Vol.2015-OS-135 No.11
Vol.2015-EMB-39 No.11
2015/11/24

同一 CPU 間でのバイナリコード比較についてのケーススタディ

赤星 博輝[†1]　松永 裕介[†2]

概要：近年の組込みシステムでは、開発したソースコードをコンパイラやマイコンを変更して利用することが多くあるが、検証作業は再度やり直すことが多い。またその検証も実行ベースのテストに依存しており、その正しさについても課題が存在する。本稿では、同一 CPU に対してバイナリレベルでのコード比較について整理し、実際のコード事例を用いて何を検証すれば良いのかについて調査した結果について報告する。

キーワード：バイナリコード比較

Case Study: binary codes comparison for CPU

HIROKI AKABOSHI[†1]　YUSUKE MATSUNAGA[†2]

Abstract: In recent embedded systems, developed source codes are reused for different CPU platform and compiler system. Code reuse makes development time short but test process is still required because same source code does not mean expected behavior. This paper reports binary level code comparison result with some examples and what should be checked.

Keywords: Binary code checking

1. はじめに

近年の組込みシステムでは様々な要望に応えるために、ソフトウェアで機能実装がなされており、C 言語で開発を行い、マイコン上に機能を実装することが一般的になっています。組込みシステムを作成するために検討すべき内容も多岐に渡っています。近年議論されることが多いものとして a) 組込みハードウェア/ソフトウェアの分離調達、b) 外部ソフトウェア部品の利用、c)商品バリエーション展開時に同一の機能を異なったマイコンで実装、d)マイコンの EOL(End of Life)問題などを挙げる事ができます。

これらに共通なものは、ソースコードの多くは共通でありながら、マイコンが違う、コンパイラが違う、あるいはコンパイラのオプションが違うことにより、検証作業が発生することです。マイコン、コンパイラ、もしくはコンパイラのオプションを変更した時に、どこまで検証し直すのかは各システムにより大きな差があるとは言えますが、工数を多く使う作業になります。

本稿では、現状の改題を整理し、問題を単純化するために同一 CPU 間での比較は何を実施すればよいかについて検討を実施しました。

†1 IAR システムズ株式会社
IAR Systems K.K.
†2 九州大学
Kyushu University

2. 現状の組込みシステムでの課題

一般的な組込み開発では C 言語によるソースコードを作成し、コンパイラによりターゲットコードに変換します。

この時にマイコン変更およびコンパイラ変更（コンパイラ自体の変更およびコンパイラオプションの変更）した場合に、生成したコードに対して検証を実施する必要があります。1)C 言語のかかえる問題、2)最適化による問題、 3) 非機能制約、4)バグの問題について簡単に説明します。

2.1 C 言語のかかえる問題

C 言語の規格には未規定の動作/未定義の動作/処理系定義の動作という項目が存在します。そのため、ソースコードが同じであってもコンパイラによって異なった結果となるコードが生成される可能性があります。複数マイコン/コンパイラを使用する場合には本質的な問題になります。

C 言語の規格 C90 では、22 の未規定の動作、97 の未定義の動作、76 の処理系定義の動作が規定されています[1]。

たとえば、データ型の char は符号付き/符号なし、負の値を持つ符号付整数の右シフト、などは実装依存となっており結果はマイコンおよび処理系によって変わる可能性があります。この問題により差異を防ぐ方法の１つに MISRA (Motor Industry Software Reliability Association)の定めたコーディング規約 MISRA-C があります。ただ、一般的には MISRA-C を 100%遵守することは難しく、逸脱も含めて検

討することが一般的になっています。逸脱すると何らかの形で、正しい結果になることを保証する必要があります。

2.2 最適化による問題

最適化はコンパイラの実装により何をどのようにやるのかが異なるため、あるコンパイラでは問題ないが、違うコンパイラを使うと期待している動作をしないといった問題が出てきます。

