A Study on a Cyclic Pipeline Computer : FLATS2

循環パイプライン計算機
ＦＬＡＴＳ２の研究

Shuichi Ichikawa

市川 周一

A Study On A Cyclic Pipeline Computer:

FLATS2

by

Shuichi Ichikawa

A Dissertation Submitted to
Department of Information Science,
Faculty of Science,
the University of Tokyo
in Partial Fulfillment of the Requirements for
the Degree of
Doctor of Science

November 1990

ABSTRACT

Using the logic devices with latch function (e.g. Josephson devices), a computer is naturally pipelined to a high-pitch and shallow-logic pipeline. To utilize such pipeline fully, Cyclic Pipeline Architecture (CPA) has been proposed and researched. Though CPA is originally introduced for the Josephson computers, CPA is also effective in silicon technology to get the higher performance per cost.

This dissertation describes a Cyclic Pipeline Computer FLATS2, which is implemented with conventional Silicon technology. In FLATS2, the internal status of each virtual processor in the pipeline is simplified to ease the migration of processes between virtual processors. The communication and synchronization between virtual processors can be implemented effectively and easily, because the hardware is shared.

This dissertation also presents the instruction set architecture of FLATS2. FLATS2 provides range-checking addressing modes (BL addressing modes) and combined arithmetic instructions, which are effective in a wide variety of array processing. In BL addressing modes, any effective address is compared against the base and limit addresses specified in the instruction, then a conditional branch is executed according to the result of range check. This scheme is applicable to loop optimization and other purposes. Combined arithmetic instructions achieve high arithmetic performance, cooperating with BL addressing modes of FLATS2.

All these features are integrated to utilize hardware potential to achieve higher performance in less hardware cost. Some evaluation results of FLATS2 architecture are also presented in this dissertation.

ACKNOWLEDGEMENTS

First, I would like to express my highest gratitude to Professor Eiichi Goto for his continual encouragement, suggestions, and advices. I am also grateful to the members of FLATS2 project, Dr. Takashi Soma, Messrs. Nobuyuki Inada, and Masayuki Suzuki at the Institute of Physical and Chemical Research, Drs. Nobuaki Yoshida, Kentaro Shimizu, Noritaka Osawa, and Shuichi Moritsugu at the University of Tokyo, Drs. Mutsumi Hosoya, Tetsu Nakano, and Yasuo Wada, Messrs. Kazunori Chihara, and Noriyuki Kato at Research Development Corporation of Japan (JRDC) for their valuable discussions and supports to FLATS2 project.

Softwares for FLATS2 were developed and implemented by my co-workers in the Architecture Group of Goto Project of JRDC. I would like to thank the following people a lot for their efforts and cooperations. Mr. Norihiro Fukazawa wrote the FLATS2 assembler, SVP monitor (FLATS2 input/output supervisor) and vast numbers of programs to test FLATS2 machine. Mr. Mitsuhisa Sato developed the FLATS2 instruction simulator, some debug tools and the FORTRAN compiler for FLATS2. Mr. Paul Spee implemented GNU C compiler for FLATS2 and developed operating systems for FLATS2. Mr. Shu Kawakami wrote and improved the arithmetic libraries for FLATS2, evaluating the numerical performance.

Also, I would like to deeply thank Messrs. Kouji Ujihara, Keiichi Tsuboi, and all other engineers and managers who have engaged in this project at Mitsui Engineering and Ship-building Corporation and Mitsui Zosen Systems Research Incorporated, for their endeavor and perseverance to construct, debug, and maintain FLATS2 machine.

Table of Contents

Chapter 1

Introduction

1. Introduction

1.1. Logic Devices with Latch Function

Josephson logic devices have been researched and developed for future computer systems [1]. The prominent merits of Josephson devices are short switching time and low power dissipation. Adding to these, some Josephson devices have latch functionality with primitive logic operations. Such a latching logic device requires a clock signal to perform logic operation. Therefore, the logic function implemented with latching logic devices is naturally pipelined.

Quantum Flux Parametron (QFP) is a basic Josephson logic device invented by Eiichi Goto [2], which is expected to operate at more than 10 GHz [3, 4]. QFP consists of a pair of Josephson junctions, and it uses the polarity of the magnetic flux in a superconducting loop to represent a binary logic value.

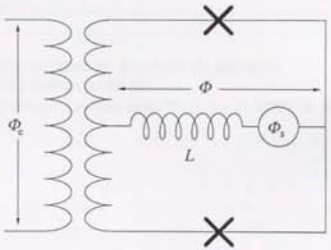


Fig. 1-1. A Schematic Drawing of QFP

Figure 1-1 illustrates QFP, where L is the external load inductance, Φ_s is the input flux, Φ_e is the excitation flux, Φ is the output flux, and $\times$ in the circuit stands for a Josephson Junction (*JJ*). Φ_s is the sum of inputs of a QFP. Φ_s has the polarity of major inputs, because inputs of opposite polarity are compensated each other. The power of QFP is supplied by a clock signal, which is called "*excitation flux*." QFP amplifies Φ_s at the rising edge of excitation flux and holds the polarity of Φ_s amplified to the output flux Φ during excitation flux is supplied (See Figure 1-2).

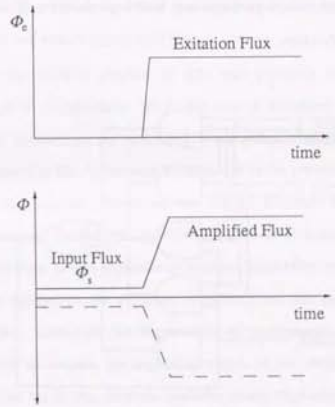


Fig. 1-2. Input/Output Flux of QFP

When excitation flux is off, input flux is passed to the output with no gain. This means that QFP acts as a latch with *majority function*, regarding excitation flux as clock signal. Any logical function can be implemented by using majority function and negation function, where the negation function for QFP is easily implemented by a

transformer which inverts the polarity of the input flux.

Latching logic devices are not peculiar to Josephson technology. Various latching logic devices have been researched and practically used in silicon technology, also. Earle [5] used a kind of latching logic gates of silicon technology (so-called *Earle Latch*) to implement the carry save adder in IBM System/360 Model 91 [6,7]. Though the first Earle latch integrated majority function with latch facility, Earle latch can be applied to any combinational logic function. For example, figure 1-3 shows a simple Earle latch, which performs two level logic function ($Z = A \cdot B + C \cdot D$) as well as the latching function.

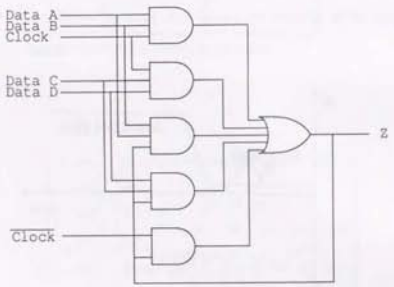


Fig. 1-3. A Simple Earle Latch

The merit of this structure is that it does not require the additional propagation delay time beyond that needed to perform the combinational function [8]. This nature is common to QFP.

1.2. Deeply Pipelined Computer

The computer implemented with latching logic devices will be naturally pipelined to many pipeline stages, each of which is logically shallow and consequently works at very high frequency. However, it is well known that the performance of a pipeline which is split into many stages is seriously degraded by the competition between instructions in the pipeline for hardware resources (*hazard* [9]). Many techniques have been researched and practiced to avoid hazards to derive higher performance [10, 11, 12], including various advanced control methods such as bypass (internal data forwarding [13]) and branch prediction [14].

However, the more a pipeline is split into pipelines stages, the harder the advanced control is to implement. First, the cost of advanced control sets the limit. Provided that n instructions are processed in a pipeline simultaneously, the difficulty of advanced control is $O(n^2)$, because hazards can occur between any combination of instructions in the pipeline. Therefore, the cost of advanced control increases radically, as n increases. Second, the available parallelism of instruction execution has a natural limit. If there is data dependency between successive instructions, or if a conditional branch appears in the sequence of instructions, the performance of pipeline will be degraded. Although this degradation of performance can be alleviated by advanced control techniques, the sequential nature of the program itself will be left unresolved. That is, if the program includes much dependency among successive instructions as its nature, most of the pipeline performance will be left unused. Resent researchers [15, 16, 17, 18, 19] have reported that the available parallelism in the instruction sequence of ordinary programs is between 2 and 4. This means that the pipeline split to far more stages than 4 will be used only partially in most situations.

1.3. The Scope of This Dissertation

As mentioned above, a single instruction stream cannot make full use of a pipeline split into many stages. To resolve this problem, Shimizu, Goto and Ichikawa [20]

proposed *Cyclic Pipeline Architecture (CPA)*, in which the pipeline is timeshared by several instruction streams. Instructions of different instruction streams do not interfere each other, because each instruction stream uses its own resource in execution. Consequently, the pipeline can be utilized fully beyond the available parallelism of a single instruction stream. The implementation cost of advanced control is reduced as well, in proportion to the number of the instruction streams, because the degree of instruction overlap for a single instruction stream is reduced by other instruction streams in the pipeline.

Though CPA was originally invented as the architecture suited for Josephson devices (especially for QFP), CPA is not peculiar to Josephson technology. CPA is also effective for deeply pipelined computer, in particular with latching logic devices. To prove the efficiency of CPA in existing silicon technology, and to obtain the experience to build future Josephson computers, a *Cyclic Pipeline Computer (CPC)* named FLATS2 was designed and implemented with existing silicon devices. This dissertation describes the architecture and the implementation of FLATS2 machine. The primary evaluation results of FLATS2 are also presented.

The purpose of FLATS2 is to fully utilize a single hardware pipelined with many stages. A multiprocessor system can be constructed with several cyclic pipelines connected each other, however such multipipeline system is beyond the scope of this study. Since FLATS2 was implemented by using existing silicon devices in market (ECL 10K/10KH [21, 22], ECL 100K [23, 24], FAST [25, 26], CMOS SRAM), the number of pipeline stages of FLATS2 is not so big. Nevertheless, FLATS2 architecture is designed to be naturally extended to the pipeline split into much more stages, like in future Josephson computers.

This dissertation presents the following contents. In chapter 2, the specification and characteristics of Cyclic Pipeline Architecture is described. Next, in chapter 3, the architecture of FLATS2 is outlined from a programmer's point of view. Chapter 4

shows how to execute various applications on FLATS2 architecture. Some examples written in high-level programming languages and FLATS2 assembly language are also given. The implementation of FLATS2 machine is detailed in chapter 5, with the description of the cyclic pipeline control of FLATS2. The primary results of performance evaluation of FLATS2 are described in chapter 6. Some famous benchmark programs are evaluated on FLATS2, showing how to execute them efficiently with FLATS2 architecture. Chapter 7 concludes the dissertation.

Chapter 2

Cyclic Pipeline Architecture

2. Cyclic Pipeline Architecture

Cyclic Pipeline Architecture (CPA) was proposed by Shimizu, Goto, and Ichikawa[20] as an architecture suited for Josephson and pipelined memory computer. This chapter outlines the idea and the characteristics of CPA as a background of FLATS2 architecture. For more details on CPA, please refer to the abovementioned paper[20].

2.1. Resource Shared MIMD

The peak performance of a pipeline computer is in inverse proportion to the segment time (τ_s), which is the time necessary to make the pipeline proceed one stage. Therefore, the shorter τ_s gets (i.e. higher the clock frequency gets), the higher peak performance is derived from the pipeline. However, shorter τ_s means that only shallower logic can be implemented in each pipeline stage, hence more pipeline stages are required to implement the same logic functions as before. On the other hand, if the pipeline is split into more stages, pipeline computer incurs more performance degradation by hazards. Consequently, the method to avoid this performance degradation has been always one of the foci in designing a pipeline computer[9, 10].

Flynn[27] proposed to share a pipeline hardware among plural *instruction streams* to resolve pipeline hazards, and he named this kind of MIMD configuration "*Resource Shared MIMD.*" Here, *instruction stream* means a *virtual processor*, which has its own program counter and register set. These virtual processors run independently, sharing a single pipeline hardware (*real processor*) in a timesharing or overlapped manner. Hazard or resource competition does not happen between instructions of different virtual processors, because each instruction stream has its own internal status. Therefore, if the instructions are orderly issued into the pipeline in turn from different virtual processors, no performance degradation will be posed by hazards.

Figure 2-1 illustrates the concept of resources shared MIMD. The figure (a) is a

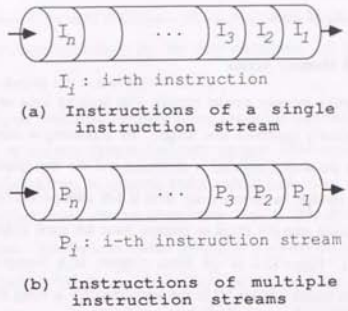


Fig. 2-1. Concept of Resource Shared MIMD

normal pipeline, in which a series of instructions ($I_1, I_2, \ldots, I_n$) of a single instruction stream is executed. On the other hand, the figure (b) shows a resource shared MIMD pipeline, in which instructions from different instruction streams ($P_1, P_2, \ldots, P_n$) are executed in the pipeline stages. Though hazards can occur among I_1 to I_n, *no* hazard can occur among P_1 to P_n.

As a resource shared MIMD computer, Denelcor HEP system[28, 29] was developed and sold in market. In HEP, memory access is left outside of the execution pipeline. That is, if a memory access was issued in an instruction stream, the instruction stream concerned is forced to wait for the completion of the memory access. Consequently, the performance of each instruction stream is strictly limited by the memory access time, because memory accesses are unable to be overlapped to the other operations in a single instruction stream. On the other hand, in CPA, memory

access is built into pipeline with advenced control to improve the performance of each instruction stream. Later in this section, HEP will be detailed as a contrast to describe the characteristics of CPA.

2.2. Pipelined Memory Access

Implementing memory control circuits with latching logic devices (e.g. QFP), the memory is naturally pipelined into stages. By pipelining a memory chip, the cycle time to accept the access requests is shortened and the throughput of memory chip is improved[20], though the access delay time is left unimproved (or slightly worsened). Even in traditional memory chips on market, there are some chips pipelined into some stages [30, 31]. Figure 2-2 is the block diagram of a 4-staged pipelined memory, where A is the access address, D is the write data for a write access, S is the output enable, W is the write enable, and Q is the output data for a read access.

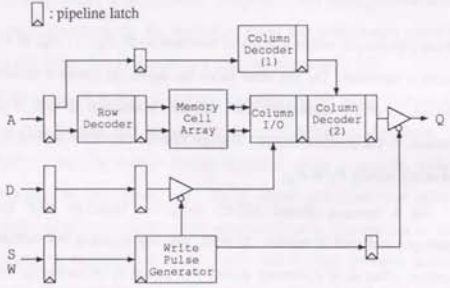


Fig. 2-2. Block Diagram of a Typical Pipelined Memory Chip

First, each input signal is latched in the input register. In the first stage, the row

address part of A is decoded to supply select signals for cell array. In the second stage, memory cells are accessed to read/write data. Also, the column address part of A is pre-decoded for the data multiplexer of the next stage. In the third stage, output data is selected from the column by the data multiplexer. In the last stage, Q is buffered and output by the output driver.

Interleaving or multi-bank memory configuration has been often used to improve the throughput of memory system, but these methods have such problems as bank conflict, access congestion on network, and uncertainty of access delay time. On the other hand, pipelining main memory at the same clock as CPU clock, no congestion or no conflict occurs. Therefore, the access delay time is always the same.

Resource shared MIMD especially fits pipelined memory very well, because memory access requests are issued in a pipelined manner from virtual processors. By timesharing main memory among virtual processors, these virtual processors naturally acts as a shared memory multiprocessor. Cyclic pipeline includes memory access in the pipeline control to overlap memory accesses to CPU operations. The certainty of access time of pipelined memory eases the advanced control in cyclic pipeline very much. Consequently, more instruction level parallelism can be utilized by overlapping instruction execution in pipeline.

Pipelined memory chip was unavailable when CPA was first proposed. However, pipelined memory has got practical and available recently[30, 31, 32]. Though the memory of FLATS2 is not pipelined in the current implementation, FLATS2 architecture was carefully designed so that future FLATS2 implementation could be implemented with pipelined memory chips.

2.3. Characteristics of Cyclic Pipeline Architecture

In designing a resource shared MIMD computer, it is an important subject to decide the number of instruction streams (m) against the number of pipeline stages (n).

Also, the scheduling scheme of virtual processors is another important theme.

In abovementioned HEP computer[28, 29], maximum 128 instruction streams share an execution unit (*Process Executing Module* or *PEM*) pipelined with 8 stages. The instruction stream of HEP is called "*process*," and the number of processes are variable in HEP pipeline. The processes of HEP are dynamically created and deleted in executing a program, while virtual processor of CPA is a permanent entity, the number of which is determined by the pipeline design.

There is a queue of processes ready for execution (*snapshot queue*) in PEM of HEP, and processes are taken from this queue to the execution pipeline (See Figure 2-3 (a)). If a process issues a memory access, the process concerned goes out from snapshot queue to another queue (*storage function unit queue*) to wait for the completion of the memory access. As the number of processes varies according to the load of CPU, when the number of processes in snapshot queue is less than the number of pipeline stages (i.e. 8), some of execution pipeline stages is left unused. As shown above, the execution order of processes in HEP is managed dynamically at execution time, affected by other processes in PEM. This makes advanced control very difficult. In fact, so far as the instruction execution of a single process is concerned, instructions are executed serially in HEP, without using the parallelism available by pipelining. Consequently, the performance of a single process is limited to a very small portion of the total performance of the execution pipeline. Briefly saying, the relation $m \geq n$ holds in HEP architecture,

On the other hand, in CPA, the number of instruction streams (m) is fixed, so that each instruction stream would always occupy the pipeline stages every m stages. Under this condition, advanced control is simplified very much. That is, advanced control is only necessary every m stages, because the scheduling of instruction stream is always static. With advanced control, the instructions from the same instruction stream can be executed overlappingly in the pipeline, improving the performance of a

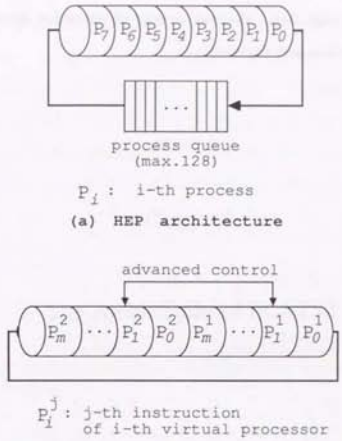


Fig. 2-3. HEP and Cyclic Pipeline Architecture

single instruction stream. Figure 2-3 (b) illustrates this nature of CPA. As shown in the figure, the relation $m < n$ holds in CPA. If the value of m is small, the total quantity of internal status of m virtual processors gets small accordingly, which means that the cost of hardware gets also small.

Another important merit is that the efficiency of cache and memory usage is improved, because the total *working set size* of memory (or cache) is basically proportional to the number of virtual processors. Too many instruction streams ($m > n$) will

enlarge the active working set and incur the degradation of memory throughput by cache miss or page fault. Moderate number of instruction streams ($m < n$) of CPA avoids such performance degradation.

Chapter 3

The Architecture of FLATS2

3. The Architecture of FLATS2

This chapter describes the architecture of FLATS2 mainly from the programmer's point of view. First, the basic specification, the purposes, and the characteristics of FLATS2 are presented. Then, the instruction set architecture of FLATS2 is outlined. For systematic description, please refer to "FLATS2 architecture handbook"[33] and "FLATS2 instruction set manual"[34]. The examples of programming on this architecture will be given in chapter 4.

3.1. Purposes, Characteristics, and Specifications

FLATS2 is a cyclic pipeline computer with 2 virtual processors. One of the purposes of FLATS2 is to verify the efficiency of cyclic pipeline architecture in existing silicon technology. Another important purpose is to obtain the experiences on cyclic pipeline design for future Josephson computers. Hence, FLATS2 architecture is designed in consideration of Josephson computers which will have many virtual processors, though the number of virtual processors of FLATS2 is small by reason of implementation technology. FLATS2 is consequently designed to have little context in each virtual processor to reduce context switch overhead. This makes *threading* [35, 36] of programs very easy and efficient, thus utilizing cooperating virtual processors for a task.

One of the most distinct characteristics of FLATS2 instruction set architecture is addressing modes with range checking facility (*BL addressing mode*). Cyclic pipeline architecture attempts to derive high throughput from a hardware pipelined into many stages by using plural instruction streams (virtual processors). Though the total throughput of a pipeline is improved by CPA, another technique is required to improve the performance of each instruction stream. In FLATS2 pipeline, some advanced control techniques such as data bypass are adopted to increase the degree of instruction overlapping to improve the performance of each instruction stream. In addition to this, FLATS2 introduced an architecture called BL addressing to utilize more parallelism at

instruction set level.

BL addressing is an extension of usual addressing modes. In BL addressing modes, any effective address is checked against the given base and limit addresses before accessing memory. If the access address is out of the range between base and limit, the memory access concerned is denied and an access error occurs. This access error causes a branch or a trap immediately, thereby the program changes the action hereafter. This feature is useful for various applications such as array processing, symbol manipulation, and dynamic array boundary check to detect software errors in program execution. Such usages are detailed later.

In addition to the range check facility, FLATS2 offers much numerical arithmetic functionality to each instruction. Maximally 8 floating-point arithmetic operations (single precision) are performed in one instruction cycle, while the operands are supplied from memory by using BL addressing. This numeric facility cooperates with BL addressing modes to accomplish effectiveness in numerical applications. The numeric performance of FLATS2 will be reported in later chapter on linear algebraic solver and other scientific codes.

Machine Cycle	65 ns
Word Length	32 bit (data) + 1 bit (tag)
Real Memory	5.5M Byte (currently)
Logic Devices	ECL10K, 10KH, 100K, FAST
Logic Board	465(W) × 323(H) multi-layered, multi-wired total 26 boards
Package	960(W) × 1200(H) × 1075(D)

Table 3-1. Basic Specification of FLATS2

Table 3-1 shows the basic specification of the current FLATS2 hardware. Machine cycle is the cycle time of primitive machine clock, by which the pipeline proceeds one stage. In FLATS2, memory cycle is equivalent to machine cycle time. The transfer rate of memory is thus 123×10^6 bytes/second for instruction and data,

respectively. Mainly ECL 10K/10KH series[21, 22] (DIP package) is used to implement FLATS2 logic. Adding to these, some ECL 100K series[23, 24] (DIP) chips are used in some units (e.g. floating-point arithmetic unit). No highly integrated devices (i.e. LSIs) are used except for 16 × 16 bit parallel multiplier chips (CMOS/SOS) for the integer/floating-point multiplier unit. TTL chips (FAST[25, 26]) were partially used in memory control circuits. FLATS2 consists of 26 logic boards, each of which contains from 200 to 400 chips on it. These boards are connected each other by a backplane board and front flat cables, mounted to a rack chassis (about 57 cm × 62 cm × 37 cm) which is air-cooled by 14 fans. This rack chassis is packed into a cubic box with power supplies.

The current FLATS2 is designed as a back end processor, which serves computational resources with only limited input/output features. Programs for FLATS2 are developed, compiled, and linked on the *front end processor (FEP)*, which serves the interface and development environments for FLATS2. The current FEP is a UNIX workstation (DEC microVAX 3500). Executable binary image is downloaded from FEP to FLATS2 via network (ethernet). The input/output of the program running on FLATS2 is also performed via network.

3.2. Hardware Overview

This section outlines FLATS2 hardware configuration. Figure 3-1 illustrates the block diagram of FLATS2. FLATS2 CPU consists of three major units, which are *Instruction Processing Unit (IPU)*, *Global/Variable register Unit (GVU)*, and *Sum and Product Unit (SPU)*. IPU fetches an instruction from *Instruction Memory (IM)*, decodes it to execute, and controls the program sequence. General registers are called Global/Variable registers (GV registers) in FLATS2 architecture. A fixed amount of GV register sets (GV frames) is prepared in an independent memory called *Global/Variable register Memory (GVM)*, in which 4096 sets of GV registers are prepared in the current FLATS2 implementation. *Current Frame Pointer (CFP)* of

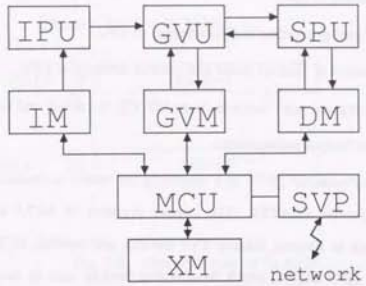


Fig. 3-1. Block Diagram of FLATS2

each virtual processor specifies which GV frame is used for this virtual processor. GVU performs arithmetic operation, logical operation, and address calculation between GV registers. SPU consists of 64 bit integer/floating-point ALU, 64 bit integer/floating-point multiplier, and 64 bit registers (*SP registers*) for SPU operations. SPU is a general purpose arithmetic unit, which can take memory operands from *Data Memory (DM)*. Memories such as IM, DM, and GVM are implemented with SRAM chips which are fast but expensive, thus the total capacity of these memories is not so big. *Extended Memory (XM)* is a staging memory of IM/DM/GVM, implemented with DRAM chips which are slower but cheaper than SRAM. The capacity of XM is planned to be 64M byte, though it is not included in the current implementation.

MCU is a *Memory Control Unit*, which has the following functions:

(1) DMA transfer to/from IM/GVM/DM,

(2) management of XM,

(3) clock signal generation and distribution to CPU,

(4) initialization of internal status and control storages of CPU,

(5) internal register scan function (*scan I/O*[37]) for debug and maintenance of CPU,

(6) and input/output management.

Service Processor (SVP) is a minicomputer which is connected to MCU to control and supervise FLATS2. The control registers of MCU are mapped on SVP memory space as devices, thereby SVP initiates and controls MCU features. SVP has its own bus (*SVP bus*) to attach input/output devices such as console, disks, and network interface. When FLATS2 is executing a program, SVP watches for the input/output requests from FLATS2, and controls devices on SVP bus to substitute input/output of FLATS2.

3.3. Memory System

3.3.1. Address Tag

The most distinguishing characteristic of FLATS2 memory system is to have an *address tag* for each word. Though number and address are the most common and fundamental types in programs, usual computers cannot distinguish them by their memory representations. In FLATS2, the address tag of each word indicates this difference. Figure 3-2 illustrates the address tag of FLATS2 memory word. A word of FLATS2 consists of 32 bit data and 1 bit address tag. If the address tag is zero, the corresponding word contains a numeric datum. If the address tag is set to one, the corresponding word contains an address. Any word in GV registers, instructions in IM, and data in DM has an address tag for each word. These address tags cooperate

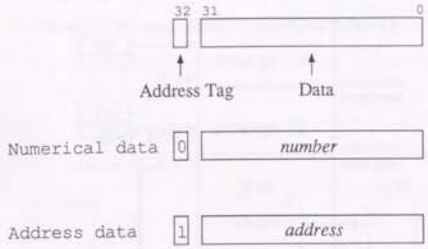


Fig. 3-2. Memory Word of FLATS2

with address range checking facility of BL addressing modes to achieve loop optimization and data protection.

3.3.2. Memory Space

FLATS2 provides a single virtual address space, which is shared among every processes in the real processor. This shared single address space greatly eases the cooperation of processes and the migration of processes among virtual processors, which are very important to utilize a resource shared MIMD. However, the memory protection from other processes can be a problem of the shared single space. Such antinomy has been usually resolved by implementing multiple virtual storage with memory sharing mechanism, or by implementing a hardware support for the group of cooperating processes (so-called *task*). Though all these techniques are applicable to FLATS2 (or shared resource MIMD), such supports are omitted in FLATS2 to reduce implementation cost. It is not important for FLATS2 to achieve perfect protection, because FLATS2 is not a commercial computer but an experimental computer.

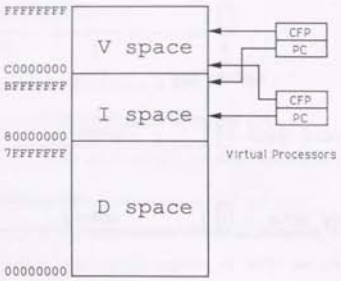


Fig. 3-3. Memory Space of FLATS2

An address in FLATS2 is 32 bit long, excluding the address tag. This single address space of FLATS2 is split into three subspaces according to its memory configuration; that are *D space* for DM, *I space* for IM, and *V space* for GVM. Representing the virtual address in hexadecimal format (preceded by 0x), D space is located from 0x00000000 to 0x7FFFFFFF, I space is located from 0x80000000 to 0xBFFFFFFF, and V space is located from 0xC0000000 to 0xFFFFFFFF. Though each memory is mapped into the corresponding address space by its association mechanism, fixed areas are reserved for trap handling routines to be resident in the lowest addresses of each space. These areas are mapped to have the same virtual address as its real address (V=R area), and are never swapped out to the secondary storage. Each trap entry corresponds to each cause of traps, and each virtual processor has its own sets of entries in resident areas.

3.3.3. Memory Hierarchy

Figure 3-4 shows the memory hierarchy of FLATS2.

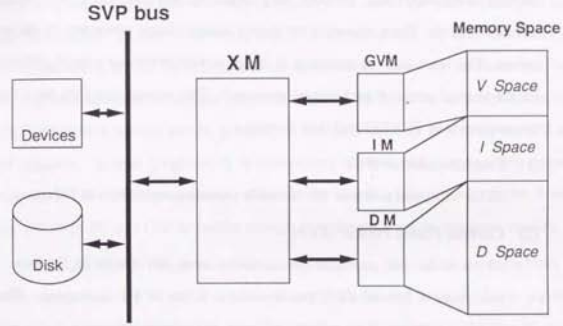


Fig. 3-4. Memory Hierarchy of FLATS2

The memory system of FLATS2 consists of 3 levels. IM, GVM, and DM act as three dedicated cache memories for IPU, GVU, and SPU, respectively. XM is to act as a backup memory for IM, GVM, and DM, while it is also used for the buffer of DMA and I/O. However, the current FLATS2 does not include XM, consequently DMA is directly performed between IM/GVM/DM and SVP bus. The virtual memory image of programs is stored in the *swap area* of the disk on SVP bus. The rest of SVP disk can be used for the file system of FLATS2 operating system.

3.4. Virtual Processor Architecture

There are two kinds of hardware resources in FLATS2; one is the shared resource which belongs to the real processor, and the other is the resource which belongs to each virtual processor. Some examples of the former resources are as follows: memory (IM/DM/GVM), memory map registers (*key memory*), and input/output memory (*iodev*). These resources are shared among virtual processors in the real processor. The latter kind of resources is called *virtual processor context*, which stands for the internal status of each virtual processor. This section describes the virtual processor context of FLATS2 and their functions.

(1) **Program counter (PC)**

32 bit wide, and points to the currently executing instruction in I space.

(2) **Current Frame Pointer (CFP)**

32 bit wide, and points to the currently using GV frame in V space. In the lowermost 6 bits of CFP, the lowermost 6 bits of PS is mapped. These bits includes *privileged bit* which indicates this virtual processor is in privileged status, *interrupt mask bit* which indicates this virtual processor is currently interrupt disabled, *trace bit* which indicates *trace trap*† is enabled in this virtual processor, and *halt bit* which indicates this virtual processor is currently halting.‡ For more detail, please refer to FLATS2 architecture handbook [33].

(3) **Condition Code (CC)**

4 bit wide, which includes Z (zero), O (overflow), N (negative), and C (carry) bits.

(4) **SP registers (S, P, T, Q, R, U)**

† *Trace trap* is initiated after every normal terminations of instruction execution, thereby a trap handler can trace PC values of successfully executed instructions.

‡ In *halt* status, the virtual processor does not execute any instruction, holding its internal status unmodified, until halt status is resolved. Virtual processors can escape from halt status by receiving an external interrupt.

64 bit wide each, which are used in M format operations.

(5) **Processor Status (PS)**

32 bit wide, which represents settings and status that is related to the behavior of processor.

By saving and restoring abovementioned registers, the context of each virtual processor can be switched. In case of interrupt, trap, and subroutine call, PC and CFP (in which the lowermost 6 bits of PS is included) are automatically saved by hardware. CC and SP registers are used to contain temporary results, therefore in case a program willingly requests a context switch, generally it can be omitted to save or restore CC and SP registers. In user programs, it is unnecessary to save and to restore PS register, because PS is a privileged register. After all, for user level context switch (for threading [35, 36]), PC and CFP is usually enough to represent a virtual processor context.

In FLATS2, GV registers act as general registers. 64 registers are available in each instruction, each of which is 33 bit wide (32 bit data + 1 bit tag). These registers are grouped into 2 frames, each of which consists of 32 registers. The first 32 registers (from 0th to 31st) are called *General Registers (GR)*. The other registers (from 32nd to 63rd) are called *Variable Registers (VR)*. Variable register frame (*V frame*) is pointed by CFP on V space, on the other hand, Global register frame (*G frame*) is the same frame regardless of CFP.* *Call* (subroutine call) instruction automatically increments CFP register and saves the old CFP value to a GV register, thereby switching the V frame in use. This old CFP value is restored from GV register to CFP register, when a *ret* (return) instruction is executed. *Interrupt* and *trap* also involve a V frame switchover to reserved frames, each of which corresponds to each reason of exceptions. After switching V frame to the reserved frame, the control is passed to the

* Precisely, there is an 8 bit GFP field in PS, which specifies one of the lowest 256 frames in V space. However, G frame does not change in user level software, because GFP is part of PS which is a privileged resource.

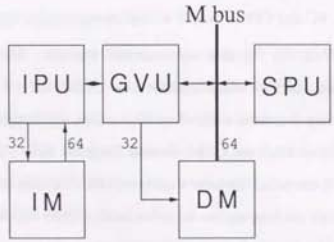
appropriate trap handler.

By using CFP register to access V frame indirectly, GV register set can be excluded from the virtual processor context of FLATS2. This reduces the overhead of context switch very much, which makes threading of a task more effective in many applications.

3.5. Instruction Set

3.5.1. Processor Model

Figure 3-5 illustrates the processor model from the viewpoint of FLATS2 instruction set architecture.




```
IPU : Instruction Processing Unit,
GVU : Gloval/Variable register Unit,
SPU : Sum and Product Unit,
IM  : Instruction Memory,
DM  : Data Memory.
```

Fig. 3-5. Processor Model of FLATS2

IPU and GVU are tightly connected eath other by internal control and data paths. IPU and GVU work together just like an independent processor, which has a self-contained

load/store architecture for GV registers. Three instruction formats (*I*, *J*, and *K* format) are prepared for this IPU/GVU subarchitecture. SPU acts as a co-processor or an integrated accelerator of IPU/GVU subprocessor for memory oriented operations, including floating-point arithmetic. SPU is connected to IPU, GVU, and DM conceptually by a bus called *M bus*,† via which the operands are transferred to/from SPU. An instruction format called *M format* is dedicated for SPU operations, which gives a general scheme to handle the whole processor resources.

The following sections describe the framework of FLATS2 instructions set architecture, including instruction formats.

3.5.2. Instruction Format

Instruction formats of FLATS2 have basically the same length. Figure 3-6 shows 4 basic instruction formats of FLATS2; that are, *I format*, *J format*, *K format*, and *M format*. All these basic formats are two word long. And adding two data words (immediate words) to these two words, each format can be extended to the corresponding *long format*; *IL format*, *JL format*, *KL format*, and *ML format*. Long formats are only used in case extended immediate words are necessary.

