情報処理学会論文誌　プログラミング　Vol.10　No.3　1–12 (June 2017)

RL78マイコン向けレジスタ割付けにおけるロードの最適化

千葉 雄司[1,a)]　西村 啓成[2]

受付日 2016年9月7日, 採録日 2016年12月27日

概要：プロセッサのアーキテクチャはしばしばレジスタの用途に制限を加えるが，そういった制限は，コンパイラにおけるレジスタ割付けの実装を難しくする要因になる．たとえば，レジスタ割付けではメモリに割り付けたデータをロードする命令を挿入することがあるが，ルネサスエレクトロニクスのRL78マイコンでは，ロード命令のロード先として利用可能なレジスタをアキュムレータレジスタのみとしているため，アキュムレータレジスタ以外の汎用レジスタにデータをロードする際にはアキュムレータレジスタの内容を破壊することになり，破壊による悪影響を回避するには，回避に特化した最適化が必要になる．本論文では，この悪影響を回避しつつロードを行うための最適化方法を提案し，その効果を評価した結果を示す．評価の結果，提案技法によって実行を21.49%高速化でき，また，コードサイズを1.58%削減できることが分かった．

キーワード：コンパイラ，レジスタ割付け，マイコン，アキュムレータ

Optimizing Loads by a Register Allocator for the RL78 Microcontroller

Yuji Chiba[1,a)]　Masanari Nishimura[2]

Received: September 7, 2016, Accepted: December 27, 2016

Abstract: A processor architecture often imposes constraints on the registers that are allocatable to a machine instruction operand, and the constraints request specific register allocation techniques. For example, a register allocator often inserts an instruction to load data allocated in memory, but the Renesas RL78 microcontroller allows the instruction to load a value only to the accumulator register, and the constraint implies loads to non-accumulator general purpose registers have side-effect to destroy the contents of the accumulator register, thus the register allocator for the RL78 microcontroller should have optimization to overcome the side-effect. This paper presents the optimization techniques to overcome the side-effect, and evaluates their effects to show that they improved the performance by 21.49% and reduced code size by 1.58%.

Keywords: compiler, register allocation, microcontroller, accumulator

1. はじめに

ローエンドマイコン製品の中には，アキュムレータマシンというアーキテクチャを採用するものがある．アキュムレータマシンとは，演算命令のオペランドの1つを，アキュムレータレジスタと呼ぶ特定のレジスタにすることを要求するアーキテクチャである．アキュムレータマシンは構造が簡潔で，安価に実現できるという利点を持つが，同時に欠点として，レジスタ割付け [1], [2] の実装を難しくするという問題をあわせもつ．

レジスタ割付けはコンパイラを構成する古典的な最適化処理の1つであり，どの場所でどのデータをどのレジスタに配置するのかを定める役割を果たす．アキュムレータマシン向けのレジスタ割付けの実装が難しい理由は，アキュムレータマシンが命令のオペランドとして利用可能なレジスタに制限を加えるからである．オペランドとして利用可能なレジスタに制限を加えるアーキテクチャはアキュム

[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
[a)] yuji.chiba.pd@hitachi.com

```
movw ax, [sp+0x08]
movw de, ax
```
(a) アキュムレータレジスタの退避復帰なし

```
movw [sp+0x0c], ax ; ax の退避
movw ax, [sp+0x08]
movw de, ax
movw ax, [sp+0x0c], ax ; ax の復帰
```
(b) アキュムレータレジスタの退避復帰あり

図 1　アキュムレータでない汎用レジスタへのロード
Fig. 1　Load to a non-accumulator general purpose register.

レータマシンに限らず，そういった制限を加えるアーキテクチャ向けのレジスタ割付けのアルゴリズムの提案は数多く存在する [3], [4], [5], [6], [7], [8], [9], [10]．しかしながらアキュムレータマシンによっては，従来のレジスタ割付けが考慮しなかった制限を加えるものもあり，そのようなアキュムレータマシン向けには固有のレジスタ割付けの技法が必要になる．

たとえば Intel Corporation の 8051 [11] やルネサスエレクトロニクスの RL78 マイコン [12] は，アキュムレータマシンではあるが，汎用レジスタをあわせ持ち，その一方で，ロード命令のロード先のオペランドをアキュムレータレジスタにするよう制限している．このようなアーキテクチャでは，ロード命令によってメモリから汎用レジスタに直接データをロードできないため，レジスタ割付けの設計に際して，メモリに割り付けたデータを，どうやって汎用レジスタにロードするかが問題になる．本論文で提案するのは，この問題に対する RL78 マイコン向けの解の 1 つである．

本論文の構成を次に示す．まず，2 章で RL78 マイコンにおける汎用レジスタへのロードの基本的な実現を示し，続く 3 章でロードを最適化する 3 つの方法を提案する．4 章では，3 つの最適化の使い分け方を検討し，我々の実装を示す．5 章では提案した最適化の評価結果を示し，6 章で関連する研究について述べる．7 章は結論である．

2. RL78 マイコンにおけるロードの実現

RL78 マイコンは 8 本の 8 bit の汎用レジスタ a，x，b，c，d，e，h，l を持ち，これらレジスタを 2 本ずつまとめて 16 bit の汎用レジスタ ax，bc，de，hl として使うこともできる．ここで 16 bit の汎用レジスタは，16 bit 以上の長さのデータを扱う際だけでなく，ポインタを保持する際にも使うものだが，本数が 4 本に限られており，データをレジスタに保持しきれないことも少なくない．保持しきれないデータはメモリにスピルすることになるが，ここで問題になるのがスピルしたデータを参照する方法である．参照する方法の 1 つは，汎用レジスタにロードしてから参照することだが，RL78 マイコンではメモリから任意の汎用レジスタにデータをロードすることができない．メモリからのロード先になれるのはアキュムレータレジスタ a もしくは ax のみであり，他の汎用レジスタにロードする際には，基本的には，**図 1** (a) に示すレジスタ de へのロードの命令列のように，メモリからアキュムレータレジスタ経由でデータをロードする命令列を使うことになる．

```
; 次の命令が ax へのロードなので，ここで ax は空いている
movw ax, !_globalVariable
addw ax, hl
```
(a) ロード挿入前

```
movw ax, !_globalVariable
movw [sp+0x0c], ax ; ax の退避
movw ax, [sp+0x08]
movw hl, ax
movw ax, [sp+0x0c] ; ax の復帰
addw ax, hl
```
(b) ロード挿入後 (最適化なし)

```
movw ax, [sp+0x08]
movw hl, ax
movw ax, !_globalVariable
addw ax, hl
```
(c) ロード挿入後 (最適化あり)

図 2　アキュムレータレジスタが空いている場所でのロード
Fig. 2　Load at a place where the accumulator is not in use.

