情報処理学会論文誌　プログラミング　Vol.9　No.3　1–20 (June 2016)

RL78マイコン向けCコンパイラCC-RLにおける機種依存最適化の設計

千葉 雄司[1]　西村 啓成[2]　中川 満[2]

受付日 2015年11月18日, 採録日 2016年3月3日

概要：本論文の目的は，ルネサスエレクトロニクスのRL78マイコン向けCコンパイラCC-RLに実装した最適化処理のうち，RL78マイコン向けに特化したものの全体設計を示すことにある．一般に最適化コンパイラの設計にあたっては，コンパイラが生成するコードの実行を高速化する，あるいはサイズを削減するために，どんな最適化が必要で，その個々をどう実現し，さらに，実現した個々をどの順序で適用すべきかが問題になるが，本論文で示すのは，我々がRL78マイコン向けの実用的なアプリケーションのコンパイル結果の解析を通じて得た，この問題に対する包括的な解の1つである．

キーワード：コンパイラ，組込み，最適化

The Design of Target Dependent Optimizations in CC-RL, an Optimizing C Compiler for the RL78 Microcontroller

Yuji Chiba[1]　Masanari Nishimura[2]　Mitsuru Nakagawa[2]

Received: November 18, 2015, Accepted: March 3, 2016

Abstract: This paper shows the total design of the target machine specific optimization part of the C compiler CC-RL for the Renesas Electronics RL78 microcontroller. The design is one of the total solutions for optimizing compiler design problems to improve performance or to reduce size of the compiled code: what optimizations we need, how we should implement them, and in what order we should apply them. We solved the problems through analysis of compiled code of practical applications for the RL78 microcontroller.

Keywords: compiler, embedded, optimization code

1. はじめに

最適化コンパイラの開発にあたっては，どんな最適化が必要で，その個々をどう実現し，実現した個々をどの順序で適用すべきかが問題になる．この問題に対する解は，最適化の目的や，開発にかけられる期間，予算，開発にかかわる人員の練度など様々な要因によって変化するが，本論文ではルネサスエレクトロニクスのRL78マイコン [1] 向け最適化Cコンパイラ製品CC-RL [2] の開発において我々が得た解を提示する．

CC-RLの開発にあたってはルネサスのコンパイラインフラストラクチャを利用した．このコンパイラインフラストラクチャはオープンソースのコンパイラインフラストラクチャLLVM [3] のversion 2.3をベースに開発したもので，CC-RLの構成要素や個々の構成要素の役割はLLVMのそれと対応づけられる．CC-RLを構成するプログラムと，その役割，LLVMの構成要素との対応関係を次に示す．

コンパイラドライバ　後述するフロントエンドとバックエンドを駆動する．

フロントエンド　C言語で記述したソースプログラムを受理して，LLVMの中間表現であるビットコードをベースとした機種非依存な中間表現に変換する．

[1] 株式会社日立製作所
Hitachi, Ltd., Yokohama, Kanagawa 244–0817, Japan
[2] ルネサスシステムデザイン株式会社
Renesas System Design Co., Ltd., Kodaira, Tokyo 187–8588, Japan

バックエンド　LLVM のバックエンドは次に示す 2 つのプログラムからなるが，CC-RL のバックエンドはこれら 2 つを統合したものになる．

`opt`　ビットコードレベルでの最適化系．ビットコードを入力として受け取り，最適化を適用して，ビットコードを出力する．

`llc`　コード生成系，兼，目的機械依存の最適化を適用する最適化系．ビットコードを入力として受け取り，受け取ったビットコードをいったん DAG（Directed Acyclic Graph）形式の中間表現に変換して命令選択を行い，目的機械の機械命令と 1:1 に対応する中間表現 `MachineInstr` に変換し（本論文で単に命令選択と記述した場合，この DAG から `MachineInstr` への変換を意味するものとする），さらに，目的機械依存の最適化を適用して，最後に目的コードを出力する．

ルネサスのコンパイラインフラストラクチャは，ルネサスの RX/RH850 マイコン [4], [5] 向けコンパイラ製品 CC-RX/CC-RH [6], [7] でも利用しているもので，CC-RX/CC-RH/CC-RL はフロントエンドからバックエンドの `opt` までの構成要素を共有している．CC-RL の開発は，CC-RX/CC-RH より後に行ったため，CC-RL の開発を開始した時点で，他のコンパイラ製品と共有する最適化に関しては一定の品質を確保できていた．このため CC-RL の開発では，開発対象の最適化を DAG/`MachineInstr` レベルで適用するものに絞り込むことができた．本論文で示すのは CC-RL が `MachineInstr` レベルで適用する最適化群の全容である．

本論文の構成を次に示す．まず，2 章で RL78 マイコンの特性が要求する最適化を概観し，次に 3 章で RL78 マイコン向けに特化した最適化処理の全体設計について述べ，4 章で個々の最適化の効果を評価した結果を示す．続く 5 章では関連研究について述べ，最後に 6 章で結論を示す．

2. RL78 マイコンの特性が要求する最適化

RL78 マイコンは基本的にはアキュムレータマシンである．このため，RL78 マイコンが提供する命令の多くは，オペランドの 1 つをアキュムレータレジスタにすることを要求する．また，RL78 マイコンの命令セットでは，アキュムレータレジスタ以外のレジスタを割り付けるオペランドについても，任意のレジスタを割り付けられないことが多い．こういった制約は，RL78 マイコン向けコンパイラに特異な最適化の実装を要求する．

たとえば RL78 マイコンがデータ転送命令 `mov`/`movw` のオペランドに課している制約から，どんな最適化が必要になるか考察してみる．RL78 マイコンは，データ転送命令についても，そのオペランドの一方をアキュムレータレジスタにすることを要求しており，結果として RL78 マイコンのデータ転送命令は次の特長を持つ．

- 任意のレジスタ間のデータ転送をサポートしない．RL78 マイコンは 8 本の汎用レジスタ `a`，`x`，`b`，`c`，`d`，`e`，`h`，`l` を持ち，これらのレジスタを 2 つずつまとめて 16 bit の汎用レジスタ `ax`，`bc`，`de`，`hl` としても使用できるが，レジスタ間のデータ転送では転送元か転送先のどちらかをアキュムレータレジスタ `a` もしくは `ax` にする必要があるので，たとえばレジスタ `b` の内容を直接レジスタ `e` に転送することはできない．
- メモリから任意のレジスタへのロード，任意のレジスタからメモリへのストアもサポートしない．メモリからのロード先，メモリへのストア元になれるレジスタはアキュムレータレジスタのみである．

こういった特長は RL78 マイコン向けコンパイラで古典的なレジスタ割付けのアルゴリズムをありのままに利用することを困難にする．なぜなら古典的なレジスタ割付けのアルゴリズムの多くは次のことを前提条件としているのに対し，RL78 マイコンはいずれの条件も満さないからである．

- 命令のオペランドに任意のレジスタを割り付けられる．
- 任意の箇所で任意のレジスタをスピル/リストアできる．

ここで RL78 マイコンが任意の箇所で任意のレジスタをスピル/リストアできないとした理由は，RL78 マイコンではアキュムレータ以外のレジスタを直接スピル/リストアの対象にできないからである．

RL78 マイコン向けコンパイラの開発にあたっては，古典的なレジスタ割付けのアルゴリズムをありのままに利用できないことから，コンパイラインフラストラクチャに備わっているレジスタ割付けを利用することが難しく，独自のレジスタ割付けの実装が必要になるが，独自の実装が必要になるのはレジスタ割付けだけでなく，その前処理として行う命令スケジューリングについても同様である．レジスタ割付けの前に命令スケジューリングを行う目的は，たとえばレジスタプレッシャを下げることによってレジスタ割付けで挿入する退避や復帰の命令の数を減らすことにあるが，RL78 マイコン向けの命令スケジューリングでは，レジスタプレッシャのほかに，命令のオペランドに割り付けられるレジスタにも着目しないと退避や復帰の命令の数を減らせない場合がある．

たとえば**図 1** (a), (c) の中間表現について考える．図 1 (a) と (c) はレジスタ割付け前の中間表現であり，命令のオペランドをたとえば `%reg1034` などと表記しているが，ここで `%reg1034` は割り付けるレジスタがまだ確定していないレジスタ（仮想レジスタ）を表すものである．

図 1 (a), (c) の違いは 2 つのロード命令 `movw %reg1034, [%reg1033]` と `movw %reg1036, [%reg1035]` の並び順だけで，レジスタプレッシャに違いはないが，それらにレジ

```
movw %reg1034, [%reg1033]
movw %reg1036, [%reg1035]
movw [%reg1036], %reg1034
```

(a) スケジュール前

```
movw ax, [de]
movw de, ax
movw ax, [hl]
movw hl, ax
movw ax, de
movw [hl], ax
```

(b) (a) にレジスタ割付けを適用した結果

```
movw %reg1036, [%reg1035]
movw %reg1034, [%reg1033]
movw [%reg1036], %reg1034
```

(c) スケジュール後

```
movw ax, [hl]
movw hl, ax
movw ax, [de]
movw [hl], ax
```

(d) (c) にレジスタ割付けを適用した結果

図 1　命令スケジューリング（レジスタ割付け前）の効果
Fig. 1　Effect of pre-register-allocation scheduling.

スタを割り付けた結果（図 1 (b)，(d)）には並び順以外の違いがある．すなわち，図 1 (b) のコードには，メモリからアキュムレータレジスタに読み込んだデータを，レジスタ de に退避する命令と，レジスタ de からアキュムレータレジスタに復帰する命令が入っているのに対し，図 1 (d) のコードにはそれらがない．この違いが生じた原因は，仮想レジスタ %reg1034 の生存区間全体にアキュムレータレジスタを割り付けることができたか否かにある．図 1 (a)，(c) のコードはともに %reg1034 の参照箇所を 2 カ所持ち，そのいずれにもアキュムレータレジスタしか割り付けられないが，図 1 (c) のコードでは %reg1034 を定義する命令と使用する命令が連続していて，途中でアキュムレータレジスタの内容の変更が生じないので，%reg1034 の生存区間全体にアキュムレータレジスタを割り付けることができるのに対し，図 1 (a) のコードでは定義と使用の命令の間に別のロード命令が入っていて，そのロード先もアキュムレータレジスタにしかできないことから，%reg1034 の生存区間全体にアキュムレータレジスタを割り付けることができず，退避と復帰の命令の追加が必要になる．

このような退避や復帰の命令の挿入を回避するには，命令スケジューリングで図 1 (a) のコードを図 1 (c) のコードに書き換えればよいのだが，この書き換えは RL78 マイコンの命令のオペランドに割り付けられるレジスタに配慮しない命令スケジューラには実施できない．RL78 マイコンに固有な事情への配慮を必要とする最適化は他にもあるが，ここで RL78 マイコン向けコンパイラの設計にあたって問題になるのが，どんな最適化が必要で，それらをどの順序で適用すべきかということである．必要な最適化は RL78 マイコン向けアプリケーションのコンパイル結果を分析して定める必要があり，最適化の適用順序は個々の最適化の相互関係を考慮して定める必要がある．次章では，我々が RL78 マイコン向けの実用的なアプリケーションのコンパイル結果を分析した結果，必要と判断した最適化のうち，CC-RL が MachineInstr レベルで実施する最適化一式と，それらの相互関係を明らかにし，相互関係から導出した実施順序を示す．

