# ５ポートレジスタファイルを用いたＶＬＩＷ型計算機ＫＩＤＯＣＨ

安倍 正人 *、城戸 健一 +
東北大学大型計算機センター *、千葉工業大学 +

現状のＶＬＩＷ型の計算機の問題として、入力データのポート数が１つしか無いことがあげられる。すなわち、たとえ演算器が複数あってもほとんどの計算において実際に動作する演算器は１つだけということになり、パフォーマンスが良くない。そこで、我々は複数のデータキャッシュをサポートするＶＬＩＷ型計算機ＫＩＤＯＣＨを開発中である。具体的には２つのデータキャッシュをサポートし、さらに５ポートレジスタファイルを用いてキャッシュ間のコヒーレンスも保つようにしている。また、この５ポートレジスタファイルはMMUのＴＬＢおよび汎用のレジスタファイルとしても効果的に用いられている。

さらに、これらの機能を有効に使うためのＣコンパイラについても特にループ展開について詳しく述べている。

# VLIW COMPUTER KIDOCH USING 5 PORT REGISTER FILES

Masato Abe *, Ken'iti Kido +

*Computer Center, Tohoku University
Katahira-2-1-1, Sendai 980,Japan
+Chiba Institute of Technology

This paper describes the architecture of a VLIW computer, KIDOCH IV, with a two-port data memory unit using 5-port register files. The two-port data memory with a 4 GB addressing space is managed by the two data memory management units using the two 5-port register files and two 64 page data cache memory units. The outputs also have the information whether or not the write access was performed through a different data memory port. This allows an easy implementation of a C compiler supporting two-port data memory.

## 1. Introduction

This paper describes the outline of a VLIW computer KIDOCH IV, which is designed to improve on the architecture of KIDOCH III.[1] KIDOCH IV is composed of 15 operation units, a program control unit (PCU), a program memory management unit (PMMU), a program cache memory unit (PMU: 64 kW/256bits), two address generators (IALUa and IALUb) for data memory units (DMUa and DMUb), two data memory management units (DMMUa and DMMUb), two cache data memory units (DMUa and DMUb: 64 kW/32bits), one floating point/integer ALU (FALU), one floating point/integer operation unit (FPU), a condition code selector for condition jump (CCS), two data bus output selectors (BOS), an immediate data generator (IMM), and an interrupt controller (IRTC). They are interconnected via two 32bit wide data buses. Most of these operation units are tightly coupled with 5-port register files. The register files are used mainly as a temporary data storage. The register files are also used as a PMMU, DMMUa and DMMUb, by which 4 GW addressing space is supported. Furthermore. the register files are used as write access indicators, whereby an easy implementation of a C compiler, which supports two-port data memory, is achieved.

## 2. System overview

Figure 1 shows the block diagram of VLIW computer KIDOCH IV. 15 operation units operate simultaneously with 256 bit wide program cache memory (PMU), and are interconnected via two 32bits wide data buses. KIDOCH is connected to a host