3. ロードの最適化

図 1 (a) の命令列はロードを実現する手段の 1 つではあるが，アキュムレータレジスタの内容を書き換えてしまうので，アキュムレータレジスタが有意な値を保持している場所では，図 1 (a) の命令列のみではロードを実現できず，何らかの対策が必要になる．対策の 1 つは図 1 (b) の命令列のように，アキュムレータレジスタの内容を退避/復帰するコードを追加することだが，それではコードサイズも実行性能も劣化してしまう．そこで本章では，この劣化を回避しつつロードを行う次の 3 つの手法を提案する．

- アキュムレータレジスタが空いている場所でのロード
- アキュムレータレジスタを書き換えない命令列によるロード
- メモリオペランドの利用

3.1 アキュムレータレジスタが空いている場所でのロード

アキュムレータレジスタが空いている場所でのロードは，アキュムレータレジスタが空いている場所でロードすればアキュムレータレジスタの退避や復帰が不要になることを利用する最適化である．たとえば**図 2** (a) のコードの 2 つ目の命令 addw ax，hl の第 2 オペランド hl にデータをロードする場合について考える．この場合，ロードのコードを命令 addw ax，hl の直前に挿入しようとする

```
movw ax, [sp+0x00]
movw hl, ax
```
(a) データ転送命令による実現

```
pop hl
push hl
```
(b) pop/push による実現

図 3　[sp+0x00] からのロード
Fig. 3　Implementations of load from [sp+0x00].

と，当該箇所でアキュムレータレジスタ ax が有意な値を保持していることから，その退避復帰も必要になり，挿入結果は図 2 (b) に示すとおりになってしまう．このような場合について，本最適化ではロードを行う位置を，アキュムレータレジスタが空いている場所に移動することで，アキュムレータレジスタの退避や復帰を挿入せずに済ませる．たとえば図 2 (a) のコードでは 1 つ目の命令 movw ax, !_globalVariable の直前の時点でアキュムレータレジスタ ax が空いているので，当該箇所でロードを行えば，図 2 (c) に示すとおり，アキュムレータレジスタの退避復帰を挿入せずに済む．

3.2　アキュムレータレジスタを書き換えない命令列によるロード

アキュムレータレジスタを書き換えない命令列によるロードでは，RL78 マイコンの命令 pop，push が任意の 16 bit レジスタをオペランドにとりうることを利用して，[sp+0x00] からのロードを，**図 3** (a) のコードではなく，図 3 (b) のコードで実現する．図 3 (b) のコードならアキュムレータレジスタを書き換えずに済む．

図 3 (a)，(b) に示したロードの実現を比較すると，実行サイクル数は同等で[*1]，コードサイズは，図 3 (b) の実現の方が 1 byte 小さい．しかしながら図 3 (b) の実現には，次に示す 2 つの問題がある．

- 利用可能な箇所が限られる．
- 共通コードの関数化の適用範囲を狭める．

それぞれの問題について順次詳述する．

3.2.1　利用可能な箇所

図 3 (b) のコードを利用可能な箇所が限られる理由は，図 3 (b) のコードでは [sp+0x00] からしかロードできないからである．これに対し図 3 (a) のコードは，オフセット，つまり [sp+0x00] の 0x00 を変更すれば，[sp+0x00] と異なる場所に配置したデータをロードでき汎用性に優れる．

図 3 (b) のコードを利用可能な箇所を増やすための手段として，[sp+0x00] に配置するデータをアクセス箇所数の多いものにすることはできるが，その効果には限りがある．

3.2.2　共通コードの関数化の適用範囲

図 3 (b) のコードではロード元を [sp+0x00] から変更できないが，そのことが原因で，共通コードの関数化 [13] という最適化の適用範囲を狭めてしまうことがある．