3. RL78 マイコン向け最適化の設計

本章ではまず，CC-RL が MachineInstr レベルで適用する最適化の一覧を示し，次に，その個々の詳細と，なぜその順序で実施するのか，その理由を示す．

3.1 MachineInstr レベルでの最適化の適用順序

CC-RL が MachineInstr レベルで実施する処理一式を実施順序どおりに列挙する．

(1) 大域的な死亡コードの削除
(2) 冗長な 0 との比較の削除
(3) 大域的な共通部分式削除 [8]
(4) cmp *, 1/-1 から dec/inc *への書き換え
(5) 命令スケジューリング（レジスタ割付け前）
(6) レジスタ割付け
(7) 命令選択（レジスタ割付け後）
(8) スタックフレーム中のデータを参照する命令の具象化
(9) 分岐の畳込み/共通コードの下方集約 [9]
(10) 8 bit の定数ロードの融合
(11) レジスタ sp（スタックポインタ）へのアクセスの最適化
(12) 共通コードの関数化 [10]
(13) 命令選択（分岐命令選択前）
(14) 命令スケジューリング（分岐命令選択前）
(15) 分岐命令選択
(16) コード生成

ここで列挙した処理のうち (8)，(16) を除く処理が最適化である．これらの最適化のうち，(1)，(2)，(3)，(9) は実装をルネサスのコンパイラ製品 CC-RX/RH/RL で共用しているが，ほかは CC-RL 向けに新規開発したものである．

ここで示した最適化の適用順序は，CC-RL が提供する

```
%reg1025 = subw %reg1024, 5
cmpw %reg1025, 0
.bz LABEL
```

(a) 削除前

```
%reg1025 = subw reg1024, 5

.bz LABEL
```

(b) 削除後

図 2　冗長な 0 との比較の削除

Fig. 2　Redundant comparison to zero elimination.

```
%reg1027 = incw %reg1026
%reg1028 = incw %reg1027
cmpw %reg1028, 0
.bz LABEL
```

(a) 削除前

```
%reg1028 = addw %reg1026, 2


.bz LABEL
```

(b) 削除後

図 3　命令の再選択をともなう冗長な 0 との比較の削除

Fig. 3　Redundant comparison to zero elimination with instruction reselection.

2 つの最適化オプション，すなわち実行の高速化を目的とするオプションである `-Ospeed` と，コードサイズの削減を目的とするオプションである `-Osize` のどちらを指定した場合でも基本的には同じである．ただし，`-Ospeed` を指定した場合には，実行速度を犠牲にしてコードサイズを小さくする最適化である (12) の全体と (13) の一部を適用しない．

他の最適化については，どちらのオプションを指定した場合も適用する．その理由は，(14) が実行を高速化する一方でサイズに変化を与えないものの，残りは実行の高速化とサイズの削減の双方に有効で，いずれにしても一方の性能を改善するために他方の性能を犠牲にしないからである．

(1) から (16) の処理のうち，実施する処理が自明な (1)，(3) および (16) 以外について，その詳細を順次示す．

3.2　冗長な 0 との比較の削除

冗長な 0 との比較の削除は，定数 0 との比較命令のうち冗長なものを見つけて削除する最適化である．たとえば**図 2** (a) のコードについて考えると，0 との比較を行う命令 `cmpw %reg1025, 0` がプログラム・ステータス・ワード `PSW` の `Z`（ゼロ）フラグにおよぼす影響は，直前の命令 `subw` がもたらす影響と同一であり，このとき命令 `cmpw` の結果の利用者が参照する `PSW` 中のフラグが `Z` フラグだけならば命令 `cmpw` を冗長と判断して削除できる．冗長な 0 との比較の削除は，この削除を行う機能であり，LLVM では version 2.8 から利用できる．この機能を，LLVM の version 2.3 をベースとするルネサスのコンパイラインフラストラクチャで利用可能な理由は，LLVM の version 2.3 より後でリリースされた有意な機能は随時，ルネサスのコンパイラインフラストラクチャに取り込んでいるためである．

冗長な 0 との比較の削除は図 2 の例のように単純に比較を削除できるケースのみに対応しているわけではなく，命令選択をやりなおせば比較を削除できるケースにも対応している．たとえば**図 3** (a) のコードについて考える．図 3 (a) のコードで命令 `incw` は `PSW` を更新しないので図 3 (a) のコードから単純に命令 `cmpw` を削除するわけにはいかないが，図 3 のように命令 `incw` を命令 `addw` に選択しなおしてからであれば，命令 `addw` が `PSW` を更新するため，削除可能になる．このような命令選択のやりなおしをともなう冗長な 0 との比較の削除に関し，CC-RL がサポートする命令選択のやりなおしのパターンを**表 1** に示す．

表 1　冗長な 0 との比較の削除がサポートする命令の再選択のパターン

Table 1　Instruction reselection patterns for redundant comparison to zero elimination.

再選択前	再選択後
`incw * + incw *`	`addw *, 2`
`incw *`	`addw *, 1`
`decw *`	`addw *, -1`
`decw * + decw *`	`addw *, -2`
`clrb *.`*bit*	`and *,` $\sim(1 << bit)$
`shl *, 1`	`add *, *`
`shlw *, 1`	`addw *, *`

なお図 3 (a) のコードは 2 を加算する処理に対してわざわざ 2 つの命令 `incw` を選択しているが，これは 2 つの命令 `incw` の方が 1 つの命令 `addw ax, 2` よりコードサイズが小さいからで（命令 `incw`/`addw ax, 2` のコードサイズはそれぞれ 1/3 byte），コンパイル時の最適化の目的がサイズの削減にある場合にはこのような命令選択を行う．その一方で表 1 には 1 bit の左シフトを加算に変換するパターンがあり，このパターンでは左シフトの方がコードサイズが大きいことがある（16 bit の左シフト/レジスタどうしの加算のコードサイズはそれぞれ 2/1 byte，8 bit の場合はともに 2 byte）．それにもかかわらず CC-RL の命令選択では，値を 2 倍にする処理に対応する命令として 1 bit の左シフトを選択するが，その理由は左シフトの方が割り付けられるレジスタの選択肢が多いからである（加算の場合に選択できるレジスタは `a`/`ax` のみだが左シフトなら `b`/`c`/`bc` も選択できる）．CC-RL の命令選択では原則としてオペランドにできるレジスタの選択肢が多い命令を選び，レジスタ

割付けの結果，より小さな命令に書き換えられると分かった場合には，その時点で，より小さな命令に書き換える．この書き換えの処理については 3.6 節で述べる．

3.3　cmp *，1/-1 から dec/inc * への書き換え

CC-RL の命令選択では 8 bit のデータと定数を比較する中間表現に対応する命令として命令 cmp を選択するが，比較対象の定数が 1 もしくは-1 の場合，命令 cmp の代わりに命令 dec もしくは inc を利用可能な場合がある．

命令 dec/inc は定数との比較を行う命令 cmp に比べて次の点で優れている．

コードサイズ　命令 dec/inc のコードサイズが 1 byte であるのに対し，定数との比較を行う命令 cmp のコードサイズは 2 byte である．

オペランドにできるレジスタ　命令 dec/inc が任意の汎用レジスタをオペランドにできるのに対し，定数との比較を行う命令 cmp はアキュムレータレジスタ a しかオペランドにできない．

これらの利点があるにもかかわらず，命令選択では，8 bit の定数 1/-1 との比較命令として，命令 dec/inc を選択しない．その理由を次に示す．

- 命令 dec/inc はオペランドの内容を更新してしまうので，比較がオペランドの内容の最後の使用者でない場合に命令 dec/inc を使うのは適切でない．たとえば**図 4** (a) のコード列では命令 cmp a，1 の直後で，命令 clrb a によってレジスタ a の内容を 0 に書き換えていることから，命令 cmp a，1 はレジスタ a の 0 に書き換える前の内容の最後の使用者であるということができる．したがって，図 4 (a) のコード列で命令 cmp a，1 の代わりに命令 dec a を使っても問題はおきない．しかしながら，図 4 (b) のコード列では命令 cmp a，1 とその直後の命令がレジスタ a にある同じ内容を使用することから，図 4 (b) のコード列で命令 cmp a，1 の代わりに命令 dec を使うと問題がおきる．
- 命令 cmp *，-1 と命令 dec/inc *では PSW に与える影響が異なる．具体的には，命令 dec/inc *は PSW のフラグ CY を更新しないので，命令 cmp *，-1 の結果の利用者がフラグ CY を参照する場合には，命令 cmp *，-1 の代わりに命令 dec/inc *を使うことができない．

ここで示した 2 つの理由のうち，後者は命令選択の段階でも考慮できるが，前者を命令選択の段階で考慮するのは難しい．その理由を次に示す．

- 命令選択の時点では明示的な依存関係のない演算の実行順序が分からない．たとえば図 4 (b) の命令列では命令 cmp の次に命令 mov があるので命令 cmp の方を先に実行する，と判断できるが，命令選択の時点での中間表現は図 4 (b) に示した命令列のようなものではなく，演算をノード，データ依存もしくは制御依存をエッジとする DAG 形式のものであり，図 4 (b) の cmp と mov のようにどちらを先に実行してもよいノードの間にはエッジを張らないので，DAG 形式の中間表現をみても，どちらが先になるか分からない．
- 一般に命令がレジスタの内容の最後の使用者になるか否かはレジスタ割付けが終わるまで分からない．たとえばレジスタ割付け前の中間表現が**図 5** (a) であるとして，図 5 (a) の命令 cmp をみると，オペランドのレジスタ%reg1030 が直後の命令 mov のオペランドにもなっていることから，命令 cmp がレジスタ%reg1030 の内容の最後の使用者でないようにみえるが，レジスタ割付け後の中間表現（図 5 (b)）をみると，命令 cmp がレジスタ a の最後の使用者になっていることが分かる．なおレジスタ割付けの結果が図 5 (b) になる理由は，RL78 マイコンの命令 mov a，[hl+*] の*のオペランドに割り付けられるレジスタが b もしくは c のみだからである．

ここで命令 cmp *，1/-1 がオペランドのレジスタの内容の最後の使用者になるか否かが，レジスタ割付けの後まで確定しないのであれば，命令 cmp *，1/-1 から命令 inc/dec への書き換えはレジスタ割付けの後に実施すればよいのかというと，そうともいえない．なぜならレジスタ割付けの後で実施するのでは，命令 dec/inc の方がオペランドに割り付けられるレジスタが多いという利点を活かすことができないからである．このため CC-RL ではレジスタ割付けの前と後の両方で cmp *，1/-1 から dec/inc * への書き換えを行う．なおレジスタ割付け後の書き換えは 3.6 節で述べる命令選択（レジスタ割付け後）で実施する．

cmp a，1	cmp a，1
	mov [hl]，a
clrb a	clrb a
.bz LABEL	.bz LABEL
(a) 使える	(b) 使えない

図 4　dec *による cmp *，1 の代用の可否

Fig. 4　Availability to substitute dec * for cmp *, 1.

