計算機アーキテクチャ　83－29
（１９９０．７．１８）

# ＶＬＩＷ型計算機ＫＩＤＯＣＨ用 Ｃコンパイラの特徴

安倍　正人 +、本郷　哲 *、中鉢　憲賢 *、城戸　健一 ++

+ 東北大学大型計算機センター
* 東北大学工学部
++千葉工業大学

ＶＬＩＷ型計算機を効果的に使うためには高度な最適化処理機能を持つ高級言語が不可欠である。本文では、開発中のＶＬＩＷ型計算機ＫＩＤＯＣＨ　ＩＶ用に開発中のＣコンパイラの特徴をハードウェアとからめて述べる。すなわち、メモリの入力ポート数が２本、演算器として、整数データ用ＡＬＵが２個、整数データ用ＭＰＹが１個、浮動小数点データ用あるいは整数データ用の演算器として用いることができるＦＰＵを２個用意し、それぞれ独立に操作できるというハードウェアの特徴を高級言語Ｃのプログラムを効率良く実行できるように使うための手法について述べる。

# Performance of the C compiler for VLIW Computer KIDOCH

Masato ABE +, Satoshi HONGO *,
Noriyoshi CHUBACHI *, and Ken'iti KIDO ++

+ Computer Center, Tohoku University
* Faculty of Engineering, Tohoku University
++Chiba Institute of Technology

This paper describes the performance of the C compiler for the VLIW type computer KIDOCH IV. The C compiler effectively supports multiple operation units: two data memory units, two ALUs for integer data, one multiplier for integer data, and two ALU/MPY/DIV/SQRTs both for integer data and for floating point data. In this paper offered are two example C programs, one is "if" statement, and the other is "for" statement to show the effectiveness of the C compiler.

## 1. まえがき

我々は、ディジタル信号処理技術の音響信号処理への応用に関する研究を行っている。例えば、多数（１００個程度）のセンサを用い、各センサ出力のＦＦＴ、音源位置の推定、音源波形の推定［１］を実時間で行なうためには、合計で、５００ＭＦＬＯＰＳの演算スピードが要求される。このうち、ＦＦＴに関しては、処理が単純なので、センサそれぞれに高速なＤＳＰチップを用いれば事足りるが、音源位置の推定と音源波形の推定は処理が複雑で、データ量も多いので、ＤＳＰによる処理は困難である。そこで、このような複雑な処理を高速に行なうために、我々はＶＬＩＷ型計算機ＫＩＤＯＣＨ　ＩＶと高級言語Ｃを開発中である。

本文では、開発中のＶＬＩＷ型計算機ＫＩＤＯＣＨ　ＩＶ用に最適化したＣコンパイラの特徴をハードウェアとからめて述べる。

## 2. ＫＩＤＯＣＨ　ＩＶのハードウェア構成

図１に本システムの全体の構成を示す［２］，［３］。ホスト計算機はＵＮＩＸマシンである。このホスト計算機とＫＩＤＯＣＨ　ＩＶはＶＭＥバスを通じて結合される。

ＫＩＤＯＣＨ　ＩＶは独立に稼働する１３個の装置からなり、約Ａ３判大のプリント基板３枚から構成される。

### 2.1. データ用内部バスの構成と入出力レジスタの実装によるバスネックの解消

ＫＩＤＯＣＨ　ＩＶもＫＩＤＯＣＨ　ＩＩＩ［４］と同様にデータ用の内部バスは３本用意してある。

ＫＩＤＯＣＨのように各装置間のデータ転送をすべて内部バスで行うアーキテクチャでは、バスネックが問題となることがある。すなわち、ＫＩＤＯＣＨ　ＩＶでは入出力できる装置が１０あるのに対し、バスの数が３本しかないことが問題となる。

ＫＩＤＯＣＨ　ＩＶでは図２に示すように各演算器の入力部に２つのレジスタを設けることにより上述の問題を解決する。すなわち、ＫＩＤＯＣＨ　ＩＩＩのように入力部にレジスタが１つしかないと、後で再び同じデータを使うとわかっていても、その前にその演算器を使わなければならないときには、そのデータを別の領域に待避しておき、必要になった時点で、再びデータを転送しなければならないが、レジスタを２つ用意しておくことにより、このような問題がかなり緩和される。

さらに、このようにレジスタを介してバッファリングを行なうことにより、データの転送をすべて同期式で行なうことができ、サイクルタイム４０ｎｓでの高速動作が可能となる。