[*1] 命令 pop や push はメモリアクセスを行うものの，命令 movw より実行に時間がかかるということはない．

```
_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 4　共通コードの関数化
Fig. 4　Procedual abstraction.

共通コードの関数化は，コードサイズの削減を目的とする最適化であり，削減のために複数の箇所にある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 4** のコードについて考える．図 4 (a) のコードを見ると，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 4 (a) のコードに共通コードの関数化を適用した結果を図 4 (b) に示す．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 4 (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処置が必要になる．図 3 (b) のコードを共通コードにできないことがあるのは，このオフセットをずらす処置を適用できないからである．

3.3　メモリオペランドの利用

メモリオペランドの利用は，メモリに配置したデータを，メモリオペランドを使って直接参照する古典的な最適化技法 [5] である．この技法を使えば，メモリから汎用レジスタ

命令	コメント	コードサイズ (byte)
movw de, ax	; ax の退避	1
movw ax, [sp+0x08]	; subw の右オペランドのロード (1)	2
xchw ax, de	; subw の右オペランドのロード (2) 兼，ax の復帰	1
subw ax, de		1
	合計	5

(a) メモリオペランドを利用しない

movw hl, sp	3
subw ax, [hl+0x08]	3
合計	6

(b) メモリオペランドを利用する

図 5　メモリオペランドの利用
Fig. 5　Using a memory operand.

にデータをロードする必要がなくなるので，アキュムレータレジスタの内容を破壊せずに済む．

RL78 マイコンにおけるメモリオペランドの利用の例を**図 5** に示す．図 5 のコードは減算命令 subw の右オペランドが参照するデータをロードする必要が生じた場合のもので，図 5 (a) のメモリオペランドを利用しないコードには，右オペランド向けのロード命令に加え，アキュムレータレジスタ ax を退避/復帰する命令も入っているのに対し，図 5 (b) のメモリオペランドを利用するコードには，右オペランド向けのロード命令もアキュムレータレジスタ ax を退避/復帰する命令もなく，代わりに減算命令 subw の右オペランドがメモリオペランドに変化し，さらに，メモリオペランドのベースレジスタが hl であることから，レジスタ hl にスタックポインタ sp の内容を複写する命令が加わっている．この複写命令は，減算命令に限らず，算術/論理演算命令全般において，メモリオペランドのベースレジスタとして利用できるレジスタが汎用レジスタ hl だけで，スタックポインタ sp は利用できないという RL78 マイコンに固有な事情から必要になるものである．

図 5 (b) のコードから分かるように，RL78 マイコンの算術/論理演算命令はメモリオペランドを利用する際に汎用レジスタ hl を必要とするが，このことは次に示す RL78 マイコンに固有な問題の原因となる．メモリオペランドの利用自体は古典的な最適化技法だが，これらの問題の解決方法に関する提案は過去にない．本節ではこれらの問題について順次詳述し，解決策を提案する．

(1) コードサイズへの影響
(2) 複写命令 movw hl，sp の挿入先
(3) レジスタ hl の用途

3.3.1　コードサイズへの影響

RL78 マイコン向けの最適化では，最適化の目的が実行速度の改善にある場合には，メモリオペランドを積極的に利用すべきだが，コードサイズの削減にある場合にはそうとも限らない．たとえば図 5 に示した例について述べると，命令数の観点ではメモリオペランドを利用しない方が 4 命令，利用する方が 2 命令で，利用する方が命令数が少なく実行速度も速い．しかしながら，コードサイズに関しては，利用しない方が合計 5 byte であるのに対し，利用する方が 6 byte となっており，メモリオペランドの利用によってコードサイズが大きくなってしまっていることが分かる．

もっとも，メモリオペランドの利用によってコードサイズを小さくできる場合もある．図 5 (b) のメモリオペランドを利用する場合のコードサイズの欄から分かるように，コードサイズが大きくなる主な原因は，複写命令 movw hl，sp のコードサイズが 3 byte と大きいことにあるが，この複写は必ずしもメモリオペランドを利用する箇所ごとに必要になるものではなく，メモリオペランドを利用する複数の箇所で 1 つの複写命令を共有できる場合もあり，その場合，メモリオペランドを利用する方がコードサイズが小さくなりうる．

