである。どちらが優れているかは、一概には言えないが、しかし次の経験的事実:

- 多くのコードでは、ストア命令の数よりもロード命令の数が多い。

は、4.2 節よりも本節のアルゴリズムを支持すると考えられる。何故ならば、4.2 節のアルゴリズムはロード命令のターゲットレジスタの使い方に無駄が多い。逆に、本節のアルゴリズムはストア命令のソースレジスタの使い方に無駄が多い。よって、本節のアルゴリズムの方が無駄が少ない、と予想されるからある。これはレジスタ数に関する予想であるが、実行時間についてどちらが優れているかは不明である。

アルゴリズムにさらなる改良を加えることもできるが、煩雑なのでこの授業ではこれ以上、深入りしない。

gcc を含め、最近のほとんどのコンパイラが本章のアルゴリズムを実装している。基本ブロックの命令スケジューリングはコード最適化におけるごく一般的な処理である。

問題　図 4.4 のグラフについて命令並列度 $d = 2$ で本章（4.3 節）のアルゴリズムを適用し、スケジューリング結果を示せ。

第5章　基本ブロックのレジスタ割付け

この授業で想定するプロセッサは多数のレジスタを有するから、レジスタ割付け（コード中に使用するレジスタ数をできるだけ少なくする最適化）に気をつかう必要はないと述べてきた。しかし、実用上は重要な事項であるから、この章で前章の図 4.6 のコードを題材にレジスタ割り付けの基本的な考え方を紹介する。このコードでは `f1`、`f2` の 2 個のレジスタは定数 3.0 と 4.0 を格納するために使用されている。これには手をつけない。`f3`〜`f10` の 8 個のレジスタは計算結果を一次的に保持するために使用されている。この 8 個のレジスタを共通化し、使用レジスタ数を減らすことが目的である。

5.1　仮想レジスタ

レジスタ割付けの準備として、プロセッサには無限個のレジスタが存在すると一次的に仮定し、それらレジスタを**仮想レジスタ**（virtual register）と呼ぶ。さらに、個々の仮想レジスタは、コード中で高々1 回しかそのレジスタに値が代入されない（高々1 回しかターゲットにならない）と約束する。そのような仮想レジスタを `v1`、`v2`、... で表わそう。なお、仮想レジスタの対語は実レジスタ（real register、actual register）である。

図 4.6 のコードにおいて `f3`〜`f10` をそれぞれ `v3`〜`v10` に置き換えたコードを図 5.1 とする。仮想レジスタ番号 `v1`、`v2` は用いない。

5.2　生存区間

ひとつの仮想レジスタ v について、

- v をターゲットレジスタする命令が発行された時点 t_s <u>の直後</u> から
- v をソースレジスタとする命令群（複数の命令がありうる）の最後の命令が発行された時点 t_e まで

を、v の**生存区間**（live range、live interval）と呼び、

$$v = (t_s, t_e]$$

```
 1        ldfd v3 = [r1];    ldfd v5 = [r3]
 2        ldfd v9 = [r5];    nop
 3        nop;               nop
 4        fadd v4 = v3,f1;  nop
 5        nop;               nop
 6        fmpy v6 = v5,v4;   stfd [r2] = v4
 7        fadd v7 = v4,f2;   nop
 8        nop;               nop
 9        fadd v8 = v6,v7;   nop
10        nop;               nop
11        fadd v10 = v8,v9;  stfd [r4] = v8
12        nop;               nop
13        stfd [r6] = v10;   nop
```

図 5.1: 図 4.6 のレジスタを仮想化したコード

と表わす。なお、以下では t_s を生存区間の開始時点、t_e を生存区間の終了時点と呼ぶ。開始時点 t_s は生存区間に含まれないことに注意する。たとえば、図 5.1 のコードに含まれる仮想レジスタの生存区間は以下の通りである。

$$\mathtt{v3} = (1,4], \quad \mathtt{v4} = (4,7], \quad \mathtt{v5} = (1,6], \quad \mathtt{v6} = (6,9],$$
$$\mathtt{v7} = (7,9], \quad \mathtt{v8} = (9,11], \quad \mathtt{v9} = (2,11], \quad \mathtt{v10} = (11,13]$$

1 これ以降、仮想レジスタとその生存区間を同一視する。

2 二つの仮想レジスタ v、v' について、それらの生存区間が重なるとき、v と v' は互いに**干渉**
3 (interfere)するという。たとえば上の例の v3 と v9 は互いに干渉する。v3 と v4 は干渉しない。
4 ここに $\mathtt{v3} = (1,4]$ の終了時点と $\mathtt{v4} = (4,6]$ の開始時点は共に時点 4 であるが、区間としては重
5 ならないことを注意する。

互いに干渉しない仮想レジスタのみを含む集合を、仮想レジスタの非干渉集合と呼ぼう。たとえば上の例では以下は非干渉集合である。

$$\{\mathtt{v3}, \mathtt{v4}, \mathtt{v7}, \mathtt{v8}, \mathtt{v10}\}$$

6 与えられた仮想レジスタ(とその生存区間)の集合 $\{v_1, ..., v_m\}$ を非干渉集合に分割することを
7 **レジスタ割付け**(register allocation)と呼ぶ。非干渉集合にはひとつの(仮想ではない)実レジス
8 タを割り付けれることができる。

たとえば、例では

$$\{\mathtt{v3}, ..., \mathtt{v10}\} = \{\mathtt{v3}, \mathtt{v4}, \mathtt{v7}, \mathtt{v8}, \mathtt{v10}\} \cup \{\mathtt{v5}, \mathtt{v6}\} \cup \{\mathtt{v9}\}$$

```
 1        ldfd f3 = [r1];    ldfd f4 = [r3]
 2        ldfd f5 = [r5];    nop
 3        nop;               nop
 4        fadd f3 = f3,f1;   nop
 5        nop;               nop
 6        fmpy f4 = f4,f3;   stfd [r2] = f3
 7        fadd f3 = f3,f2;   nop
 8        nop;               nop
 9        fadd f3 = f4,f3;   nop
10        nop;               nop
11        fadd f3 = f3,f5;   stfd [r4] = f3
12        nop;               nop
13        stfd [r6] = f3;    nop
```

図 5.2: 図 5.1 のコードに 3 個の実レジスタを割り付けた場合

あるいは

$$\{\mathtt{v3}, ..., \mathtt{v10}\} = \{\mathtt{v3}, \mathtt{v4}, \mathtt{v7}, \mathtt{v8}\} \cup \{\mathtt{v5}, \mathtt{v6}\} \cup \{\mathtt{v9}, \mathtt{v10}\}$$

などの三つの非干渉集合への分割が可能である。このことから、図 5.1 のコードは高々三つのレジスタを使用するだけで十分であることが分かる。たとえば上の最初の例の分割の場合には、集合 $\{\mathtt{v3}, \mathtt{v4}, \mathtt{v7}, \mathtt{v8}, \mathtt{v10}\}$ に実レジスタ f3 を割り当て、$\{\mathtt{v5}, \mathtt{v6}\}$ に f4 を、$\{\mathtt{v9}\}$ に f5 を割り当てれば、図 5.1 は図 5.2 のようにレジスタ割り付けができる。もちろん自明な分割

$$\{\mathtt{v3}, ..., \mathtt{v10}\} = \{\mathtt{v3}\} \cup \{\mathtt{v4}\} \cup ... \cup \{\mathtt{v10}\}$$

もありうる。しかし、我々が欲しいのはできるだけ少ない数の非干渉集合への分割である。そのような分割を最適なレジスタ割り付けと呼ぼう。

5.3　生存区間グラフ

さて、最適なレジスタ割り付けのアルゴリズム（仮想レジスタ集合を最小数の非干渉集合に分割するアルゴリズム）を検討しよう。それにはグラフを用いる方法が分かりやすい。

図 5.3 は、時間軸を垂直方向に取り、各仮想レジスタの生存区間を線分で図示したものである。ここに、慣習に従い、白丸 ○ はその時点を含まないことを表わす。黒丸 ● はその時点を含むことを表わす。

次にこの図を水平方向に眺め、時点と時点の境界に交差する生存区間の個数を調べる。たとえば、時点 1 と時点 2 の境界に交差する生存区間は、仮想レジスタ v3 と v5 の 2 個である。時点

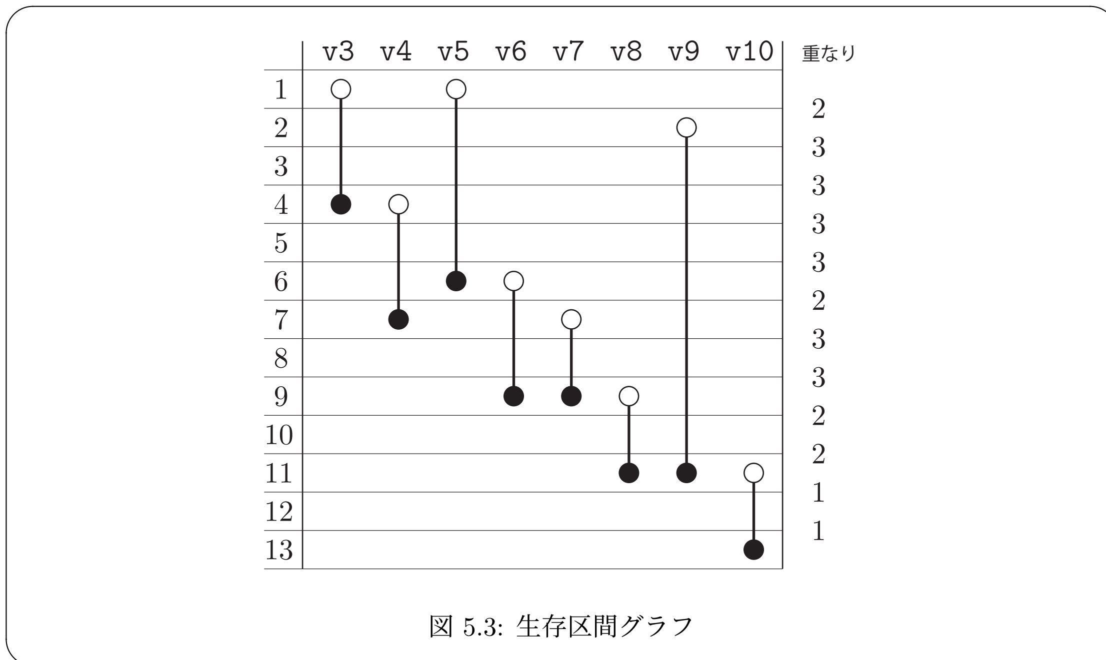

図 5.3: 生存区間グラフ

2 と時点 3 の境界を縦断する生存区間は、仮想レジスタ v3、v5、v9 の 3 個である。このような生存区間の個数をその時点の境界における区間の重なり数と呼ぼう。重なり数は、その瞬間に必要な実レジスタの数を表わしている。よってコード全体について重なり数の最大値 R を求めるならば、それがそのコードで最低限、必要なレジスタ数である。基本ブロックの場合には実際に R 個のレジスタで割付けできる [1]（基本ブロックでない場合にはそう単純に行かない）。図 5.3 の重なりの最大値は 3 である。よって 3 個のレジスタがあればよいことになる。実際、図 5.2 で見た通りである。

グラフを用いたレジスタ割り付けのアルゴリズムは以下の通りである。仮想レジスタの生存区間の集合 $V = \{v_1, ..., v_N\}$ が与えられたとしよう。このとき、

1. V の中から、最も早い開始時点を持つ仮想レジスタ v をひとつ選び、

$$V = V - \{v\}$$
$$V' = \{v\}$$

   とする。もしそのような v が複数存在するならば、その中の適当なひとつを選ぶ。

2. v の終了時点を t_e とする。

3. V の中に、その開始時点 t'_s が t_e 以上である（つまり、$t_e \leq t'_s$ である）ような仮想レジスタ v' が存在しないならば、6. へ行く。

[1] 余裕のある人は証明してみなさい。

4. 上記 2. の v' のようなレジスタの中で、終了時点と開始時点の差 $t'_s - t_e$ が最小であるような仮想レジスタ $\tilde{v'}$ をひとつ選び、

$$\begin{aligned} V &= V - \{\tilde{v'}\} \\ V' &= V' \cup \{\tilde{v'}\} \end{aligned}$$

とする。もしそのような $\tilde{v'}$ が複数存在するならば、その中の適当なひとつを選ぶ。

5. $\tilde{v'}$ の終了時点を新たに t_e とし、3. へ戻る。

6. V' に含まれる全ての仮想レジスタにひとつの実レジスタ番号を割り当てる。

7. V が空（から）でないならば、1. へ戻る。

上のアルゴリズムの手順 1. においてひとつの仮想レジスタ v を選んだ後、それに続くレジスタ $\tilde{v'}$ が順次選ばれていく。これを仮想レジスタの鎖とみなすとき、図 5.4、図 5.5 は、上の方法で作った仮想レジスタの鎖の例である。これらの図では、仮想レジスタ間の終了時点と開始時点を点線で結び、鎖を表現した。上のアルゴリズムでは、1.、4. で仮想レジスタの選び方に任意性が残るため、鎖の作り方は一意ではなく、図 5.4、図 5.5 のように複数の鎖の作り方がありうる。しかしそれは本質的ではなく、必ず必要レジスタを最小にする割付けができる。しかも、上のアルゴリズムの手数は仮想レジスタの数 N に比例するから、計算量は高々、$O(N)$ である。

計算量が N の線形オーダーである点は重要である。一般にこの種の最適化問題は NP 困難な問題、すなわち計算量は指数オーダーとなることが多いが、基本ブロックの最適なレジスタ割り付け問題は例外的に線形オーダーの計算量である。後章においてループ・コードのレジスタ割り付け問題に触れるが、最適化は案の定、指数オーダーの計算量である。さらにその後に、rotating register を用いた場合のループ・コードのレジスタ割り付け問題に触れるが、その計算量は一転して線形オーダーの計算量となる。これは理論的に非常に興味深い性質である。

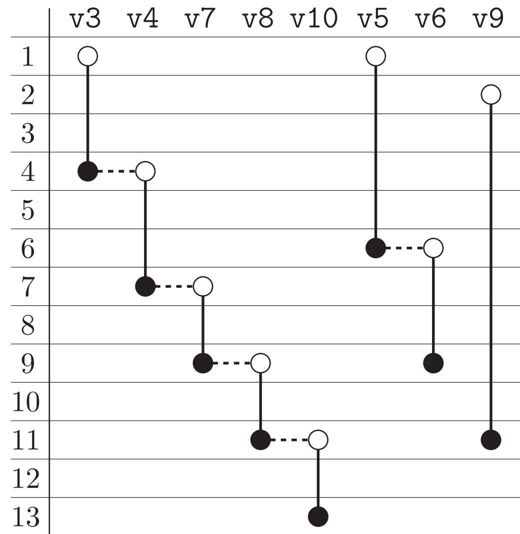

図 5.4: 生存区間グラフによるレジスタ割付け例（その 1）

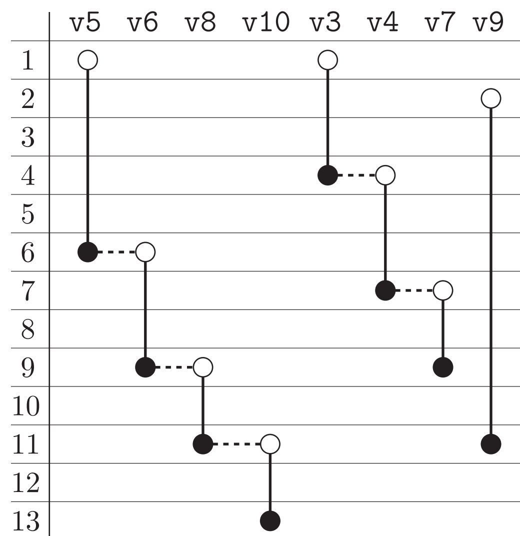

図 5.5: 生存区間グラフによるレジスタ割付け例（その 2）

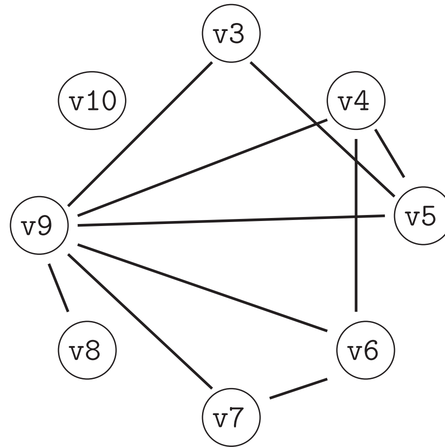

図 5.6: 仮想レジスタの干渉グラフ

5.4　グラフ彩色

上ではレジスタ割り付けを生存区間グラフを用いて理解した。しかし歴史的には、あるいは一般化された問題としては、レジスタ割り付け問題は**グラフ彩色問題**（graph coloring problem）の応用例として長らく研究されてきた。

グラフ彩色問題とは、無向グラフの頂点に色（赤や青など）を割り当てる問題である。ただし、枝で結ばれた 2 頂点に同じ色を割り当ててはいけない。この条件の下で、できる限り少ない数の色でグラフの頂点を塗り分けることを目的としている [2]。

今、仮想レジスタをグラフの頂点とし、仮想レジスタの生存区間が互いに干渉するときに頂点間に枝を張るとする。このようにして作られたグラフを**レジスタ干渉グラフ**（register interference graph）と呼ぶ。そうすると、レジスタ割り付け問題はグラフ彩色問題に置換できる。図 5.6 は、我々の例題について干渉グラフを作成したものである。

グラフ彩色問題の解き方は種々研究されているが、最も有効なアイディアは、より多くの枝を持つ頂点から順に彩色していく方法である。たとえば図 5.6 の場合、頂点 `v9` が 6 本の枝を持ち、最も多い。そこで、`v9` に実レジスタ `f3` を割り付けよう。次に枝が多いのは　`v4`、`v5`、`v6` であり、それぞれ 3 本である。そこで、この順番に、`v9` に既に割り当てられた `f3` と衝突しないように（`v9` と枝で結ばれている場合には `f3` を割り当てないように）注意しながら、実レジスタ `f4`、`f5` を割り当てて行く。これを繰り返すと、たとえば図 5.7 のような実レジスタの割り当てが可能である。この場合、三つのレジスタで割り当てが完了しており、必要レジスタ数は前節の割り当てと同じである。

[2]たとえば平面グラフの彩色問題は有名であり、どんな平面グラフも必ず 4 色以下で塗り分けられることが経験上、知られていた。長らく証明できなかった難問であるが、1970 年代にコンピュータを使って証明された。

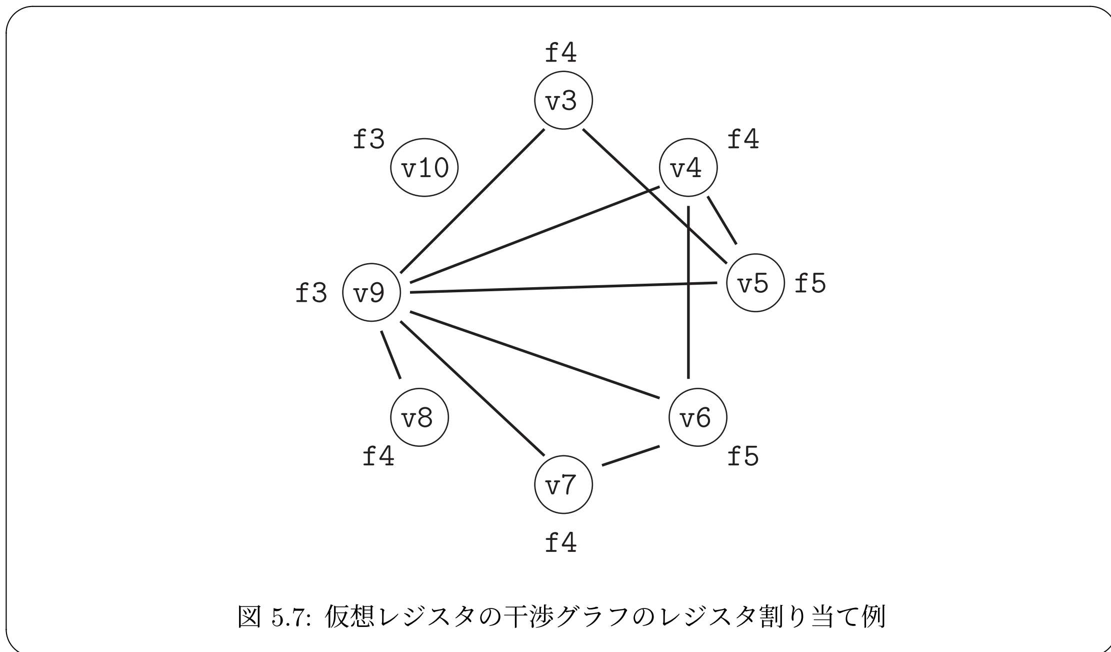

図 5.7: 仮想レジスタの干渉グラフのレジスタ割り当て例

最適なグラフ彩色問題（頂点への彩色数を最小にする問題）は、一般には NP 困難な問題であることが知られており、頂点が増えるとたちまち最適な彩色（レジスタ割付け）が困難になる。それに対して、前節の生存区間グラフを用いる方法では線形オーダーの計算時間で割り当てが可能であると述べた。この違いが生じる理由は、

1. 彩色による方法では、単に仮想レジスタ間に干渉があるか否かの情報しか用いていないのに対して、

2. 生存区間グラフの方法では、各仮想レジスタの生存区間の開始時点と終了時点の情報を用いており、

後者の方法がより多くの情報を用いているため、少ない計算量で精度の高い割り当てが可能なのである。

第6章　ループ・プログラムのコード生成

図 6.1 は典型的なループプログラムであり、そのループ本体は基本ブロックである。このプログラムに対応するアセンブリ・コードはどのように作ればよいだろうか。この章では、高速化については当面考慮せず、まずは正しく動作するアセンブリ・コードの定型的な作り方を述べる。

6.1　iteration

標準的な `for` ループではループ変数（ループ・カウンタ）`i` の値を増やしながらループ本体を繰り返し実行する。各 `i` に関するループ本体を **iteration**（iteration の適切な訳語が見つからないので、ここでは英名をそのまま使用する）と呼ぶ。図 6.1 のプログラムの場合、`i = 0`、`1`、`2`、`...`、`99` と 100 個の iteration を実行することになる。

今、ループ変数 `i` の値が汎用レジスタ `r0` に初期値 0 で格納されているとする。そのとき、目的のコードは各 iteration で `r0` の値を `1` だけ増やし、`r0` の値が `100` になったらループの実行を終了するように作ればよい。以下がそのパターンである。

```
L1:     cmp.>=  r0,100    //  条件 r0 >= 100 を判定
        br.cond L2        //  もし条件が成り立てばループを脱出
        ...
        （ここにループ本体を置く。）
        ...
        add r0 = r0,1     //   r0 の値を 1 増やす
        br L1             //  次の iteration のために L1 へ戻る。
L2:
```

しかし上のコードでは分岐命令が 2 箇所に現れる。分岐命令は、それが無条件分岐であっても、しばしばプロセッサのパイプライン実行を乱すため、分岐命令の実行は最小限に抑えるべきである。分岐命令を減らすため、しばしば以下のコード・パターンが使用される。

```
L1:     ...
        （ここにループ本体を置く。）
        ...
        add r0 = r0,1 //   r0 の値を 1 増やす
        cmp.<  r0,100 //  条件 r1 < 100 を判定
        br.cond L1 //  もし条件が成り立てば L1 へ戻る
L2:
```

```
1        for(i = 0; i < 100; i++){
2                z[i] = 2.0*x[i]+y[i];
3        }
```

図 6.1: ループプログラムの例

もちろん前者から後者への変形が可能なのは、ループの本体が必ず 1 回以上実行される保証のあるときである。さもなくば、ループ本体を全く実行しない場合を別途考慮する必要がある。たとえば for 文が

```
for(i = 0; i < n; i++)
```

となっており、n が定数でないときがそうである。そのような場合にはループに入る直前に、そのループが本体を 1 回以上実行するか否か、事前に n の値をチェックすればよい。また for ループではなく、while ループの場合には、この講義の最適化技法を改良せねばならない。

6.2　アドレス値の更新

図 6.1 のプログラムについて、配列 x、y、z の先頭アドレスが r1、r2、r3 に格納されていると仮定しよう。その先頭アドレスは配列要素 x[0]、y[0]、z[0] のアドレスでもある。よってロード命令：

```
ldfd f1 = [r1]
ldfd f2 = [r2]
```

によって x[0]、y[0] の値が f1、f2 へロードされ、

```
stfd [r3] = f3
```

によって f3 の値が配列要素 z[0] へストアされる。

さて、ここでは全てのデータは double 型と想定しているから、x[1]、y[1]、z[1] のアドレスは x[0]、y[0]、z[0] のアドレスにそれぞれ 8 を足した値である。そこで以下の命令群：

```
add r1 = r1,8
add r2 = r2,8
add r3 = r3,8
```

によって x[1]、y[1]、z[1] のアドレスを得ることができる（add 命令については 1 章を参照のこと）。

```
 1 L1:    ldfd f2 = [r1]
 2        ldfd f4 = [r2]
 3        nop
 4        fmpy f3 = f1,f2
 5        nop
 6        nop
 7        fadd f5 = f3,f4
 8        nop
 9        stfd [r3] = f5
10        add r1 = r1,8
11        add r2 = r2,8
12        add r3 = r3,8
13        add r0 = r0,1
14        cmp.<  r0,100
15        br.cond L1
```

図 6.2: 図 6.1 のループプログラムに対応するコード

```
 1 L1:    ldfd f2 = [r1],8 // ！
 2        ldfd f4 = [r2],8 // ！
 3        nop
 4        fmpy f3 = f1,f2
 5        nop
 6        nop
 7        fadd f5 = f3,f4
 8        nop
 9        stfd [r3],8 = f5 // ！
10        add  r0 = r0,1
11        cmp.< r0,100
12        br.cond L1
```

図 6.3: 図 6.2 の改良版

1　以上の実行を繰り返すことで、各 x[i]、y[i]、z[i]（i= 1,2,...,99）のアドレスを順次求め
2 ることができる。
3　ところで、4 章の図 4.1 のプログラムは図 6.1 のプログラムの本体と同じ形状である。このコー
4 ドは、命令並列度を 1、命令のレーテンシを

命令	レーテンシ（MC）
ldfd	3
stfd	1
fmpy	3
fadd	2

10 と仮定するとき、図 4.5 の通りであった（4.2 節参照）。そこで、同じ命令並列度、同じレーテンシ
11 の場合、図 6.1 のプログラムのコードは、図 4.5 を参考にして、図 6.2 のように作ればよい。ただし
12 整数系演算 add、cmp のレーテンシは 1MC とする。分岐命令のペナルティ（6.4 節参照）は 0MC、
13 すなわち分岐によるプロセッサのパイプライン実行の乱れは生じないと仮定する）。
14　ここで 1 章の命令形式

15 - ldfd　fi = [rj],8
16 - stfd　[rj],8 = fi

```
1 L1:    ldfd f2 = [r1],8;   ldfd f4 = [r2],8
2        nop;                nop
3        nop;                nop
4        fmpy f3 = f1,f2;    nop
5        nop;                nop
6        nop;                nop
7        fadd f5 = f3,f4;    add  r0 = r0,1
8        nop;                cmp.<  r0,100
9        stfd [r3],8 = f5;   br.cond L1
```

図 6.4: 図 6.1 のプログラムに対応する命令並列度 2 のコード

を思い出そう。これらは rj の値をアドレスとして使用した直後に rj に 8 を加算する命令形式であった [1]。これを使用すると図 6.2 のコードは図 6.3 のように短縮できる。

この授業では、図 6.3 のような形式をループの定型パターンとして採用する。

なお、このコードのループ制御の命令列 10〜12 行目の実行には 3MC かかり、全体の中でこの部分の占める割合が決して小さくない。しかし、最近の RISC プロセッサではループ制御を 1MC で行うような仕組みが様々用意されており [2]、実際には効率的なループ制御が可能である。この授業では、理解の容易さを優先し、あえて上のような単純な（しかしやや効率の悪い）パターンをそのまま用いる。