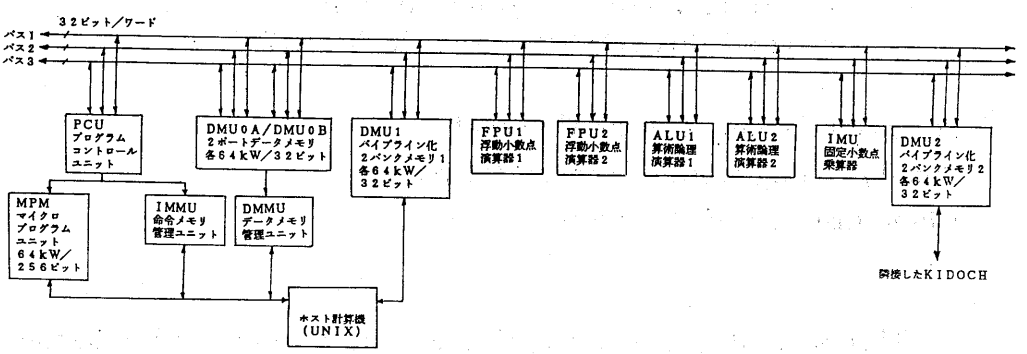


図１　本システム全体の構成

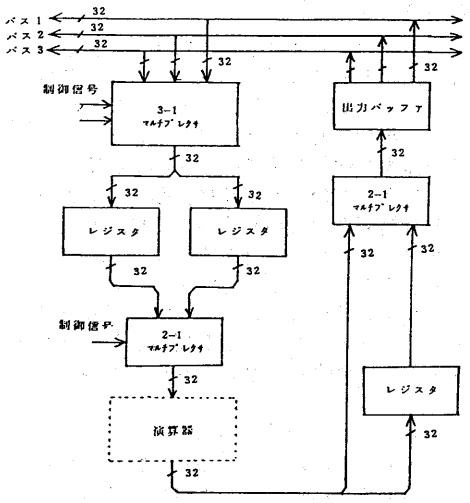


図2　入出力レジスタの実装による
バスネックの解消

### 2.2. 算術論理ユニット（ＡＬＵ）

ＡＬＵ１，ＡＬＵ２はそれぞれＴＩ社の３２ビットプロセッサＳＮ７４ＡＣＴ８８３２を用いて構成している。演算結果のコンディションにより条件ＪＵＭＰを行なうことができる。ＡＬＵ１およびＡＬＵ２は固定小数点のデータの演算（加減算および論理演算）のみでなくデータメモリのアドレス計算にも用いられる。

### 2.3. 固定小数点乗算ユニット（ＩＭＵ）

ＩＭＵはＴＩ社の３２ビットプロセッサＳＮ７４ＡＣＴ８８３６を用いて構成されている。本ユニットは主に配列のアドレスを計算するのに用いられる。

### 2.4. 浮動小数点演算ユニット（ＦＰＵ）

ＦＰＵ１，ＦＰＵ２はそれぞれＴＩ社の３２ビットプロセッサＳＮ７４ＡＣＴ８８４７を用いて構成している。このプロセッサは単精度（３２ビット）、ならびに倍精度（６４ビット）浮動小数点および３２ビット固定小数点データの四則演算と平方根、論理演算および型変換を行なうことができる。ただし、内部はパイプライン化されていて、単精度浮動小数点および固定小数点の加減算と乗算は３ステップ、倍精度の加減算と乗算は４ステップ、除算は単精度が８ステップ、倍精度が１４ステップ、固定小数点が１６ステップで実行できる。一方、平方根は、単精度が１１ステップ、倍精度が１７ステップ、固定小数点が２０ステップで実行できる。さらに、演算結果のコンディションにより条件ＪＵＭＰを行なうことができる。

### 2.5. 制御装置（ＭＰＭ，ＰＣＵ，ＩＭＭＵ）

ＭＰＭはマイクロプログラムメモリで、６４ｋＷ×２５６ビットのＲＡＭで構成されており、１ｋＷずつの６４個のページに分けて管理される。ＰＣＵはプログラムコントロールユニットで、サイクルタイム４０ｎｓで動作させるために、４ビットのカウンタ７４ＡＳ１６３を８個用いて構成している。ＩＭＭＵは命令メモリ管理ユニットで、図３に示すように各ページに入っているオブジェクト（機械語）が有効かどうかを示すフラッグと実際のアドレス（３２ビット）を保持しており、ダイレクトマッピング方式のページングを行なう。