2.3 非機能制約

組込みシステムの場合には、結果としてはメモリやレジスタ値が同じ値になったとしても、時間的な制約、消費電力的な制約など結果（機能以外の項目）についても満たすべき項目があります。

2.4 バグの問題

マイコンやコンパイラにもバグが存在する場合があり、よりケースを複雑にします。C 言語の規格で解釈をすれば正しいソースコードであったとしても、ソースコード変更などでバグを回避するケースなどがあります。

3. バイナリ比較をしたいケース

それでは実際にどうした時にターゲットコードを比較したいケースがあるかについて明らかにします。

3.1 ケース 1: コンパイラのオプションを変更する場合

コンパイラの最適化をデバッグ時とリリース時に変更することが一般的に行われています。一般的に最適化を上げるとデバッグ性は低下します。最適化を速度優先にすると、デバッグ性は極端に低下します。そのため、デバッグ時とリリース時でターゲットコードが異なるために、バイナリレベルでコード比較が出来れば作業効率化が可能となります。

よく発生する問題としては、待ち時間を以下のループで入れた場合に、最適化などによりループの実装が省かれる場合があります。

```
for (i=0; i<100; i++);
```

変数アクセスの順序もコンパイラオプションの変更に組込みシステムに影響を与える場合も多く発生しています。バイナリレベルで差分を比較する技術が確立することで、デバッグ時のコードとリリース時の最適化の有無でコードの正当性が確認出来る事になります。

3.2 ケース 2: ミドルウェアの可搬性

ミドルウェアや RTOS(リアルタイム OS)は、使用するユーザが使用するマイコンおよびコンパイラによって再度コンパイルおよび検証をする必要があります。その工数が大変なために、実際には使用できるマイコンやコンパイラが指定されることが一般的です。こうした工数が何らかの技術で解消されれば、ユーザの利便性の向上が見込まれます。

3.3 ケース 3: ハード・ソフトの分離調達

組込みシステムの複雑化により、ハードウェアとソフトウェアの分離調達がビジネス上すすんでいます。例としては、車載電子システムでは標準規格 AUTOSAR が導入されつつあり、アプリケーション層のソフトウエアコンポーネントは再利用されることになります[2]。こうした背景から、マイコンやコンパイラが変更したときにターゲットコードレベルで比較ができることで分離調達の自由度があがります。

3.4 ケース 4: 商品バリエーションによる複数マイコン

商品バリエーションがある場合に、同一マイコンを使うことが難しいケースでは、たとえば、ローエンドは 16 ビット、ハイエンドは 32 ビットマイコンが使わるケースがあります。そのため同一のソースコードを、それぞれ異なったマイコンおよびコンパイラを利用してターゲットコードに変換します。単純にコンパイラおよびマイコン違いという点には含まれますが、“ そもそも int サイズの違う ” という本質的な差があるために、結果の不整合などが出ることがあります。