6.3　命令並列度が２以上の場合

命令並列度が 2 の場合のループ・コードはたとえば図 6.4 のように作ることができる。命令並列度が 3 以上の場合も同様である。

なお、ループ制御のための三つの命令 `add r0 = r0,1`、`cmp.< r0,100`、`br.cond L1` はコードの末尾三行の第 2 スロット（右側の位置）に配置すると約束する。そうすると第 1 スロット（左側の位置）が空くから、そこにはループ本体の命令を置くことができる。上のコードでは、そこに `fadd f5 = f3,f4`、`stfd [r3],8 = f5` を配置した。さて、このコードには多くの `nop` が現れており、プロセッサ利用効率の観点からは非常に効率の悪いコードである。これらの `nop` をつぶすために次章以降、ソフトウェア・パイプライン化を行う。

[1] ただし命令発行の 1MC 後には rj への 8 の加算が終了しており、後続の命令は直ちに rj を使用可能とする。これは多くのプロセッサでそのように実装されている。

[2] たとえば IA-64 や PowerPC ではループ・カウンタ専用のレジスタやそのレジスタの更新と条件分岐を同時に行う特殊命令などを持つ。

6.4　補足: 分岐ペナルティ

1 章で述べたように、RISC プロセッサは命令をフェッチ、デコード、実行のステップの順にパイプライン的に実行していく。このパイプライン動作と分岐命令は相性が良くない。

今、条件付き分岐命令がデコードされていると仮定しよう。このとき同時に、この分岐命令の直後の命令がフェッチされなければならない。しかし「直後の命令」が何であるかは分岐命令を実行してみなければ分からない。もし条件が成り立ち、分岐するならば、直後の命令とは分岐先の先頭の命令である。もし条件が成り立たず、分岐しないならば、直後の命令とは文字通り分岐命令の直後に置かれている命令である。このいずれの命令が実際の直後の命令に当たるかは分岐命令を実行してみるまで不明であるから、分岐命令をデコードしているときに同時に後続の命令をフェッチすることは不可能である。よって、分岐命令のところでパイプラインが途切れることになり、これがプロセッサの演算性能を落とす原因となる。これを**分岐ペナルティ**（branch penalty）と呼ぶ。またパイプラインが途切れることを「パイプに泡（bubble）が入る」とも言う。

1 章ではフェッチ、デコード、実行の 3 ステップを考えたが、最近のプロセッサではパイプライン・ステップはより細かく分解され、10 ステップ以上持つものもある。そうすると、分岐命令の実行時にパイプラインが 10MC 以上途切れることもあり、分岐ペナルティによる性能低下は深刻である。そこで最近では、分岐命令を実行する直前に実際に分岐するか否かを予測するハードウェア機構も開発されている。分岐予測が当たった場合にはペナルティが発生しない（ようなハードウェア機構になっている）。

たとえば、図 6.3、図 6.4 のようなループ・コードで iteration を N 回繰り返すと仮定しよう。そうすると、N 回の条件付分岐のうち、$N-1$ 回は実際に分岐を行う（この分岐をループバックと呼ぶ）。よって、分岐を行う（はず）と事前に予測しておいたならば、分岐ペナルティが発生する確率は $1/N$ であり、N が大きければ大きいほどペナルティの頻度は減る。そこでこの授業では、分岐予測は常に当たり、分岐ペナルティは発生しないと仮定する。この仮定が成り立たないのは、ループ・プログラム中に if 文（条件分岐）が含まれる場合だが、それについては 13 章以降で論じることとなる。

第7章　ソフトウェア・パイプライン化

図 4.5 のコードは図 4.1 のプログラムを最適に実行するものであった。しかし、それをほぼそのまま引き継ぐ図 6.3 のコードを図 6.1 のプログラムの最適なコードと考えることは間違いである。ソフトウェア・パイプライン化技法を用いることで、さらに高速なコードが生成可能である。

ソフトウェア・パイプライン化の説明法は様々考えられるが、ここではソフトウェア・パイプライン化されたコードをまず例示し、何故それが高速なのかを解説する。そして、その理解の後に、次章においてソフトウェア・パイプライン化コードの生成法を詳細に述べる。

なお、「プロセッサにおける 命令パイプライン」というときのパイプラインはプロセッサ内のハードウェア・パイプラインを意味し、「ループにおける ソフトウェア・パイプライン」とは全く別ものであることを注意しておく。前者は、命令がプロセッサ内でパイプライン的に実行されていくことを意味するが、後者は、ループの iteration がパイプライン的に実行されていくことを意味する。

7.1　iteration のオーバーラップ

まず、命令のレーテンシは以下の通りとする。

命令	レーテンシ（MC）
ldfd	3
stfd	1
fmpy	3
fadd	2
add	1
comp	1
br.cond	1

そして、図 7.1 のコードは、プロセッサの命令並列度が 1 の場合の、図 6.1 のプログラムのソフトウェア・パイプライン化されたコードである。ただし r0 には 0 を、r1、r2、r3 には配列 x、y、z の先頭アドレスを、f1 には定数 2.0 をあらかじめ格納しておく。ここに 0 行〜4 行のコードを**プロローグ部**（prologue part、前処理部）、5 行〜12 行を**カーネル部**（kernel part、定常部）、13 行〜15 行を**エピローグ部**（epilogue part、後処理部）と呼ぶ。直観的には

```
 1        ldfd f2 = [r1],8      // プロローグ部
 2        ldfd f4 = [r2],8
 3        nop
 4        fmpy f3 = f1,f2
 5        add  r0 = r0,1
 6 L1:    ldfd f2 = [r1],8      // カーネル部
 7        fadd f5 = f3,f4
 8        ldfd f4 = [r2],8
 9        fmpy f3 = f1,f2
10        stfd [r3],8 = f5
11        add  r0 = r0,1
12        cmp.< r0,100
13        br.cond L1
14        fadd f5 = f3,f4      // エピローグ部
15        nop
16        stfd [r3],8 = f5
```

図 7.1: 図 6.1 のプログラムに対するソフトウェア・パイプライン化コード（その 1）

1 プロローグ部 ... パイプラインへ命令を徐々に埋めていく段階、

2 カーネル部 ... パイプラインに命令が完全に充たされた状態でデータが処理されていく段階、

3 エピローグ部 ... パイプラインから命令が徐々に掃けていく段階

と考えればよい。カーネル部には nop 命令が全くないことを注意する。すなわちプロセッサが休みなく実行することとなり、プロセッサの最大性能を引き出していると言えよう。カーネル部は 99 回繰り返し実行されるが、iteration 当たり 8MC で実行する。プロローグ部の実行時間は 5MC でなる。エピローグ部の実行時間は 3MC である。よって

$$[\text{図 7.1 のコードの実行時間}] = 5 + 99 \times 8 + 3 = 800(\text{MC})$$

これに対して、図 6.3 のコードは iteration 当り 12MC で実行するから

$$[\text{図 6.3 のコードの実行時間}] = 100 \times 12 = 1200(\text{MC})$$

4 となり、ソフトウエア・パイプライン化されたコードはそうでないコードよりも 1.5 倍、高速で
5 ある。

6 　さて、図 7.1 のコードは正しいコードであろうか（正しい結果を与えるだろうか）。それを見る
7 ために、図 7.2 のタイミング・チャートを用いる。この図の列（縦軸）には図 7.1 のコードの i=0、
8 i=1、i=2、... の各 iteration に関係する命令群を配置する。行（横軸）は命令の発行タイミング
9 （MC）である。

MC	i=0	i=1	i=2	i=3	i=4
1	ldfd f2 = [r1],8				
2	ldfd f4 = [r2],8				
3	nop				
4	fmpy f3 = f1,f2				
5	add r0 = r0,1				
6		ldfd f2 = [r1],8			
7	fadd f5 = f3,f4				
8		ldfd f4 = [r2],8			
9		fmpy f3 = f1,f2			
10	stfd [r3],8 = f5				
11		add r0 = r0,1			
12		cmp.< r0,100			
13		br.cond L1			
14			ldfd f2 = [r1],8		
15		fadd f5 = f3,f4			
16			ldfd f4 = [r2],8		
17			fmpy f3 = f1,f2		
18		stfd [r3],8 = f5			
19			add r0 = r0,1		
20			cmp.< r0,100		
21			br.cond L1		
22				ldfd f2 = [r1],8	
23			fadd f5 = f3,f4		
24				ldfd f4 = [r2],8	
25				fmpy f3 = f1,f2	
26			stfd [r3],8 = f5		
27				add r0 = r0,1	
28				cmp.< r0,100	
29				br.cond L1	
30					ldfd
31				fadd f5 = f3,f4	
32					ldfd
33					fmpy
...				...	...

図 7.2: 図 7.1 のコードのタイミング・チャート

この図から複数の iteration がオーバーラップして実行していることが分かる。オーバーラップしない場合には図 6.3 のように nop 命令がコード中に現れる（nop 命令を配置しないならばプロセッサがストールしてしまう）が、図 7.1 のコードではその nop 命令の場所に別の iteration の命令が配置されているのである。

図 6.1 のプログラムのひとつの iteration の実行が図 7.1 のコードではカーネル・ループの二つの iteration にわたって実行されている。すなわち、図 6.1 の i=k の iteration は i=$k+1$ の iteration とオーバーラップして実行されている。二つの iteration が重なるとき、このコードの**ソフトウェア・パイプライン・ステージ**（software pipeline stage[1]）の数は 2 であると言う。

[1] stage ではなく、slip と呼ぶ場合もある。

二つのソフトウェア・パイプライン・ステージのうち、最初のステージ — これを第1ステージ、またはステージ番号1のステージとも呼ぼう — で発行される命令は

```
        ldfd f2 = [r1],8
        ldfd f4 = [r2],8
        fmpy f3 = f1,f2
```

であり、第2ステージで発行される命令は

```
        fadd f5 = f3,f4
        stfd [r3],8 = f5
```

である。なお、ループ制御命令

```
        add  r0 = r0,1
        cmp.< r0,100
        br.cond L1
```

は特定のステージと結び付けては考えない。

一般に i=k の iteration が、i=$k+1, ..., k+n$ の iteration とオーバーラップするとき、ソフトウェア・パイプライン・ステージ数は $n+1$ である。たとえば図7.3のコードも図6.1のプログラムをソフトウェア・パイプライン化したコードであるが、この場合のステージ数は4である。

ソフトウェア・パイプライン化コードを理解する方法として、図7.2のように、連続する複数の iteration の命令発行タイミングを見る方法もある。しかし慣れてくると、ひとつの iteration の命令発行タイミングを見ただけでコードの様子が理解できるようになる。図7.4のタイミング・チャートは、カーネル部を実行中の（ソース・プログラムの）ひとつの iteration の命令群を発行タイミング毎に図示したものである。図中の記号*は、その位置で他の iteration の命令が発行されることを意味し、記号#はループ制御命令が実行されることを意味する。

7.2　命令並列度が2以上の場合

前節の議論は、もちろん命令並列度が2以上の場合にも成り立つ。

図7.5は、命令並列度が2のときのソフトウェア・パイプライン化されたコードの例である。ただし煩雑なのでカーネル部だけを示す（プロローグ部、エピローグ部はカーネル部から導出できることを後に述べる）。このコードはカーネル部を4MCで実行する。nop 命令が全く含まれていないから、図7.1のコードの場合と同様に、プロセッサの最大性能を引き出すコードである。前章、図6.4にソフトウェア・パイプライン化されていないコードを示した。このコードの iteration 当たりの実行時間は9MCであった。図7.5のコードはそれよりも $9/4 = 2.25$ 倍、高速である。

問題　図7.4のタイミング・チャートと同様のものを図7.5のコードについて作れ。

```
 1        ldfd f2 = [r1],8       // プロローグ部
 2        add  r0 = r0,1
 3        nop
 4        fmpy f3 = f1,f2
 5        ldfd f4 = [r2],8
 6        ldfd f2 = [r1],8
 7        add  r0 = r0,1
 8        fadd f5 = f3,f4
 9        fmpy f3 = f1,f2
10        ldfd f4 = [r2],8
11        ldfd f2 = [r1],8
12        add  r0 = r0,1
13 L1:    stfd [r3],8 = f5        // カーネル部
14        fadd f5 = f3,f4
15        fmpy f3 = f1,f2
16        ldfd f4 = [r2],8
17        ldfd f2 = [r1],8
18        add  r0 = r0,1
19        cmp.< r0,100
20        br.cond L1
21        stfd [r3],8 = f5        // エピローグ部
22        fadd f5 = f3,f4
23        fmpy f3 = f1,f2
24        ldfd f4 = [r2],8
25        stfd [r3],8 = f5
26        nop
27        fadd f5 = f3,f4
28        nop
29        stfd [r3],8 = f5
```

図 7.3: 図 6.1 のプログラムに対するソフトウェア・パイプライン化コード（その 2）

7.3　実行時間の計算

図 7.1 のコード、図 7.3 のコードではカーネル部は 8MC 毎に次々と iteration を開始していく。図 7.5 のコードではカーネル部は 4MC 毎に iteration を開始していく。この時間間隔のことを **iteration initiation interval**（iteration 立ち上げ間隔、簡単に initiation interval とも呼ぶ）と呼び、III、II、I.I. などの記号で表すのが一般的である。このテキストでは II で表す。

II はコードの実行性能を決める最も重要な因子である。以下に説明しよう。

コードの総実行時間 T は、

$$T = [\text{プロローグ部の実行時間}] + [\text{カーネル部の実行時間}] + [\text{エピローグ部の実行時間}]$$

```
13        *                        // 第 1 ステージ
14        *
15        *
16        *
17        ldfd f2 = [r1],8
18        #
19        #
20        #
-------------------------------------------------
13        *                        // 第 2 ステージ
14        *
15        fmpy f3 = f1,f2
16        ldfd f4 = [r2],8
17        *
18        #
19        #
20        #
-------------------------------------------------
13        *                        // 第 3 ステージ
14        fadd f5 = f3,f4
15        *
16        *
17        *
18        #
19        #
20        #
-------------------------------------------------
13        stfd [r3],8 = f5         // 第 4 ステージ
14        *
15        *
16        *
17        *
18        #
19        #
20        #
```

図 7.4: 図 7.3 のコードのタイミング・チャート

```
1 L1:    fadd f5 = f3,f4;     ldfd f4 = [r2],8
2        fmpy f3 = f1,f2;     add  r0 = r0,1
3        ldfd f2 = [r1],8;    cmp.< r0,100
4        stfd [r3],8 = f5;    br.cond L1
```

図 7.5: 図 6.1 のプログラムに対応する命令並列度 2 のソフトウェア・パイプライン化コード

である。そして、ソフトウェア・パイプライン・ステージ数が n、ループ長（= ベクトル長、iteration 回数）が L ならば、

$$[\text{カーネル部の実行時間}] = (L - (n-1)) * \mathrm{II}$$

である。プロローグ部の実行時間、エピローグ部の実行時間はそれぞれ $(n-1) * \mathrm{II}$ と同程度である（詳細は次章で述べる）から、結局、

$$T \simeq (n-1) * \mathrm{II} + (L - (n-1)) * \mathrm{II} + (n-1) * \mathrm{II} = (L + n - 1) * \mathrm{II}$$

となる。

ループ長 L はコード生成法に依らず不変である。もし $L \gg n$ ならば、ソフトウェア・パイプライン・ステージ数 n の実行時間への影響は無視できる。よって実行時間を決定する最も重要な数値は II であり、実行時間は II に比例する。

第8章　モジュロ・スケジューリング

基本ブロックのスケジューリングでさえ NP 完全であった（4 章）のだから、ソフトウェア・パイプライン化はもちろん NP 完全である。しかし基本ブロックのスケジューリングを解く近似アルゴリズムとしてリスト・スケジューリングがあったように、ソフトウェア・パイプライン化されたコードを作る近似アルゴリズムとして**モジュロ・スケジューリング**（modulo scheduling）が知られている。

「モジュロ」と言えば、剰余数を思い出す人も多いだろう。たとえば数列 0、1、2、3、4、5、6、7、.... について、3 を基数とする剰余数は 0、1、2、0、1、2、0、1、... である。モジュロ・スケジューリングも命令のスケジューリングについて、これと類似した操作を行うことを頭の隅に置いておくと分かりやすい。

モジュロ・スケジューリングは以下のステップからなる。

1. ループ本体のデータ依存グラフを作成する（4 章参照）。
2. データ依存グラフをもとに iteration 立ち上げ間隔 II（の最小値）を定める。
3. II をもとに空（カラ）のリソース予約表を作る。この予約表は 4 章のものとは少し異なる。
4. データ依存グラフ中の全ての頂点について優先度を定める。
5. 優先度の高い頂点から順にリソース予約表に命令を埋めていく。この操作を命令スケジューリングと呼ぶ。命令スケジューリングに失敗したならば、II の値を 1 増やしてステップ (3) から再試行する。
6. 命令スケジューリング結果からプロローグ部、カーネル部、エピローグ部のコードを生成する。
7. レジスタ割付を行う。

なお、この授業ではレジスタ割付は省略する。以下では 1. から 6. の各ステップについて詳しく述べる。

```
1        for(i = 0; i < 100; i++){
2                 z[i] = 2.0*x[i]+y[i];
3        }
```

図 8.1: ループプログラムの例

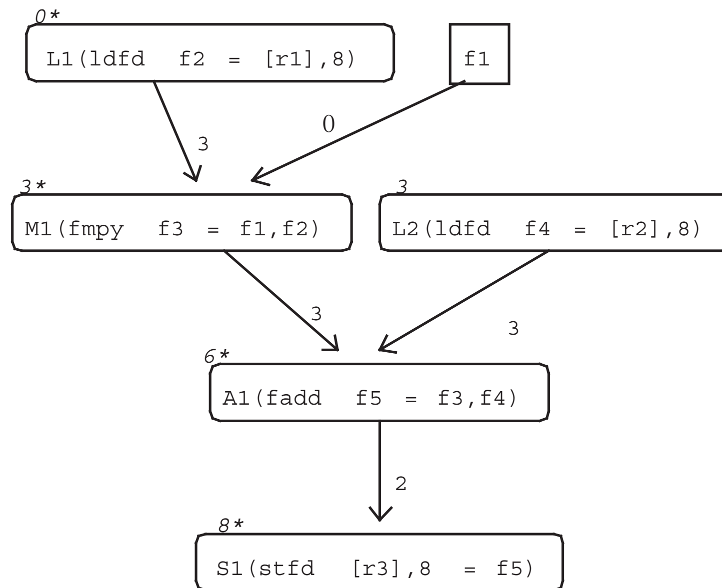

図 8.2: 図 8.1 のプログラムのループ本体のデータ依存グラフ

8.1　データ依存グラフの作成

データ依存グラフの作成方法は基本的には 4.1 節と同じである。たとえば図 8.1 は図 6.1 を再掲したプログラムである。これのループ本体のデータ依存グラフを作ると図 8.2 のようになる。ここに f1 には定数 2.0 が既に格納されており、各命令のレーテンシは 6 章と同じく以下の通りとする。

```
1       for(i = 0; i < 100; i++){
2               acc = acc*x[i]+y[i];
3       }
```

図 8.3: ループ運搬依存のあるループプログラムの例

命令	レーテンシ（MC）
ldfd	3
stfd	1
fmpy	3
fadd	2
add	1
comp	1
br.cond	1

ここまででは 4 章と変わりないように思えるかもしれない。本節と 4.1 節のリスト・スケジューリングの議論との大きな相違点は、基本ブロックのデータ依存グラフは決して閉路を持たないのに対して、ループ本体のデータ依存グラフは**閉路**（closed path）を持つ場合があることである。

たとえば図 8.3 のプログラムを考えてみよう。このプログラムでは変数 acc の値は直前の iteration の acc の値に依存する。このような依存を**ループ運搬依存**（loop-carried dependence）と呼ぶ。この例のデータ依存グラフは図 8.4 の通りである。命令 A1 の結果は次の iteration の命令 M1 で使用されるため、それを閉路として表現するのである。

また図 8.5 のプログラムもループ運搬依存を持つ。

ループ運搬依存を持つループのソフトウェア・パイプライン化は持たない場合に比べ難易度が上がるから、まずはループ運搬依存のない場合を検討し、ループ運搬依存のある場合については章を改めて述べることとする。

8.2 iteration 立ち上げ間隔 II の決定

II の下限は次の二つの要因で決定される。

(1) プロセッサの命令並列度とデータ依存グラフ中の命令数

(2) ループ運搬依存によるデータ依存グラフ内の閉路の長さ

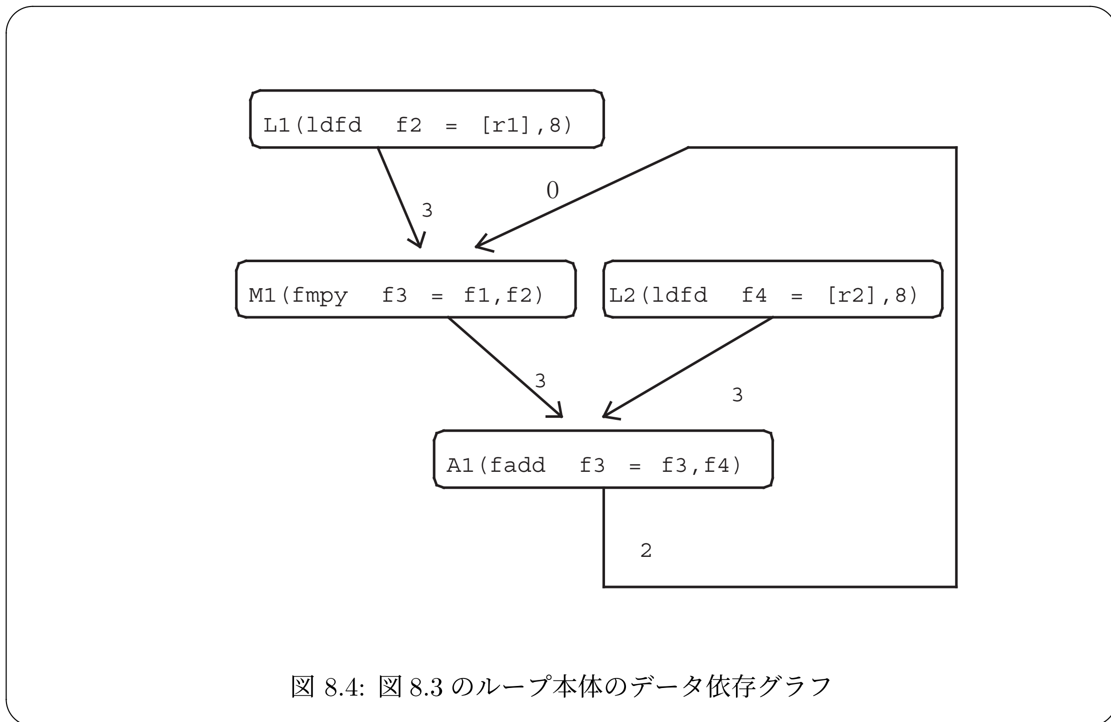

図 8.4: 図 8.3 のループ本体のデータ依存グラフ

```
1        for(i = 1; i < 100; i++){
2                z[i] = z[i-1]*x[i]+y[i];
3        }
```

図 8.5: ループ運搬依存のあるループプログラムの例

先に述べたように、当面 (2) は扱わないから (1) のみが II を決めると考えてよい。以下、それを命令並列度毎に検討する。

命令並列度が 1 の場合

命令並列度が 1 のときのループ・コードは以下のような定型パターンであると約束した（6.1 節）。

```
ループ・コード・パターン 1:
    L1:     命令 1
            命令 2
            ...
            命令 k
            add  r0 = r0,1
            cmp.< r0,100
            br.cond L1
```

これをソフトウェア・パイプライン化されたコードにも採用する。このとき、もしデータ依存グラフ中の命令数が k であり、コード中に nop が現れないならば、

$$\mathrm{II} = (k+3)\ \mathrm{MC}$$

になるはずである。たとえば図 8.2 のグラフの場合、$k=5$ であるから II は 8MC と計算される。

なお、この値は、nop がコードに全く現れずにソフトウェア・パイプライン化に成功した場合の II であることを注意する。後に述べる様々な理由で nop が不可避の場合には実際の II は上に計算した II よりも大きくなる。ここで計算する II は、あくまでもコード最適化前の推定値である。

命令並列度が 2 の場合

命令並列度が 2 のときのループ・コードは以下のような定型パターンであると約束した（6.3 節）。

```
ループ・コード・パターン 2:
    L1:    命令 1 ;    命令 2
           命令 3 ;    命令 4
           ...         ...
           命令 k-4 ;  命令 k-3
           命令 k-2 ;  add  r0 = r0,1
           命令 k-1 ;  cmp.< r0,100
           命令 k ;    br.cond L1
```

このパターンを基にして、データ依存グラフ中の命令数と考えうる最小の II の関係を計算すると以下のようになる。

命令数	II
1	3
2	3
3	3
4	4
5	4
6	5
7	5
...	...