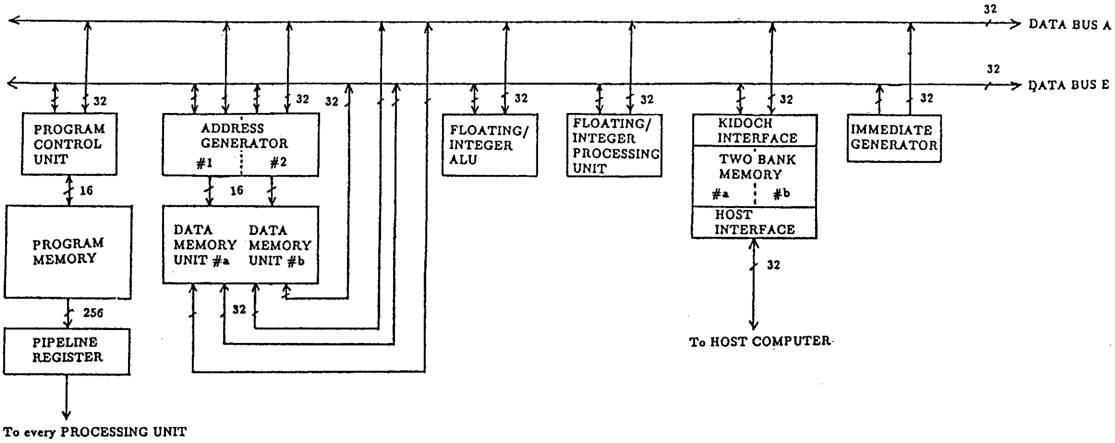

Fig. 1 Operation units of KIDOCH IV.

computer (a UNIX workstation) via VME bus. KIDOCH has only cache memories; one program memory cache of 64 kW/ 256bits (PMU) and two data memory caches of 64 kW/32bits (DMUa and DMUb).

In case of a cache miss, KIDOCH interrupts the host computer and stops. Then, the host computer changes the contents of the cache memory of the KIDOCH, and makes KIDOCH continue the processing. The cycle time of KIDOCH is 37.5 ns when data cache memory access is performed. Otherwise, it is 25 ns.

### 2.1. Operation unit and the register file

Figure 2 shows the block diagram of the FALU. 5-port register receives a loop-back data from FALU through an internal port B and two independent sets of data from other operation units through two input ports A and E, and transmits one data to other operation units through port E. It also transmits two independent sets of data to the FALU through the ports C and D. The same register file is provided for FPU, DMUa and DMUb. Therefore, the data transfer capacity of KIDOCH is twice compared to that in conventional computer systems.

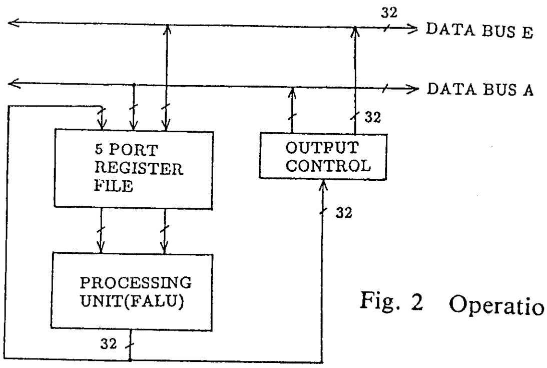

Fig. 2 Operation unit (FALU) with a 5-port register file.

### 2.2. Two-port data memory unit

Figure 3 shows the schematic view of two-port data memory unit. Two independent logical data memory addresses (A31-A0, B31-B0) are provided by the two integer ALU (IALUa and IALUb). Two independent data memory management units (DMMUa and DMMUb) support 4 GW/32bits memory space. Two data cache memory units (DMUa and DMUb) are managed by direct mapping method. They are divided into 64 page/1kW. When data memory access starts through port A, the logical address A15-A10 are transferred to the data memory management unit (DMMUa), and the output of the data management unit, MA15-MA0, are compared with the latched logical address LA31-LA16. In case LA31-LA16 are the same as MA31-MA16, and the valid flag is on, the memory access continues, that is, the content of the data cache memory (DMUa) is transferred to the data bus. Otherwise, KIDOCH asks the host computer via an interrupt to change the contents of the data