たとえば**図 6** (a) の中間表現にレジスタを割り付ける場合について考える．図 6 (a) では命令のオペランドをたとえば %reg1024 と表記しているが，ここで %reg1024 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．レジスタ割付けの結果，図 6 (a) 中の 2 つの減算 subw の右オペランド %reg1025，%reg1027 がともにスピルの対象になり，個々の減算の時点でロードする必要が生じたものとする．このときメモリオペランドを利用せずにコンパイルした結果は図 6 (b) のとおりで，そのコードサイズは合計 10 byte，メモリオペランドを利用してコンパイルした結果は図 6 (c) のとおりで，そのコードサイズは合計 9 byte である．すなわち，図 6 の例では，図 5 の例とは異なり，メモリオペランドを利用する方がコードサイズを小さくできるが，その理由は，2 つのメモリオペランドが 1 つの複写命令 movw hl，sp を共有していることにある．

図 6 の例のように，コードサイズが減る箇所だけを選ん

```
%reg1026 = subw %reg1024, %reg1025
%reg1028 = subw %reg1026, %reg1027
```

(a) レジスタ割付前

命令	コードサイズ (byte)
movw bc, ax	1
movw ax, [sp+0x04]	2
xchw ax, bc	1
subw ax, bc	1
movw ax, bc	1
movw ax, [sp+0x06]	2
xchw ax, bc	1
subw ax, bc	1
合計	10

(b) メモリオペランドを利用しない

movw hl, sp	3
subw ax, [hl+0x04]	3
subw ax, [hl+0x06]	3
合計	9

(c) メモリオペランドを利用する

図 6　複写命令 movw hl, sp の共有
Fig. 6　Sharing a copy instruction movw hl, sp.

でメモリオペランドを利用するためには，複写命令 movw hl, sp を何カ所挿入する必要があるのかあらかじめ見積もって，挿入しても全体ではコードサイズを小さくできるか判断する必要がある．

3.3.2　複写命令 movw hl, sp の挿入先

図 6 (c) の例では 2 つのメモリオペランドが 1 つの複写命令 movw hl, sp を共有しているが，この共有を実現するためには最適化処理によって複写命令の挿入先を必要最低限に絞り込む必要がある．この絞り込みの最適化は，その目的が冗長な複写命令の削減のみであるなら，次の要領で実現することもできる．

(1) レジスタ割付けの際にはメモリオペランドを利用する全命令の前に複写命令を挿入しておく．
(2) レジスタ割付けの後で大域的な共通部分式の削減のような汎用の最適化処理を実施することで，冗長な複写命令を削除する．

この実現にはレジスタ割付けの実装を単純にできるといった利点はあるが，レジスタ割付けの時点で複写命令がいくつどこに残るか分からないという欠点がある．したがって，この実現は，メモリオペランドをコードサイズの削減に活用したい場合，すなわち，レジスタ割付けの時点で複写命令のもたらすコードサイズの増加量を見積もりたい場合には有益でない．メモリオペランドをコードサイズの削減に活用することを目指すならば，レジスタ割付けの一部として複写命令 movw hl, sp の挿入箇所を定める処理を実装する必要がある．ここで必要最低限の複写命令の挿入箇所を求めるには，次の地点をソース/シンクとする最小カット問題を解けばよい．

ソース：関数の入口，sp を定義する命令の直後，レジスタ h/l/hl の他用途への使用の直後
シンク：メモリオペランドを利用する命令の直前

この最小カット問題の解があれば，コードサイズの増加量を見積もることが可能になり，したがって，どこでメモリオペランドを利用すべきか判断できるようになる．しかしながら，この最小カット問題は，レジスタ h/l/hl の他の用途に使う箇所を定めた後でないと解けない．他の用途に使う箇所が定まるのはレジスタ割付けの終了後になるが，レジスタ割付けの終了後に解を得るのでは遅い．なぜなら，どこでメモリオペランドを利用すべきかの判断が必要になるのは，レジスタ割付けの最中だからである．

最小カット問題の解を得られるタイミングが遅いという問題は，次の方法で解決できるが，それぞれに固有な問題がある．個々の解決策とその問題について順次述べる．

- メモリオペランドの利用の中止
- レジスタ割付けのやり直し

3.3.2.1　メモリオペランドの利用の中止

メモリオペランドの利用の中止は，レジスタ割付けの終了後，メモリオペランドを利用するとコードサイズが増えてしまうと判明した箇所を，メモリオペランドを利用しないコードに書き換える技法である．たとえば**図 7** (a) のメモリオペランドを使う命令列は図 7 (b) のメモリオペランドを使わない命令列に書き換えることができる．

図 7 (b) のコードでは add の右オペランドがレジスタ l になっているが，その理由は，中止前の add が確保していたレジスタ hl の一部を利用したためである．しかしながら，図 7 (b) のコードは，次の理由でレジスタを活用しきっていない可能性がある．

- 中止前の add が確保していたレジスタ hl，すなわちレジスタ h と l のうち，レジスタ l しか使っていないので，レジスタ h が空いている．
- 中止後の add の右オペランドをレジスタ l にすることが最適とは限らない．たとえば add の時点でレジスタ b が空いているなら b を使い，レジスタ hl を別の用途に使う方が適切な場合もありうる．

	mov a, [sp+0x05]
	mov l, a
mov a, [sp+0x04]	mov a, [sp+0x04]
movw hl, sp	
add a, [hl+0x05]	add a, l

(a) 中止前　(b) 中止後

図 7　メモリオペランドの利用の中止
Fig. 7　Memory operand usage cancellation.