たとえば命令数が 1 の場合 — そのようなデータ依存グラフは稀であるが — 三つのループ制御命令と合わせて総命令数は 4 となる。これを 2 で割った数 2 が II の最小値となりそうだが、しかし三つのループ制御命令は同時に実行できないから、II は 3 以下にはできない。命令数が 4 の場合は三つのループ制御命令と合わせて総命令数は 7 となる。この値は 2 で割り切れないから II の最小値は 4 と計算する。以上のことから、命令数 k (≥ 2) について、

$$\mathrm{II} = \lceil (k+3)/2 \rceil\ \mathrm{MC}$$

が成り立つ。$\lceil x \rceil$ は小数点以下を切り上げる天井関数（ceil function）を表す。$k=1$ の場合はこの式を用いない。たとえば図 8.2 の場合には $\mathrm{II} = \lceil (5+3)/2 \rceil = 4\mathrm{MC}$ である。

命令並列度が 3 以上のときも同様である。ループ・コードの定型パターンをひとつ定め、そのパターン中に k 個の命令を nop 命令ができる限り現れないように詰めた場合の II を計算すればよい。

8.3　空のリソース予約表

リソース予約表とは、前節のループ・コード・パターン 1、2 をそのまま表とみなしたものである。特に、命令をスケジューリングする予定の各エントリを空に初期化した表を空（カラ）のリソース予約表という。図 8.2 の例題については命令並列度 1 の場合には II=8MC であるから、以下のような表となる。

0	
1	
2	
3	
4	
5	add r0 = r0,1
6	cmp.< r0,100
7	br.cond L1

5 行目から 7 行目にはあらかじめループ制御命令を埋めておく。なお、4 章では表の 1 列目の時刻（MC）を 1 から始め、1, 2, 3, ... としたが、モジュロ・スケジューリングでは剰余数を用いるから、その時刻（MC）を 0 から始め、0, 1, 2, ... とする。どちらを用いても本質的な差は無い。単に便宜上の理由である。

命令並列度 2 の場合は II=4MC であるから以下のような表となる。

0		
1		add r0 = r0,1
2		cmp.< r0,100
3		br.cond L1

やはり、表の 1 列目の時刻（MC）は 0 から始め、0, 1, 2, ... とする。

8.4　命令の優先度

命令スケジューリングの準備として、データ依存グラフ中の全ての命令にあらかじめ優先度を決めておく。これはリスト・スケジューリングの場合（4 章）と全く同様に行えばよい。4 章では優先度の決め方として

- 4.2 節 (I) + (II) + (III)、または
- 4.2 節 (I) + (II) + 4.3 節 (III')

の二つを示したが、どちらを採用しても大差はないと思われる。この章では前者を採用する。すなわち、図 8.2 の各頂点の肩の優先度パラメータの値が小さなものほど優先度が高いとする。

8.5　命令スケジューリング

8.3 節で作った空のリソース予約表に命令を順次埋めていく作業を以下のステップで行う。これはモジュロ・スケジューリング全体の中で最も重要な処理である。

1. データ依存グラフ G から、先行頂点を持たない頂点（命令）を全て集め、スケジューリング候補集合 S とする。
2. 候補集合 S の中で最も優先度の高い頂点 n を取り出す。
3. 頂点 n について、以下のいずれかを適用し、n を表に埋め、かつ n の発行時点 $t(n)$ を 決定する。ここに $t(n)$ は、ソフトウェア・パイプラインの第 1 ステージの 1 行目（0MC 目）の時点から n が発行される時点までの経過時間（MC）とする。

   (a) もし既にスケジューリングされた頂点（命令）m_1、...、m_k から n へのデータ依存が存在するならば、まず以下の値 start、end を求める。ここに L_1、...、L_k は頂点 m_1、...、m_k のレーテンシとする。

$$\begin{aligned}\mathrm{start} &= \max(t(m_1)+L_1,...,t(m_k)+L_k)\\ \mathrm{end} &= \min(t(m_1)+\mathrm{II},...,t(m_k)+\mathrm{II})\end{aligned}$$

   簡単に言えば、start は n を発行できる最も早い時点であり、それよりも早い時点にスケジューリングするとプロセッサ・ストールが起きる。end は n を発行できる最も遅い時点であり、それよりも遅い時点にスケジューリングすると、次の iteration の値で計算してしまい、正しい計算にならない（例題を用いた解説を後に述べる）。

   i. もし start $>$ end ならば、この II でのモジュロ・スケジューリングは失敗とみなす。

   ii. もし start $\leq$ end ならば、まず表の start%II 行目に空のエントリが残っているか否か調べる。もし残っているならば、そこに n を埋める。さもなくば、順に (start+1)%II、(start + 2)%II、...、end%II の各行を調べ、空のエントリが見つかれば、対応する命令をそこに埋める。もし全ての可能な行について空のエントリがないならば、この II でのモジュロ・スケジューリングは失敗とみなす。もし n を (start + j)%II 行目に埋めることができたならば、$t(n) = \mathrm{start} + j$ とする。

   (b) もし n へのデータ依存を持つ頂点が表に存在しない（すなわち、初期のデータ依存グラフに元々、n の先行頂点が存在しない）ならば、n の優先度パラメータ p について、表

表 8.1: 図 8.2 のスケジューリング過程（命令並列度 1, II=8）

MC	#1	#2	#3	#4	#5
0	L1(0)	L1(0)	L1(0)	L1(0)	L1(0)
1				A1(9)	A1(9)
2					×
3		M1(3)	M1(3)	M1(3)	M1(3)
4			L2(4)	L2(4)	L2(4)
5	*	*	*	*	*
6	*	*	*	*	*
7	*	*	*	*	*

の p%II 行目に空のエントリが残っているか否か調べる。もし残っているならば、そこに n を埋める。さもなくば、順に $(p+1)$%II、$(p+2)$%II、...、$(p+\mathrm{II}-1)$%II の各行を調べ、空のエントリが見つかれば、n をその行に埋める (なお、この操作では必ず空のエントリが見つかるから失敗は起きない)。n を $(p+j)$%II 行目に埋めることができたならば、$t(n) = p + j$ とする。

4. 頂点 n をグラフ G から取り除く。

5. もし G が空ならば、スケジューリングは終了。さもなくば 1. へ戻る。

8.3 節のリソース予約表（命令並列度 1）に図 8.2 の命令をスケジューリングしていく過程を示したのが、表 8.1 である。ただしループ制御命令は * で表した。これについて以下、命令毎に詳細に解説する。

1. まず図 8.2 の中で最も優先度の高い頂点 L1 をスケジューリングすることを試みる。このとき規則 3(b) を適用する。L1 の優先度パラメータは 0 であるから、0%8 = 0 行目に L1 を埋める。よって $t(\mathtt{L1}) = 0$ である。表 8.1 では $t(\mathtt{L1})$ の値を命令の後ろに付記し、L1(0) と表した。

2. 次に優先度の高いのは M1 である。M1 は L1 に依存するから規則 3(a) を適用する。ここで

$$\begin{aligned}\mathrm{start} &= \max(t(\mathtt{L1}) + [\mathtt{L1}\text{ のレーテンシ}]) = \max(0+3) = 3\\ \mathrm{end} &= \min(t(\mathtt{L1}) + \mathrm{II}) = \min(0+8) = 8\end{aligned}$$

ここに start ≤ end であるから、規則 3(a)i. は適用されず、この時点ではこのモジュロ・スケジューリングはまだ失敗しない（後のステップで失敗と判断されるかもしれないが）。さて、start%8 = 3 であり、かつ 3 行目は空であるから、規則 3(a)ii. により M1 を 3 行目に埋める。よって $t(\mathtt{M1}) = 3$ である。表 8.1 ではこれを M1(3) と表した。

3. 次に優先度の高いのは `L2` である。`L2` はそれが依存する命令を持たないから規則 3(b) を適用する。そこで $3\%8 = 3$ 行目に命令を埋めたい。しかしそこには既に `M1` が埋まっているため、次の 4 行目に埋める。よって $t(\texttt{L2}) = 4$ である。表 8.1 ではこれを `L2`(4) と表した。

4. 次に優先度の高いのは `A1` である。`A1` は `M1` と L2 に依存する。よって start、end は以下のように計算される。

$$\begin{aligned}\text{start} &= \max(t(\texttt{M1}) + [\texttt{M1}\text{ のレーテンシ}], t(\texttt{L2}) + [\texttt{L2}\text{ のレーテンシ}]) \\ &= \max(3+3, 4+3) = 7 \\ \text{end} &= \min(t(\texttt{M1}) + \text{II}, t(\texttt{L2}) + \text{II}) \\ &= \min(3+8, 4+8) = 11\end{aligned}$$

この場合、start $\leq$ end を満たしているから、規則 3(a).i は適用されない。規則 3(a)ii. により、まず $7\%8 = 7$ 行目に `A1` を埋めたいが、そこには既に `br.cond` が埋まっている（7 行目の∗)。そこで次に $(7+1)\%8 = 0$ 行目に埋めたいが、そこには既に `L1` が埋まっている。そこでさらに $(7+2)\%8 = 1$ 行目を調べる。1 行目は空いているから `A1` を 1 行目に埋める。よって $t(\texttt{A1}) = 9$ である。なお、$t(\texttt{A1})$ の値は剰余数 $9\%8 = 1$ ではないことを注意する。表 8.1 ではこれを `A1`(9) と表した。

5. 最後に `S1` である。`S1` は `A1` に依存するから

$$\begin{aligned}\text{start} &= \max(t(\texttt{A1}) + [\texttt{A1}\text{ のレーテンシ}]) = \max(9+2) = 11 \\ \text{end} &= \min(t(\texttt{A1}) + \text{II}, t(\texttt{L2}) + \text{II}) = \min(9+8) = 17\end{aligned}$$

となる。この場合も、start $\leq$ end を満たしているから、規則 3(a).i は適用されない。そこで規則 3(a)ii. により、まず $11\%8 = 3$ 行目に `S1` を埋めたいが、そこには既に `M1` が埋まっている。そこで $(11+1)\%8 = 4$、$(11+2)\%8 = 5$、$(11+3)\%8 = 6$、$(11+4)\%8 = 7$、$(11+5)\%8 = 0$、end$\%8 = 17\%8 = 1$ の各行を順に調べるがいずれも既に命令が埋まっているため、`S1` を埋めることができない。よって、このスケジューリングは失敗である。2 行目（2MC 目）が空いてるが、そこに `S1` を埋めることはできない。何故ならば、`A1` が 1 行目にあるため、2 行目ではプロセッサ・ストールが起こるからである。

スケジューリングに失敗した場合には、II の値をひとつ大きくして、再試行する。一般に II が大きくなるほど、表に空のエントリが増えるから、スケジューリングが成功する可能性が大きくなり、ある II で必ずスケジューリングは成功する。何故ならば、II の値が増加し、十分大きな値になると、モジュロ・スケジューリングはリスト・スケジューリングと等価になるからである。しかし前章で述べたように、コードの実行時間は II に比例するから、II が大きくなるほど実行速度が

表 8.2: 図 8.2 のスケジューリング過程（命令並列度 1, II=9）

MC	#1	#2	#3	#4	#5
0	L1(0)	L1(0)	L1(0)	L1(0)	L1(0)
1				A1(10)	A1(10)
2					
3		M1(3)	M1(3)	M1(3)	M1(3)
4			L2(4)	L2(4)	L2(4)
5					S1(14)
6	*	*	*	*	*
7	*	*	*	*	*
8	*	*	*	*	*

表 8.3: 図 8.6 の命令を逆順にスケジューリングしたコード

MC	命令
0	S1(24)
1	A1(17)
2	M1(10)
3	L2(11)
4	L1(4)
5	*
6	*
7	*

低下することを忘れてはならない。上の例題の場合、II=8 で失敗したため、II = 9 とし、再試行する。その結果が表 8.2 のスケジューリング過程である。今度は S1 を 5 行目に埋めることができ、命令スケジューリングは成功する。

前章で示したソフトウェア・パイプライン化コード（図 7.1 のコード、図 7.3 のコード）は II が 8MC であったが、実はそれらは人手で試行錯誤をしながら求めたコードである。機械的なコード生成は人手のようにはうまく行かない。上の例題でうまく行かなかった原因は、命令 S1 を埋めるところにある。S1 を埋めることのできる唯一の空白行 2 行目は A1 と近すぎるのである。このような不都合を解決する手段として様々な方法が考えられる。たとえば先行命令を持たない L1 や L2 を埋める位置を工夫する、バックトラックを導入するなど様々である。

もしソフトウエア・パイプライン・ステージ数が増えても構わないならば、ソフトウエア・パイプライン化コードを作る単純な方法がある。まず、素朴なコードを用意する。たとえば図 8.6 は図 6.1 のプログラムを素朴にコード化したものであった。このコード中の命令の順番 L1、L2、M1、

```
 1 L1:    ldfd f2 = [r1],8 // L1
 2        ldfd f4 = [r2],8 // L2
 3        nop
 4        fmpy f3 = f1,f2  // M1
 5        nop
 6        nop
 7        fadd f5 = f3,f4  // A1
 8        nop
 9        stfd [r3],8 = f5 // S1
10        add  r0 = r0,1
11        cmp.< r0,100
12        br.cond L1
```

図 8.6: 図 6.1 のプログラムのコード例（図 6.3 の再掲）

表 8.4: 図 8.2 のスケジューリング過程（命令並列度 2, II=4）

	#1		#2		#3		#4	
0	L1(0)		L1(0)		L1(0)	L2(4)	L1(0)	L2(4)
1		*		*		*	×	*
2		*		*		*	×	*
3		*	M1(3)	*	M1(3)	*	M1(3)	*

A1、S1 を逆順にリソース予約テーブルに詰め、$t()$ を矛盾なく決めれば、そのコードは II=8 で実行可能なソフトウエア・パイプライン化コードになっている（表 8.3 参照）。

この方法は単純であるにも関わらず、多くの場合にうまくいくため、ソフトウエア・パイプライン化コードを人手で作る場合に有効である。ただし、ソフトウエア・パイプライン・ステージ数が異常に大きくなる欠点があるため、短ループ長のプログラムに適用すると、実行効率を落とす。レジスタの使用数も増える。表 8.3 の例ではステージ数は 4 である。実は、前章の図 7.3 のコードはこの方法で作られたコードである。

あらゆる場合に有効な、万能アルゴリズムは存在しないのである（だからこそ NP 困難な問題である）。

同じ例題を命令並列度 2 について考察しよう。

表 8.4 の II=4MC の場合には A1 のスケジューリングに失敗している。何故ならば、L2 のスケジューリングを終えたところで $t(\texttt{M1}) = 3$、$t(\texttt{L2}) = 4$ である。よって、A1 について、

$$\text{start} = \max(t(\texttt{M1}) + [\texttt{M1}\text{ のレーテンシ}], t(\texttt{L2}) + [\texttt{L2}\text{ のレーテンシ}])$$

表 8.5: 図 8.2 のスケジューリング過程（命令並列度 2, II=5）

	#1	#2	#3	#4	#5
0	L1(0)	L1(0)	L1(0)	L1(0)	L1(0)　S1(10)
1					
2	*	*	*	A1(7)　*	A1(7)　*
3	*	M1(3)　*	M1(3)　*	M1(3)　*	M1(3)　*
4	*	*	L2(4)　*	L2(4)　*	L2(4)　*

$$
\begin{aligned}
&= \max(3+3, 4+3) = 7 \\
\text{end} &= \min(t(\texttt{M1}) + \text{II}, t(\texttt{L2}) + \text{II}) \\
&= \min(3+4, 4+4) = 7
\end{aligned}
$$

上の start、end の値から、$7\%4 = 3$ 行目にのみスケジューリング可能であるが、そこには既に M1 と br.cond が埋まっているからである。

II をひとつ増やし、II=5 ならば、スケジューリングは成功する（表 8.5 参照)。

ところで、先に与えた規則 3(b) には別案もある。

たとえば C をある正定数とするとき、3(b) の中の式 $p\%\text{II}$、$(p+1)\%\text{II}$、... を式 $(p+C)\%\text{II}$、$(p+C+1)\%\text{II}$、... に置き換えても不都合はないように思われる。これによって当然スケジューリング結果は変わってくる。うまく C を選べば、初期値の II でスケジューリングに成功するかもしれない。どのような C が適切なのか等の詳細は、プロセッサによっても異なり、コンパイラ開発者が個々に経験的に知っているような事柄になろう。C を変えながらスケジューリングを繰り返してもよいだろう。ループを高速に実行することは極めて重要であるから、コード最適化に少々時間がかかったとしてもその価値はある。

8.6　コードの生成

命令スケジューリング後のリソース予約表と、その表の中の各命令のソフトウェア・パイプライン・ステージ番号が分かれば、実行可能なコードの生成は容易である。

以下は、コード生成のステップである。ただし、ソフトウェア・パイプライン・ステージ数を S とする。また、命令 n が発行されるソフトウエア・パイプライン・ステージ番号を $s(n)$ とする。なお、$s(n)$ の値はスケジューリング時に計算される $t(n)$ を用いて、$s(n) = \lfloor t(n)/\text{II} \rfloor + 1$ と計算できる。$\lfloor x \rfloor$ は小数点以下を切り捨てる床関数である。

プロローグ部の生成

$i = 1, 2, ..., S - 1$ について以下を繰り返せ。

$j = 0, 1, 2, ..., \mathrm{II} - 1$ について、リソース予約表の j 行目に配置された命令を n_j として、以下のいずれかを実行せよ。

1. $s(n_j) \leq i$ ならば、その命令をそのまま出力せよ。
2. $s(n_j) > i$ ならば、`nop` 命令を出力せよ。
3. リソース予約表の空のエントリは `nop` 命令と見なして出力せよ。
4. n_j がループ制御命令 `add r0 = r0,1` ならば、それをそのまま出力せよ。
5. n_j がループ制御命令 `cmp.< r0,100`、または `br.cond L1` ならば、`nop` 命令を出力せよ。

カーネル部の生成

まずラベル `L1:` を出力せよ。次にリソース予約表の中の 0 行目から $\mathrm{II} - 1$ 行目の全ての命令を順にそのまま出力せよ。空のエントリは `nop` 命令と見なして出力せよ。

エピローグ部の生成

$i = 1, 2, ..., S - 1$ について以下を繰り返せ。

$j = 0, 1, 2, ..., \mathrm{II} - 1$ について、リソース予約表の j 行目に配置された命令を n_j として、以下のいずれかを実行せよ。

1. $s(n_j) > i$ ならば、その命令をそのまま出力せよ。
2. $s(n_j) \leq i$ ならば、`nop` 命令を出力せよ。
3. リソース予約表の空のエントリは `nop` 命令と見なして出力せよ。
4. n_j がループ制御命令 `add r0 = r0,1`、`cmp.< r0,100`、`br.cond L1` ならば、`nop` 命令を出力せよ。

たとえば表 8.6 は、表 8.2 のスケジューリング結果について、表の各エントリを具体的な命令に置き換え、かつ $t()$ と $s()$ の計算値を付けた表である。図 8.7 はこの表から生成したコードである。

表 8.7 は、表 8.5 のスケジューリング結果についてまとめた表である。以下のコード 7.2 は、この表から生成したコードである。

表 8.6: 表 8.2 のスケジューリング結果のまとめ

MC	命令	$t()$	$s()$
0	lfdf f2 = [r1],8	0	1
1	fadd f5 = f3,f4	10	2
2			
3	fmpy f3 = f1,f2	3	1
4	ldfd f4 = [r2],8	4	1
5	stfd [r3],8 = f5	14	2
6	add r0 = r0,1	-	-
7	cmp.< r0,100	-	-
8	br.cond L1	-	-

表 8.7: 表 8.2 のスケジューリング結果のまとめ

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	lfdf f2 = [r1],8	0	1	stfd [r3],8 = f5	10	3
1						
2	fadd f5 = f3,f4	7	2	add r0 = r0,1	-	-
3	fmpy f3 = f1,f2	3	1	cmp.< r0,100	-	-
4	ldfd f4 = [r2],8	4	1	br.cond L1	-	-

```
 1       ldfd f2 = [r1],8      //プロローグ部
 2       nop
 3       nop
 4       fmpy f3 = f1,f2
 5       ldfd f4 = [r2],8
 6       nop
 7       add  r0 = r0,1
 8       nop
 9       nop

10 L1:   ldfd f2 = [r1],8      //カーネル部
11       fadd f5 = f3,f4
12       nop
13       fmpy f3 = f1,f2
14       ldfd f4 = [r2],8
15       stfd [r3],8 = f5
16       add  r0 = r0,1
17       cmp.< r0,100
18       br.cond L1

19       nop                   //エピローグ部
20       fadd f5 = f3,f4
21       nop
22       nop
23       nop
24       stfd [r3],8 = f5
25       nop                   //これ以降は削除可
26       nop
27       nop
```

図 8.7: 表 8.6 から生成したコード

```
 1        lfdf f2 = [r1],8;        nop                // プロローグ部
 2        nop;                     nop
 3        nop;                     add  r0 = r0,1
 4        fmpy f3 = f1,f2;         nop
 5        ldfd f4 = [r2],8;        nop
 6        lfdf f2 = [r1],8;        nop
 7        nop;                     nop
 8        fadd f5 = f3,f4;         add  r0 = r0,1
 9        fmpy f3 = f1,f2;         nop
10        ldfd f4 = [r2],8;        nop

11 L1:    lfdf f2 = [r1],8;        stfd [r3],8 = f5   // カーネル部
12        nop;                     nop
13        fadd f5 = f3,f4;         add  r0 = r0,1
14        fmpy f3 = f1,f2;         cmp.< r0,100
15        ldfd f4 = [r2],8;        br.cond L1

16        nop;                     stfd [r3],8 = f5   // エピローグ部
17        nop;                     nop
18        fadd f5 = f3,f4;         nop
19        nop;                     nop
20        nop;                     nop
21        nop;                     stfd [r3],8 = f5
22        nop;                     nop                //これ以降は削除可
23        nop;                     nop
24        nop;                     nop
25        nop;                     nop
```

図 8.8: 表 8.7 から生成したコード

第9章　ループのレジスタ割付け

ソフトウェアパイプライン化されたループについてもレジスタ割付けの方法を概説する。

例題として、まず前章の図 8.7 のコードを考えよう。プロローグ部、エピローグ部のレジスタ割付けは、カーネル部のレジスタ割付けの結果をそのまま用いることができるので、カーネル部だけを考えればよい。

仮想レジスタの概念は 5 章と変わらないから、図 8.7 のコードの `f2`、...、`f5` をそのまま `v2`、...、`v5` に置き換えておく。結果、我々がレジスタ割付けの対象とするコードは図 9.1 である。このコードでは 4 本の仮想レジスタを使用しているが、実レジスタへの最適な割付けを行なうことで、その使用本数を減らせるだろうか。

9.1　巡回生存区間

生存区間の考え方は、基本的に 5.2 節と同じである。たとえば、図 9.1 のコードの場合、

$$\texttt{v2} = (10, 13], \quad \texttt{v5} = (11, 15]$$

である。しかし、ループの場合、iteration を跨（また）いで生存し続けるレジスタが存在する。たとえば、図 9.1 の `v3` は時点 13 で定義され（ターゲットレジスタとなり）、カーネル部の末尾まで生存し、次の iteration の時点 11 で参照される（ソースレジスタとなる）。このレジスタの生存区間を便宜上、以下のように定義し、

$$\texttt{v3} = (-\infty, 11] \cup (13, \infty)$$

このような生存区間を巡回生存区間（cyclic live range, cyclic interval）と呼ぶ。また「生存区間が巡回的である」、「仮想レジスタは巡回的である」と言う。実は仮想レジスタ `v4` も巡回的であり、

$$\texttt{v4} = (-\infty, 11] \cup (14, \infty)$$

となる。

巡回生存区間を含む生存区間の間の干渉（生存区間が重なること）の有無も 5 章と同様に求めることができる。ここで次の事実に注意する。

- 任意の二つの巡回的仮想レジスタは必ず干渉している。

```
10 L1:    lfdf v2 = [r1],8      //カーネル部
11        fadd v5 = v3,v4
12        nop
13        fmpy v3 = f1,v2
14        ldfd v4 = [r2],8
15        stfd [r3],8 = v5
16        add  r0 = r0,1
17        cmp.< r0,100
18        br.cond L1
```

図 9.1: 図 8.7 のレジスタを仮想化したコード

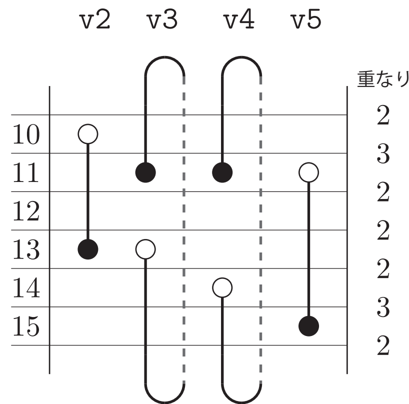

図 9.2: 巡回生存区間グラフ

さて、干渉の有無が決まれば、与えられた仮想レジスタ（とその生存区間）の集合 $\{v_1, ..., v_m\}$ を非干渉集合に分割し、ひとつの非干渉集合にひとつの（仮想ではない）実レジスタを割り付ければよい。このとき、できるだけ少ない数の実レジスタを割り付けたいのだが、5 章とは異なり、巡回生存区間を含む場合の実レジスタ数の最適化問題は NP 困難な問題であることが知られている。つまり、仮想レジスタ数の増加とともに最適なレジスタ割付に掛かる時間は指数的に増加する。これについては後にも触れる。

9.2　巡回生存区間グラフ

5 章と同様に、仮想レジスタの生存区間をグラフとして図示すると、干渉の様子が分かりやすい上に、レジスタ割り付けのアルゴリズムも直感的に理解できる。上の図 9.1 のコードの仮想レジスタ v2、...、v5 の生存区間を図示したものが図 9.2 である。このような図を巡回生存区間グラフと

呼ぼう。この図では各時点での生存区間の重なりの最大値は3であるから、少なくとも 3 本の実レジスタが必要であることが分かる。しかし、実際には4本の仮想レジスタは全て互いに干渉していることから、4 本の実レジスタが必要であることが直ちに分かる。ソフトウェアパイプライン化されたコードはデータの流れがギチギチに詰まっていることが多いため、複数の仮想レジスタをひとつの実レジスタへまとめることは難しいことが多い。

一般に巡回生存区間グラフを用いるレジスタ割付けは5.3節のアルゴリズムを拡張し、以下のように行なえばよい。前節で触れたように、レジスタ割り付け問題はNP困難な問題であるから、我々は実用上は近似アルゴリズムしか与えることができないのだが、以下の方法がひとつの妥当な近似アルゴリズムと考えられる。

今、仮想レジスタ（とその生存区間および巡回生存区間）の集合 $V = \{v_1, ..., v_m\}$ が与えられたとしよう。このとき、グラフを用いたレジスタ割り付けのアルゴリズムは以下の通りである。

1. V の中から、巡回的な仮想レジスタ v をひとつ選び、

$$
\begin{aligned}
V &= V - \{v\} \\
V' &= \{v\}
\end{aligned}
$$

   とする。もしそのような v が複数存在するならば、その中の適当なひとつを選ぶ。もし巡回的な仮想レジスタが存在しないならば、5.3節のアルゴリズムを適用する。

2. v の巡回生存区間を $(-\infty, t_e] \cup (t_s, \infty)$ とする。

3. V の中に、その開始時点 t'_s が t_e 以上（つまり、$t_e \leq t'_s$）であり、かつその終了時点 t'_e が t_s 以下（つまり、$t'_e \leq t_s$）であるような、巡回的ではない仮想レジスタが全く存在しないならば、6. へ行く。

4. V の中の、上記3. の条件を満たす仮想レジスタの中で、差 $t'_s - t_e$ が最小であるような仮想レジスタ v' をひとつ選び、

$$
\begin{aligned}
V &= V - \{v'\} \\
V' &= V' \cup \{v'\}
\end{aligned}
$$

   とする。もしそのような v' が複数存在するならば、その中の適当なひとつを選ぶ。

5. v' の終了時点を新たに t_e とし、3. へ戻る。

6. V' に含まれる全ての仮想レジスタに同じ実レジスタ番号を割り当てる。

7. V が空（から）でないならば、1. へ戻る。

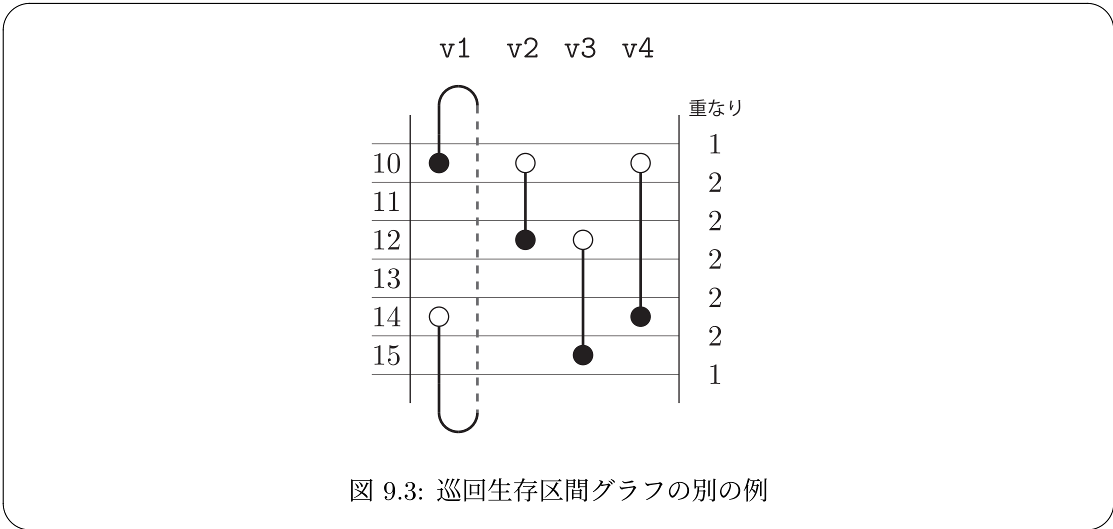

図 9.3: 巡回生存区間グラフの別の例

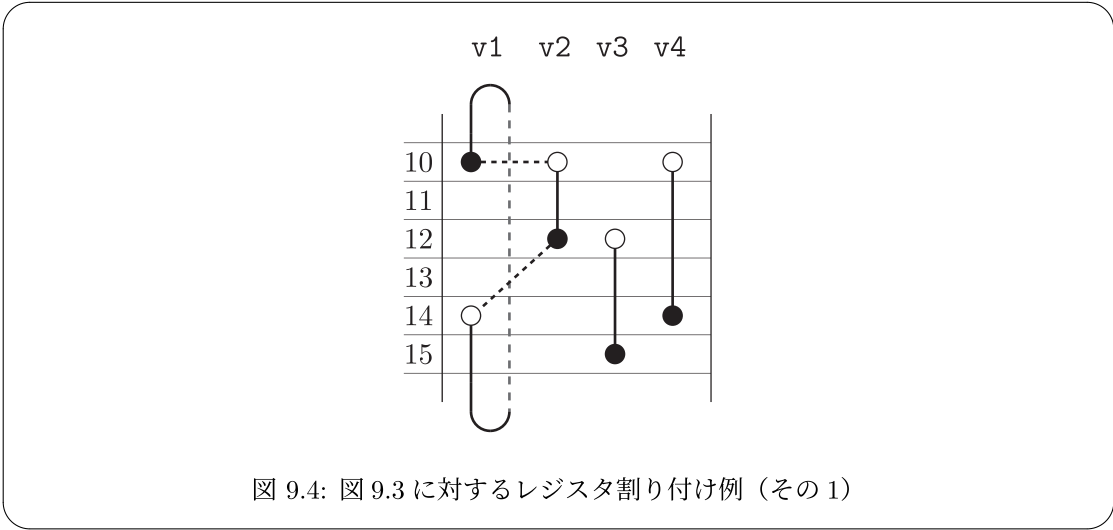

図 9.4: 図 9.3 に対するレジスタ割り付け例（その 1）

上のアルゴリズムの手順 1. においてひとつの巡回的仮想レジスタ v を選んだ後、それに続くレジスタ v' を順次選んで行く。これを仮想レジスタの鎖とみなすとき、鎖の末端は最初に選んだ v の開始時点に連結されて、**仮想レジスタの輪**と見なすことができる。その輪にひとつの実レジスタを割り付けるのである。