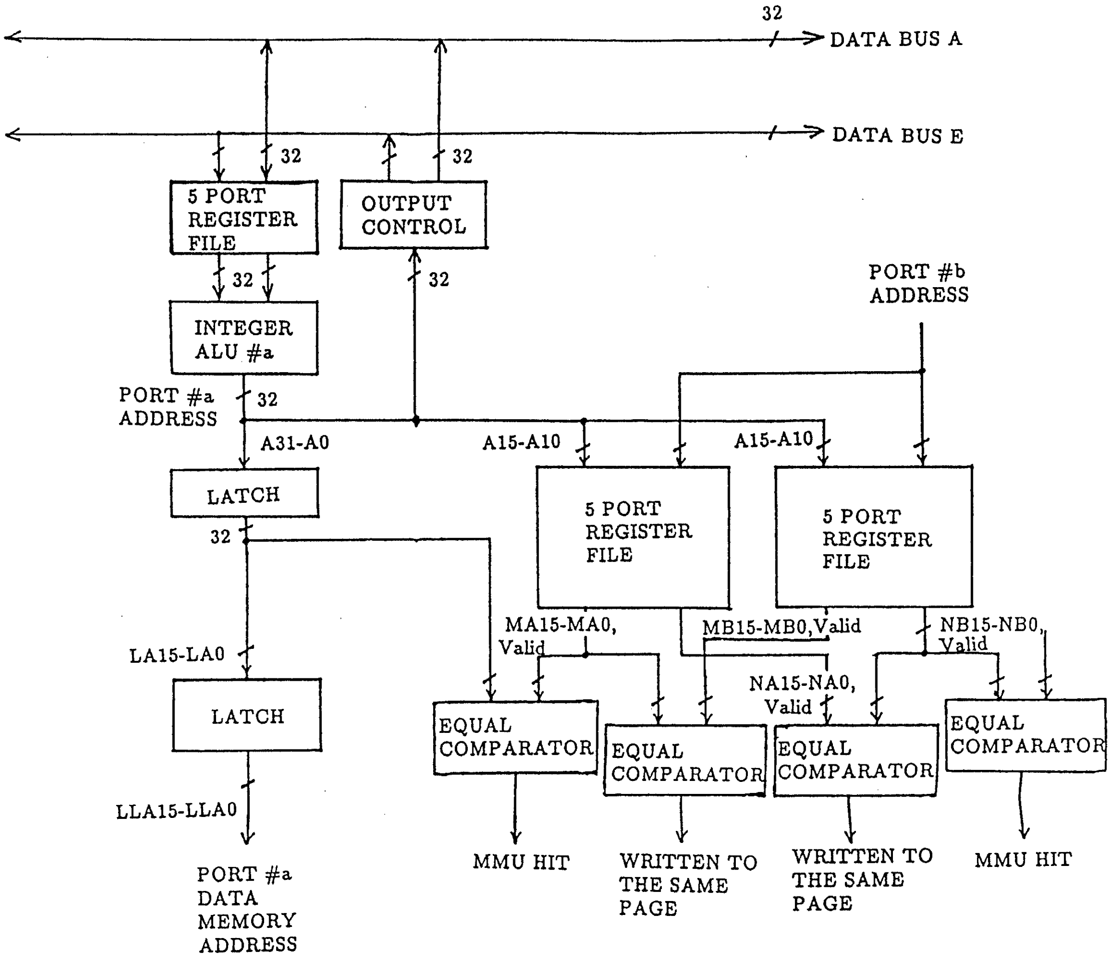

Fig. 3 Data memory management units (DMMUa and DMMUb) with 5-port register files.

cache memory. The other two 5-port register files are used to indicate whether or not memory write access was performed through the other memory port. That is, the logical addresses A15-A10 (port A) and B15-B10 (port B) are provided to the two register files. When write access is going to be performed through port A, the output addresses MA15-MA0 of DMMUa and MB15-MB0 of DMMUb are compared. If both valid flags of DMMUa and DMMUb are on and the addresses are the same, the write access flag is set in DMMUb. When a read access is going to be performed through port B and the write access flag is on, KIDOCH asks the host computer via an interrupt to change the contents of the data cache memory. This architecture provides us with an easy implementation of a C compiler, supporting the two-port memory. A very similar memory management unit is used for the program memory unit (PMMU), which offers 4 GW/256bits memory addressing space.