4. バイナリ比較をするための問題の整理

4.1 実際に変更する項目

バイナリ比較をする対象を作成する時に、以下の 3 つのパターンを想定しています。

1)マイコン・コンパイラは同一で、コンパイラオプションが異なる場合。

2)マイコンは同一で、コンパイラが異なる場合。コンパイラが異なっている場合にはコンパイラオプションが同じであっても機能が異なる可能性が高いので別と判断します。

3)マイコンが異なっているケースは、コンパイラも含めてすべて異なっていると考えられる。

4.2 どの単位で比較するのか？

ここではソフトウェアについて比較をするので、いくつかのレベルで比較をすることが可能となります。

- 単体関数を実行して比較
- 組合せて実行結果を比較
- システムとしての機能を確認

4.3 今回を検討する項目は

本報告では同一 CPU、同一コンパイラで、オプション違い、

Vol.2015-OS-135 No.11
Vol.2015-EMB-39 No.11
2015/11/24

および、単体関数ごとで比較について検討します(図 1)。

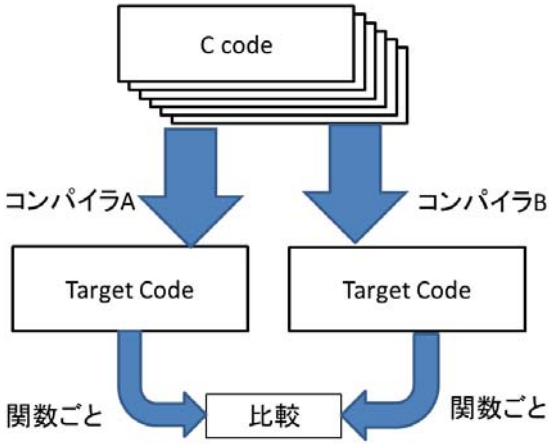

図 1　今回想定しているバイナリ比較

4.4 ケーススタディで調査する内容

バイナリで比較を行って何を持って等しいかを明確にする必要があります。実際のコンパイルされるコードで何を持って同じと判定するかを決定する必要があります。単純には、ソフトウェア見えするレジスタ、メモリが同じなっているという判定方法もあります。後で述べますが関数、変数のアドレスが異なるために単純な比較では現実的には同一と判定できるものが不一致と判定されます。そのためケーススタディで実際のターゲットコードをもとに調査を実施します。

5. 同一 CPU でのバイナリ比較ケーススタディ

同一 CPU のバイナリコードを関数レベルで比較を行うために、ソースコードとコンパイラの動作の点からプログラムは以下の組み合わせで実現されるため以下の項目を検討します。

- ストレートコード
- 分岐
- ループ
- その他

また今回は CPU として ARM 社の Cortex-M3 を題材として調査を行いました。アセンブラも Cortex-M3 の記述になります[3]。

ケース A1：ストレートコードで演算
分岐やループ構造がないストレートコードの関数の例を示します。

```
int func(int a ) {    return a+1;}
```

この関数の場合は機能と配置の 2 点を考慮する必要があります。ここではターゲットコードを表示しても解り難いので逆アセンブラした以下のフォーマットで表示します(図 2)。

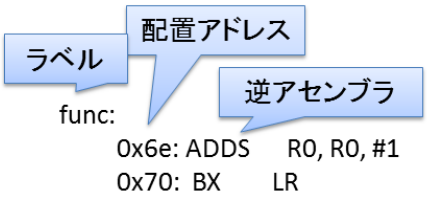

図 2 アセンブラの記述内容

まず、コンパイルオプションの違い、あるいは他のソースコードの違いなどでこの関数自体のコンパイルには差がなくても配置違いが存在します。例としては以下のコードになります。

```
func:
        0x64: ADDS      R0, R0, #1
        0x66: BX        LR
```

以下のように内部で値を他レジスタにコピーする場合も関数の実行結果としては同じと見なすことが出来ます。

```
func:
  0x19ec:   MOV       R1, R0
  0x19ee:   ADDS      R0, R1, #1
  0x19f0:   BX        LR
```

また、R1 と R2 を入れ替えても動作としては同じと見なすことが出来ます。

```
func:
  0x19ec:   MOV       R2, R0
  0x19ee:   ADDS      R0, R2, #1
  0x19f0:   BX        LR
```

しかし、以下のようになると関数の返り値としては一致するが、同じ結果とは言えない状況になります。

```
func:
  0x19fe:   MOV       R5, R0
  0x1a00:   ADDS      R0, R5, #1
  0x1a02:   BX        LR
```

これは、CPU のレジスタの役割が同じではないことに起因します。Cortex-M3 では汎用レジスタが 3 種類 1)スクラッチレジスタ、2)保護レジスタ、3)専用レジスタ に分類されます。スクラッチレジスタは呼びだされた関数で自由に使用してよいレジスタ（呼び出しもとで値を保存する必要がある）で Cortex-M3 では R0~R3 および R12 になります。保護レジスタは使用する場合には呼びだされた関数で保存するレジスタで Cortex-M3 では R4~R11 になります。専用レジスタはプログラムカウンタ(PC/R15)、スタックポインタ(SP/R13)、戻りアドレスを保存するリンクレジスタ（LR/R14）