The first word of each instruction format is identical. *GVop field* is the opcode for GVU, *md field* selects the operands, and *r1*/*r2*/*r3 fields* designate GV registers which are used in the instruction. The address tag of this first word is ignored, though it should be zero.

3.5.3. I format

I format is mainly used for GVU operations without address calculation. In this format, three GV registers are used for the operands (r1 and r2 for source, r3 for destination). GVop field of I format designates the operation in GVU. The second word

† This conceptual bi-directional bus is actually implemented by three one-way data paths of ECL level signals.

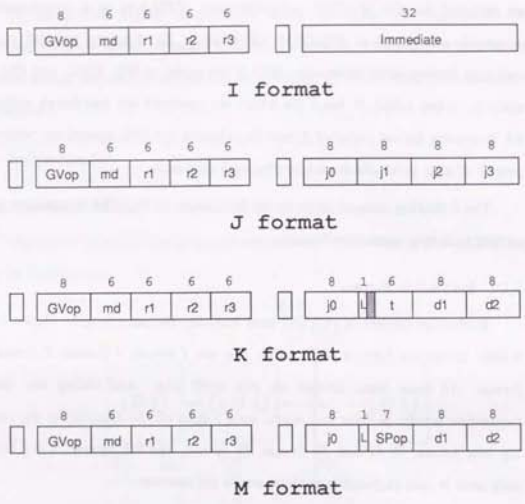


Fig. 3-6. Basic Instruction Formats of FLATS2

of I format is an immediate word, which can be used one of source operands. For example, *call* (subroutine call) instruction and *addr* (add GV registers) instruction belong to I format instructions. In IL format instruction, the third and fourth words follow as a double word immediate.

3.5.4. J format

J format is the format for various *conditional branches*. Conditional branches of J format are basically 4 way branch instructions, where GVop field designates the kind of branch operation and md field encodes a branch condition. The second word of J format is split to four signed 8-bit offsets, each of which represents the offset from the current PC to each branch target. Conditional branch instructions of FLATS2 are classified into three kinds, that are *branch on condition code*, *compare and branch*, and *compare and branch with arithmetic*. Register fields (r1/r2/r3) are used for operands of comparison and arithmetic. One of branch offset fields can be altered by a signed 8-bit immediate for arithmetic operand, if the branch is 3 way. FLATS2 executes all these conditional branch instructions in one instruction cycle.

If any of branch offset is beyond the limit of 8 bits, or if absolute address is necessary for branch target, *conditional jump* instructions of JL format must be used. Conditional jump instructions of FLATS2 are also classified into three kinds, that are *jump on condition code*, *compare and jump*, and *compare and jump with arithmetic*. In JL format, each of the second, third, and fourth words contains a full word offset or address. There is no field for the fourth target, thus the fourth target is defined to be the next instruction.

All these instructions are implemented by using mostly the same hardware as BL addressing. One of distinguishing characteristics of FLATS2 from the similar instructions of other machines (e.g. *acb* instruction of VAX [38]) is that FLATS2 can perform range checking also in *compare and branch*. The operand or the arithmetic result can be compared either to another operand (comparand) or against the range specified by a pair of operands. In the latter case, *compare and branch* operation works just like address calculation of BL addressing besides address tag checking with 4 way branch targets.

3.5.5. K format

K format is the format for GVU operations with address calculation. For example, *lea* (load effective address) instruction and its variants, various load instructions, and various store instructions are included in K format instructions. FLATS2 can calculate two effective addresses in one instruction with address range checking (*BL check*). K format thus contains the fields for *BL addressing* in the second word; that are *j0* field for the branch offset of BL addressing, *d1* and *d2* fields for 8-bit signed displacements to calculate two effective addresses. K format includes another register field (*t field*) to designate a target register, in addition to three source registers (r1/r2/r3). *L field* is always zero in K format.

If displacements beyond 8 bits are required for address calculation, KL format can be used. *L field* or *L bit* differentiates KL format from K format. If L bit is set, this instruction is a long format, which means that two more words follow after this second word. In KL format, the third word (*D1*) and the fourth word (*D2*) substitute for short displacements (d1 and d2), respectively.

3.5.6. M format

M format is the format for general arithmetic instructions of FLATS2. Maximally two operands can be taken from data memory, one is for source and the other is for destination.‡ The effective addresses of these operands are calculated by GVop field. M format instruction can take the following four kinds of operands such as GV register, immediate, data memory, and SP register. The selection of operands from these four kinds is encoded in md field. M format and K format only differ in having *SPop field* instead of *t field*. This SPop field designates the operation in SPU, which

‡ FLATS2 can not perform arithmetic with two memory data for source operands, even if the destination is a register. It is possible only in special cases to use both memory operands for source. Please refer to "FLATS2 architecture handbook"[33] for *R/R addressing modes* and *special numeric instructions*.

instructs how to operate on the operands specified by GVop and md fields.

If displacements beyond 8 bits are required for address calculation, ML format can be used instead of M format by setting L bit to one. In ML format, the third word (*D1*) and the fourth word (*D2*) substitute for short displacements (d1 and d2), respectively.

3.6. BL Addressing Modes

3.6.1. Address Tag

FLATS2 distinguishes an address from a numerical data by the address tag of each word. This is a very simplified implementation of *tag architecture*, which is often used in LISP machines and other symbolic manipulation machines (for example, Symbolics 3600 [39], Xerox Dorado [40], Fujitsu ALPHA [41], UCB SPUR[42], Riken FLATS [43, 44, 45, 46]). The application of address tags to such purpose is described in later sections.

In FLATS2, the memory must be accessed with an address, that is, with a word which has its address tag set to one. FLATS2 thus checks the values of address tags in calculating an effective address. When an addition is performed in address calculation, the resulting type is defined as Table 3-2.

number	+	*number*	→	*number*
address	+	*number*	→	*address*
number	+	*address*	→	*address*
address	+	*address*	→	*number*

Table 3-2. Type Arithmetic Rule in Addressing Calculation

Here, *address* stands for the word of address tag set to one, and *number* stands for the word of address tag cleared. To perform more complicated address calculation, this rule is recursively applied. The type of the final result (effective address) must be an address to access memory data successfully. Otherwise, the memory access is denied

and an access error occurs.

3.6.2. Range Check Facility

In any address calculation of FLATS2, the effective address is compared against a given pair of *base* and *limit* addresses (*BL pair*). A BL pair stands for a memory area, which is between *base* and *limit* addresses including both ends. Figure 3-7 illustrates a BL pair and the corresponding memory area.

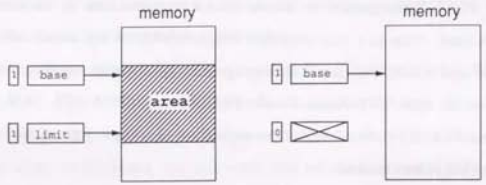


Fig. 3-7. BL pair and memory area

If both of *base* and *limit* have their address tag asserted, this BL pair stands for a minimum array which includes only one element. If either *base* or *limit* is not an *address*, that is, if either of them has its address tag cleared, this BL pair does not represent a memory area any more. There is no restriction on the values of base and limit. Base address can be higher than, equal to, or less than the limit address. This symmetric nature enables the uniform handling of both kinds of array accesses (from the top and from the bottom) in the same way.

A BL pair is specified by a register field (r1/r2/r3) in K/M format instructions. The specified register is used as *base register*, and the binary pair† of the specified

† *Binary pair* means the number which has the least significant bit inversed from the original number represented in binary form. For example, the binary pair of 12 is 13; the binary pair of 7 is 6.

register is used as *limit register*. Both base and limit registers must contain addresses, otherwise this access will fail to cause an access error. Also, in case the effective address is out of the range between given base and limit addresses, the access is denied and an access error occurs.

FLATS2 makes a conditional branch according to the results of memory accesses in BL addressing modes.

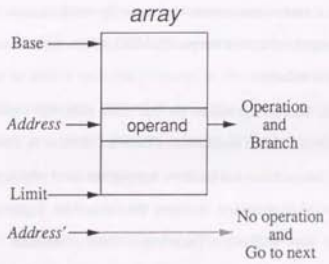


Fig. 3-8. Branch on range check result

Figure 3-8 illustrates this branch scheme.

(1) If an access error does *not* occur, the instruction concerned is successfully executed and then the branch to the offset *j0* is performed immediately.

(2) If an access error *does* occur, the instruction execution is aborted, consequently proceeding to the next instruction.

Note that the branch is taken in case the access succeeded. This specification is fully

utilized in loop optimization and other applications of FLATS2, which will be detailed in the next chapter.

Though the branch on range check is effective in wide variety of programming, it is common to set *j0* to the next instruction in many situations. Users frequently assume that a memory access should succeed, because in many cases the nature of the memory access can be definitely known in the program. Therefore, branch-on-range-check facility of memory access can be often left unused in BL addressing mode. Even in such a case, access errors can occur by some reasons that users did not expect. For these unexpected access errors, FLATS2 offers *address calculation trap* facility to catch abnormal accesses.

Also, BL addressing makes the run time subscript checking of arrays very easy and very effective. Most architectures have no devices to check the subscript of arrays at run time. In such an architecture, a program must check the range of subscript by the sequence of instructions, making the code size bigger and the execution time longer than the original. Therefore, many compilers provide the option to enable/disable this subscript range checking and users often disables the range checking in executing program to avoid the loss of performance in exchange for the reliability of the program[47]. This is stupid, because the subscript out of range is one of the most popular bugs in programs. The architectural supports for this problem will reduce the effort of debugging and improve the reliability very much. BL addressing can be used to check the array subscript at run time with the least overhead and cost.

The following piece of code is a commonplace sequence that loads an operand (*8(vr1)*) to add it to a register (*gr1*).

Example 1:

```
        mov.l   8(vr1), gr0     ; move a long word to gr0
        addr    gr0, gr1, gr1   ; add register gr0 to gr1
```

The access to *8(vr1)* is assumed to succeed certainly, though it can fail if *vr1* is broken

by some unexpected reasons. In FLATS2 code, this sequence is implemented in the following manner.

Example 2:

```
        mov.l.j 8(vr1), gr0, L001   ; move and jump to L001
L001:   addr    gr0, gr1, gr1       ; add register gr0 to gr1
```

The first instruction *mov.l.j* is an M format instruction, *j0* of which points to the label *L001*. Let the BL pair for this access specify all the data area for this user. Here, L001 is the next instruction of *mov.l.j*, thus *addr* instruction seems to be executed anyway after *mov.l.j* is executed whether the access error occurs or not. However, if an access error occurred in such a case that *j0* points to the next instruction, FLATS2 initiates an *address calculation trap* instead of proceeding to the next instruction. This feature enables to check unexpected memory accesses by hardware in FLATS2.

In FLATS2 assembler, the description on branch can be omitted from assembler notation of BL addressing, if the branch target is meant to be the next instruction. Thus removing "*.j*" and the branch target description, **Example 1** and **Example 2** generate the same code by FLATS2 assembler. Users can use the former style in case branch target is unnecessary in BL addressing.

3.6.3. Addressing Modes

Table 3-3 shows addressing modes of FLATS2. In assembler notation, *b* (base register) can be omitted, in which case the default BL (a whole user's data area) is assumed as the base. Basically, only one addition can be performed to calculate an effective address for a memory access. Effective addresses of FLATS2 are classified into three kinds according to the operands of address calculation; *pointer (p)*, *index (x)*, and *offset (o)*.

Addressing Mode	Notation	Effective Address	Side Effect
Immediate	#xxx	xxx	
GV register	grn, vrn	(GV register)	
SP register	S, P, T, Q, R, U	(SP register)	
Pointer	*b:d(p)*	*p + d*	
Pointer Push	*b:>d(p)*	*p + d*	*p += d*
Pointer Pop	*b:<d(p)*	*p*	*p += d*
Index	*b@x*	*b + x*	
Index Push	*b@>x*	*b + x*	*b += x*
Index Pop	*b@<x*	*b*	*b += x*
Offset	*b@d*	*b + d*	
Offset Push	*b@>d*	*b + d*	*b += d*
Offset Pop	*b@<d*	*b*	*b += d*
Offset Index	*b@d(x)*	*b + d + x*	
Pointer Index	*b:d(p)x*	*p + d + x*	

b is base register, *p* is pointer register, *x* is index register, *d* is displacement.

Table 3-3. Addressing Modes of FLATS2

Class	Address Calculation		
Pointer	pointer	+	displacement
Index	base	+	index
Offset	base	+	displacement

In the Table 3-3, *pointer*, *base*, and *index* are GV registers which are designated by *r1/r2/r3* fields. The *displacement* is either *d1* or *d2* field in the instruction. The *base* register is also used for the range checking of BL addressing with the corresponding *limit* register. It varies according to the combination of operands which register field to be used for each of *base*, *index*, and *pointer*.†

Offset class is used typically to access the array specified by *base* with a constant index. This case appears very frequently to access the variables defined in user program, because any user-defined variable can be represented by the *displacement* from

† See "Architecture Handbook"[33] and "Instruction Set Manual"[34] for more details.

the *base* of the user's data segment. In this case, the specification of *base* can be omitted, in which case the entire user's data segment is used as default *base*. *Index* class is used typically to access the array specified by *base* with a variable subscript in *index* register.‡ These three classes can be easily implemented by using the data path of GVU, which is required for register arithmetic operations of I format.

Each of these classes has two variants on side effects, *push (pre-modify)* and *pop (post-modify)*. These two variants are used typically for push and pop operations of a stack. In *push*, the calculated effective address is first used to access memory, and then written back to *pointer* or *base* register. In *pop*, the value of *pointer* or *base* is used as the effective address to access memory, and then the calculated address is written back to *pointer* or *base* register. This *pop* variant plays an important role in array processing of numerical applications, which is described in later sections.

In addition to these three (and their variants), two more classes are available in address calculation instructions that do not access memory but only calculate effective addresses. *Pointer Index* and *Pointer Offset* perform two additions to calculate an effective address, as shown in Table 3-3. The second addition is performed using the time for the memory access, which is vacant in these two classes.

3.7. Numerical Instructions

3.7.1. Arithmetic Unit Architecture

SPU is the arithmetic unit to perform logical operations, integer arithmetic operations, and floating-point arithmetic operations of M format. Figure 3-9 illustrates the conceptual framework of SPU. The ALU in the figure is called *S unit*, which is a 64 bit wide integer/floating-point ALU. The output of S unit can be sent to DM via M bus, or stored into either *S register* or *T register*. The multiplier is called *P unit*,

‡ FLATS2 does not offer the facility to scale *index* according to the operand size, while this feature is supported in CISCs like VAX[38] and MC68020[48].

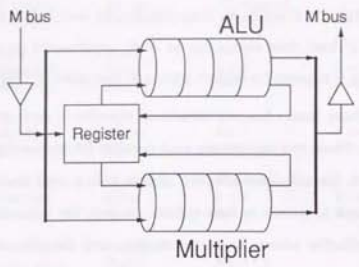


Fig. 3-9. Block Diagram of SPU

which is a 64 bit wide integer/floating-point multiplier. The output of P unit can be sent to DM via M bus, or stored into either *P register* or *Q register*. R and U registers are used to fetch and hold data from M bus. By using R and U registers as input buffer register, load operations from DM to R/U registers can be overlapped to the operations in S and P unit between S/T/P/Q registers. Both S unit and P unit can be used twice in each instruction cycle. Each instruction is therefore able to perform maximally 2 multiplications and 2 ALU operations (all 64 bit wide) in one instruction cycle.

3.7.2. Combined Arithmetic Instructions

A usual M format instruction only uses either S unit or P unit. For example, *add.f* instruction uses only S unit once, and *mul.d* instruction uses only P unit once. However, it is no use leaving any of arithmetic unit idle, because both S and P unit are pipelined to accept operations every pipeline cycle. Therefore, it is effective to define

and use such instructions that perform plural arithmetic operations; both S and P unit simultaneously or S or P unit twice in a single instruction. Such instructions are called *combined arithmetic instructions*. FLATS2 offers many kinds of combined arithmetic instructions, some of which are described hereafter.

Most typical example of combined arithmetic is *poly (polynomial evaluation)* instructions. This instruction performs two multiplications and one addition simultaneously to calculate the value of a polynomial. The ~~Figure 3-10~~ Table 3-4 shows the specification of *poly* instructions.

Notation	*poly.f* <mem>	
	poly.d <mem>	
Operation	T	← T + Q
	R	← <mem>
	Q	← R × P
	P	← P × S

Table 3-4. Poly Instruction Specification

It frequently occurs in numerical computation that we wish to evaluate a polynomial

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_0 x^n.$$

This polynomial is usually evaluated by using Horner's rule

$$h_n(x) = a_n + x(a_{n-1} + x(a_{n-2} + \cdots + x(a_1 + x a_0) \cdots),$$

since it requires only $O(n)$ operations. However, Hornor's rule requires the result of multiplication as a source of the next addition, and the result of addition as a source of the next multiplication. This nature prevents FLATS2 from exploiting the parallel execution of S and P unit operations, and consequently increases the calculation time, though the number of arithmetic operations are decreased by using Hornor's rule. Therefore, *poly* instruction of FLATS2 evaluates the polynomial in the former way $p_n(x)$. This $p_n(x)$ can be restated as a program of the form

```
      T = 0.0
      do 100 i = 1, n
         sum = sum + a_i × x^i
100   continue
```

which computes the value of the polynomial in N iterations. FLATS2 *poly* instruction implements the loop body of this program in a single instruction, by taking a_i from its memory operand, using T register as *sum*, and updating x^i in P register. This enables the parallel execution of addition and multiplication, thus improving the performance.

Floating-point division is performed by approximating the reciprocal value of the divisor by using iterative multiplications (*convergence division* [49]) with P unit. FLATS2 offers two divide step instructions (*divs (divide start)* and *divl (divide loop)*) which are a kind of combined arithmetic, in which two multiplications are performed in a single instruction cycle. For more details, please refer to "FLATS2 Architecture Handbook"[33].

It is also no use leaving M bus unused, because M bus can transfer data every cycle. If a simple move instruction is executed to transfer data via M bus, S and P unit gets idle during this move operation. On the other hand, if an arithmetic instruction between SP registers is performed, this instruction will leave M bus unused during its execution, which is also a waste of M bus cycle. Thus, it is also effective for full utilization of processor performance to overlap M bus transfer to S or P unit operations. This overlap between M bus transfer and SP arithmetic is usually achieved by specifying memory operands in M format instruction. However, the typical usage of memory operands in M format is restricted to one for read and one for write by reason of pipeline control. For such cases that both of two operands are necessarily to be read from memory, FLATS2 offers the combined instructions that overlap the load operation from M bus to R/U register and the arithmetic operation between S/T/P/Q registers. To use such a combined instruction that reads two memory operands, the corresponding addressing modes (*Read-Read addressing modes*) must be used with the

combined arithmetic instruction.

The *rip (real inner product)* instructions of single and double precision are typical examples of such instruction that overlaps load operations and arithmetic operations between SP registers. The specification of *rip* instructions is shown in Table 3-5.

Notation	*rip.f*	<mem1>, <mem2>	
	rip.d	<mem1>, <mem2>	
Operation	S	←	P + S
	P	←	R × U
	U	←	<mem1>
	R	←	<mem2>

Table 3-5. Rip Instruction Specification

Apparently, *rip* instruction overlaps one addition, one multiplication, and two load operations in a single instruction (single cycle). These *rip* instructions play very important roles in the optimization of various numerical computations as described in later chapters.

3.7.3. Complex Number Processing

FLATS2 offers single- and double-precision *complex number* as primitive data types with hardware supports. A complex number consists of two floating-point numbers; real part and ideal part. The arithmetic on complex numbers therefore involves two or more floating-point arithmetic operations. That is, complex number arithmetic instructions are naturally included in combined arithmetic instructions. Though S unit and P unit of FLATS2 are 64 bit wide each, each of them is designed to work also as two 32 bit wide arithmetic units to support single precision complex number data type. Each FLATS2 instruction can use both S unit and P unit twice in an instruction cycle, thus maximally 4 ALU operations and 4 multiplications (32 bit wide) can be invoked in one complex number arithmetic instruction. Table 3-6 lists primitive instructions of complex number arithmetic supported by FLATS2.

Name	Notation		Comments
move	mov.c	<src1>, <dst>	<src1> → <dst>
add	add3.c	<src1>, <src2>, <dst>	<src1> + <src2> → <dst>
sub	sub3.c	<src1>, <src2>, <dst>	<src1> − <src2> → <dst>
scale	mul3.c	<src1>, <srcd>, <dst>	<src1>.*r* × <srcd> → <dst>.*r* <src1>.*i* × <srcd> → <dst>.*i*
multiply	mov.c	<src1>, P	<src1> → P
	mulc1	<src2>	<src2>.*r* → R.*r* <src2>.*i* → R.*i* P.*r* × R.*r* → P.*r* P.*r* × R.*i* → P.*i* P.*i* × R.*r* → Q.*r* P.*i* × R.*i* → Q.*i*
	mulc2	<dst>	P.*r* − Q.*i* → <dst>.*r* P.*i* + Q.*r* → <dst>.*i*

In the table, *.r* and *.i* stands for the real part and the ideal part, respectively.

Table 3-6. Complex Number Arithmetic Instructions

Multiplication between two complex number is performed by succeeding three instructions, as shown in Table 3-6. The second *mulc1* performs 4 multiplications of single precision floating-point in one instruction.

There are some more complicated complex type arithmetic instructions prepared in FLATS2. The most impressive is maybe *cip (complex inner product)* instructions, which utilizes all of ALU operations, multiplications, and M bus throughput maximally. For example, Table 3-7 shows the specification of *cip.f* instruction (single precision complex number inner product). This instruction loads two memory operands and performs 4 multiplications and 4 ALU operations simultaneously. That is, *cip.f* makes maximal use of FLATS2 performance.

3.8. Virtual Processor Management

Notation	*cip.f*	<mem1>, <mem2>
Operation	P.*r*	← R.*r* × U.*r*
	P.*i*	← R.*i* × U.*r*
	S.*r*	← S.*r* + T.*r*
	S.*i*	← S.*i* + T.*i*
	R.*r*	← <mem1>.*r*
	R.*i*	← <mem1>.*i*
	Q.*r*	← R.*r* × <mem2>.*i*
	Q.*i*	← R.*i* × <mem2>.*i*
	T.*r*	← P.*r* − Q.*i*
	T.*i*	← P.*i* + Q.*r*
	U.*r*	← <mem2>.*r*
	U.*i*	← <mem2>.*i*

Table 3-7. Complex Number Inner Product

3.8.1. Threading

FLATS2 is a cyclic pipeline computer which acts as a memory shared MIMD computer. Virtual processors in CPC shares the hardware in a time-shared manner, thus the following characteristics are presented particularly in FLATS2.

(1) The cost required for process migration between virtual processors is low, because registers (GVM) and memory contents (DM) are shared.

(2) The cost required for inter-processor communication and synchronization is low, because the same pipeline is shared between virtual processors.

FLATS2 thereby accelerates a single task by utilizing cooperating processes sharing memory (*threads* [35, 36]). Of course, it is also possible that each virtual processor would execute an independent job separately. In case a single task can not utilize all virtual processors in a cyclic pipeline, the rest of the virtual processors can be allocated to other tasks.

Threads cooperate each other to accomplish a task, communicating via shared memory (DM in FLATS2). Data area in DM is naturally shared in cyclic pipeline without congestion, therefore a task can be effectively executed by threads in virtual

processors. It is not guaranteed at all that the number of threads are the same as the number of available virtual processors. Threads are mapped onto available virtual processors and cooperate each other in self-scheduling manner.

Each virtual processor of FLATS2 has such internal status as mentioned in this chapter. In normal context switches like external interrupts and traps, all of its internal status must be saved. However, for the context switch between threads, execution environment is basically the same and the switchover is initiated by user's program, therefore some of internal status can be assumed unnecessary to save and to restore. Consequently, only PC, CFP, and little information* is enough to switch thread in FLATS2, which takes only a few cycles.

3.8.2. Mutual Exclusive Memory Access

FLATS2 offers only the most primitive methods as hardware features for communication and synchronization between virtual processors. Each software can actually use more elegant methods for the communication and synchronization by implementing them in software with primitive methods offered by hardware. Primitive methods are enough for FLATS2, because the current FLATS2 implementation has only two virtual processors, in which there is no absolute necessity to implement complicated methods particularly. Actually implemented in FLATS2 are simple but generally applicable methods for many virtual processors regardless of the current implementation of FLATS2.

For the cooperation of virtual processors, some method is essential to access memory in mutual exclusive manner. FLATS2 offers *LAS (Load And Store) instruction*, which is a slight extension of *TS (Test and Set) instruction* of IBM System/370 [50]. TS instruction of System/370 tests the lowermost bit of a byte on memory to set

* Such as data stack pointer in GV register. Stack operations are available on any GV registers in FLATS2 addressing modes.

condition code, and then sets all bits of the memory byte to one, while protecting the byte from the accesses of other CPUs during TS operation. That is, TS can handle only one bit information on memory. LAS instruction of FLATS2 loads a word from memory to a GV register and stores a word from a GV register to the same address as the source operand of load operation, preventing other virtual processors from accessing that address during LAS operation. LAS is superior a little to TS in the point that LAS can handle a whole word, thereby it can handle a *queue* or *list* entry directly.

Functionally LAS is a swap operation between DM and GVM, but it is more than a simple swap in having source register and destination register separately. LAS instruction is a K format instruction in which BL addressing is used to access memory. The specification of LAS is as follows.

Notation	*las* <src_reg>, <memory>, <dst_reg>		
	las,j <src_reg>, <memory>, <dst_reg>, <branch>		
Operation	<memory> → <dst_reg>	...	Load and
	<src_reg> → <memory>	...	Store

Usually a mutual exclusive memory access completely blocks other processors to access the memory until the exclusive access ends. However, in FLATS2 implementation, a mutual exclusive access of a virtual processor does not affect the performance of other virtual processors at all, even if other virtual processor would access the same memory address. This trick of pipeline implementation is described in later chapter. By using LAS instruction (or TS instruction), more elaborate methods can be prepared to handle inter-processor communication or synchronization, such as *semaphore*[51] or *queue handling*[38].

3.8.3. Interprocessor Interrupt

To send information among virtual processors asynchronously, FLATS2 offers *inter-processor interrupt* facility. There are two kinds of inter-processor interrupt in

FLATS2. One is *maskable* and the other is *non-maskable*, each of which has the corresponding trap entry in resident area on memory. If a virtual processor is interrupt disabled,† a maskable inter-processor interrupt is left pending until interrupt is enabled in that virtual processor (or the interrupt request is cleared). A non-maskable interrupt is always accepted immediately regardless of interrupt mask bit. The handling procedure of an inter-processor interrupt is all the same as that of an external interrupt, except that an external interrupt is accepted by only one virtual processor, on the other hand an inter-processor interrupt initiates all virtual processors into the corresponding trap handling.

Inter-processor interrupt is initiated when one of virtual processors in the cyclic pipeline invoked the corresponding privileged instruction (inter-processor interrupt request instruction). The specifications of inter-processor interrupt instructions are as follows.

Notation *ipint*
ipnmi

Operation No operand. Privileged instruction.‡
Requests an inter-processor interrupt
which is maskable (*ipint*) or non-maskable (*ipnmi*).

In FLATS2, the request of inter-processor interrupts can be informed to every virtual processors within one instruction cycle. This is one of the advantages of cyclic pipeline architecture, in which all virtual processors share a single hardware. The implementation in FLATS2 pipeline will be described later.

† There is an *interrupt mask bit* in PS. If this bit is zero, interrupt is enabled. Otherwise, interrupt is disabled (masked).

‡ A privileged instruction is executable only by the virtual processor in privileged status. If a privileged instruction is invoked in user status, the virtual processor is trapped by a *privilege violation trap*.

3.9. Discussion

3.9.1. BL addressing and Unified Vector/Scalar

One of the major applications of BL addressing is array processing. For array processing, *vector architecture* has been often adopted and practiced. However, vector architecture suits for the applications that perform a uniform operation on many (or every) array elements. Contrary, BL addressing is basically a scalar architecture, which is applicable to wider varieties of processing. Thus BL addressing applies to non-uniform processing which does not consist of loop structure, such as symbol manipulation. Even if dependencies are present between the array elements which are accessed in a loop, no special tricks are required in the program, because BL addressing is an extension of scalar architecture. BL addressing directly handles data on memory without using vector registers. Therefore, the same procedure is applicable regardless of the length of the array. Though it is more difficult in scalar architecture (e.g. BL addressing) to fully exploit the data parallelism which is available in vector architecture, such parallelism has been already exploited in FLATS2 by implementing virtual processors of CPA.

Jouppi, et al.[52] attempted to integrate scalar processing and vector processing in *Unified Vector/Scalar Floating-Point Architecture* of Multi-Titan. In this architecture, the format of FPU ALU instructions includes 4 bit vector length field. Scalar processing is realized by setting the vector length to one. FPU register file is shared between vector and scalar instructions, thus the register interlocking mechanism is also shared. Unifying vector processing and scalar processing, vector instructions are implemented by the same hardware as scalar instructions. Consequently, more applications are to get vectorizable and more performance improvement can be expected in many applications.

Unified vector/scalar and BL addressing have at least one common characteristic. They both accelerate array processing by using enhanced scalar instruction set.

However, the implementation is totally different each other. Unified vector/scalar approach (UVS) is only applicable to uniform processing of array elements, because FPU ALU instructions of Multi-Titan have adopted vector architecture after all. The operands are limited to be FPU registers in UVS, thus each instruction can handle at most 16 array elements. The length of operations are statically designated by the vector length field of the instruction. The accesses to not contiguous addresses are unable to be processed in a vector processing manner. On the other hand, BL addressing is applicable to non-vectorizable jobs such as Lisp. The length of array is not limited, because BL addressing handles data on memory. Briefly saying, Unified Vector/Scalar is a vector architecture which is improved for scalar operations. Contrary BL addressing is a scalar architecture enhanced for array handling including vector processing.

3.9.2. INDEX and ACB instruction

INDEX (Index Calculation) instruction of VAX-11 [38] checks the result of arithmetic against the given range. This instruction is intended to check array subscript against the defined range at run-time. INDEX instruction thus issues an exception or trap, in case the value is out of the specified range, to pass the control to privileged software. On the other hand, BL addressing can either branch to user's handler or issue a trap, according to user's preference. This branch on out-of-range facility enables users to control various kinds of processing from loop optimization to run-time type check in LISP [53].

ACB (Add, Compare, and Branch) instruction of VAX-11 performs arithmetic, compare, and conditional branch in one instruction. However, it performs neither address calculation nor memory access. FLATS2 adopts the instruction set of 2 memory operands to perform memory accesses and arithmetics on operands in one instruction, consequently specifying more operations in each instruction than a sole ACB instruction.

In cyclic pipeline computer, the pipeline hardware is shared among virtual processors in a time-sharing manner. Therefore it is preferable directing many operations statically in each instruction to overlapping instructions dynamically in execution time. In that sense, BL addressing suits for cyclic pipeline architecture. Dynamic control of instruction overlaps involves the increased internal status of each instruction stream, while the quantity of internal status of each virtual processor directly affects the total cost of hardware in CPA. As mentioned in chapter 2, CPA attempts some advanced control to improve scalar performance. However, the scheduling of each virtual processor is always well defined in the pipeline to retain the quantity of the internal status of each virtual processor.

3.9.3. VLIW and FLATS2 instruction set

VLIW (Very Long Instruction Word architecture [54] seems to have a common point to FLATS2, in the sense that both specify many operations in each instruction. VLIW uses a very long instruction format (typically 512 bit) to utilize the large numbers of data paths and functional units. Each field of VLIW's instruction directly and independently controls each functional unit in every cycle. A technique called trace scheduling[55, 56] is used to exploit the parallelism of a program by scheduling multiple functional units. VLIWs can optimize the program beyond the limit of basic blocks and can combine arithmetic by using the trace scheduling technique, while vector architecture can only deal with relatively simple and regular innermost loops. VLIW approach aims to exploit maximum parallelism in the program to achieve higher peak performance by using multiple functional units and data paths, while trying to utilize these rich hardware fully in more general cases by trace scheduling. VLIW has a possibility to exploit the maximum parallelism in a program, achieving high peak performance. However, if the program does not have enough parallelism in it (or the compiler cannot optimize enough to exploit parallelism), many of arithmetic units and data paths can be left unused. In fact, Sohi and Vajapeyam [57] has reported that

perfectly horizontal architectures are sometimes overkill and the restricted architecture can achieve almost the same performance. This means that the performance per cost of VLIW is not always good.

VLIW assumes many register files corresponding each of many arithmetic units, which means the internal status of the processor is very big. By using this large quantity of registers, a VLIW performs many operations in parallel to exploit maximal parallelism. However, this characteristic is disadvantageous for CPA. To share the hardware among virtual processors, internal status of each virtual processor should be little. Therefore, FLATS2 instruction supports the limited combinations of operations which often appear in programs, with limited quantity of internal status (registers) and less hardware cost than typical VLIWs. The parallelism which can be exploited by FLATS2 instruction is regarded to be less than VLIWs, but the benchmark results show that there is not so much difference between the performance of FLATS2 and VLIW which has much more hardware and almost the same cycle time [58]. It is another merit of FLATS2 that the utilization of instruction set is easier than that of VLIWs, because FLATS2 instruction is a natural extension of usual architectures with a single opcode and addressing modes. In most cases, such limited parallelism of operations as FLATS2 is enough.

Chapter 4

Programming on FLATS2

4. Programming on FLATS2

4.1. Array Processing by BL Addressing

In many array processing programs, the array elements are accessed orderly with a fixed stride in a loop structure. After accessing the last element, the processor escapes from the loop. In FLATS2, branch-on-range-check facility of BL addressing can be utilized for such array processing. Figure 4-1 illustrates the policy to apply BL addressing to array processing.

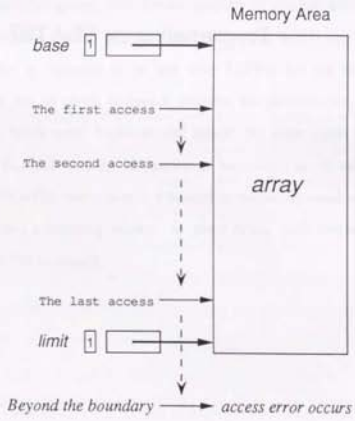


Fig. 4-1. BL Addressing for Array Processing

Here, BL pair points to the base and limit addresses of the array. The elements of the

array is accessed one after another, modifying the effective address each time. After accessing the last element, the effective address goes out of the range, consequently a memory access error occurs at the next access to change the control flow hereafter.