## 3. C compiler supporting the two-port data memory

We have already developed a C

compiler for the VLIW computer KIDOCH III. It covers the full set of standard C plus some extensions such as complex data type. Loop unrolling technique is also supported in the optimization. Figure 4 shows a sample C program, and Fig. 5 shows the assembler output of KIDOCH with the two-port memory using loop unrolling technique. The number $N$ of steps for KIDOCH IV without the loop-unrolling technique is expressed as

$$N=8M,M>1 \tag{1}$$

The number $N_1$ of steps for KIDOCH IV without the two-port memory (with a one-port memory) using the loop-unrolling technique is expressed as

$$N_1=6+3M,M>1 \tag{2}$$

The number $N_2$ of steps for KIDOCH IV with the two-port memory using the loop-unrolling technique is expressed as

$$N_2=6+2M,M>1 \tag{3}$$

where $M$ is the number of multiplications in the loop. If $M$ is sufficiently large, $N_2/M\approx$ 2, and $N_1/M\approx$ 3 respectively since it requires $3M$ memory accesses for one multiplication. This means that KIDOCH IV with the two-port memory is 1.5 times faster than that without the two port memory.

```
float a[1000], b[1000], c[1000];

main()
{
        int     i;

        for(i=0;i<1000;i++)
                c[i]=a[i]*b[i];
}
```

Fig. 4 C sample program.

## 4. Conclusion

This paper describes the architecture of the VLIW computer KIDOCH, which has 15 operation units. Since it has two-port memory (two independent data cache memory units and data memory management units), the data transfer capacity is twice compared to that in a conventional computer. The data write access flag offers an easy implementation of a C compiler effectively supporting the two-port memory unit.