あまり面白い例ではないが、図 9.2 について実際にレジスタ割り付けを行ってみよう。初めに巡回的仮想レジスタ v3 を選んだとする。しかし、これと連結できるレジスタは存在しないため、 v3 にひとつの実レジスタを割り付ける。次に、巡回的仮想レジスタ v4 を選んだとする。この場合も、これと連結できるレジスタは存在しないため、 v4 にひとつの実レジスタを割り付ける。残る v2、v5 は非巡回的であるから、5 章のアルゴリズムを適用する。この場合、 v2、v5 は干渉するから、それぞれにひとつの実レジスタを割り付ける。結果、この例では 4 本の実レジスタが必要である。

上の例は面白い例ではなかった。図 9.3 は、あるコードから求めた、上とは別の巡回生存区間グ

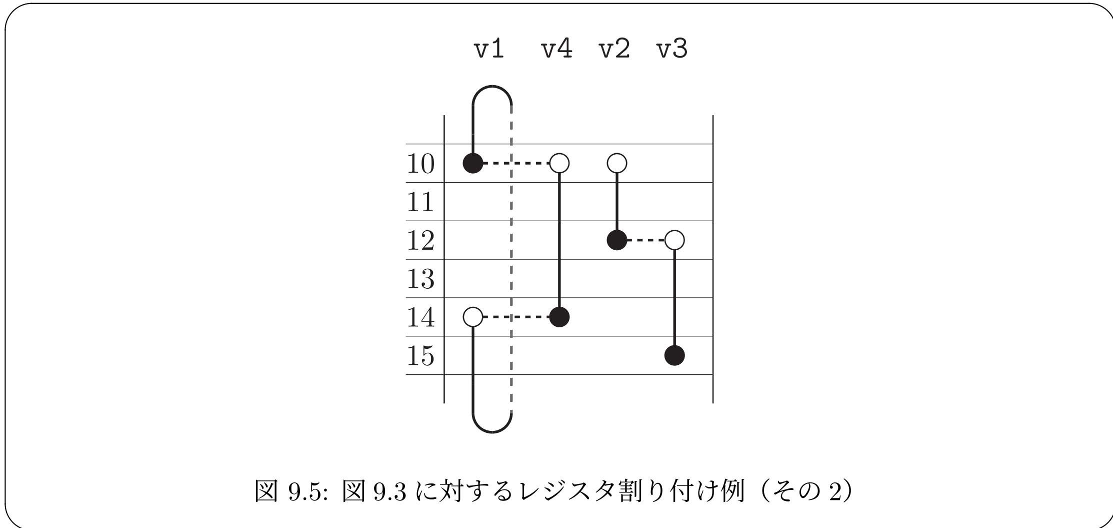

図 9.5: 図 9.3 に対するレジスタ割り付け例（その 2）

ラフの例である。

このグラフのレジスタ割り付けでは、まず巡回的仮想レジスタ v1 を選ぶ。この v1 につなぐことのできる候補レジスタは v2 と v4 の二つである。v1 の終了時点と v2 の開始時点の差、v1 の終了時点と v4 の開始時点の差は共に 0 であるから、アルゴリズムではどちらも採用してもよい。そこで、仮に v2 を選ぶとする。そうすると、上のアルゴリズムに基づくレジスタ割り付けでは、図 9.4 のような割り付け結果となり、3 本の実レジスタが必要である。逆に v4 を選ぶとする。そうすると、図 9.5 のような割り付け結果となり、2 本の実レジスタが必要である。結果、図 9.5 の方が良い結果を与えた。しかし、v1 につなぐレジスタとして v4 の優位性を事前に判断することは容易ではない。あえて言えば、v4 の生存区間が v2 よりも長いことがひとつの根拠になりそうだが、それはこの事例に限ってのことであり、反例を示すことはたやすい。

5 章では仮想レジスタの**鎖**を作ることを考えた。それに対して本章では仮想レジスタの**輪**を作ることを考えねばならない。輪を作るには、鎖の先頭と末尾をうまく連結する必要があるのだが、それを事前に見通して輪を構成することは難しい。このことが本章のレジスタ割り付けを難しくしている。

9.3　レジスタ干渉グラフ

図 9.3 の例について言えば、レジスタ彩色法を用いる方が見通しが立て易い。図 9.6 がそのレジスタ干渉グラフである。枝の多い v3、v4 から順に彩色すれば自ずと 2 色で彩色できる。

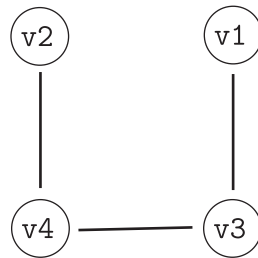

図 9.6: 図 9.3 に対するレジスタ干渉グラフ

第10章　ループ運搬依存がある場合のソフトウェアパイプライン化

ここまでの例題は全てデータ依存が当該 iteration の中で閉じている場合を扱った。ループ本体が条件分岐を含むような複雑な場合も扱ったが、データ依存は当該 iteration の中で閉じていた。この章では iteration を越えたデータ依存（ループ運搬依存）を持つ計算のソフトウェア・パイプライン化を考察する。これは 7 章、8 章の内容の拡張に当たる。なお、この章では「ループ運搬依存」を「iteration を越えたデータ依存」という場合もある。

10.1　アイディア

8 章で触れた図 8.3 のプログラムを再度検討する。データ依存グラフは図 10.1 の通りである。これは図 8.4 と同じものであるが、この章の議論ではもっぱら図 10.1 を参照してほしい。この図では iteration を越える依存 `A1`→`M1` を点線の枝で表した。配列要素 `x[i]`、`y[i]` のメモリ・アドレスは `r1`、`r2` にそれぞれ格納されているとし、配列要素の値は `f1`、`f2` にロードするものとする（図 10.1 の命令 `L1`、`L2` を見よ）。変数 `acc` の初期値は `f4` に事前に格納されているものと約束する。各命令のレーテンシは、8 章と同様に以下の通りとする。

命令	レーテンシ（MC）
`ldfd`	3
`fmpy`	3
`fadd`	2

8 章で述べたように、このプログラムの難しさは iteration を越えるデータ依存が存在することである。当該 itereation において変数 `acc` の値を計算するためには直前の iteration の `acc` の値が必要である。そのため、直前の iteration の `acc` の計算が終了するまで当該 itereation の `acc` の計算は開始できない。しかし、ソフトウェアパイプライン化では隣接する複数の iteration の計算をオーバーラップさせて同時実行することで高速化を目指す。この両者が相反する方向にある。スケジューリング過程を一般化し、最適なオーバーラップを行うことがこの章のコード最適化のポイントになる。

ループ運搬依存を持つプログラムのモジュロ・スケジューリング・アルゴリズムを述べる前に、まず図 8.3 のプログラムを最適化したコード例を示す。表 10.1 は、命令スケジューリング後のリソース予約表である。ここではプロセッサの命令並列度が 2 の場合を示した。表中の $t()$、$s()$ は 8

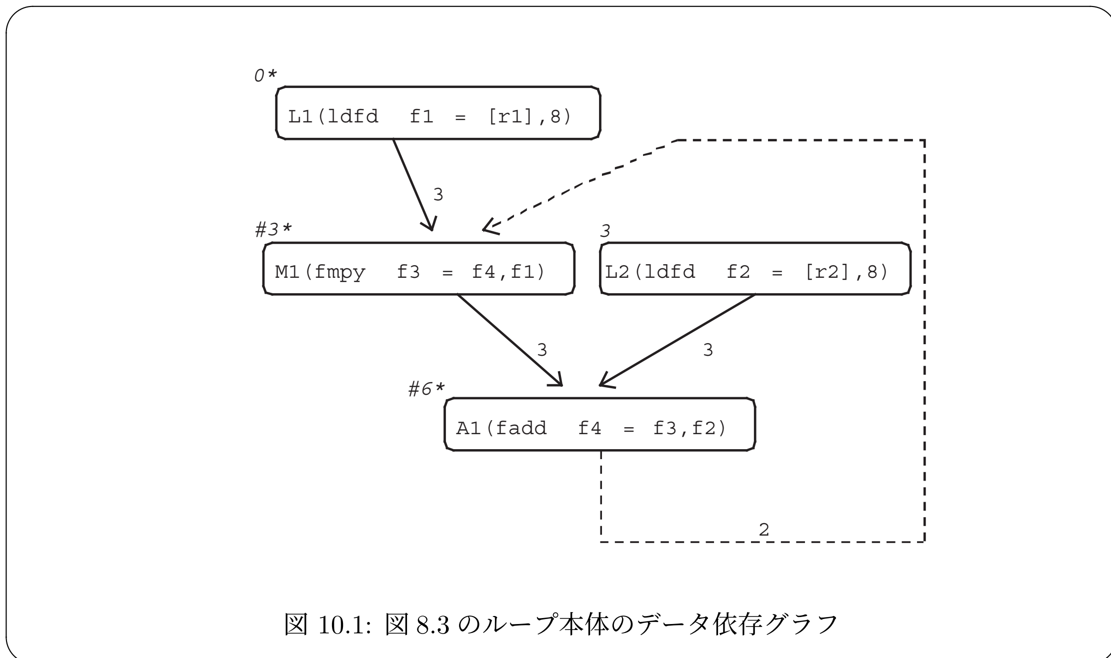

図 10.1: 図 8.3 のループ本体のデータ依存グラフ

表 10.1: 図 8.3 のプログラムのソフトウエア・パイプライン化コードの例

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	L1:ldfd f1 = [r1],8	0	1			
1	A1:fadd f4 = f3,f2	6	2			
2	L2:ldfd f2 = [r2],8	2	1	add r0 = r0,1	-	-
3	M1:fmpy f3 = f4,f1	3	1	cmp.< r0,100	-	-
4				br.cond L1	-	-

章と同じく、命令の発行タイミングとステージ番号を表す。この表中には空白が多いが、しかしこれは最速なコードである。

この表の II が 5MC であることを注意してほしい。図 10.1 に含まれる命令数は 4 であるから、命令並列度が 2 の場合、8 章の議論に従うならば II= $\lceil (4+3)/2 \rceil$ =4MC になりそうであるが、実際にはこのプログラムを II=4MC でスケジューリングすることは不可能である。

その理由を明らかにするために、表 10.1 のコードのタイミング・チャートを表 10.2 に示す。なお、ループ制御命令は省略する。

まずチャート 3 行目の `fmpy f3 = f4,f1` に注目したい。この加算の結果は 6 行目の乗算 `fadd f4 = f3,f2` において使用されるが、両者の発行タイミングは 3MC 空いており、プロセッサ・ストールは起きない。6 行目の演算結果は 8 行目の乗算 `fmpy f3 = f4,f1` において使用されるが、両者の発行タイミングは 2MC 空いており、プロセッサ・ストールは起きない。8 行目の結果は 11 行目の加算において使用されるが、両者の間は 3MC だけ空いており、やはりプロセッサ・ストー

表 10.2: 表 10.1 のコードのタイミング・チャート

MC	i=$k-1$	i=k	i=$k+1$	i=$k+2$	i=$k+3$
0	...	ldfd f1 = [r1],8			
1					
2		ldfd f2 = [r2],8			
3	→→	fmpy f3 = f4,f1			
4		↓			
5		↓	ldfd f1 = [r1],8		
6		fadd f4 = f3,f2			
7		↓	ldfd f2 = [r2],8		
8		→→→→→→→	fmpy f3 = f4,f1		
9			↓		
10			↓	ldfd f1 = [r1],8	
11			fadd f4 = f3,f2		
12			↓	ldfd f2 = [r2],8	
13			→→→→→→→	fmpy f3 = f4,f1	
14				↓	
15				↓	
16				fadd f4 = f3,f2	
...				↓	...

ルは起きない。さらに 11 行目の結果は... と続く。これは図 10.1 のデータ依存グラフで言えば、M1→A1→M1→A1→... のデータ依存に相当する。

この M1→A1→M1→A1→... の中の最初の M1 は、i=k の iteration の乗算命令であり、2 番目の M1 は i=$k+1$ の iteration の同じ乗算命令である。よって、

- 最初の M1 と 2 番目の M1 の発行タイミングは厳密に II MC だけ離れている。

ところで、M1、A1 のレーテンシはそれぞれ 3MC、2MC であるから、M1→A1→M1 の経路の重みは 5MC である。これは

- 最初の M1 と 2 番目の M1 の発行タイミングが少なくとも 5MC 以上離れていなければならない

ことを意味する。この二つの事実から、II は 5MC 以上以上でなければならない。つまり表 10.1 は最適なコードなのである。

上の議論を一般化しよう。プログラムがループ運搬依存を含む場合のモジュロ・スケジューリングは以下のステップからなる。

1. ループ本体のデータ依存グラフを作成する。
2. データ依存グラフをもとに iteration 立ち上げ間隔 II（の最小値）を定める。
3. II をもとに空（カラ）のリソース予約表を作る。

4. データ依存グラフ中の全ての頂点について優先度を定める。
5. 優先度の高い頂点から順にリソース予約表に命令を埋めていく。命令スケジューリングに失敗したならば、II の値を 1 増やしてステップ (3) から再試行する。
6. 命令スケジューリング結果からプロローグ部、カーネル部、エピローグ部のコードを生成する。
7. レジスタ割付を行う。

この中で 2.、4.、5. の内容が、8 章から拡張される。

10.2　データ依存グラフの作成

8 章とほとんど同じである。ただし、ここではループを越える依存の枝は点線で表し、ループを越えない依存は実線で表すこととし、両者が明確に区別できるようにしておく。

10.3　iteration 立ち上げ間隔 II の決定

II の下限は次の二つの要因で決定される。

(1) プロセッサの命令並列度とデータ依存グラフ中の命令数

(2) ループ運搬依存によるデータ依存グラフ内の閉路の重み

(1) から定まる II と (2) から定まる II の大きい方が下限の II である。もし閉路が複数個存在する場合には、それらの重みの最大値を用いる。

(1) から定まる II はプロセッサの命令並列度に逆比例して小さくなるが、(2) から定まる II は命令並列度に依存しないことを注意する。どんなに命令並列度の大きなプロセッサを使用しても、(2) から定まる II を小さくすることはできない。

10.4　空のリソース予約表

8 章と同じである。

10.5　命令の優先度

優先度パラメータの求め方は 5 章と同じである。ただし、パラメータ値の計算では iteration を越える依存は無視する。5 章では、同じパラメータを持つ場合、クリティカル・パス上の頂点が優先されたが、同様の優先規則をここでも採用する。加えて、以下の優先規則を新たに設ける。

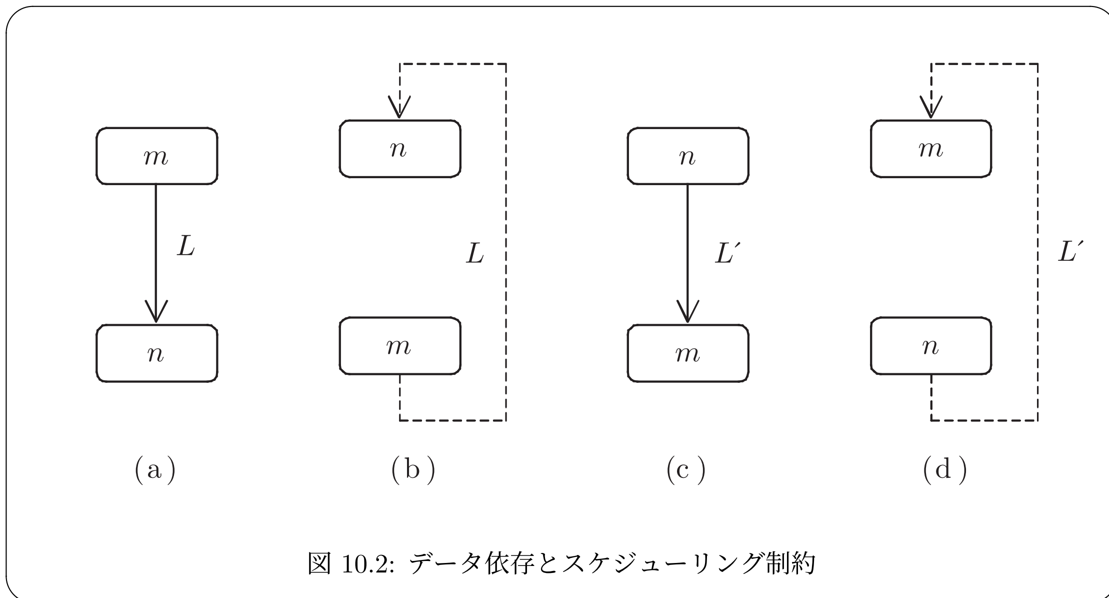

図 10.2: データ依存とスケジューリング制約

- 閉路上にある頂点は、閉路上にない頂点よりも優先度が高い。
- 複数の閉路が存在する場合、より大きな重みを持つ閉路上の頂点の優先度が高い。

8.3 節で述べたように、閉路は高速化のネックになる。そのため、閉路上の頂点が適切にスケジューリングされることが最も優先されるべきである。そこで図 10.1 では閉路上の頂点の優先度パラメータに'#' を付けた。この印の付いた頂点から優先してスケジューリングすべきである。

10.6　命令スケジューリング

10.6.1　スケジューリング・アルゴリズム（その 1）

ループ運搬依存を含む場合のスケジューリング・アルゴリズムは前章よりも複雑である。というのも、8 章では先行頂点を持たない頂点から順にスケジューリングを適用した。そのため、後続の頂点が先行頂点よりも先にスケジューリングされていることはなく、プロセッサ・ストールを起こさないためのスケジューリングの制約条件（命令を埋め込むことができる場所）の計算が先行頂点から後続頂点へ一方向に進むからである。これに対して、ループ運搬依存を含む場合には閉路上の頂点を最優先にスケジューリングするため [1]、両方向の制約条件を想定したスケジューリングが必要である。

アルゴリズムを説明する準備として、命令のデータ依存とスケジューリング位置の関係を図 10.2 に示す 4 通りに分けて解説する。ここに頂点 m が発行タイミング $t(m)$ でスケジューリング済みであると仮定する。すなわち m はリソース予約表の $t(m)$%II 行目に既に埋められている。

[1] 7 節と同様のスケジューリングも可能ではある。しかし 7 節の方法では閉路上の頂点を優先しないため、結果として本章の方法よりも II が大きくなってしまうと思われる。

図 10.2(a) の場合: m から n への iteration を越えないデータ依存があるならば、n はタイミング $t(m)+L$ 以降に発行せねばならない。ここに L は m に対応する命令のレーテンシである。これは 8 章と同じ理屈である。また n は $t(m)+\mathrm{II}$ よりも後に発行してはいけない。これも 8 章と同じ理屈である。よって n を発行できる区間は $[t(m)+L, t(m)+\mathrm{II}]$ である。

図 10.2(b) の場合: m から n への iteration を越えるデータ依存があるならば、n はタイミング $t(m)-\mathrm{II}+L$ 以降に発行せねばならない。何故ならば、当該 iteration における m の発行タイミングが $t(m)$ ならば、直前の iteration での発行タイミングは $t(m)-\mathrm{II}$ である。よって、それから L だけ進んだポインタが n の最も早い発行タイミングである。また n は $(t(m)-\mathrm{II})+\mathrm{II}=t(m)$ よりも後に発行してはいけない。よって n を発行できる区間は $[t(m)-\mathrm{II}+L, t(m)]$ である。

図 10.2(c) の場合: n から m への iteration を越えないデータ依存があるならば、n はタイミング $t(m)-L'$ 以前に発行せねばならない。ここに L' は n に対応する命令のレーテンシである。これは (a) の逆パターンである。また n は $t(m)-\mathrm{II}$ よりも前に発行してはいけない。よって n を発行できる区間は $[t(m)-\mathrm{II}, t(m)-L']$ である。

図 10.2(d) の場合: n から m への iteration を越えるデータ依存があるならば、n はタイミング $t(m)+\mathrm{II}-L'$ 以前に発行せねばならない。何故ならば、当該 iteration における m の発行タイミングが $t(m)$ ならば、直後の iteration での m の発行タイミングは $t(m)+\mathrm{II}$ である。よって、それから L' だけ先行するタイミングが n の最も遅い発行タイミングである。また n は $t(m)$ よりも前に発行してはいけない。よって n を発行できる区間は $[t(m), t(m)+\mathrm{II}-L']$ である。

頂点 n は、この 4 種類のパターンから定まる区間の共通区間内にスケジューリングされるべきである。

以上の事柄を踏まえ、スケジューリング手順を定式化する。実はこの定式化はまだ荒っぽく、不完全であるが、理解を容易にするために、敢えてこの定式化を紹介する。空のリソース予約表に命令を順次埋めていく作業を以下のステップで行う。

1. データ依存グラフ G から最も優先度の高い頂点 n を取り出す。優先度は以下のように決める [2]。
    - 閉路上の頂点が最も優先される。
    - 同じ閉路上の頂点の中では、優先度パラメータの小さいものほど優先される。
    - 複数の閉路が存在する場合、より大きな重みの閉路が優先される。
    - 閉路上にない頂点では、優先度パラメータの小さいものほど優先される。

[2] これらは講義担当者の経験に基づく規則に過ぎない。より実証的な研究が必要である。

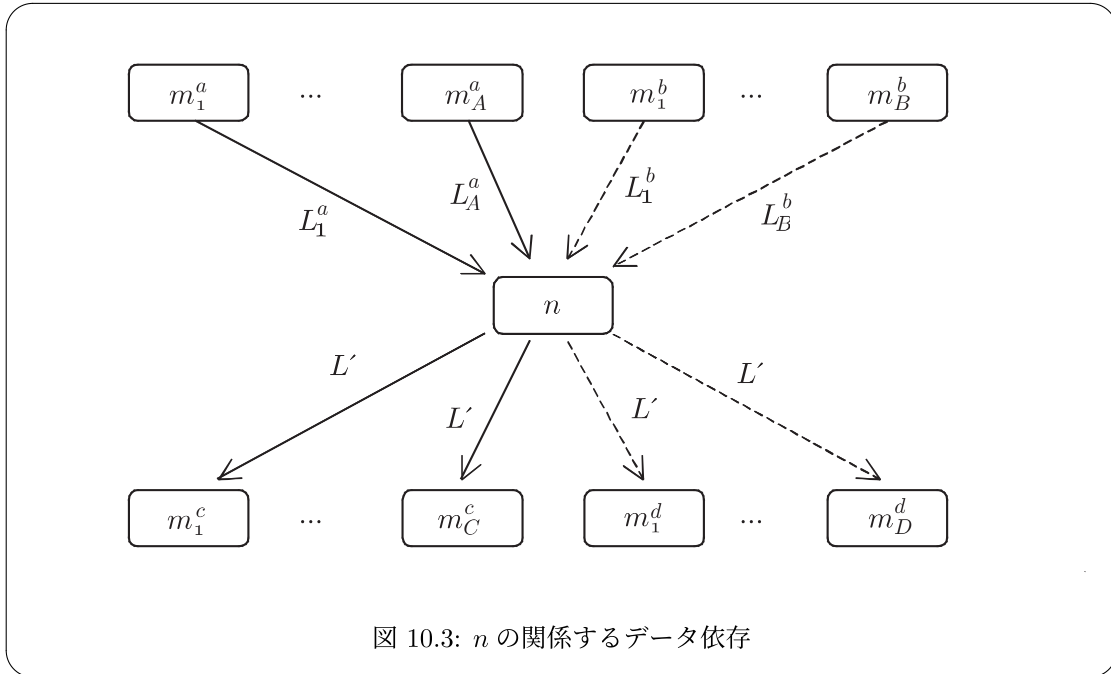

図 10.3: n の関係するデータ依存

1

- 優先度パラメータが同じ場合にはクリティカル・パス上の頂点が優先される。

2. 頂点 n について、既にスケジューリングされている頂点 m_1^a、...、m_A^a、m_1^b、...、m_B^b から n へのデータ依存が存在するとする （$A, B \geq 0$)。ここに m_1^a、...、m_A^a から n への依存は iteration を越えない依存とし、m_1^b、...、m_B^b から n への依存は iteration を越える依存とする。また n から既にスケジューリングされている頂点 m_1^c、...、m_C^c、m_1^d、...、m_D^d へのデータ依存が存在するとする （$C, D \geq 0$)。ここに n から m_1^c、...、m_C^c への依存は iteration を越えない依存とし、n から m_1^d、...、m_D^d への依存は iteration を越える依存とする（図 10.3 参照)。なお、L_1^a、...、L_A^a は頂点 m_1^a、...、m_A^a に対応する命令のレーテンシ、L_1^b、...、L_B^b は頂点 m_1^b、...、m_B^b に対応する命令のレーテンシ、L' は頂点 n に対応する命令のレーテンシとする。このとき start、end の値を以下のように求める。

$$\begin{aligned}
\text{start} &= \max(\text{start}^a, \text{start}^b, \text{start}^c, \text{start}^d) \\
\text{start}^a &= \max(t(m_1^a) + L_1^a, \cdots, t(m_A^a) + L_A^a, -\infty) \\
\text{start}^b &= \max(t(m_1^b) - \text{II} + L_1^b, \cdots, t(m_B^b) - \text{II} + L_B^b, -\infty) \\
\text{start}^c &= \max(t(m_1^c) - \text{II}, \cdots, t(m_C^c) - \text{II}, -\infty) \\
\text{start}^d &= \max(t(m_1^d), \cdots, t(m_D^d), -\infty)
\end{aligned}$$

$$\begin{aligned}
\text{end} &= \min(\text{end}^a, \text{end}^b, \text{end}^c, \text{end}^d) \\
\text{end}^a &= \min(t(m_1^a) + \text{II}, ..., t(m_A^a) + \text{II}, +\infty)
\end{aligned}$$

$$\begin{aligned}\mathrm{end}^b &= \min(t(m_1^b), ..., t(m_B^b), +\infty)\\ \mathrm{end}^c &= \min(t(m_1^c) - L', ..., t(m_C^c) - L', +\infty)\\ \mathrm{end}^d &= \min(t(m_1^d) + \mathrm{II} - L', ..., t(m_D^d) + \mathrm{II} - L', +\infty)\end{aligned}$$

7.5 節の start、end は上の start^a、end^a に相当する。$A = B = 0$、すなわち頂点 m_1^a、...、m_A^a、m_1^b、...、m_B^b が存在しないならば、$\mathrm{start}^{ab} = -\infty$、$\mathrm{end}^{ab} = +\infty$ となることを注意する。同様に、$C = D = 0$、すなわち頂点 m_1^c、...、m_C^c、m_1^d、...、m_D^d が存在しないならば、$\mathrm{start}^{cd} = -\infty$、$\mathrm{end}^{cd} = +\infty$ となる。

3. 以下の (a)〜(e) のいずれかを適用し、n を表に埋め、かつ値 $t(n)$ を決定する。

   (a) もし start $>$ end ならば、この II でのモジュロ・スケジューリングは失敗とみなす。

   (b) もし start、end が共に有限値であり [3]、かつ start $\leq$ end が成り立つならば、表の start%II、(start + 1)%II、...、end%II 行目に空のエントリが残っているか否か調べる)。もし残っているならば、その中に n を埋める。複数の空のエントリが存在する場合にはその中のどのエントリに埋めても良い。もし空のエントリがないならば、この II でのモジュロ・スケジューリングは失敗とみなす。n を $(\mathrm{start} + j)\%\mathrm{II}$ 行目に埋めることができたならば、$t(n) = \mathrm{start} + j$ とする。

   (c) もし $\mathrm{start} = -\infty$、$\mathrm{end} = +\infty$ ならば、n の優先度パラメータ p について、表の $p\%\mathrm{II}$ 行目に空のエントリが残っているか否か調べる [4]。もし空のエントリが残っているならば、そこに n を埋める。さもなくば、$p\%\mathrm{II}$ 行目の近傍の $(p + 1)\%\mathrm{II}$ 行目、$(p - 1)\%\mathrm{II}$ 行目、$(p + 2)\%\mathrm{II}$ 行目、$(p - 2)\%\mathrm{II}$ 行目、... を順に調べ、そこに空のエントリが見つかれば、n をその行に埋める（なお、この操作では必ず空のエントリが見つかるから失敗は起きない)。n を $(p + j)\%\mathrm{II}$ 行目 （j は負値の場合もありうる）に埋めることができたならば、$t(n) = p + j$ とする。