3.4　命令スケジューリング（レジスタ割付け前）

LLVM では DAG 形式の中間表現で命令選択を行った後，MachineInstr に変換する過程で命令スケジューリングを行い，その際にレジスタプレッシャを下げることで，レジスタ割付けで発生する退避や復帰を減らそうとする．しかしながら，2 章で述べたように，RL78 マイコン向けのコンパイルでは，レジスタプレッシャのみに着目して命令をスケジュールしても退避や復帰を減らしきれない場合がある．そこで CC-RL では別途，RL78 マイコンの命令がオペランドに割り付けられるレジスタを考慮した命令スケジューリングを行って，退避や復帰を減らす．この命令ス

```
%reg1030 = inc %reg1029

cmp %reg1030, 1
mov %reg1032, [%reg1031 + %reg1030]
```

(a) レジスタ割付け前

```
inc a
mov b, a
cmp a, 1
mov a, [hl + b]
```

(b) レジスタ割付け後

図 5　生存区間の終端

Fig. 5　End of a live range.

ケジューリングは基本ブロック単位で実施し，基本ブロックをまたがる命令の移動は行わない．基本ブロック内での命令スケジューリングの実施要領を次に示す．

- 基本ブロックの出口側にある命令から順に走査する．
- 走査対象の命令 Iu が汎用レジスタを介して入力値を受け取るオペランドを持つならば，当該オペランドの内容を定義する命令 Id と命令 Iu の距離を縮めることを試みる．具体的には次の要領で命令を移動することを試みる．
  - 命令 Id を命令 Iu の直前に移動する．
  - 命令 Iu を命令 Id の直後に移動する（命令 Iu が汎用レジスタを定義しない場合のみ）．

  命令 Id が命令 Iu と同じ基本ブロックに存在しない場合や，命令 Id と命令 Iu の間に移動を阻む要因（データ依存など）がある場合には移動を諦める．

  命令 Iu が汎用レジスタを介して入力値を受け取るオペランドを複数持つ場合には，アキュムレータを割り付けられるものから順にこの試みを行い，1 度でも試みに成功した，もしくはすべてのオペランドに対する試みに失敗した時点で命令 Iu の走査を終了し，次の命令の走査に移る．

3.5　レジスタ割付け

CC-RL のレジスタ割付けは，最終的には，生存区間の干渉グラフを構築し，構築したグラフを彩色することで割付けを行うが，そこに至るまでにいくつかの処理を行う．それらの処理を含めた，CC-RL のレジスタ割付けの処理の全体を次に示し，その個々について順次詳述する．

- 逆 SSA 変換
- 冗長な複写命令の削除
- 干渉グラフの構築
- グラフ彩色
- スタックフレームへのデータの配置
- 割付け結果の MachineInstr への反映

3.5.1　逆 SSA 変換

命令選択が終わってからレジスタ割付けまでの間, CC-RL の中間表現は static single assignment（SSA）形式 [11], [12] になっているが，SSA 形式はレジスタ割付けには適さない．たとえば図 5 (a) はレジスタ割付け前の中間表現で，その最初の命令は `%reg1030 = inc %reg1029` となっており，ここで命令 `inc` が入力値を受け取るレジスタと出力値を格納するレジスタを別々のレジスタにしているのは，中間表現を SSA 形式にするためだが，レジスタ割付けでは命令 `inc` が入力値を受け取るレジスタと出力値を格納するレジスタを同一にする必要があるので，これらのレジスタを別々にしておくことはレジスタ割付けにとって都合がよくない．そこで次の処理を行って，中間表現をレジスタ割付けに適した形式に変換する．

```
%reg1030 = inc %reg1029
```

(a) 割付け前

```
mov %reg1030, %reg1029
%reg1030 = inc %reg1030
```

(b) 割付け後

図 6　2 番地形式のオペランドに対する同一仮想レジスタの割付け

Fig. 6　Single virtual register allocation to two address operands.

2 番地形式のオペランドへの同一仮想レジスタの割付け

命令 `inc` のオペランドのように，入力値と出力値に同じレジスタを割り付ける必要があるオペランドに，同一の仮想レジスタを割り付ける．具体的には，次の処理を行う．

(1) 入力値と出力値に同じレジスタを保持する必要があるオペランドを持つ命令の直前に，入力オペランドのレジスタから出力オペランドのレジスタへの複写命令を挿入する．

(2) 入力オペランドのレジスタを出力オペランドのレジスタに書き換える．

これらの処理を**図 6** (a) の命令 `inc` に適用すると，その結果は図 6 (b) の命令列になる．

ϕ 関数の複写命令による置換　ϕ 関数は SSA に固有な中間表現であり，複数の制御フローが合流する地点に現れ，当該箇所で 1 つの変数に対する複数の定義が合流することを表す．ϕ 関数の複写命令による置換では，ϕ 関数を削除し，その代わりとなる複写命令を制御フローの合流直前の位置に挿入する．

ϕ 関数の複写命令による置換の適用事例を**図 7** に示す．図 7 (a) では基本ブロック `BB0`，`BB1` からの制御フローが `BB2` で合流しているが，合流地点の `BB2` の冒頭にある ϕ 関数を複写命令で置換した結果を図 7 (b) に示す．図 7 (b) では ϕ 関数がなくなり，代わりに ϕ 関数が入力として受け取っていた `%reg1037`，`%reg1038` から，置換用に新規生成した仮想レジスタ `%reg1040` への複写がそれぞれ `BB1`/`BB2` の末尾に，`%reg1040` から ϕ 関数の出力値の代入先である `%reg1039` への複写

```
BB0:
    ...

    br BB2
BB1:
    ...

BB2:
    %reg1039 = φ %reg1037, %reg1038
```

(a) 置換前

```
BB0:
    ...
mov %reg1040, %reg1037
    br BB2
BB1:
    ...
    mov %reg1040, %reg1028
BB2:
    mov %reg1039, %reg1040
```

(b) 置換後

図 7　ϕ 関数の複写命令による置換

Fig. 7　ϕ function replacement with copy instructions.

```
movw %reg1041, [%reg1040]
movw [%reg1042], %reg1043
movw ax, %reg1041
ret
```

(a) 削除前

```
movw ax, [%reg1040]
movw [%reg1042], %reg1043


ret
```

(b) 削除後

図 8　複写命令の削除がレジスタ割付けにもたらす影響

Fig. 8　Live range expansion by eliminating a copy instruction.

が BB2 の冒頭に，それぞれ挿入されている．

3.5.2　冗長な複写命令の削除

逆 SSA 変換で挿入する複写命令の中には冗長なものもあるので，冗長なものを探して削除する処理を行う．たとえば図 6 (b) に挿入した複写命令は，命令 `inc` より先に `%reg1029` を使用する命令がないなら冗長であり，冗長である場合，複写命令が定義するレジスタ `%reg1030` の使用箇所を `%reg1029` に書き換えれば，当該複写命令を削除可能になる．

逆 SSA 変換と冗長な複写命令の削除は，ともに LLVM に備わっている機能だが，CC-RL では LLVM が提供する冗長な複写命令の削除の機能のうち，複写元もしくは複写先が物理レジスタの複写命令を削除する機能は利用しない．その理由は，複写元もしくは複写先が物理レジスタの複写命令を削除すると，物理レジスタの生存区間が延びてレジスタを割り付けられなくなることがあるからである．

たとえば**図 8** (a) の中にある複写命令 `movw ax, %reg1041` を削除すると，削除後の中間表現（図 8 (b)）の中にあるストア命令 `movw [%reg1042], %reg1043` にレジスタを割り付けることができなくなる．なぜなら，削除すると，物理レジスタ `ax` の生存区間が延び，ストア命令 `movw [%reg1042], %reg1043` をまたいでしまい，ストア命令のオペランドにレジスタ `ax` を使えなくなってしまうが，一方で RL78 マイコンの命令セットがもたらす制約により `%reg1043` にはレジスタ `ax` しか割り付けることができず，結果として `%reg1043` に割付け可能なレジスタがなくなってしまうからである．

この問題を回避するために，CC-RL では，LLVM が提供する，複写元もしくは複写先が物理レジスタの複写命令を削除する機能を利用しないが，冗長な複写命令を放置するとコンパイラの生成コードの品質は悪くなる．そこで CC-RL では，3.5.6 項で述べるように，レジスタ割付けの最後の段階で再度，複写命令の削除を試みる．

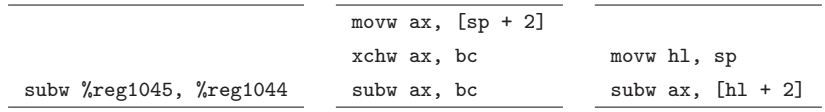
3.5.3　干渉グラフの構築

CC-RL のレジスタ割付けでは干渉グラフのノードを仮想レジスタの生存区間全体にしない．その理由は，RL78 マイコンの命令のオペランドに使えるレジスタが限られていることから，生存区間の全体に 1 つのレジスタを割り付けようとしても割り付けられないケースが多くあるからである．たとえば図 5 (a) の仮想レジスタ `%reg1030` について考えると，仮想レジスタ `%reg1030` を参照する箇所は 3 カ所あるが，うち 1 カ所の命令 `inc` には任意のレジスタを割り付けられるものの，残りの命令 `cmp` にはレジスタ `a` のみ，命令 `mov` の当該オペランドにはレジスタ `b` もしくは `c` しか割り付けられないので，レジスタ `reg1030` の生存区間全体に割り付けられるレジスタは 1 つもない．この問題に対応するため，CC-RL のレジスタ割付けでは，仮想レジスタの生存区間を，仮想レジスタの定義および使用地点の前後で分割し，分割後の生存区間の断片をノードとして干渉グラフを構築する．分割後の生存区間は次の 2 種類に分類できるが，前者を spot，後者を span と呼ぶことにする．

- 仮想レジスタの定義もしくは使用地点のみを含む生存区間
- spot と spot に挟まれた区間

3.5.4　グラフ彩色

干渉グラフを構築したらノードを彩色して，どのノードにどのレジスタを割り付けるのかを定める．彩色にあたり，spot には必ず彩色する．これは spot が命令のオペランドに対応していることから，彩色しないとオペランドにどのレジスタにすべきかが定まらないからである．spot に



(a) レジスタ割付前　(b) 融合無　(c) 融合有

図 9　リストアの融合

Fig. 9　Coalescing a restore.