## REFERENCE

[1] M. Abe, A. Shima, T. Ueda, H. Kanai, S. Makino, and K. Kido,"High speed computing machine $\mu$KIDOCH for digital signal processing on acoustics," J. Inf. Proc. Soc. Jpn., Vol. 28, No. 12, pp.1306-1317(1987)(in Japanese)

```
_main:
        imm         XADR_0;         &a[0]
        bso_a       imm;            &a[0] -> bus A
        ialu_a      a_1;            &a[0] -> REG#1 of IALUa via port A
        falu        a_1;            &a[0] -> REG#1 of FALU via port A
/
        imm         XADR_1;         &b[0]
        bso_a       imm;            &b[0] -> bus A
        ialu_b      a_1;            &b[0] -> REG#1 of IALUb via port A
/
        imm         XADR_2;         &c[0]
        bso_a       imm;            &c[0] -> bus A
        ialu_a      a_2;            &c[0] -> REG#2 of IALUa via port A
/
        imm         XADR_3;         &a[1000]
        bso_a       imm;            &a[1000] -> bus A
        falu_a      a_2;            &a[1000] -> REG#2 of FALU via port A
/
loop:
        ialu_a      c_1;            REG#1 (address &a[i]) of IALUa -> IALUa
        ialu_b      c_1;            REG#1 (address &b[i]) of IALUb -> IALUb
/
        ialu_a      pass_c;         IALUa (address &a[i]) -> DMUa
        ialu_b      pass_c;         IALUb (address &b[i]) -> DMUb
/
        ialu_a      c_1;            REG#1 (address &a[i]) of IALUa -> IALUa
        ialu_b      c_1;            REG#1 (address &b[i]) of IALUb -> IALUb
        falu        c_1;            REG#1 (address &a[i]) of FALU -> FALUa
/
        ialu_a      inc_c;          IALUa increment (address &a[i+1])
        ialu_b      inc_c;          IALUb increment (address &b[i+1])
        falu        inc_c;          FALU increment (address &a[i+1])
        ialu_a      b_1;            &a[i+1] -> REG#1 of IALUa via internal port B
        ialu_b      b_1;            &b[i+1] -> REG#1 of IALUb via internal port B
        falu        b_1;            &a[i+1] -> REG#1 of FALU via internal port B
        bso_a       dmu_a;          DMUa (a[i]) -> bus A
        bso_e       dmu_b;          DMUa (b[i]) -> bus E
        fpu         a_1, e_2;       a[i] -> REG#1 via port A, b[i] -> REG#2 via port E
        ialu_a      c_2;            REG#2 (address &c[i]) of IALUa -> IALUa
        ialu_b      c_1;            REG#1 (address &b[i+1]) of IALUb -> IALUb
/
        falu        c_1, d_2;       REG#2 (address &a[1000]) and REG#1 (address
                                    &a[i+1]) -> FALU
        fpu         c_1, d_2;       a[i] and b[i] -> FPU
        ialu_a      pass_c;         IALUa (address &c[i]) -> DMUa
        ialu_a      c_1;            REG#1 (address &a[i+1]) of IALUa -> IALUa
        ialu_b      c_1;            REG#1 (address &b[i+1]) of IALUb -> IALUb
        falu        c_1;            REG#1 (address &a[i+1]) of FALU -> FALUa
        ialu_b      pass_c;         IALUb (address &b[i+1]) -> DMUb
/
        falu        icmp;           integer compare &a[1000] with &a[i+1]
        imm         loop_end:       jump address "loop_end"
        bso_a       imm;            address "loop_end" -> bus A
        cjp         set;            set jump address
        fpu         fmpy;           floating multiply a[i]*b[i]
        ialu_a      inc_c;          IALUa increment (address &a[i+2])
        ialu_b      inc_c;          IALUb increment (address &b[i+2])
        falu        inc_c;          FALU increment (address &a[i+2])
        ialu_a      b_1;            &a[i+2] -> REG#1 of IALUa via internal port B
        ialu_b      b_1;            &b[i+2] -> REG#1 of IALUb via internal port B
        falu        b_1;            &a[i+2] -> REG#1 of FALU via internal port B
        bso_e       dmu_a;          DMUa (a[i+1]) -> bus A
        fpu         e_1;            a[i+1] -> REG#1 of FPU via port E
        ialu_a      c_2;            REG#2 (address &c[i]) of IALUa -> IALUa
/
        dcjp        falu, ge;       delayed condition jump if "ge" flag of falu is set
        bso_a         fpu;              FPU a[i]*b[i] -> bus A
        dmu_a         dat_a;            a[i]*b[i] -> DMUa via port A
        bso_e         dmu_b;            DMUa (b[i+1]) -> bus E
        fpu           e_2;              b[i+1] -> REG#2 of FPU via port E
        falu          c_1, d_2;         REG#2 (address &a[1000]) and REG#1 (address
                                        &a[i+2]) -> FALU
        fpu           c_1, d_2;         a[i+1] and b[i+1] -> FPU
        ialu_a        c_2;              REG#2 (address &c[i+1]) of IALUa -> IALUa
        ialu_a        inc_c;            IALUa increment (address &c[i+1])
        ialu_a        b_2;              &c[i+1] -> REG#2 of IALUa via internal port B
/
        falu          icmp;             integer compare &a[1000] with &a[i+2]
        imm           loop:             jump address "loop_end"
        bso_a         imm;              address "loop_end" -> bus A
        cjp           set;              set jump address
        fpu           fmpy;             floating multiply a[i+1]*b[i+1]
        ialu_a        inc_c;            IALUa increment (address &c[i+2])
        ialu_a        b_2;              &c[i+2] -> REG#2 of IALUa via internal port B
/
        dcjp          falu, lt;         delayed condition jump if "lt" flag of falu is set
        bso_a         fpu;              FPU a[i+1]*b[i+1] -> bus A
        dmu_a         dat_a;            a[i+1]*b[i+1] -> DMUa via port A
/
        nop;                            dummy instruction for jump
/
_loop_end:
```

Fig. 5 Assembler output for KIDOCH IV with the two-port memory with the loop-unrolling technique

_main: means the label
/ means the end of a step
words following ; are the comment