4. 頂点 n をグラフ G から取り除く。

5. もし G が空ならば、スケジューリングは終了。さもなくば 1. へ戻る。

上の手順の中の 2.、3. が 8 章からの大きな変更点である。

例として、図 10.1 を命令並列度が 1 の場合について検討しよう。

[3] start、end の一方だけが有限値、他方が $\pm\infty$ ということは定義からあり得ない。

[4] 7.5 節に述べたように、$p\%\mathrm{II}$ 行目を最初に調べることについては必ずしも強い根拠がある訳ではない。$\mathrm{start} = -\infty$、$\mathrm{end} = +\infty$ の場合とは、要するに、命令をスケジューリングすべき場所について制約がないことを意味する。ただし、少しだけ考慮すべき事情がある。たとえば頂点 m がスケジューリングされ、次に m と直接には連結していない頂点 n が手順 3(c) に従ってスケジューリングされるとき、m がスケジューリングされる場所と n がスケジューリングされる場所とはある種の関係性を持っており、間接的には互いのスケジューリングに影響を及ぼし合うと考えられる。そのような影響を事前に考慮できることは望ましい。しかし、はたしてそこまで神経質になる必要があるのか、詳細は不明である。

表 10.3: 図 10.1 のスケジューリング過程（命令並列度 1, II=7）

MC	#1	#2	#3	#4
0			L1	L1
1		A1	A1	A1
2				L2
3	M1	M1	M1	M1
4	*	*	*	*
5	*	*	*	*
6	*	*	*	*

1　図 10.1 の命令数 4 にループ制御命令数 3 を加えた数は 7 である。閉路の重みは 5 である。よっ
2 て II の初期値は前者から決まる 7 である。表 10.3 はリソース予約表へのスケジューリング過程で
3 ある。

4　まず頂点 M1 がスケジューリング対象となる。何故ならば、M1 は閉路 M1 $\rightarrow$ A1 $\rightarrow$ M1 上にあり、
5 かつ優先度パラメータ値が A1 よりも小さいからである。そこで、手順 3(c) に従い、M1 を 3 行目
6 に埋める。よって、$t(\texttt{M1}) = 3$ である。

次にスケジューリング対象となるのは A1 である。A1 には、M1 $\rightarrow$ A1 の図 10.2(a) のパターンの依存と、A1 $\rightarrow$ M1 の図 10.2(d) のパターンの依存がある。よって、

$$
\begin{aligned}
\text{start}^a &= \max(t(\texttt{M1}) + 3, -\infty) = 6 \\
\text{start}^b &= \max(-\infty) = -\infty \\
\text{start}^c &= \max(-\infty) = -\infty \\
\text{start}^d &= \max(t(\texttt{M1}), -\infty) = 3 \\
\text{start} &= \max(6, -\infty, -\infty, 3) = 6
\end{aligned}
$$

$$
\begin{aligned}
\text{end}^a &= \min(t(\texttt{M1}) + \text{II}, +\infty) = 10 \\
\text{end}^b &= \min(+\infty) = +\infty \\
\text{end}^c &= \min(+\infty) = +\infty \\
\text{end}^d &= \min(t(\texttt{M1}) + \text{II} - 2, +\infty) = 8 \\
\text{end} &= \min(10, +\infty, +\infty, 8) = 8
\end{aligned}
$$

7 よって、手順 3(b) に従い、$6\%7 = 6$ 行目、$7\%7 = 0$ 行目、$8\%7 = 1$ 行目のエントリを調べる。0
8 行目と 1 行目が空であるから、たとえば 1 行目に A1 を埋める。よって、$t(\texttt{A1}) = 8$ である。

次のスケジューリング対象は L1 である。L1 には、L1 $\rightarrow$ M1 の図 10.2(c) のパターンの依存があ

る。よって、

$$\begin{aligned}
\mathrm{start}^a &= \max(-\infty) = -\infty \\
\mathrm{start}^b &= \max(-\infty) = -\infty \\
\mathrm{start}^c &= \max(t(\mathtt{M1}) - \mathrm{II}, -\infty) = -4 \\
\mathrm{start}^d &= \max(-\infty) = -\infty \\
\mathrm{start} &= \max(-\infty, -\infty, -4, \infty) = -4
\end{aligned}$$

$$\begin{aligned}
\mathrm{end}^a &= \min(+\infty) = +\infty \\
\mathrm{end}^b &= \min(+\infty) = +\infty \\
\mathrm{end}^c &= \min(t(\mathtt{M1}) - 3, +\infty) = 0 \\
\mathrm{end}^d &= \min(+\infty) = +\infty \\
\mathrm{end} &= \min(+\infty, +\infty, 0, +\infty) = 0
\end{aligned}$$

1 よって、手順 3(b) に従い、$(-4)\%7 = 3$ 行目、$(-3)\%7 = 4$ 行目、...、$0\%7 = 0$ 行目のエントリを
2 調べる。0 行目が空いているから、そこへ L1 を詰める。よって、$t(\mathtt{L1}) = 0$ である。

最後のスケジューリング対象は L2 である。L2 には、L2 → A1 の図 10.2(c) のパターンの依存がある。よって、

$$\begin{aligned}
\mathrm{start}^a &= \max(-\infty) = -\infty \\
\mathrm{start}^b &= \max(-\infty) = -\infty \\
\mathrm{start}^c &= \max(t(\mathtt{A1}) - \mathrm{II}, -\infty) = 1 \\
\mathrm{start}^d &= \max(-\infty) = -\infty \\
\mathrm{start} &= \max(-\infty, -\infty, 1, \infty) = 1
\end{aligned}$$

$$\begin{aligned}
\mathrm{end}^a &= \min(+\infty) = +\infty \\
\mathrm{end}^b &= \min(+\infty) = +\infty \\
\mathrm{end}^c &= \min(t(\mathtt{A1}) - 3, +\infty) = 5 \\
\mathrm{end}^d &= \min(+\infty) = +\infty \\
\mathrm{end} &= \min(+\infty, +\infty, 5, +\infty) = 5
\end{aligned}$$

3 よって、手順 3(b) に従い、$1\%7 = 1$ 行目、$2\%7 = 2$ 行目、...、$5\%7 = 5$ 行目のエントリを調べる。
4 2 行目が空いているから、そこへ L2 を詰める。よって、$t(\mathtt{L2}) = 2$ である。

以上、スケジューリングは成功した。しかし、もし A1 を 1 行目ではなく、0 行目に埋めたならば、スケジューリングは失敗する。その場合には II を増やして再度スケジューリングしなければならない。

10.6.2　スケジューリング・アルゴリズム（その 2）

手順 3(b) は、空のエントリが複数個ある場合に実際にどのエントリを選べばよいか、全く規程しておらず、非決定的である。もし次の方針:

- ソフトウエア・パイプライン・ステージ数を可能な限り小さくする。

を満たすようにスケジューリングするならば、エントリの選び方を以下のように改良すべきである。

2. 頂点 n について、図 10.3 と同じデータ依存があるとする。このとき start^{ab}、start^{cd}、end^{ab}、end^{cd} の値を以下のように求める。ただし、start^{a}、...、start^{d}、end^{a}、...、end^{d} の計算式は前と同じとする。

$$\begin{aligned}\mathrm{start}^{ab} &= \max(\mathrm{start}^{a}, \mathrm{start}^{b})\\ \mathrm{start}^{cd} &= \max(\mathrm{start}^{c}, \mathrm{start}^{d})\end{aligned}$$

$$\begin{aligned}\mathrm{end}^{ab} &= \min(\mathrm{end}^{a}, \mathrm{end}^{b})\\ \mathrm{end}^{cd} &= \min(\mathrm{end}^{c}, \mathrm{end}^{d})\end{aligned}$$

7.5 節の start、end は上の start^{ab}、end^{ab} に相当するとも言える。$A = B = 0$、すなわち頂点 m^a_1、...、m^a_A、m^b_1、...、m^b_B が存在しないならば、$\mathrm{start}^{ab} = -\infty$、$\mathrm{end}^{ab} = +\infty$ となることを注意する。同様に、$C = D = 0$、すなわち頂点 m^c_1、...、m^c_C、m^d_1、...、m^d_D が存在しないならば、$\mathrm{start}^{cd} = -\infty$、$\mathrm{end}^{cd} = +\infty$ となる。start^{ab}、end^{ab} は n の先行頂点から n へのデータ依存によって生じる制約を表している。対して start^{cd}、end^{cd} は n から n の後続頂点へのデータ依存によって生じる制約を表している。

3. 以下の (a)〜(e) のいずれかを適用し、n を表に埋め、かつ値 $t(n)$ を決定する。

   (a) もし $\mathrm{start}^{ab} > \mathrm{end}^{cd}$ ならば、この II でのモジュロ・スケジューリングは失敗とみなす。

   (b) もし start^{ab}、end^{cd} が共に有限値であり、かつ $\mathrm{start}^{ab} \leq \mathrm{end}^{cd}$ が成り立つならば、新たに $\mathrm{start} = \max(\mathrm{start}^{ab}, \mathrm{start}^{cd})$、$\mathrm{end} = \min(\mathrm{end}^{ab}, \mathrm{end}^{cd})$ を計算する。もし $\mathrm{start} > \mathrm{end}$ ならば、この II でのモジュロ・スケジューリングは失敗とみなす。もし $\mathrm{start} \leq \mathrm{end}$ ならば、表の start%II、(start+1)%II、...、end%II 行目に空のエントリが残っているか否か調べる)。もし空のエントリが残っているならば、その中のひとつに n を埋める。複数

の空白が存在する場合にはその中のどの空白に埋めても良い)。もし n を $(\mathrm{start}+j)\%\mathrm{II}$ 行目に埋めることができたならば、$t(n) = \mathrm{start} + j$ とする。

(c) もし start^{ab} が有限値、end^{cd} が $+\infty$ ならば、start^{ab} と end^{ab} について以下を行う。もし $\mathrm{start}^{ab} > \mathrm{end}^{ab}$ ならば、この II でのモジュロ・スケジューリングは失敗とみなす。もし $\mathrm{start}^{ab} \le \mathrm{end}^{ab}$ ならば、表の $\mathrm{start}^{ab}\%\mathrm{II}$ 行目に空のエントリが残っているか否か調べる。もし空のエントリが残っているならば、そこに n を埋める。さもなくば、順に $(\mathrm{start}^{ab}+1)\%\mathrm{II}$、$(\mathrm{start}^{ab}+2)\%\mathrm{II}$、...、$\mathrm{end}^{ab}\%\mathrm{II}$ の各行を調べ、空のエントリが見つかれば、n をそこに埋める。もし空のエントリがないならば、この II でのモジュロ・スケジューリングは失敗とみなす。もし n を $(\mathrm{start}^{ab}+j)\%\mathrm{II}$ 行目に埋めることができたならば、$t(n) = \mathrm{start}^{ab} + j$ とする。

(d) もし start^{ab} が $-\infty$、end^{cd} が有限値ならば、start^{cd} と end^{cd} について以下を行う。もし $\mathrm{start}^{cd} > \mathrm{end}^{cd}$ ならば、この II でのモジュロ・スケジューリングは失敗とみなす。もし $\mathrm{start}^{cd} \le \mathrm{end}^{cd}$ ならば、表の $\mathrm{end}^{cd}\%\mathrm{II}$ 行目に空のエントリが残っているか否か調べる。もし空のエントリが残っているならば、そこに n を埋める。さもなくば、順に $(\mathrm{end}^{cd}-1)\%\mathrm{II}$、$(\mathrm{end}^{cd}-2)\%\mathrm{II}$、...、$\mathrm{start}^{cd}\%\mathrm{II}$ の各行を調べ、空のエントリが見つかれば、n をそこに埋める。もし空のエントリがないならば、この II でのモジュロ・スケジューリングは失敗とみなす。もし n を $(\mathrm{end}^{cd}-j)\%\mathrm{II}$ 行目に埋めることができたならば、$t(n) = \mathrm{end}^{cd} - j$ とする。

(e) もし start^{ab} が $-\infty$、end^{cd} が $+\infty$ ならば、n の優先度パラメータ p について、表の $p\%\mathrm{II}$ 行目に空のエントリが残っているか否か調べる)。もし空のエントリが残っているならば、そこに n を埋める。さもなくば、$p\%\mathrm{II}$ 行目の近傍の $(p+1)\%\mathrm{II}$ 行目、$(p-1)\%\mathrm{II}$ 行目、$(p+2)\%\mathrm{II}$ 行目、$(p-2)\%\mathrm{II}$ 行目、... を順に調べ、そこに空のエントリが見つかれば、n をその行に埋める（なお、この操作では必ず空のエントリが見つかるから失敗は起きない)。n を $(p+j)\%\mathrm{II}$ 行目 （j は負値の場合もありうる）に埋めることができたならば、$t(n) = p + j$ とする。

上の、修正された手順を用いても、3(b) になお非決定性が残るが、これを解消する有効な手順とその根拠が存在するか否かは不明である。

10.7　コードの生成

8 章と全く同じである。

```
1        for(i = 2; i < 100; i++){
2            Z[i] = Z[i-2]*x[i]+y[i];
3        }
```

図 10.4: 2 周以上離れたループ運搬依存のあるループプログラムの例（その 1）

```
1        acc0 = 1.0;
2        acc1 = 1.0;
1        for(i = 0; i < 100; i++){
2            acc2 = acc1;
3            acc1 = acc0;
4            acc0 = acc2*x[i]+y[i];
5        }
```

図 10.5: 2 周以上離れたループ運搬依存のあるループプログラムの例（その 2）

10.8　2 周以上離れたループ運搬依存の扱い

たとえば図 10.4、図 10.5 のようなプログラムでは、当該 iteration の計算に二つ前の iteration の計算結果を参照している。このような例では、閉路の重みを 2 で割った数をループ運搬依存から定まる II の下限とすればよい。一般に n 個前の iteration の計算結果を参照する場合には閉路の重みを n で割ればよいが、煩雑なので詳細は述べない。

第11章　rotating registerを用いたモジュロ・スケジューリング

8章のモジュロ・スケジューリング・アルゴリズムでは命令スケジューリングに失敗することがあった。この失敗は、命令をstart%IIからend%IIまでの間に埋めねばならないという制約に起因している（たとえば8.5節の手順3(a)を見よ）。しかしここで紹介するrotating registerを用いるならば、命令をstart%II以降の任意の場所に埋めることができるため、この制約から解放される。つまり、8.2節で見積もったIIで必ずソフトウェア・パイプライン化コードを生成できることとなり、常にプロセッサの性能を最大限に引き出すことができる。

別の言い方をすれば、rotating registerを用いることで、NP問題であるスケジューリング問題がP（決定性多項式時間）問題に単純化される。

11.1　rotating register

たとえば`f32`は、レジスタ番号32で指定される浮動小数点レジスタを意味する。通常、このレジスタを指し示す番号は常に32であり、不変である。これは当然のことと思うかもしれないが、ある種のプロセッサではレジスタ番号をアセンブリ命令で明示的に変更することができる。この変更操作をレジスタの**リネーミング**（renaming、改名）と呼ぶ 。

対象とするプロセッサが、以下の命令:

```
rotate.f
```

を持つと約束する。この命令を実行すると、実行前の

```
f32、f33、...、f126、f127
```

は実行後（発行の1MC後）には

```
f33、f34、...、f127、f32
```

に一斉にリネーミングされると約束しよう。つまり、`f32`は`f33`に改名し、`f33`は`f34`に改名し、...、`f127`は`f128`に、`f128`は`f32`に同時に改名される。そうすると、この命令を繰り返すことで、レジスタ番号が`f32`から`f128`の間で循環的に変化していく。その意味で、このようなレジスタのことを**rotating register**（あるいはrotating register file、敢えて訳せば「回転レジス

```
1        fadd f40 = f31,f32
2        rotate.f
3        fadd f50 = f31,f41
4        rotate.f
```

図 11.1: rotating register を用いたコード片の例

```
0        fadd f40 = f31,f32;     rotate.f
1        fadd f50 = f31,f41;     rotate.f
```

図 11.2: rotating register を用いたコード片の例（命令並列度 2 の場合）

1 タ」）と呼んでいる。ただし f0、...、fr31 や GR はリネーミングせず、前節までと同じ取り扱い
2 ができるとする 。
3 rotating register の機能は、歴史的には CMU（カーネギー・メロン大）の Cydra 5 という計算
4 機で発明された 。その後、筑波大学/日立製作所の CP-PACS プロセッサにおいても実装され、近
5 年ではインテルの 64bit プロセッサ IA-64 に採用されているハードウェア機構である。
6 簡単な例を示す。今、f31、f32 に数値 1.0、2.0 がそれぞれ格納されているとしよう。このと
7 き図 11.1 のコードを実行したと仮定する。そうすると、1 行目の実行後 f40 に 3.0 が格納される。
8 2 行目の実行後、f32、f40 はそれぞれ f33、f41 にリネーミングされる。f33、f41 にはそれぞれ
9 2.0、3.0 が格納されている（ようにソフトウェア的に見える）。f31 はリネーミングされないことを
10 注意する。よって 2 行目の実行後、f50 には 4.0 が格納される。3 行目の実行後、f33、 f41、f50
11 はさらに f34、f42、f51 へリネーミングされる。結局この 4 行を実行後、f34、f42、f51 にはそ
12 れぞれ 2.0、3.0、4.0 が格納されている。
13 命令並列度 2 の場合も同様である。たとえば図 11.2 のコードは図 11.1 のコードと同じ効果を
14 持つ。
15 ここで、**論理レジスタ番号**と**物理レジスタ番号**の区別について触れよう。論理レジスタ番号と
16 は、命令中で指示するレジスタ番号であり、その指し示すレジスタ実体が変更されるかもしれない
17 レジスタ番号である。それに対して、物理レジスタ番号とはプロセッサ内部で使用するレジスタ
18 番号であり、その番号の指し示すレジスタ実体は固定されている。通常のレジスタでは論理レジス
19 タ番号から物理レジスタ番号への写像は不変であるため、両者の違いは意識されないが、rotating
20 register ではその写像を命令 rotate.f でプログラマが操作できるのである。

```
1        for(i = 0; i < 100; i++){
2                z[i] = 2.0*x[i]+y[i];
3        }
```

図 11.3: ループプログラムの例

表 11.1: 図 6.1 のプログラムのソフトウェア・パイプライン化コードの例

MC	命令	$t()$	$s()$
0	lfdf f40 = [r1],8	0	1
1	ldfd f60 = [r2],8	10	2
2	fmpy f50 = f1,f42	20	3
3	fadd f70 = f51,f62	30	4
4	stfd [r3],8 = f71	40	5
5	add r0 = r0,1	-	-
6	cmp.< r0,100	-	-
7	rotate.f	-	-
8	br.cond L1	-	-

このようなリネーミングのハードウェア機構はさほど複雑ではない[1]。プロセッサはリネーミングのための **Register Rename Base**（これは IA-64 での名称である、敢えて訳せばレジスタ改名基底値）と呼ばれる値 `rrb` を保持しており、ある命令が fn （$32 \leq n \leq 127$） にアクセスするときには、

$$(n - 32 + \texttt{rrb})\%96 + 32$$

という計算式で、論理レジスタ番号から物理レジスタ番号へ変換すればよいのである。命令 `rotate.f` は `rrb` の値をひとつ減らす。そうすると `rrb` 変更後に、変更前と同じレジスタにアクセスするためには、論理レジスタ番号をひとつ増やして指定せねばならない。よって、`rotate.f` の実行前に `f32` として指定した物理レジスタは実行後には `f33` として指定せねばならない。

11.2　rotating register を用いたソフトウェア・パイプライン化

rotating register がソフトウェア・パイプライン化にどのように寄与するか、それを明らかにするために、ここまで何度も用いたループプログラム（図 11.3）を rotating register を用いてソフト

[1] ハードウェア実装上の問題をひとつ挙げておこう。まさにリネーミングしている瞬間に命令例外割り込み（0 で除算したなど）が起きたとする。そのとき、いったいどのレジスタ値に起因して例外が発生したのか、レジスタの特定が難しくなるという問題がある。コンパイラ技術者はそういうことをあまり気にしないが、ハードウェア設計者には気になる問題である。

ウェア・パイプライン化する。

表 11.1 は、命令並列度 1 の場合の、既にスケジューリングを終えた、リソース予約表である。ただし命令のレーテンシは

命令	レーテンシ（MC）
`ldfd`	20
`stfd`	1
`fmpy`	5
`fadd`	3
`add`	1
`cmp`	1
`br.cond`	1

と約束する。ロード命令のレーテンシがこれまでになく大きな値に設定されていることを注意する。rotating register を用いない 8 章の方法でスケジューリングするならば、II が 20 以下では決してスケジューリングに成功しない。と言うのも、一般に rotating register を用いないスケジューリング法では II の値をスケジューリングする予定の命令の中の最大のレーテンシよりも小さくすることはできないからである。しかし rotating register を用いるならば II にそのような制約はない。

表 11.1 の 7 行目には `rotate.f` を配置した。結果、このコードでは、5 行目から 8 行目までがループ制御命令であり、以後この 4 行をループ制御部の慣用句として新たに使用する。またコードの可読性をよくするため、rotating register の論理レジスタ番号の使用では故意に間隔を空け、`f40`、`f41`、`f42`、`f50`、`f51`、`f60`、`f61`、`f70`、`f71` を使用した。

レジスタが使用される様子を見てみよう。たとえば 0 行目で定義された `f40` は 2 行目で `f42` として参照されている。何故ならば、ステージ番号 1 で定義したレジスタがステージ番号 3 で参照される間には 7 行目の rotate.f が 2 回実行され、論理レジスタ番号が 2 増えるからである。もしレジスタ番号が変化しないならば、ある iteration で使用中の `f40` は直前の iteration、直後の iteration での `f40` の使用と衝突してしまう。しかし rotating register では、`f40` $\rightarrow$ `f41` $\rightarrow$ `f42` という番号の変化によってその衝突を巧妙に回避できる。

表 11.2 は、$\texttt{i} = k$、$k+1$、$k+2$、$k+3$ のカーネル部の iteration に関連する命令のタイミング・チャートである。iteration 間でレジスタの衝突がないことを注意してほしい。

命令並列度 2 の場合のスケジューリング例を表 11.3 に示す。II は 5 である。パイプライン・ステージ数は 7 である。命令並列度が上がるとその分だけ II が小さくなるから、命令のレーテンシを隠すために多くのステージ数が必要となる。

表 11.2: 表 11.1 のタイミング・チャート

MC	i=k	i=$k+1$	i=$k+2$	i=$k+3$
0	lfdf f40 = [r1],8			
1				
2				
3				
4				
5				
6				
7	rotate.f			
8				
9		lfdf f40 = [r1],8		
10	ldfd f60 = [r2],8			
11				
12				
13				
14				
15				
16	rotate.f			
17				
18			lfdf f40 = [r1],8	
19		ldfd f60 = [r2],8		
20	fmpy f50 = f1,f42			
21				
22				
23				
24				
25	rotate.f			
26				
27				lfdf f40 = [r1],8
28			ldfd f60 = [r2],8	
29		fmpy f50 = f1,f42		
30	fadd f70 = f51,f62			
31				
32				
33				
34	rotate.f			
35				
36				
37				ldfd f60 = [r2],8
38			fmpy f50 = f1,f42	
39		fadd f70 = f51,f62		
40	stfd [r3],8 = f71			
...				

表 11.3: 図 6.1 のプログラムのソフトウエア・パイプライン化コードの例

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	lfdf f40 = [r1],8	0	1	ldfd f60 = [r2],8	5	2
1	fmpy f50 = f1,f44	21	5	fadd f70 = f51,f64	26	6
2	stfd [r3],8 = f71	32	7	add r0 = r0,1	-	-
3				cmp.< r0,100	-	-
4	rotate.f	-	-	br.cond L1	-	-

11.3　rotating register を用いたモジュロ・スケジューリング

8 章で述べたモジュロ・スケジューリング・アルゴリズムを rotating register を用いる場合に修正しよう。以下が全体のステップである。

0. 各命令で使用する浮動小数点レジスタを**仮想レジスタ**（virtual register）としておく。
1. ループ本体のデータ依存グラフを作成する。
2. データ依存グラフをもとに iteration 立ち上げ間隔 II を定める。
3. II をもとに空（カラ）のリソース予約表を作る。
4. データ依存グラフ中の全ての頂点について優先度を定める。
5. 優先度の高い頂点から順にリソース予約表に命令を埋めていく。これを全ての頂点について行う。
6. レジスタ割付を行う。
7. 命令スケジューリング結果からプロローグ部、カーネル部、エピローグ部のコードを生成する。

8 章との違いは、0. が新たに加わったこと、5. で失敗する場合がない点である [2]。

11.3.1　仮想レジスタ番号

rotating register を使用する場合、コードの実行と共にレジスタ番号が変化する。そのため、前章までのように、レジスタ番号をスケジューリング前に確定しておくことが困難である。そこで、命令中のレジスタ参照を f32、f33、. . . という形式ではなく、f@1、f@2、. . . の形式にしておく。@1、@2、. . . はレジスタ番号を意味する変数である。この変数の値はスケジューリング後のレジス

[2]ただし、ループ運搬依存がない場合に限る。ループ運搬依存がある場合には rotating register を用いてもスケジューリングに失敗する場合がありうる。

タ割り付けにおいて確定する。ただし GR や FR f0〜f31 の rotate しないレジスタはその限りでないから、スケジューリング前に確定してよい。

たとえば、この章の例題（図 11.3 のプログラム）では命令を以下のように仮定しておく。

```
        lfdf f@1 = [r1],8
        fmpy f@2 = f1,f@3
        ldfd f@4 = [r2],8
        fadd f@5 = f@6,f@7
        stfd [r3],8 = f@8
```

そして変数@1〜@8 の値はスケジューリング後に確定する。

11.3.2　データ依存グラフの作成

8.1 節と同じである。

11.3.3　iteration 立ち上げ間隔 II の決定、空のリソース予約表作成

8.2 節、8.3 節と異なるのは、新たに rotate.f がループ制御命令に加わった点である。その結果、II の計算法が修正される。グラフに頂点が k 個含まれるとき、命令並列度 1 の場合には $\mathrm{II} = (k{+}4)\mathrm{MC}$、命令並列度 2 の場合には $\mathrm{II} = ((k+1)/2+2)\mathrm{MC}$ となる。それに伴い、空のリソース予約表の作り方も rotate.f を加えた形に修正する。

11.3.4　頂点の優先度の決定

8.4 節と全く同じである。結果、我々の例題では図 11.4 のようなデータ依存グラフとその優先度になる。グラフ自体は 4 章と同じであるが、命令のレーテンシが異なる（この章では ldfd のレーテンシが極めて大きい）から、優先度パラメータが 4 章とは異なる。

11.3.5　命令スケジューリング

以下の通りである。手順 3(a) のみが 8.5 節と異なる。このアルゴリズムではスケジューリングは決して失敗しない。11.3.3 節で求めた II で必ずスケジューリングに成功する。

1. データ依存グラフ G から、先行頂点を持たない頂点を全て集め、スケジューリング候補集合 S とする。
2. 候補集合 S の中で最も優先度の高い頂点 n を取り出す。