彩色できる色の候補は命令がオペランドとして利用できるレジスタに応じて定まるが，次に示す例外もある．

レジスタ間の複写命令 mov/movw のオペランド　レジスタ間の複写命令 mov/movw のオペランドは一方をアキュムレータレジスタにする必要があるが，レジスタ割付けの際には，これに加えて複写元/複写先のオペランドに同一のレジスタを割り付けることを許す．許す理由は，複写元と複写先が同一の複写命令は冗長なものと見なして削除できるからである．また 16 bit の複写命令 movw に関しては複写元/複写先のオペランドに任意のレジスタを割り付けることを許す．許す理由は，命令 push と pop が任意の 16 bit レジスタをオペランドにとれることから，命令 push と pop を使って任意の 16 bit レジスタ間の複写に相当する動作を実現できるからである．

リストアの融合　隣接する全 span が spill 対象になった spot について，当該 spot に対応するオペランドの参照先をレジスタではなくメモリに変更できる場合には，オペランドの参照先をメモリ（スピルスロット）に変更し，spot にメモリを参照するオペランドのベースレジスタ hl を割り付けることを許す．

リストアの融合の例を**図 9** に示す．図 9 (a) にあるレジスタ割付け前の中間表現中の命令 subw %reg1045, %reg1044 の前後でレジスタ %reg1044 がスピルされているとする．その場合，命令 subw の前にレジスタ %reg1044 をリストアするコードを挿入することはできるが，その結果として生じるコードは図 9 (b) のとおりで効率的でない．図 9 (b) のコードではリストアのコード movw ax, [sp + 2] のリストア先が，reg1044 に割り付けたレジスタ bc ではなく ax になっているが，これは RL78 マイコンの命令セットではリストア先をアキュムレータレジスタにしかできないからで，このため図 9 (b) のコードではレジスタ ax を経由してリストアしており，レジスタ ax を経由することが効率の悪さの原因となっている．この効率の問題を改善する手段の 1 つがリストアの融合である．融合後のコード（図 9 (c)）は命令 subw に到達するアキュムレータレジスタの内容を破壊しないという意味でも効率的ではあるが，参照先をメモリ（スピルスロット）に変更したオペランドで利用できるベースレジスタが sp でなく hl であることから次の問題が生じる．

- レジスタ hl を他の用途に使っている箇所ではリストアを融合できない．
- リストアを融合した箇所より前に，レジスタ sp の内容をレジスタ hl に複写する命令 movw hl, sp を挿入しなければならない．

ここで命令 movw hl, sp は必ずしもリストアを融合した箇所すべての直前に挿入する必要はなく，複数のリストアの融合先で共有することができる．必要最小限の命令 movw hl, sp の挿入先は，次の箇所をソース/シンクとする最小カット問題を解けば求められる．

ソース　リストアの融合以外の目的でレジスタ hl を割り付けた生存区間の終点および，レジスタ sp の値を変更する箇所と，関数の入口

シンク　リストアを融合した箇所

最小カット問題の解は必ずしも 1 つでないが，解が複数ある場合にはレジスタ hl にレジスタ sp の内容のコピーを保持する区間の長い解を選ぶ．その理由は，レジスタ hl にレジスタ sp の内容のコピーを保持している区間の中では 3.10 節で述べる最適化を適用できるからである．

3.5.5　スタックフレームへのデータの配置

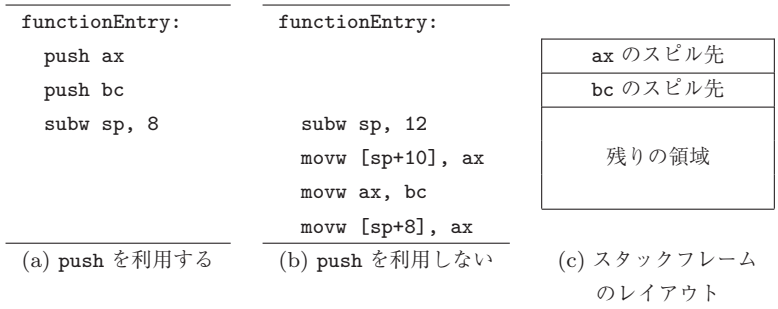
彩色が終わり，スピル対象の生存区間がどれか明らかになったら，スピルスロットなどスタックフレームに配置するデータ一式を，スタックフレーム中のどこに配置するのかを定める．配置にあたっては次の点に配慮する．

(1) スタックフレームのサイズを小さくする．
(2) スタックフレームを参照する命令のできるだけ多くが [sp+0/1] を参照するようにして，3.10 節で述べる最適化の適用先を増やす．
(3) レジスタ経由で受け取る仮引数をスピルする場合，スピル先をスタックフレームの端に寄せる．

ここで (1)，(2) を実現するためには生存区間が重複しないデータをスタックフレーム内の同一の場所に配置する必要がある．生存区間が重複しているか否かの判定は，レジスタ割付けの過程で利用する生存区間の情報があれば容易に実現できる．

(3) で仮引数のスピル先をスタックフレームの端に寄せる理由は，仮引数のスピルは，関数の入口で，スタックフレームを確保する処理をかねて命令 push で実施できるが，仮引数のスピル先をスタックフレームの端に寄せてあれば，スタックフレームの確保に必要な命令（レジスタ sp の減算）の数を最小にできるからである．

仮引数のスピルを命令 push で実現するコードの例を



(a) push を利用する　(b) push を利用しない　(c) スタックフレームのレイアウト

図 10　スタックフレームの確保
Fig. 10　Stack frame allocation.

図 10 (a) に示す．図 10 (a) のコードでは，まず，レジスタ ax，bc を介して受け取った仮引数を命令 push でスピルし，次に，命令 subw によって，スタックフレームの仮引数のスピルスロット以外の領域を確保している．ここで仮引数のスピルスロット以外の領域を命令 subw 1 つで確保できるのは，図 10 (c) に示すように，仮引数のスピルスロットをスタックフレームの一端に寄せて，それ以外の領域を連続した領域にしているからである．図 10 (a) のコードは，命令 push を利用しない図 10 (b) のコードに比べ次の点で優れている．

- 命令数が少ない．
- コードサイズが小さい（図 10 (a)/(b) のコードサイズはそれぞれ 4/7 バイト）．
- レジスタ ax の内容を破壊しない．

3.5.6　割付け結果の MachineInstr への反映

彩色が終わり，さらにスタックフレームに配置するデータのレイアウトが決まって，スピルにどんな命令を使うかも確定したら，その結果を MachineInstr に反映させる．具体的には，オペランドが参照する仮想レジスタを物理レジスタに書き換え，また，スピルやリストアのコードを挿入する．この際，レジスタ間の複写命令のうち，複写元と複写先に同じ物理レジスタを割り付けたものを冗長と見なして削除する処理もあわせて実施する．

3.6　命令選択（レジスタ割付け後）

レジスタ割付けが終わったら，レジスタ割付けで確定した情報を使って命令選択をやりなおす．具体的には次の処理を行う．

(1) shlw ax，1 から addw ax，ax への書き換え
(2) cmp a，1/-1 から dec/inc a への書き換え
(3) cmpw ax，0 から or a，x への書き換え

これら 3 つの最適化のうち (1)，(2) については，それぞれ 3.2 節，3.3 節で述べたので，ここでは (3) について述べる．

(3) はコードサイズの削減を目的とした最適化であり（命令 cmpw ax，0/or a，x のコードサイズはそれぞれ 3/2 byte），命令 cmpw ax，0 と命令 or a，x が PSW の Z フラグにおよぼす影響が同一であることを利用し，次の条件を満たす命令 cmpw ax，0 を命令 or a，x に書き換える．

- 命令 cmpw ax，0 の結果の利用者が参照する PSW 中のフラグが Z フラグのみである．
- 命令 cmpw ax，0 がレジスタ ax の内容の最後の利用者である．

ここで 2 つ目の条件は書き換え後の命令 or a，x がレジスタ ax の内容を破壊することから必要になる．(3) の最適化をレジスタ割付けの後まで実施しないのは，3.3 節で述べたように，命令がレジスタの内容の最後の利用者になるか否かはレジスタ割付けが終わるまで分からないからである．

3.7　スタックフレーム中のデータを参照する命令の具象化

命令選択（レジスタ割付け後）と同じく，レジスタ割付けの後処理として，3.5.5 項の処理で定めた，スタックフレームに配置するデータのレイアウトを，MachineInstr に反映する処理を行う．レジスタ割付けが終わって，どれだけのデータがスピルの対象になるかが確定するまで，どれだけのデータがスタックフレームに配置されるか分からないことから，LLVM では，スタックフレームを参照する命令を，レジスタ割付けが終わるまでは抽象的な命令で表現しておき，レジスタ割付けの後で具象化するが，その流れは CC-RL でも変わらない．具象化の対象になる命令は次の 3 種類であり，それぞれの具象化の例を**表 2** に示す．表 2 中の $\&autoVar$ はスタックフレーム中にあるデータのアドレスを表す抽象的なオペランドである．表 2 の具象化後のコードは，3.5.5 項の処理で $\&autoVar$ が sp+12 に定まった場合のものである．

(1) スタックフレーム中のデータのロード
(2) スタックフレーム中へのデータのストア
(3) スタックフレーム中のデータのアドレスの取得

3.8　分岐の畳込み/共通コードの下方集約

分岐の畳込みおよび共通コードの下方集約は LLVM に備わっている機種非依存な機能であり，CC-RL をはじめ，ルネサスのコンパイラプラットフォームをベースとするコ

表 2　スタックフレーム中のデータを参照する命令の具象化
Table 2　Stack frame allocation.

命令の種別	具象化前	具象化後
ロード	`movw ax, [&autoVar]`	`movw ax, [sp+12]`
ストア	`movw [&autoVar], ax`	`movw [sp+12], ax`
アドレスの取得	`movw ax, &autoVar+12`	`movw ax, sp` `addw sp, 24`

図 11　分岐の畳込み
Fig. 11　Branch folding.

ンパイラ製品はその改訂版を利用している．

3.8.1　分岐の畳込み

分岐の畳込みは分岐命令の削除を目的とする最適化であり，たとえば分岐命令の分岐先が無条件分岐命令である場合に，分岐命令の分岐先を無条件分岐命令の分岐先に変更し，これによって無条件分岐命令に到達する制御フローがなくなったら，無条件分岐命令を削除する，といった最適化を行う．たとえば**図 11** (a) の冒頭にある条件分岐命令 `.bz BB3` があり，その分岐先 BB3 には無条件分岐命令 `br BB7` がある．ここで分岐の畳込みは，冒頭の条件分岐命令 `.bz BB3` の分岐先を BB7 に変更し（図 11 (b)），これによって BB3 に到達する制御フローがなくなったら，基本ブロック BB3 を削除し，さらにそれによって BB3 の直上にある無条件分岐命令 `br BB4` が冗長になるのでそれも削除する（図 11 (c)）．

なお，ルネサスのコンパイラプラットフォームでは，命令の投機実行化をともなう分岐の畳込みもサポートしている．これは基本ブロックの末尾に無条件分岐があるとき，当該基本ブロック内にある無条件分岐以外の命令を制御フローの直上にある基本ブロックに移動したうえで，無条件分岐を畳み込む最適化である．この最適化では，たとえば**図 12** (a) の基本ブロック BB3 の冒頭にある命令 `movw ax, de` を，制御フローの直上にあたる基本ブロック BB1 に移動したうえで，BB3 の末尾にある無条件分岐 `br` を畳み込む．移動と畳み込みを行った後のコードを図 12 (b) に示す．ここで命令 `movw ax, de` を BB1 に移動可能なのは，命令 `movw ax, de` が定義するレジスタの内容を，BB1 のもう一方の分岐先 BB2 で使用しないからで，使用する場合には移動できない（条件付きの命令実行をサポートする目的機械ではその限りでないが，RL78 マイコンは条件付きの命令実行をサポートしない）．

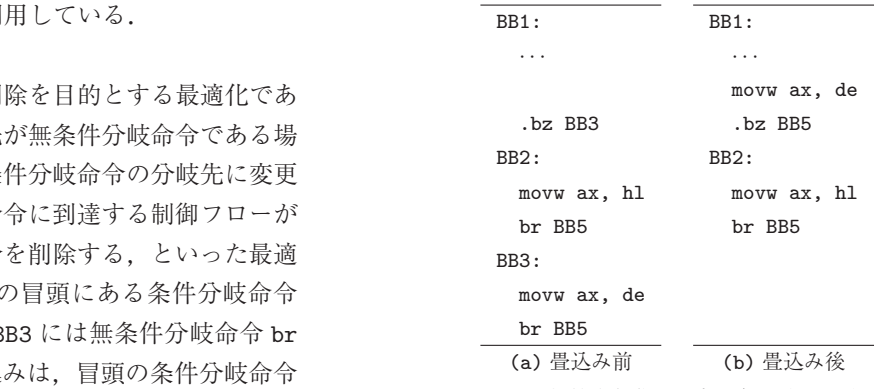


図 12　投機実行化をともなう分岐の畳込み
Fig. 12　Branch folding with instruction execution speculation.