```
 1:  BB0:
 2:    movw ax, [hl] ; 他用途への hl の使用の終端 (直後がソース)
 3:    addw ax, ** ; **がメモリオペランドの利用箇所 (直前がシンク)
 4:    br BB2
 5:  BB1:
 6:    movw ax, [hl] ; 他用途への hl の使用の終端 (直後がソース)
 7:    addw ax, ** ; **がメモリオペランドの利用箇所 (直前がシンク)
 8:    incw ** ; **がメモリオペランドの利用箇所 (直前がシンク)
 9:  BB2:
10:    subw ax, ** ; **がメモリオペランドの利用箇所 (直前がシンク)
```

図 8　部分問題への分割
Fig. 8　Division to subproblems.

3.3.2.2　レジスタ割付けのやり直し

これらの問題を解決するために，たとえば次の要領でレジスタ割付けを行うことは可能である．

(1) レジスタ割付けを行う．
(2) 最小カット問題を解く．
(3) メモリオペランドの利用によってコードサイズが増えてしてしまう箇所をみつけたら，当該箇所でメモリオペランドを利用しないよう記録を残して (1) に戻る．

しかしながら，この方法ではレジスタ割付けをやり直すことになるため，コンパイル時間に悪影響を及ぼすことになる．したがって，この方法を使うのは，やり直しのもたらすメリットが，コンパイル時間の増加というデメリットを上回る場合に限るべきである．

ただ，レジスタ割付けのやり直しがもたらすメリット，すなわちコードサイズの削減の量は容易に見積もれない．なぜなら，レジスタ割付けをやり直す場合，メモリオペランドの利用の中止では活用しきれなかったレジスタを活用できるため，中止した箇所とは直接関係ない箇所でコードサイズを削減できることがあるからである．直接関係ない箇所で生じる削減の量は正確には見積もり難いが，我々の実装では，様々なアプリケーションを用いた評価の結果から，直接関係ない箇所から生じる削減の量を，メモリオペランドを利用する箇所から生じる削減の量の半分と見積もっている．

3.3.2.3　最小カット問題の分割

メモリオペランドの利用の中止やレジスタ割付けのやり直しを実施するか否か判断する際には，まずメモリオペランドの利用によってコードサイズが減るか否かを見積もる必要があるが，見積りに際しては，どの単位で見積もるかが問題になる．一案は関数単位で見積もり，見積りの結果，コードサイズが増えると分かったら，関数内のすべての箇所でメモリオペランドの利用を止めることだが，それでは関数内の一部の箇所で利用するとコードサイズを小さくできるケースに対応できない．

この問題を解決するには，メモリオペランドの利用を止めるかどうかの見積りと判断を，関数より細かな単位で実施すればよい．その具体案の 1 つは，最小カット問題を分割し，分割して得た部分問題を見積りと判断の単位とすることである．ただし，一般に最小カット問題はソースとシンクを断面として部分問題に分割できるが，この一般的な分割方法によって見積りと判断の単位とする部分問題を求めるのは適切でない．たとえば**図 8** のコードに対応する最小カット問題について考える．図 8 のコードに対応する最小カット問題は一般には次の 2 つの部分問題に分割できる．

部分問題 1　2 行目をソース，3 行目をシンクとする問題
部分問題 2　6 行目をソース，7 行目をシンクとする問題

しかしながら，これらの部分問題を見積りや判断の単位とすることには次の欠点がある．

- 8，10 行目におけるメモリオペランドの利用がもたらすコードサイズの削減量をどの問題の見積りに含めるべきかが定かでない．
- 8，10 行目にあるメモリオペランドの利用箇所に与える影響を考慮すると，利用を止めるかどうかの判断を個別の問題ごとに下せない．

これらの欠点に対処する方法の 1 つは，部分問題に分割する際に，シンクのうち，他のシンクから制御フローを遡ってソースを経由せずに到達できるものを断面にしないことである．この方法を使うと，たとえば図 8 のコードでは 3，7，8 行目のシンクが断面にならず，結果として図 8 の全体が 1 つの問題になる．我々の実装では，この要領で最小カット問題を部分問題に分割する．そして，個々の部分問題ごとに，メモリオペランドの利用によってコードサイズが減るかどうかを見積もり，減るならば部分問題内にあるメモリオペランドの利用箇所の全候補でメモリオペランドを利用し，減らないなら全候補にメモリオペランドを利用させない．

なお，ここでメモリオペランドを利用するか否かを，部分問題内の候補に対して一律に定める必要は必ずしもない．たとえば図 8 のコードについて，6 行目の直後にのみ複写命令 `movw hl, sp` を挿入して，7，8 行目でのみメモリオペランドを利用し，他では利用しなくてもよい．ただ

表 1　提案した 3 つの最適化の比較
Table 1　Comparison of the proposed 3 optimizations.

提案した最適化	適用に必要な条件	ロードに必要な命令数
アキュムレータレジスタが空いている場所でのロード	アキュムレータレジスタの空いている場所があること	2
アキュムレータレジスタを書き換えない命令列によるロード	ロード元が [sp+0x00] であること	2
メモリオペランドの利用	メモリオペランドへの書換が可能であること	0
	汎用レジスタ hl が空いていること	

し，我々の評価した範囲では，一律でない解が有益であったことはなかった．

3.3.3　レジスタ hl の用途

メモリオペランドを利用する際には，ベースレジスタとしてレジスタ hl を確保する必要があるが，レジスタ hl は他の用途にも利用できるものなので，用途を定めるにあたっては，何に使うのが有益か判断する必要が生じる．

しかしながら，この判断の最適解は容易には求められない．なぜならこの判断はレジスタ割付けにおける，どこでどのレジスタを何に使うべきか，という判断の一部であり，その全体の最適解を求める問題は，一般には NP 完全問題だからである．

そこで本論文では，この問題の近似解を得るために，次のヒューリスティックを利用すること提案する．

(1) 割り付けるレジスタに選択の余地があるならば，レジスタ hl/h/l を選ばないようにする．
(2) ただし，メモリオペランドに書き換えられるオペランドのうち，参照する仮想レジスタがスピルの対象になると予測できるものには，レジスタ割付けの開始時にレジスタ hl/l を割り振ってしまう．

これらのヒューリスティックの目的は，ともに，レジスタ割付けの過程で，メモリオペランドを利用したいのに利用できないという事態が発生するのを防ぐことにある．すなわち，メモリオペランドを利用するためには，利用する命令をまたぐ生存区間にレジスタ hl/h/l を割り付けてはならないが，(1) のヒューリスティックでは極力，レジスタ hl/h/l を使わないようにすることでメモリオペランドを利用できる機会を増やし，(2) のヒューリスティックでは他の生存区間に割り付けるより前にメモリオペランドを利用する候補にレジスタ hl/l を割り振ってしまうことで利用ができなくなることを防ぐ．(2) のヒューリスティックを実現する際には仮想レジスタがスピルの対象になるかどうかを，どう予測するかが問題になるが，我々の実装では，仮想レジスタを参照する命令の前後 3 命令が同一の仮想レジスタを参照しない場合にはスピルの対象になるものと予測することにした．

4.　実装

3 章で提案した 3 つの最適化はいずれも任意のロードに適用できるものでなく，**表 1** に示すとおり，それぞれが必要とする条件を満たすロードにしか適用できない．しかしながらロードに適用できる最適化は 1 つでない場合もあり，そのような場合，どの最適化を適用するのかが問題になる．この問題への対応として，我々の実装では，最適化を適用する優先順位を次のとおりに定めた．

(1) メモリオペランドの利用
(2) アキュムレータレジスタが空いている場所でのロード
(3) アキュムレータレジスタを書き換えない命令列によるロード

優先順位をこのように定めた根拠を次に示す．

- メモリオペランドの利用は 3 つの最適化の中で，実行速度を最も大きく改善できる．コードサイズの改善量は多いとは限らないが，メモリオペランドを利用する箇所を増やせば，より多くのメモリオペランドの利用箇所が複写命令 movw hl，sp を共有し，結果として，コードサイズの改善量が大きくなる．これらの理由からメモリオペランドの利用を最優先で適用する．
- 残り 2 つの最適化の適用に必要な条件を比較すると，アキュムレータレジスタが空いている場所でのロードの方は，レジスタ割付けより前に定まっている要因（命令の並び順）によって適用の可否が依存することが分かる．そこでアキュムレータレジスタが空いている場所でのロードの適用を先に試み，残りをアキュムレータレジスタを書き換えない命令列によるロードで最適化できるか試みることにする．

我々は提案した最適化をルネサスエレクトロニクスの最適化コンパイラ製品 CC-RL [14] に実装した．CC-RL はオープンソースのコンパイラプラットフォーム llvm-2.3 をベースに開発したもので，フロントエンドとバックエンドの 2 つの構成要素からなる．フロントエンドはソースコードを機種非依存の中間表現（ビットコード）に変換する役割を果たし，バックエンドはビットコードをアセンブリコードに変換する役割を果たす．ここでバックエンドの概要は次のとおりとなっている．バックエンドはビットコードを受け取ると，まず最初にビットコードへ機種非依存な最適化を適用し，次にビットコードを DAG（Directed Acyclic Graph）形式の中間表現に変換して命令選択を行う．命令選択により，中間表現を機械命令と 1:1 に対応