Let's see a simple example written in FORTRAN (Program 4-1). This piece of code takes the sum of array elements.

```
      sum = 0.0
      do  100  i = 1, 10
         sum = sum + array(i)
100   continue
```

Program 4-1. Summation of Array Elements

Here, let *sum* be a *real*† variable, and *array* be a *real* array of 10 elements. This program can be converted into the FLATS2 instruction sequence shown in Code 4-1.

```
      mov.f     #0f0.0, S
      lea       dseg@_array, vr2
      lea       dseg@_array+36, vr3
      movw      vr2, vr0
L001: add3.f.j  vr2:<4(vr0), S, S, L001
      mov.f     S, dseg@_sum
```

Code 4-1. Summation of Array Elements

The first *mov.f* initializes S register to zero. The second *lea* instruction loads the effective address of the first element (*array(0)*) to vr2, which is used as base register at *L001*. In the code, *dseg*‡ appears three times as base register, which is the alias of a

† Single precision floating-point data type (i.e. *real*) is 4 byte long in FLATS2.

‡ This *dseg* register is set up by system program before the user program is invoked.

GR that specifies user's whole data segment. However, this specification can be omitted, because *dseg* is assumed as default wherever base register is not specified. In the following examples, *dseg* is omitted in principle. The next *lea* similarly loads the effective address of the last element (*array(9)*) to vr3. Vr3 is automatically used as the limit register at *L001*, because vr3 is the binary pair of the base register (vr2). The *movw* instruction moves a word from vr2 to vr0. Here, the address of the first element (vr2) is copied to the pointer register (vr0), which is used in the loop. The body of *do* statement is integrated into a single instruction (*add3.f.j*). This *add3.f.j* instruction is an M format instruction, which performs the following operations.

(1) It first calculates an effective address to fetch memory operand. The effective address is the value of vr0 in this addressing mode (pointer pop mode).

(2) The access address is checked against the range specified by BL pair (vr2 and vr3). If an access error occurs here, this instruction is aborted, consequently proceeding to the next instruction.

(3) The operand (an array element) is fetched and added to S register.

(4) The side-effect of addressing is performed. According to the addressing mode, a new address is calculated to modify the register. In this example, the specified displacement (i.e. *4*) is added to the pointer register (vr0). The calculated address is checked against the BL pair before written into the register. If the new address is out of the range, the address tag is cleared.

(5) Branch is taken to the target specified by *j0* field, that is, to the label *L001*. In this case the same instruction is repeated, because *L001* designates the current instruction itself (*j0* is zero).

The array elements are thus summed up into S register, beginning from *array(0)* to *array(9)*. When *array(9)* is added, vr0 is modified to the address of *array(10)*, yet with its address tag cleared because it is out of the range. Then *add3.f.j* is again executed, but this time the address tag of vr0 is cleared, consequently it is aborted at the

range check, escaping the loop to the next instruction. Table 4-1 shows the register contents at the end of each iteration.

Iteration Count	Address tag of vr0	Content of vr0	Content of S register	Comment
0	1	_array	0.0	before L001
1	1	_array+4	array(0)	
2	1	_array+8	array(0)+array(1)	
3	1	_array+12	array(0)+...+array(2)	
4	1	_array+16	array(0)+...+array(3)	
5	1	_array+20	array(0)+...+array(4)	
6	1	_array+24	array(0)+...+array(5)	
7	1	_array+28	array(0)+...+array(6)	
8	1	_array+32	array(0)+...+array(7)	
9	1	_array+36	array(0)+...+array(8)	
10	0	_array+40	array(0)+...+array(9)	
11	0	_array+40	array(0)+...+array(9)	exit from iteration

Table 4-1. Register Contents in Program 4-1

The last *movf* instruction moves the calculated sum from S register to memory variable (*sum*). In this *movf* instruction, there is no branch target and no base register specified. In such a case, the default branch target is set to *j0*, that is, *j0* points to the next instruction. Also, the default BL (*dseg*) is assumed by assembler.

As shown, the loop to sum up the elements consists of a single instruction, which is executed every instruction cycle. Consequently, one floating-point addition is performed every cycle in the loop without incurring the overhead of loop termination test. This is the maximum performance expected in the loop. Actually the initialization and termination overhead is accumulated to the execution time of loop, therefore the overall performance is degraded from the ideal performance. In Code 4-1, four instructions are executed to prepare the loop (*L001*), and one instruction is executed for termination. Decreasing these loop setup overhead, better performance is observed overall.

Here, I would like to introduce a measure called "*arithmetic density*" to evaluate the efficiency of numeric calculation. Arithmetic density is defined as the number of floating-point arithmetic operations per instruction cycle. Obviously, arithmetic density can be converted to arithmetic performance by multiplying it by instruction cycle time. The expected arithmetic density is 1 within the loop of Code 4-1. However, the overall arithmetic density is 0.625, because 10 floating-point operations are executed in 16 instruction cycles including initial and terminal overhead. To obtain more arithmetic density, the setup time of the loop must be reduced.

Code 4-1 can be improved slightly more with some other facilities of FLATS2. Code 4-1a shows the improved version of code sequence.

```
        mov.f     #0f0.0, S
        mkbl      @_array, @_array+36, vr2
L001:   add3.f.j  vr2@<4, S, S, L001
        mov.f     S, @_sum
```

Code 4-1a. Summation of Array Elements (improved)

This sequence only contain 4 instructions. First, two *lea* instructions are integrated into a single *mkbl (make BL)* instruction.* This instruction calculates two effective addresses in one instruction (one instruction cycle), writing the results into a binary pair of GV registers. Second, the addressing mode for *add3.f.j* instruction is changed to *offset pop* mode, consequently the initial setup of pointer register is omitted. The resulted overall arithmetic density is 0.769, 10 *FLOP*s (*floating-point operations*) per 13 cycles.

* FLATS2 can calculate two effective addresses simultaneously, because FLATS2 instruction is designed to take maximally 2 memory operands. *Mkbl* instruction is a K format instruction, in which three source register and two displacements is available for two address calculations.

The other way to improve arithmetic density is to perform more arithmetic operations in each cycle. As mentioned in chapter 3, FLATS2 has arithmetic instructions to deal with complex number. A complex number consists of two single precision floating-point numbers. A complex addition instruction therefore performs two floating-point additions in one instruction cycle. By using this, the peak arithmetic density within the loop can be improved up to 2.

```
        mov.c     #0f0.0#0f0.0, S
        mkbl      @_array, @_array+36, vr2
L001:   add3.c.j  vr2@<8, S, S, L001
        mov.c     S, temp(sp)
        mov.f     temp(sp), S
        add3.f    temp+4(sp), S, @_sum
```

Code 4-1b. Summation of Array Elements (complex)

Code 4-1b is the code sequence with complex instructions. The first *mov.c* instruction initialize S register to zero as a complex number. The loop section is implemented by *add3.c.j* instruction, which performs a complex addition (two single float additions) in one cycle. Executing L001 5 times, 10 array elements are accumulated in S register as one complex number, in which each of real and ideal part accumulates 5 elements. At the sixth time, *add3.c.j* aborts into the termination procedure to add the real part and the ideal part of S register together. First, the complex sum is stored into a temporary area on stack† (*temp(sp)*), then the real part is again loaded to S register as a single precision floating-point number. The last *add3.f* instruction adds the ideal part of *temp(sp)* and S register to store the final result to *sum*. Code 4-1b is thus executed in 11 cycles. The number of executed operations is 11 in this code, because the initial value of S register is duplicated into the real part and the ideal part. Excluding the

† Stack pointer (*sp*) is the alias of a GR defined by software protocol.

meanless operation, the resulted arithmetic density is 0.909, 10 FLOP per 11 cycles. Code 4-1b has more termination overhead than Code 4-1a, while having more arithmetic density in the loop.

Code 4-1, 4-1a and 4-1b reflect the loop structure of Program 4-1 directly, but the loop structure is not necessary to take the sum of *array*, because the iteration count of Program 4-1 is a small constant (i.e. *10*). In such a case, in-line expansion of loop is effective. Abolishing the loop structure of the program (Program 4-1), the sum of *array* can be calculated into *sum* in only 7 cycles. The overall arithmetic density of Code 4-1c is 1.29, 9 FLOPs per 7 cycles.

```
mov.c    @_array, S
add3.c   @_array+8, S, S
add3.c   @_array+16, S, S
add3.c   @_array+24, S, S
add3.c   @_array+32, S, temp(sp)
mov.f    temp(sp), S
add3.f   temp+4(sp), S, @_sum
```

Code 4-1c. Summation of Array Elements (expanded)

Though Program 4-1 was a simple loop that has a fixed limit and fixed step, the policy shown in Code 4-1, 4-1a and 4-1b is also applicable to such loops that has variable limit or variable step.

```
      do  100  i = k, m, n
          sum = sum + array(i)
100   continue
```

Program 4-2. Summation of Array Elements (2)

Program 4-2 has both limit and step as variables in *do* statement. Even in this case, FLATS2 can implement this *do* statement with the loop which consists of a single instruction, by calculating the addresses of *array(k)* and *array(m)* before entering the loop and by using *index pop* mode in the loop. Though the setup procedure of Program 4-2 involves a little more overhead than that of Program 4-1, this overhead is not so serious in FLATS2.

```
        asl3.l    @_k, #2, vr4
        asl3.l    @_m, #2, vr5
        asl3.l    @_n, #2, vr6
        mkbl      @_array-4(vr4), @_array-4(vr5), vr2
L001:   add3.f.j  vr2@<vr6, S, S, L001
        mov.f     S, @_sum
```

Code 4-2. Summation of Array Elements (2)

Code 4-2 shows the code sequence which corresponds to the loop body of Program 4-2.† Only usual floating-point arithmetic is used in Code 4-2, that is, the Code 4-2 of Program 4-2 corresponds to the Code 4-1a of Program 4-1. The last *mov.f* instructions is the same as Code 4-1a. Three *asl3.l* (*arithmetic shift left* for long word) instructions are inserted to deal with three variables (*k, m, n*) to prepare for the indexing of the array. The fourth instruction (*mkbl*) calculates the base (*array(k)*) and the limit (*array(m)*) by using *offset index* addressing mode. The loop at *L001* sums the array elements up executing one addition every cycle, thus the peak arithmetic density is still 1. The initial overhead is 7 cycles, because each *asl3.l* takes 2 cycles in the current FLATS2. Note that this code works well regardless of the sign of *k, m, n* or the direction of access.

† Before this part is executed, the relation among *k, m, and n* must be checked to determine whether loop body is executed or not, according to the specification of *do* statement of FORTRAN.

The complex addition instruction is applicable also here as well as Program 4-1. By checking the iteration count is even or odd outside of the loop, (and by adjusting the odd case,) the loop body of Program 4-2 can be improved as well as Program 4-1. This involves slightly more overhead for loop setup, but it is not serious and can be ignored in case of many repetitions.

4.2. Other Techniques for Loop Optimization

By using BL addressing, loops can be optimized in such array processing that the elements are accessed orderly. Preparing the addresses of the first and the last access in a BL register pair, loop termination test can be integrated into array element access regardless of constant or variable stride. The performance of small loop is greatly improved by BL addressing, because the overhead of loop termination test is comparable to the number of arithmetic operations performed in a small loop.

However, it is not always possible to transform a program to use BL addressing instead of indexing by using loop counter. In such cases that the loop counter itself must be used in the loop body or in loop termination check, various *compare and branch* instructions of FLATS2 can be utilized. In particular, *add, compare, and branch* instruction (*acb* or *acj*) is frequently used to implement loop structure. It increments loop counter, tests termination condition, and then branches to execute the loop body again. Even if arithmetic is unnecessary or inapplicable at the loop terminal, various *compare and branch* instructions are available to test condition to make a branch. All these instructions are implemented by using mostly the same hardware as BL addressing, thereby the dedicated cost is cut down. Another distinguishing point is the 4 way branch facility. This multi-way branch of FLATS2 is beneficial to some purposes such as the implementation of *arithmetic IF* and *computed GOTO* statements of FORTRAN, tag dispatch in LISP[53], and case statement implementation in C language[59].

A FLATS2 instruction can perform maximally the following kinds of works together: operations in GVU, compare, branch, two memory accesses, and operations in SPU. To accelerate the job in FLATS2, it is important to pack as many functions as possible into each instructions. The most typical example is the M format instruction in a small loop, which includes all of these functions. K format instructions such as *lea*, *mkbl*, and *las* include most functions besides SP operation. J format instructions (e.g. *acb* and *cb*) perform compare, branch, and sometimes GVU operations. Even if instructions usually do not include so many functions, the succeeding instructions are sometimes able to be combined by *peep-hole* optimization.

Unconditional branches are particularly easy to be combined with other instructions. If there is a K or M format instruction without branch target specification, unconditional branch can be combined to that instruction by using the default BL pair and the target of unconditional branch together. In this case, it is regarded certain that the preceding K or M format instruction will succeed, and therefore the branch will be taken. The execution time of the unconditional branch is practically omitted by overlapped to the preceding operation.

```
mov.l   gr0, @_var          ; a M format instruction
jump    L001                ; an unconditional branch
                    ↓
mov.l.j gr0, @_var, L001    ; combined into one mov.l.j
trap    #access_error       ; can not reach here
```

Code 4-3. An Example of Branch Optimization

Code 4-3 shows an example of this peep-hole optimization. An unconditional branch to L001 is combined to the preceding M format instruction. The access to *var* is assumed to succeed by using the whole data segment as BL pair, therefore the branch

to L001 is always taken in *mov.l.j* instruction. The original *mov.l* instruction issues an *address calculation trap* in case of access error. To trap such a case even in the optimized code, a *trap* instruction‡ can be inserted just after the *mov.l.j* to ensure safety execution of the code as before.

Many other techniques are developed and implemented in FLATS2 FORTRAN [60] and the GNU C compiler retargeted for FLATS2 [59]. For details, please refer to these reports.

4.3. Combined Arithmetic

There are several combined arithmetic instructions supported in FLATS2. Here, some typical pieces of codes are presented for readers' interests. For more details, please refer to *FLATS2 Architecture Handbook* [33].

4.3.1. Inner Product Calculation

One of the most typical combined arithmetic instructions may be *inner product* instructions. In inner product instructions, multiplication and addition are performed in parallel, together with two overlapped load operations. Program 4-3 shows the innermost loop of a BLAS routine called *ddot*, which stands for *double precision dot product*. This *ddot* routine has two innermost loops, one of which is for constant stride, the other is for variable stride. Though only the code of the latter *do* loop body is given here for the simplicity, the former loop can be coded similarly with *index* class addressing modes and some additional initial overhead. Code 4-3 shows the FLATS2 assembler code corresponding to the latter *do* loop of Program 4-3. The loop body is integrated into a single *rip* instruction (*L11*) which performs two loads, one add, and one multiply every cycle.

‡ *Trap* instruction issues an exception to pass the control to trap handling routine. *Trap* instruction is mainly used to request the services of privileged software.


```
c     code for unequal increments or equal increments
c     not equal to 1
                    do 10 i = 1,n
                       dtemp = dtemp + dx(ix)*dy(iy)
                       ix = ix + incx
                       iy = iy + incy
           10       continue
                    ddot = dtemp
                    return

c     code for both increments equal to 1
                    do 30 i = 1,n
                       dtemp = dtemp + dx(i)*dy(i)
           30       continue
                    ddot = dtemp
                    return
```

Program 4-3. Double Dot Product (ddot)

```
        asl3.l    vr6,#3,vr0              ; vr8 = dx ; vr10 = dy
        addr      vr0,#-8,vr0             ; vr0 = (n-1)*8
        lea       dseg:(vr8)vr0,vr9       ; vr9  = &dx[n-1]
        lea       dseg:(vr10)vr0,vr11     ; vr11 = &dy[n-1]
        mov.d     #0d0, S                 ; initializing S register
        mov.d     (vr9),P                 ; initializing P register (1)
        mul.d     (vr11),P                ; initializing P register (2)
        mov.d     vr8@<8, R               ; initializing R register
        mov.d     vr10@<8, U              ; initializing U register
L11:    rip.d.j   vr8@<8, vr10@<8, L11    ; self loop of rip
        add.d     P, S                    ; terminating the loop
L1:     movw      fp,sp                   ; restore stack pointer
        ret                               ; return to caller
```

Code 4-3. Double Dot Product (ddot)

4.3.2. FFT (Fast Fourier Transform)

The innermost loop of *Fast Fourier Transform (FFT)* is typically programmed as Program 4-4 [61].

```
      do  10  i  =  k, kmax
          j      =  i      +   ngrp
          p      =  w      *   z(j)
          z(j)   =  z(i)   -   p
  10      z(i)   =  z(i)   +   p
```

Program 4-4. Fast Fourier Transform

Here, *w*, *p*, and the array *z* are all complex numbers. Though there are some variations in indices of *z* [62], the sequence of arithmetic operations is all the same. This small loop is coded into 2-cycle loop (Code 4-4) by using dedicated combined instructions *cfft1* and *cfft2*. The loop of Program 4-4 is implemented in FLATS2 by the code shown in Code 4-4.

```
iloop1:  cfft1.j   vr10@<vr22, vr6@<vr23, iloop2
           :
           :       (loop exit part - omitted)
           :
iloop2:  cfft2.j   vr8@<vr22, vr4@<vr23, iloop1
```

Code 4-4. Fast Fourier Transform

This code was taken from the actual hand-coded fft program on FLATS2. The loop consists of *cfft1* instruction at *iloop1* and *cfft2* instruction at *iloop2*, where *vr10@<vr22* represents the address of *z(j)*, *vr6@<vr23* represents the address of *z(j–ngrp)*, *vr8@<vr22* represents the address of *z(i)*, and *vr4@<vr23* represents the address of *z(i–ngrp)*. In the loop, *cfft1* and *cfft2* instructions are executed alternately. These two instructions perform one complex multiplication, one complex addition, and one complex subtraction. That are, 4 multiplications and 6 ALU operations of single precision floating-point are performed in every two instruction cycles.

4.4. Symbolic Manipulation

LISP is one of the most popular language for symbolic manipulation applications such as natural language processing, artificial intelligence, and formula manipulation. Address tag and BL addressing of FLATS2 can be utilized not only in array processing but also in such symbolic manipulation applications. The outline of *Common LISP* [63] implementation for FLATS2 is described in the reference [53]. This section describes the architectural supports of FLATS2 for symbolic manipulation in its instruction set architecture. For the details of language implementation, please refer to the above paper [53].

4.4.1. Type Check

LISP datum consists of type and entity (or the pointer to entity). The actual operation of a LISP function varies according to the types of operands, therefore the types of operands must be checked before every operations. These checks are serious overhead in execution, thus many LISP machines take tags into their architecture to assist run-time data type checking. For example, Riken FLATS [64, 45, 46] has 5 bit tag, and Symbolics 3600 [39] has 2 bit major tag and 4 bit minor tag. These tags represent the type of datum, which is interpreted by hardware in performing operation.

On the other hand, FLATS2 has only a single bit address tag supported by hardware. This means that FLATS2 supports only two data types by hardware; *address* and *number*. As mentioned in chapter 3, FLATS2 can check the address tags in address calculations. Also in GV arithmetic instructions of I format, FLATS2 can check the address tags of operands.‡ If address type operands appear in operands of

‡ *Md* field specifies whether address tags would be checked or not, together with operand

arithmetic, FLATS2 issues a *LISP Service Trap* and passes the control to the trap handler routine. Such run-time address tag check facility enables FLATS2 to emulate generic operations in LISP.

One bit address tag of FLATS2 is not enough to represent all data types required for applications. Therefore, the data part of a word must be used to represent the type of the operand. Usually, a few bits in the word (typically uppermost or lowermost part) are used as tag bits: e.g. Utilisp implementation by Chikayama [65]. Such in-word tag implementation is equivalent to partitioning of memory space. FLATS2 has 4-way compare and branch instructions of J format, which can efficiently dispatch the code according to two bit in-word tag. This approach assumes the fixed partitioning of memory space to data types.

The range check facility of BL addressing can be utilized to support more data types with run-time type checking in FLATS2. Gathering the objects of the same type in a contiguous area, the type can be represented by the corresponding BL pair. By using this BL pair in an operation, the operand is confirmed to belong to a certain type in parallel with the operation itself. This characteristic can be utilized for run time type checking. The operation is first tried with the most probable BL pair. If this BL pair includes the operand, the tag of the operand is checked in parallel with the operation, consequently incurring no overhead for tag check. If this BL pair does not include the operand, i.e., if the operand does not belong to the assumed type, the access error occurs to check and to handle the operand adequately. In this case, the operand type must be investigated here to handle it appropriately. The default type of operand is often apparent from the source program. For example, *car* and *cdr* operations assume a cell as their operands, therefore the BL of list area should be used as the default BL in *car* and *cdr* operation. Using appropriate default BL for the operation, type check is integrated into BL addressing in most cases, which alleviates the overhead of run-time type check in symbolic manipulation applications. One of other merits of this approach is that types can be arbitrarily defined by software. On the other hand, types defined by in-word tag have less flexibility, especially in case tags are supported by hardware.

selection. FLATS2 can either check or ignore address tags in arithmetics according to the *md* field.

4.4.2. Memory Allocation

In LISP and other symbolic manipulation languages, program dynamically consumes free storage to allocate objects. When running short of free storage, *garbage collection* procedure is invoked to reclaim the storage from the objects out of use. FLATS2 manages such memory allocation by using BL addressing.

FLATS2 offers *alloc (allocate storage)* instruction as a primitive to allocate memory area to an object. The following is the specification of *alloc* instruction.

Notation	*alloc* <base>, <src1>, <src2>, #<imm>, <dst> *alloc.j* <base>, <src1>, <src2>, #<imm>, <dst>, <branch>
Operation	<src1> → memory(<base>) <src2> → memoy(<base> + 4) <base> → <dst> <base> : <base> + #<imm> → <base>

Alloc is a K format instruction, which performs address calculation to store data with dedicated side effects. This instruction can be used to support many kinds of *linked list* structures including *cons* and *make vector*. The *cons* implementation with this *alloc* instruction is described in the abovementioned paper [53].

4.4.3. Bit Instructions

If free storage runs short in memory allocation, disused objects must be reclaimed to make more free storage (*garbage collection*). To collect free cells together for BL addressing (for *alloc* instruction), compactifying garbage collection is required. The garbage collection of FLATS2 is to base on the algorithm adopted in Riken FLATS,

which is described in [64]. For this compactifying garbage collection, *bit operations* are effective. FLATS2 offers some bit manipulating instructions as well as Riken FLATS, though FLATS2 has more simple instructions than FLATS. FLATS2 bit instructions include the following.

Bit set

sets the specified bit of a word to one.

Bit clear

clears the specified bit of a word to zero.

Bit change

negates the specified bit of a word from zero to one or from one to zero.

Bit test

tests the specified bit of a word and sets the condition code accordingly.

Bit count

counts the number of the bit which value is one in the specified word (in 32 bit excluding address tag) and stores the result to a register.

Bit reverse

reverses the bit position of the specified word (32 bit excluding address tag), which means the i-th bit of source operand is moved to the $(31-i)$-th bit of destination.

Bit find

finds the first bit set, searching from the lowermost bit of source operand word, and then stores the bit position to destination operand. If no bit is set to one, *zero* flag of condition code is set.

The bit manipulation instructions of FLATS2 is rather simple compared to the bit instructions of Riken FLATS or DEC VAXen. FLATS or VAX can handle bit string, in which start bit position, size, and operation direction can be specified. On the other

hand, FLATS2 only handles bits in a fixed length, in a fixed direction, and in a fixed manner. However, FLATS2 performs all these bit operations in a single instruction cycle by simplifying the function of each instruction. Using together with other instructions like various *shift* and logical operations, any other complicated operations can be emulated. Usually simple function is enough for bit manipulation, and fast operation is preferable to complicated and slow operations.

4.4.4. Multiple Precision Integer

Arbitrary precision integer (*bignum*) is one of the most distinguishing features of LISP. The performance of bignum is important especially for such applications as formula manipulation, which frequently has to deal with huge coefficients in expanding and handling formulae. FLATS2 supports primitive instructions for addition, subtraction, and multiplication of multiple precision integer data. The division of multiple precision integer is too complicated to be supported by hardware, therefore it is emulated by software. With fast addition, subtraction, multiplication, and shift operations of multiple precision integer, the performance of division is expected to be improved [66].

The memory representation of multiple precision integer on FLATS2 is defined to be a natural extension of signed integer in 2's complement representation. A multiple precision integer is an array of words, each word of which is processed by iterating primitive instructions, using BL addressing just like usual numerical array processing.

Chapter 5

The Implementation of FLATS2

5. The Implementation of FLATS2

5.1. Pipelining of Instruction

As mentioned in chapter 2, each virtual processor of cyclic pipeline is statically scheduled in the pipeline. An instruction in the pipeline never stops to wait, but just continues to flow in the pipeline in accordance with the predefined time schedule. Each virtual processor must evacuate the pipeline stage every cycle for the use of the next virtual processor. Provide that a virtual processor could not execute an instruction because a necessary resource is reserved for the use of other instruction. This virtual processor has to wait for the hardware resource gets released. Consequently, it just wastes this instruction cycle, and retries the pending instruction at the next instruction cycle assigned to this virtual processor. In a cyclic pipeline, each instruction is processed in the template of pipeline. This template is assigned to each virtual processor once an instruction cycle, and the assigned template is never shared between virtual processors. Even if an instruction uses only a limited portion of the template, the rest of the template is never used by other virtual processor. The template includes one memory read and one memory write with GVU and SPU operation slots, which are enough to execute one M format instruction with two memory operands (one for read, one for write) with side effects. Usual instruction is executed in one template, though some portions of which can be left unused. A certain kind of instructions requires two contiguous templates, consequently taking two instruction cycles for execution. The *read-read-addressing modes* requires a dedicated form of template, but it still resides in one virtual processor.

Figure 5-1 illustrates the template of FLATS2 pipeline. The instruction of FLATS2 is executed by 10 staged pipeline. Let the logic of the *i*-th stage be L*i* in the following explanation. Each stage of the template works as follows:

L0 The instruction is fetched from IM to IPU.

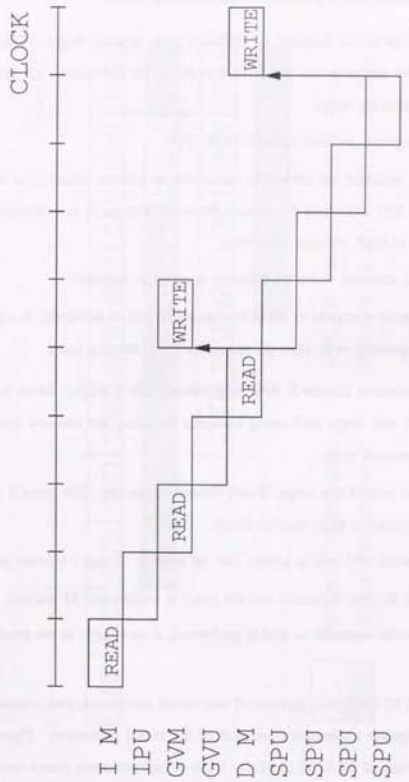


Fig. 5-1. Template of FLATS2 Pipeline

L1 The instruction is decoded for the following stages.

The instruction decoder is pipelined into several stages. This first stage of decoder performs the primitive decode of the instruction, for more decoding in the following stages.

L2 GV registers are read from GVM to GVU.

L3 GVU performs the arithmetic operations or address calculations on GV registers, while IPU calculates the branch target (if necessary) and determine the next PC value to fetch the next instruction.

L4 DM is accessed to fetch a memory operand (if necessary).

L5 The write operation to GVM is also performed, if necessary, to store the result of GV operation or to store the side effect of addressing mode.

As shown in chapter 3, SPU is pipelined with 4 stages. From this stage (L5), S and P unit begin performing operands including the memory operand fetched in the previous stage.

L6 At the end of this stage, S unit finishes operation. The latter 2 stages of S unit are dummy to align itself to P unit.

L7 The result of S unit is written into SP register. P unit continues processing.

L8 P unit finishes arithmetic and the result is written into SP register.

L9 The write operation to DM is performed, if necessary, as the result of the instruction.

FLATS2 is a cyclic pipeline of two virtual processors, thus instructions are issued into the pipeline alternately from each of the virtual processors. Figure 5-2 illustrates the time chart of FLATS2 pipeline. Each virtual processor issues one instruction into the pipeline every 4 machine cycles. Therefore, the time for 4 machine cycles is called 1 *instruction cycle*. Shared by two virtual processors, the cyclic pipeline of

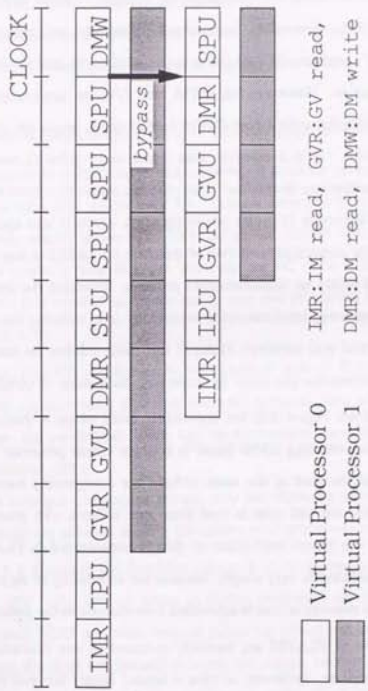


Fig. 5-2. Time Chart of FLATS2 Pipeline

FLATS2 accepts instructions every two cycles.

FLATS2 instruction set is targeted for memory oriented applications like Lisp program and array processing, and allows maximally two memory operands per instruction. GV registers (GVM) are also designed to be read once and written once in each instruction. However, IM, GVM, and DM are implemented in the current FLATS2 by using the same 1-port CMOS SRAM chips, which can either read or write at the same time. It thus takes at least 2 memory cycles (2 machine cycles) for FLATS2 to execute one instruction. The obtained instruction issue rate of FLATS2 is 0.5 instruction per cycle (2 cycles per instruction), which is also apparent from Figure 5-2. Though the instruction issue rate of the current FLATS2 is less than 1 instruction per cycle, it is purely an implementation problem. Doubling the throughput of GVM and DM by improving implementation technology (e.g. reducing the memory cycle by half, or using dual port memory), FLATS2 can easily achieve the instruction issue rate as much as 1 instruction per cycle, by increasing the number of virtual processors to 4.

As seen from Figure 5-2, the succeeding DMR phase is executed one machine cycle before the preceding DMW phase in a single virtual processor. This means that if the write and the read of the same address are contiguously executed in the same virtual processor, the old data is read from that address. To guarantee Read-after-Write on DM, the *bypass* mechanism of data is implemented in FLATS2. Such control can be implemented very simply, because the scheduling of each virtual processor is fixed and the memory access is scheduled deterministic in the instruction pipeline.

Instructions of FLATS2 are basically executed in one instruction cycle, tracing the pipeline template. However, in case a *hazard* occurs between instructions of the same virtual processor, the execution of the succeeding instruction is deferred for one instruction cycle. Such resource competitions occur, for example, in the following cases.

(1) Writing the result of SPU operation into GV register.

The result of SPU is available for store operation at L9. Therefore GVW phase must be deferred to the ninth stage, which collides to the GVW cycle of the next instruction. Consequently, the next instruction gets deferred for one instruction cycle.

(2) Branching on condition code, just after a floating-point multiplication.

A floating-point multiplication takes a whole 4 machine cycles, thus the resulted condition code does not fix until L9. This L9 corresponds to the L5 of the next conditional branch instruction. The conditional branch instruction requires the value of condition code in L3 to determine the PC value of the next instruction. Therefore, this conditional branch has to wait one instruction cycle for the condition code to fix.

However, FLATS2 can determine the condition code of P unit until L7 in most cases, by approximating the result from the exponent parts of operands [67]. In this case, the conditional branch can be successfully executed without incurring any additional delay.

Even if such resource competition occurs, only the following instructions of the same virtual processor are affected, leaving the other virtual processors unaffected.

Figure 5-3 illustrates the simplified data path of PC (program counter) in the first 4 stages of IPU. The internal status of virtual processors are connected to form a cyclic data path. CFP and other internal status has almost the same data path structure. Pipeline flip-flops are inserted in every two stages, because the instruction issue rate of the current FLATS2 is 0.5 instruction per cycle. Each pipeline flip-flop holds the PC value of currently executing instruction of each virtual processor. In the normal execution of instruction, *MUX1* selects the input *next* to increment PC. When a branch instruction is executed, *MUX1* select the input *jump*. In case the instruction execution is deferred because of resource competition, *MUX1* selects the input *retry* to

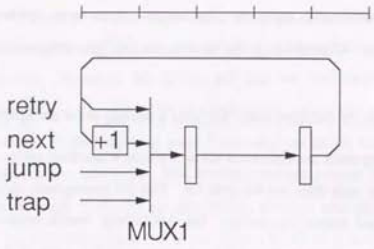


Fig. 5-3. Data Path of PC in IPU

hold the current PC and to retry this instruction in the next instruction cycle. If an exception occurs, *MUX1* selects the input *trap* to select the corresponding trap handler entry.

By making such loops for each internal status of virtual processor, even when some resource competition makes instruction execution deferred, the deferred virtual processor can evacuate the pipeline stage for other virtual processors ready for execution. The waiting virtual processor can retry the deferred instruction at the next instruction cycle. The resource competition is resolved during waiting in the loop because the preceding instruction which caused competition progresses in the pipeline and releases the resource.

5.2. Communication and Synchronization between Virtual Processors

In cyclic pipeline, the communication and synchronization mechanism can be easily implemented between virtual processors, because all virtual processors share the same hardware. The overhead of communication and synchronization is small by the same reason. This section briefly describes the mechanism of communication and synchronization between virtual processors in FLATS2. The case with more virtual processors is also discussed.

FLATS2 supports only the most primitive methods by hardware for inter-virtual-processor synchronization and communication. The actual primitives for communication and synchronization are supported by software. The current FLATS2 implementation includes only two virtual processors, thus there is little demand for specific fast methods supported by hardware. Nevertheless, the adopted implementation method is generally applicable regardless of the number of virtual processors. As a result, *inter-processor interrupt* is implemented as a method to make asynchronous communications between virtual processors. *LAS (Load And Store)* instruction is also implemented as a most primitive operation to implement arbitrary synchronization methods.

The inter-processor interrupt of FLATS2 is a kind of *broadcast*, which interrupts all virtual processors simultaneously, This specification can be implemented easily by preparing a hardware flag to request an inter-processor interrupt. Virtual processors circulate in the cyclic pipeline once every instruction cycle. Each virtual processor thereby certainly comes across the pipeline stage of interrupt request flag within one instruction cycle, consequently receiving the inter-processor interrupt request. The inter-processor interrupt request is thus notified to every virtual processors in one instruction cycle. Each virtual processor starts the interrupt handling as soon as interrupt is enabled. In cyclic pipeline, this method enables to send interrupt request to all virtual processors within one instruction cycle regardless of the number of virtual processors in cyclic pipeline. Though FLATS2 implements only *broadcast*, the point-to-point communication between two virtual processor is also possible in the same way. Let n be the number of virtual processors in the cyclic pipeline. Preparing n^2 bit request flags in hardware, interrupt requests can be sent within one instruction cycle

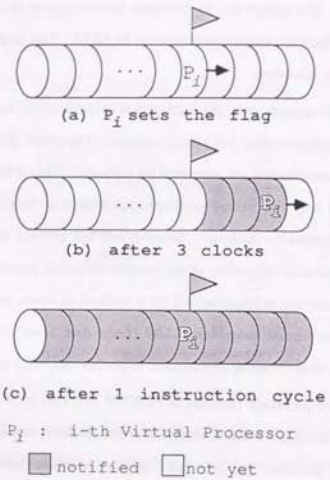


Fig. 5-4. Interrupt Request Flag

between any combination of virtual processors.