命令の投機実行化をともなう分岐の畳込みの目的は，コードサイズの削減にあり，必ずしも実行速度の向上にはない．実行速度はむしろ劣化することもあるが，これは移動した命令が投機実行の対象になるためである．

3.8.2　共通コードの下方集約

共通コードの下方集約は，コードサイズの削減を目的とした最適化であり，次の基本ブロック群の末尾に等価な命令列があるか調べ，あるなら集約する．

- 制御フローの合流点の直上に位置する基本ブロック群
- 関数の出口にあたる基本ブロック群

たとえば**図 13** (a) では基本ブロック BB1 からの制御フローと基本ブロック BB2 からの制御フローが基本ブロック BB3 で合流しているが，ここで BB1 の末尾の 4 命令と BB2 の末尾の 3 命令は実質的に等価である．共通コードの下方

```
BB1:                BB1:
  ...                 ...
  movw ax, bc         br BB5
  cmpw ax, de
  bnz BB3
  br BB4
BB2:                BB2:
  ...                 ...
                      BB5:
  movw ax, bc         movw ax, bc
  cmpw ax, de         cmpw ax, de
  .bz BB4             .bz BB4
BB3:                BB3:
```

(a) 集約前　　(b) 集約後

図 13　共通コードの下方集約

Fig. 13　Tail merge.

```
                    mov x, 3        movw ax, 0x0203
func(2, 3);         mov a, 2
                    call func       call func
```

(a) ソースコード　(b) 融合前　(c) 融合後

図 14　8 bit の定数ロードの融合

Fig. 14　Coalescing 8 bit immediate loads.

集約ではこれらの等価な命令を集約して図 13 (b) の命令列に書き換える．

3.9　8 bit の定数ロードの融合

8 bit の定数ロードの融合では，8 bit の定数を 16 bit レジスタの上位 8 bit/下位 8 bit にそれぞれロードする命令があるとき，それらを融合して 16 bit の定数をロードする命令にする．たとえば**図 14** (a) のソースコードをコンパイルすると図 14 (b) の中間表現を得るが，本最適化では図 14 (b) の中間表現の中にある 2 つの 8 bit 定数ロード命令 `mov` を融合して 1 つの 16 bit 定数ロード命令 `movw` にする（図 14 (c)）．

本最適化の適用箇所を増やすための手段として，レジスタ割付けの際に，8 bit の定数ロード命令に割り付けるレジスタに配慮することは可能だが，我々のレジスタ割付けはそのような配慮をしない．その理由は，本最適化の適用先は主に図 14 (a) のような定数を実引数とする関数呼び出しで，引数の受渡しに使うレジスタに定数を格納するコードであり，受渡しに使うレジスタは ABI（Application Binary Interface）で定まっているため，レジスタ割付けで割り付けるレジスタに配慮する余地が少ないからである．

我々は本最適化の適用条件を次に示すとおりとした．

(1) 16 bit レジスタの上位/下位 8 bit に 8 bit の定数をロードする命令が同一の基本ブロック内にある．

(2) それらの命令のうち，基本ブロックの入口の近くにある方を Ip，もう一方を If とおくとき，If を Ip の直後に移動できる．厳密には，命令 If と Ip の間に命令 Ip が定義するレジスタを定義/使用する命令がない．

(3) 最適化のもたらす効果がコンパイラユーザの要望に反しない．

これらの条件を命令 Ip，If が満たすとき，本最適化は命令 Ip を書き換えて 16 bit の定数ロード命令とし，命令 If を削除する．

ここで条件 (3) は，最適化のもたらす効果が，実行サイクルの改善やコードサイズの削減といったコンパイラユーザの要望に反しないか否かを検証するものである．本最適化は実行サイクル数を改善するが，コードサイズを削減するとは限らないので，コンパイラユーザの要望がコードサイズの削減である場合，次のケースにあてはまるなら最適化を適用しない．

(1) ロードする 16 bit 定数が `0x0100` か `0x0101` であり，なおかつ，ロード先の 16 bit レジスタが `ax` か `bc` である．

(2) 他の命令と融合するとコードサイズが減る．

ケース (1) で最適化を適用しない理由は，8 bit の定数 0/1 に限っては，それらをロードするための，命令長の短い（1 byte）命令 `clrb`/`oneb` があり，それらを使ってロードする方が，汎用の 16 bit 定数ロード命令（3 byte）を使ってロードするよりコードサイズを小さくできるからである．ケース (1) の適用条件に，ロード先の 16 bit レジスタが `ax` か `bc` であること，とあるのは，命令 `clrb`/`oneb` のオペランドになりうるレジスタが `a`/`x`/`b`/`c` のみだからである．

ケース (2) は，ある 8 bit 定数ロードと融合できる 8 bit 定数ロード命令が複数ある場合に，コードサイズを最終的に最も小さくする命令と融合するためのものである．たとえば融合前の中間表現が**図 15** (a) である場合について考える．図 15 (a) の命令 `mov x, 3` は，命令 `mov a, 1` と `mov a, 4` のどちらとも融合しうるが，後者と融合する方が最終的にコードサイズを小さくできる．その理由は，後者と融合して前者を残しておけば，3.12 節で述べる最適化で，前者はより短い命令 `oneb a` に変わるが（図 15 (b)），前者と融合して後者を残すのではそうならないからである（図 15 (c)）．

なお，図 15 (a) に示した，本最適化の適用前の中間表現で，8 bit の定数 1 をロードする命令が `oneb a` ではなく `mov a, 1` になっている理由は，命令選択の際に，定数ロードの命令として，ロード先のレジスタの選択肢が多い命令を選ぶためである（命令 `mov` reg, imm は 8 bit の汎用レジスタならどれでもオペランドにとれる）．

3.10　レジスタ sp へのアクセスの最適化

レジスタ `sp` へのアクセスの最適化では，次の命令を最適化する．それぞれの命令に適用する最適化について順次詳述する．

(1) `sp` から汎用レジスタへの複写命令

(2) `[sp+0/1]` から/へのロード/ストア命令

```
mov a, 1
mov [sp+0x00], a
mov x, 3
mov a, 4
```
(a) 融合前

```
oneb a
mov [sp+0x00], a
movw ax, 0x0304
```
(b) mov a, 4 との融合

```
movw ax, 0x0103
mov [sp+0x00], a

mov a, 4
```
(c) mov a, 1 との融合

図 15　融合先が複数ある場合

Fig. 15　Multiple coalesce candidates case.

```
func(&autoVar1,
     &autoVar2);
```
(a) ソースコード

```
movw ax, &autoVar2
movw bc, ax
movw ax, &autoVar1

call func
```
(b) 具象化前

```
movw ax, sp
movw bc, ax
movw ax, sp ; 冗長
addw ax, 12
call func
```
(c) 具象化後

図 16　スタックフレーム中のデータのアドレスを取得する処理の具象化がもたらす冗長な複写命令

Fig. 16　A redundant copy-from-stack-pointer inserted in concretizing address calculation of data in the stack frame.

3.10.1　sp から汎用レジスタへの複写命令の最適化

sp から汎用レジスタへの複写命令に適用する最適化には，次の 2 種類がある．それぞれについて順次詳述する．

(1) 冗長な複写命令の削除

(2) 汎用レジスタ間の複写への書き換え

3.10.1.1　冗長な複写命令の削除

冗長な複写命令の削除では，3.7 節で述べた，スタックフレーム中のデータのアドレスを取得する処理の具象化の結果として現れる命令列中の複写命令 `movw ax, sp` のうち，冗長なものを削除する．

たとえば**図 16** (a) のソースコードをコンパイルした結果の中間表現は，3.7 節で述べた処理を適用する前の段階で図 16 (b) に示すとおりになるが，ここで自動変数 `autoVar1`，`autoVar2` の格納先がそれぞれ `sp+12`/`sp` に定まった場合，3.7 節で述べた処理によって中間表現は図 16 (c) に示すとおりとなり，ここで図 16 (c) にある 2 つの複写命令 `movw ax, sp` のうち，一方は冗長である．本最適化ではこのような冗長な複写命令をみつけて削除するほか，命令の移動や書き換えをともなう削除も行う．

命令の移動をともなう削除では，複写命令 `movw ax, sp` が複数あるとき，そのいずれかが定義する値を使用する命令を移動して別の複写命令が定義する値を使用させ，これによって移動前に使用していた値を定義する `movw ax, sp` を冗長にして削除する．たとえば**図 17** (a) の命令列には 2 つの複写命令 `movw ax, sp` があるが，その一方の定義する値を使用する命令 `movw bc, ax` を移動し，もう一方が定義する値を使用するように改めれば，命令列は図 17 (b) に示すとおりとなり，移動前に利用していた値を定義する複写命令 `movw ax, sp` が冗長になるのでこれを削除する．

命令の書き換えをともなう削除では，複写命令 `movw ax, sp` が定義する値の利用者が唯一アキュムレータレジスタ

```
movw ax, sp

addw ax, 12
movw de, ax
movw ax, sp
movw bc, ax ; 移動前
movw ax, 1
call function
```
(a) 移動前

```
movw ax, sp
movw bc, ax ; 移動後
addw ax, 12
movw de, ax
movw ax, sp ; 冗長

movw ax, 1
call function
```
(b) 移動後

図 17　命令の移動による複写命令 `movw ax, sp` の冗長化

Fig. 17　Copy-from-sp elimination after instruction scheduling.

```
movw ax, sp
addw ax, offset2
movw bc, ax
movw ax, sp
addw ax, offset1
call function
```
(a) 最適化前

```
movw ax, sp
addw ax, offset2
movw bc, ax

addw ax, offset1 − offset2
call function
```
(b) 最適化後

図 18　命令の書き換えをともなう複写命令 `movw ax, sp` の削除

Fig. 18　Elimination of a `movw ax, sp` with rewriting an instruction.

ax に定数 $offset1$ を加算する命令であって，なおかつ，当該複写命令に到達した時点でのアキュムレータレジスタの内容が `sp +` $offset2$ である場合に，加数を $offset1$ から $offset1 - offset2$ に書き換えたうえで複写命令 `movw ax, sp` を削除する．たとえば**図 18** (a) の命令列では加算命令 `addw ax,` $offset1$ がその直前にある複写命令 `movw ax, sp` の結果の唯一の使用者で，なおかつ当該複写命令に到達するアキュムレータレジスタの内容が `sp +` $offset2$ であるので，本最適化では加数を $offset1 - offset2$ に書き換えたうえで複写命令を削除する．削除後の命令列は図 18 (b) に示すとおりとなる．

ロード		ストア	
(a) 代替前	(b) 代替後	(c) 代替前	(d) 代替後
movw ax, [sp+0]	pop de	movw ax, de	pop ax
movw de, ax	push de	movw [sp+0], ax	push de

図 19　[sp+0] から/へのロード/ストアの命令 pop と push による代替
Fig. 19　Substitute for a load from or a store to [sp+0] by a pop and a push.