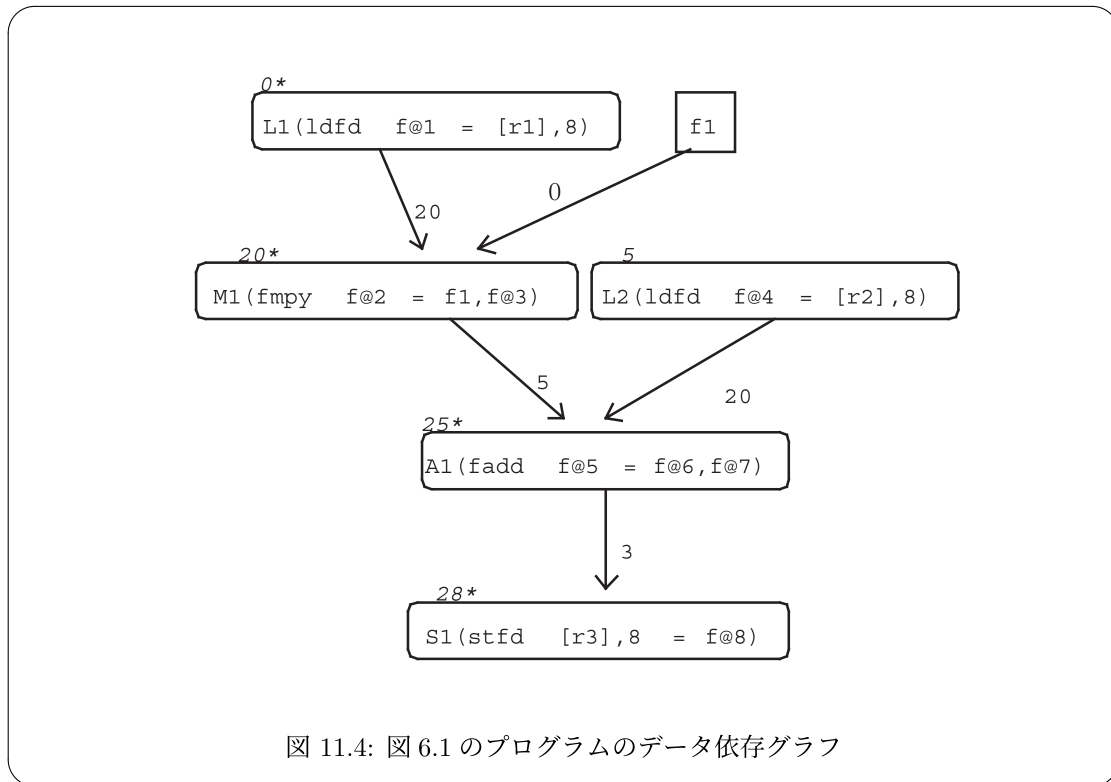

図 11.4: 図 6.1 のプログラムのデータ依存グラフ

3. 頂点 n について (a)、(b) のいずれかを適用し、n を表に埋め、かつ値 $t(n)$ を決定する。

   (a) もし既にスケジューリングされた頂点 m_1、...、m_k から n へのデータ依存が存在するならば、

      i. まず以下の値 start を求める。ここに L_1、...、L_k は m_1、...、m_k に対応する命令のレーテンシである。なお、rotating register を用いる場合には end を求める必要はない。

$$\text{start} = \max(t(m_1) + L_1, ..., t(m_k) + L_k)$$

      ii. 次に、表の start%II 行目に空白があるか否か調べる。もし空白があるならば、そこに n を埋める。さもなくば、順に (start+1)%II、(start+2)%II、...、(start+II−1)%II の各行を調べ、見つけた空白行に命令 n を埋める。もし n が $(\text{start} + j)$%II 行目に埋められたならば、$t(n) = \text{start} + j$ とする。

      iii. 最後に、頂点 m_i のターゲットレジスタの仮想レジスタ番号が`@x`ならば、それに対応する n のソース・レジスタの仮想レジスタ番号`@y`を以下の等式で計算するものとメモしておく。実際に等式の解を求める作業はレジスタ割り付けにおいて行う。

$$@\text{y} = @\text{x} + t(n)/\text{II} - t(m_i)/\text{II}$$

表 11.4: 図 8.2 のスケジューリング過程（命令並列度 3, II=3）

	#1	#2	#3
0	L1(0) *	L1(0) *	L1(0) M1(21) *
1	*	*	*
2	* *	L2(5) * *	L2(5) * *

	#4	#5
0	L1(0) M1(21) *	L1(0) M1(21) *
1	A1(28) *	A1(28) S1(31) *
2	L2(5) * *	L2(5) * *

表 11.5: 表 11.4 のコード（スケジューリング後）

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$	命令 3	$t()$	$s()$
0	lfdf f@1 = [r1],8	0	1	fmpy f@2 = f1,f@3	21	8	add r0 = r0,1	-	-
1	fadd f@5 = f@6,f@7	28	10	stfd [r3],8 = f@8	31	11	cmp.< r0,100	-	-
2	ldfd f@4 = [r2],8	5	2	rotate.f	-	-	br.cond L1	-	-

ここに $t(n)/\mathrm{II} - t(m_i)/\mathrm{II}$ は、それぞれ n と m_i のソフトウェア・パイプライン・ステージ番号の差である。

(b) さもなくば、n の優先度パラメータを p とし、表の $p\%\mathrm{II}$ 行目に空白のエントリがあるか否か調べる。もし空白があるならば、そこに n を埋める。さもなくば、順に $(p+1)\%\mathrm{II}$、$(p+2)\%\mathrm{II}$、...、$(p+\mathrm{II}-1)\%\mathrm{II}$ の各行を調べ、空白が見つかれば、n をその行に埋める。n が $(p+j)\%\mathrm{II}$ 行目に埋められたならば、$t(n) = p + j$ とする。

4. 頂点 n をグラフ G から取り除く。

5. もし G が空ならば、スケジューリングは終了。さもなくば 1. へ戻る。

例を示そう。表 11.4 は、命令並列度 3 の場合のスケジューリング過程である。表中の括弧（ ）内は $t()$ の値である。$t()$ の値が分かれば、式 $s() = t()\%\mathrm{II} + 1$ がその命令のステージ番号になる。∗はループ制御命令を表す。

スケジューリング結果を具体的な命令と $t()$、$s()$ で書き下したものは表 11.5 の通りである。

11.3.6　レジスタ割付

上記の手順 3 (a) iii. で述べたように、スケジューリングによって仮想レジスタ番号の間に等式関係が発生する。たとえば、表 11.5 のスケジューリングの場合、以下の通りである。

表 11.6: 表 11.4 のコード（レジスタ割り付け後）

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$	命令 3	$t()$	$s()$
0	lfdf f40 = [r1],8	0	1	fmpy f50 = f1,f47	21	8	add r0 = r0,1	-	-
1	fadd f70 = f52,f68	28	10	stfd [r3],8 = f71	31	11	cmp.< r0,100	-	-
2	ldfd f60 = [r2],8	5	2	rotate.f	-	-	br.cond L1	-	-

```
@3 = @1+7
@6 = @2+2
@7 = @4+8
@8 = @5+1
```

これらの等式が成り立つように各仮想レジスタ番号に 32 以上 127 以下の番号を割り付けねばならない。

我々の例題では、ひとつの割り付け例として、

```
@1 = 40、@2 = 50、@3 = 47、@4 = 60、@5 = 70、@6=52、@7 = 68、@8 = 71
```

と割り付けることができる。これらを代入したリソース予約表の内容は図 11.6 の通りである。

もし rotating register の使用数を最小化したいと思うならば、

```
@1 = 32、@2 = 39、@3 = 39、@4 = 41、@5 = 42、@6 = 41、@7 = 49、@8 = 43
```

と置いてもよい。レジスタ数最小化については次章でより詳しく述べる。

11.3.7　コードの生成

8.6 節と同じである。なお、表 11.6 のソフトウェア・パイプライン・ステージ数は 11 であるから、プロローグ部、エピローグ部のコード・サイズはそれぞれ $\mathrm{II} \times (11-1) = 3 \times 10 = 30$ 行近くに達する相当に大きなものになることを注意しておく。

11.4　さらなる高速化（loop unrolling）

この章のここまでの方法では、ひとつのループ中にループ実行を制御する 4 個の命令: `add r0 = r0,1`、`cmp.< r0,100`、`rotate.f`、`br.cond L1` が必要である。このコストが無視できない。たとえば表 11.1 の場合、ループ本体の命令は iteration 当たり 5 個しか含まれず、ループ制御命令がループ実行全体に占める割合は実に $4/(4+5) = 44\%$ に達する。これを解消する方法として **loop unrolling**（ループ展開）がよく知られている。

```
1        for(i = 0; i < 100; i+=2){
2            z[i] = 2.0*x[i]+y[i];
3            z[i+1] = 2.0*x[i+1]+y[i+1];
4        }
```

図 11.5: 図 11.3 のプログラムの 2 倍展開例

```
1        for(i = 0; i < 100; i+=4){
2            z[i] = 2.0*x[i]+y[i];
3            z[i+1] = 2.0*x[i+1]+y[i+1];
4            z[i+2] = 2.0*x[i+2]+y[i+2];
5            z[i+3] = 2.0*x[i+3]+y[i+3];
6        }
```

図 11.6: 図 11.3 のプログラムの 4 倍展開例

この技法を簡単に説明すれば、図 11.3 のプログラムを図 11.5 のように変換することである。これを 2 倍展開と呼ぶ。4 倍展開は図 11.6 の通りである。もしループ長が展開数で割り切れないときには端数処理が必要となる。上の例では、100 は 2、4 で割り切れるため、その必要はない。

図 11.5 のプログラムの場合、ひとつの iteration に 10 個の命令が含まれるから、ループ制御命令の実行がループ実行全体に占める割合は $4/(4+10) = 29\%$である。展開しない場合に比べた実行時間の短縮は $((4+10)*50)/((4+5)*100) = 78\%$である。同じく、図 11.6 の場合にはそれぞれ $4/(4+20) = 17\%$、$((4+20)*25)/((4+5)*100) = 67\%$となり、その効果は大きい。最近の多くの商用コンパイラ（gcc を含む）はループ本体が小さい場合には積極的に loop unrolling を行うようである。

ただし、展開数に反比例してループ長自体は短くなることに注意が必要である。ループ長が短くなると、ソフトウェア・パイプライン実行そのものの効率が落ちるから、その辺のバランスには注意が必要である。また消費するレジスタ数は展開数にほぼ正比例して増えるから、レジスタ数が少ないプロセッサではその注意も必要である [3]。

[3] レジスタ数が不足した場合、データを一時的にメモリへ待避させるのが一般的だが、そうすると、メモリアクセスによる実行時間の低下が問題になってくる。

第12章　rotating registerへのレジスタ割り付け

rotating registerを用いてソフトウェアパイプライン化されたループについてレジスタ割付けの方法を概説する。

例題として、前章で天下り的に与えた命令並列度1のパイプライン化コード（表11.1）を用いる。9章と同様に、プロローグ部、エピローグ部のレジスタ割付けは、カーネル部のレジスタ割付けの結果をそのまま用いることができるので、カーネル部だけを考えればよい。

仮想レジスタの概念は5章と変わらない。前章では`f@`nという形式の仮想レジスタも用いたが、ここでは5章、9章の議論を踏襲し、仮想レジスタ名は`v`nとする。そのために、形式的に、表11.1の`f40`、`f50`、`f60`、`f70`などを`v40`、`v50`、`v60`、`v70`へ置換しておく。置換して得られたコードが表12.1である。

12.1　らせん生存区間グラフ

rotating registerでは、iterationを経る毎に（`rotate.f`を実行する毎に）論理レジスタ番号が1ずつ増えて行くため、生存区間の有り様は5章、9章とかなり異なる。

たとえば表12.1の`v40`を考えてみよう。$t = 0$ (MC）で定義され（ロード命令のターゲットレジスタとなり）、$t = 20$ (MC) で参照されている（乗算命令のソースレジスタとなっている）。この間、レジスタ番号は`v40` $\rightarrow$ `v41` $\rightarrow$ `v42`と変化していく。これを9章で用いた生存区間グラフに表わすならば、図12.1のように**らせん**（spiral）構造として表わすのが、この生存区間の様子を最もうまく表わしている。これを**らせん生存区間**（spiral live range, spiral live interval）と呼ぼう。仮想レジスタ`v50`、`v60`、`v70`も同様に図12.1にあるようならせん構造として表現できる。また、らせん生存区間からなるグラフを**らせん生存区間グラフ**と呼ぼう。なお、図12.1のグラフの縦軸の範囲は本来は0 MC〜8 MCであるが、5 MC 〜 8 MCについては、その範囲で生存区間が始まったり、終わったりすることはないため、まとめて1行に省略した。

らせん生存区間のレジスタ割り付けのアイディアは単純である。複数のらせん構造をできるだけ隙間が小さくなるように連結するという問題とみなせるからである。

例題について述べる。

表 12.1: 図 11.3 のプログラムのソフトウェア・パイプライン化コードの例（再掲）

MC	命令	$t()$	$s()$
0	lfdf v40 = [r1],8	0	1
1	ldfd v60 = [r2],8	10	2
2	fmpy v50 = f1,v42	20	3
3	fadd v70 = v51,v62	30	4
4	stfd [r3],8 = v71	40	5
5	add r0 = r0,1	-	-
6	cmp.< r0,100	-	-
7	rotate.f	-	-
8	br.cond L1	-	-

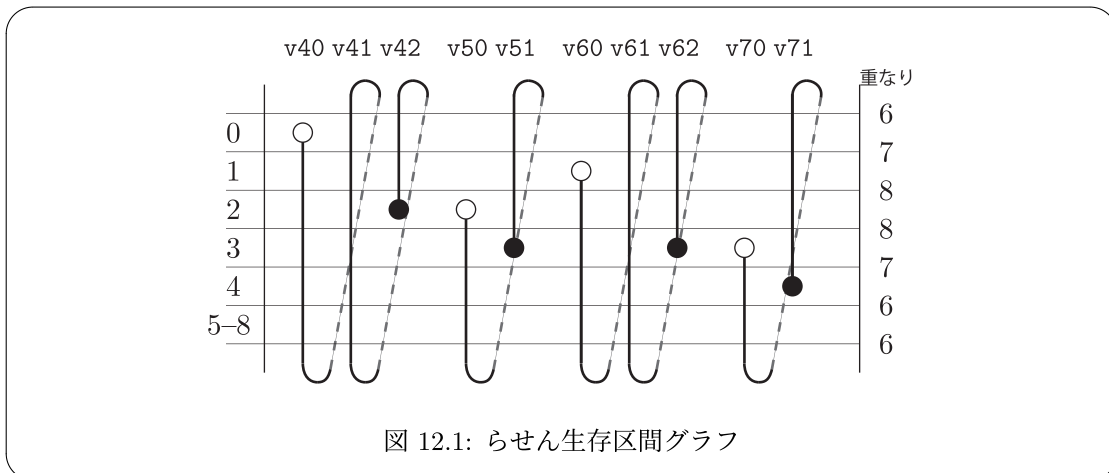

図 12.1: らせん生存区間グラフ

今、仮想レジスタ v40 を出発点として、それにできるだけ小さな隙間で連結できる仮想レジスタを探す。v50 を連結する場合、図 12.2 のように v40 の終了時点と v50 の開始時点が重なり、全く隙間のない（無駄のない）レジスタの使用が可能である。しかし、もし v70 を連結するならば、図 12.3 のように v40 の終了時点と v70 の開始時点の間に 1MC の隙間（無駄）ができてしまう。

rotating register におけるレジスタ割り付けとはこのような隙間（無駄）をできるだけ小さくする問題となる。

結局、上の例題の場合、図 12.4 のような割り当てが最適であり、必要レジスタ数は 8 本である。

ここで図 12.1 または図 12.4 のグラフの右に記載されている仮想レジスタ（またはらせん生存区間）の重なりについて注目しよう。重なりの最大値は 8 である。そして、最適なレジスタ割り付けのレジスタ本数は 8 である。この二つには強い関連があり、

- 最適なレジスタ割り付けで必要とされるレジスタの本数は、仮想レジスタ（またはらせん生存区間）の重なりの最大値＋1 を越えない。

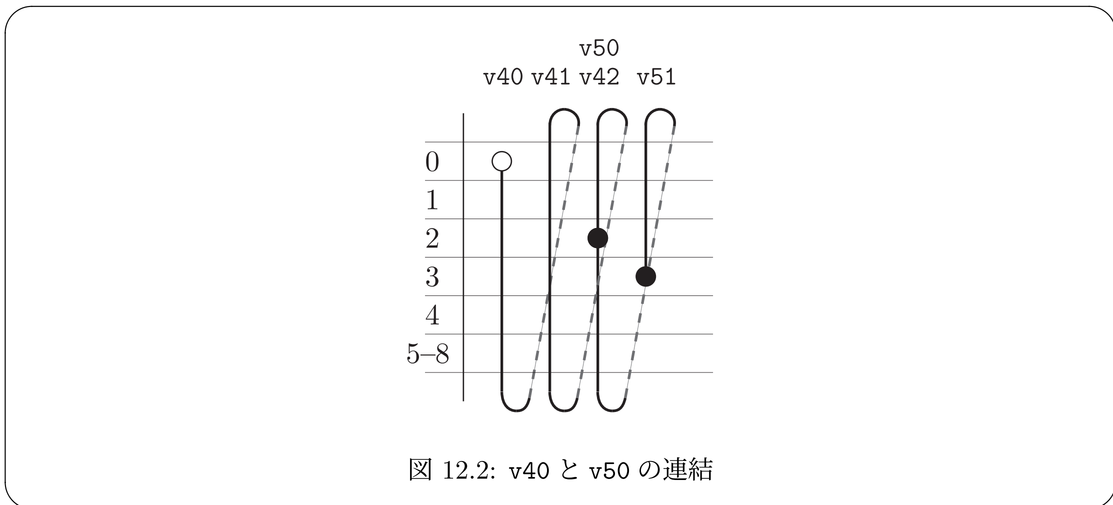

図 12.2: v40 と v50 の連結

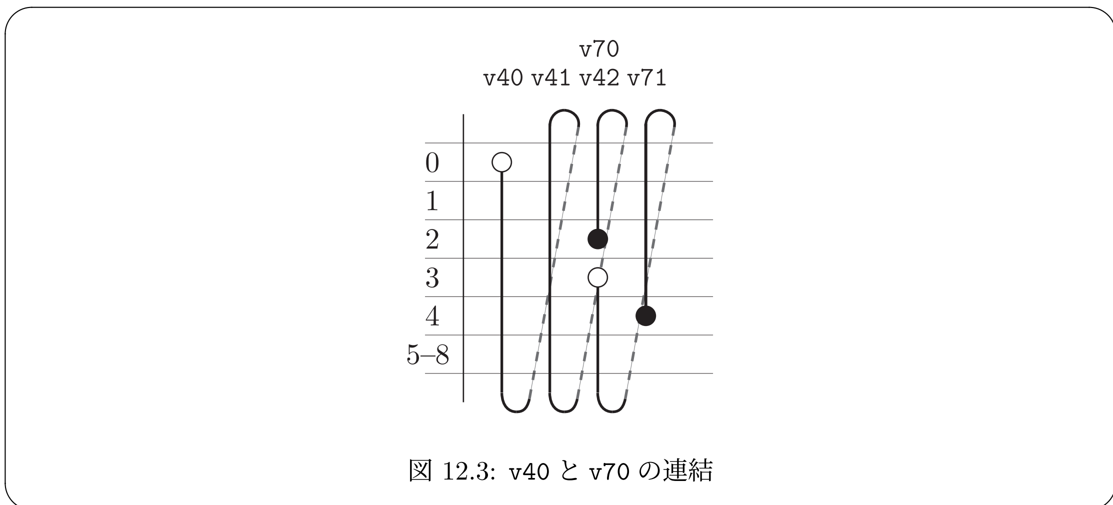

図 12.3: v40 と v70 の連結

ことが分かっている。ここに「＋１」は、いわゆる端数の処理のために必要とされる 1 本である。図 12.5 にその例を示す。図 12.5 の場合、重なりの最大値は３であるが、レジスタは 4 本が必要である。

　また、最適なレジスタ割り付けを行なうには、単純に個々のレジスタ間の隙間が最小となるように仮想レジスタを連結すればよいことも分かっている [1]。よって連結処理は仮想レジスタの数に比例する時間で終了できることとなり、rotating register の最適なレジスタ割り付け問題は多項式時間問題であることになる。

　これらの計算量的性質は、論理レジスタ番号が動的に変化していくという rotating register の見かけの複雑さとは逆に、そのレジスタ割り付け問題が非常に（！）扱い易い問題であることを示している。

　これについては次のような理解もできる。1 本の「らせん」は、その渦をほぐすと、1 本の「線

[1] これを証明するには、少し煩雑な数学的準備が必要となるが、本質的に難しい証明にはならない。

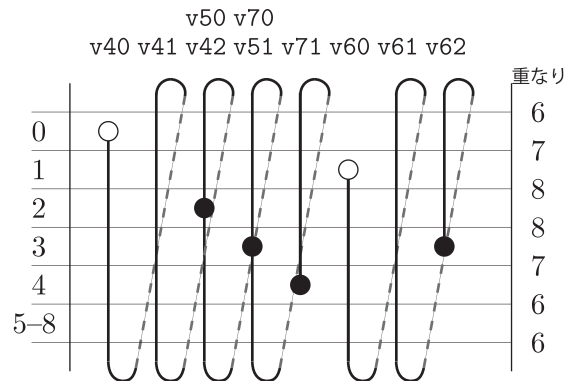

図 12.4: 最適なレジスタ割り付け

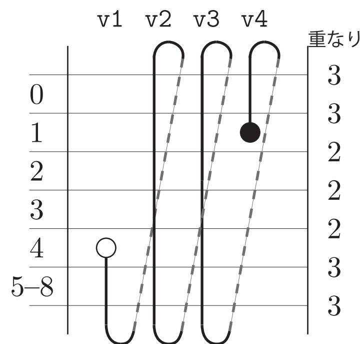

図 12.5: 必要レジスタ数が重なり＋1となる場合

分」になる。「らせんとらせん」を連結することは、その渦をほぐすと、「線分と線分」を連結することと等しい。そう見なすならば、実は、rotating register を用いたループのレジスタ割り付け問題は、5 章の基本ブロックのレジスタ割り付け問題と本質的に等価なのである。よって多項式時間で最適に解くことができる。

第13章　条件分岐を含むループのソフトウェア・パイプライン化（その1）：Hierarchical Reduction

ループ本体が基本ブロックであるようなループのソフトウェア・パイプライン化技法は、モジュロ・スケジューリングと loop unrolling でおおむね十分と考えてよい。特に rotating register が使用できるならば、常に最小の II でスケジューリングできるため、最速なコードを生成できる。

この章では、ここまでの技法をさらに拡張し、ループ本体に条件分岐節（if-then-else 節）が現れる場合の最適化を検討する。

これを扱う技法として、以下の三つが知られている。

Hierarchical Reduction (HR) ... if-then-else 節全体をひとつの大きな疑似命令と見なし、ソフトウェア・パイプライン化を行う。

Predicated Execution (PE) ... predicate register という特殊なハードウェアを用いる。

Enhanced Modulo Scheduling (EMS) ... predicate register を用いないで実現する。

この三つの中では HR は多くの場合に PE、EMS よりも劣ると考えられている。PE と EMS の比較では、then/else 節の本体プログラムが比較的小さく、条件分岐のコストが大きいときには PE が優れ、さもなくば EMS が優れている。ここではまず、条件分岐を含むループの紹介を兼ねて HR について述べる。次章において PE を解説し、次々章において EMS を述べる。

13.1　条件分岐を含むループ

図 13.1 は if 構文をひとつ含むプログラムの例である。ここに命令並列度 1 とし、命令のレーテンシを以下のように仮定する。

命令	レーテンシ（MC）
ldfd	5
stfd	1
fmpy	3
fadd/fsub	2

```
1        for(i = 0; i < 100; i++){
2            if(a[i] > 0.0){
3                x[i] = 2.0*y[i]*z[i];
4            }else{
5                x[i] = y[i]+z[i];
6            }
7        }
```

図 13.1: 条件分岐を含むループ・プログラムの例

fcmp	2
add	1
cmp	1
br.cond	1

図 13.1 のプログラムを素朴にコード化すると図 13.2 の通りである。ただし a[i]、x[i]、y[i]、z[i] のアドレスはそれぞれ r1、r2、r3、r4 に格納されており、浮動小数点数 0.0、2.0 は f0、f1 に事前に格納されているとする。簡単のため、rotating register は用いていない。

13.2　HR によるソフトウエア・パイプライン化

ソフトウェアパイプライン化の準備のために、図 13.2 のコードについて then 部と else 部を横に並べ、図 13.3 のように表してみる。このコードの中の then 部と else 部:

```
*8                             br.cond L2
*9      fmpy f5 = f3,f4                                   L2: fadd f6 = f3,f4
*10     nop                                                   nop
*11     nop
*12     fmpy f6 = f1,f5
*13     nop
*14     br L3
```

をあたかもひとつの命令であるかのように扱い、この部分の構造が壊れないように注意し、その他の命令を nop 命令の箇所に埋めるソフトウエア・パイプライン化を行うと、図 13.4 のコードを得る。ここに、図 13.4 の*5 行目、*6 行目、*8 行目の命令は、then 部と else 部の両方へ同じ命令を振り分けたものである。

さて、図 13.4 のコードはこのままでは実行できない。そこで、図 13.2 から図 13.3 のコードを求めた場合の逆変形を行い、図 13.4 から図 13.5 を得る。これは、ソフトウエア・パイプライン化されたコードのカーネル部に他ならない。

```
 1 L1:    ldfd f2 = [r1],8        //a[i]
 2        lddf f3 = [r3],8        //y[i]
 3        ldfd f4 = [r4],8        //z[i]
 4        nop
 5        nop
 6        fcmp.<=  f2,f0          //a[i] <= 0 ?
 7        nop
 8        br.cond L2
 9        fmpy f5 = f3,f4         //y[i]*z[i]
10        nop
11        nop
12        fmpy f6 = f1,f5         //2.0*y[i]*z[i]
13        nop
14        br L3
15 L2:    fadd f6 = f3,f4         //y[i]+z[i]
16        nop
17 L3:    stfd [r2],8 = f6        //x[i]
18        add r0 = r0,1
19        cmp.<  r0,100
20        br L1
```

図 13.2: 図 13.1 のプログラムのコード例

このコードの実行の様子を検討しよう。もし $i = k-1$ の iteration で if 節の条件が成り立ち（すなわち、a[k-1] > 0 が成り立ち）、$i = k$ の iteration で if 節の条件が成り立たなかった（すなわち、a[k] > 0 が成り立たなかった）ならば、$i = k$ の iteration に関して実行されるコードは、図 13.6 の通りである。

このようにして、図 13.5 のコードの各 iteration は 12MC または 10MC 毎に次々と立ち上がって実行されていく。図 13.2 のコードについて同様に考察すれば、各 iteration は 18MC または 14MC 毎に立ち上がっていくから、結局、図 13.5 のコードは、図 13.2 のコードに比べ、1.4 倍（=14/10）から 1.5 倍（=18/12）の間の高速化が実現できていることになる。

HR（Hierarchical Reduction=階層的縮約）とは、その名前の通り、複数の命令が高度な構造を持つようなプログラムについて、その構造部分（階層部分）をあたかもひとつの命令であるかのように見なし（ひとつの命令に縮約し）、通常のソフトウエア・パイプライン化のアイディアを適用する手法である。条件節のような構造の場合にはその then 部と else 部をその形を保ったまま、ひとつの命令と見なす。HR の大きな欠点は、縮約した部分の内部にはソフトウエア・パイプライン化を適用できないことにある。そのため、コード全体に占める縮約部の比率が大きい場合には十分な高速化ができない。その部分の大きさがネックになってしまうのである。

HR は比較的単純な手法であるが、PE や EMS に劣る手法であるため、このテキストでは HR

```
*1                        L1: ldfd f2 = [r1],8
*2                            lddf f3 = [r3],8
*3                            ldfd f4 = [r4],8
*4                            nop
*5                            nop
*6                            fcmp.<=  f2,f0
*7                            nop
*8                            br.cond L2
*9      fmpy f5 = f3,f4                                 L2: fadd f6 = f3,f4
*10     nop                                                 nop
*11     nop
*12     fmpy f6 = f1,f5
*13     nop
*14     br L3
*15                       L3: stfd [r2],8 = f6
*16                           add r0 = r0,1
*17                           cmp.<  r0,100
*18                           br L1
```

図 13.3: 図 13.2 のコードの変形

1　を紹介するだけにとどめ、アルゴリズムを定式化することはしない。

```
*1                         L1: fcmp.<=  f2,f0
*2                             stfd [r2],8 = f6
*3                             br.cond L2
*4      fmpy f5 = f3,f4                                L2: fadd f6 = f3,f4
*5      ldfd f2 = [r1],8                                   ldfd f2 = [r1],8
*6      lddf f3 = [r3],8                                   lddf f3 = [r3],8
*7      fmpy f6 = f1,f5
*8      ldfd f4 = [r4],8                                   ldfd f4 = [r4],8
*9      br L3
*10                        L3: add r0 = r0,1
*11                            cmp.<  r0,100
*12                            br L1
```

図 13.4: 図 13.3 のコードの変形

```
 1 L1:     fcmp.<=  f2,f0
 2         stfd [r2],8 = f6
 3         br.cond L2
 4         fmpy f5 = f3,f4
 5         ldfd f2 = [r1],8
 6         lddf f3 = [r3],8
 7         fmpy f6 = f1,f5
 8         ldfd f4 = [r4],8
 9         br L3
10 L2:     fadd f6 = f3,f4
11         ldfd f2 = [r1],8
12         lddf f3 = [r3],8
13         ldfd f4 = [r4],8
14 L3:     add r0 = r0,1
15         cmp.<  r0,100
16         br L1
```