となります。

先ほどの例のようにデータのコピー(移動)がレジスタ間で行われるだけでなく、CPU ではスタックを使用する可能性があります。そのためスタックも含めて判定をする必要があります。以下の例では、R6 をスタックに保存しその R6 に中間値を代入して演算を実行して関数の終了前にスタックから R6 の値を復帰させています。関数の動作としては同じと見なすことが出来ます。

```
func:
  0x1a14:   PUSH      {R6}
  0x1a16:   MOV       R6, R0
  0x1a18:   ADDS      R0, R6, #1
  0x1a1a:   POP       {R6}
  0x1a1c:   BX        LR
```

ケース A2：ストレートコードでメモリアクセス
グローバルもしくはスタティック変数へのアクセスを含む関数を示します。
void func2(int a) { gc = a++;}
このコードでは gc がグローバル変数となります。
生成されたコードは gc へのアクセスが PC 相対で作成されています。

```
func2:
   0x94: LDR.N    R1, [PC, #0x4]; [0x9c] gc
   0x96: STR      R0, [R1]
   0x98: ADDS     R0, R0, #1
   0x9a: BX       LR
   0x9c: DC32     gc
```

変数の配置が換わるとこうしたアドレスも違うことになります。そのためターゲットコードだけで比較するとアドレスに関する部分の対応付けを見ることが難しくなるため、変数に関する情報を利用することで精度の高い比較を実施できることになります。変数情報はマップファイルに含まれることで比較時に利用できます。

ケース A3：ストレートコードで関数呼出
以下のような関数を見てみます。
int goo(int a) { return func(a); }
以下のコードが出力されています。

```
fx:
  0x18a2:  B.N      func
```

この結果は逆アセンブラした結果ですが、アセンブラのフォーマットを見えると B 命令は以下の即値 8 ビットを持つ命令であり、PC+即値 8 ビットでジャンプ先が決定します。

B<条件> 即値 8 ビット

バイナリ比較をする場合には関数アドレスの情報も重要となりますが、同様に変数アドレスの情報も必要となります。関数情報はマップファイルから取得することが出来ます。

ケース B1: 分岐
バイナリレベルで比較をするときの if 文やケース文での代表的な例を確認します。
if (a) { x1 = 100; } else { x1 = 200; }
このコードを EWARM7.40.4 で最適化を高と低の 2 つでコード生成をすると以下のようになります。

```
0x18a8: CMP      R0, #0
0x18aa: LDR.N    R1, [PC, #0xf8]   ; [0x19a4] x1
0x18ac: ITE      NE
0x18ae: MOVNE    R0, #100          ; 0x64
0x18b0: MOVEQ    R0, #200          ; 0xc8
0x18b2: STR      R0, [R1]
```

上記の最適化高で実行した場合にはジャンプを使用せずに、条件実行の命令(MOVNE, MOVEQ)にて分岐処理を実現しています。

```
0x18ae: CMP      R0, #0
0x18b0: BEQ.N    0x18ba
0x18b2: MOVS     R0, #100          ; 0x64
0x18b4: LDR.N    R1, [PC, #0x148] ; [0x1a00] x1
0x18b6: STR      R0, [R1]
0x18b8: B.N      0x18c0
0x18ba: MOVS     R0, #200          ; 0xc8
0x18bc: LDR.N    R1, [PC, #0x140] ; [0x1a00] x1
0x18be: STR      R0, [R1]
```

この最適化低で実行した場合には比較と条件ジャンプを利用して実現しています。分岐などの構造は一致しない点を考慮する必要があります。

ケース B2: switch 文
switch 文に関しても最適化オプションにより出力されるコードが変わります。以下の switch 文を変えてコンパイルを実施した結果を示します。

```
int f4(int a ) {
  switch(a) {
  case 0:    return 0;
  default:   return 100;
  }
}
```