3.10.1.2　汎用レジスタ間の複写への書き換え

RL78 マイコンでは sp から汎用レジスタ *rp* への複写命令 movw *rp*，sp のコードサイズが比較的大きく，複写先の汎用レジスタがアキュムレータレジスタ ax の場合には 2 byte，アキュムレータレジスタ以外の汎用レジスタの場合には 3 byte になるので，これをより小さな命令に書き換えることでコードサイズを削減する．具体的には，sp から汎用レジスタへの複写命令のある場所で，3.5.4 項で述べた理由により，汎用レジスタ hl に sp と同一の値が入っているならば，当該複写命令を汎用レジスタ hl からの複写命令に書き換える．汎用レジスタ hl からの複写に必要なコードサイズは，複写先がアキュムレータレジスタである/でない場合，それぞれ 1/2 byte で，いずれの場合も sp からの複写命令に比べて 1 byte 小さい．なお，汎用レジスタ hl からアキュムレータレジスタ以外への複写は，3.5.4 項で述べたように，最終的には push と pop（それぞれ 1 byte）の 2 命令で実現することになるので，実行サイクル数の点で 1 つの命令 movw *rp*，sp による複写に劣る．したがって，複写先がアキュムレータレジスタでない場合の書き換えは，最適化の目的が実行速度の向上にある場合には実施しない．

3.10.2　[sp+0/1] から/へのロード/ストア命令の最適化

[sp+0/1] から/へのロード/ストア命令を対象とした最適化には，次の 2 種類がある．

(1) 命令 pop と push による代替
(2) オペランド [sp+0] の [hl] による代替

本項では，まず，これらの最適化について順次詳述し，次に，これらの最適化の適用順序について述べる．

3.10.2.1　命令 pop と push による代替

[sp+0/1] から/へのロード/ストアの命令 pop と push による実現は，[sp+0] から/への 16 bit データのロード/ストアおよび，[sp+0/1] からの 8 bit データのロードを対象とした最適化である．本最適化についてはまず 16 bit データのロード/ストアについて述べ，次に 8 bit データのロードについて述べる．

16 bit データのアドレス [sp+0] から/へのロード/ストアは命令 pop と push を使って実現することも可能だが，命令 pop/push が任意のレジスタをオペランドにとれることを考慮すると，命令 pop と push を使う方がコードサイズを削減できる場合がある．

たとえば**図 19** (a) のコードは [sp+0] にあるデータをアキュムレータレジスタ ax 経由で汎用レジスタ de にロードするが，汎用レジスタ de へのロードであれば図 19 (b) のコードでも実現でき，なおかつ，図 19 (b) のコードの方がコードサイズが小さい（図 19 (a)/(b) のコードサイズはそれぞれ 3/2 バイト）．同様に，図 19 (c) のコードは汎用レジスタ de にあるデータをアキュムレータレジスタ ax 経由で [sp+0] にストアするが，汎用レジスタ de からのストアであれば図 19 (d) のコードでも実現でき，なおかつ，図 19 (d) のコードの方がコードサイズが小さい．

そこで，16 bit データのロード/ストアの pop と push による代替では，図 19 (a)/(c) のようにアキュムレータレジスタ経由でロード/ストアを行うコードを，図 19 (b)/(d) のように pop と push でロード/ストアを行うコードに書き換えることでコードサイズを削減する．この書き換えは，書き換え前のコードが次のいずれかに該当する場合には実施できないので，書き換えの可否を事前に確認する．

(1) 書き換え前のロード/ストアのコードがアキュムレータレジスタに代入した値を当該ロード/ストアとは異なる用途にも利用しうる場合：書き換え後のコードはアキュムレータレジスタを書き換えないので，書き換え前のコードの代替にならない．
(2) ロード元/ストア先が非同期な参照の対象になりうる場合：書き換え後の命令 pop はロード元/ストア先の領域を一時的にせよ解放するが，解放した領域は割込みなどによる上書きの対象になるので，当該領域が非同期な参照の対象になりうるのに書き換えを行うと，非同期な参照の結果を保証できなくなる．
(3) 書き換え前のロード/ストアがソースコード上の volatile なメモリ参照に対応する場合：書き換え後のコードはロード/ストアのどちらかだけのはずの処理を，ロード/ストア双方の処理で実現するので，コンパイラユーザがメモリアクセスの最適化を望まない（キーワード volatile で修飾した）ロード/ストアの実現として妥当でない．

8 bit データのロードについても，16 bit データと同様に図 19 (b) のコードに書き換えることでコードサイズを削減できる場合はある．ただ RL78 マイコンの命令 push/pop には 16 bit データ向けのものしかないことから，書き換え可能な 8 bit データのロードは (1)，(2)，(3) に加え，次のいずれにも該当しないものに限られる．

(4) ロード先の 8 bit レジスタとペアになって 16 bit レジ

```
movw ax, [sp+0]
movw de, ax       ; 直前で書込み
movw ax, [de]     ; ストールする
```

(a) 代替前

```
pop de            ; 2 命令前で書込み
push de
movw ax, [de]     ; ストールしない
```

(b) 代替後

図 20　命令 pop と push によるロードの代替がもたらすパイプラインストールの解消
Fig. 20　Pipeline stall elimination by substitution to a pop and a push.

スタを構成する，もう一方の 8 bit レジスタに，ロードを行う場所で，有意な値が入っている場合：pop がもう一方の 8 bit レジスタの内容を破壊するので書き換えは不適切である．

(5) ロード先のレジスタとロード元のアドレスがマッチしない場合：レジスタをロード先にできるか否かは，ロード元のアドレスに応じて定まり，ロード元が [sp+0/1] の場合，ロード先にできる 8 bit レジスタは 16 bit レジスタの下/上 8 bit にあたるもののみになる．

(6) 書き換え後の pop/push 命令が，ロード元の 8 bit の領域と一緒にアクセスする，ロード元と隣接する 8 bit の領域が非同期もしくは volatile な参照の対象になりうる場合：(2)，(3) と同じ理由により書き換えは不適切である．

3.10.2.2　オペランド [sp+0] の [hl] による代替

オペランド [sp+0] の [hl] による代替は，3.10.1 項で述べた最適化と同様に，レジスタ hl と sp に同じ値が入っている区間内にある命令に適用可能な最適化であり，命令 mov/movw のオペランドが [sp+0] であるとき，それを [hl] に書き換えることで，コードサイズを 2 byte から 1 byte に削減する．この削減ができる理由は，RL78 マイコンの命令セットでは，sp をベースとするロード/ストア命令にはオフセット（[sp+$offset$] の $offset$）を指定できるものしかないのに対し，汎用レジスタをベースとするロード/ストア命令には，オフセットを指定できない代わりにコードサイズの小さなものがあるからである．

3.10.2.3　[sp+0/1] から/へのロード/ストア命令向け最適化の適用順序

命令 pop と push による代替とオペランド [sp+0] の [hl] による代替は，それぞれを単独でみれば，コードサイズを同じく 1 バイト減らす最適化である．しかしながらこれらの最適化には，実行速度に与える影響や，他の最適化に与える影響に違いがあるため，最適化の適用順序を定めるにあたっては，それらの影響を考慮する必要がある．

まず，実行速度に与える影響について述べる．これらの最適化はともに命令数に変化を与えるものではないので，直接的に実行速度を改善することはない．しかしながら，命令 pop と push によるロードの代替ではスタックからレジスタにデータを読み込むタイミングが 1 命令分速くなり，結果としてパイプラインのストールを解消して実行速度を改善することがある．

たとえば**図 20** の命令列について考えると，図 20 (a) の代替前の命令列では 3 命令目の命令 movw ax, [de] の直前の命令でレジスタ de に書き込んでいるが，RL78 マイコンでは命令 movw ax, [de] のようにレジスタの値をアドレス計算に使う命令の直前で，当該レジスタ（ここでは de）に書き込むと，パイプラインがストールして実行速度が劣化してしまう．これに対し図 20 (b) の代替後の命令列ではレジスタ de に書き込むタイミングが早くなり，その結果，パイプラインがストールしなくなっている．

パイプラインのストールの解消は命令 pop と push による代替のみが保有する利点であり，そのような利点はオペランド [sp+0] の [hl] による代替にはない．このため CC-RL では，基本的には命令 pop と push による代替を先に適用する．しかしながら命令 pop と push による代替には固有の欠点もある．それは，3.11 節で述べる，共通コードの関数化という最適化の妨げるということである．この欠点を回避するため，CC-RL では最適化の目的がコードサイズの削減にある場合に限っては命令 pop と push による代替を共通コードの関数化の後に適用する．

```
_func0:
  ...
  movw ax, [sp+4]
  movw [sp+8], ax
  movw ax, [sp+0]
  ...
_func1:
  ...
  movw ax, [sp+4]
  movw [sp+8], ax
  movw ax, [sp+0]
  ...
```

(a) 最適化前

```
_func0:
  ...
  call !_commonCode
  ...
_func1:
  ...
  call !_commonCode
  ...
_commonCode:
  movw ax, [sp+8]
  movw [sp+12], ax
  movw ax, [sp+4]
  ret
```

(b) 最適化後

図 21　共通コードの関数化
Fig. 21　Procedual abstraction.

3.11　共通コードの関数化

共通コードの関数化は，コードサイズの削減を目的とした最適化であり，同一の命令列を関数として切り出して集約する．たとえば**図 21** (a) のコードについて考える．

`movw hl, bc`	`push bc` `pop hl`	`movw ax, bc` `movw hl, ax`
(a) 具象化前	(b) 命令 `push` と `pop` による具象化	(c) 命令 `movw` による具象化

図 22　16 bit の非アキュムレータレジスタ間の複写
Fig. 22　Data transfer between 16 bit non-accumlator registers.