サイクルタイム４０ｎｓの高速動作をさせるために、様々な工夫がなされている。すなわち、

（１）ＭＰＭの前段（ＰＣＵの出力）と後段にはパイプラインレジスタがあり、ＭＰＭの遅延を吸収している。しかし、このために、命令の存在するアドレスを指定するステップと実際にその命令を実行するステップが１だけずれるので、これを救うために、Ｃコンパイラで自動的に遅延分岐を行なうように設定している。

（２）さらに、分岐する場合には、各装置に供給される同期クロックを１ステップ分だけ止めて、その間に分岐すべきアドレスが実際にＭＰＭ中に存在するかを判断するようにハードウェアを構成した。この機能がないと、サイクルタイムを一様に約１００ｎｓにしなければならないので、ロスが大きい。

（３）ＩＭＭＵは１ページ１ｋＷ単位でＭＰＭを管理しているので、次に行なうべき命令が次のページに移る場合に、次のページがＭＰＭ中に存在するかどうかを上述の（２）と同様に１ステップ分のロスのみで判断する。

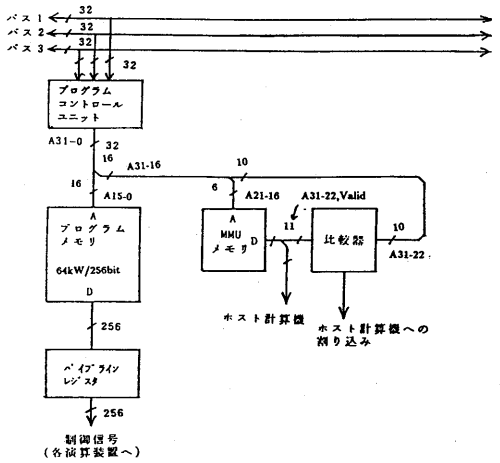


図3　命令メモリ管理ユニットの構成

はじめに、ホスト計算機は、ＫＩＤＯＣＨＩＶのオブジェクト（機械語）をプログラムメモリに転送し、さらに、ＩＭＭＵにそのアドレスを送って、オブジェクトが有効であることを示すフラッグを立てる。

ＫＩＤＯＣＨが動作を始め、アドレス（３２ビット）がプログラムメモリに送られると同時に、そのアドレスはＩＭＭＵにも送られ、アドレスＡ３１－Ａ２２の値とアドレスＡ２１－Ａ１６が指し示すＩＭＭＵの内容とが比較される。両者が一致し、かつ有効フラッグが立っていれば、そのまま次のステップに移るが、そうでない場合には、ＫＩＤＯＣＨは動作を停止し、ホスト計算機に割り込みがかかる。そのとき、ホスト計算機は割り込み原因を解析することにより、プログラムメモリにおけるＩＭＭＵエラーであることがわかるので、そのページの部分だけ、ＫＩＤＯＣＨのプログラムメモリ及び対応するＩＭＭＵの部分を書換える。そして、ＫＩＤＯＣＨを再起動させる。このようにして、ＫＩＤＯＣＨは最大４Ｇステップのプログラムを動かすことができる。

これと同様のメモリ管理機構ＤＭＭＵ０Ａ，ＤＭＭＵ０ＢがそれぞれＤＭＵ０Ａ及びＤＭＵ０Ｂに対して備えてあり、それぞれ最大４ＧＷ／３２ビットのデータを扱うことができる。

### 2.6. デュアルポートメモリ（ＤＭＵ０Ａ，ＤＭＵ０Ｂ）

ＫＩＤＯＣＨのようなＶＬＩＷ型計算機では、演算器の数に見合ったデータの供給能力を必要とするため、高速化のためにはデータバスを複数用意するだけでなく、データメモリ自体も複数用意する必要がある。さらに、Ｃプログラムのような高級言語においては、他の計算機との互換性を考えなければならないため、マシンに依存するようなプログラムはできるだけ書かない方が望ましい。そのため、ＫＩＤＯＣＨのクロスＣコンパイラにおいても、メモリユニットが複数あることをプログラマに意識させないようにしている。

### 2.7. ループカウンタ（ＬＣ）

Ｆｏｒ文等のループにおける制御変数は１づつ増加あるいは減少する場合が多い、そこで、ＡＬＵの使用効率を上げるために、ループカウンタ（ＬＣ）を用意した。また、ＶＬＩＷ型計算機の効率を高めるために、良く用いられているループ展開（Ｌｏｏｐ　Ｕｎｒｏｌｌｉｎｇ）とＬＣによる条件分岐を組み合わせることにより、次節で述べるように、より効率の良いコードを生成することができる。