最適化が低い場合には比較的ソースコードに忠実な形でコードが生成されています。

```
f4:
 0x18a2: CMP          R0, #0
 0x18a4: BNE.N        0x18aa
 0x18a6: MOVS         R0, #0
 0x18a8: B.N          0x18ac
 0x18aa: MOVS         R0, #100        0x64
 0x18ac: BX           LR
```

最適化を高にすると以下のようなコードが生成されました。ここで CBZ はゼロとレジスタを比較してジャンプする命令ですが、引数がゼロの場合には 0 への代入はこの関数では実行しないコードとなっています。レジスタ R0 が引数として渡され、内容が 0 の場合にはそのままリターンする効率のよいコードとなります。

```
f4:
 0x18a2:   CBZ          R0, 0x18a6
 0x18a4:   MOVS         R0, #100     ; 0x64
 0x18a6:   BX           LR
```

ケース B3: 最適化により分岐が無くなる例
ソースコードには分岐があった場合でも、ターゲットコードでは分岐が無くなっている場合もあります。

```
int    gg(unsigned short a, unsigned short b) {
   if ( c == ( a | ~b) )     return 0;
   else                      return 1;
}
```

この例では、以下のようなターゲットコードが生成されます。コンパイラによっては分岐構造が残ることもありますが、結果は同じケースがあります。

```
gg:
 0x18c2:   MOVS         R0, #1
 0x18c4:   BX           LR
```

分岐やループ記述は実現できる構造が複数方法あり、構造が一致しない点を考慮する必要があるのはケーススタディ B1,B2 と同様です。

ケース C1:ループ
for 文に最適化を変更してコードを生成してみます。

```
for (i=0; i<100; i++ ) {       x2 += i;    }
```

```
0x96: MOVS          R0, #0
0x98: B.N           0xa6
0x9a: LDR.N         R1, [PC, #0x1c]
0x9c: LDR           R1, [R1]
0x9e: ADDS          R1, R0, R1
0xa0: LDR.N         R2, [PC, #0x14]
0xa2: STR           R1, [R2]
0xa4: ADDS          R0, R0, #1
0xa6: CMP           R0, #100
0xa8: BLT.N         0x9a
```

デバッグ性を高めるためにはソースコードに近い構造をもっていますが、最適化レベルを上げると以下のコードが生成されています。

```
0xb6: MOVS          R1, #0
0xb8: LDR           R2, [R0, #0x4]
0xba: ADDS          R2, R1, R2
0xbc: ADDS          R1, R1, #1
0xbe: CMP           R1, #100
0xc0: STR           R2, [R0, #0x4]
0xc2: BLT.N         0xb8
```

最適化有り無しで分岐命令数も異なるため、比較する上で何を確認するかを明確にする必要があります。

ケース C2:ループアンローリング
ループに対するコンパイラの最適化技術にループアンローリングがあります。

```
#define MAX (4)
int dat[MAX];
int loop1( ) {
   int i;
   for ( i=0; i< MAX; i++ )    dat[i]=i;
}
```

ループアンローリングをしない場合

```
loop1:
 0x60: MOVS          R1, #0
 0x62: MOVS          R0, R1
 0x64: CMP           R0, #4
 0x66: BGE.N         0x72
 0x68: LDR.N         R1, [PC, #0x8]
 0x6a: STR.W         R0, [R1, R0, LSL #2]
 0x6e: ADDS          R0, R0, #1
 0x70: B.N           0x64
 0x72: BX            LR
```

ループアンローリングをする場合

```
loop1:
 0x18ae: LDR.N       R0, [PC, #0xe0]
 0x18b0: MOVS        R1, #0
 0x18b2: STR         R1, [R0]
 0x18b4: MOVS        R1, #1
 0x18b6: STR         R1, [R0, #0x4]
 0x18b8: MOVS        R1, #2
 0x18ba: STR         R1, [R0, #0x8]
 0x18bc: MOVS        R1, #3
 0x18be: STR         R1, [R0, #0xc]
```

```
0x18c0: BX          LR
```