図 21 (a) のコードをみると，関数 `func0()` と `func1()` の中に同一の命令列があることが分かるが，共通コードの関数化では，このような命令列を切り出して，新たな関数を作り，切出し元には，切り出した命令列の代わりに，新たに作った関数を呼ぶ命令を挿入する．図 21 (a) のコードに共通コードの関数化を適用した結果は図 21 (b) に示すとおりであり，適用前後のコードを比較すると，適用後のコードには共通コードを納める関数 `commonCode()` が作成されており，また適用前に共通コードがあった箇所には関数 `commonCode()` を呼ぶ命令 `call !_commonCode` が挿入されていることが分かる．

図 21 (a)，(b) のコードを比較すると関数化の前後で共通コード内にある `sp` 相対参照のオフセットが 4 だけずれていることが分かるが，このずれは共通コードを命令 `call` で呼び出す場合に必要になる．すなわち，RL78 マイコンの命令 `call` は実行時に戻り番地をスタックに積み，その際，`sp` の値が変化するので，命令 `call` で呼び出す共通コードでは，その変化に対応するために `sp` 相対参照のオフセットをずらすといった対応が必要になる．この `sp` の変化に対応できないコード，たとえば命令 `pop` によるスタック参照などは共通化の対象にしない．

ただし，切出し前の共通コードが関数の終端に位置する場合，切り出した共通コードを末尾呼び出しでき，その場合，戻り番地をスタックに積まないので，`sp` の変化に対応できないコードも共通化の対象にする．

共通コードの関数化を，コンパイラで実施するのは必ずしも適切でない．その理由は，リンク後の方がより多くの共通コードを発見できる場合もあるからである．それにもかかわらず CC-RL が共通コードの関数化を実施する理由は次の 2 つである．

(1) リンク時最適化を利用できないコンパイラユーザへの配慮．リンク前のオブジェクトコードで品質を確保するユーザは，リンク時最適化による品質の変化を許容しないので，リンク時最適化を利用できない．
(2) 他の最適化との連携．たとえば 3.10.2.1 項で述べた最適化は，命令 `pop` の利用箇所を増やすため，共通コードの関数化を阻害するが，そういった最適化と，共通コードの関数化のどちらを優先するかといった調整は，コンパイラの方が容易に実現できる．3.10.2.3 項で述べたように，CC-RL では最適化の目的がコードサイズの削減にある場合には共通コードの関数化を先に適用する．

3.12　命令選択（分岐命令選択前）

分岐命令選択前の命令選択では次の最適化を実施する．これらの最適化を分岐命令選択より前に行う理由は，原則として分岐命令選択の前までに最適化を終えてどの命令を使うのかを確定しないと，分岐の距離が定まらず，距離に応じた分岐命令の選択ができないからである．これらの最適化を分岐命令選択の直前まで実施しない理由は，これらの最適化がいずれもシンプルな中間表現をより複雑なものに書き換えることから，先に実施してしまうと，その後に実施する最適化の実装を複雑にしてしまうからである．

16 bit の非アキュムレータレジスタ間の複写の具象化
16 bit の汎用レジスタ間の複写命令 `movw` *dst*，*src* のうち，オペランド *dst*，*src* がいずれもアキュムレータレジスタ `ax` でないものを，複写を実現する命令列 `push` *src* + `pop` *dst* に書き換える．書換え前後のコードの例を**図 22** (a)，(b) に示す．
なお，アキュムレータレジスタが空いている箇所では，アキュムレータレジスタ経由で複写する命令列 (図 22 (c)) に書き換えることもできるが，現状の RL78 マイコンにとって図 22 (b)，(c) の書き換え結果に優劣はなく，コードサイズも実行サイクル数も同一である (RL78 マイコンがレジスタにアクセスするのにかかる時間と，スタックの配置先である内蔵 RAM にアクセスするのにかかる時間は同一であり，レジスタ間転送命令の方が速いということはない)．このため CC-RL では，アキュムレータが空いている場所でも，16 bit の非アキュムレータレジスタ間の複写の実現に，アキュムレータレジスタ経由での複写を使うことはしない．

定数ロードの短縮　定数を汎用レジスタにロードする命令 `mov`/`movw` を，同一の定数をロードする，よりコードサイズの小さい命令（列）に書き換える．書き換えのパターン一式を**表 3** に示す．書き換えのパターンがロード先のレジスタに応じて変化するのは，書き換え後の命令がオペランドにとれるレジスタが制限されているからである．なお，書き換えのパターンによっては命令数が増えることもあるが，そのような書き換えは最適化の目標がコードサイズの削減にある場合のみ実施する．

3.13　命令スケジューリング（分岐命令選択前）

分岐命令選択前の命令選択が終了した時点で，CC-RL が生成する命令のうち，分岐以外の命令はすべて確定する．

表 3　定数ロードの短縮
Table 3　Shorter sized immediate-load instruction sequences.

ロードする値	bit 幅	ロード先	書き換え後の命令列	備考
0	8	a/x/b/c	clrb *r*	
1	8	a/x/b/c	oneb *r*	
-1	16	ax/bc	clrw *rp* + decw *rp*	
0	16	ax/bc	clrw *rp*	
0	16	de/hl	clrw ax + mov *rp*, ax	ax が空いている場合のみ実施可
1	16	ax/bc	onew *rp*	
1	16	de/hl	onew ax + mov *rp*, ax	ax が空いている場合のみ実施可
2	16	ax/bc	onew *rp* + incw *rp*	

```
incw bc
movw de, ax
movw ax, [de]
```
(a) スケジュール前

```
movw de, ax
incw bc
movw ax, [de]
```
(b) スケジュール後

図 23　命令スケジューリングによるパイプラインストールの回避
Fig. 23　Instruction scheduling to avoid pipeline stalls.

```
.bc L_far
br L_near
```
(a) 選択前

```
bnc L_near
br L_far
```
(b) 選択後

図 24　条件分岐+無条件分岐の選択
Fig. 24　Instruction selection for a conditonal and unconditional branch sequence.

この状態に至ったら，パイプラインのストールを回避するための命令スケジューリングを行う．3.10.2.3 項で述べたように，RL78 マイコンのパイプラインは，レジスタの値をアドレス計算に使う命令の直前で，当該レジスタに書き込むとストールする．このストールを回避するため，レジスタに書き込む命令と，当該レジスタの値をアドレス計算に使う命令が連続しないようにスケジューリングし，たとえば**図 23** (a) の命令列を図 23 (b) の命令列に並べ換える．

ここでストールを回避するための命令スケジューリングを，分岐命令選択前の命令選択より後に実施する理由は，分岐命令選択前の命令選択が終わらないとスケジュール対象の命令が確定しないからであり，分岐命令選択より前に実施できる理由は，分岐命令選択がスケジュール対象の命令を生成しないからである．

3.14　分岐命令選択

RL78 マイコンが提供する分岐命令には，条件付き分岐命令と無条件分岐命令があり，それぞれに分岐可能な距離の異なるいくつかのバリエーションがある．しかしながら命令選択の時点では，分岐の距離が不明であるため，条件付き/無条件分岐をどの命令で実現できるか判断できない．このため CC-RL では，次の要領で分岐命令を選択する．

(1) 命令選択の段階では，分岐を，任意のオフセットの分岐が可能な命令にする．条件付き分岐命令については任意のオフセットの分岐が可能な命令は存在しないが，仮にそのような命令があることにして，その命令（擬似命令）を選択しておく．
(2) MachineInstr レベルでの最適化が終わり，分岐の距離がコンパイラの分かる範囲では確定した時点で，距離に応じて分岐命令を選択しなおす．

ここでコンパイラの分かる範囲では，と記述した理由は，コンパイラには大きさの分からない命令があるためである．たとえばコンパイル対象のソースプログラムとは異なるファイルにある関数を呼び出す命令の大きさは，コンパイラには分からない．なぜなら当該関数までの距離は，コンパイルの段階では確定しないことからである．この距離が確定するのはリンクの段階であり，したがって距離に応じた選択はコンパイラではなくリンカが実施する方が合理的という考え方もある．それにもかかわらず CC-RL が距離に応じた選択を行う理由は，共通コードの関数化と同様に，コンパイラユーザによってはリンク時最適化を利用できないことに配慮しているからである．なおルネサスの最適化リンカは距離に応じて分岐命令を選択しなおす機能を提供しており，当該機能を利用すれば，コンパイラのみで選択するよりもコードサイズを小さくできる．

CC-RL の距離に応じた選択では，無条件分岐命令を分岐先までの距離に応じて適切に選択するほかにも，次の選択を行う．なお，選択を行うと分岐の命令長が小さくなり，それをまたぐ分岐をさらに小さくできる可能性が生じることから，選択を行った場合には，それをまたぐ分岐の再選択も行う．

3.14.1　条件分岐の選択

直下の 1 命令をまたぐ条件分岐は skip 命令にする．それ以外の条件分岐は，分岐先での距離が-128 から 127 までに収まるなら RL78 マイコンのネイティブな条件分岐命令 b*cnd* にし，収まらないなら skip 命令と無条件分岐命令を組み合わせた，条件分岐に相当する命令列にする．

3.14.2　条件分岐+無条件分岐の選択

条件分岐の直下に無条件分岐があるケースについては，それら 2 つの命令があわせて最適になるようにする．たとえば選択前の命令列が**図 24** (a) 示すもので，条件分岐の擬似命令.bc（ここで接頭辞．は擬似命令であることを表現

表 4　最適化が実行速度にもたらす効果
Table 4　Optimization effects on performance.

最適化名	削減率 (%)
死亡コードの削除	2.9
冗長な 0 との比較の削除	0.8
共通部分式削除	0.5
`cmp *, 1/-1` から `dec/inc *`への書き換え	0.0
命令スケジューリング（レジスタ割付け前）	5.7
命令選択（レジスタ割付け後）	0.0
分岐の畳込み/共通コードの下方集約	1.0
8 bit の定数ロードの融合	0.0
レジスタ `sp`（スタックポインタ）へのアクセスの最適化	0.7
命令選択（分岐命令選択前）	0.0
命令スケジューリング（分岐命令選択前）	0.3

するものとする）から分岐先 $\mathtt{L}_{far}$ までの距離は遠くて，条件分岐命令では分岐できないが，無条件分岐 `br` の分岐先 $\mathtt{L}_{near}$ は近傍にあり，そこまでなら条件分岐命令で分岐できたとする．そのような場合には分岐条件を反転し，さらに条件分岐と無条件分岐の分岐先を交換した図 24 (b) の命令列を選択する．

4. 最適化の効果

本論文で示した最適化の効果を評価した．評価の対象は実行速度およびコードサイズの削減率とした．それぞれの評価結果を順次示す．

4.1 実行速度向け最適化の評価

実行速度向け最適化の評価では，測定対象の最適化のみ適用しない場合に比べ，適用した場合にどれだけ実行を高速化できるかを調べた．評価対象のプログラムとしては組込みマイコンの性能測定を目的としたベンチマークである CoreMark [13] を用いた．CoreMark は，現実的な組込み機器向けアプリケーションが含みうる演算から，組込みマイコンやコンパイラの生成したコードの性能から大きな影響を受けるものを抜粋したベンチマークで，具体的には次の演算を構成要素とする．

- 行列演算
- リストの操作
- ステートマシンの操作
- 巡回冗長検査

CoreMark のコンパイルに際しては，コンパイラにオプション `-Ospeed` を指定し，実行の高速化を目的とする最適化を実施させた．測定の結果を**表 4** に示す．

表 4 において，8 bit の定数ロードの融合の効果がないことになっているいるが，その理由は，評価対象のプログラム中に適用対象のパターン（主に 8 bit の定数 2 つを実引数とする関数呼び出し）が現れなかったためである．またレジスタ割付け後と分岐命令選択前の命令選択も効果がないことになっているが，その理由は，これらの命令選択の目的が基本的にはコードサイズの削減にあるからである．コードサイズを削減する最適化は，3.14.1 項で述べた分岐命令選択の結果を改善することがあり，結果として，実行も高速化しうるが，CoreMark の実行速度には影響しなかった．

なお，表 4 には，共通コードの関数化を記載していないが，その理由は，3.1 節で述べたように，オプション `-Ospeed` の指定時には当該最適化を適用しないためである．

4.2 コードサイズ向け最適化の評価

コードサイズ向け最適化の評価では，測定対象の最適化のみ適用しない場合に比べ，適用した場合にどれだけコードサイズを削減できるかを調べた．評価対象のプログラムとしては，CoreMark ではなく，EEMBC [14] の**表 5** に示すプログラムを利用した．表 5 のプログラムはいずれも現実的な組込み機器向けアプリケーションであり，組込みマイコンの性能の影響を受ける部分だけを抜粋したベンチマークである CoreMark よりもコードサイズの評価に適している．表 5 のプログラムのコンパイルに際しては，コンパイラにオプション `-Osize` を指定し，コードサイズの削減を目的とする最適化を実施させた．ここで分岐命令選択前の命令スケジューリングはコードサイズに影響を与えないため測定の対象外とした．測定の結果を**表 6** に示す．

表 6 の結果では `cmp *, 1/-1` から `dec/inc *`への書き換えと，8 bit の定数ロードの融合の効果がないことになっているが，その理由は，前者については命令選択（レジスタ割付け後）に同様の最適化があるため，前者を適用しなくても命令選択（レジスタ割付け後）で最適化できるケースが多いためであり，後者については，評価対象のプログラム中には適用対象のパターンが現れなかったためである．

5. 関連研究

最適化技法については，これまでに数多くの提案がある．

表 5　コードサイズ向けベンチマークの内訳
Table 5　Benchmark items for code size.