## 3. ＶＬＩＷ型計算機用Ｃコンパイラ

ＶＬＩＷ型計算機用のコンパイラ技術［５］としては、トレーススケジューリング法が有名であるが、プログラムによっては弊害も多いので、我々はより一般的な手法のみを用いてコンパイラを開発している。すなわち、基本ブロックのサイズを大きくすることで、効率を高めている。

### 3.1. ｉｆ文の最適化

```
main()
{
    int i,a[10],b;

    if(a[i]<0)
        b -= a[i];
    else
        b += a[i];
}
```

図4　ｉｆ文最適化チェック用
サンプルＣプログラム

```
#   8              if(a[i]<0)
        lw         $3,  76($sp)
        mul        $3,  $3,  4
        addu       $2,  $sp,  $3
        lw         $2,  36($2)
        bge        $2,  0,  $32
        .loc       2 9
#   9                          b -= a[i];
        lw         $14,  32($sp)
        subu       $15,  $14,  $2
        sw         $15,  32($sp)
        b          $33
$32:
        .loc       2 11
#  10              else
#  11                         b += a[i];
        lw         $24,  32($sp)
        addu       $25,  $2,  $24
        sw         $25,  32($sp)
$33:
```

図５ａ　ｉｆ文最適化の結果
ＭＩＰＳ　Ｒ２０００コンパイラ

図４のプログラムにおいて、ａ［ｉ］とｂのアドレス計算およびそれぞれの値のロードは共通なので、一回しか計算する必要はない。そこで、ＫＩＤＯＣＨコンパイラでは、図５に示すように、それらを条件分岐の前に移動し、一回だけ計算している。また、基本ブロックの長さを長くするために、ｉｆ文におけるａ［ｉ］と０の比較、ｂとａ［ｉ］の減算およびｅｌｓｅ文におけるｂとａ［ｉ］の加算を全て条件分岐の前に移動している。

```
 1( LOAD  ICON_ 12                    T_  1 )
 2(  ADD     T_  1  REG1_ 15          T_  2 )
 3( LOAD  REG1_ 15                    T_  3 )
 4( LOAD     T_  3                    T_  4 )
 5(  ADD     T_  2     T_  4          T_  5 )
 6( LOAD     T_  5                    T_  6 )
 7( LOAD  ICON_  0                    T_  7 )
 8(   LT     T_  6     T_  7          T_  8 )
 9( LOAD  ICON_ 13                    T_  9 )
10(  ADD     T_  9  REG1_ 15          T_ 10 )
11( LOAD     T_ 10                    T_ 11 )
12(  SUB     T_ 11     T_  6          T_ 12 )
13(  ADD     T_ 11     T_  6          T_ 13 )
14(   JF     T_  8                 LABL_  0 )
15( SAVE     T_ 10     T_ 12                )
16(  JMP  LABL_  1                          )
17( LABL                           LABL_  0 )
18( SAVE     T_ 10     T_ 13                )
19( LABL                           LABL_  1 )
```

図５ｂ　ｉｆ文最適化の結果
ＫＩＤＯＣＨコンパイラ

### 3.2. ループ文の最適化

ループに無関係な計算は外に出すというグローバル最適化を行なった後のループ展開のことを考える。例えば、図６のプログラムを実行させる場合、ループの中には乗算が一回しかないので、たとえ演算器が複数存在しても、このままでは一つの演算器しか使われないことになる。また、基本ブロックの長さが短いため、ＶＬＩＷ型計算機の特徴が活かせない。そこで、ループ展開の手法を用いることにした。この結果を図７に示す。この例では一つのループで実質的に２回分の仕事をするように展開してある。ただし、ループの回数がコンパイル時点ではわからない場合ことも考慮して、条件分岐は２系統用意してある。しかし、この２つの分岐先はコンパイル時点でわかり、かつＰＣＵ（プログラムコントロールユニット）の分岐先アドレス保持部にも

```
float a[10],b[10],c[10];

main()
{
   int i;

   for(i=0;i<10;i++)
      c[i]=a[i]*b[i];
}
```