```
 1:  OperandSet nonProfitableOperands = φ;
 2:  do{
 3:    graph = createInterferenceGraph();
 4:    precolor(graph);
 5:
 6:    color(graph);
 7:    useMemoryOperands(graph, nonProfitableOperands);
 8:    color(graph);
 9:
10:    decideSiteToLoad(graph);
11:    discardUncoloredNodes(graph);
12:    createSpillNodes(graph);
13:    color(graph);
14:
15:    decideStackFrameLayout(graph);
16:    decideSiteToLoadWithoutClobberingAccumulator(graph);
17:    color(graph);
18:  }while(decideSiteToCopyFromSPToHL(nonProfitableOperands));
```

図 9　彩色処理のアルゴリズム
Fig. 9　Coloring algorithm.

するもの（MachineInstr）に変換したら，MachineInstr に機種依存な最適化を適用し，最後に MachineInstr をアセンブリコードに変換する．ここでレジスタ割付けは MachineInstr 向けの最適化処理の 1 つである．llvm ではビットコードへの最適化で，間接参照の対象にならないデータを仮想レジスタにのせるので，レジスタ割付けでは，それら仮想レジスタの生存区間のどこからどこまでにどの物理レジスタを割り付けるかを定める処理を行う．

我々が実装のベースとして利用した CC-RL のレジスタ割付け [15] は，生存区間の干渉グラフを構築し，構築したグラフの彩色問題を解くことでレジスタ割付けを行う．本論文で提案した最適化を組み込んだ後の彩色処理の全体像を**図 9** に示す．

図 9 の処理では，まず，1 行目でメモリオペランドにするとサイズに悪影響を与えるオペランドの集合 nonProfitableOperands を空にする．この集合は 3.3.2.2 で述べたレジスタ割付けのやり直しのためのものである．続く 3 行目で干渉グラフを構築する．構築する干渉グラフのノードは，仮想レジスタの生存区間そのものではなく，その断片である．この断片は，仮想レジスタの生存区間を，当該仮想レジスタを参照する点の前後で切り分けて作成する．結果としてできる断片は，次のいずれかになる．

- 参照する点のみを含む区間
- 参照する点と点を結ぶ区間（点は含まない）

次に 4 行目に進み，参照する点のみを含む区間のうち，対応するオペランドに彩色できるレジスタが 1 色のみのものの彩色と，3.3.3 項で述べたヒューリスティックの 2 つ目，すなわちメモリオペランドに書き換えられるオペランドのうち，参照する仮想レジスタがスピル対象になると予測できるものの彩色を行う．

続く 6 行目では干渉グラフ中のノードの彩色を，彩色できるものがなくなるまで繰り返し試みる．彩色は次の規則に従って実施する．

- 参照する点のみを含む区間に対応するノードは，対応するオペランドに割り付けられるレジスタのいずれかに彩色する．
- 参照する点と点を結ぶ区間に対応するノードは，隣接する同一仮想レジスタの生存区間の断片のノードと同じレジスタに彩色できる場合に限り，隣接する断片のノードと融合したうえで彩色する．この際，隣接する断片のノードの色を上塗りするので，結果として 4 行目で実施した彩色をキャンセルすることがある．

点と点を結ぶ区間に対応するノードをアキュムレータレジスタに彩色する際には，干渉するノードがアキュムレータレジスタに彩色済みでないか，だけでなく，当該区間内に，別の仮想レジスタのデータをレジスタからメモリへストアする，あるいはメモリからレジスタにロードする箇所がないかということにも注意を払う．なぜならこれらの箇所でもデータのストアやロードのためにアキュムレータレジスタが必要になるからである．ここでレジスタからメモリへのストアは，当該レジスタに値を書き込む命令の直後で行うものとし，メモリからレジスタへの値のロードは，この時点では，レジスタから値を読み込む命令の直前で行うものと想定する（3.1 節で述べた，ロードを行う位置をアキュムレータレジスタが空いている場所にする処理は，10 行目で行う）．

次の 7 行目では，3.3 節で提案した最適化，すなわちメモリオペランドに書き換えるレジスタオペランドを定める

処理を行う．ここでは次の条件を満たすノードに対応するオペランドを書き換えの対象と見なし，当該ノードを汎用レジスタ hl に彩色する．

- メモリオペランドに書き換え可能なレジスタオペランドに対応するノードで，6 行目の彩色処理で隣接する生存区間の断片に対応するノードと融合されていない．
- 干渉するノードが汎用レジスタ h/l/hl のいずれにも彩色されていない．

ここでメモリオペランドに書き換えることにしたノードは，オペランドにデータをロードするためにアキュムレータレジスタを必要としなくなる．これによって別のノードをアキュムレータレジスタに彩色できる機会が生まれることがあるので，続く 8 行目で再度，彩色を試みる．彩色の過程で，7 行目の処理においてメモリオペランドに書き換えると定めたオペランドに対応するノードが，隣接する生存区間の断片のノードと融合可能になる場合があるが，その場合は 7 行目での決定を覆して，オペランドをレジスタオペランドに戻し，融合することを優先する．

次に 10 行目で，スピルすることにしたデータをメモリからレジスタにロードする位置を確定する．ロードを行う位置は，3.1 節で提案したように，可能ならばアキュムレータレジスタが空いている場所とし，可能でないならロード先のレジスタからデータを読む命令の直前とする．10 行目でロードする位置が確定したら，11 行目でこれまでに彩色できなかったノードを捨て，代わりに 12 行目でメモリ上でのデータの生存区間に相当するノードを作成する．メモリ上でのデータの生存区間はデータをロードする場所（メモリオペランドで参照する位置も含む）から制御フローを遡り，データを生成する命令に到達するまでに通過する区間とする．続く 13 行目では，12 行目で生成したノードへの彩色を試みる．これは空いているレジスタをデータのストア先として利用するための処理であるのと同時に，10 行目でアキュレータレジスタの使用箇所を移動した結果，アキュムレータレジスタに彩色可能になったノードを彩色する処理でもある．13 行目の彩色処理で，彩色可能と判断したノード（メモリ上での生存区間）からデータを読むノードの中に，7 行目でメモリオペランドに変更すると決めたノードがある場合には，8 行目の彩色処理と同様に，7 行目での決定を覆し，ノードを融合したうえで彩色する．

ここまでの処理で，どのデータをメモリに配置するのかが決まるので，続く 15 行目でそれらのデータをスタックフレームのどこに配置するのかを定める．15 行目の処理により，3.2 節で提案したアキュムレータレジスタを書き換えない命令列によるロードを利用可能な場所が定まるので，16 行目でその場所を記録する．この記録により，アキュムレータレジスタを書き換えると見なさなければならない箇所が減り，結果として，減った箇所をまたぐノードをアキュムレータレジスタに彩色できる可能性が生まれるので，続く 17 行目で再度，彩色処理を行う．最後に 18 行目でメモリオペランドの利用に必要な命令 movw hl, sp の挿入箇所を定めると同時に，最適化の目的がコードサイズの削減である場合には，メモリオペランドの利用によってコードサイズが増えないか検証する．検証の結果，増えるならば，増加の原因となったオペランドを集合 nonProfitableOperands に追加したうえで 3 行目に戻り，レジスタ割付けをやりなおす．

5. 評価

我々が提案し，実装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．

5.1 実行速度への影響