プログラム	ベンチマークが含むの代表的な処理
AutoBench 1.1	車内ネットワークによる通信処理/エンジン制御処理
ConsumerBench 1.1	デジタルカメラによる画像の圧縮伸長処理
Networking 1.1	ネットワーク機器によるパケット通信処理
OABench 1.1	プリンタによる画像の回転処理
TeleBench 1.1	モデムによる通信処理

表 6　最適化がコードサイズにもたらす効果
Table 6　Optimization effects on code size.

最適化名	削減率 (%)
死亡コードの削除	0.1
冗長な 0 との比較の削除	0.2
共通部分式削除	-0.3
`cmp *, 1/-1` から `dec/inc *`への書き換え	0.0
命令スケジューリング（レジスタ割付け前）	1.9
命令選択（レジスタ割付け後）	0.6
分岐の畳込み/共通コードの下方集約	2.6
8 bit の定数ロードの融合	0.0
レジスタ `sp`（スタックポインタ）へのアクセスの最適化	1.0
共通コードの関数化	6.2
命令選択（分岐命令選択前）	3.0

それらの最適化技法の中で，RL78 マイコン向けコンパイラにとって有意なものがどれで，どの順に適用すべきかという問題への解は，たとえば gcc [15], [16] のような，RL78 マイコン向けのコード生成をサポートするオープンソースのコンパイラを解析して得ることもできる．しかしながら本論文のように，RL78 マイコン向けには，どんな最適化を，なぜ，どの順番で適用べきかを，コンパイラ製品の開発経験に基づいて詳述している文献はこれまでにない．

最適化のどれをどの順で適用すべきかという問題の解決方法に関しても，これまでに様々な提案がある．その 1 つは，コンパイラに備わっている最適化オプションのどれを on にしてどれを off にすべきかを検討するもので [17], [18], [19], [20], [21], [22]，そういった検討は，既存のコンパイラに適用できる，コンパイラ開発者でなくても利用できるという意味で有用である．しかしながら最適化オプションのどれを on にしてどれを off にするかの検討は，コンパイラがあらかじめ定めている最適化の実施順序を変更するものではないことから，実施順序を変更すればより良い結果が得られるか否かを検討する手段にはならない．実施順序を変更する研究もあるが [23], [24], [25], [26], [27], [28]，実用的なコンパイラの最適化の実施順序を変更するのは必ずしも容易でない．その理由は，最適化の実施順序は必ずしも自由ではないので，変更する前に，変更しても問題がおきないか検証する必要があるが，検証を行うためには，たとえばコンパイラのソースコードの解析といった必ずしも容易でない作業が必要になりうるからである [29]．実施順序の自動的なチューニングを可能にするためには，コンパイラの設計にあたって，個々の最適化の設計者に，最適化間の依存関係の明示を義務付けるなどして，自動チューニングに配慮する必要がある [30]．なお，CC-RL の設計は，現状では自動チューニングに配慮していないが，その理由は，本論文で示した `MachineInstr` レベルの最適化群については相互に依存するものが多いため，最適化の実施順序の自由度が小さく，自動チューニングに配慮した設計にしても得られるメリットが小さいからである．

6. 結論

ルネサスエレクトロニクスの RL78 マイコン向け C コンパイラ CC-RL に実装した最適化処理のうち，RL78 マイコン向けに特化したものについて，その個々の内容を示した．さらに，それらの適用順序と，その根拠（最適化間の依存関係など）を明らかにした．

謝辞　本論文で述べた最適化のうち，大域的死亡コード削除の実装に際しては株式会社日立ソリューションズの永井佑樹様，レジスタ `sp`（スタックポインタ）へのアクセスの最適化および共通コードの関数化の実装に際しては株式会社ティー・エイ・シーの島田友行様，株式会社日立産業制御ソリューションズの國島政雄様にご協力いただき，また貴重な助言をいただいた．謹んで感謝の意を表する．

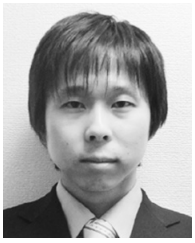
参考文献

[1] Renesas Electronics Corporation: RL78 Family (2015), available from ⟨http://www.renesas.com/products/mpumcu/rl78/⟩.

[2] Renesas Electronics Corporation: C Compiler Package for RL78 Family (2015), available from ⟨http://am.renesas.com/products/tools/coding_tools/c_compilers_assemblers/rl78_compiler/index.jsp⟩.
[3] Lattner, C.: The LLVM Compiler Infrastructure (2003), available from ⟨http://www.llvm.org⟩.
[4] Renesas Electronics Corporation: RX Family (2015), available from ⟨http://www.renesas.com/products/mpumcu/rx/⟩.
[5] Renesas Electronics Corporation: RH850 Family (Automotive only) (2015), available from ⟨http://www.renesas.com/products/mpumcu/rh850/⟩.
[6] Renesas Electronics Corporation: C/C++ Compiler Package for RX Family (2015), available from ⟨http://am.renesas.com/products/tools/coding_tools/c_compilers_assemblers/rx_compiler/index.jsp⟩.
[7] Renesas Electronics Corporation: C Compiler Package for RH850 Family (2015), available from ⟨http://am.renesas.com/products/tools/coding_tools/c_compilers_assemblers/rh850_compiler/index.jsp⟩.
[8] Cocke, J.: Global Common Subexpression Elimination, *Proc. Symposium on Compiler Optimization*, New York, NY, USA, pp.20–24, ACM (1970).
[9] Wulf, W.A., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O. and Geschke, C.M.: *The Design of an Optimizing Compiler*, Elsevier Science Inc., New York, NY, USA (1975).
[10] Standish, T.A., Neighbors, J., Kibler, D.F. and Harriman, D.C.: The Irvine program transformation catalogue A stock of ideas for improving programs using source-to-source transformations, Technical Report TR 76, University of California at Irvine (CA US) (1976).
[11] Shapiro, R.M. and Saint, H.: The representation of algorithms, Technical Report RADC-TR-69-313, Rome Air Development Center (1969).
[12] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Trans. Program. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
[13] The Embedded Microprocessor Benchmark Consortium: CoreMark An EEMBC Benchmark (2009), available from ⟨http://www.eembc.org/coremark/⟩.
[14] The Embedded Microprocessor Benchmark Consortium: Industry-Standard Benchmarks for Embedded Systems (1997), available from ⟨http://www.eembc.org⟩.
[15] The GCC team: GCC, the GNU Compiler Collection (1987), available from ⟨https://gcc.gnu.org⟩.
[16] KPIT Technologies Ltd.: KPIT GNU Tools & Support (2011), available from ⟨http://www.kpitgnutools.com⟩.
[17] Pinkers, R.P.J., Knijnenburg, P.M.W., Haneda, M. and Wijshoff, H.A.G.: Statistical Selection of Compiler Options, *Proc. IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, *MASCOTS '04*, Washington, DC, USA, pp.494–501, IEEE Computer Society (2004).
[18] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: Optimizing General Purpose Compiler Optimization, *Proc. 2nd Conference on Computing Frontiers*, *CF '05*, New York, NY, USA, pp.180–188, ACM (2005).
[19] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: Code Size Reduction by Compiler Tuning, *Proc. 6th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, *SAMOS'06*, Berlin, Heidelberg, pp.186–195, Springer-Verlag (2006).
[20] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: On the Impact of Data Input Sets on Statistical Compiler Tuning, *Proc. 20th International Conference on Parallel and Distributed Processing*, *IPDPS'06*, Washington, DC, USA, pp.385–385, IEEE Computer Society (2006).
[21] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: Generating New General Compiler Optimization Settings, *Proc. 19th Annual International Conference on Supercomputing*, *ICS '05*, New York, NY, USA, pp.161–168, ACM (2005).
[22] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics, *Proc. 14th International Conference on Parallel Architectures and Compilation Techniques*, *PACT '05*, Washington, DC, USA, pp.123–132, IEEE Computer Society (2005).
[23] Cooper, K.D., Schielke, P.J. and Subramanian, D.: Optimizing for Reduced Code Space Using Genetic Algorithms, *Proc. ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, *LCTES '99*, New York, NY, USA, pp.1–9, ACM (1999).
[24] Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L. and Waterman, T.: Finding Effective Compilation Sequences, *Proc. 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, *LCTES '04*, New York, NY, USA, pp.231–239, ACM (2004).
[25] Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S., Subramanian, D., Torczon, L. and Waterman, T.: Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms, *J. Supercomput.*, Vol.36, No.2, pp.135–151 (2006).
[26] Jantz, M.R. and Kulkarni, P.A.: Eliminating False Phase Interactions to Reduce Optimization Phase Order Search Space, *Proc. 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, *CASES '10*, New York, NY, USA, pp.187–196, ACM (2010).
[27] Kulkarni, S. and Cavazos, J.: Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications*, *OOPSLA '12*, New York, NY, USA, pp.147–162, ACM (2012).
[28] Jantz, M.R. and Kulkarni, P.A.: Exploiting Phase Interdependencies for Faster Iterative Compiler Optimization Phase Order Searches, *Proc. 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, *CASES '13*, Piscataway, NJ, USA, pp.7:1–7:10, IEEE Press (2013).
[29] Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C. and O'Boyle, M.: MILEPOST GCC: machine learning based research compiler, *Proc. GCC Developers' Summit* (2008).
[30] Whitfield, D.L. and Soffa, M.L.: An Approach for Exploring Code Improving Transformations, *ACM Trans. Program. Lang. Syst.*, Vol.19, No.6, pp.1053–1084 (1997).

千葉 雄司　（正会員）

1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．株式会社日立製作所においてコンパイラの開発に従事．中央大学非常勤講師，中央大学大学院客員教授を兼任．



西村 啓成

1983 年生．2008 年関西学院大学大学院理工学研究科情報科学専攻博士課程前期課程修了．ルネサスシステムデザイン株式会社においてコンパイラの開発に従事．

中川 満

1980 年生．2005 年大分大学大学院工学研究科知能情報システム工学専攻博士前期課程修了．ルネサスシステムデザイン株式会社においてコンパイラの開発に従事．