As a most primitive method to synchronize virtual processors, *las* instruction is supported in FLATS2. As described in chapter 3, *las* instruction loads a memory word to a destination register and stores the source register value to the same address as load operation in a mutual exclusive manner. In performing a mutual exclusive memory access on a memory shared MIMD computer, the other processors have to wait for the completion of the mutual exclusive access to guarantee the atomic nature of the access. To implement such mechanism, the *mutual exclusive flag* in a pipeline stage is enough

to force other virtual processors to wait for the completion of the mutual access. This mechanism is all the same as *interrupt request flag*. However, in this implementation, all other virtual processors are forced to waste instruction cycles during the *mutual exclusive flag* is set. In this case, the total number of wasted cycles increase in proportion to the number of virtual processors. This is a great deal in the cyclic pipeline with a large number of virtual processors. FLATS2 consequently adopted another method of implementation.

In FLATS2 architecture, the succeeding instruction cycles of the same virtual processor are to occupy the succeeding memory cycles. As known from the pipeline chart of FLATS (Figure 5-2), DMR of the succeeding instruction and DMW of the preceding instruction use the DM access contiguously. Exploiting this nature, the mutual exclusive access instruction can use two contiguous memory cycle indivisibly, by only guaranteeing for the mutual exclusive access instruction to use the contiguous two instruction cycles indivisibly. The most important merit of this technique is that the other virtual processors *do not* have to wait at all during this instruction is executed on a virtual processor.

Figure 5-5 illustrates the time chart of *las* instruction. *Las* instruction is executed by using two indivisible instruction cycles, as mentioned. In FLATS2, DMR of the succeeding instruction is executed one machine cycle earlier than DMW of the preceding instruction. Usual instructions therefore bypasses the write data of DMW to the read data of DMR, if the addresses are same. However, *las* instruction disables bypass mechanism, while using the same address for DMW and DMR. This control forces to read the old data in DMR and to write the new data in DMW, which is a desirable result for *las* operation. Consequently, *las* instruction stores the source register to memory operand in the first cycle, then loads the memory operand to the destination register in the second cycle, disabling bypass mechanism. *Las* instruction never affects the other virtual processor, because only the cycles of a single virtual processor are

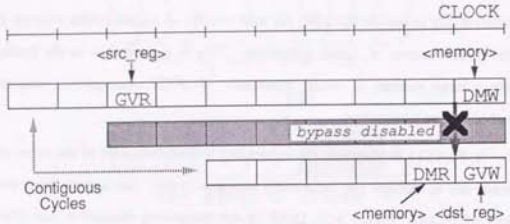


Fig. 5-5. LAS instruction Execution

concerned, though it takes two instruction cycles in the virtual processor that issued *las* instruction.

Regardless of the number of virtual processors in cyclic pipeline, the adequate pipeline design enables such characteristics that the mutual exclusive memory access does not interfere with other virtual processors. Under such a pipeline design, the mutual exclusive memory access instruction is always executed in a fixed time. The wasted cycle is equal to the execution cycles of mutual exclusive memory access instruction, because no other virtual processors lose their cycles. That is, the wasted cycle is constant regardless of the number of virtual processors. This means that such pipeline design is especially effective in the cyclic pipeline shared by many virtual processors.

5.3. Truncated Multiplier

In P unit, there is a multi-functional multiplier, which performs 32 × 32 bit full integer multiplication, two single precision floating-point multiplications (a pair of 32 × 32 bit multiplications), and one double precision floating-point multiplication (64 × 64 bit multiplication). Floating-point multiplier was one of the most difficult parts in FLATS2 to implement in the limited hardware resources. Here, we invented and adopted a new rounding method called "*Pseudo-random Rounding*"[68] in P unit to save circuitry by nearly half.

In order to build $N \times N$ bit multiplier, the circuits for N^2 bits and supplementary adders would be required. In floating-point calculations, however, the circuits for $\frac{1}{2}N(N-1)$ out of N^2 bits are used only to calculate lower N bits which are thrown away after rounding. The *pseudo-random rounding* economizes these multiplier circuits. By this rounding method, only $\frac{1}{2}N(N+1)$ bit circuits are used. Most of the lower N bits are not calculated: the average of that neglected part is compensated for by using the statistical properties of medium significant bits. Only problem is the error caused by truncation and compensation. The validity of error was thus verified by simulation. Also, the design of pseudo-random rounding of P unit was simulated and verified.

For more details on pseudo-random rounding, see the reference [68].

Chapter 6

The Evaluation of FLATS2

6. The Evaluation of FLATS2

Table 6-1 shows the results of some simple benchmark tests evaluated by a single virtual processor of FLATS2. The results of other processors are also presented for contrast. The source programs of these benchmarks are available from *netlib* service via electric mail (Dongarra [69]).

Benchmark Results					
Architecture CPU + FPU	Clock MHz	average MIPS	Dhrystone KD/s	Whetstone MW/s	Linpack MFLOPS
68020 + 68881	20	3	4.3	1.2	0.12
FLATS2 (1 proc.)	15	3.6	6.0	3.1	2.5
SPARC + Weitek	16.7	10	19.	3.9	1.1
R2000 + R2010	16.7	12	25.	9.0	1.8

Table 6-1. Some Benchmark Results

In the table, *MIPS* stands for "Mega Instruction Per Second," *KD/s* stands for "Kilo Dhrystones per Second," *MW/s* stands for "Mega Whetstones per Second," and *MFLOPS* stands for "Mega Floating-point Operations Per Second."

Though the column of *Clock* shows the primitive machine clock frequency of each processor in MHz, the clock frequencies is not directly comparable, because the definition of "clock" is not common among the processors. For example, MC68020 and FLATS2 have only one system clock signal, while MIPS R2000 requires two kinds of clock inputs, phases of which are shifted 90 degrees each other, and the frequency of which is twice as high as the frequency shown in the table. Also FLATS2 has a peculiarity in hardware, which is shared by two virtual processors in time sharing manner. Thus the clock of FLATS2 does not directly corresponds to that of other processors. Even if the primitive clock of FLATS2 is 15 MHz as shown, the share of each virtual processor is only a half.

Average MIPS in the figure is similar. The complexity of instructions of each processor is different, and the result is affected by instruction mixes, therefore *MIPS* rating can be only a rough measure. Instead, some benchmark tests are used in the following sections to evaluate various aspects of FLATS2.

6.1. Dhrystone

Dhrystone benchmark [70] is a benchmark program to measure integer operation performance. There are many versions of *Dhrystone* benchmark program according to the implementation language, though the original was written in Ada. Here, *Dhrystone* benchmark version 2.1 of C language was used for evaluation.† The measured Dhrystone performance of each processor is almost proportional to its average *MIPS* rate. Normalizing the measured Dhrystone performances by *average MIPS*, the normalized Dhrystone values distribute between 1.4 and 2.1. Considering the "*average MIPS*" is a very rough measure, the dispersion as much as 50% in the normalized Dhrystone values seems to be rational.

Needless to say, the evaluation result also reflects the quality of C compiler and related library functions, together with the bare hardware performance. For FLATS2, GNU C compiler (*gcc*, version 1.37)[71] was retargeted with some additional features to be adapted for FLATS2 architecture. This modified compiler, which is called FLATS2 C compiler (*fcc* [59]), was used for the evaluation. The library functions of FLATS2 are also written in C languages and compiled by *fcc*. For other processors, vendor's standard C compilers and libraries were used for the evaluation.

Dhrystone benchmark often uses character string handling routines such as *string copy (strcpy)* and *string compare (strcmp)*. The execution profile on Sun-3 (MC68020) and DEC microVAX-3500 shows that around 25 percent of execution time is consumed in these two functions (*strcpy* and *strcmp*). This means that byte handling performance has much effect on dhrystone performance. However, FLATS2 can

† On Dhrystone version 1.1, the result of FLATS2 is 6.5 KD/s.

handle bytes only with GVU, that is, with load/store architecture. Moreover, the byte load instruction of FLATS2 takes 2 instruction cycles for execution by implementation reasons. These characteristics of FLATS2 are rather disadvantageous for dhrystone measurement. Consequently the execution time of Dhrystone benchmark is extended, which worsens measured dhrystone performance, while increasing the ratio of string routines in the total execution time. In fact, it is observed in FLATS2 that about 37 percent of total execution time of Dhrystone elapses in *strcmp* and *strcpy*. This is more than other processors shown in Table 6-2. By implementing byte instructions more efficiently, Dhrystone performance is to be improved.

Percentage of *strcpy* and *strcmp*					
CPU/compiler	Sun/cc	Sun/gcc	VAX/cc	VAX/vcc	FLATS2/fcc
strcmp	16.0	16.9	12.3	13.6	18.0
strcpy	9.9	9.7	11.1	11.7	18.7

Table 6-2. Share of Strcpy and Strcmp in Dhrystone Execution Time

6.2. Whetstone

Whetstone is a benchmark program to evaluate the floating-point arithmetic performance, in particular the performance of some *elementary functions* such as logarithm function, exponential function, and trigonometrical functions. Whetstone benchmark program was originally written in Algol 60, but here we use a C language version, which can be found in *netlib* benchmark collection. In measuring Whetstone performance on FLATS2, elementary functions were hand-coded and optimized by using BL addressing modes and combined arithmetic instructions (especially with *poly* instruction). As shown in Table 6-1, the Whetstone performance of FLATS2 is rather good compared to its *MIPS* rate, and considering this performance is achieved by only a half of its hardware potential.

In case the elementary functions are compiled by C compiler with neither loop optimization by BL addressing nor combined arithmetic instructions, the measured Whetstone performance gets worse about 12%. This portion of performance (lost 12%) is regarded to be the performance improvement obtained by BL addressing and combined arithmetic. However, even if this portion is lost, the Whetstone performance of FLATS2 is still good compared to other processors, considering its *MIPS* rate. This fact suggests that there are still some other reasons than BL addressing and combined arithmetic. One of the reasons is guessed to be that FLATS2 instructions can naturally overlap arithmetic operations and memory accesses by using two memory operands in each instruction. This architecture can potentially reduce the load/store overhead of memory variables, thus improving the performance. Another more likely reason is that FLATS2 can finish a floating-point operation in one instruction cycle. By overlapping the long latency time of a floating-point operation to the execution time of other virtual processors, each virtual processor can use the result of the previous floating-point arithmetic instructions just after one instruction cycle. This characteristic will improve floating-point arithmetic performance more in comparison with its *MIPS* rate.

6.3. Linpack

Though the name of *Linpack* originally stands for "LINear equation systems PACKage," recently it is very popular to use Linpack as a benchmark test. The benchmark program collection in *netlib* thus includes the Linpack benchmark of various kinds, in which the C version and FORTRAN version are used to evaluate FLATS2 performance.

The most primitive routines in Linpack are called *BLAS (Basic Linear Algebra Subprograms)*, each of which performs a simple array processing procedure with a small loop structure. As shown in chapter 3 and 4, such small loops can be implemented effectively by using FLATS2 instruction set. Table 6-3 lists the number of instruction cycles which is necessary to perform one iteration of the innermost loop in

each BLAS routine optimized by hand-coding, together with the arithmetic density in the innermost loop of each BLAS routine.

Linpack routines (BLAS)			
name	note	cycle/loop	FLOP/cycle
dasum	$\sum \lvert x_i \rvert$	2	1
daxpy	$y^v = y^v + \alpha x^v$	2	1
dcopy	$y^v = x^v$	1	1
ddot	$\sum x_i y_i$	1	2
drot	Rotation	6	1
dscal	$x^v = \alpha x^v$	1	1
dswap	Swap x^v and y^v	3	2/3
idamax	Return i of max $\lvert x_i \rvert$	3 to 5	2/3 to 2/5

Table 6-3. Double Precision BLAS Routines

As shown in Table 6-3, most of BLAS routines achieve the arithmetic density as high as equal or more than 1 FLOP/cycle. Only *ddot* has its arithmetic density as high as 2, because a combined instruction (*rip.d*) is adopted in the innermost loop of *ddot* to utilize ALU and multiplier in parallel. The other routines only adopt usual arithmetic instructions, each of which executes a single arithmetic operation. The result shown in Table 6-3 represents that as many operations as loop termination test, conditional branch, and operand fetches are perfectly overlapped to the arithmetic operations, because the arithmetic density of each routine is exactly 1 (or 2 in *ddot*) without incurring any additional overhead.

The arithmetic density of *idamax* has a variation of 2/3 to 2/5, because *idamax* includes a conditional branch in its innermost loop. It is one of the merits of BL addressing which is a scalar architecture and applicable to the loop including conditional branches. Another merit of BL addressing is that it can handle both loops with constant stride and variable stride. Though each BLAS routine is coded to have two innermost loops for constant stride and variable stride, FLATS2 can implement both

loops at the same arithmetic density.

Linpack benchmark solves linear simultaneous equations of dimension 100 (or 200, etc.), mainly using a BLAS routine daxpy. The benchmark result of FLATS2 is 2.5 MFLOPS on Linpack by a single virtual processor, as shown in Table 6-1. This is rather fast, comparing the *MIPS* rate of FLATS2. In fact, the run-time profile on FLATS2 shows that the 69 percent of execution time elapses in the innermost loop of *daxpy*, where the arithmetic density as high as 1 FLOP/cycle is achieved.

Linpack performs matrix computations, in which much parallelism can be found, because the computation of each element can be done in parallel. Therefore, it is very easy to parallelize Linpack by using plural virtual processors of a cyclic pipeline computer, because virtual processors share the same main memory. FLATS2 FORTRAN offers some features to utilize the parallelism of virtual processors. The measured performance of parallel version of Linpack is 5.0 MFLOPS by using two virtual processors with FLATS2 FORTRAN. The overhead of parallelization is little, thus almost twice performance is achieved by two virtual processors.

6.4. Livermore Kernels

Livermore Loop is a benchmark test developed in Lawrence Livermore National Laboratory to evaluate the performance of supercomputers. This benchmark consist of 24 small program pieces (*Livermore kernels*) taken from scientific calculation programs frequently used in physics. Here, the first 14 loops (*original Livermore kernels*) are examined and optimized by hand-coding for FLATS2 architecture to evaluate the numerical performance of FLATS2. Then, the measured performance of full 24 kernels are reported by using FLATS2 FORTRAN compiler and libraries.

Table 6-4 shows the specifications of the innermost loops of 14 kernels. Figure 6-1 illustrates this fourth column as a bar graph. In the table, *FLOP* stands for "Floating-point Operations," and *H.M.* stands for "Harmonic Mean."‡ The first

‡ Harmonic mean is more rational than arithmetic mean in this case [72], because arithmetic

Livermore Kernels							
kernel	FLOP/loop	cycle/loop			FLOP/cycle		
		basic	+rip	+all	basic	+rip	+all
1	5	5	5	5	1.00	1.00	1.00
2	4	6	6	6	0.67	0.67	0.67
3	2	3	1	1	0.67	2.00	2.00
4	2	3	1	1	0.67	2.00	2.00
5	2	3	3	2	0.67	0.67	1.00
6	2	3	1	1	0.67	2.00	2.00
7	16	17	17	17	0.94	0.94	0.94
8	36	54	54	54	0.67	0.67	0.67
9	17	31	15	15	0.55	1.13	1.13
10	9	21	21	14	0.43	0.43	0.64
11	1	2	2	1	0.50	0.50	1.00
12	1	2	2	1	0.50	0.50	1.00
13	7	40	40	40	0.18	0.18	0.18
14	11	33	33	33	0.33	0.33	0.33
H.M.					0.50	0.58	0.68

Table 6-4. Innermost Loops of 14 Kernels

column shows the kernel number, and the second column shows the number of floating-point arithmetic operations required in the innermost loop of each kernel. The third column shows the instruction cycles necessary for FLATS2 to execute each loop body once. The fourth column shows the arithmetic density in the innermost loop of each kernel, that is, it represents the *peak arithmetic density* expected within the loop. Note that arithmetic density (FLOP/cycle) is directly convertible to arithmetic performance (FLOPS) by multiplied by instruction cycle time. The arithmetic density 1.0 FLOP/cycle corresponds to 3.8 MFLOPS in the current FLATS2 implementation. If highly integrated circuits are used for the future implementation, the corresponding performance will get higher.

In the Table 6-4, FLATS2 architecture is evaluated at the following three stages.

(1) **basic**

The case with basic instructions of FLATS2. That is, each instruction uses

density is the measure which has the dimension of $[T^{-1}]$ as well as arithmetic ratio (FLOPS).

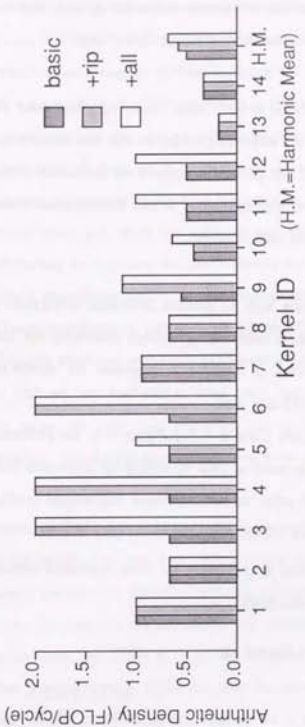


Fig. 6-1. Peak Arithmetic Density

maximally two memory operands, one for read and the other for write operand. No combined arithmetic instruction is used, thus only one floating-point operation is performed at most in one instruction.

(2) **+rip**

In addition to the "*basic*" case, *rip (Real Inner Product)* instruction is adopted. Also, the addressing modes to read two memory operands (*Read-Read addressing modes*) are allowed to support *rip* instruction. This combination enables to overlap two load operations to two floating-point arithmetic operations (one addition and one multiplication).

(3) **+all**

The case with all possible combined instructions which can be implemented by additional microcoding without modifying the current FLATS2 hardware. The *+all* case is regarded to represent the maximum performance of the current FLATS2 data path.

As seen from Table 6-4 and Figure 6-1, the performance of Livermore kernels are improved as much as 16% by adding *rip* instruction and *Read-Read addressing modes*. Adding all other ad hoc combined instructions (*+all*), the performance is improved further more (17%), compared to the case of *+rip*.

To show improvement by other combined instruction than *rip*, let's examine a loop with recurrence.

Livermore Kernel 11:

```
      do  11  k = 2, n
11    x(k)  =  x(k-1) + y(k)
```

In this loop, the element of array *x* is read from the memory just after being written in the previous instruction. Typical vector computers can not vectorize this loop and

shows very poor performance. *Recurrence* is not a problem for the FLATS2, because of the scalar nature of BL addressing, but the maximal memory transfer rate limits the peak performance in this case. Without using combined arithmetic instructions, this loop is coded by two instructions, achieving the arithmetic density 0.5. However, this loop can be improved with a new combined instruction which omits one memory read by writing the arithmetic result into both the memory and the register. With this new combined instruction, the load of *x(k–1)* from the memory can be omitted, because it is already available on the register. Consequently, the loop body can be implemented by this single instruction, which loads *y(k)* from the memory and stores *x(k)* to the memory and the register, achieving the improved arithmetic density as high as 1. With some new types of combined instructions which can be implemented by additional microcoding (without hardware modification), kernel 5, 10, 11, and 12 are improved. However, no generally applicable combined arithmetic was found in Livermore kernels besides *rip*, therefore no such ad hoc instructions have been implemented in the current FLATS2. In that sense, the *+rip* case represents the performance of the current FLATS2 implementation. The following evaluation was done under this condition (*+rip*).

The previous evaluation of Table 6-4 represents the peak performance of the innermost loop. Actually the overhead of initial and terminal procedure burdens the loop to degrade the measured performance from the expected peak performance. To evaluate this overhead of loop initialization and termination, the arithmetic density was measured on various iteration count (n). Table 6-5 lists the measured arithmetic density of each kernel on three iteration counts. Figure 6-2 also illustrates the variations of the measured arithmetic density of 14 Livermore kernels as a bar graph. The column "$n = \infty$" stands for the peak arithmetic density shown in Table 6-5. In case n is big enough, the arithmetic density within the innermost loop gets dominant, therefore the case $n = \infty$ is regarded to represent the upper limit of expected performance to

Livermore Kernels				
kernel	FLOP/cycle			MFLOPS
	n = 16	n = 64	n = ∞	n = 64
1	0.85	0.96	1.00	3.7
2	0.35	0.51	0.67	2.0
3	1.28	1.75	2.00	6.7
4	0.31	0.89	2.00	3.4
5	0.59	0.65	0.67	2.5
6	0.88	1.52	2.00	5.8
7	0.91	0.93	0.94	3.6
8	0.66	0.67	0.67	2.6
9	1.11	1.13	1.13	4.3
10	0.42	0.43	0.43	1.7
11	0.42	0.48	0.50	1.8
12	0.46	0.49	0.50	1.9
13	0.17	0.17	0.18	0.7
14	0.32	0.33	0.33	1.3
H.M.	0.46	0.54	0.58	2.1

Table 6-5. Measured Arithmetic Density

be achieved. In Table 6-5, the arithmetic performance (ratio) of the case $n = 64$ is also presented for the reference.

As seen in Figure 6-2, the performance decreases gradually in proportion as the iteration count decreases. However, the degree of performance degradation is small, which means that the overhead of loop preparation is small. One of the reason is that BL addressing is a scalar architecture, which has little internal status dedicated for BL addressing. BL addressing mode uses general registers in operation, and the initialization consequently gets easy and fast. Another reason is possibly cyclic pipeline architecture, which alleviates the latency of instruction execution by overlapping it to the other virtual processors.

Consequently, the harmonic mean of the arithmetic density of 14 original Livermore kernels is 0.58 (in case $n = \infty$), 0.54 ($n = 64$), and ~~0.50~~ 0.46 ($n = 16$), respectively. Converting this arithmetic density to arithmetic performance, the performance of a single virtual processor of FLATS2 is measured to be 2.1 MFLOPS (in case of n = 64,

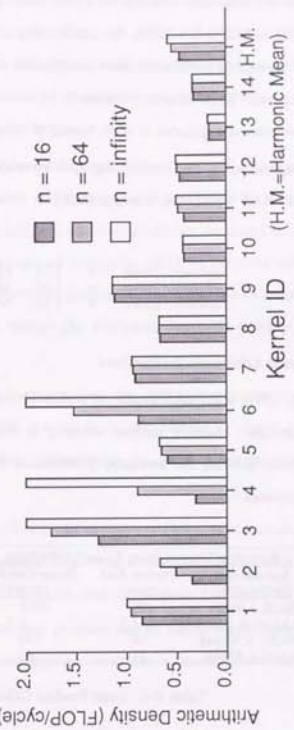


Fig. 6-2. Measured Arithmetic Density

harmonic mean). Using arithmetic mean and geometric mean, the performance is represented as 3.0 MFLOPS and 2.6 MFLOPS, respectively. Considering the *MIPS* rate of FLATS2 is around 3.5 MIPS, this performance of original Livermore kernels is rather good. Livermore kernels are more complicated than Linpack and more similar to real applications. High efficiency shown in 14 Livermore kernels therefore proves that this architecture is applicable to wide variety of computations.

The complete set of Livermore Loops (24 kernels) were also evaluated by using FLATS2 FORTRAN (*ff77*) with a single FLATS2 virtual processor. The result is as follows:

arithmetic mean	=	1.74	MFLOPS,
geometric mean	=	1.00	MFLOPS,
harmonic mean	=	1.13	MFLOPS.

6.5. Combined Arithmetic Instructions

FLATS2 offers some other combined arithmetic instructions in addition to *rip* and *poly*. In particular, complex number handling is one of unique characteristics of FLATS2. Table 6-6 lists the measured performance of the inner product calculation on some processors.

Execution Time of Inner Product Calculation (dimension = 10000)			
Architecture CPU + FPU	Double Real (msec)	Single Complex (msec)	Double Complex (msec)
68020 + 68881	106	1063	1503
FLATS2 (1 proc.)	2.7	2.7	5.3
SPARC + Weitek	16	133	111
R2000 + R2010	21	40	72

Table 6-6. Inner Product Calculation

In double precision real inner product calculation, *rip* instruction performs one multiplication and one addition in each cycle. Therefore, the inner product of dimension 10000 is calculated in 10000 instruction cycles (2.7 ms).

In single complex inner product calculation, *cip.f* instruction is applied to perform 4 multiplications and 2 additions for a complex multiplication and 2 additions for a complex addition in each cycle. The inner product of dimension 10000 is therefore calculated in 10000 cycles (2.7 ms). Consequently the execution time is the same as the real inner product calculation, though the arithmetic density is 4 times higher in single complex than in double real.

In double complex inner product calculation, a pair of *cip1.d* and *cip2.d* instructions are applied to perform 4 multiplications and 2 additions for a complex multiplication and 2 additions for a complex addition (all double precision) in every two instruction cycles. The inner product of dimension 10000 is therefore calculated in 20000 cycles (5.3 ms). Consequently the execution time is exactly twice as long as the real inner product calculation, though the arithmetic density is twice in double complex compared to double real.

Converting from the execution time, the arithmetic performance in inner product calculation is 7.4 MFLOPS (double real), 30. MFLOPS (single complex), and 15. MFLOPS (double complex), respectively, by using a single virtual processor of FLATS2.

Similarly, FLATS2 offers a pair of combined arithmetic instructions dedicated for single complex FFT calculation. By using these *cfft1* and *cfft2* instructions, FLATS2 performs a 1024-point FFT in 11 ms, and a 8192-point FFT in 95 ms.

All these array processing program can be easily parallelized for plural virtual processors, because a cyclic pipeline computer is naturally a memory shared MIMD computer.

Chapter 7

Conclusion

7. Conclusion

Using the logic devices with latch function (e.g. Josephson devices), a computer is naturally pipelined to a high-pitch and shallow-logic pipeline. To utilize such pipeline fully, Cyclic Pipeline Architecture (CPA) has been proposed and researched. CPA is a kind of MIMD computer called resource shared MIMD, in which the pipeline is shared among plural virtual processors in a time-sharing manner. Though CPA was originally introduced for the Josephson computers, CPA is also effective in existing silicon technology to get the higher performance per cost.

In this dissertation, the architecture of a Cyclic Pipeline Computer FLATS2 is described and evaluated. FLATS2 is implemented with conventional Silicon technology. However, its architecture is not specific to the current implementation technology but applicable to the cyclic pipeline with more virtual processors, including future Josephson computers. In FLATS2, the internal status of each virtual processor is simplified and hardware resources are shared as much as possible to ease the migration of processes between virtual processors. This dissertation described the current FLATS2 implementation, as well as the design consideration of the cyclic pipeline with much more number of virtual processors. The communication and synchronization between virtual processors can be implemented effectively and easily, because of the shared hardware. The design and implementation of such communication methods are described.

The instruction set architecture of FLATS2 was also presented. FLATS2 provides addressing modes with range check facility (BL addressing modes) and combined arithmetic instructions, which are effective in a wide variety of array processing. In BL addressing modes, any effective address is compared against the base and limit addresses specified in the instruction, then a conditional branch is executed according to the result of range check. This dissertation presented the way to apply this scheme to loop optimization and other purposes. Combined arithmetic instructions are also

effective to achieve high arithmetic performance, cooperating with BL addressing modes of FLATS2. Some examples of programming on FLATS2 were given, and some benchmark programs were investigated and evaluated on FLATS2 architecture. Obtained evaluation results of FLATS2 architecture were also presented in this dissertation.

FLATS2 is an MIMD computer of 2 instruction streams. Though the evaluation of a single virtual processor is now in progress from various aspects, the utilization of plural virtual processors is still not satisfactory. It is strongly desired that the software system is prepared to exploit the parallelism by virtual processors. Especially in the cyclic pipeline with further more number of virtual processors, it is regarded to be impossible to utilize virtual processor without elaborated software supports. In developing software system for cyclic pipeline computer, it is expected that more suitable methods could be found for cyclic pipeline to exploit more parallelism.

References

1. A. Barone and G. Paterno, *Physics and Applications of the Josephson Effect,* John Wiley & Sons, New York, 1982.

2. K. F. Loe and E. Goto, "Analysis of Flux Input and Output Josephson Pair Device," *IEEE Trans. Magn.*, vol. MAG-21, no. 2, pp. 884-887, Mar. 1985.

3. E. Goto and K. F. Loe, *DC Flux Parametron,* World Scientific, Singapore, 1986.

4. Y. Harada, H. Nakane, N. Miyamoto, U. Kawabe, E. Goto, and T. Soma, "Basic Operations of the Quantum Flux Parametron," *IEEE Trans. Magn.*, vol. MAG-23, no. 5, pp. 3801-3807, Sept. 1987.

5. J. G. Earle, "Latched Carry-Save Adder," *IBM Technical Disclosure Bulletin,* vol. 7, pp. 909-910, March 1965.

6. D. W. Anderson, F. J. Sparacio, and F. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM J. Res. and Dev.*, vol. 11, no. 1, pp. 8-24, 1967.

7. S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Res. and Dev.*, vol. 11, pp. 34-53, 1967.

8. S. R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers," *Proc. 13th Annual Sym. on Comp. Arch.*, pp. 404-411, ACM, 1986.

9. C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *Computing Survey,* vol. 9, no. 1, pp. 61-102, Mar. 1977.

10. H. S. Stone, *High-Performance Computer Architecture,* Addison-Wesley, Reading, Mass., 1987.

11. K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill, 1984.

12. D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982.

13. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. and Dev.*, vol. 11, no. 1, pp. 25-33, 1967.

14. D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," *Computer*, vol. 21, no. 7, pp. 47-55, July 1988.

15. R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Comput.*, vol. C-35, no. 9, pp. 815-828, 1986.

16. A. R. Pleszkun and G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors," *Proc. 15th Annual Sym. on Comp. Arch.*, pp. 37-44, 1988.

17. N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 272-282, 1989.

18. M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on Multiple Instruction Issue," in *Proc. Third International Conference on Architectural Support for Programming and Operating Systems (ASPLOS III)*, pp. 290-302, 1989.

19. S. Weiss and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," *Proc. Second International Conference on Architectural Support for Programming and Operating Systems (ASPLOS II)*, pp. 105-109, 1987.

20. K. Shimizu, E. Goto, and S. Ichikawa, "CPC(Cyclic Pipeline Computer)–An Architecture Suited for Josephson and Pipelined Machines," *IEEE Trans. Comput.*, vol. C-38, no. 6, pp. 825-832, June 1989.

21. Motorola Inc., *MECL Device Data*, Motorola Semiconductor Products Inc., 1983.

22. W. R. Blood, Jr., *MECL System Design Handbook*, Motorola Semiconductor Products Inc., 1983.

23. Fairchild Corp., *F100K ECL Data Book*, Fairchild Camera and Instrument Corporation, 1982.

24. Fairchild Corp., *F100K ECL User's Handbook*, Fairchild Camera and Instrument Corporation, 1982.

25. Fairchild Corp., *FAST Data Book*, Fairchild Camera and Instrument Corporation, Portland, Maine, 1985.

26. Fairchild Japan Corp., *Fairchild Advanced Schottky TTL Application Manual*, CQ publishing, Tokyo, 1982. (in Japanese)

27. M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948-960, Sept. 1977.

28. H. F. Jordan, "Performance Measurements on HEP – A Pipelined MIMD Computer," *Proc. 10th Annual Sym. on Comp. Arch.*, pp. 207-212, Stockholm, Sweden, June 1983.

29. H. F. Jordan, "HEP Architecture, Programming and Performance," in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, ed. J. S. Kowalik, MIT Press, 1985.

30. Motorola Inc., "16K × 4 bit Synchronous Static RAM with Output Registers : MCM6293," Motorola Semiconductor Technical Data, 1988.

31. D. Bursky, "Advanced Self-Timed SRAM Pares Access Time To 5 ns," *Electronic Design*, pp. 145-147, Feb. 22, 1990.

32. B. A. Chappel and T. I. Chappel, "Pipelined Memory Chip Structure Having Improved Cycle Time," U. S. Patent 4845677, July 1989.

33. S. Ichikawa, "FLATS2 Architecture Handbook," Reseach Development Corporation of Japan, 1990. (Appendix A of this dissertation, in Japanese)

34. Computer Architecture Group of GOTO Project, "FLATS2 Instruction Set Manual," Reseach Development Corporation of Japan, 1990. (Appendix B of this dissertation)

35. A. Tevanian and R. F. Rashid, "MACH: A Basis for Future UNIX Development," Technical Report CMU-CS-87-139, Carnegie Mellon University, June 1987.

36. A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young, "Mach threads and the UNIX kernel: the battle for control," in *Proceedings of Summer Usenix*, June 1987.

37. E. H. Frank and R. F. Sproull, "Testing and Debugging Custom Integrated Circuits," *Computing Survey*, vol. 13, no. 4, pp. 425-451, 1981.

38. Digital Equipment Corp., *VAX Architecture Handbook*, 1981.

39. D. A. Moon, "Architecture of the Symbolics 3600," *Proc. of 12th Int'l Symp. on Computer Architecture*, pp. 76-83, 1985.

40. A. D. Kenneth, "A Retrospective on the Dorado: A High-Performance Personal Computer," in *Proc. of 10th Int'l Symp. on Computer Architecture*, 1983.

41. H. Hayashi, et al., "ALPHA: A High-Performance LISP Machine Equipped with a New Stack Structure and Garbage Collection System," in *Proc. of 10th Int'l Symp. on Computer Architecture*, pp. 342-348, 1983.

42. G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn, "Evaluation of the SPUR Lisp Architecture," *Proc. of 13th Int'l Symp. on Computer Architecture*, pp. 444-452, 1986.

43. E. Goto, T. Soma, N. Inada, T. Ida, M. Idesawa, K. Hiraki, M. Suzuki, K. Shimizu, and B. Philipov, "Design of a Lisp Machine – FLATS," in *Conference*

Record of the 1982 ACM Symp. on Lisp and Functional Programming, pp. 208-215, Pittsburgh, Aug. 1982.

44. N. Inada, M. Suzuki, K. Shimizu, and M. Sato, "Overview and Current Status of the FLATS," in *The Second RIKEN Int'l Symp. on Symbolic and Algebraic Computation by Computers*, ed. N. Inada and T. Soma, pp. 35-40, World Scientific, Singapore, 1985.

45. K. Hiraki and E. Goto, "An Architecture of FLATS - A Computer for Symbolic and Algebraic Computations," *Trans. of Information Processing Sciety of Japan*, vol. 27, no. 1, Jan. 1986. (in Japanese)

46. K. Hiraki, "A Study on Formula Manipulation Computer," Doctoral Dissertation, Univ. of Tokyo, 1986. (partially in Japanese)

47. G. J. Myers, *Advances in Computer Architecture*, John Wiley & Sons, New York, 1978.

48. Motorola Inc., *MC68020 32-bit Microprocessor User's Manual (2nd Edition)*, Prentice-Hall Inc., Eaglewood Cliffs, N.J., 1985.

49. Kai Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York, 1978.

50. International Business Machines Corp., "IBM System/370 Principles of Operation," Form GA22-7000.

51. E. W. Dijkstra, "Co-operating Sequential Processes," in *Programming Language*, ed. F. Gennys, pp. 43-112, Academic Press, 1968.