図６　ｆｏｒ文最適化チェック用
サンプルＣプログラム

```
#  10            for(i=0;i<10;i++)
        move     $6,  $0
        la       $3,  c
        la       $4,  a
        la       $2,  b
        la       $5,  b+40
$32:
        .loc     2 11
#  11                          c[i]=a[i]*b[i];
        .noalias               $4, $3
        .noalias               $2, $3
        .noalias               $2, $4
        l.s      $f4,  0($4)
        cvt.d.s  $f6,  $f4
        l.s      $f8,  0($2)
        cvt.d.s  $f10,  $f8
        mul.d    $f16,  $f6,  $f10
        cvt.s.d  $f18,  $f16
        s.s      $f18,  0($3)
        addu     $3,  $3,  4
        addu     $4,  $4,  4
        addu     $2,  $2,  4
        bltu     $2,  $5,  $32
```

図７ａ　ｆｏｒ文最適化の結果
ＭＩＰＳ　Ｒ２０００コンパイラ

他の演算器における入力部と同じように、レジスタを２つ用意してあるので、弊害はない。

```
 1( LOAD  REG1_ 15                    T_  1 ) Load &i
 2( LOAD  ICON_  0                    T_  2 ) Load #0
 3( SAVE     T_  1      T_  2                ) Save #0 to &i
 4( LABL                              LABL_  0 ) Label#0
 5( LOAD     T_  1                       T_  5 ) Load i
 6( LOAD  XADR_  0                       T_  6 ) Load &a[0]
 7(  ADD     T_  6      T_  5            T_  7 ) Calculate &a[i]
 8( LOAD     T_  7                       T_  8 ) Load a[i]
 9( LOAD  XADR_  1                       T_  9 ) Load &b[0]
10(  ADD     T_  9      T_  5            T_ 10 ) Calculate &b[i]
11( LOAD     T_ 10                       T_ 11 ) Load b[i]
12(  MPY     T_  8      T_ 11            T_ 12 ) a[i]*b[i]
13( LOAD  XADR_  2                       T_ 13 ) Load &c[0]
14(  ADD     T_ 13      T_  5            T_ 14 ) Calculate &c[i]
15( SAVE     T_ 14      T_ 12                ) Save T12
16(  INC     T_  5                       T_ 16 ) i+1
17( SAVE     T_  1      T_ 16                ) Save i+1
18( LOAD  ICON_ 21                       T_ 18 ) Load #10
19(   LT     T_ 16      T_ 18            T_ 19 ) Compare i+1
20(   JF     T_ 19                    LABL_  1 ) Jmp to Label#1
21( LOAD  XADR_  0                       T_ 21 ) Load &a[0]
22(  ADD     T_ 21      T_ 16            T_ 22 ) Calculate &a[i+1]
23( LOAD     T_ 22                       T_ 23 ) Load a[i+1]
24( LOAD  XADR_  1                       T_ 24 ) Load &b[0]
25(  ADD     T_ 24      T_ 16            T_ 25 ) Calculate &b[i+1]
26( LOAD     T_ 25                       T_ 26 ) Load b[i+1]
27(  MPY     T_ 23      T_ 26            T_ 27 ) a[i+1]*b[i+1]
28( LOAD  XADR_  2                       T_ 28 ) Load &c[0]
29(  ADD     T_ 28      T_ 16            T_ 29 ) Calculate
30( SAVE     T_ 29      T_ 27                ) Save T27
31(  INC     T_ 16                       T_ 31 ) i+2
32( SAVE     T_  1      T_ 31                ) Save i+2
33( LOAD  ICON_ 21                       T_ 33 ) Load #10
34(   LT     T_ 31      T_ 33            T_ 34 ) Compare i+2
35(   JT     T_ 34                    LABL_  0 ) Jmp to Label#0
36( LABL                              LABL_  1 ) Label#1
```