実行速度向け最適化の評価では，本論文で提案した最適化をすべて適用した場合の実行サイクル数と，評価対象の最適化のみ適用しない場合の実行サイクル数を比較することで，評価対象の最適化の効果を求めた．評価対象のプログラムとしては組込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 [16] を用いた．CoreMark は，現実的な組込機器向けアプリケーションが含みうる演算から，組込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．プログラムの実行には RL78 マイコンそのものではなく，RL78 マイコンの CPU シミュレータを用いた．当該 CPU シミュレータを用いて評価対象のプログラムを実行する場合，実行サイクル数の測定結果は，プログラムのコンパイル結果に対して唯一に定まり，実行ごとに変化することはない．測定の結果を**表 2** に示す．表 2 の性能向上率は次に式によって求めた．

$$\frac{\text{評価対象の最適化項目のみ適用しなかった場合の実行サイクル数}}{\text{提案した最適化をすべて適用した場合の実行サイクル数}} - 1$$

表 2 から，個々の最適化項目の性能向上率の総和より，全項目を適用しなかった場合に対する性能向上率 21.49%の方が大きいことが分かるが，その理由は，(a) のアキュムレータレジスタが空いている場所でのロードと，(c) のメモリオペランドの利用で最適化対象が重複していることにある．すなわち，(c) を抑止しても (c) で最適化していた命令の一部には (a) の最適化がかかってしまうので，(c) だけを実施しない場合に生じる実行速度の劣化率は大きくなり

表 2　実行速度への影響
Table 2　Effect on performance.

最適化項目	性能向上率（%）
(a) アキュムレータレジスタが空いている場所でのロード	0.36
(b) アキュムレータレジスタを書き換えない命令列によるロード	0.00
(c) メモリオペランドの利用	15.96
(d) 全項目（(a)，(b)，(c)）	21.49

表 3　コードサイズ向けベンチマークの内訳
Table 3　Benchmark items for code size.

プログラム	ベンチマークが含む代表的な処理
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 4　コードサイズへの影響
Table 4　Effect on code size.

最適化項目	削減率（%）
(a) アキュムレータレジスタが空いている場所でのロード	0.81
(b) アキュムレータレジスタを書き換えない命令列によるロード	0.22
(c) メモリオペランドの利用	0.11
(d) 全項目（(a)，(b)，(c)）	1.58

にくいといえる．

なお実行速度への影響の評価では`-Ospeed`の指定時には 3.3.3 項で述べたヒューリスティック (1)，(2) の双方を利用した．双方を利用すると，ヒューリスティック (1) のみ利用する場合に比べ，性能が 3.57%向上したので，実行速度を目的とした最適化ではヒューリスティック (2) が有意に働いているといえる．

5.2　コードサイズへの影響

コードサイズ向け最適化の評価では，本論文で提案した最適化をすべて適用した場合のコードサイズと，測定対象の最適化のみ適用しない場合のコードサイズを比較することで，測定対象の最適化の効果を求めた．評価対象のプログラムは，CoreMark ではなく，EEMBC [17] の**表 3** に示すプログラムとした．表 3 のプログラムはいずれも現実的な組込機器向けアプリケーションであり，組込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 3 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．測定の結果を**表 4** に示す．表 4 の削減率は次の式によって求めた．

$$\frac{\text{評価対象の最適化項目のみ適用しなかったコードのサイズの総和}}{\text{提案した最適化をすべて適用したコードのサイズの総和}} - 1$$

表 4 では，個々の最適化項目がもたらすコードサイズの削減率の総和より，全項目を適用しなかった場合に対する削減率 1.58%の方が大きくなっている．その理由は表 2 と同様で，本論文で提案した最適化の最適化対象が重複していることにある．

表 4 では，アキュムレータレジスタを書き換えない命令列によるロードがもたらす削減率が 0.22%と 0%より大きくなっているが，これはアキュムレータレジスタを書き換えない命令列によるロードのメリットがデメリットを上回ることを意味する．アキュムレータレジスタを書き換えない命令列のデメリットとは，3.2.2 項で述べた，共通コードの関数化の適用範囲を制限することである．共通コードの関数化は，オプション`-Osize`を指定した際に動作するので，コードサイズの評価の際には動作しているが，これを抑止した環境で，アキュムレータレジスタを書き換えない命令列によるロードのもたらすコードサイズの削減率を評価したところ，結果は 0.25%と，表 4 に示した 0.22%より大きかった．

表 4 では，メモリオペランドの利用がもたらす削減率は 0.11%となっている．この 0.11%という値は 3.3.2.2 で述べたレジスタ割付けのやり直しを実施した場合のものである．ここでレジスタ割付けのやり直しによるコンパイル時間の増加率は 17%であったので，コンパイル時間を 17%増やしてでもコードサイズを 0.11%削減したいのでなければ，レジスタ割付けのやり直しによってコードサイズの削減を試みない方が良いといえる．レジスタ割付けのやり直しのほ

表 5　コードサイズの増加抑止処理の効果
Table 5　Effect of coalesce site selection for code size reduction.

メモリオペランド	抑止処理	削減率（%）
利用する	レジスタ割付けのやり直し	0.11
利用しない	-	0.00
利用する	利用の中止	−0.19
利用する	なし	−0.81

かに，3.3.2.1 で述べたメモリオペランドの利用の中止も，メモリオペランドの利用がもたらすコードサイズの増加を抑止する手段になるが，増加の抑止手段の効果を評価した結果は**表 5** に示すとおりである．表 5 はメモリオペランドを利用しない場合を基準としたコードサイズの削減率を示すものだが，表 5 から分かるように，メモリオペランドの利用の中止ではコードサイズの増加を抑止しきれず，メモリオペランドを利用しない場合に比べ，コードサイズが 0.19%増加してしまっている．もっとも，メモリオペランドの利用の中止にも一定の効果はある．表 5 から分かるように，コードサイズの増加の抑止手段をまったく適用しない場合に比べれば，メモリオペランドの利用を中止する方がコードサイズを小さくできている．

なおコードサイズへの影響の評価では 3.3.3 項で述べたヒューリスティック (1) のみを利用した．これはヒューリスティック (2) も利用すると，メモリオペランドを利用しない場合よりコードサイズが 0.19%増加したからである．メモリオペランドの利用によってコードサイズが増加する場合にはレジスタ割付けをやり直しているにもかかわらず，コードサイズが増加してしまう理由は，ヒューリスティック (2) はレジスタ `hl` もしくは `1` を割り付けるものであって，直接メモリオペランドの利用を強制するわけではないので，メモリオペランドの利用箇所におけるコードサイズの増減を根拠にレジスタ割付けをやり直すか否かを定めるだけでは，ヒューリスティック (2) がコードサイズに悪い影響をもたらしている箇所を解消できないからである．

6. 関連研究

RL78 マイコンは命令の個々のオペランドとして利用可能なレジスタに，様々な制限を加えているが，本論文ではその中でも特に，ロード命令のロード先をアキュムレータレジスタとする制限がレジスタ割付けにもたらす影響を明らかにした．そして，その影響を軽減する手法を提案し，提案した手法の効果を明らかにした．同様の制限は Intel Corporation の 8051 アーキテクチャにもみられるが，本論文と同様の議論を行った論文は過去にない．