52. N. P. Jouppi, J. Bertoni, and D. W. Wall, "A Unified Vector/Scalar Floating-Point Architecture," in *Proc. Third International Conference on Architectural Support for Programming and Operating Systems (ASPLOS III)*, pp. 134-143, 1989.

53. M. Sato, S. Ichikawa, and E. Goto, "Run-Time Checking in LISP by Integrating

Memory Addressing and Range Checking," *Proc. 16th Annual Sym. on Comp. Arch.*, pp. 290-297, Jerusalem, Isreal, May 1989.

54. J. A. Fisher, "Very Long Instruction Word Architectures and The ELI-512," *Proc. 10th Annual Sym. on Comp. Arch.*, pp. 140-150, ACM, 1983.

55. J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478-490, July 1981.

56. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass., 1986.

57. G. S. Sohi and S. Vajapeyam, "Tradeoffs in Instruction Format Design for Horizontal Architectures," in *Proc. Third International Conference on Architectural Support for Programming and Operating Systems (ASPLOS III)*, pp. 15-25, 1989.

58. R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. Comput.*, vol. C-37, no. 8, pp. 967-979, August 1988.

59. P. Spee, "Dynamic Type and Range Checking in C using a Tagged Architecture," Reseach Development Corporation of Japan, 1988.

60. M. Sato, "Evaluation of FLATS2 FORTRAN Compiler," *The seventh RIKEN symposium on Josephson Electronics*, pp. 1-9, Wako-shi, March 1990. (in Japanese)

61. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

62. R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithm*, Adam Hilger, Bristol, 1981.

63. Guy Steele, Jr., *Common LISP: The Language*, Digital Press, 1984.

64. E. Goto and T. Soma, et al., "FLATS: A Machine for Symbolic and Algebraic Manipulation," in *The Second RIKEN Int'l Symp. on Symbolic and Algebraic*

Computation by Computers, ed. N. Inada and T. Soma, pp. 231-246, World Scientific, Singapore, 1985.

65. T. Chikayama, "Implementation of the Utilisp System," *Transactions of Information Processing Society of Japan*, vol. 24, no. 5, pp. 599-604, 1983.

66. D. E. Knuth, *The Art of Computer Programming (second edition)*, Volume 2/ Seminumerical Algorithms, Addison-Wesley, Massachusetts, 1981.

67. S. Ichikawa, "Fast Branching with Exceptional Delay," in *A Study on a Cyclic Pipeline Computer: FLATS2*, Dept. of Information Science, University of Tokyo, February 1987.

68. N. Yoshida, E. Goto, and S. Ichikawa, "Pseudo-random Rounding for Truncated Multipliers," *IEEE Trans. Comput.*, (to be published).

69. J. Dongarra and E. Grosse, "Distribution of Mathematical Software via Electronic Mail," *CACM*, vol. 30, no. 5, pp. 403-407, May 1989.

70. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *CACM*, vol. 27, no. 10, pp. 1013-1030, October 1984.

71. R. M. Stallman, "Internals of GNU CC," Free Software Foundation, 1987.

72. J. E. Smith, "Characterizing Computer Performance With A Single Number," *CACM*, vol. 31, no. 10, pp. 1202-1206, October 1988.

INDEX

Appendix A

FLATS2

Architecture Handbook

(*in Japanese*)

Table of Contents

データ型とデータ表現

本章では、FLATS2マシンの扱うデータの型(タイプ)と、その表現形式について述べる。

「データ型」とは、命令がオペランドを処理する際に指定する、オペランドの型をいう。一方「データ表現」は、データ型が実際のハードウェア上で表現される方法を指す。例えば浮動小数点演算を行なう際には、命令はオペランドの型として単精度浮動小数点を指定するが、実際のオペランドはメモリ上にあってもレジスタ上にあってもよい。このとき、同じデータでもメモリ上にあるかレジスタ上にあるかで、その存在形態が異なっていることがある。この例では、単精度浮動小数点が「データ型」、データの存在形態が「データ表現」である。

1. データ型

FLATS2が命令で扱うことのできるデータ型は、基本的に以下の8種である。

表：FLATS2のデータ型

型名	幅/bit	説明
Bit	1	操作は4種(set/clr/tst/chg)
Byte	8	操作は3種(ld/ext/st)
Half word	16	操作は3種(ld/ext/st)
Word	32 + 1	データ部 + アドレス・タグ
Quad	64 + 2	ワードが2つ
Float	32 + 1	アドレス・タグはつねに0
Double float	64 + 2	アドレス・タグはつねに0
Complex	32 + 1	Float × 2

これに加えて、Double complex と Multi-word integer について、演算を支援するためのプリミティブ命令を実装している。以下の節で、これらのデータ型に対するFLATS2のアーキテクチャ上のサポートについて、説明してゆく。

アーキテクチャ上の大きな特色として、FLATS2においては、各ワード(データ部32ビット)に1ビットのアドレス・タグがついている。アドレス・タグが1である場合、データ部の32ビットはアドレスであると解釈される。アドレス・タグが0である場合は、データ部は数値であると解釈される。以下では、まずアドレス型について概説し、続いて各数値型について簡単に解説する。

1.1. アドレス

FLATS2のアドレス空間は、用途に従って3つに分かれている。それぞれ、I空間(命令用)/V空間(レジスタ用)/D空間(データ用) とよばれ、アドレスの上位2ビットで区別される。残りの(下位)30ビットの解釈も、空間によって異なっている。

1.1.1. D空間のアドレス

D空間のアドレスは、上位1ビットが0になっている。下位31ビットは、D空間内でのバイト単位のアドレスを表す。

1.1.2. I空間のアドレス

I空間のアドレスは、上位2ビットが2(2進の10)になっている。下位30ビットは、I空間内での倍語単位のアドレスを表す。

1.1.3. V空間のアドレス

V空間のアドレスは、上位2ビットが3(2進の11)になっている。続く24ビット(bit29-bit6)は、レジスタ・フレーム番号を表す。下位6ビットには、アドレスとしての意味はない。(この部分は、CFPにPSWの一部を合成するとき使用される。) 一つのレジスタ・フレームには32の全語が含まれるので、結局下位の30ビット(の下位6ビットを0にマスクしたもの)は、V空間内の半語単位のアドレスであると見ることができる。

1.2. 数値

アドレス・タグの値が0であるデータは、数値である。アドレス・タグは各ワードに1ビットしかないし、そもそもデータ部が32ビットに満たない型はアドレスではあり得ないから、ビット/バイト/半語は全て数値として扱われる。

数値型は、整数と浮動小数点数に大別できる。整数型には、ビット/バイト/半語/全語/倍語/多倍長語が含まれる。浮動小数点型には、単精度/倍精度/単精度複素数が含まれる。以下、これらの数値型について概観する。

1.2.1. ビット

ビット演算は、4種類提供されている（ビットのテスト、セット、クリア、反転)。これらの演算のオペランドは3つで、ソース・オペランド(ワード)とビット位置(ワード内オフセット=5ビット)、およびデスティネーション(ワード)である。ソース及びデスティネーションには、メモリまたはGVレジスタが使用できる（これが、オペランドがバイト長でなくワード長である理由)。ビット位置の指定は、GVレジスタの下位5ビットを使用して行なう。ビット番号は、下位ビットから順に(0から31)番号付けられている。

1.2.2. バイト

バイト処理の方法は、3種類提供されている(ldb/extb/stb)。 ldbは、ソース・オペランドの1バイトを、符号なし数として零詰めでワードに拡張し、GVレジスタに格納する。 extbは、ソースの1バイトを、符号つき数としてワードに符号拡張し、 GVレジスタに格納する。 stbは、ソースの下位1バイトを、デスティネーションの1バイトに書き込む。バイトについては、FLATS2はロード・ストア・アーキテクチャであり、バイト演算やメモリ間バイト移動は現在サポートしていない。

1.2.3. 半語

半語処理の方法は、3種類提供されている(ldh/exth/sth)。 ldhは、ソース・オペランドの2バイトを、符号なし数として零詰めでワードに拡張し、GVレジスタに格納する。 exthは、ソースの2バイトを、符号つき数としてワードに符号拡張し、 GVレジスタに格納する。 sthは、ソースの下位2バイトを、デスティネーションの2バイトに書き込む。半語についても、FLATS2はロード・ストア・アーキテクチャであり、半語演算やメモリ間半語移動は現在サポートしていない。

1.2.4. 全語

FLATS2における、最も基本的な演算単位。 32ビットの整数演算/論理演算がGV命令、SP命令の両方で可能である。

1.2.5. 倍語

全語二つの対。64ビット整数として、オペランドに使用可能。 SP命令を用いて、64ビットの整数演算/論理演算が実行できる。

1.2.6. 多倍長語

複数のワードからなる配列。FLATS2には、このワード列を多倍長の整数として扱うためのプリミティブ命令が用意されている。用意されているプリミティブ命令は、多倍長加算用が2つ、減算用が2つ、乗算用が2つである。(除算は、加減乗算の組合せで対処する。) これらの命令とその使用法については、数値演算用特殊命令の項を参照すること。

1.2.7. 単精度浮動小数点数

32ビット長の浮動小数点数。符号部 1ビット、指数部 8ビット(-127から+127)、仮数部 24ビット(0.5から1.0)からなる。 DEC社のF浮動小数点数に準拠しているので、正確な定義についてはDECのマニュアルを参照すること。

1.2.8. 倍精度浮動小数点数

64ビット長の浮動小数点数。符号部 1ビット、指数部 8ビット(-127から+127)、仮数部 56ビット(0.5から1.0)からなる。 DEC社のD浮動小数点数に準拠しているので、正確な定義についてはDECのマニュアルを参照すること。

1.2.9. 単精度複素数

2つの単精度浮動小数点数の対。見かけは倍精度複素数と同じで、64ビット長である。単精度複素数用基本算術演算命令には、移動、加減算、実数倍、乗算プリミティブ(2命令)がある。この型は、特殊命令のうちでも複合演算のオペランドになっていることが多い。

2. データ表現

本節では、前節で説明した各種のデータ型が、実際のハードウェア資源上でどのように表現されているかを概観する。

データを保持するハードウェア資源は、大きく3つに分類できる。 GVレジスタ、SPレジスタ、メモリである。以下この順で、各データ型の各ハードウェア資源上のデータ表現について説明を進める。

命令のオペランドとしては、この他に命令内イミディエイトが使用できるが、イミディエイトは通常データとは言わないので、ここでは触れない(命令セットの章を参照のこと)。

2.1. GVレジスタ上の表現

GVレジスタは、基本的に、整数/論理演算やアドレス計算にしか使用しない。従って、浮動小数点系のデータ表現については、ここでは述べない。

2.1.1. ビット

ビット命令は、オペランドをワードのアドレス+ワード内オフセットで指定する。従ってビット型は、レジスタ上では32ビットまとめて、ワードとして扱われる。

2.1.2. バイトと半語

バイトや半語は、レジスタの下位(8ビットないし16ビット)に置かれる。上位ビットには、レジスタへのロード時にゼロか符号ビットが拡張される。バイトや半語をメモリにストアする際にも、当然レジスタの下位が格納される(このとき上位の24ビット/16ビットは無視される)。

2.1.3. 全語と倍語

全語はレジスタと同じ長さなので、そのままレジスタに格納される。倍語はレジスタ2つ分の長さなので、レジスタ対に格納される。

レジスタ対とは、連続したレジスタ番号を持つ2つのレジスタのことである。ただしレジスタが対になるには、レジスタ番号が偶数とその次の奇数でなければならない。例えば、GR0, GR1はレジスタ対だが、 GR1, GR2はレジスタ対ではない。

レジスタ対を指定するには、レジスタ番号を指定すればよい。指定されたレジスタと対になるレジスタは、暗黙のうちにオペランドとして使用される。このとき、陽に指定されたレジスタが倍語の下位ワードとして使用され、対をなすレジスタが上位ワードとして使用される。

GVレジスタ上の倍語は、ロード/ストアやアドレス・ペアの演算には使用できるが、SP命令の倍語演算オペランドとしては使用できない。

2.2. SPレジスタ上の表現

SPレジスタは、SP命令のオペランドとして使用される。 SPレジスタにはアドレス・タグがないので、アドレスを格納することはできない。また、SP命令にはビット/バイト/半語を扱う命令がないので、これらのデータ表現については本節では触れない。

以下の節では、これら以外の数値型の表現について、順次説明してゆく。

2.2.1. 全語と倍語

全語は、レジスタの下位32ビットに置かれる。このとき一般には、上位の32ビットは無意味になる。倍語の幅は、レジスタの幅(データ部の幅)と等しいので、そのままレジスタ上に保持される。

SPレジスタ(のデータ部)を整数として使用する際には、タグ部は一般に無意味な値になる。従って、レジスタに整数をロードしてから、浮動小数点演算にそのレジスタを使用した場合、結果は予測できない。

2.2.2. 単精度・倍精度浮動小数点数

単精度・倍精度を問わず、浮動小数点数はSPレジスタ上では、一定の内部形式に展開されて保持される。

このため、SPレジスタ上の浮動小数点数には型の互換性がある。すなわち、単精度浮動小数点演算のデスティネーションがSPレジスタなら、その演算結果を型変換せずに、そのまま倍精度浮動小数点演算に使用できる。この逆も可能である。

この内部形式は、仮数部64ビット/指数部12ビット/タグ 4ビットからなる。仮数部は、(整数と同様に)レジスタのデータ部に保持される。指数部はタグ部の下位12ビット(bit11-0)に、タグはその上の4ビット(bit15-12)に保持される。タグ部の残り16ビット(bit31-16)は、ここでは無意味である (この部分は単精度複素数で使用される)。

仮数部は、正規化されていることを前提とし、 0.5以上1.0未満の数を表す固定小数点形式になっている。データ型としての浮動小数点数では、仮数部は24ビットないし56ビット(うち hidden bit が1ビット)であるが、内部表現に展開する際にhidden bitは最上位に補われ、下位ビット(40ないし8ビット)には0がパディングされる。

指数部は、符号を持つ12ビットの整数である。データ型としての浮動小数点数では、指数の幅が8ビットで128のオフセットが加わっているが、内部表現に展開する際にこのオフセットは除かれ、符号が拡張されて12ビットの指数となる。

タグ部の4ビットは、以下のような意味を持っている:

bit 0	符号
bit 1	(内部的に使用)
bit 2	アンダーフロー
bit 3	オーバーフロー

符号は、浮動小数点数の符号そのもので、負のとき1。アンダーフローは、演算結果がアンダーフローした時か、このデータがゼロである時に1になる。オーバーフローは、演算結果がオーバーフローした時か、このデータが予約オペランドである時に1になる。

2.2.3. 単精度複素数

単精度複素数は、2つの単精度浮動小数点数の対である。これら2つの浮動小数点数は、それぞれ複素数の実部と虚部に相当する。

これら2つの数も、SPレジスタ上では展開されて、2組の内部表現になる。内部表現のうち、データ部は64ビットを等分して32ビット2組として用いる。このとき、実数部の仮数が下位を、虚数部の仮数が上位を使用する。タグ部32ビットも16ビット2組に等分して、下位16ビットを虚数部が、上位16ビットを実数部が使用する。タグ部の上位下位と仮数部の上位下位が逆転しているのは、間違いではない。ハードウェアの仕様である(ハードウェア削減の都合上こうなっている)。

レジスタ上で複素数がどう表現されていようと、最終的に複素数移動命令でメモリ上に書き戻せば、正しく2つの単精度浮動小数点数に戻る。また、レジスタ上で複素演算を行なっている限り、上記の内部表現については意識しなくて良い。内部表現が問題になるのは、内部表現をそのまま退避しようとする場合 (コンテクスト・スイッチ時など)だけである。

2.3. メモリ上の表現

FLATS2におけるデータのメモリ表現は、基本的にDECのVAX等に準拠したものとなっている。しかし詳細を見てゆくと、 FLATS2の表現は必ずしもDECの表現と同じではないので、メモリ表現に依存したデータ参照には注意が必要である。

2.3.1. ビット

ビットは32ビットまとめてワードとして扱われるため、実際にはビット型のメモリ表現と言うものは存在しない。ワードのメモリ表現とワード内のビット表現があるだけである。

しかし、FLATS2では仮想的に、最下位ビットが最下位アドレスにあると考えて良い。実際にビット命令を使用する際には、メモリがビット・アドレスでないため、ワード・アドレスとワード内オフセットを指定する必要がある。これについても、メモリがバイト・アドレスであるに

も関わらず、ワード・アドレスを使用することに注意してほしい。

2.3.2. バイト、半語、全語、倍語

これらのデータ型については、データの表現はDECのVAX等と同じである。 FLATS2のメモリはバイト・アドレスであるので、バイトは指定されたアドレスに存在する。半語の場合、下位1バイトが指定されたアドレスに、上位1バイトが(指定されたアドレス+1)番地に置かれる。全語も同様で、最下位バイトが指定アドレスに、最上位バイトが(指定アドレス+3)番地に置かれる。倍語では、最下位バイトが指定アドレスに、最上位バイトが(指定アドレス+7)番地である。

しかし、FLATS2ではVAXと違って、データのアラインメントが必要である。すなわち、半語のアドレスは2の倍数、全語のアドレスは4の倍数、倍語のアドレスは8の倍数バイト境界にならなければならない。この制限を犯してデータにアクセスすると、マシンはアドレス境界例外を発生する。

2.3.3. 多倍長語

多倍長語は、32×nビット長の整数を表現する、全語の配列である。この、長さn語の整数を32ビット毎に分割し、下位のワードから順にメモリ上に配置したものを、多倍長整数と呼ぶ。最上位ワードの最上位ビットが、この多倍長語の符号と解釈される。

この表現は、バイトから倍語にいたるデータ型の自然な拡張になっている。実際、倍語と長さ2ワードの多倍長語は、メモリ上で全く同じ表現を持つ。ただしアラインメント条件だけは別で、倍語は8バイト境界、2倍長語は4バイト境界になる。

多倍長語を扱う命令は加減乗算用プリミティブだけをサポートし、タイト・ループ内で使用することを前提に設計されている。これに対して、倍語はハードウェアで一度に処理するので、演算の種類が豊富で実行も速い。

2.3.4. 単精度浮動小数点数

単精度浮動小数点数は、4バイト長であるので、基本的に全語(ワード)と同じメモリ表現を持つ。アラインメントの条件も同じである。

単精度浮動小数点数をワードとして見たとき、最上位ビットが符号、続く8ビットが指数、下位23ビットが仮数である。これら32ビットが、ワードと同じ表現でメモリに格納される。

ところが、このメモリ表現はVAXのそれとは少々異なっている。 VAXの場合、単精度浮動小数点数の第0バイト(最下位バイト)が(指定アドレス+2)番地に、第1バイトが(指定アドレス+3)に、第2バイトが(指定アドレス)番地、第3バイト(最上位バイト)が(指定アドレス+1)番地に格納される。従って、FLATS2とVAXでは、データ型はコンパチブルでも、データ表現が若干異なっている。しかしこのことは、FLATS2の浮動小数点データをバイナリで VAXのデータと比較しない限り、特に問題とはならない。

2.3.5. 倍精度浮動小数点数

倍精度浮動小数点数は、8バイト長であるので、基本的に倍語と同じメモリ表現を持つ。アラインメントの条件も同じである。

倍精度浮動小数点数を倍語として見たとき、最上位ビットが符号、続く8ビットが指数、下位55ビットが仮数である。これら64ビットが、倍語と同じ表現でメモリに格納される。

ところが、このメモリ表現はVAXのそれとは少々異なっている。 VAXの場合、単精度浮動小数点数の第0バイト(最下位バイト)が(指定アドレス+6)番地に、第1バイトが(指定アドレス+7)に、第2バイトが(指定アドレス+4)番地、第3バイトが(指定アドレス+5)番地、第4バイトが(指定アドレス+2)番地、第5バイトが(指定アドレス+3)番地、第6バイトが(指定アドレス)番地、第7バイト(最上位バイト)が(指定アドレス+1)番地に格納される。従って、FLATS2とVAXでは、データ型はコンパチブルでも、データ表現が若干異なっている。しかしこのことは、FLATS2の浮動小数点データをバイナリで VAXのデータと比較しない限り、特に問題とはならない。

2.3.6. 単精度複素数

単精度複素数は、それぞれ実数部と虚数部を表現する、単精度浮動小数点数2つの対である。これら2つの数は、実数部が指定アドレスに、虚数部が(指定アドレス+4)番地に置かれる。各浮動小数点数のメモリ表現は、前節で説明した通りである。

プロセッサ状態とプロセッサ資源

まず、本章で使う言葉を定義しよう。プロセッサ状態とは、プロセッサの命令処理に影響を与えて動作結果を左右するような、プロセッサの内部状態（論理的実体）を言う。また、プロセッサ資源という言葉は、プロセッサからプログラム的に(即ち命令を使って)使用可能な、プロセッサの内部状態(物理的実体;レジスタ)のことを指す。

メモリも広い意味ではプロセッサ資源であるが、本章ではメモリはプロセッサ外の実体であると考えて、以下の説明から除外する。また、レジスタはプロセッサの内部状態であってプログラム的に変更できるからプロセッサ資源であるが、GVレジスタは実体がメモリであるから、本章の説明からは原則的に除外する。

FLATS2は、プロセッサ数２の循環パイプライン計算機であるから、プロセッサ固有の内部状態はそれぞれ二組ずつ存在する。（当然、プロセッサ状態も二組存在する。）また、これらのプロセッサは一つのハードウェアを共有しているので、各プロセッサ固有の資源以外にも、プロセッサ間で一部の資源を共有している。これらの共有資源をアクセスする時には、ソフトウェア的なロックが必要である。

以下、本章では、FLATS2のプロセッサ状態とプロセッサ資源について説明してゆく。まず最初にプロセッサ状態について述べ、次にユーザ状態で操作できるプロセッサ資源について述べる。続いて、システム状態で扱うプロセス資源についてまとめる。

3. プロセッサ状態

プロセッサ状態には、以下のものが含まれる:

1) プログラム・カウンタ (Program Counter ; PC)
2) カレント・フレーム・ポインタ (Current Frame Pointer ; CFP)
3) コンディション・コード (Condition Code ; CC)
4) プロセッサ・ステータス・ワード (Processor Status Word ; PSW)

このうち、ユーザ状態で直接操作できるのは、 PC、CFP、CCだけである。 PSWは特権命令で操作する。本節では、以下これらのプロセッサ状態について、詳細を述べる。

3.1. プログラム・カウンタ

プログラム・カウンタ(PC)は、プロセッサが現在実行している命令へのポインタである。したがってPCの値は、常にI空間上のアドレスになっている。

つまり、PCのアドレス・タグは常に1、上位2ビット(PC<31:30>)は常に2(2進の10)。下位30ビット(PC<29:00>)が、命令メモリ内でのオフセットを示している。 (I空間は Double Word Address である。)

3.2. カレント・フレーム・ポインタ

カレント・フレーム・ポインタ(CFP)は、プロセッサが現在使用中のローカル・レジスタ・フレームへのポインタである。すなわちCFPの値は、V空間上のアドレスになっている。

したがって、CFPのアドレス・タグは常に1、上位2ビット(CFP<31:30>)は常に3(2進の11)である。中位24ビット(CFP<29:06>)はレジスタ・メモリ内でのレジスタ・フレーム番号を示している。また、下位6ビット(CFP<05:00>)には、 PSの下位6ビット(PS<05:00>)と同じものが入っていて、プロセッサ状態の一部がCFPと共に退避されるようになっている。これは、例外発生時に、PSの一部をPC/CFPと共に自動的に退避する必要があるからである。

CFP<29:06>の指すレジスタ・フレームには、32本のフルワード (= 32 bit data + 1 bit address-tag)・レジスタが含まれている。レジスタ・フレームはV空間上に連続して並んでいるが、各レジスタ・フレームは独立しており、オーバーラップはしていない。また、V空間を線形に連続してアクセスする方法は、プログラム的には提供されていない。

3.3. コンディション・コード

コンディション・コード(条件コード ; CC)は、演算結果のタイプ情報を示す 4 ビットのフラグである。各ビットの位置と内容は、以下のようになっている。

CC<0>	N flag	符号(Negative)
CC<1>	Z flag	ゼロ(Zero)
CC<2>	V flag	桁あふれ(オーバーフロー;Overflow)
CC<3>	C flag	桁上がり(キャリー;Carry)

各条件コードの意味は通常のマシンと同様である。また、上位28ビットは無意味である(常に0)。整数演算/論理演算/浮動小数点演算、GV命令/SP命令を問わず、条件コードは同一のものを使用する。

3.4. プロセッサ・ステータス・ワード

プロセッサ・ステータス・ワード(プロセッサ状態語;PS)は、プロセッサの実行状態を示す様々なフラグを含む。幅は32ビットであるが、その全てが使われているわけではない。下位の6ビットは、例外時やcall時に、 CFPの下位6ビットとして自動的に退避される。 ret命令が実行された場合も、CFPの下位6ビットから復帰される。

各ビットの位置と意味は、以下のようになっている。

PS<0>	PRV	特権ビット;プログラムの実行モードを示す
PS<1>	INTMSK	割り込み禁止状態を示す

PS<2>	TRC	トレース例外の要求
PS<3>	HALT	プロセッサが停止中であることを示す
PS<5:4>	(予約)	(未使用)
PS<6>	FV	浮動小数点オーバーフロー例外許可
PS<7>	IV	整数演算オーバーフロー例外許可
PS<12:8>	TRAP_ID	最も最近発生したトラップの例外番号
PS<13>	IM_REAL	IMのアドレス変換抑止ビット
PS<14>	GV_REAL	GVMのアドレス変換抑止ビット
PS<15>	DM_REAL	DMのアドレス変換抑止ビット
PS<16>	NO_PERR	パリティ・エラー例外抑止ビット
PS<20:17>	(予約)	(未使用)
PS<21>	MCHNO	プロセッサ番号
PS<22>	IPIREQ	プロセッサ間マスク可能割り込み要求
PS<23>	IPNREQ	プロセッサ間マスク不能割り込み要求
PS<31:24>	GFID	グローバル・フレーム番号

3.4.1. 特権ビット

プログラムの実行モードを示すビット。ユーザ・モードの時に0、特権モードの時に1。

3.4.2. 割り込みマスク

プロセッサが割り込みを禁止しているかどうかを示す。 0の時は割り込み許可、1の時は割り込み禁止。 1の時には、外部割り込みもプロセッサ間割り込みも抑止される。どちらかを選択的に抑止する方法は、提供されていない。

例外発生時には自動的に1にセットされる (元の状態はCFPと共に退避される)ので、特権状態で割り込みを可能にしたい場合は、プログラム内で陽にこのビットをクリアしなければならない。

3.4.3. トレース・ビット

プロセッサが、トレース例外を発生するか否かを示すビット。このビットが1のときは、命令が正常に実行を終了する毎にトレース例外が発生し、トレース例外ハンドラに制御が移る。

3.4.4. ホールト・ビット

このビットが1であると、プロセッサは命令の実行を行なわない。すなわち、命令のフェッチや実行を行なわず、現在の内部状態を保持し続ける。この状態をホールト状態と呼ぶ。

ホールト状態のプロセッサは、外部からの割り込みを受けると命令の処理を再開して、例外処理を開始する。ただし、この場合も割り込みマスク・ビットは有効なので、割り込み要求を受け付けない(保留する)こともある。

例外処理開始時には、ホールト・ビットの状態がCFPと共に退避されるので、例外処理ハンドラからRET命令で復帰すると、プロセッサは再びホールト状態に戻る。これを避けるには、RET前にCFP内のホールト・ビットをクリアしなければならない。

3.4.5. オーバーフロー例外許可ビット

演算命令の実行結果がオーバーフローしたとき、これらの許可ビットが1であると例外を発生する。整数演算でのオーバーフローと浮動小数点演算でのオーバーフローは、それぞれ独立に例外を許可できる。

実際にオーバーフロー例外が発生した場合、発生する例外の種類(番号)は例外を発生したハードウェアの種類によって定まる。即ち、オーバーフローが乗算器で発生すれば、演算の種類が整数乗算であれ浮動小数点乗算であれ同じ例外ハンドラ(PU_ARITH)に制御が移る。これは、例外ハンドラにとっては不利な仕様であるが、ハードウェア実装上の都合でこうなっている。

3.4.6. 例外番号

最も最近発生した例外の例外番号が保持されている。

3.4.7. アドレス変換抑止ビット

IM、DM、GVM用にそれぞれ1ビットずつ、計3ビットある。これらのビットが1であると、対応するメモリをアクセスする際にアドレス変換作業が抑止される。即ち、アクセス・アドレスを物理アドレスであると解釈して、アクセスを行なう。

これらのビットはアドレス変換部にのみ作用し、アドレス計算部には作用しない。つまり、アドレス計算やアドレス範囲検査は、これらのビットの値にかかわりなく行なわれ、メモリ・アクセス・エラー等のアドレス変換にともなう例外事象も通常通り発生する。

3.4.8. パリティ・エラー例外抑止ビット

メモリのパリティ・エラーが発生したとき、このビットが0であれば、原因(IM/DM/GVM)に応じた例外が発生する。通常このビットは0にしておくが、パリティ・エラーの解消を試みる場合、エラーを無視するためにこのビットをセットする。

3.4.9. プロセッサ番号

このプロセッサのプロセッサ番号。

FLATS2は循環パイプライン計算機であるから、等価な複数の論理的計算機(プロセッサ)が、一つのハードウェアを共有しているように見える。このフィールドは、これら等価なプロセッサに固有のIDを与える(数える)。

3.4.10. プロセッサ間割り込み要求ビット

これらのビット(IPNREQ, IPIREQ)は、それぞれプロセッサ間割り込み(IPCNMI, IPINTR)の要求が保留されていることを示す。すなわち、これらのビットの値が1になっていると、プロセッサ割り込みが許可され次第、例外を発生して処理を開始する。

IPCNMIは基本的にマスク不能なので、 IPCNMIより優先順位の高い例外要因が存在しない限り、即座に例外を発生して処理が開始される (処理開始後は0にリセットされる)。

IPINTRは割り込みマスク・ビットでマスク可能なので、割り込み禁止状態の間 IPIREQビットで割り込み要求を保持し、割り込みを保留する。割り込み保留中にIPIREQビットをクリアすると、割り込み要求は消失し、このプロセッサではこのIPINTRが失われる。

3.4.11. グローバル・フレーム番号

グローバル・フレーム番号(GFID)とは、プロセスが使用するグローバル・レジスタのフレーム番号のことである。この番号は 8 ビット長であるから、0〜255の値を持つ。これらのグローバル・フレームはV空間の最も低位に位置し、例外ハンドラの使用するローカル・フレーム(32×2=64個)を含んでいる。

グローバル・フレーム・ポインタ(Global Frame Pointer ; GFP) という言葉は、GFIDが指定するグローバル・フレームへのポインタを意味する。GFPには、物理的実体はない (GFPレジスタは存在しないし、GFPを扱う命令もない)が、 CFPと対応する既念なので形式的に定義しておく。具体的には、CFPと同じ形式を持つV空間上のアドレスで、以下のようなビット割り当てを持つといえる。

```
GFP<32>     :=  1          /* address tag */
GFP<31:30>  :=  3          /* V space tag */
GFP<29:14>  :=  0          /* The lowest of V space */
GFP<13:06>  :=  GFID<7:0>  /* Frame ID */
GFP<05:00>  :=  0          /* No meaning */
```

GFPの指すグローバル・フレームや、CFPの指すローカル・フレーム、および例外ハンドラが使用する例外処理用ローカル・フレームは、すべてのプロセッサで共有の単一仮想空間(V空間)上に存在する。すなわち、アドレスさえ指定すればグローバルにでもローカルにでも、またいずれのプロセッサからでも、使用することができる。

4. プロセッサ資源

以下の節では、プロセッサ資源の性質および使用法、そして、資源への具体的なアクセス方法について説明する。

プロセッサ資源には、ユーザ・プロセスに属するもの(ユーザ・レベルの資源)と、それ以外のもの(システム・レベルの資源)がある。ユーザ・レベルの資源にはユーザ・モードでアクセスすることができるが、システム・レベルの資源には特権モードで特権命令を用いてアクセスしなければならない。

システム・レベルの資源は、さらに、各仮想プロセッサに固有のもの(PSW)と、物理プロセッサに属するもの(仮想プロセッサ間で共有されるもの) に分類される。共有資源に対するアクセス要求(すなわち特権命令の実行)はプロセッサ間で競合する可能性があるが、この場合ハードウェアによってアービトレーションが行なわれ、要求が早かった命令から順に実行が行なわれる。ただし、資源へのアクセス順序が重要である場合は、プロセッサ間で陽にソフトウェア的同期をとる必要がある。このための命令(不可分命令)は、命令セットに含まれている。

4.1. ユーザ・レベルのプロセッサ資源

ユーザ・レベルのプロセッサ資源には、以下のものが含まれる:

1) PC レジスタ (Program Counter Register ; PCR)
2) CFP レジスタ (Current Frame Pointer Register ; CFPR)
3) CC レジスタ (Condition Code Register ; CCR)
4) グローバル・レジスタ (Global Registers ; GR0〜GR31)
5) ローカル・レジスタ (Local Registers ; LR0〜LR31)
6) S P レジスタ (SP registers ; S, P, T, Q, R, U)

ユーザ・モード・プロセスの内部状態は以上で全てであるから、これらのプロセッサ資源を退避/復帰することで、ユーザ・モード・プロセスのコンテクスト・スイッチを行なうことができる。

PCレジスタは、物理的には30ビットの幅を持つレジスタである。しかし、命令を用いてアクセスする場合、アドレス・タグとI空間のタグがレジスタ値の上位に補われて、全体としては全語の実体として扱われる。 PCの値は、call命令の実行によって得ることができる。また、jump命令を使うと、 PCに所望の値を書き込むことができる。書き込む値は、正当なI空間アドレスでなければならない。

CFPレジスタは、物理的には24ビットの幅を持つレジスタである。しかし、命令を用いてアクセスする場合、上位にアドレス・タグとV空間のタグが補われて、全体としては全語の実体として扱われる。 CFPの値は、call命令またはldcfp命令によって得ることができる。また、stcfp命令によって、 CFPに値を書き込むことができる。書き込む値は、正当なV空間アドレスでなければならない (不当な値を書き込もうとすると、例外が発生する)。

CCレジスタは、物理的には4ビットの幅を持つレジスタである。 CCレジスタの値は、ldcc命令によって読み、stcc命令によって書き込むことができる。

ローカル・レジスタ/グローバル・レジスタは、 mov命令で読み書きできる。ただし、ローカル・レジスタ・フレームは、CFPの変更と同時に切り替わるので退避が省略できることが多い。

SPレジスタも通常のmov命令で読み書きできる。ただし、S/P/T/Qレジスタは浮動小数点数を内部形式に展開して保持するため、物理的には96ビットの幅を持っている。従って、通常のmov命令を使うと、これらのレジスタの値(浮動小数点数)をメモリに書き込む際に、内部表現からメモリ表現への変換(丸めと詰め込み)が発生し、データの精度が損なわれる。これを防ぐためには、内部表現の仮数部(64ビット)とタグ部(計32ビット)を、それぞれ別の命令で(2回に分けて)退避/復帰しなければならない。そのための命令は、別に用意されている(ldtag/sttag命令)。 R/Uレジスタは各66ビットの倍長語レジスタで、内部表現は持たない。