最適化によりループ構造が無くなってしまうために、比較する時にこうした点を考慮する必要があります。

ケース D1: 不要コードの削除
以下のようなループはコンパイラによって削除されることがあります。

```
for (i=0; i<100; i++);
```

タイミングなどを制御している場合もあり単純に問題が無いかを判断することが難しいケースとなります。

ケース D2:変数アクセスの最適化
以下のような変数アクセスもコンパイラの最適化により出力が変わる可能性があります。

```
x01 = 100;
x01 = 200;
```

最適化が低い場合には 2 つのメモリアクセスが出力されます。

```
0x19a0: MOVS        R0, #100
0x19a2: LDR.N       R1, [PC, #0x54]
0x19a4: STR         R0, [R1]
0x19a6: MOVS        R0, #200
0x19a8: LDR.N       R1, [PC, #0x4c]
0x19aa: STR         R0, [R1]
```

最適化が高い場合には、最後の出力だけがコードとして実装されています。

```
0x198e: LDR.N       R1, [PC, #0x10]
0x1990: MOVS        R0, #200
0x1992: STR         R0, [R1]
```

組込みシステムでは､周辺回路を制御している場合も多く、記述した変数アクセスが外部に対して出力されることが重要な場合があります。また、DMA でハードウェアによりメモリ値が変更されることもあり、こうした点はチェックする上で重要となります。
そのためメモリ領域の最終結果だけをもって一致とするのか、もしくはアクセスがすべて同じ事をもって一致とするのかを情報として与える必要があります。メモリもしくは I/O 領域に記述したアクセスをかならず生成する場合には、通常 volatile というキーワードを使用しプログラム上で制御することが一般的です。マップファイルでは通常変数および関数に関する情報が取得できますが、volatile 属性については別途情報が必要となります。

ケース D3: 演算子の最適化
最適化ではCで記述した演算子と違う演算の命令に置き換えられてしまうことがあります。以下の記述では、引数の値を 2 倍する関数ですが、シフト命令で実現されることが多いです。

```
a*2;
```

ここで、LSLS が Logical Shift Left 命令になります。

```
0x66: LSLS          R0, R0, #1
```

演算子がそのまま残った場合には以下の記述などが生成される可能性があります。

```
0x90: MOVS          R0, #2
0x9e: MULS          R0, R1, R0
```

除算でも同様にシフト命令に変わることがあります。こうした演算子が異なるケースも想定する必要があります。

6. バイナリ比較手法の検討

6.1 バイナリコードを比較する方法

バイナリコードを比較する方法は大きく分けると、1)実行をして結果を確認する方法(実行型と呼ぶ)、2）実行することなく確認する(非実行型)の 2 つに分類することが出来ます。
実行型は一般的にはテストという形で実行しており、バイナリコードを直接比較する手法ではありませんが、結果で確認する方法になります。本稿では、非実行型で比較する方法について検討を行います。

6.2 H1:命令の一致

比較で簡単なものは、命令レベルで完全に一致する場合です。最適化レベルを変えた場合でも同一の命令列が生成される場合もありますので、こうしたケースは命令レベルの一致で判定可能です。2 つの生成されたコードだけで比較が可能です。この時に、プログラムアドレスは変更される可能性があります。

6.3 H2:命令＋アクセスデータ＋ジャンプ先の一致

ケース A2 や A3 に対応するためには､命令列が同じというだけでなく、アクセスする変数やジャンプする関数が同じであれば同じと判定することが出来ます。そのため生成されたコードに加えてマップファイルに有る情報をもとに比較が可能となります。プログラムの配置アドレス、変数アドレス、ジャンプ先アドレスは異なっている可能性があります。