図７ｂ　ｆｏｒ文最適化の結果
ＫＩＤＯＣＨコンパイラ

```
 1( LOAD  REG1_ 15                    T_  1 ) Load &i
 2( LOAD  ICON_  0                    T_  2 ) Load #0
 3( SAVE     T_  1      T_  2                ) Save #0 to &i
 4( LOAD  ICON_ 21                       T_  4 ) Load #10
 5( SAVE    LOOPC      T_  4                ) Save #10 to Loop Counter
 6( LABL                              LABL_  0 ) Label#0
 7( LOAD     T_  1                       T_  7 ) Load i
 8( LOAD  XADR_  0                       T_  8 ) Load &a[0]
 9(  ADD     T_  8      T_  7            T_  9 ) Calculate &a[i]
10( LOAD     T_  9                       T_ 10 ) Load a[i]
11( LOAD  XADR_  1                       T_ 11 ) Load &b[0]
12(  ADD     T_ 11      T_  7            T_ 12 ) Calculate &b[i]
13( LOAD     T_ 12                       T_ 13 ) Load b[i]
14(  MPY     T_ 10      T_ 13            T_ 14 ) a[i]*b[i]
15( LOAD  XADR_  2                       T_ 15 ) Load &c[0]
16(  ADD     T_ 15      T_  7            T_ 16 ) Calculate &c[i]
17( SAVE     T_ 16      T_ 14                ) Save T14 to &c[i]
18(  INC     T_  7                       T_ 18 ) i+1
19( SAVE     T_  1      T_ 18                ) Save i+1 to &i
20(  DEC    LOOPC                        T_ 20 ) Decrement Loop Counter
21(   JZ     T_ 20                    LABL_  1 ) Jmp to Label#1
22( LOAD  XADR_  0                       T_ 22 ) Load &a[0]
23(  ADD     T_ 22      T_ 18            T_ 23 ) Calculate &a[i+1]
24( LOAD     T_ 23                       T_ 24 ) Load a[i+1]
25( LOAD  XADR_  1                       T_ 25 ) Load &b[0]
26(  ADD     T_ 25      T_ 18            T_ 26 ) Calculate &b[i+1]
27( LOAD     T_ 26                       T_ 27 ) Load b[i+1]
28(  MPY     T_ 24      T_ 27            T_ 28 ) a[i+1]*b[i+1]
29( LOAD  XADR_  2                       T_ 29 ) Load &c[0]
30(  ADD     T_ 29      T_ 18            T_ 30 ) Calculate &c[i+1]
31( SAVE     T_ 30      T_ 28                ) Save T28 to &c[i+1]
32(  INC     T_ 18                       T_ 32 ) i+2
33( SAVE     T_  1      T_ 32                ) Save i+2 to &i
34(  DEC    LOOPC                        T_ 34 ) Decrement Loop Counter
35(  JNZ     T_ 34                    LABL_  0 ) Jmp to Label#0
36( LABL                              LABL_  1 ) Label#1
```

図８　ｆｏｒ文最適化の結果
ＫＩＤＯＣＨコンパイラ
（ＬＣ使用時）

さらに、ループカウンタを用いて最適化を行なった結果を図８に示す。この場合、ＡＬＵはアドレス計算だけに用いることができるので、ステップ数はさらに短くできる。

## 4. ＫＩＤＯＣＨ　ＩＶの性能

開発中のＫＩＤＯＣＨ　ＩＶの浮動小数点データの単精度演算の能力はピークで１００ＭＦＬＯＰＳ，整数データの演算能力はピークで１７５ＭＩＰＳとなる。しかし、実際にはバスが３本しかないため、データの転送能力が足らず、これほどの演算速度は達成できないと考えられる。ちなみに、単精度浮動小数点データの複素数１０２４点ＦＦＴの実行時間を試算してみたところ、約１．９ｍｓであった。これは約３０ＭＦＬＯＰＳの演算速度である。

## 5. むすび

本文では、はじめに開発中のＫＩＤＯＣＨ　ＩＶの概要を述べ、続いて、この計算機の特徴を効果的に活かすことができるように最適化を施したＶＬＩＷ型計算機用Ｃコンパイラの特徴を述べた。

## 参考文献


[1] 藤井　清人、永田　仁史、安倍　正人、曽根　敏夫、城戸　健一：多数センサによる残響下における音源位置と波形の推定、電子情報通信学会技術研究報告ＥＡ９０－１（１９９０）

[2] 安倍　正人、永田　仁史、牧野　正三、城戸　健一：ＶＬＩＷ型計算機ＫＩＤＯＣＨのメモリ管理機構、電子情報通信学会技術研究報告ＣＰＳＹ８９－４０（１９８９）

[3] 永田　他：ＶＬＩＷ型計算機μＫＩＤＯＣＨ用Ｃコンパイラ，第１５回東北大学応用情報学研究センターシンポジウム予稿集　（１９８９）

[4] 安倍　他：音響ディジタル信号処理を主目的とする高速演算装置μＫＩＤＯＣＨ，情報処理学会論文誌２８、１２、ｐｐ．１３０６－１３１７　（１９８７）

[5] 中谷　登志男：ＶＬＩＷ計算機のためのコンパイラ技術、情報処理３１，６（１９９０）