図 13.5: 図 13.4 のコードを実行可能な通常のコードへ変換

```
 1        *                  //第 1 ステージ
 2        *
 3        *
 4        *
 5        ldfd f2 = [r1],8
 6        lddf f3 = [r3],8
 7        *
 8        ldfd f4 = [r4],8
 9        *
14        #
15        #
16        #
-----------------------------------------------
 1        fcmp.<=  f2,f0     //第 2 ステージ
 2        *
 3        br.cond L2
10 L2:    fadd f6 = f3,f4
11        *
12        *
13        *
14        #
15        #
16        #
-----------------------------------------------
 1        *                  //第 3 ステージ
 2        stfd [r2],8 = f6
...
```

図 13.6: 図 13.5 のコードのタイミング・チャート（else 部を実行した場合）

第14章　条件分岐を含むループのソフトウェア・パイプライン化（その2）：述語付き実行

規模の小さい条件分岐節の実行においては分岐ペナルティ（5.4 節参照）が実行時間の大半を占め、実行速度が低下する場合がある。PE は、それを防ぐために、分岐命令そのものを無くしてしまう手法であるが、これにはプロセッサに特殊なハードウェアが必要である。この章はそのハードウェア機構の解説から入る。

14.1　predicate registers

我々が想定するプロセッサは **predicate register** （述語レジスタ）と呼ばれる特殊なレジスタを 128 個持つとし、このレジスタを `p0`、`p1`、...、`p127` で表す。各 predicate register は 1bit の情報（真/偽または 1/0）を持っている。簡単のため、この predicate register を単に**述語**と呼ぶこともある。この基本的な使用法は以下の通りである。

述語の参照　任意の命令 `inst` は、任意の述語 `pi` （$i = 0, ..., 127$）を用いて修飾し、以下のような形式で実行することができると約束する。

```
(pi)inst
```

この意味は「述語 `pi` が真ならば `inst` を実行し、`pi` が偽ならば `inst` を実行しない」というものである。このような実行を **predicated execution** （述語付き実行）と呼ぶ。述語で修飾されていない命令はこれまでと同じく常に実行されると考える。命令 `inst` は、述語 `pi` が偽ならば実行されないのだが、命令の実行直前までの準備（述語値のチェック、命令のフェッチ、デコードまで）はプロセッサ内で行われることを注意する。つまり述語が偽であり、命令を実行しない場合でも、この述語付き命令は一旦はプロセッサの命令パイプラインに投入され、実行直前までのプロセッサのリソースを消費することとなる。これは突き詰めて考えれば無駄である。しかし、そのような無駄を行っても高速なコードが生成できることを後に見る。

述語の定義　述語の値は以下の形式の比較命令で定義することができると約束する。

- `cmp.`*`rel`* `pi,pj = rm,rn`　（GR 間の比較）

```
1        if(a > 0.0){
2            x = 2.0*y*z;
3        }else{
4            x = y+z;
5        }
```

図 14.1: if 節を含むプログラムの例

1 • `cmp.`*rel* `pi,pj = rm,`*imm2*　（GR と整数定数の比較）

2 • `cmp.`*rel* `pi,pj = `*imm1*`,rn`　（整数定数と GR の比較）

3 • `fcmp.`*rel* `pi,pj = fm,fn`　（FR 間の比較）

4 これらは、関係演算子 *rel*（==、 !=、 >、 <、 >=、 <=）のもとで二つの数値を比較した結果が真
5 ならば、述語 pi に真値（true value）を、述語 pj に偽値（false value）を代入する。さもなくば、
6 逆に pi に偽値を、pj に真値を代入する。
7 　比較命令を述語修飾することもできる。

8 • `(pk)cmp.`*rel* `pi,pj = rm,rn`　（GR 間の比較）

9 • `(pk)cmp.`*rel* `pi,pj = rm,`*imm2*　（GR と整数定数の比較）

10 • `(pk)cmp.`*rel* `pi,pj = `*imm1*`,rn`　（整数定数と GR の比較）

11 • `(pk)fcmp.`*rel* `pi,pj = fm,fn`　（FR 間の比較）

12 これらは、述語 pk が真ならば比較命令を実行し、pi、pj に所定の値を代入する。さもなくば、pi、
13 pj に偽値を代入すると約束する（比較命令を実行しない訳ではないことを注意する）。これら述語
14 修飾の比較命令は条件分岐が入れ子をなすときに使用する。

15 　述語の使用例を紹介する。まず、図 14.1 の簡単なプログラムを検討しよう。これを命令並列度
16 1 で述語を用いないでコード化すると図 14.2 の通りである [1]。ただし、命令レーテンシは前章と
17 同じとする。a、x、y、z のアドレスはそれぞれ r1、r2、r3、r4 に格納されており、浮動小数点
18 数 0.0、2.0 は f0、f1 に事前に格納されているとする。このコードの 8 行目、14 行目に分岐命令
19 がある。特に 8 行目は条件付き分岐であり、分岐するか否か（条件式が成り立つか否か）によって
20 プロセッサの命令パイプラインに流し込む命令列が異なる。分岐するか否かは実際にコードを実行
21 するまで不明であるため、結局、この条件付き分岐の実行直後にプロセッサの実行が遅延し、パ

[1] このコードは、前章の図 13.2 の素朴なコードと類似している。

```
 1        ldfd f2 = [r1]     //a
 2        lddf f3 = [r3]     //y
 3        ldfd f4 = [r4]     //z
 4        nop
 5        nop
 6        fcmp.>  f2,f0
 7        nop
 8        br.cond L1
 9        fmpy f5 = f3,f4
10        nop
11        nop
12        fmpy f6 = f1,f5
13        nop
14        br L2
15 L1:    fadd f6 = f3,f4
16        nop
17 L2:    stfd [r2] = f6     //x
```

図 14.2: 図 14.1 のコード（predicate register を用いない場合）

```
 1        ldfd f2 = [r1]     //a
 2        lddf f3 = [r3]     //y
 3        ldfd f4 = [r4]     //z
 4        nop
 5        nop
 6        fcmp.>  p0,p1 = f2,f0
 7        nop
 8        (p0)fmpy f5 = f3,f4
 9        (p1)fadd f6 = f3,f4
10        nop
11        (p0)fmpy f6 = f1,f5
12        nop
13        nop
14        stfd [r2] = f6     //x
```

図 14.3: 図 14.1 のコード（predicate register を用いる場合）

イプラインにバブル（泡）が入り込む可能性がある。仮に 8 行目の平均分岐ペナルティを 5MC とし [2]、14 行目のペナルティを 0MC とする [3] ならば、図 14.2 のコードで then 部を実行した場合の実行時間は 20MC、else 部を実行した場合の実行時間は 16MC となる。

これに対して、図 14.3 は述語を用いたコードである。このコードには分岐命令がないため、分岐ペナルティが発生しない。結果、このコードの実行時間は then 部/else 部のどちらを実行しても 14MC であり、図 14.2 のコードよりも高速である。

上のように、コードから条件分岐を取り除き、その代わりに命令を述語で修飾する技法を一般に **if 変換**（if-conversion）と呼ぶ。

さらに例題を挙げる。図 14.4 のプログラムは、if 構文が入れ子をなす例である。これを if 変換し、コード化すると図 14.5 の通りである。ただし定数 1.0 は `f1` に事前に格納されているとする。8 行目において、もし `p0` が偽ならば `p3`、`p2` は共に偽になる。よってそのときには 9 行目、10 行目、12 行目は実行されないことを注意する。

条件分岐を含む（ループでない）プログラムのスケジューリングは、リスト・スケジューリング

[2] プロセッサは条件付き分岐命令を実行する直前にどちらに分岐するかを予測する。これを分岐予測と呼ぶ。予測が正しかったときには分岐ペナルティは 0MC に近い数字になるが、予測が間違った場合には命令パイプライン内を一掃し、新たに命令を詰め直す必要があるため、大きなペナルティとなる。ここではその平均のペナルティを 10MC とする。

[3] 無条件分岐では分岐予測は常に正しく行うことができるため、ペナルティはないと仮定する。

```
1     if(a > 0.0){
2         if(y <= 0.0){
3             x = 2.0*y*z;
4         }else{
5             x = y-z;
6         }
7     }else{
8         x = y+z;
9     }
```

図 14.4: 2 重の if 節を含むプログラムの例

```
1        ldfd f2 = [r1]     //a
2        lddf f3 = [r3]     //y
3        ldfd f4 = [r4]     //z
4        nop
5        nop
6        fcmp.> p0,p1 = f2,f0
7        nop
8        (p0)fcmp.<=  p2,p3 = f3,f0
9        (p1)fadd f5 = f3,f4
10       (p2)fmpy f6 = f3,f4
11       (p3)fsub f5 = f3,f4
12       nop
13       (p2)fmpy f5 = f1,f6
14       nop
15       nop
16       stfd [r2] = f5     //x
```

図 14.5: 図 14.4 のコード（predicate register を用いる場合）

と同様に行えばよい。たとえば図 14.4 のプログラムのデータ依存グラフは図 14.6 のようになる [4]。命令 C1 から C2、A2 への点線の枝は述語 p0、p1 に関する依存を表す。命令 C2 から M1、M2、A1 への点線の枝は述語 p2、p3 に関する依存である。

14.2 predicate register を用いたソフトウェア・パイプライン化

11 章の rotating register の議論を predicate register にも適用しよう。その場合、predicate register は rotating register でなければならない。そこで浮動小数点レジスタと predicate register を同時にリネーミングする命令を新たに以下のように導入する。

```
rotate.fp
```

この命令を実行する前のレジスタ番号:

f32、f33、...、f126、f127　と　p32、p33、...、p126、p127

は、実行後（発行の 1MC 後）には以下のレジスタ番号:

f33、f34、...、f127、f32　と　p33、p34、...、p127、p32

に一斉にリネーミングされると定める。

[4] 図 14.6 は純粋なデータ依存グラフではなく、部分的には制御依存グラフ（control dependence graph）あるいは制御フローグラフ（control flow graph）を含んでいるとする研究者もいる。しかし述語による依存はデータ依存に他ならず、図 14.6 には命令の実行順序を意図的に規定する如何なる制約も含まれていないから、データ依存グラフであると考えるのが妥当である。

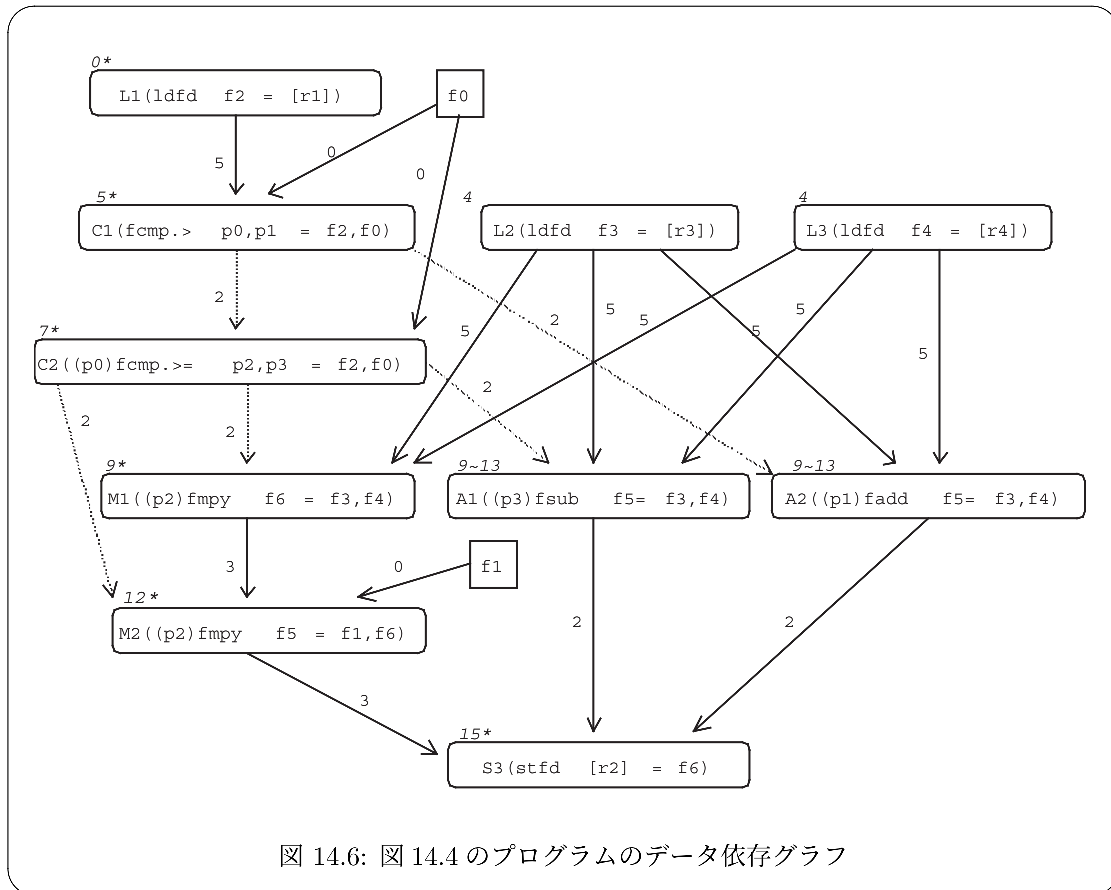

図 14.6: 図 14.4 のプログラムのデータ依存グラフ

ソフトウェア・パイプライン化の例を挙げよう。図 14.7 のプログラムは、図 14.1 のプログラムをループ本体とするようなループ・プログラムである。これを、ここまでと同じレーテンシを想定し、命令並列度 2 でモジュロ・スケジューリングした結果が表 14.1 である。このコードのタイミング・チャートを表 14.2 に示す。次節においてこれを導出するアルゴリズムを述べる。なお、ループ制御命令においても以下のように述語を使うことができる。

```
        cmp.<  p1,p0 = r0,100
        (p1)br L1
```

つまり、述語を用いるならば条件付き分岐命令は不要であり、その代わりに述語修飾の無条件分岐命令を用いる。この整数比較演算のレーテンシは 1MC とする。

表 14.2 のタイミング・チャートの 5 行目において、比較命令を実行し、述語 p40、p50 を定義している。この述語で修飾される命令はいずれも次のステージで実行されるため、p41、p51 という番号を用いて修飾している。このとき、もちろん、浮動小数点レジスタも rotate するため、ロード命令で f50、f60 に格納された値は、次のステージの乗算、加算命令では f51、f61 で参照される。

```
1        for(i = 0; i < 100; i++){
2            if(a[i] > 0.0){
3                x[i] = 2.0*y[i]*z[i];
4            }else{
5                x[i] = y[i]+z[i];
6            }
7        }
```

図 14.7: if 節を含むプログラムの例

表 14.1: 図 14.7 のプログラムのソフトウェア・パイプライン化コードの例

MC	命令	$t()$	$s()$
0	lfdf f40 = [r1],8	0	1
1	(p41)fmpy f70 = f51,f61	13	2
2	lddf f50 = [r3],8	2	1
3	ldfd f60 = [r4],8	3	1
4	(p51)fadd f80 = f51,f61	16	2
5	fcmp.> p40,p50 = f40,f0	5	1
6	(p41)fmpy f80 = f1,f70	18	2
7	stfd [r2],8 = f81	31	3
8	add r0 = r0,1	-	-
9	cmp.< p1,p0 = r0,100	-	-
10	rotate.fp	-	-
11	(p1)br L1	-	-

14.3　predicate register を用いたモジュロ・スケジューリング

述語修飾が追加された点を除けば、モジュロ・スケジューリング・アルゴリズムは 11 章とほとんど同じと考えてよい。以下が全体のステップである。

0. 各命令で使用する浮動小数点レジスタ、predicate register を仮想レジスタとし、さらに各命令を if 変換する。
1. ループ本体のデータ依存グラフを作成する。
2. データ依存グラフをもとに iteration 立ち上げ間隔 II を定める。
3. II をもとに空（から）のリソース予約表を作る。
4. データ依存グラフ中の全ての頂点について優先度を定める。

表 14.2: 表 14.1 のタイミング・チャート

MC	i=k	i=$k+1$	i=$k+2$
0	ldfd f40 = [r1],8		
1			
2	lddf f50 = [r3],8		
3	ldfd f60 = [r4],8		
4			
5	fcmp.> p40,p50 = f40,f0		
6			
7			
8			
9			
10	rotate.fp		
11			
12		ldfd f40 = [r1],8	
13	(p41)fmpy f70 = f51,f61		
14		lddf f50 = [r3],8	
15		ldfd f60 = [r4],8	
16	(p51)fadd f80 = f51,f61		
17		fcmp.> p40,p50 = f40,f0	
18	(p41)fmpy f80 = f1,f70		
19			
20			
21			
22	rotate.fp		
23			
24			ldfd f40 = [r1],8
25		(p41)fmpy f70 = f51,f61	
26			lddf f50 = [r3],8
27			ldfd f60 = [r4],8
28		(p51)fadd f80 = f51,f61	
29			fcmp.> p40,p50 = f40,f0
30		(p41)fmpy f80 = f1,f70	
31	stfd [r2],8 = f81		
32			
33			
34	rotate.fp		
35			
...			

5. 優先度の高い頂点から順にリソース予約表に命令を埋めていく。これを全ての頂点について行う。

6. レジスタ割付を行う。

7. 命令スケジューリング結果からプロローグ部、カーネル部、エピローグ部のコードを生成する。

以下、図 14.7 のプログラムを例題として各項目を述べていく。

14.3.1　if 変換、仮想レジスタ番号

図 14.7 のループ本体は図 14.1 のプログラムと同じ形状であるから、図 14.7 のプログラムで実行される命令は、図 14.3 のコードに現れる命令を仮想レジスタ化し、かつ if 変換したものである。つまり、以下の 8 個からなる。

```
ldfd f@1 = [r1],8
lddf f@2 = [r3],8
ldfd f@3 = [r4],8
fcmp.>  p#1,p#2 = f@4,f0
(p#3)fmpy f@5 = f@6,f@7
(p#4)fmpy f@8 = f1,f@9
(p#5)fadd f@10 = f@11,f@12
stfd [r2],8 = f@13
```

ここに、`@1`〜`@13` が浮動小数点レジスタの仮想レジスタ番号であり、`#1`〜`#5` が predicate register の仮想レジスタ番号とする。

14.3.2　データ依存グラフの作成

前章までと同じである。図 14.8 がデータ依存グラフである。

14.3.3　iteration 立ち上げ間隔 II の決定、空のリソース予約表作成

11 章と同じである。ここで命令並列度 2 を考えるならば、図 14.8 の中の命令数は 8 であるから、II = 6MC となる。

14.3.4　頂点の優先度の決定

11 章と同じである。我々の例題では図 14.8 に示す通りである。なお、命令 `A1` はクリティカル・パス上にないため、4 章の計算方法では優先度パラメータは一意に定まらず、7 から 11 の間の任意の数でよい。そこで、ここではその中間値 9 と定める。

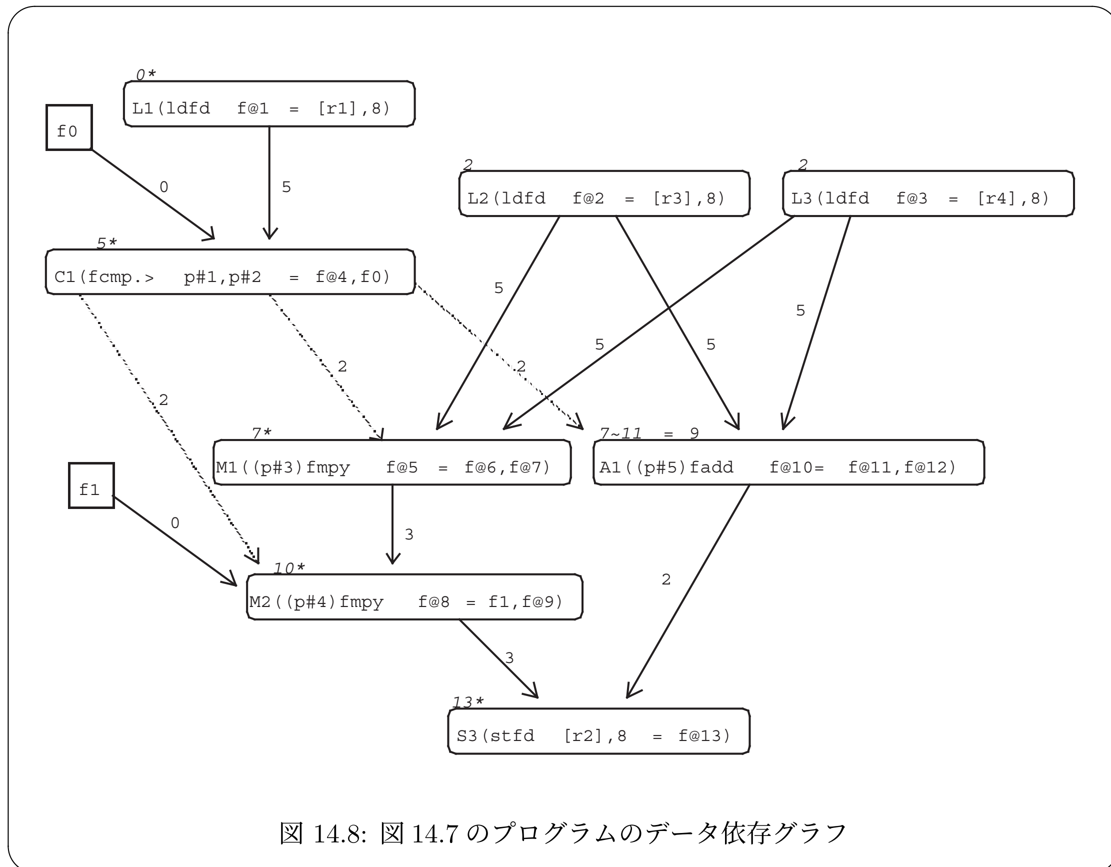

図 14.8: 図 14.7 のプログラムのデータ依存グラフ

14.3.5　命令スケジューリング

11 章と同じでよい。表 14.3 に、図 14.8 のデータ依存グラフをもとにしたスケジューリング過程を示す。結局、スケジューリング後のリソース予約表の内容は以下の通りである。括弧 ( ) 内はステージ番号（ = t()/II+1）を示す。

ここに仮想レジスタ番号には以下の制約が付く。

```
#3 = #1
#4 = #1+1
#5 = #2
@4 = @1+1
@6 = @2+1
@7 = @3+1
@9 = @5+1
@11 = @2+1
@12 = @3+1
@13 = @8+1 = @10+2
```

表 14.3: 図 14.8 のスケジューリング過程（命令並列度 2, II=6）

	#1		#2		#3		#4		#5		#6		#7		#8	
0	L1		L1		L1		L1	C1	L1	C1	L1	C1	L1	C1	L1	C1
1													M2		M2	S1
2			L2		L2	L3	L2	L3	L2	L3	L2	L3	L2	L3	L2	L3
3		*		*		*		*	M1	*	M1	*	M1	*	M1	*
4		*		*		*		*		*	A1	*	A1	*	A1	*
5	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

表 14.4: 表 14.3 のコード（スケジューリング後）

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	ldfd f@1 = [r1],8	0	1	fcmp.> p#1,p#2 = f@4,f0	6	2
1	(p#4)fmpy f@8 = f1,f@9	13	3	stfd [r1],8 = f@13	19	4
2	lddf f@2 = [r3],8	2	1	ldfd f@3 = [r4],8	2	1
3	(p#3)fmpy f@5 = f@6,f@7	9	2	add r0 = r0,1	-	-
4	(p#5)fadd f@10 = f@11,f@12	10	2	cmp.< p1,p0 = r0,100	-	-
5	rotate.fp	-	-	(p1)br L1	-	-

14.3.6　レジスタ割付

11 章と同じである。

我々の例題の場合、たとえば

#1 = 40、#2 = 50、@1 = 40、@2 = 50、@3 = 60、@5 = 70、@10 = 80

とすると、結果は表 14.5 になる。

レジスタ数を節約したいならば、

#1 = 32、#2 = 34、@1 = 32、@2 = 33、@3 = 36、@5 = 37、@10 = 34

とすれば、predicate register は 3 本、浮動小数点レジスタは 7 本で済む。結果は表 14.6 の通りである。

14.3.7　コードの生成

11 章と同じである。図 14.9 は、表 14.6 からコード生成したものである。

表 14.5: 表 14.3 のコード（レジスタ割り付け後）

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	ldfd f40 = [r1],8	0	1	fcmp.> p40,p50 = f41,f0	6	2
1	(p41)fmpy f81 = f1,f71	13	3	stfd [r1],8 = f82	19	4
2	lddf f50 = [r3],8	2	1	ldfd f60 = [r4],8	2	1
3	(p40)fmpy f70 = f51,f61	9	2	add r0 = r0,1	-	-
4	(p50)fadd f80 = f51,f61	10	2	cmp.< p1,p0 = r0,100	-	-
5	rotate.fp	-	-	(p1)br L1	-	-

表 14.6: 表 14.3 のコード（レジスタ割り付け後、レジスタ数節約）

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	ldfd f32 = [r1],8	0	1	fcmp.> p32,p34 = f33,f0	6	2
1	(p33)fmpy f35 = f1,f38	13	3	stfd [r1],8 = f36	19	4
2	ldfd f33 = [r3],8	2	1	ldfd 36 = [r4],8	2	1
3	(p32)fmpy f37 = f34,f37	9	2	add r0 = r0,1	-	-
4	(p34)fadd f34 = f34,f37	10	2	cmp.< p1,p0 = r0,100	-	-
5	rotate.fp	-	-	(p1)br L1	-	-

```
// プロローグ部
 1       ldfd f32 = [r1],8;        nop
 2       nop;                      nop
 3       ldfd f33 = [r3],8;        ldfd 36 = [r4],8
 4       nop;         add  r0 = r0,1
 5       nop;                      nop
 6       rotate.fp;                nop
 7       ldfd f32 = [r1],8;        fcmp.> p32,p34 = f33,f0
 8       nop;                      nop
 9       ldfd f33 = [r3],8;        ldfd 36 = [r4],8
10       (p32)fmpy f37 = f34,f37;  add  r0 = r0,1
11       (p34)fadd f34 = f34,f37;  nop
12       rotate.fp;                nop
13       ldfd f32 = [r1],8;        fcmp.> p32,p34 = f33,f0
14       (p33)fmpy f35 = f1,f38;   nop
15       ldfd f33 = [r3],8;        ldfd 36 = [r4],8
16       (p32)fmpy f37 = f34,f37;  add  r0 = r0,1
17       (p34)fadd f34 = f34,f37;  nop
18       rotate.fp;                nop
// カーネル部
19 L1:   ldfd f32 = [r1],8;        fcmp.> p32,p34 = f33,f0
20       (p33)fmpy f35 = f1,f38;   stfd [r1],8 = f36
21       ldfd f33 = [r3],8;        ldfd 36 = [r4],8
22       (p32)fmpy f37 = f34,f37;  add  r0 = r0,1
23       (p34)fadd f34 = f34,f37;  cmp.<  p1,p0 = r0,100
24       rotate.fp;                (p1)br L1
// エピローグ部
25       nop;                      fcmp.> p32,p34 = f33,f0
26       (p33)fmpy f35 = f1,f38;   stfd [r1],8 = f36
27       nop;                nop
28       (p32)fmpy f37 = f34,f37;  nop
29       (p34)fadd f34 = f34,f37;  nop
30       rotate.fp;                nop
31       nop;                      nop
32       (p33)fmpy f35 = f1,f38;   stfd [r1],8 = f36
33       nop;                      nop
34       nop;                      nop
35       nop;                      nop
36       rotate.fp;                nop
37       nop;                      nop
38       nop;                      stfd [r1],8 = f36
39       nop;                      nop
40       nop;                      nop
41       nop;                      nop
42       rotate.fp;                nop
```

図 14.9: 表 14.6 から生成されたコード

第15章　条件分岐を含むループのソフトウェア・パイプライン化（その3）: Enhanced Modulo Scheduling

15.1　述語付き実行の問題点

前章では predicate register を使った条件分岐を含むループのソフトウェア・パイプライン化を述べた。この方法は then/else 節が大きい場合にあまり有効ではない。