命令のオペランドとして利用可能なレジスタに制限を加えるアーキテクチャ向けのレジスタ割付けの研究は過去に数多く存在するが [3], [4], [5], [6], [7], [8], [9], [10]，それらはロード命令のロード先のオペランドに任意のレジスタを割り付けられることを前提としているため，それらの成果をそのまま RL78 マイコンに適用することはできない．ただし，RL78 マイコンが加える制限のうち，本論文で扱ったものとは異なる制限への対応にこれらの研究の成果を活用することは可能である．過去の研究の中にはレジスタ割付けで挿入するストアやロードのコストを正確に見積もり，整数計画法などの手段を用いて，その最小値をもたらす解を求めるものもある [5], [6], [8]．そういった手法を利用すると，ヒューリスティックを利用する現在の我々の実装より良い解を得られる可能性がある．

7. 結論

本論文では RL78 マイコン向けのレジスタ割付けにおいて，メモリに配置したデータをロードする際に，アキュムレータレジスタの内容を書き換えることが問題になると指摘し，問題を回避するための最適化技法を提案した．評価の結果，提案技法によって実行を 21.49%高速化でき，また，コードサイズを 1.58%削減できることが分かった．

参考文献

[1] Chaitin, G.J.: Register Allocation & Spilling via Graph Coloring, *Proc. 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN '82*, New York, NY, USA, ACM, pp.98–105 (online), DOI: 10.1145/800230.806984 (1982).

[2] Briggs, P., Cooper, K.D. and Torczon, L.: Improvements to Graph Coloring Register Allocation, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.3, pp.428–455 (online), DOI: 10.1145/177492.177575 (1994).

[3] Weber, M. and Bernstein, S.L.: Global Register Allocation for Non-equivalent Register Sets, *SIGPLAN Not.*, Vol.15, No.2, pp.74–81 (online), DOI: 10.1145/947586.947594 (1980).

[4] Kong, T. and Wilken, K.D.: Precise Register Allocation for Irregular Architectures, *Proc. 31st Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 31*, Los Alamitos, CA, USA, pp.297–307, IEEE Computer Society Press (online), available from ⟨http://dl.acm.org/citation.cfm?id=290940.291002⟩ (1998).

[5] Appel, A.W. and George, L.: Optimal Spilling for CISC Machines with Few Registers, *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01*, New York, NY, USA, ACM, pp.243–253 (online), DOI: 10.1145/378795.378854 (2001).

[6] Scholz, B. and Eckstein, E.: Register Allocation for Irregular Architectures, *SIGPLAN Not.*, Vol.37, No.7, pp.139–148 (online), DOI: 10.1145/566225.513854 (2002).

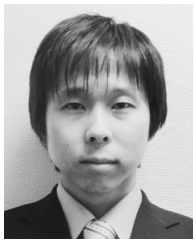
[7] Daveau, J.-M., Thery, T., Lepley, T. and Santana, M.: A Retargetable Register Allocation Framework for Embedded Processors, *Proc. 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '04*, New York, NY, USA, ACM, pp.202–210 (online), DOI: 10.1145/997163.997192 (2004).

[8] Koes, D. and Goldstein, S.C.: A Progressive Regis-

ter Allocator for Irregular Architectures, *Proc. International Symposium on Code Generation and Optimization, CGO '05*, Washington, DC, USA, IEEE Computer Society, pp.269–280 (online), DOI: 10.1109/CGO.2005.4 (2005).
[9] Ahn, M., Lee, J. and Paek, Y.: Optimistic Coalescing for Heterogeneous Register Architectures, *Proc. 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES '07*, New York, NY, USA, ACM, pp.93–102 (online), DOI: 10.1145/1254766.1254781 (2007).
[10] Ahn, M. and Paek, Y.: Register Coalescing Techniques for Heterogeneous Register Architecture with Copy Sifting, *ACM Trans. Embed. Comput. Syst.*, Vol.8, No.2, pp.16:1–16:37 (online), DOI: 10.1145/1457255.1457263 (2009).
[11] Atmel Corporation: 8051 Architecture Microcontroller (2016). available from ⟨http://www.atmel.com/products/microcontrollers/8051Architecture/default.aspx⟩.
[12] Renesas Electronics Corporation: RL78 Family (2015). available from ⟨http://www.renesas.com/products/mpumcu/rl78/⟩.
[13] 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).
[14] 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⟩.
[15] 千葉雄司，西村啓成，中川　満：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，情報処理学会論文誌プログラミング（PRO），Vol.9, No.3, pp.1–20 (2016).
[16] The Embedded Microprocessor Benchmark Consortium: CoreMark An EEMBC Benchmark (2009). available from ⟨http://www.eembc.org/coremark/⟩.
[17] The Embedded Microprocessor Benchmark Consortium: Industry-Standard Benchmarks for Embedded Systems (1997). available from ⟨https://www.eembc.org⟩.

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．