4.2. システム・レベルのプロセッサ資源

システム・レベルのプロセッサ資源には、以下のものが含まれる:

1) PS レジスタ (Program Status Register ; PSR)
2) IM キー・メモリ (IM Key Memory ; imkey)
3) GV キー・メモリ (GV Key Memory ; gvkey)
4) DM キー・メモリ (DM Key Memory ; dmkey)
5) IM LRU メモリ (IM LRU Memory ; imlru)
6) DM LRU メモリ (DM LRU Memory ; dmlru)
7) IODEV メモリ (IODEV Memory ; idv)

PSRはプロセッサごとに独立に存在する。キー・メモリとLRUメモリは仮想プロセッサ間で共有され、IODEVメモリは仮想プロセッサ及びSVP間で共有されている。

PSRは、ldps/stps 命令を使用して読み/書きする。 PS内のビットの持つ意味については、前節を参照のこと。ユーザ・レベルの資源に加えてPSRを退避/復帰すれば、プロセスのコンテクスト・スイッチが実現できる。

IM キー・メモリは ldimkey/rmimkey/stimkey 命令によって、同じく GV キー・メモリは ldgvkey/rmgvkey/stgvkey 命令によって、そして DM キー・メモリは ldkey/rmkey/stkey 命令によって、アクセスを行なう。各キー・メモリは、アドレス10ビット、データ16ビットの構成を持つ。データ部のビット割り当て、アドレスのマッピング等については、メモリ管理の章を参照すること。

IM LRU メモリは ldimlru/stimlru 命令、同じく DM LRU メモリは ldlru/stlru 命令を使用してアクセスする。各LRUメモリは、アドレス10ビット、データ16ビットの構成を持つ。各命令の仕様やLRUメモリの解釈については、メモリ管理の章を参照のこと。

IODEVメモリは、getidv/putidv命令でアクセスする。 IODEVメモリは、アドレス10ビット、データ幅16ビットの共有メモリで、 FLATS2プロセッサとSVP間の通信・割り込みに使用される。IODEVの使い方については、「SVP/MCU解説書」を参照すること。

メモリ管理

本章では、FLATS2マシンのメモリ構造と管理方法について述べる。実際の操作方法(命令)の詳細については、各命令のドキュメントを参照すること。

5. メモリ・マップ

FLATS2の論理アドレス空間は、大きく3つに分割されている。この分割は、論理アドレスの上位2ビットに応じて行なわれ、各空間は以下のような名前を持つ。

00または01	D空間(データ空間)
10	I空間(命令空間)
11	V空間(レジスタ空間)

FLATS2ではアドレス・タグを用いてアドレスを数値と区別しているため、どの空間のアドレスでも形式は同じである必要がある。そのため、これらの空間は全く独立であるにも関わらず、(見かけ上)一つの連続した論理アドレス空間上にマップされている。その意味では、アドレスの上位2ビットは、アドレスの型を区別する一種のワード内タグであると考えてよい。

実際には各空間は、アクセス方法からデータの配置まで全てが異なっている。例えば、アクセスする空間に応じて、アクセスに使用する命令は変えなければならない。選択した命令の使途と、使用したオペランドのアドレス空間が一致しない場合には、例外が発生する。

5.1. I空間

I空間は、命令列を格納する命令空間である。アドレスの下位30ビットは、命令の位置(66ビットを単位としたオフセット値)を表している。

I空間の最下位には、例外ハンドラのエントリが並んでいる。すなわち、例外番号(0から31)に応じて、それぞれ対応する例外ハンドラの入口アドレスが用意されている。例外ハンドラのエントリ・アドレスは、以下のように定義される:

```
entry<31:30>  :=  2
entry<29:05>  :=  0
entry<04:00>  :=  TRAP_ID<4:0>
```

ここでTRAP_IDは、PSW内のフィールドである(PSWの項参照)。

例外ハンドラのエントリは、プロセッサ間で共有されている。これは、メモリ共有型の対称的マルチ・プロセッサとしては、自然な仕様であると考えられる。プロセッサによって異なった処理を行ないたい場合、ハンドラ内のコードで陽にディスパッチする必要がある。

例外ハンドラ・エントリを含むページは、常にメモリ内に常駐させておく必要がある。これを怠った場合、無限にトラップがかかり続けてプロセッサがハング・アップする可能性がある。この時は、外部からのいかなる割り込みも利かなくなるので、SVPからプロセッサをホールトして対処するしかない。

各プロセッサは、プログラムに従ってPCを更新してゆくが、PC(30ビット幅)の桁溢れを検出していない。従って、PCは永久にI空間を指し続け、実行を続ける。

また、I空間にはBLスキーマのようなプロテクト機構が無いので、基本的にI空間内のどのアドレスのコードでも実行できる。しかし、命令ではレジスタ相対ジャンプが不可能なことと、絶対アドレスには必ずアドレス・タグが付いていることにより、実行時の暴走を防いでいる。(ユーザは、プログラムでI空間のアドレスを自由に生成することができない。) このプロテクト機構は完全ではないが、完全なプロテクトを施せば実装コストが上昇することと、コードの共有が不便になる可能性があることを勘案して、このような仕様を定めた。

5.2. V空間

V空間は、レジスタ・フレームを格納するレジスタ空間である。アドレスの第29ビットから第6ビットが、V空間内のレジスタ・フレームの番号を表す。

V空間の最下位には、例外処理フレームとグローバル・フレームが存在する。それより上位は、ローカル・フレームに自由に割り当てることができるが、これらの空間はプロセッサ間(またプロセス間)で共有されているので、割り当てには注意が必要である。V空間にはBLスキーマのような保護機構がないが、ユーザは(I空間同様)V空間のアドレスを自由に生成することができないので、実質的には他のプロセッサ(プロセス)のフレームを破壊することはできない。

5.2.1. 例外処理用フレーム

V空間の最下位には、例外ハンドラが使用するレジスタ・フレームがアロケートされている。これらのフレームは、プロセッサ別に用意しなければならないので、プロセッサ別にアドレスが異なっている。同様に、例外処理用フレームは例外要因別に用意しなければならないので、例外処理用フレームはプロセッサあたり32個存在する。

例外処理フレームのCFP(V空間内のアドレス)は、以下の通りである:

```
TRAP_CFP<31:30>  :=  3
TRAP_CFP<29:12>  :=  0
TRAP_CFP<11>     :=  MCHNO
TRAP_CFP<10:06>  :=  TRAP_ID<4:0>
TRAP_CFP<05:00>  :=  0
```

MCHNO、TRAP_IDは、PSW内のフィールドである。

これらの例外ハンドラ用フレームは、V空間の下位にあって、常にメモリ上に常駐している必要がある。これを怠ると、プロセッサがハングアップする可能性がある。

5.2.2. グローバル・フレーム

グローバル・フレームはV空間の最下位に存在し、256個用意されている。そのプロセッサが使用するグローバル・フレームは、PSW内のGFID部(8ビット;値は0〜255)で決定される。このとき、GFIDとV空間内のアドレス(GFP)との対応は以下のようになっている:

```
GFP<31:30>  :=  3
GFP<29:14>  :=  0
GFP<13:06>  :=  GFID<7:0>
GFP<05:00>  :=  0
```

このアドレスからわかるように、グローバル・フレームは例外処理フレームを全て包含している。

また、グローバル・フレーム群はプロセッサ間で共有されている。これは、プロセスのプロセッサ間移動(migration)を容易にするためである。すなわち、PSR(GFID)をコピーすれば、物理的なプロセッサがどれであれ、グローバル・フレームは同じレジスタが使用されるので、レジスタの内容をコピーする必要がなくなる。

5.3. D空間

D空間は、プログラムのデータを格納するデータ空間である。D空間のアドレスは、最上位ビットが0であって、残る31ビットはそのデータのD空間内でのオフセット(バイト単位)を表している。

D空間の最下位には、例外処理時に情報が書き込まれる領域(例外データ領域)が存在する。この領域は、基本的には物理アドレスと論理アドレスを一致させて、常にメモリ上に常駐させておく必要がある。これを怠ると、プロセッサがハング・アップする可能性がある。

例外データ領域のアドレスは、以下の通りである:

```
TRAP_DAT<31>    :=  0
TRAP_DAT<30:9>  :=  0
TRAP_DAT<8>     :=  MCHNO
TRAP_DAT<7:3>   :=  TRAP_ID<4:0>
TRAP_DAT<2:0>   :=  0
```

MCHNOとTRAP_IDは、PSW内のフィールドである。例外データ領域は、一つの例外番号について2ワード(66ビット)必要なので、アドレスの下位3ビットは0になる。また、これらの例外データ領域はプロセッサ毎に別個に持たねばならないので、MCHNOで例外データ領域自体をプロセッサ数ぶん確保する。

これより上位のメモリは、原則的に自由にプロセスに割り当てることができる。しかし例外データ領域のすぐ上には、例外ハンドラが使用するデータ領域を置いて、この部分も(例外データ領域同様)物理アドレスと論理アドレスを一致させて、メモリに常駐させておくことになってい

る。

6. メモリの構成と物理アクセス

FLATS2では、論理空間の分割にそれぞれ対応して、物理メモリも別々に存在している。すなわち、I空間に対応してIメモリ、V空間に対応してGVメモリ、D空間に対応してDメモリが存在する。物理メモリがこのように分割されているのは、パイプラインの各ユニットから並行してメモリにアクセスすることを許し、メモリからの転送幅を広げるためである。

これら3つの物理メモリは、それぞれセット・アソシアチブ型キャッシュの形式を持つ。特にIメモリとDメモリの構成は良く似ていて、共に 4 way set associative cache の構成になっている。各ウェイ(メモリ・バンク)は、66ビット幅で深さ64Kのデータ・メモリと、16ビット幅で深さ1Kのキー・メモリからなっており、さらにキーの一致を判定するアソシエーション論理が内蔵されている。

また、キャッシュの形式としてはコピーバック方式をとっているので、そのブロックが書き込みによって変更されているか否かを、「dirty」ビットによって表示する。この情報は、ハードウェア的には独立したdirty memory (1bit×1K)に記録されるが、命令上はキーと一緒に扱われる(キーの16ビット目として扱う)。また、IメモリとDメモリには複数のバンクが存在するので、セット毎にLRUを管理するために、LRUメモリとLRU論理が外付けされている。

以下の節では、まずメモリの物理構成を簡単に説明し、それらに付された物理アドレス、およびそのプログラミング・モデルについて概説する。

6.1. キー・メモリ

キー・メモリは、アドレス幅10ビット/データ幅16ビットのメモリであるが、命令上はdirtyメモリを含めて17ビット幅のメモリとして扱う。また、キー・メモリのアドレスには、データ・メモリのアドレス(16ビット)の上位10ビットを与える。つまり一つのキー・エントリは、連続する64倍語のデータ・メモリを代表していることになる。この、各キー・エントリが担当するデータ・メモリの単位を、データ・ブロックと呼ぶ。メモリ管理上の最小管理単位は、この「ブロック」である。

結局、キー・メモリのデータ17ビットの内容は以下の通りである:

keydat<16>	dirty
keydat<15>	parity
keydat<14>	valid
keydat<13:00>	key

dirtyは、このデータ・ブロックが書き込みで変更されていることを示す。parityは、キー・メモリ<15:00>のパリティ。 validは、このデータ・ブロックが使用可能であることを示す。 keyは、このデータ・ブロックの仮想アドレス(の一部)である。これらのビットの詳細な意味については、後の節(アドレス変換の詳細)を参照のこと。

キー・メモリのアクセスは、メモリの物理アドレスを使用して行なう。ある物理アドレスでキー・メモリをアクセスすると、その番地を含むメモリ・ブロックのキー・エントリが参照される。

6.2. LRUメモリ

Iメモリ、Dメモリはマルチ・ウェイのキャッシュであるので、ブロックを入れ換える際には、何らかの方策を用いて、どのウェイのブロックを追い出すか決定しなければならない。FLATS2では、これをLRUアルゴリズムに従って決定している。

LRUアルゴリズムでは、各キー・エントリについて「各ウェイが最後に使用された時間が一番古いもの」を LRUウェイ(Least Resently Used Way)として、追い出しの対象に選ぶ。このために、FLATS2ではLRU更新論理とLRUメモリを設けて、各キー・エントリについて「各ウェイが最後に使用された時間の順序」を覚えておく。

この方法の実現には、Linked-List方式を用いている。この方法は、設計の現場では比較的ポピュラーな方法であると考えられるが、実際にこの実装方法に言及した文献は少ないので、本節ではFLATS2の実装を例にとって、簡単に方針をまとめておく。

FLATS2のLRUメモリは、深さ1K幅6ビットのメモリである。各ビットの名前と位置は、以下のようになっている:

bit 5	LRUarc01
bit 4	LRUarc02
bit 3	LRUarc03
bit 2	LRUarc12
bit 1	LRUarc13
bit 0	LRUarc23

各ウェイ(0から3)をグラフの頂点として完全グラフを作った場合、弧の数は6になり、上記の6ビットに1対1で対応する。それらの弧に方向を与えることによって、弧の同端の頂点に順序を与えることができる。LRUメモリの各ビットの名前(LRUarcij)は、そのビットがウェイ i からウェイ j への弧であることを表す。各ビットの値は、ウェイ i がウェイ j よりも新しい(最近アクセスされた)場合に、1になる。この6ビットによって、ウェイ 0, 1, 2, 3は新しい順に順序付けられている。他の全てのウェイから指されている(全ての弧が自分を向いている) ウェイが、LRUウェイである。

4つのものを並べる方法は $4! = 24$ 通りであるから、本当は、これらの順序関係は5ビットで表現できる。そこを敢えて、冗長に6ビットにエンコードすることが、この実装方法の核心である。LRU更新論理は、ウェイ i をアクセスする際、アクセスと同時に、ウェイ i につながっている弧の向きを全て変える (自分が弧の出発点になるようにする)。これによって、ウェイ i が「最も最近使用されたウェイ」であることを表現する。各弧は、ウェイ間の新旧関係を表しているから、変更されなかった弧については元の情報 (残る3ウェイ間の新旧関係)が保存されている。従って、LRUメモリへの書き込み1回(読み出しなし)で、正しく4ウェイ間の新旧関係を更新することができる。

こうして、LRU更新論理は常時メモリ・アクセスを監視して、各データ・ブロックでウェイ間の時間的順序関係を更新してゆく。そして実際に、あるアドレスについてLRUウェイが知りたい場合には、対応するアドレスのLRUメモリを読み出して、その値をデコードすればLRUウェイの番号がわかる。 FLATS2では、LRUメモリ読み出し命令を実行すると、ハードウェア(LRUウェイ検出論理)がデコードを行なうが、もちろんLRUメモリの内容6ビットを用いてソフトウェアでデコードしてもよい。

6.3. Iメモリの構成

Iメモリは、データ幅66ビット深さ64Kのメモリ・バンクを4つ持つ、4 way set associativeのメモリである。仮想アドレスでアクセスする際には、同じアドレスを各バンクに与えてアドレスの連想処理を行ない、キーが一致した(ヒットした)ウェイのデータが有効となる。どのウェイでもヒットしなかった場合(ミス・ヒット)は、ページ・フォールト処理を行なって、当該ブロックをいずれかのウェイ(通常LRUウェイ)にロードしなければならない。

I空間は倍語単位でアドレスするので、物理アドレスも倍語単位に付されている。具体的には、各ウェイに割り振られている物理アドレスは、以下の通りである:

ウェイ 0	$00000000～$0000FFFF
ウェイ 1	$00010000～$0001FFFF
ウェイ 2	$00020000～$0002FFFF
ウェイ 3	$00030000～$0003FFFF

物理アドレスは単なる整数に過ぎないので、アドレス・タグやアドレス空間のタグ(上位2ビット)を持たない。また、上の表では物理アドレスの上位14ビットを全て0として表したが、実際には現在の実装では上位14ビットは無視されている。

6.4. Vメモリの構成

Vメモリは、データ幅66ビット深さ64Kのメモリであるが、読みだしポートを3つ書き込みポートを1つ持つ、マルチポート・メモリである。物理的には、同じメモリ(66bit×64K)を3つ並列に並べ、書き込み時には常に同じデータを書き込むようにして、全てのウェイで内容が同一になるようにしてある。これによって、並列に3つのレジスタを読みだすことが可能になる。

Vメモリでも、キー・メモリを用いてアドレス変換を行なっている。ただしVメモリでは、4 wayでなく、1 way set associativeである。これは、IメモリやDメモリが 1 read/1 write portで 4 way set associativeであるのに対して、Vメモリでは3つのウェイを組退させて常に同じ内容を保持しているからである。実際、実装に使用したメモリ・ボード自体は、I/V/Dのどのメモリでも同一である。

Vメモリでも、アドレス変換の処理は、他のメモリと同様に行なう。アドレスの連想処理を行ない、キーが一致(ヒット)すれば、そのブロックのデータが有効である。ヒットしなかった場合(ミス・ヒット)は、ページ・フォールト処理を行なって、当該ブロックをメモリ上にロードしなければならない。ただ、Vメモリではアドレスのマップ方法が、少々IやDと異なっている。これについては、次の節(アドレス変換の詳細)も参照して欲しい。

V空間は、CFPを介してレジスタ・フレーム単位にアドレスする。そのため、物理アドレスも32ワード単位に付されている。また、物理アドレスの長さはCFPの長さと同じ24ビットである(上位8ビットは無視される)。さらに、Vメモリは全体が二分されて、下位半分がV空間最下位の常駐レジスタに、上位半分が仮想化レジスタに使用される。このマッピングを簡単にまとめると、以下のようになる:

常駐レジスタ群	$000000～$0007FF
仮想レジスタ群	$000800～$000FFF

常駐レジスタ群とは、例外処理フレームやグローバル・フレーム等の、常にメモリ上に存在しなければならないフレームのことを言う。これらのレジスタはV空間の最下位に存在し、仮想アドレスと実アドレスを一致させておかなければならない。物理アドレスまたは仮想アドレスの上位13ビットが0である場合は、そのフレームは常駐レジスタ群であると解釈される。

一方、仮想レジスタとは、必要に応じてスワップ・アウトすることが可能なレジスタ (ユーザのローカル・レジスタ等)をいう。物理アドレスまたは仮想アドレスの上位13ビットが0でない場合は、そのアドレスは仮想レジスタを指すものとして解釈される。

6.5. Dメモリの構成

Dメモリは、Iメモリ同様、データ幅66ビット深さ64Kのメモリ・バンクを4つ持つ 4 way set associative のメモリである。仮想アドレスでアクセスする際には、同じアドレスを各バンクに与えてアドレスの連想処理を行ない、キーが一致した(ヒットした)ウェイのデータが有効となる。どのウェイでもヒットしなかった場合(ミス・ヒット)は、ページ・フォールト処理を行なって、当該ブロックをいずれかのウェイ(通常LRUウェイ)にロードしなければならない。

D空間はバイト単位でアドレスするので、物理アドレスもバイト単位に付されている。各ウェイに割り振られている物理アドレスは、以下の通りである:

ウェイ 0	$00000000～$0007FFFF
ウェイ 1	$00080000～$000FFFFF
ウェイ 2	$00100000～$0017FFFF
ウェイ 3	$00180000～$001FFFFF

物理アドレスは単なる整数に過ぎないので、アドレス・タグやアドレス空間のタグ(上位2ビット)を持たない。また、上の表では物理アドレスの上位11ビットを全て0として表したが、実際には現在の実装では上位11ビットは無視されている。

D空間最下位の例外データ領域は、常にウェイ0に置かれる。すなわち、例外時には、例外データ領域のアドレスを実アドレスとして解釈して、例外データの書き込みを行なう。従って、例外データ領域のある部分は、つねに仮想アドレスを実アドレスと一致させておく必要がある。

7. アドレス変換の詳細

本章では、FLATS2のアドレス変換処理の仕組みを説明する。同時に、その仕組みを用いて、実際にはどのような方針でメモリ管理を行なったら良いか、概観する。

まず最初に、各空間についてアドレス変換の仕組みを解説し、最後にページ・フォールト時の処理をどうするか、方針を示す。

7.1. I空間でのアドレス変換

I空間のアドレスは、アドレス変換において、以下のように解釈される:

address<31:30>	I空間のタグ(=2)
address<29:16>	アドレス・キー
address<15:06>	データ・ブロック番号
address<05:00>	データ・ブロック内オフセット

I空間は倍語単位にアドレスがついているので、アドレス変換は比較的単純でわかりやすい。

Iメモリは4ウェイあるが、処理方法はいずれのウェイでも同じである。

1) まず、データ・ブロック番号をキー・メモリのアドレスとして与え、キーの内容を読み出す。
2) アドレス・キーとキー・メモリの内容を比較する。比較結果が等しくて、そのキー・エントリがvalidならば、このウェイはヒットしている。全てのウェイがミス・ヒット(ヒットしなかった)ならば、アドレス変換例外を発生する。
3) データ・ブロック番号とブロック内オフセットをデータ・メモリのアドレスに与えて、ヒットしたウェイのデータ・メモリにアクセスする。ヒットしていない(ミスした)ウェイでのアクセスは無効にする。

7.2. V空間でのアドレス変換

V空間のアドレスは、アドレス変換において、以下のように解釈される:

address<31:30>	V空間のタグ(=3)
address<29:17>	アドレス・キー
address<16:08>	データ・ブロック番号
address<07:06>	データ・ブロック内オフセット
address<05:00>	(PSの一部なので無関係)

V空間はフレーム単位にアドレスがついているが、さらに命令内のレジスタ番号フィールドを用いないと、最終的なアクセス対象(レジスタ)が決定できない。

レジスタのアクセス・アドレス決定は、以下のような手順で行なわれる:

1) 命令内から、レジスタ番号フィールド6ビットを取り出す。レジスタ番号の最上位ビットが0ならばGFP、1ならばCFPを、V空間のアドレスとして使用する。
2) アドレスのアドレス・キー部が0ならば、キー・メモリのアドレスの最上位ビットに0を与える。アドレス・キーが0でないなら、最上位ビットは1にする。さらに、アドレスのデータ・ブロック番号をキー・メモリのアドレスの下位9ビットに与え、キーの内容を読み出す。
3) アドレス・キーとキー・メモリの内容を比較する。比較結果が等しくて、そのキー・エントリがvalidならば、このレジスタはメモリ上に存在する。メモリ上にこのアドレスが存在しなければ、アドレス変換例外を発生する。
4) データ・メモリのアドレスに、以下のようなデータを与えて、メモリにアクセスする。

   最上位ビットには、キー・メモリに与えたアドレスの最上位ビットと同じ値を与える。次の9ビットには、データ・ブロック番号。次の2ビットには、ブロック内オフセット。最下位の4ビットには、レジスタ番号の4ビット目から1ビット目まで。(レジスタの最下位ビットは、倍語内のワード・オフセットである。)

7.3. D空間でのアドレス変換

D空間のアドレスは、アドレス変換において、以下のように解釈される:

address<31:19>	アドレス・キー
address<18:09>	データ・ブロック番号
address<08:03>	データ・ブロック内オフセット
address<02:00>	倍語内オフセット

D空間では、最上位ビットにあるD空間のタグが(address<31>は0のはずである)、アドレス・キーの中にも含まれている。また、D空間はバイト単位でアドレス付けされているので、最下位3ビットは倍語内のバイト・オフセットになる。

Dメモリは4ウェイあるが、処理方法はいずれのウェイでも同じである。

1) まず、データ・ブロック番号をキー・メモリのアドレスとして与え、キーの内容を読み出す。
2) アドレス・キーとキー・メモリの内容を比較する。比較結果が等しくて、そのキー・エントリがvalidならば、このウェイはヒットしている。全てのウェイがミス・ヒット(ヒットしなかった)ならば、アドレス変換例外を発生する。
3) データ・ブロック番号とブロック内オフセットをデータ・メモリのアドレスに与えて、ヒットしたウェイのデータ・メモリにアクセスする。ヒットしていない(ミスした)ウェイでのアクセスは無効にする。
4) この命令が、ビット命令/バイト命令/半語命令ならば、倍語内オフセットを用いて、必要なデータを倍語から切り出す。

7.4. ページ・フォールト処理の概略

アドレス変換例外が発生した場合は、通常ページ・フォールトであるから、スワップ等の処理を行なって命令を再試行しなければならない。本節では、ページ・フォールトの処理手順について、簡単に方針だけを示す。

1) アドレス変換例外が発生すると、フォールトを起こしたアクセス・アドレスが、ハンドラに渡される。
2) スワップの対象になるページ(またはブロック)を決定する。この際、必要があればLRUメモリの内容を参照する。 FLATS2では、メモリは連想キャッシュ方式であってページ・マップ方式ではないため、スワップの候補は高々ウェイ数(4)しかない。
3) ステップ2)で選んだページをinvalidにする。具体的には、rmkey/rmimkey/rmgvkey命令を用いて、当該アドレスのキーのvalidビットをクリアする。これで、このページに対するアクセスは不能となる。同時に、このページに対応するページ・マップ・エントリに、ページ・アウト中のマークをつける。
4) SVPに、当該ページのページ・アウト(DMA転送)を依頼する。具体的には、IODEVを用いて要求をSVPに送る(SVPに割り込む)。ここで、(通常は)このプロセスは転送終了までsleepする。
5) ページ・アウトが終了したら、ページ・マップを更新して、今度はフォールトを発生したページをページ・インする。具体的には、SVPからページ・アウト終了の割り込みが入ったら、追い出したページのページ・マップ・エントリとこれから取り込むページのエントリを適切に更新する。そして、SVPにページ・インの要求を行なう (この手順はページ・アウトと同様)。
6) ページ・インが終了したら、取り込んだページのページ・マップ・エントリを使用可能状態にする。この時、同時にこのページをロックして、ページ・アウトを禁止しておく。これは、フォールトを起こした命令の再試行が成功するまで、このページをメモリ上に固定しておくためである (そうしないとデッド・ロックを起こす可能性がある)。
7) 対応するキー・メモリの内容を正しく設定し、 validビットも立てる。この時から、このページは使用可能になる。
8) フォールトからリターンして、ページ・フォールトを起こした命令を再実行する。

9) 命令が正常に実行されたら、ロックしたページのロックを解除しておく。ステップ8)で、リターン時にトレース・ビットを立てておけば、命令実行が成功した時点でトラップするので、そこで解除することができる。しかし、解除は遅くても(メモリの使用効率は下がるが) 解除を正しく行なえば問題はない。

例外処理

本章では、FLATS2における例外処理について述べる。まず、その概要とメカニズムについて説明し、続いてFLATS2の例外要因について詳説する。

8. 例外処理の概要

8.1. 用語の説明

通常、命令の処理はプログラマがプログラミング時に指定した順序で行なわれる。しかしそれ以外に、外部的または内部的要因によって命令の処理順序が変更される場合がある。

こうした状況を 例外状況 と呼び、例外状況を扱うために処理順序を変更して必要な処理を行なうことを 例外処理 と呼ぶ。また、例外状況を引き起こした要因を、 例外要因 と呼ぶことにする。例外処理のうちでも、内部的例外要因によって発生する例外を 割り出し と呼び、外部的例外要因によって起こるものを 割り込み と呼ぶ。割り出しの代表的な例は、トラップ命令によるプログラムの割り出しであり、割り込みの代表的な例は、SVPによる外部割り込みである。

さらに、例外処理にはもう一つの分類基準がある。それは、例外を起こしたプロセスが停止する状態に基づいた分類である。例外によってプロセスの内部状態が一部失われ、再開不能になる場合、そのような例外を アボート と呼ぶ。プロセスが再開可能である場合でも、例外を発生した命令の副作用がプロセス状態に残る場合を トラップ と呼ぶ。副作用が残らず、例外を発生した命令から再試行できる場合を フォールト と呼ぶことにする。アボートの代表例はメモリのパリティ・エラー。トラップの代表例は浮動小数点演算時のオーバーフロー。フォールトの代表例はメモリのページ・フォールトである。

例外要因が命令外にある場合、すなわち割り込みにおいてはフォールトとトラップを区別することは無意味であるが、以下では便宜上トラップに含めることにする。

8.2. 例外ベクタ

FLATS2の場合、ハードウェア的には32種の例外が区別される。これら32種の例外はそれぞれ固有の番号(例外ベクタ番号 ; 0~31)を持ち、それぞれ固有の例外処理プログラム(例外ハンドラ)にディスパッチされる。例外ハンドラ・プログラムの入口は、命令空間の低位アドレスにベクタ番号順に並んでいる。同様に、各例外ハンドラの使用するレジスタ・フレームも、レジスタ空間の低位に番号順に並んでいる。また例外発生時には、データ空間の低位アドレスを介して、各ハンドラに例外要因についての情報が引き渡される。このために使用するデータ領域(例外データ領域)も、データ空間の低位アドレスに番号順に並んでいる。

以下に、FLATS2における例外ベクタを示す。現在使用されていない番号は空けてあるが、将来使用される可能性がある。

表 : FLATS2の例外ベクタ

名称	番号	説明
RESET	0	/* Hardware Reset */
HARDWARE	1	/* Serious Hardware Error */
PU_ARITH	4	/* Arithmetic Error in Multiplier */
SM_ARITH	5	/* Arithmetic Error in Rounding */
GV_ERROR	7	/* GV : Memory Error */
ILLOPR	8	/* Illegal Operand */
LISP_SVC	9	/* Lisp Service Request */
GV_ARITH	10	/* Arithmetic Error in GV unit */
ADRCAL	11	/* Error in Address Calculation */
ADRBND	12	/* Address Boundary Error */
DM_ERROR	13	/* DM : Memory Error */
DM_MLTHIT	14	/* DM : Multi-way Hit */
PF_DM	15	/* DM : Page Fault */
SU_ARITH	17	/* Arithmetic Error in ALU */
TRACE	18	/* Trace Trap */
SVP_NMI	19	/* External Non-Maskable Interrupt */
SLF_NMI	20	/* Internal Non-Maskable Interrupt */
IM_ERROR	22	/* IM : Memory Error */
IM_MLTHIT	23	/* IM : Multi-way Hit */
PF_IM	24	/* IM : Page Fault */
ILLINST	25	/* Illegal Instruction */
PRVVIO	26	/* Priviledge Violation */
PF_GV	27	/* GV : Page Fault */
SYSCALL	29	/* System Call */
SLF_INT	30	/* Internal Interrupt Request */
SVP_INT	31	/* External Interrupt Request */

8.3. 例外処理時の動作

例外処理は、次のような順序で行なわれる。

1) 要因に応じた例外番号を決定する。複数の例外要因が同時に存在する場合は、優先順位が高いものから処理する。優先順位は、基本的には発生時間の古い順に高くなっているが、ハードウェア・エラー等はその限りでない。
2) CFPを、例外番号によって定まる値に切り替える。これによってローカル・レジスタ・フレームが、例外番号に対応する例外処理ハンドラ固有のフレームに切り替わる。すなわち、ユーザのローカル・レジスタ・フレームがアクセス不能になり、保護される。 GFPは変わらないので、グローバル・レジスタは切り替わらない。
3) 例外を発生したプロセスのCFPとリターン・アドレスを、例外処理ハンドラのローカル・フレーム(0~1番)に退避する。退避されるCFPの下位6ビットには、プロセッサ・ステータスの一部 (例外処理時に変更される部分)が含まれている。
4) 例外を発生したプロセスが、例外処理開始時に実行していた命令のアドレス演算部の出力を、例外データ領域に退避する。退避するアドレスは、例外番号と1:1に対応して定まっている。この情報は、例外処理ハンドラに対する引数に相当する。 (例えばページ・フォールト時のアクセス・アドレスは、この方法で例外ハンドラに引き渡される。)
5) プロセッサ・ステータスを、特権状態かつ割り込み禁止にし、ホールト状態とトレース状態を解除する。
6) 例外番号に対応した例外処理ハンドラの入口に飛び込む。すなわち、例外処理ハンドラのエントリ・アドレスをPCに書き込む。以後は、例外処理ハンドラのプログラムに制御が移る。

例外処理が終了したら、RET命令を用いて元のプロセスに復帰する。

9. 例外処理の実現

FLATS2はパイプライン計算機であるから、例外要因はパイプライン内のいろいろなステージで発生する。パイプライン内では同時に複数の命令が処理されているために、例外要因は例外を発生した命令だけでなく、後続命令の実行制御にも影響を及ぼす。

以下では、FLATS2において各種の例外がどのように処理されているか、論理設計者の立場から説明する。本稿の目的はハードウェアの仕様を記述することではないので、処理の概要と方針だけを述べることにする。 FLATS2のパイプライン処理については、基本的な知識があるものと仮定して話を進める。

9.1. 例外処理方法の分類

例外要因は、パイプライン内のいろいろな箇所で発生する。パイプライン内では最大5命令が並行して処理されているが、2つの命令流(プロセッサ)の命令が交互に投入されるため、パイプライン内で相互に干渉する可能性があるのは最大3命令までである。そこで、以下では、発生した例外要因がいくつの命令に干渉しうるかという観点から、例外要因を3つに大別する。

第一に、『即座に例外処理が開始される』例外要因がある。例外要因がパイプラインの入口近くで発生した場合、まだ後続の命令はパイプラインに投入されていない。従って、プロセッサは例外を発生した命令の次の命令から、直ちに例外ハンドラの実行を開始することができる。つまり、この場合、例外要因が後続の命令実行に影響を及ぼすことはない。これをタイプ0と呼ぶ。

第二に、『後続の命令を取り消してから例外処理を開始する』場合がある。これをタイプ1と呼ぶ。この場合、後続の命令は既にパイプラインに投入されているが、まだ後続命令の副作用がプロセッサ状態に残っていないので、例外処理に移る前に後続命令の実行を取り消すことができる。従ってタイプ1の例外要因では、後続の命令が一つ取り消され、例外要因の発生から1命令サイクル遅れて例外処理が開始される。

同様に、後続の命令を取り消してから例外処理に移るが、取り消す命令が2つである場合もある。これをタイプ2と呼ぶ。タイプ2では、例外要因が発生した時点で既に2つの後続命令がパイプラインに投入されているため、これらを取り消してから例外処理を開始する。この場合、例外処理の開始は、タイプ0に比べて2命令サイクル遅くなる。

タイプ0(とタイプ1の一部)の例外処理では、例外を発生した命令自体の副作用を完全に抑止することができるので、フォールト処理が可能である。ただし、フォールト処理が可能な場合でも、トラップ処理を行なうことが望ましい場合には、当然トラップ処理が行なわれる。タイプ2(とタイプ1の一部)では、例外要因発生時に既に命令の副作用が残っているため、フォールト処理ができない(トラップ処理になる)。

さて、以上の説明では、例外要因がパイプライン内部から発生することを前提とした。しかし、割り込みについては、例外要因が基本的にパイプライン外部から発生するため、割り出しとは別な説明が必要である。また、ハードウェア障害などによる例外(アボート処理)についても、本節では触れなかった。次の節では、これらについても簡単に触れることにする。