6.4 H3:データフローと ABI 制約を考慮して比較

H1 および H2 は CPU のレジスタおよびメモリが同じ値になることを前提とした一致になります。しかし、こうした厳密な一致をしないにもかかわらず実際には同じと判定できる場合があります。そのために比較するための条件を検討します。C 言語ではスタックを利用することが多く、関数の実行が終了するとその関数が使用した領域も破棄され

次の関数が使用する事が出来ます。
ここでは関数のバイナリ比較について以下のように整理することが出来ます。

- 入力
  - 関数の引数（レジスタおよびスタック渡し）
  - ヒープ領域のメモリ領域
  - CPU の保存レジスタ
- 出力
  - 関数の返り値
  - ヒープ領域のメモリ領域
  - CPU の保存レジスタ
- 比較
  - 同一の入力に対して、出力が同一であること

この時のヒープ領域には、グローバル/スタティック変数および周辺レジスタなどが含まれます。関数の引数や返り値については、通常 ABI(Application Binary Interface)と呼ばれるものになります。
これでケース A1 の例をデータフローグラフで記載してみると、図 3 のように表現できます。このグラフを比較することで一致の判定が可能となります。CPU のスクラッチレジスタを比較する必要が無ければ、図 3 の左と真ん中のグラフは一致することになります。一般的には C 言語でデータフローを作成することがありますが、今回はバイナリ比較をするためにアセンブラレベルでのデータフローになります。

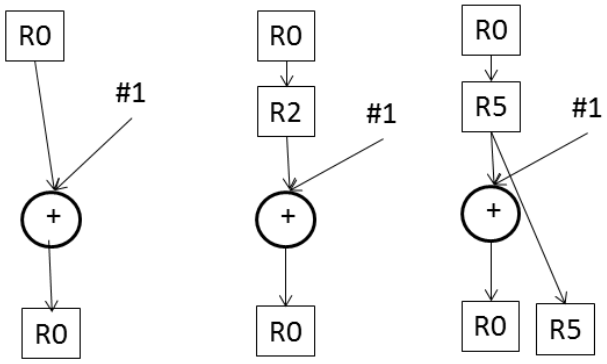

図 3 データフローによる表現

6.5 H4:コントロールデータフローと ABI 制約を考慮して比較

データフローに加えてコントロールフローを入れることで、比較が可能となります(図 4)。単純にコントロールフローを作るだけでなく、本当に条件が一致しているかまで判定する必要があります。

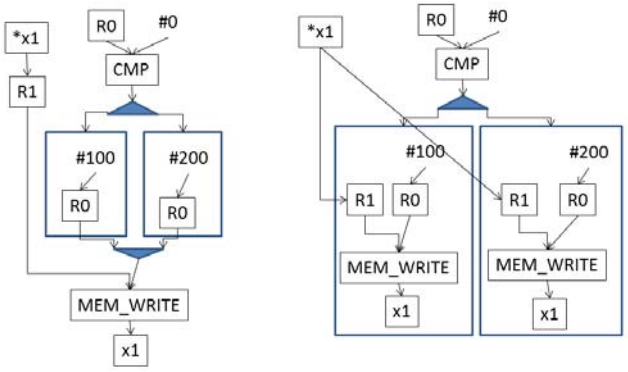

図 4 コントロールデータフローでの表現

こうした点では、ハードウェア設計で用いられる等価性検証といった技術が利用できる可能性があります[4,5]。ケース B3 のような条件判定が不要となるケースでは、こうした論理的な構造を評価するための仕組みが必須になります。

6.6 H5: データフローでの演算子考慮

データフロー（コントロールデータフロー）だけではケース D3 の演算子の最適化が発生する/しない場合では同一判定をすることが出来ません。こうした部分については、違う演算子で生成された部分についての等価性を判定する必要があります。D4 のケースをデータフローで表すと図 5 となります。この四角で囲った部分が等価であることを示す必要があります。