理由を述べよう。典型的な if-then-else 節を並列並列度 1 の場合には if 変換すると以下のような命令列が生成される。ここに `cinst1`、...、`cinsti` は条件判定のための命令群、`tinst1`、...、`tinstj` は then 部の命令群、`einst1`、...、`einstk` は else 部の命令群とする。

```
cinst1
...
cinsti
fcmp.rel p0,p1 = ...
(p0)tinst1
...
(p0)tinstj
(p1)einst1
...
(p1)einstk
```

もしこのループがたまたま then 部しか実行しないならば、`(p1)einst1`、...、`(p1)einstk` は全くの無駄である。無駄であるにも関わらず、その命令の実行には 1MC を費やすため、結果、全体の実行効率を落とすことになる。このとき、全命令に占める無駄な命令の率は $k/(i+j+k)$ である。もし then 部と else 部が同じ回数だけ実行されるならば無駄な命令の率は平均して $(j+k)/2(i+j+k)$ となる。もしさらに $i \ll j,k$ が成り立つとするならば、約 50%が無駄な命令となる。この無駄は大きい。

述語付き実行は、分岐による性能低下が問題とされるときに有効な最適化技法である。しかし、then/else 節の命令数がある一定数を越えるときには、分岐による性能低下は相対的に減少し、本来実行する必要のない命令による無駄なプロセッサ・リソースの浪費の比重が増えてくる。これを解決する方法のひとつが、この章で解説する Enhanced Modulo Scheduling（以下、EMS）である。

```
1        for(i = 0; i < 100; i++){
2            if(y[i] > 0.0){
3                x[i] = 2.0*y[i]*z[i]+1.0;
4            }else{
5                x[i] =(y[i]+2.0)*(z[i]+3.0);
6            }
7        }
```

図 15.1: if 節を含むループ・プログラム

15.2　Enhanced Modulo Scheduling の考え方

EMS は、その名前が示す通り Modulo Scheduling を条件分岐を含む場合に**拡張した**（enhanced）手法である。よって、ここまで述べてきたソフトウェア・パイプライン化のアイディアは EMS においても有効である。述語付き実行と異なるのは、then 部と else 部の命令を 選択的に 実行し、無駄な命令の発行は一切行わないことである。それを実現するため、EMS では

- then 部と else 部の命令間で命令対（pair）を作り、
- その命令対についてソフトウェア・パイプライン化を行い、
- コード生成時に、命令対の中の適切な命令を実際に生成する。

図 15.1 のプログラム例で具体的に述べよう。このプログラムは、EMS を説明するために意図的に作った例題である。まず図 15.1 のループ本体の中の if 節の条件判定部および then 部を実行するときの命令群を以下に示す。ただし配列 x、y、z のアドレスは r1、r2、r3 にそれぞれ事前に格納されているものとする。定数 0.0、1.0、2.0、3.0 は f0、f1、f2、f3 に格納されているものとする。その他のレジスタには仮想レジスタ番号を付ける。さらに predicate register としては p2、p3 を使用すると決めておく。

```
ldfd f@1 = [r2],8
ldfd f@2 = [r3],8
fcmp.> p2,p3 = f@3,f0         // @3 <-- @1
(p2)fmpy f@4 = f@5,f@6        // @5 <-- @1, @6 <-- @2
(p2)fmpy f@7 = f2,f@8         // @8 <-- @4
(p2)fadd f@9 = f@10,f1        // @10 <-- @7
stfd [r1],8 = f@11            // @11 <-- @9
```

ここに、//以降のコメント @i <-- @j は、f@j がリネーミングされ、f@i として参照されることを表す。同様に、条件判定部および else 部を実行するときの命令群は以下の通りである。

```
          ldfd f@1 = [r2],8
          ldfd f@2 = [r3],8
          fcmp.> p2,p3 = f@3,f0
          (p3)fadd f@12 = f@13,f2        // @13 <-- @1
          (p3)fadd f@14 = f@15,f3        // @15 <-- @2
          (p3)fmpy f@16 = f@17,f@18      // @17 <-- @12, @18 <-- @14
          stfd [r1],8 = f@11             // @11 <-- @16
```

さて、ここで新しい処理を行う。then 部にのみ現れる 3 個の命令と else 部にのみ現れる 3 個の命令について以下のような**命令対**（instruction pair）を作り、それぞれの命令対をひとつの**擬似命令**（pseudo-instruction）と見なす。ただし、命令対の中の述語修飾 (p2)、(p3) は以後使用しないため削除しておく。

```
          (fmpy f@4 = f@5,f@6; fadd f@12 = f@13,f2)
          (fmpy f@7 = f2,f@8;  fadd f@14 = f@15,f3)
          (fadd f@9 = f@10,f1; fmpy f@16 = f@17,f@18)
```

そして、ループ本体の命令群を新たに以下のように設定し直す。ただし、後々の都合上、`fcmp` に対応する分岐命令 (p3)`br L...` をあらかじめ追加しておく。

```
          ldfd f@1 = [r2],8
          ldfd f@2 = [r3],8
          fcmp.> p2,p3 = f@3,f0
          (p3)br L...
          (fmpy f@4 = f@5,f@6; fadd f@12 = f@13,f2)
          (fmpy f@7 = f2,f@8;  fadd f@14 = f@15,f3)
          (fadd f@9 = f@10,f1; fmpy f@16 = f@17,f@18)
          stfd [r1],8 = f@11
```

命令対をひとつの疑似命令と見なすと、上の命令群は基本ブロックと見なすことができる。基本ブロックならば、通常のモジュロ・スケジューリングを適用可能である。これが EMS のアイディアである。

図 15.2 は、命令間のデータ依存グラフである。ただし各命令のレーテンシは以下のように設定した。

命令	レーテンシ（MC）
ldfd	5
stfd	1
fmpy	3
faddf/sub	2
fcmp	2

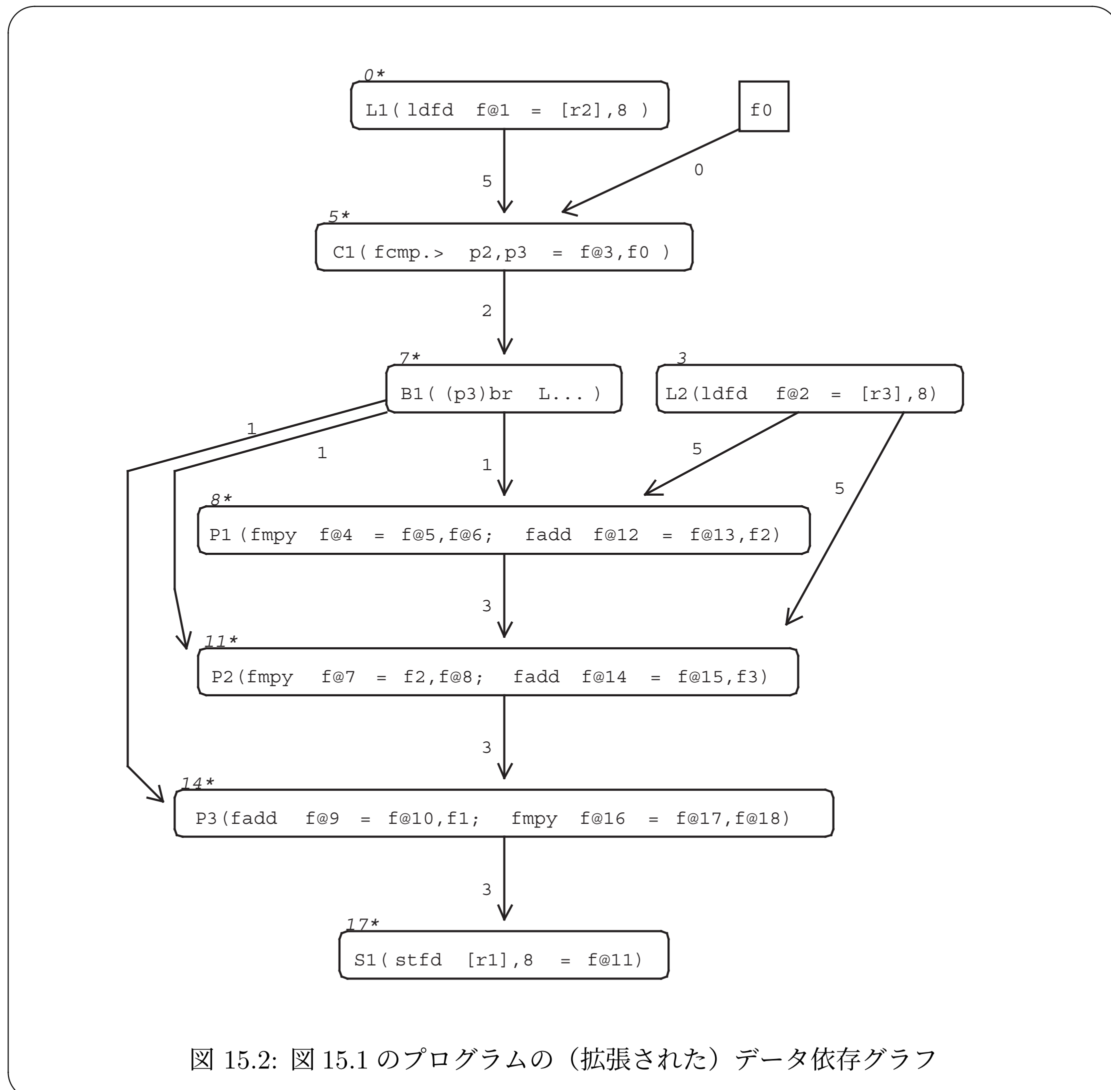

図 15.2: 図 15.1 のプログラムの（拡張された）データ依存グラフ

また分岐のペナルティは 0MC とする。よって、分岐命令 B1 から出る枝には 1MC というレーテンシが付いている [1]。命令対は分岐命令 B1 の後に発行すべきであるから、B1 から P1、P2、P3 への枝がある。

ところで、図 15.2 では命令対 P1 から命令対 P2 への枝は 1 本しか存在しない。しかし命令対に含まれる個々の命令に着目するならば、実際にはこの部分は図 15.3 のような依存グラフである。図 15.2 は、二つの頂点を合併し、依存の枝を単純化したものと考えればよい。

図 15.2 は基本ブロックと見なすことができるため、8 章（あるいは 11 章）のモジュロ・スケジューリングをそのまま適用できる。表 15.1 は、命令並列度 2 の場合のスケジューリング結果である。レジスタ割付も行った。

ところで表 15.1 には疑似命令である命令対が含まれるから、8 章のコード生成法は適用できな

[1] レーテンシは ペナルティ + 1 である。

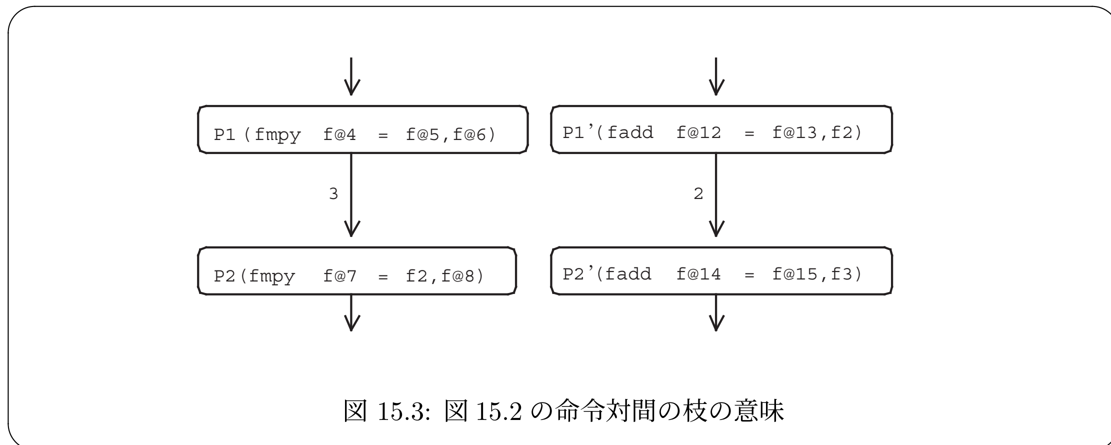

図 15.3: 図 15.2 の命令対間の枝の意味

表 15.1: 図 15.2 のスケジューリング結果（命令並列度 2）

MC	命令 1	$t()$	$s()$	命令 2	$t()$	$s()$
0	ldfd f40 = [r2],8	0	1	fcmp.> p2,p3 = f41,f0	6	2
1	(fmpy f55 = f2,f51; fadd f70 = f47,f3)	13	3	(fadd f60 = f56,f1; fmpy f60 = f67,f71)	19	4
2	stfd [r1],8 = f61	26	5	(p3)br L...	14	2
3	ldfd f45 = [r3],8	3	1	add r0 = r0,1	-	-
4	(fmpy f50 = f41,f46; fadd f65 = f41,f2)	10	2	cmp.< p1,p0 = r0,100	-	-
5	rotate.f	-	-	(p1)br L1	-	-

い。仮に適用してコードを生成しても、そのコードは実行できないコードである。

そこで EMS のコード生成を検討する前に、表 15.1 の内容を吟味してみよう。比較命令 `fcmp` はソフトウェア・パイプライン第 2 ステージの 0MC で実行されている。もし比較判定の結果が真であるならば、同じステージの 4MC では `fmpy f50 = f41,f46` を実行すべきであり、もし偽ならば `fadd f65 = f41,f2` を実行すべきである。第 3 ステージの 1MC では、その位置から数えて 7MC 前の `fcmp` の判定結果が真ならば `fmpy f55 = f2,f51` を、偽ならば `fadd f70 = f47,f3` を実行すべきである。第 4 ステージの 1MC では、その位置から数えて 13MC 前の `fcmp` の判定結果が真ならば命令 `fadd f60 = f56,f1` を、偽ならば `fmpy f60 = f67,f71` を実行すべきである。この「実行すべき」内容をコード生成時に実現すればよい。

表 15.1 には異なるステージ番号を持つ三つの命令対が含まれる。そして、それぞれの命令対の中のどの命令を実行すべきかは、三つの連続する iteration における判定結果に依存している。述語付き実行では判定結果を predicate register に保持したが、EMS では predicate register を用いる代わりに、判定結果を コード上の位置 として保持する。すなわち、表 15.1 に相当するコードを判定結果に応じて $2^3 = 8$ 個だけ用意し、実際に実行するコードを順次変えていくのである。その

ような考え方に基づいてコードを作る過程を**逆 if 変換**（reverse if-conversion）と呼んでいる。図15.4は表15.1に逆if変換を適用して得られたカーネル部のコードである。これでひとつのカーネル部を形成していることを注意する。非常に大きなコードであるから、後半は省略した。ラベル`Ltt`から始まるコードは、直前の比較判定結果が真、二つ前の比較判定結果が真であるようなコードに対応する。ラベル`Lft`から始まるコードは、直前の比較判定結果が偽、二つ前の比較判定結果が真であったコードに対応する。このように、ラベルが Lb_1b_2 という形式で始まるコードは、直前の比較判定結果が b_1 （=`t` （真）、または`f` （偽））、二つ前の比較判定結果が b_2（=`t` （真）、または`f` （偽））であったコードに対応する。同じく、ラベルが L$b_1b_2b_3$ という形式で始まるコードは、直前の比較判定結果が b_1（=`t` （真）、または`f` （偽））、二つ前の比較判定結果が b_2（= `t` （真）、または`f` （偽））、三つ前の比較判定結果が b_3（=`t` （真）、または`f` （偽））であったコードに対応する。

さて、Lb_1b_2 から実行を始めた場合、新たな比較判定によって、Ltb_1b_2 または Lfb_1b_2 のいずれかで始まるコードへ実行が進む。そして、次のiterationを進むときには、Ltb_1b_2 の方は Ltb_1 へ、Lfb_1b_2 の方は Lfb_1 へ分岐し、さらに実行を続ける。命令対のどちらを実行すべきかは、b_1、b_2、b_3 の値に応じ、選択すればよい。なお、ラベル`LttEpilogue`、`LftEpilogue`はカーネル部を終了し、エピローグ部へ分岐するときの飛び先である[2]。

たとえば判定結果が全て真であったとする。そうすると、実行パスは`Ltt` → `Lttt` → `Ltt` → `Lttt` → ... という軌跡を描く。もし判定結果が真偽を交互に繰り返すならば、実行パスは`Ltf` → `Lftf` → `Lft` → `Ltft` → `Ltf` → ... という軌跡を描く。これら軌跡をグラフ化したものが図15.5である。この遷移グラフはラベル`Ltt`、`Ltf`、`Lft`、`Lff`を頂点とする正方形であるが、ソフトウェア・パイプライン・ステージ数がより大きな数である場合や、ループに複数のif節が含まれる場合には、グラフはラベルを頂点とする**ハイパーキューブ**（hypercube）となる。

条件分岐によるペナルティが0であるならば、図15.4のコードの実行パスのiteration立ち上げ間隔IIは6MCである。同じ例題プログラムを述語付き実行した場合のIIは7MCになる[3]。よって、EMSでは述語付き実行に比べ、1MCだけ実行時間を短縮できる。この例では高々1MCだけしか実行時間は短縮されないが、then/else部の命令数が増えれば増えるほど、その差はさらに拡がる。また、条件分岐によるペナルティが0MCである[4]ならば、EMSは述語付き実行よりも常に効率の良いコードを生成できる。

[2]つまり、エピローグ部への出口は一カ所ではなく、カーネル部終了時の比較判定結果に応じて複数個ありうる。

[3]図15.2に含まれる命令は全部で10個である。ただし、命令対は2命令と見なす。ループ制御命令の数は4であるから、ループ内の命令数は計14となり、命令並列度が2の場合には、II= $(10+4)/2=7$ と計算できる。

[4]PowerPCでは比較演算と条件付き分岐命令を数MCだけ離して実行することによって、分岐ペナルティを0にできる。

```
Ltt:
        ldfd f40 = [r2],8;          fcmp.> p2,p3 = f41,f0
        fmpy f55 = f2,f51;          fadd f60 = f56,f1
        stfd [r1],8 = f61;          (p3)br Lftt
Lttt:
        ldfd f45 = [r3],8;          add  r0 = r0,1
        fmpy f50 = f41,f46;         cmp.< p0,p1 = r0,100
        rotate.f;                   (p0)br Ltt
        nop;                        (p1)br LttEpilogue
Lftt:
        ldfd f45 = [r3],8;          add  r0 = r0,1
        fadd f65 = f41,f2;          cmp.< p0,p1 = r0,100
        rotate.f;                   (p0)br Lft
        nop;                        (p1)br LftEpilogue

Lft:
        ldfd f40 = [r2],8;          fcmp.> p2,p3 = f41,f0
        fadd f70 = f47,f3;          fadd f60 = f56,f1
        stfd [r1],8 = f61;          (p3)br Lfft
Ltft:
        ldfd f45 = [r3],8;          add  r0 = r0,1
        fmpy f50 = f41,f46;         cmp.< p0,p1 = r0,100
        rotate.f;                   (p0)br Ltf
        nop;                        (p1)br LtfEpilogue
Lfft:
        ldfd f45 = [r3],8;          add  r0 = r0,1
        fadd f65 = f41,f2;          cmp.< p0,p1 = r0,100
        rotate.f;                   (p0)br Lff
        nop;                        (p1)br LffEpilogue

Ltf:
        省略
Lttf:
        省略
Lftf:
        省略

Lff:
        省略
Ltff:
        省略
Lfff:
        省略
```

図 15.4: 15.1 のプログラムの EMS のコード例

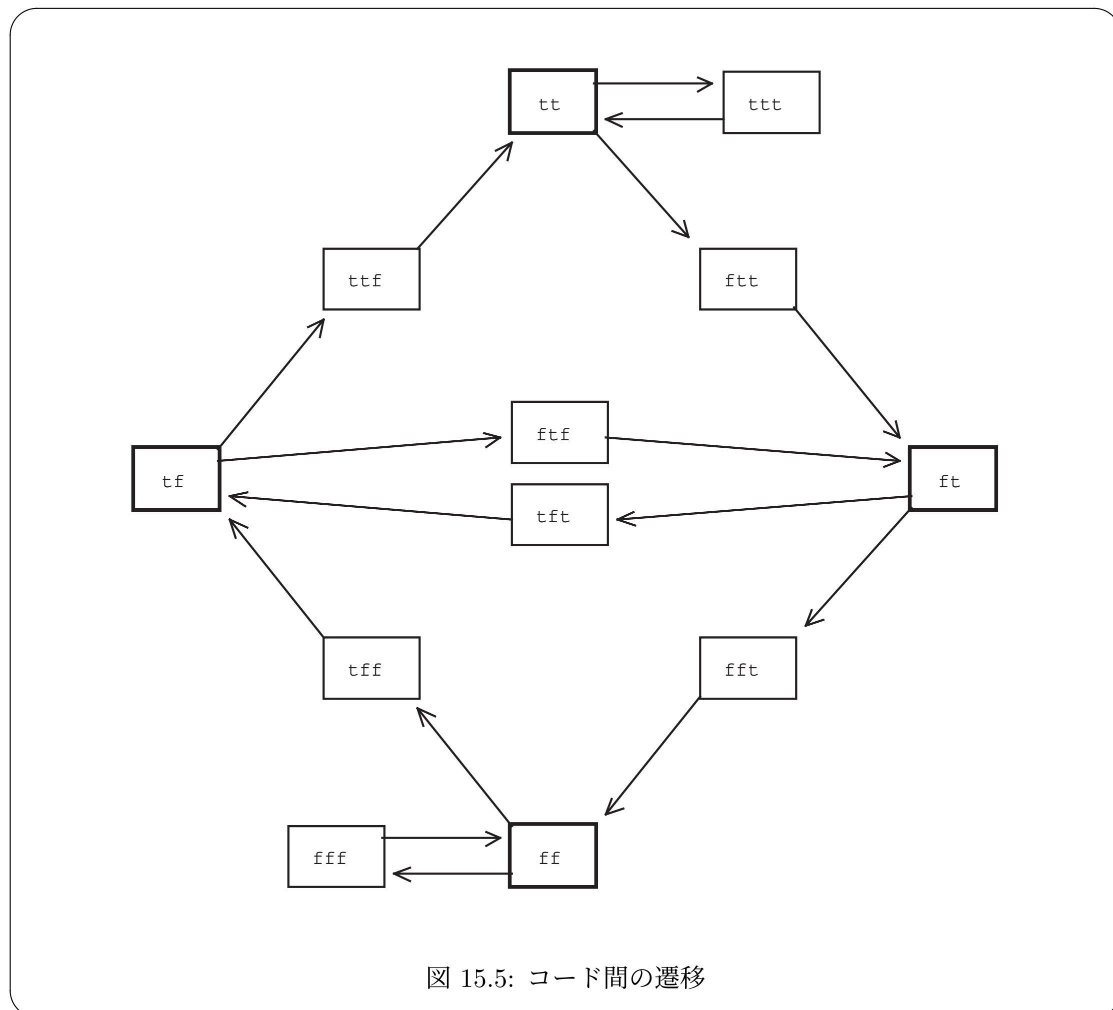

図 15.5: コード間の遷移

15.3　Enhanced Modulo Scheduling の問題点

EMS は、条件分岐を含むループのソフトウェア・パイプライン化において非常に効果的なコード最適化手法である。しかし幾つかの問題点もある。

- 分岐のペナルティが無視できない場合に十分な性能ができない。逆に、これは述語付き実行の特長である。分岐ペナルティの大きなプロセッサには EMS は不向きである。
- コード・サイズが指数的に増大する。連続する n 個の iteration の比較判定結果をコードの位置として保持する EMS の方法では、カーネル部全体のコード・サイズはリソース予約表・サイズの最大 2^{n-1} 倍に達する。巨大なコード・サイズは、命令キャッシュ（instruction cache）溢れの原因となり、結果として実行効率を落とす。仮にキャッシュが溢れないとしても、命令をキャッシュに載せるための最初の 1 回のミスヒット動作も無視できないかもしれない。

この二つの問題点は、EMS の本質に由来するため、アルゴリズムの改良による解決が困難であ

1 る。特に (1) はプロセッサの問題である。これに対し、以下は今後の研究課題である。

2 • 最適化アルゴリズムが非常に複雑である。特に、現実の商用 RISC プロセッサでは命令の同
3 時発行に様々な制約があり、この章の例題のように単純な議論ができない。

4 • then 部と else 部の演算量が異なる場合の扱いがさらにアルゴリズムを複雑にする。例題で
5 は、then 部、else 部ともに 3 個の命令であった。しかし多くの if-then-else 節では else 部が
6 ない場合がしばしば現れる。

参考文献

プロセッサ・アーキテクチャについて一通り学ぶには、以下が最も良いだろう。「パタヘネ本」と呼ばれている。

- デイビッド A パターソン、ジョン L ヘネシー：コンピュータの構成と設計 第5版 上下、日経BP社 (2014).

同じ著者らの名著（「ヘネパタ本」）：

- ジョン・L・ヘネシー、デイビッド・A・パターソン: コンピュータアーキテクチャ　定量的アプローチ 第5版、翔泳社 (2014).

はハードウェアの専門家向きという位置づけにあるようだ。

以下は邦人によるハードウェア専門書である。第一線で商用計算機を作ってきた著者の特徴がよく出た本で、大学関係者にとっては新鮮である。コード最適化についての記述もある。

- 中澤喜三郎：計算機アーキテクチャと構成方式、朝倉書店 (1995).

コンパイラとコード最適化全般を知るには以下がある。この授業と重複する内容も多い。

- 中田育男：コンパイラの構成と最適化、朝倉書店 (1999).

以下は、誤植が多いが、内容的には良い。ソフトウェア・パイプライン化に関する詳しい記述が載っている。同じシリーズに「... in Java」もあるが、「...in C」と内容はほとんど同じである。

- A. W. Appel : Modern Compiler implementation in C、Cambridge University Press(1998).

ソフトウェア・パイプライン化のサーベィは以下がある。ただし初学者にはとっつきにくい書き方である。

- V. H. Allans, R. B. Jones, R. M. Lee, and S. J. Allans : Software Pipelining, ACM Computing Survey, 27-3(1995)pp.367-432.

以下はrotating registerの発明者らによるコード最適化のサーベィである。この分野の概観に適している。同じ雑誌の同じ巻に関連サーベィがあり、そちらも読みごたえがある。

- B.R. Rau and J.A. Fisher : Instruction-Level Parallel Processing: History, Overview, and Perspective, Jour. Supercomputing, 7, 9-50(1993).

この授業で何度も言及し、また授業で用いた仮想プロセッサのモデルでもあるインテルIA-64アーキテクチャはインテルまたはHP社のwebページからPDFファイルがダウンロード可能である（各自で検索せよ）。邦訳もダウンロード可能である。まずはVolume1を読み込むことを薦める。述語付き実行、ソフトウェア・パイプライン化のとても良い教科書になっている。

- Intel IA-64 Architecture Software Developer's Manual, Volume 1: IA-64 Application Architecture

以前は日本訳（Intel(R)Itanium(TM) アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、第1巻：アプリケーション・アーキテクチャ）がインテルのサイトにあったが、現在は英語版しか取得できないようだ。

EMS の原論文は以下である。ただし 11 章はこの論文とは違う方法で説明した。

- N. J. Warter, G. E. Haab, and J. W. Bockhaus : Enhanced Modulo Scheduling for Loops with Conditional Branches, Proc. IEEE MICRO-25(1992)pp.170-179.

以下はこの授業に関係する講義担当者の論文である。

- Yoshiyuki Yamashita and Masato Tsuru : Rule Pattern Parallelization of Packet Filters on Muti-Core Environments, HPCC 2011 (2011 IEEE International Conference on High Performance Computing and Communications), IEEE Computer Society (2011) pp. 116–125. (Correction: Muti -¿ Multi)
- Yoshiyuki Yamashita and Masato Tsuru : Implementation and Evaluation of Fast Parallel Packet Filters on a Cell Processor, NDT2010 (The Second International Conference on Networked Digital Technologies), LNCS-CCIS 87 (2010) pp. 197–212.
- Yoshiyuki Yamashita and Masato Tsuru : Implementing Fast Packet Filters by Software Pipelining on x86 Processors, APPT09 (the 8th international Conference on Advanced Parallel Processing Technologies), LNCS 5737 (2009) pp. 420–435.
- Yoshiyuki Yamashita and Masato Tsuru : Software Pipelining for Packet Filters, HPCC07 (High Performance Computing and Communications), LNCS 4782 (2007), pp. 446–459.