9.2. 例外処理の概要

前節での分類に従って、各タイプの例外処理がどのように実現されているか、概説する。

9.2.1. タイプ0の例外処理

例外要因がパイプラインのフェーズ2(P2)までに検出された場合、タイプ0の例外処理が行なわれる。タイプ0の例外には、IM_ERROR、IM_MLTHIT、PF_IM、 ILLINST、PRVVIO、PF_GV、SYSCALLなどが含まれる。これらのうち、SYSCALLだけがトラップで、残りはフォールトである。

例外要因がP2までに検出されると、P3では以下のような手順で例外の処理が行なわれる。まず、優先順位に従って例外ベクタが判定され、ベクタ番号に応じたジャンプ先(次のPC)とCFPの値が決定される。さらに、新たな(次の命令で用いられる) プロセッサ状態も作成される。それと同時に、例外を発生した命令が「無効」であることを宣言し、パイプライン下流の論理に対して、命令処理の実行を抑止することを要求する。「無効宣言」と共に、以下の処理が例外処理で

あることを宣言して、次段以降の論理に例外処理を要求する。

P4では、例外を発生した命令のPCとCFPとPSRから、例外ハンドラに渡す(PC/CFP)ペアを作成する。このペアは、P5で例外ハンドラのレジスタ・フレームに書き込まれる。一方、P4は次の命令のP0であるから、P3で作成された PCやCFPを用いて、例外処理ハンドラの実行を開始する。以後は、通常の命令処理と同様である。

9.2.2. タイプ1の例外処理

例外要因がパイプラインのフェーズ3(P3)からフェーズ7(P7)までに検出された場合は、タイプ1の例外処理が行なわれる。タイプ1の例外のうち、フォールト処理されるものは、GV_ERROR、ILLOPR、LISP_SVC、GV_ARITH、ADRCAL、 ADRBND、DM_MLTHIT、PF_DM。トラップ処理されるものは、DM_ERROR、SU_ARITH、 TRACEである。

タイプ1の例外要因の内、P5までに検出されたものはフォールト処理を行なうことができる。P5でこれらの要因が存在していると、「取り消し要求」と「無効宣言」が行なわれる。「無効宣言」によって、以後この命令の副作用は抑止される。また、「取り消し要求」によって、後続の命令(命令のフェッチとデコードが進んでいる)は取り消される。取り消しは、実際にはP7(後続命令のP3)まで遅延されてから、後続命令に対して無効宣言を行なうことで実現される。また、例外の処理は、取り消された後続命令の実行サイクルを利用して行なわれる。すなわち、後続命令への取り消し要求が、後続命令への例外処理要求を兼ねる形で実現されている。

タイプ1の例外でも、トラップ処理を行なう場合は、例外を発生した命令の無効宣言が出ない。従って、例外を発生した命令は最後まで実行され、副作用を残す。後続命令に対する取り消し要求は出るので、後続命令の動作は、フォールト時と全く同じである。

9.2.3. タイプ2の例外処理

例外要因がパイプラインのフェーズ8(P8)からフェーズ9(P9)までに検出された場合は、タイプ2の例外処理が行なわれる。タイプ2の例外は、全てトラップである。タイプ2の例外には、PU_ARITH、SM_ARITHがある。

タイプ2の例外処理の実現は、後続命令がタイプ1の例外をフォールトとして処理する場合と性質同じである。例外を発生した命令は、P9では取り消すことができないので、最後まで実行され副作用を残す。しかし、先行する命令のP9は後続命令のP5にあたるので、後続命令のタイプ1(フォールト)と同様に、取り消し要求と無効宣言で処理することができる。

結局、後続命令が無効となり、さらにその後に続く命令が取り消されて例外処理を行なうことになる。例外ハンドラに分岐するのは、さらにその後の命令となる。

9.2.4. その他の例外処理

例外要因は以上で全てではない。この他、割り込みに起因する例外(RESET、SVP_NMI、SVP_INT、SLF_NMI、SLF_INT)や、ハードウェア異常(HARDWARE)などがある。

通常の割り込みについては、外部からの割り込み要求(リクエスト)をハード的にホールドしておき、実際に割り込みを受け付けた段階で応答(アクノリッジ)を返して、リクエストをクリアする。すなわち、「ハンドシェイク」を行なっている。割り込みの受け付けは、各プロセッサが命令処理の合間に割り込み要求をチェックする形で実装されている。従って、SVP_NMIやSLF_NMIのようなマスク不能割り込みをかけても、プロセッサの状態が破壊されることはない。(つまり、割り込みで命令処理のアボートが起こることはない。)

一方、ハードウェア障害(HARDWARE)やリセット割り込み(RESET) については、命令の実行中であっても強制的に制御を奪い取るように、実装されている。従って、命令処理がアボートされることがある。

10. 例外処理の詳細

各例外番号について例外要因は一つとは限らないし、また、例外処理もそれぞれ固有の特徴を持っている。そこで本節では、FLATS2における32種の例外処理について、ひとつひとつ見てゆくことにする。本節では、特にシステム・プログラマの立場から見た特徴に留意して話を進めるが、例外処理の実現に関する基本的な知識はあるものと仮定する。

以下の説明では、各例外処理について、例外の名称、内容の概略、例外ハンドラへの引数(例外データ領域や例外ハンドラのレジスタ・フレームから得られる情報)、考えられる例外要因、例外処理の方針、の順で説明を進める。

10.1. リセット

名称 RESET

概略 リセット・スイッチを押すことによって生じる例外。強制的システム初期化に用いる。

種別 アボート。

引数 なし。

要因 ハードウェア・リセット・スイッチをオンにした。

処理 システムの初期化を行なって、システムを再スタートする。この例外からは、復帰することはないと考えている。

10.2. ハードウェア障害

名称 HARDWARE

概略 ハードウェアに緊急性の高い障害が発生した。

種別 アボート。

引数 なし。

要因 具体的な要因は、ハードウェアの実装設計者に問い合わせること。ハードウェア製造業者の予約例外番号である。

処理 多分、ソフトウェアでは何もできない。パニックのメッセージを出し、最低限の内部状態を退避した後、マシンを停止する。この例外からは、復帰できない。この例外を発生したプロセスは、中断される。

10.3. 命令メモリのパリティ・エラー

名称 IM_ERROR

概略 命令メモリでパリティ・エラーが発生した。

種別 フォールト。

引数 退避されたPCから、エラーを発生したアドレスが特定できる。 IMアクセス命令実行中であった場合、例外データ領域にアクセスしたアドレスが残っている。

要因 命令メモリでのパリティ・エラー。 4 wayあるメモリの、データあるいはキー部のメモリを読み出した結果、パリティが合致しなかった。エラー発生時に、PSのマスク・ビットが1であればこの例外は抑止される。

処理 4 wayの内いずれのウェイであるかは、アクセス・アドレスを実アドレスに変換しないとわからない。また、データ部のエラーかキー部のエラーかは、実際に内容を調べてみないとわからない。命令が長形式であった場合、エラーを起こしたアドレスは退避されたPC+1である場合がある。

フォールトなのでプロセスを再開することはできるが、データまたはキーのメモリ内容が壊れているので、正常に実行を継続することは期待できない。例外を発生したプロセスは中断すべきである。その後、そのプロセスの使用していたページのメモリ・チェックを行なう必要がある。恒久的な故障でなければ、再書き込みによってエラーを解決することができる。

10.4. レジスタ・メモリのパリティ・エラー

名称 GV_ERROR

概略 レジスタ・メモリでパリティ・エラーが発生した。

種別 フォールト。

引数 退避されたCFPから、エラーを出したフレームが特定できる。さらに、退避されたPCの指す命令を読めば、レジスタ番号も3つにまで絞り込むことができる。

要因 レジスタ・メモリでのパリティ・エラー。データ・メモリ(3 way)のいずれか、またはキー・メモリ(1 way)でパリティ・エラーが発生した。エラー発生時に、PSのマスク・ビットが1であればこの例外は抑止される。

処理 フォールトなのでプロセスを再開することはできるが、レジスタ(またはキー)の内容が壊れているので、正常な実行継続は期待できない。例外を発生したプロセスは中断すべきである。その後、そのプロセスの使用していたページのメモリ・チェックを行なう必要がある。恒久的な故障でなければ、再書き込みによってエラーを解決することができる。

10.5. データ・メモリのパリティ・エラー

名称 DM_ERROR

概略 データ・メモリでパリティ・エラーが発生した。

種別 トラップ。

引数 例外データ領域に、アクセスしたデータ・アドレスが残っている。

要因 データ・メモリでのパリティ・エラー。 4 wayあるメモリの、データあるいはキー部のメモリを読み出した結果、パリティが合致しなかった。エラー発生時に、PSのマスク・ビットが1であればこの例外は抑止される。

処理 4 wayの内いずれのウェイであるかは、アクセス・アドレスを実アドレスに変換しないとわからない。また、データ部のエラーかキー部のエラーかは、実際に内容を調べてみないとわからない。 DMをR/Rのモードで使用した場合、 2組目の読み出しデータについてはパリティ・エラーが検出できない (ハードウェア上の制限)。

トラップなので、例外処理ハンドラが起動された時点で既に、パリティ・エラーを起こした命令の実行が終了している。従って、異常なデータを使用した演算結果によって、プロセスの内部状態が変更を受けている。エラーを無視すればプロセスの実行を再開することはできるが、データまたはキーのメモリ内容が壊れているので、正常に実行を継続することは期待できない。例外を発生したプロセスは中断すべきである。その後、そのプロセスの使用していたページのメモリ・チェックを行なう必要がある。恒久的な故障でなければ、再書き込みによってエラーを解決することができる。

10.6. 命令メモリのマルチ・ヒット

名称 IM_MLTHIT

概略 命令メモリにおいて、アドレス変換時に複数のウェイがヒットした。

種別 フォールト。

引数 退避されたPCを読めば、アクセスしたアドレスはわかる。 IMアクセス命令の実行中であった場合は、例外データ領域にアクセス・アドレスが残っている。

要因 命令メモリのヒット検出論理で、マルチ・ヒットが検出された。命令メモリは 4 way set associative 構成を持つため、キー・メモリの設定が誤っていると、アドレス変換時に複数のウェイがヒットすることがある。原因としては、特権ソフトウェアでのキー・メモリ管理の誤りか、キー・メモリのハード・エラー等が考えられる。

処理 いずれにしてもキーの内容は矛盾したものとなっており、正常な実行継続は期待できない。命令のフェッチができないので、この例外を発生したプロセスは中断せざるを得ない。ページ・フォールトとマルチ・ヒットは、アドレス変換が抑止されていれば発生しないので、変換抑止ビットをセットして作業すれば、原因を調べることはできる。基本的には、起こってはならない(はずの)例外である。

10.7. データ・メモリのマルチ・ヒット

名称 DM_MLTHIT

概略 データ・メモリにおいて、アドレス変換時に複数のウェイがヒットした。

種別 フォールト。

引数 例外データ領域に、アクセスしたデータ・アドレスが残っている。

要因 データ・メモリのヒット検出論理で、マルチ・ヒットが検出された。データ・メモリは 4 way set associative 構成を持つため、キー・メモリの設定が誤っていると、アドレス変換時に複数のウェイがヒットすることがある。原因としては、特権ソフトウェアでのキー・メモリ管理の誤りか、キー・メモリのハード・エラー等が考えられる。

処理 いずれにしてもキーの内容は矛盾したものとなっており、正常な実行継続は期待できない。データのフェッチができないので、この例外を発生したプロセスは中断せざるを得ない。ページ・フォールトとマルチ・ヒットは、アドレス変換が抑止されていれば発生しないので、変換抑止ビットをセットして作業すれば、原因を調べることはできる。基本的には、起こってはならない(はずの)例外である。

10.8. 命令メモリでのページ・フォールト

名称 PF_IM

概略 命令メモリのアクセス時にページ・フォールトが発生した。

種別 フォールト。

引数 命令フェッチ時の例外ならば、退避されたPC。 IMアクセス命令実行中の場合は、例外データ領域にアクセスしたアドレスが残っている。

要因 命令メモリでのページ・フォールト。 PC番地またはアクセス・アドレスでミス・ヒットが発生した。命令が長形式であった場合は、 PC+1番地でフォールトした可能性もある。

処理 必要なページをページ・インした後、命令を再試行する (RET命令で元のプロセスに復帰すれば良い)。プロセッサ・ステータスの変換抑止ビットをオンにすれば、この例外は回避できる。

10.9. 不当命令

名称 ILLINST

概略 命令コードが不当である。

種別 フォールト。

引数 退避されたPC。

要因 退避されたPCの指している命令の、オプコード部が不当である。すなわち、オプコード部の値が命令コードとしては未定義である。

処理 命令列が不当なものであるか、長形式命令のイミディエイト・データ部を命令として実行している可能性がある。いずれにしても、エラーの場合実行は継続できないので、プロセスの実行は中断する必要がある。

ソフトウェアのプロトコルによっては、この例外を、エミュレーション・トラップに使用することもできる。退避されたPCの指す命令を読み出して、エミュレータでデコードして処理を行なえば良い。例外の種別はフォールトであるから、エミュレーション実行後はプロセスを再開することができる。ただし、そのままRET命令で復帰すると例外を起こした命令から再実行してしまうので、デコード結果に応じてPC等を加減してから復帰する必要がある。

10.10. 特権違反

名称 PRVVIO

概略 命令の実行権限に違反した。

種別 フォールト。

引数 退避されたPC。

要因 ユーザ・モードで、特権命令を実行しようとした。または、RET命令等のユーザ命令で、ユーザ・モードから特権モードにプロセッサ・モードを変更しようとした。

処理 プロセスの実行を中断するか、適切なエミュレーションを行なった後にプロセスの実行を再開する。

10.11. レジスタ・メモリでのページ・フォールト

名称 PF_GV

概略 現在のCFPが指しているレジスタ・フレームが、実メモリ(レジスタ)上に存在しない。

種別 フォールト。

引数 退避されたCFP。

要因 レジスタ空間でのアドレス変換に伴うページ・フォールト。 CFPの指しているローカル・レジスタ・フレームが、実レジスタ・メモリ上に存在しない。命令がグローバル・レジスタしか使用していない場合でも、CFPが実レジスタ・メモリ上にないページを指していると、発生する。

処理 必要なページをページ・インした後、命令を再実行する。プロセッサ・ステータスの変換抑止ビットをオンにすれば、この例外は回避できる。

10.12. 不当オペランド

名称 ILLOPR

概略 アドレス演算部(GVユニット)で、不当なオペランドが検出された。

種別 フォールト。

引数 例外データ領域に、不当なオペランドが残っている。

要因 オペランドがI空間またはV空間上のアドレスでなければならないとき、それ以外のタイプであった。

処理 プログラム・エラーであるので、ふつうはプロセスを中断する。しかし、種別はフォールトであるので、必要があれば適切なエミュレーションの後に実行を再開することもできる。

10.13. LISPサービス

名称 LISP_SVC

概略 LISPを指向する命令において望ましいような、特定の状況においてトラップを発生する。これによって、LISPにおける Generic Type Operations をサポートする。

種別 フォールト。

引数 オペランド自身は残らない。ただし、不当なオペランドはレジスタ内にあるので、退避されたPCとCFPを用いて捜し出すことはできる。例外データ領域には、異常なオペランドを用いた演算結果（時には異常なオペランドそのもの)が残っているので、参考にすることはできる。

要因 GV系の論理/整数演算命令をLISP指向のモードと共に使用した時、いずれかのソース・オペランドがアドレスであったか、演算結果がオーバーフローを起こした。

処理 動作をエミュレートしてから、実行を再開する。ソースとターゲットには、退避されたPCとCFPを用いてアクセスする。種別はフォールトなので、最後にPCを加減してからRET命令で復帰する。

10.14. GV系数値演算エラー

名称 GV_ARITH

概略 GV系の整数演算命令の結果、オーバーフローが発生した。

種別 フォールト。

引数 オーバーフローを起こした演算結果が、例外データ領域に残る。

要因 GV系数値演算命令で、結果がオーバーフローを起こし、さらにプロセッサ・ステータス内の整数オーバーフロー例外エネーブル・ビットが立っている場合。

処理 適切にエミュレーションを行なえば、フォールトなので実行を再開することもできる。しかし、通常はプロセスの実行を中断する。

10.15. アドレス演算例外

名称 ADRCAL

概略 アドレス演算において、例外処理にトラップが指定されている時、演算オペランドまたは演算結果に異常な値が検出された。

種別 フォールト。

引数 例外を発生したアドレス演算の結果が、例外データ領域に残る。

要因 ありうべき要因は、以下の通り。

1) アドレス演算のソース・オペランドがアドレス同士であった、あるいは整数同士であった。
2) BLレジスタの、ベースまたはリミット・レジスタが、データ空間のアドレスでなかった。
3) アドレス演算の結果が、BLレジスタの指す範囲に入っていなかった。
4) アドレス演算の結果が、データ空間のアドレスではなかった。

処理 適切にエミュレーションを行なえば、フォールトなので実行を再開することができる。しかし、通常はプロセスの実行を中断する。

10.16. アドレス境界例外

名称 ADRBND

概略 データのアクセスに用いた実行アドレスが、オペランドのサイズによって定まっているアクセス境界に合致していない。

種別 フォールト。

引数 例外を発生したアドレス演算の結果が、例外データ領域に残る。

要因 データ・メモリに対して半語/全語/倍長語でアクセスする場合は、実効アドレスがそれぞれ2/4/8の倍数境界を指していなければならない。この制限を満たしていない実行アドレスでデータ・メモリをアクセスすると、この例外が発生する。

処理

10.17. データ・メモリでのページ・フォールト

名称 PF_DM

概略 データ・メモリのアクセス時にページ・フォールトが発生した。

種別 フォールト。

引数 例外データ領域に、アクセスしたアドレスが残っている。

要因 データ・メモリでのページ・フォールト。アクセス・アドレスをアドレス変換しようとして、ミス・ヒットが発生した。

処理 必要なページをページ・インした後、命令を再試行する（RET命令で元のプロセスに復帰すれば良い)。プロセッサ・ステータスの変換抑止ビットをオンにすれば、この例外は回避できる。

10.18. Sユニット系数値演算エラー

名称 SU_ARITH

概略 Sユニット系の算術演算命令の結果、オーバーフローが発生した。

種別 トラップ。

引数 ソース・オペランドがメモリだった場合、そのアドレスが例外データ領域に残る。ソース・オペランドがGVレジスタであった場合、そのデータが例外データ領域に残る。ソース・オペランドがSPレジスタであった場合、例外データ領域に残ったデータは無意味である。オーバーフローした演算結果は、そのままデスティネーション・オペランドに書き込まれている。

要因 Sユニットを用いる数値演算命令で、

1) 整数演算を行なった結果オーバーフローが発生し、なおかつ、プロセッサ・ステータス内で整数オーバーフロー例外がエネーブルされていた。

2) 浮動小数点演算を行なった結果オーバーフローが発生し、なおかつ、プロセッサ・ステータス内で浮動小数点演算オーバーフロー例外がエネーブルされていた。

処理 退避されたPCは、例外を発生した命令を指していない。すなわち、(トラップであるから)実際に例外を起こした命令の実行は終了している。退避されているPCが指しているのは、例外を発生した命令の次に実行すべき命令である。この命令はまだ実行されていないので、RET命令で復帰すればその命令から実行が再開される。

10.19. Pユニット系数値演算エラー

名称 PU_ARITH

概略 Pユニット系の算術演算命令の結果、オーバーフローが発生した。

種別 トラップ。

引数 例外データ領域に残ったデータは無意味である。オーバーフローした演算結果は、そのままデスティネーション・オペランドに書き込まれている。

要因 Pユニットを用いる数値演算命令で、

1) 整数演算を行なった結果オーバーフローが発生し、なおかつ、プロセッサ・ステータス内で整数オーバーフロー例外がエネーブルされていた。

2) 浮動小数点演算を行なった結果オーバーフローが発生し、なおかつ、プロセッサ・ステータス内で浮動小数点演算オーバーフロー例外がエネーブルされていた。

処理 退避されたPCは、例外を発生した命令を指していない。すなわち、(トラップであるから)実際に例外を起こした命令の実行は終了している。退避されているPCが指しているのは、例外を発生した命令の次に実行すべき命令である。この命令はまだ実行されていないので、RET命令で復帰すればその命令から実行が再開される。

10.20. 浮動小数点データのメモリ書き込み時オーバーフロー

名称 SM_ARITH

概略 SPユニットの浮動小数点データ(内部形式)を、メモリに格納するためにメモリ表現に変換しようとした時、オーバーフローが発生した。

種別 トラップ。

引数 例外データ領域に残ったデータは無意味である。オーバーフローした演算結果は、そのままデスティネーション・オペランドに書き込まれている。

要因 SP系浮動小数点演算命令でデスティネーションがメモリだった場合に、演算の結果を内部形式からメモリ表現に変換する際オーバーフローが発生し、さらにその時プロセッサ・ステータス内の浮動小数点演算オーバーフロー例外エネーブル・ビットがセットされていた。

処理 退避されたPCは、例外を発生した命令を指していない。すなわち、(トラップであるから)実際に例外を起こした命令の実行は終了している。退避されているPCが指しているのは、例外を発生した命令の次に実行すべき命令である。この命令はまだ実行されていないので、RET命令で復帰すればその命令から実行が再開される。

10.21. ソフトウェア例外

名称 SYSCALL

概略 TRAP命令による、ソフトウェア的例外。

種別 トラップ。

引数 TRAP命令で指定した引数が、例外データ領域に格納されている。

要因 TRAP命令の実行。

処理 必要な処理を行なってから、RET命令で復帰する。種別はトラップであるから、TRAP命令の次の命令から実行が再開される。

10.22. トレース例外

名称 TRACE

概略 PSのトレース・ビットがセットされた状態で、命令の実行が正常に終了した。

種別 トラップ。

引数 なし。

要因 PSのトレース・ビットがセットされた状態で何らかの命令の実行が正常に終了した時、命令の実行終了を通知するために発生する。

処理 命令が正常に終了したのだから、本来は、何の処理も必要ない。そのままRET命令で復帰すれば、正常に実行が再開される。システム・ソフトウェア(OSやデバッガ)にフックを提供するための例外である。

10.23. マスク不能外部割り込み

名称 SVP_NMI

概略 SVPが、CPUに対してマスク不能な割り込みを要求した。

種別 トラップ。

引数 なし。ソフトウェア的に、SVPとCPUの共有メモリ(IODEV)上でデータを受け渡すことを想定している。

要因 SVPからのマスク不能割り込み要求。

処理 割り込み処理ハンドラで要求内容を解釈し、しかるべき処置をとる。

10.24. マスク不能プロセッサ間割り込み

名称 SLF_NMI

概略 循環パイプライン内のいずれかのプロセッサが、マスク不能なプロセッサ間割り込みを要求している。

種別 トラップ。

引数 なし。特権ソフトウェアで、ソフトウェア的にデータを受け渡すことを想定している。

要因 いずれかのプロセッサが、IPCNMI命令を実行した。

処理 全プロセッサが、ほぼ同時にプロセッサ間通信ハンドラのコードを実行開始する。通信ハンドラで相互排除に注意しながら通信要求の内容を解釈し、然るべき処理を行なう。

10.25. マスク可能外部割り込み

名称 SVP_INT

概略 SVPが、CPUに対してマスク可能な外部割り込みを要求した。

種別 トラップ。

引数 なし。ソフトウェア的に、SVPとCPUの共有メモリ(IODEV)上でデータを受け渡すことを想定している。

要因 SVPから、マスク可能な外部割り込み要求があった。そのとき、プロセッサの割り込みマスク・ビットがクリアされていた。

処理 割り込み処理ハンドラで要求内容を解釈し、しかるべき処置をとる。

10.26. マスク可能プロセッサ間割り込み

名称 SLF_INT

概略 循環パイプライン内のいずれかのプロセッサが、マスク可能なプロセッサ間割り込みを要求している。

種別 トラップ。

引数 なし。特権ソフトウェアで、ソフトウェア的にデータを受け渡すことを想定している。

要因 いずれかのプロセッサが、IPINTR命令を実行した。

処理 割り込み可能状態にあるプロセッサから、プロセッサ間通信ハンドラのコードを実行し始める。通信ハンドラで相互排除に注意しながら通信要求の内容を解釈し、然るべき処理を行なう。割り込み禁止状態のプロセッサも、割り込み可能状態になって、他に優先順位の高い例外が存在しなくなると、割り込みを起こして例外処理を開始する。

プロセッサ間通信の例外処理要求ビットは、プロセッサ状態内に含まれている。従って、割り込み禁止状態でプロセッサ状態を見ると、プロセッサ間通信の処理要求の有無を判定できる。また、割り込み禁止状態でこのビットをクリアすれば、この時の割り込み要求は失われる。

命令セット

本章では、FLATS2マシンの命令セットについて、その概要をまとめる。

まず、FLATS2命令セットの持つ基本的枠組みと命令形式を説明する。さらに、FLATS2命令セットの際だった特徴である、 BLアドレシングについて述べる。 FLATS2のもう一つの特徴である数値演算用特殊命令については、本章では、命令セット内での位置づけだけを述べる。特殊命令の実際の動作と仕様の詳細については、別項『数値演算用特殊命令』を設けて説明するので、そちらを参照されたい。同様に、各命令の仕様と動作の詳細、およびアセンブラ言語レベルの記法については、『命令セット・リファレンス・マニュアル』を参照のこと。

11. 命令形式

FLATS2の命令は、大きく2つのカテゴリーに分類できる。「GV命令」と「SP命令」である。GV命令は命令形式で言えばI/J/K形式の命令で、GVレジスタ間演算や分岐命令、アドレス計算命令、メモリに対するロード命令/ストア命令などを含む。GV命令は、GVレジスタに対するロード・ストア・アーキテクチャを持つ命令セットであるといえる。一方のSP命令は、命令形式としてはM形式を持ち、オペランドにメモリ/GVレジスタ/SPレジスタ/イミディエイト値を指定することができる、いわば汎用型の命令セットである。以下の節では、これらFLATS2の命令形式について、そのあらましを説明する。

11.1. FLATS2命令の骨格

FLATS2の命令には、4つの基本命令形式(I/J/K/M)があり、さらにそれぞれに長形式命令と短形式命令がある。短形式命令は命令語(66 bit)だけからなり、長形式命令では命令語の後ろにイミディエイト値を含むデータ語(66 bit)が続く。長形式命令を表現するときは、基本命令形式の名前にL (long)をつけて呼ぶ。例えば、M形式に対してML形式と呼ぶ。

命令語の下位ワード(33 bit)の解釈は、どの形式でも一定である。

名前	ビット	意味
atagl	32	アドレス・タグ(ここでは無意味)
opcode	31-24	命令コード(アドレシング・モード)
mode	23-16	動作モード(オペランド指定子)
s1	15-11	レジスタ番号(1)
s2	10-6	レジスタ番号(2)
s3	5-0	レジスタ番号(3)

opcodeは、GV命令においては命令コードとして処理の種類を指示し、SP命令においてはオペランドのアドレシング・モードを指定する。modeは、オペランドを指定する。例えば、演算命令ではオペランドの種類/組合せを指定するし、条件分岐命令では分岐判定に使う条件や分岐の種類を指定する。s1/s2/s3は、命令実行時に使用するレジスタまたはレジスタ・ペアの番号を指定する。例えば、レジスタ間演算命令のオペランド・レジスタや、アドレス計算に使うレジスタの番号は、s1/s2/s3で指定される。

以下では説明を簡潔にするため、s1/s2/s3で指定するレジスタをそれぞれr1/r2/r3と呼ぶことにする。また、BLペアや倍語命令でr1/r2/r3とペアになるレジスタを、それぞれr1'/r2'/r3'と呼ぶことにする。具体的には、レジスタ番号の最下位ビットを反転した番号が、そのレジスタとペアになるレジスタの番号になる。例えば、0番のペアは1番、3番のペアは2番である。

さて、命令語の上位ワード(33 bit)の解釈は、各基本命令形式によって異なる。以下の節では、各命令形式で命令語がどの様に解釈されるのか、その概略をまとめる。

11.2. I/IL形式

I形式は、GVレジスタ間演算命令や、CALL/RET/JUMP命令、その他のシステム制御命令で用いる命令形式である。I形式では、命令語の上位ワードがイミディエイト値として用いられる。

名前	ビット	意味
atagh	32	アドレス・タグ
I0	31-0	イミディエイト値

従って、I形式ではアドレスを含む任意の一語が、イミディエイト値としてオペランドに指定できる。IL形式では、さらに命令語に続く2全語(データ語)がイミディエイトとして指定できる。この場合、データ語の下位ワードをI1、上位ワードをI2と呼ぶ。

11.3. J形式

J形式は、条件分岐命令で使用される命令形式である。FLATS2では1命令内で4方向までの多方向分岐を行なうので、J形式命令の命令語の上位ワードは8ビットづつに4分割されて、それぞれが分岐先への命令オフセットとして使用される。ここで、各フィールドは8ビットの符号付き整数として解釈され、-128〜+127の値をとる。分岐先のオフセットがこれよりも大きいときは、長形式の条件分岐命令を使用しなければならない。

名前	ビット	意味
atagh	32	アドレス・タグ(ここでは無意味)
j0	31-24	分岐先オフセット(0)
j1	23-16	分岐先オフセット(1)
j2	15-8	分岐先オフセット(2)
j3	7-0	分岐先オフセット(3)

11.4. JL形式

JL形式は、長形式の条件分岐命令で使用される命令形式である。JL形式の命令では、命令語の上位ワード(J0)、データ語の上位(J1)/下位ワード(J2)が、それぞれ分岐先への命令オフセット(または分岐先の絶対アドレス)として使用される。FLATS2では、1命令内で4方向までの多方向分岐を行なうが、JL形式では、J0/J1/J2と次命令(PC+2番地)の4方向を分岐先として扱う。FLATS2のアドレス空間は32ビット幅なので、JL形式の命令で任意のアドレスに分岐することができる。

11.5. K/KL形式

K/KL形式は、GVのロード/ストア/アドレス計算命令など、GV命令のうちでも、アドレス計算またはメモリ・アクセスを行なう命令で使用する命令形式である。これらの命令に共通していることは、BLアドレシングを行なってアドレスを生成することである。このため、j0/d1/d2フィールドが命令内に含まれている。また、tフィールドでデスティネーションとなるGVレジスタ番号を指定することができる。

まとめると、K/KL形式は、BLアドレシングを行なうGV命令のための命令形式であるということができる。BLアドレシングについては、すぐ後の章で詳しく説明する。

名前	ビット	意味
atagh	32	アドレス・タグ(ここでは無意味)
j0	31-24	分岐先オフセット
L-bit	23	長形式指定ビット
-	22	(未使用)
t	21-16	ターゲット・レジスタ番号
d1	15-8	ディスプレースメント(1)
d2	7-0	ディスプレースメント(2)

j0は、デフォルトの分岐先である。すなわち、命令の実行中に例外が発生しない限り、j0の指す分岐先への分岐が実行される。L-bitが1の場合、命令はKL形式であると解釈される。KL形式では、ディスプレースメントに32ビットのイミディエイト(D1, D2)が用いられ、命令語内のd1, d2は無視される。このときD1/D2には、それぞれデータ語の下位/上位ワードが用いられる。

11.6. M/ML形式

M/ML形式は、メモリを含む一般のオペランドに対して、SPユニットで固定小数点/浮動小数点演算を行なう命令で用いられる。M/ML形式がK/KL形式とよく似ているのは、どちらもメモリ・アクセスにBLアドレシングを用いるためである。M形式とK形式の唯一の相違は、K形式のtフィールドがM形式ではaopフィールド(SPユニットでの演算種類指定)に置き換わっていることである。

M/ML形式を汎用の命令形式と呼ぶのは、命令のオペランドにGV命令より多くのバリエーションがあるからである。例えば、M形式ではオペランドにGVレジスタ/SPレジスタ/メモリ/イミディエイト値を指定することができるが、I形式ではGVレジスタかイミディエイト値のいずれかだけである。また、指定できる演算の種類も、M形式では加減乗除/論理演算/シフト/型変換など広範囲であるのに対して、I/J/K形式では加減/論理演算/アドレス計算だけである。

名前	ビット	意味
atagh	32	アドレス・タグ(ここでは無意味)
j0	31-24	分岐先オフセット
L-bit	23	長形式指定ビット
aop	22-16	SPユニット演算指定
d1	15-8	ディスプレースメント(1)
d2	7-0	ディスプレースメント(2)

M/ML形式では、命令語下位ワードのopcodeフィールドでオペランドのアドレシング・モードを指定し、オペランドの組合せをmodeフィールドで指定し、さらにaopフィールドで演算の種類(move/add/mul等)を指定する。 j0は、デフォルトの分岐先である。すなわち、BLアドレシングが失敗しない限り、j0の指す分岐先への分岐が実行される。

L-bitが1の場合、命令はML形式であると解釈される。 ML形式では、ディスプレースメントとして、 8ビット・イミディエイト(d1, d2)の代わりに 32ビット・イミディエイト(D1, D2)が用いられる。このとき、命令語内のイミディエイトd1, d2は無視される。ML形式のD1/D2には、それぞれデータ語の下位/上位ワードが用いられる。

12. アドレシング・モード

FLATS2では、GV命令/SP命令に共通して、メモリ・オペランドのアドレシングに「BLアドレシング」を採用している。以下では、まずBLアドレシングについて簡単に説明し、そのあとFLATS2の持つアドレシング・モードについて具体的に説明する。

12.1. BLアドレシング

12.1.1. アドレス演算

FLATS2では、メモリおよびGVレジスタ上の全語にアドレス・タグがついているため、データにアドレスと数値の区別(型)がある。アドレス計算では、オペランドにレジスタとイミディエイトを指定することができ、それらのオペランドを加えることによってアドレスを生成するが、この加算において、2つのオペランドの型間の演算規則は以下の通りである。

アドレス	+	アドレス	=	数値
アドレス	+	数値	=	アドレス
数値	+	アドレス	=	アドレス
数値	+	数値	=	数値

3つ以上のオペランドを加える場合は、この規則を繰り返し適用する。

12.1.2. アドレス範囲検査

FLATS2では、データ・メモリをアクセスする際に、アクセスするアドレスが指定した範囲内にあるかどうか、必ずチェックを行なう。この時、アクセス可能な範囲を指定するために、アドレスの下限(ベース・アドレス)と上限(リミット・アドレス)を値として持つ GVレジスタのペア(偶数番号+奇数番号)を指定する。このレジスタ・ペアを「BLペア」と呼ぶ。また、BLペアによる範囲検査をともなうアドレシング方式を、「BLアドレシング」と呼ぶ。

BLペアを指定するには命令内のレジスタ番号フィールドを用いるが、このとき指定したレジスタ番号のレジスタがベース・レジスタになり、その番号の下1ビットを反転した番号のレジスタがリミット・レジスタになる。

BLアドレシングでは、 BLペアおよびアクセス・アドレスが D空間上のアドレスでない場合にはアクセスが失敗する。また、アクセス・アドレスがBLペアで指定された範囲内にない場合も、当然メモリ・アクセスは失敗する。 FLATS2では、 I空間やGV空間ではBLアドレシングを採用していないので、以下の話はデータ・アクセスについてだけ適用される。

BLアドレシングが成功する条件を、以下にまとめておく。ここで、Bはベース・レジスタ、Lはリミット・レジスタ、Aはアクセス・アドレスであるとする。

(1) B, L, Aのアドレス・タグは全て1
(2) B, L, Aの最上位ビットは全て0(D空間)
(3) $B \geq A \geq L$ または $B \leq A \leq L$