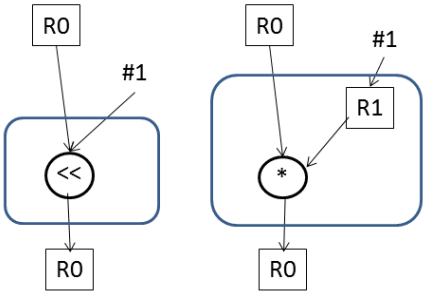

図 5 演算子最適化された場合への対応

6.7 H6:ループ構造をもつケースの検討

ループ構造を持つ場合の比較は、もうすこし検討が必要となります。ケーススタディ C1 にあるようにループ構造をもつソフトウェアに関しては、コード列の比較やデータフローグラフでの単純に比較することが出来ないため、H1,H2,H3,H4 で検討した内容では比較は困難です。
先ほどのケーススタディのループはシンプルなものですが、出力されたコードだけからこのコードの等価性を比較することは困難です。
ハードウェアの等価性検証などでは、等価性ポイントを利用することで、問題を解きやすくする方法などが用いられています。ソフトウェアのバイナリコードレベルでの精度よく判定をするために、ターゲットコード、マップファイ

ルの他に利用できるものを検討する必要があります。バイナリコード比較では、もととなるソースコードおよびデバッグ情報について利用できるものとして考えられます。デバッグ情報ではソースコードの位置情報が与えられるため、バイナリ比較時に利用することが出来ます。こうした情報を入力として使用してバイナリ比較を実施する手法が必要です。

6.8 その他検討が必要な点

今回は Cortex-M3 の CPU を例に検討を行いました。最近の CPU で使用されている構造でいくつか検討出来ていない点があり、バイナリレベルでの同一性を比較する上で必要となる項目をここで整理します。

- SIMD 命令の有無
  SIMD 命令とは Single Instruction Multiple Data の略で 1 つの命令で複数のデータ処理を行う命令となります。組込みシステムで使用される CPU としては Cortex-A があり、その中で NEON 命令が使用されています。NEON 命令の使用有無はコンパイラのオプションで規定されるため、差分として発生する可能性があります。
- ハードウェア演算使用の有無
  浮動小数点演算などではハードウェア演算器を用意している CPU も数多くあります。ハードウェアを利用せずにソフトウェアライブラリを利用することがあります。
- ロードストアユニットとキャッシュ構造
  高機能な CPU ではメモリアクセスをするためのロードストアユニットが実行時にメモリアクセスの最適化を実行しているケースがあります。たとえば、連続したバイトアクセスをワードアクセスにして実行することが行われています。また、クロック速度が高い CPU ではキュッシュメモリが採用されます。キャッシュの設定によりメモリアクセスが毎回外部に出る場合やキャッシュアウトする時にだけ出力される場合などがあります。

7. まとめ

現状、コンパイラおよびそのオプション変更時の動作確認では、一般的に実行型のテストが利用されています。
本稿では、非実行型で今回は同一 CPU に対するバイナリコード比較を実施するためのケーススタディを行なった。単純に CPU レジスタや周辺レジスタ/メモリの値での等価だけではなく ABI を考慮した等価とすることで、機能的な等価および非等価を判断できることが分かった。
今回は機能的な点を中心に検討を行なったが、非機能的な面での比較も重要となるので、6.8 の内容を含めてさらに検討する予定です。さらに、ABI 制約を使うことで異なった CPU 間でのバイナリ比較についても検討を進める予定です。

参考文献

1) MISRA-C 研究会: 組込み開発者におくる MISRA‐C:2004―C 言語利用の高信頼化ガイド, 日本規格協会(2006)
2) AUTOSAR で変わる車載ソフトウェア開発 (1/4), http://ednjapan.com/edn/articles/0910/01/news123.html
3) Cortex-M3 テクニカルリファレンス マニュアル, http://infocenter.arm.com/help/index.jsp
4) ウィキペディア形式等価判定, https://ja.wikipedia.org/wiki/形式等価判定
5) William K. Lam, "Hardware Design Verification: Chapter 8 Decision Diagrams, Equivalence Checking, and Symbolic Simulation", Prentice Hall, 2005.