これら全ての条件が満たされた場合のみ、アドレス計算の結果が正しくアドレスとなり、メモリ・アクセスが成功する。いずれかが満たされなかった場合、アドレス計算の結果は数値型となり、メモリ・アクセスも失敗する。アドレシング・モードに副作用がある場合、範囲検査の結果に応じてアドレス型または数値型のデータが副作用としてレジスタに書き込まれる。

12.1.3. 命令内分岐

FLATS2では、BLアドレシングの結果に応じて分岐を行なうことができる。具体的に言えば、K/KL/M/MLの各形式の命令には命令内に8ビットの分岐先オフセットがあって、アドレス計算の結果が正しくアドレスになった場合に指定されたアドレスにジャンプする。アドレス計算の結果が数値型になった場合には、ジャンプを行なわずに次の命令を実行する。

ただし、命令内の分岐オフセットが1(K/M形式)または2(KL/ML形式)である場合には、アドレス計算が成功した場合に次命令を実行し、アドレス計算に失敗した場合はアドレス演算例外を発生する。

12.2. 基本的なアドレス計算法

I/J/K形式の命令でアドレス計算を行なう命令は、ロード、ストア、およびアドレス計算命令だけである。ロード/ストア命令では、一命令で1つだけメモリ・オペランドをとることができる。また、副作用付きロード/ストア命令(プッシュ/ポップ)では、アドレス計算の副作用をレジスタに残すことができる。アドレス計算命令では、1命令で2つまでのアドレスを計算することができる。これらの命令において、メモリ・アクセスにともなうアドレス計算では加算を1回、メモリ・アクセスを伴わないアドレス計算では加算を2回まで実行することができる。アドレス計算のオペランドにはGVレジスタかイミディエイト値を使用する。

M形式の命令では、一命令で2つまでのメモリ・オペランドをとることができる。つまり、一命令で2つのアドレス計算を行なうことができる。また、これらのアドレス計算においては、メモリ・アクセスにともなうアドレス計算では加算を1回、メモリ・アクセスを伴わないアドレス計算では加算を2回まで実行することができる。これらのアドレス計算のオペランドには、GVレジスタかイミディエイト値を使用することができ、さらに副作用としてオペランド・レジスタを変更することができる。以下では、これらのアドレシング・モードを大別して、その概要を述べる。

FLATS2の命令セットは直交していないので、1命令内におけるアドレシング・モードの組合せには制限があるが、それらの制限については後の章で述べることにし、以下では各アドレシング・モードの概要だけを説明することにする。以下の文中、BaseRegは範囲検査におけるベース・レジスタの値。 IndexRegは、レジスタ番号フィールドで指定するインデックス・レジスタ。PointerRegは、レジスタ番号フィールドで指定するポインタ・レジスタ。 immediateは、命令内のイミディエイト・フィールドで指定する変位値(displacement)。どのレジスタ番号フィールドをどのように解釈するかは、アドレシング・モードの組合せに依存する。これについては後の章でそのおおよそを述べる。

12.2.1. オフセット・モード

Address = BaseReg + immediate。ベースからのオフセットでアドレスする。副作用がある場合、BaseRegが変更される(Addressになる)。

12.2.2. インデックス・モード

Address = BaseReg + IndexReg。ベース・レジスタにインデックス・レジスタの値を加えて、アドレスを生成する。 FLATS2ではスケーリングは行なわないので、 IndexRegの値はそのままBaseReg値に加算される。副作用がある場合、BaseRegが変更される(Addressになる)。

12.2.3. ポインタ・モード

Address = PointerReg + immediate。ポインタ・レジスタに変位値を加えて、実効アドレスを生成する。計算されたアドレスは、命令で別に指定したBLペアと比較されて、正当なアドレスであるかどうか範囲検査を受ける。副作用がある場合、PointerRegが変更される(Addressになる)。

12.2.4. オフセット・インデックス・モード

Address = BaseReg + IndexReg + immediate。命令で指定するベース・レジスタの値にインデックス・レジスタの値と変位値を加えてアドレスを計算する。副作用がある場合、 BaseRegが変更される(Addressになる)。このモードではアドレス計算に加算が2回必要なので、アドレス計算命令にはこのモードを指定できるが、メモリ・アクセスを行なう命令には使用できない。

12.2.5. ポインタ・インデックス・モード

Address = PointerReg + IndexReg + immediate。命令で指定するポインタ・レジスタに、インデックス・レジスタの値と変位値を加えてアドレスを計算する。計算されたアドレスは、命令で別に指定したBLペアと比較されて、範囲検査を受ける。副作用がある場合、 PointerRegが変更される(Addressになる)。このモードではアドレス計算に加算が2回必要なので、アドレス計算命令にはこのモードを指定できるが、メモリ・アクセスを行なう命令には使用できない。

13. GV命令

この章では、 GV命令のあらましについて説明する。 GV命令は、 GVレジスタに対するロード・ストア・アーキテクチャをもつ、 FLATS2アーキテクチャのサブセット上の命令であると見ることができる。具体的には、 GVレジスタ上の整数/論理演算とアドレス演算、 GVレジスタとメモリ間のロード/ストア命令、 GVレジスタ上での演算型条件分岐命令、条件コードを用いる条件分岐命令、およびオペレーティング・システムで用いるシステム制御命令 (特権命令)などからなる。

13.1. GVデータ移動命令

GVレジスタから/へのデータ移動、または型変換命令には以下のようなものがある。これらの命令はすべてI形式なので、レジスタの性か32bitのイミディエイトをオペランドに取ることができる。

cpr	レジスタ値のコピー
cprp	レジスタ・ペアのコピー
extrb	レジスタ上の符号付きバイトをワードに拡張
extrh	レジスタ上の符号付き半語をワードに拡張
ldcc	条件コード(CC)をレジスタに読み込む
stcc	レジスタの下位4ビットをCCに書き込む
ldcfp	CFPのさすアドレスをレジスタに読み込む
stcfp	レジスタ内のアドレスをCFPに書き込む

13.2. GV整数/論理演算命令

GVレジスタ上での整数/論理演算には、以下のようなものがある。これらの命令は全てI形式で、3オペランド命令である。また、全語のイミディエイト値をソース・オペランドに取ることもできる。 nopやcmpでは、不要なオペランドが無視される。

nop	何もしない(No OPeration)
add	加算
sub	減算
and	論理積
or	論理和
xor	排他的論理和
cmp	比較(結果を書き込まないsub)

nopは何もしない演算と定義できる。乗算と除算はGV命令には含まれないので、 SP命令を用いて実現する。 cmp命令では、ccだけがセットされる。

13.3. アドレス計算命令

アドレス演算命令には、以下のようなものがある。アドレス演算命令は、 setaを除いてK/KL形式である。 seta命令だけは、演算命令との中間的性格からI形式となっている。

lea	アドレス計算
mkea	アドレス計算
lbl	アドレス・ペアの計算
mkbl	アドレス・ペアの計算
seta	セット・アドレス・タグ
stea	アドレス計算結果をメモリに格納
pea	アドレス計算結果をメモリにプッシュ

leaは、 modeフィールドに応じてオフセット/ポインタ/インデックス・モードのアドレス計算を行なう。 mkeaは、 modeフィールドに応じてオフセット・インデックスやポインタ・インデックス・モードのアドレス計算を行なう。 lbl(load BL)は、 leaを2組行なってBLペアを作成する。 mkbl(make BL)は、 mkeaを2組行なってBLペアを作成する。 lea/mkeaやlbl/mkblが分かれているのは、デコードの都合による。即ちアドレス計算命令において、加算の回数はopcodeで制御するのでlea/lblとmkea/mkeaは別のopcodeになり、加算のオペランドはmodeで制御するのでモード

ごとに別なmodeが使用される。 setaは、レジスタの値とイミディエイト値を足して、その結果(数値)がBLの範囲内の値ならば、演算結果のアドレス・タグを立てる(アドレスにする)。
stea/peaは、アドレス計算の結果をメモリにストア/プッシュする。その意味ではこれらの命令は、次の項「ロード/ストア命令」にも含まれる。 stea/peaでは、オフセット/ポインタ/インデックス・モードだけが許される。

13.4. ロード/ストア命令

GV命令で、メモリとGVレジスタ間でデータを移動する命令は、以下の通り。ただし、特権命令は後の項「システム制御命令」で述べる。ロード/ストア命令は全てK/KL形式である。

ldb	メモリ上のバイトを0拡張してレジスタにロード
ldbm	アドレス計算の副作用付きldb
ldh	メモリ上の半語を0拡張してレジスタにロード
ldhm	アドレス計算の副作用付きldh
ldw	メモリ上のワードをレジスタにロード
ldq	メモリ上の倍語をレジスタ・ペアにロード
extb	メモリ上のバイトを符号拡張してレジスタにロード
extbm	アドレス計算の副作用付きextb
exth	メモリ上の半語を符号拡張してレジスタにロード
exthm	アドレス計算の副作用付きexth
stb	レジスタの下位1バイトをメモリにストア
stbm	アドレス計算の副作用付きstb
sth	レジスタの下位2バイトをメモリにストア
sthm	アドレス計算の副作用付きsth
stw	レジスタ上のワードをメモリにストア
stq	レジスタ・ペアの値をメモリにストア
las	不可分なロードとストア(レジスタ値とメモリ値の交換)
alloc	やや特殊なアドレス計算+メモリ・アクセス

ldb/ldhではレジスタの上位には0が入るのに対して、 extb/exthではレジスタの上位にデータの符号ビットが拡張される。 stb/sthではレジスタの下位1/2バイトがメモリにストアされる。
ldbm/ldhm/extbm/exthm/stbm/sthmはこれらと性格同じだが、アドレシング・モードに副作用が許される。 ldw/ldq/stw/stqは、レジスタ/レジスタ・ペアのロード/ストア命令。 lasはload and storeの略で、メモリ上のワードとレジスタ上のワードを不可分に(即ち他のプロセッサに知られずに)交換する。マルチ・プロセッサでの同期機構を実現するプリミティブ命令である。 allocは、lispやOSでメモリ領域を割り当てるとき使用する、複合命令である。

13.5. ビット命令

FLATS2には、メモリ上/レジスタ上のワードの特定のビットを操作する命令が用意されている。用意されている操作の種類は、セット(1にする)、クリア(0にする)、チェンジ(反転する)、テスト(0かどうか調べる)の4種類である。さらに、メモリ上/レジスタ上のワードに対して、その中に何ビットの1が含まれているか(bit count)、および、その中の1は何ビット目に初めてあるか(bit find)、という処理を行なう命令が用意されている。これらの命令はOSやlispなどで、オブジェクトの管理を表で行なう際に有効である。

以下の命令は、K/KL形式。

bitset	メモリ上のワード内のビットのセット
bitclr	メモリ上のワード内のビットのクリア
bitchg	メモリ上のワード内のビットの反転
bittst	メモリ上のワード内のビットのテスト
bitcnt	メモリ上のワード内の値1のビットの数を数える
bitfnd	メモリ上のワード内で最初の1のビット番号

以下の命令は、I形式。

bitsetr	レジスタ上でのbitset

bitclrr	レジスタ上でのbitclr
bitchgr	レジスタ上でのbitchg
bittstr	レジスタ上でのbittst
bitcntr	レジスタ上でのbitcnt
bitfndr	レジスタ上でのbitfnd

13.6. 分岐命令

FLATS2の分岐命令は、大きく3つに分類できる。以下では、その分類にしたがってFLATS2の分岐命令を説明して行く。命令の名前で、 bはbranch(短分岐)、 jはjump(長分岐)を意味する。短分岐の命令はJ形式で、4方向に各8ビットまでのオフセットで分岐できる。長分岐の命令はJL形式で、3方向に各32ビットのオフセットで分岐でき、さらに次命令を含めると4方向への分岐が可能である。

13.6.1. 無条件分岐命令

無条件でプログラムの実行番地を変更する命令。これらの命令は、全てI形式である。

jump	(無条件)分岐
call	サブプログラムの呼び出し
ret	サブプログラムからの復帰
trap	トラップ(ソフトウェア割り込み)

13.6.2. (非演算型)条件分岐命令

条件コードの状態、またはオペランドの型(アドレス・タグの値)に応じて分岐する命令。bcc/jccは4方向分岐、adrb/adrjは2方向分岐。

bcc	条件コードに応じて短分岐
jcc	条件コードに応じて長分岐
adrb	アドレスか否かで短分岐
adrj	アドレスか否かで長分岐

13.6.3. 演算型条件分岐命令

ソース・オペランドを演算してデスティネーションに書き込み、さらに演算結果をレジスタ(またはレジスタ・ペア)と比較して、その比較結果に応じて2～4方向条件分岐を行なう命令。演算は、加算(acb/acj)/減算(scb/scj)/しない(cb/cj)の3通り。比較には、符号付き/符号無しの両方がある。さらに比較対象は、レジスタ値でもBLで指定する範囲でもよい。これらの命令は、DO文/for文などのループ末端で頻繁に使用される命令である。これらの命令の実装には、 BLアドレシングの仕組みがそのまま使用されている。

cb	比較と短分岐
cj	比較と長分岐
acb	加算と比較と短分岐
acj	加算と比較と長分岐
scb	減算と比較と短分岐
scj	減算と比較と長分岐
eqab	アドレス比較と短分岐
eqaj	アドレス比較と長分岐

13.7. システム制御命令

以下の命令は、すべてプロセッサ状態の管理に使う特権命令なので、特権モードでないと使用できない(使用すると特権例外が発生する)。命令形式は、すべてI形式である。

ldps	PSをレジスタに読み込む
stps	レジスタの値をPSに書き込む
ldkey	DMのkeyメモリをレジスタに読み込む

stkey	レジスタをDMのkeyメモリに書き込む
rmkey	DMのkeyメモリ・エントリを無効にする
ldlru	DMのlruメモリをレジスタに読み込む
stlru	レジスタをDMのlruメモリに書き込む
pld	物理アドレスでDMをレジスタに読み込む
pst	物理アドレスでレジスタをDMに読み込む
ldgvkey	GVMのkeyメモリをレジスタに読み込む
stgvkey	レジスタをGVMのkeyメモリに書き込む
rmgvkey	GVMのkeyメモリ・エントリを無効にする
pstcfp	CFPに値を書き込む
ldim	IM上のワードをレジスタに読み込む
stim	レジスタの値をIMに書き込む
pldim	物理アドレスでIMをレジスタに読み込む
pstim	物理アドレスでレジスタをIMに読み込む
ldimkey	IMのkeyメモリをレジスタに読み込む
stimkey	レジスタをIMのkeyメモリに書き込む
rmimkey	IMのkeyメモリ・エントリを無効にする
ldimlru	IMのlruメモリをレジスタに読み込む
stimlru	レジスタをIMのlruメモリに書き込む
kseta	レジスタ上の値をアドレスにする
ipintr	プロセッサにSLF_INTをかける
ipcnmi	プロセッサにSLF_NMIをかける

これらの命令の詳細については、『プロセッサ状態』や『メモリ管理』の項を参照のこと。

14. SP命令

この章では、SP命令のあらましについて説明する。SP命令は、オペランドにGVレジスタ/SPレジスタ/メモリ/イミディエイト値を取ることができる、汎用型の命令セットである。SP命令は命令形式としてはM形式をとり、opcodeフィールドでオペランドのアドレシング・モードを指定し、modeフィールドでオペランドの種別を指定し、さらにaopフィールドで演算の種類を指定する。

オペランドにはメモリ・オペランドを最大2つまで指定できるが、一般の命令では読みだしオペランドが1つ(または0個)　書き込みオペランドが1つ(または0個)でなければならない。opcodeフィールドに特殊なアドレシング・モード(後述)を指定すれば、読みだしオペランドにメモリを2つ指定することもできるが、これらのアドレシング・モードはそれ専用の数値演算用特殊命令をaopに指定しないと意味を持たない。

以下の節では、まず最初にSP命令で使用するM形式のアドレシング・モードを説明し、その後にFLATS2で実行できる演算の種類(aopの種類)を略述する。数値演算用特殊命令についてもごく簡単に触れるが、詳細については別項『数値演算用特殊命令』で述べることにする。

14.1. SP命令におけるアドレシング・モード

14.1.1. アドレシング・モードの命名規則

SP命令のアドレシング・モードでは、メモリ・オペランドとそのアドレシング・モード(+副作用)を指定する。以下、オペランドの数とアドレシング方法によって、全体を5つのグループに分けて解説を行なう。ここで、アドレシング・モード名はそれぞれ次のような文字からなり、それぞれの文字は次のようなアドレシング・モードを意味する。

r	非メモリ・オペランド
x	インデックス・モード
o	オフセット・モード
p	ポインタ・モード
y	インデックス・モード
z	オフセット・モード
u	副作用postmodify
v	副作用なし
w	副作用premodify

x/o/pでは、メモリ・オペランドが2つあった場合でも、BLには共通のレジスタ・ペア(r3とr3')を用いる。このとき、第1メモリ・オペランドのインデックスやポインタにr1が、第2メモリ・オペランドのインデックスやポインタにr2が使用される。

y/zでは、メモリ・オペランドが2つあった場合、第1メモリ・オペランドのBLにはr1とr1'のペア、第2メモリ・オペランドのBLにはr2とr2'のペアを使用する。第1または第2のいずれか片方のメモリ・オペランドがyモードであった場合、　r3がインデックス・レジスタとして用いられる。第1、第2ともメモリ・オペランドがyモードだった場合は、　r3が第1オペランドのインデックスに、r3'が第2オペランドにインデックスに使用される。

o/p/zモードでのディスプレースメントとしては第1メモリ・オペランドにd1が、第2メモリ・オペランドにd2が使用される。

アドレシングの副作用は、u/v/wの文字で表わされる。uと書いてある場合、計算前のアドレス(x/o/y/zならベース、pならポインタ)でメモリをアクセスして、計算後のアドレスをアドレスに使ったレジスタに書き戻す。　vと書いてある場合、計算されたアドレスでメモリをアクセスし、副作用は残さない。wと書いてある場合、計算後のアドレスでメモリをアクセスして、そのアドレスをアドレスとして使ったレジスタ(x/o/y/zならベース、　pならポインタ)に書き込む。たとえて言えば、uはスタックからのポップに、wはプッシュに用いる。

14.1.2. SP命令で使用できるオペランド(modeフィールド)

アドレシング・モード名においてrで表わされるオペランドは、メモリ・オペランド以外のオペランドである。これらのオペランドについては、それが実際には何であるのかその種類をmodeフィールドで指定する。この項では、SP通常命令(特殊命令以外)で使用するオペランドにつ

いて、それぞれどのようなmodeフィールドを使用するのか説明する。併せて、modeの命名規則を述べる。

以下に、通常命令で使用する32のmodeを表にして掲げる。それぞれのモード名は3文字からなり、1文字目が第1ソース・オペランド、2文字目が第2ソース・オペランド、3文字目がデスティネーション・オペランドを示している。ソース・オペランドのsはSレジスタ、pはPレジスタ、gはGVレジスタ、iはイミディエイト値を表わしている。デスティネーション・オペランドのfは SPレジスタ(SかPかはaopの種類によって異なる)、vはGVレジスタを表わす。

ssf	gsf	ssv	gsv
spf	gpf	spv	gpv
sgf	ggf	sgv	ggv
sif	gif	siv	giv
psf	isf	psv	isv
ppf	ipf	ppv	ipv
pgf	igf	pgv	igv
pif	iif	piv	iiv

modeフィールドは6ビットなので、上記32のモード以外にも、さらに32のモードが指定可能である。これらのモードは、数値演算用特殊命令で使用する専用モードなどに割り当てられているが、それについてはここでは触れない。

アドレシング・モードでメモリ・オペランドを一つも指定しない場合、すべてのオペランドが上記のモードに従って選択される。メモリ・オペランドがある場合、アドレシング・モードでメモリが指定されたオペランドでは、 modeフィールドでの指定は無視されてメモリ・オペランドが使用される。それ以外のオペランド(アドレシング・モード名でrを指定されたオペランド)については、modeフィールドの指定通りにオペランドが選択される。

14.1.3. 0メモリ・オペランド

オペランドにメモリ・オペランドがない場合、アドレシング・モードにはrrrを使う。

名前

rrr

この場合、それぞれのrが示すオペランドが何であるかは、modeフィールドによって指定される。

14.1.4. 1メモリ・オペランド(x/o/p ; R)

メモリ・オペランドが読みだしオペランド1つだけである場合、使用できるモードは{xr|or|pr|rx|ro|rp}r{u|v}の 12通りである。以下に、それぞれのアドレシング・モードで使用するBLや、計算されるアドレス、レジスタに残る副作用などをまとめる。アドレスの前には、そのアドレスの範囲検査を行なうBLレジスタを" : "で区切って書くことにする。また、アドレシングに副作用がない場合は、副作用の項を-と表わす。

名前	アドレス(R)	副作用
xrru	r3:r3	r3+r1
xrrv	r3:r3+r1	-
orru	r3:r3	r3+d1
orrv	r3:r3+d1	-
prru	r3:r1	r1+d1
prrv	r3:r1+d1	-
rxru	r3:r3	r3+r1
rxrv	r3:r3+r1	-
roru	r3:r3	r3+d1
rorv	r3:r3+d1	-
rpru	r3:r1	r1+d1
rprv	r3:r1+d1	-

14.1.5. 1メモリ・オペランド(x/o/p ; W)

メモリ・オペランドが書き込みオペランド1つだけである場合、使用できるモードはrr{x|o|p}{w|v}の6通りである。以下に、それぞれのアドレシング・モードで使用するBLや、計算されるアドレス、レジスタに残る副作用などをまとめる。

名前	アドレス(W)	副作用
rrxw	r3:r3+r2	r3+r2
rrxv	r3:r3+r2	-
rrow	r3:r3+d2	r3+d2
rrov	r3:r3+d2	-
rrpw	r3:r2+d2	r2+d2
rrpv	r3:r2+d2	-

14.1.6. 2メモリ・オペランド(x/o/p ; RW)

メモリ・オペランドが読みだしと書き込みオペランド各1つづつである場合、BLペアを1組しか必要としないならば、使用できるモードは {rx|ro|rp|xr|or|pr}{x|o|r}{u|v|w} の54通りである。さらに、ReadとWriteで同じアドレスを使用する場合、アドレス計算を一つ減らすことができる。これらはRMW(Read Modify Write)のアドレシング・モードと呼ばれ、 {p|x}rm{u|v|w}の6種類用意されている。ここで、mはmodifyの略で、readと同じアドレスを表わすとする。

以下に、それぞれのアドレシング・モードで使用するBLや、計算されるアドレス、レジスタに残る副作用などをまとめる。

名前	アドレス(R)	アドレス(W)	副作用
rxxu	r3:r3	r3:r3+r2	r3+r1
rxxv	r3:r3+r1	r3:r3+r2	-
rxxw	r3:r3+r1	r3:r3+r2	r3+r2
rxou	r3:r3	r3:r3+d2	r3+r1
rxov	r3:r3+r1	r3:r3+d2	-
rxow	r3:r3+r1	r3:r3+d2	r3+d2
rxpu	r3:r3	r3:r2+d2	r3+r1
rxpv	r3:r3+r1	r3:r2+d2	-
rxpw	r3:r3+r1	r3:r2+d2	r2+d2
roxu	r3:r3	r3+r2	r3:r3+d1
roxv	r3:r3+d1	r3:r3+r2	-
roxw	r3:r3+d1	r3:r3+r2	r3+r2
roou	r3:r3	r3:r3+d2	r3+d1
roov	r3:r3+d1	r3:r3+d2	-
roow	r3:r3+d1	r3:r3+d2	r3+d2
ropu	r3:r3	r3:r2+d2	r3+d1
ropv	r3:r3+d1	r3:r2+d2	-
ropw	r3:r3+d1	r3:r2+d2	r2+d2
rpxu	r3:r1	r3:r3+r2	r1+d1
rpxv	r3:r1+d1	r3:r3+r2	-
rpxw	r3:r1+d1	r3:r3+r2	r3+r2
rpou	r3:r1	r3:r3+d2	r1+d1
rpov	r3:r1+d1	r3:r3+d2	-
rpow	r3:r1+d1	r3:r3+d2	r3+d2
rppu	r3:r1	r3:r2+d2	r1+d1
rppv	r3:r1+d1	r3:r2+d2	-
rppw	r3:r1+d1	r3:r2+d2	r2+d2
xrxu	r3:r3	r3:r3+r2	r3+r1
xrxv	r3:r3+r1	r3:r3+r2	-
xrxw	r3:r3+r1	r3:r3+r2	r3+r2

xrou	r3:r3	r3:r3+d2	r3+r1
xrov	r3:r3+r1	r3:r3+d2	-
xrow	r3:r3+r1	r3:r3+d2	r3+d2
xrpu	r3:r3	r3:r2+d2	r3+r1
xrpv	r3:r3+r1	r3:r2+d2	-
xrpw	r3:r3+r1	r3:r2+d2	r2+d2
orxu	r3:r3	r3:r3+r2	r3+d1
orxv	r3:r3+d1	r3:r3+r2	-
orxw	r3:r3+d1	r3:r3+r2	r3+r2
orou	r3:r3	r3:r3+d2	r3+d1
orov	r3:r3+d1	r3:r3+d2	-
orow	r3:r3+d1	r3:r3+d2	r3+d2
orpu	r3:r3	r3:r2+d2	r3+d1
orpv	r3:r3+d1	r3:r2+d2	-
orpw	r3:r3+d1	r3:r2+d2	r2+d2
prxu	r3:r1	r3:r3+r2	r1+d1
prxv	r3:r1+d1	r3:r3+r2	-
prxw	r3:r1+d1	r3:r3+r2	r3+r2
prou	r3:r1	r3:r3+d2	r1+d1
prov	r3:r1+d1	r3:r3+d2	-
prow	r3:r1+d1	r3:r3+d2	r3+d2
prpu	r3:r1	r3:r2+d2	r1+d1
prpv	r3:r1+d1	r3:r2+d2	-
prpw	r3:r1+d1	r3:r2+d2	r2+d2
prmu	r3:r2	r3:r2	r2+d2
prmv	r3:r2+d2	r3:r2+d2	-
prmw	r3:r2+d2	r3:r2+d2	r2+d2
xrmu	r3:r3	r3:r3	r3+r2
xrmv	r3:r3+r2	r3:r3+r2	-
xrmw	r3:r3+r2	r3:r3+r2	r3+r2

14.1.7. 2メモリ・オペランド(y/z ; RW)

メモリ・オペランドが読みだしと書き込みオペランド各1つづつである場合、BLペアが2組必要ならば、使用できるモードは {ry|rz|yr|zr}{y|z}{u|v}{u|v}の32通りである。以下に、それぞれのアドレシング・モードで使用するBLや、計算されるアドレス、レジスタに残る副作用などをまとめる。

名前	アドレス(R)	アドレス(W)	副作用(R)	副作用(W)
ryyvv	r1:r1+r3	r2:r2+r3'	-	-
ryyvu	r1:r1+r3	r2:r2	-	r2+r3'
ryyuv	r1:r1	r2:r2+r3'	r1+r3	-
ryyuu	r1:r1	r2:r2	r1+r3	r2+r3'
ryzvv	r1:r1+r3	r2:r2+d2	-	-
ryzvu	r1:r1+r3	r2:r2	-	r2+d2
ryzuv	r1:r1	r2:r2+d2	r1+r3	-
ryzuu	r1:r1	r2:r2	r1+r3	r2+d2
rzyvv	r1:r1+d1	r2:r2+r3	-	-
rzyvu	r1:r1+d1	r2:r2	-	r2+r3
rzyuv	r1:r1	r2:r2+r3	r1+d1	-
rzyuu	r1:r1	r2:r2	r1+d1	r2+r3
rzzvv	r1:r1+d1	r2:r2+d2	-	-
rzzvu	r1:r1+d1	r2:r2	-	r2+d2
rzzuv	r1:r1	r2:r2+d2	r1+d1	-
rzzuu	r1:r1	r2:r2	r1+d1	r2+d2

yryvv	r1:r1+r3	r2:r2+r3'	-	-
yryvu	r1:r1+r3	r2:r2	-	r2+r3'
yryuv	r1:r1	r2:r2+r3'	r1+r3	-
yryuu	r1:r1	r2:r2	r1+r3	r2+r3'
yrzvv	r1:r1+r3	r2:r2+d2	-	-
yrzvu	r1:r1+r3	r2:r2	-	r2+d2
yrzuv	r1:r1	r2:r2+d2	r1+r3	-
yrzuu	r1:r1	r2:r2	r1+r3	r2+d2
zryvv	r1:r1+d1	r2:r2+r3	-	-
zryvu	r1:r1+d1	r2:r2	-	r2+r3
zryuv	r1:r1	r2:r2+r3	r1+d1	-
zryuu	r1:r1	r2:r2	r1+d1	r2+r3
zrzvv	r1:r1+d1	r2:r2+d2	-	-
zrzvu	r1:r1+d1	r2:r2	-	r2+d2
zrzuv	r1:r1	r2:r2+d2	r1+d1	-
zrzuu	r1:r1	r2:r2	r1+d1	r2+d2

14.1.8. 2メモリ・オペランド(y/z ; RR)

メモリ・オペランドに読みだしオペランドが2つ必要である場合、 FLATS2では、BLペアは常に2組指定しなければならない。使用できるモードは{y|z}{y|z}r{u|v}{u|v}の16通りである。以下に、それぞれのアドレシング・モードで使用するBLや、計算されるアドレス、レジスタに残る副作用などをまとめる。

名前	アドレス(1)	アドレス(2)	副作用(1)	副作用(2)
yyrvv	r1:r1+r3	r2:r2+r3'	-	-
yyrvu	r1:r1+r3	r2:r2	-	r2+r3'
yyruv	r1:r1	r2:r2+r3'	r1+r3	-
yyruu	r1:r1	r2:r2	r1+r3	r2+r3'
yzrvv	r1:r1+r3	r2:r2+d2	-	-
yzrvu	r1:r1+r3	r2:r2	-	r2+d2
yzruv	r1:r1	r2:r2+d2	r1+r3	-
yzruu	r1:r1	r2:r2	r1+r3	r2+d2
zyrvv	r1:r1+d1	r2:r2+r3	-	-
zyrvu	r1:r1+d1	r2:r2	-	r2+r3
zyruv	r1:r1	r2:r2+r3	r1+d1	-
zyruu	r1:r1	r2:r2	r1+d1	r2+r3
zzrvv	r1:r1+d1	r2:r2+d2	-	-
zzrvu	r1:r1+d1	r2:r2	-	r2+d2
zzruv	r1:r1	r2:r2+d2	r1+d1	-
zzruu	r1:r1	r2:r2	r1+d1	r2+d2

14.2. SP命令で使用する演算の種類(aop)

いままで述べたアドレシング・モード(opcode)やオペランド・モード(mode)で指定された演算オペランドは、 aopフィールドで指定された演算の種類にしたがって処理される。以下では、演算の種類をいくつかに大別して、FLATS2で提供されている演算の種類を簡単に説明して行く。

14.2.1. データ移動命令

FLATS2では、SPレジスタに浮動小数点数の内部表現があるために、整数のmovと浮動小数点数のmovをaopフィールドで区別する必要がある。SPレジスタ内部のタグを読み書きするには、ldtag/sttagを使用する。 SPでの内部表現の詳細については、『データ表現』を参照のこと。ハードウェア実装上の都合で、 Pレジスタへのmovは他と区別しなければならないため、その場合だけは別のaop(mop)を使うようになっている。

s_nop	何もしない(No OPeration)
s_movw	全語の移動
s_movq	倍語の移動
s_movf	単精度浮動小数点数の移動
s_movd	倍精度浮動小数点数の移動
s_movff	単精度複素数の移動
s_sttag	Sレジスタ内部のタグを読みだす
s_ldtag	Pレジスタ内部のタグへ書き込む
s_mopw	Pレジスタへの全語の移動
s_mopq	Pレジスタへの倍語の移動
s_mopf	Pレジスタへの単精度浮動小数点数の移動
s_mopd	Pレジスタへの倍精度浮動小数点数の移動
s_mopff	Pレジスタへの単精度複素数の移動

14.2.2. データ型変換命令

データの型変換を行なう命令には、以下のようなものがある。

s_ext	全語から倍語への符号拡張を行なう
s_floatq	倍語を倍精度浮動小数点数に変換する
s_ifixd	倍精度浮動小数点数を倍語に変換する

14.2.3. 整数/論理演算命令

FLATS2のSP命令で提供する整数/論理演算には、以下のようなものがある。ここで、命令名のwは全語(word)、qは倍語(quad byte)、uは符号なし(unsiged)、sは符号つき(signed)を意味する。

s_addw	全語の加算
s_addq	倍語の加算
s_subw	全語の減算
s_subq	倍語の減算
s_muluw	符号なし全語の乗算
s_mulsw	符号つき全語の乗算
s_muluq	符号なし倍語の乗算
s_mulsq	符号つき倍語の乗算
s_cmpw	全語の比較
s_cmpq	倍語の比較
s_absw	全語の絶対値
s_absq	倍語の絶対値
s_andw	全語の論理積
s_andq	倍語の論理積
s_orw	全語の論理和
s_orq	倍語の論理和
s_xorw	全語の排他的論理和
s_xorq	倍語の排他的論理和
s_notw	全語の論理否定
s_notq	倍語の論理否定
s_negw	全語の符号反転
s_negq	倍語の符号反転

14.2.4. シフト命令

FLATS2では、GV命令にシフト命令はないため、シフト操作では常にM形式のシフト命令を使う。ローテイト命令はFLATS2には用意されていないので、必要ならばシフト命令と論理和命令でソフトウェア的に実現する。 bitrev命令は、シフトとは少し違うが、全語内でのビット順序を反転する(上下逆に入れ換える)。この機能は、bitfndを用いるとき必要に応じて使用する。

bitfndのサーチ方向はハードウェア的に定まっているので、逆側からサーチしたいときにはあらかじめこの命令でオペランドのビット順序を上下逆にしておく。

s_lsrw	全語の論理右シフト
s_lsrq	倍語の論理右シフト
s_asrw	全語の算術右シフト
s_asrq	倍語の算術右シフト
s_lslw	全語の論理左シフト
s_lslq	倍語の論理左シフト
s_bitrev	全語のビット順序反転

14.2.5. 浮動小数点演算命令

FLATS2の提供する浮動小数点演算命令は、以下の通りである。

s_addf	単精度加算
s_addd	倍精度加算
s_addf	単精度複素加算
s_subf	単精度減算
s_subd	倍精度減算
s_subff	単精度複素減算
s_mulf	単精度乗算
s_muld	倍精度乗算
s_mulff	単精度複素数の単精度倍
s_cmpf	単精度比較
s_cmpd	倍精度比較
s_absf	単精度絶対値
s_absd	倍精度絶対値
s_negf	単精度符号反転
s_negd	倍精度符号反転

14.2.6. 数値演算用特殊命令

FLATS2の提供する数値演算用特殊命令は以下の通り。これらの命令の使用法については、別項『数値演算用特殊命令』を参照のこと。 ripf/ripd/ripff/cipf/cipd1/cipd2/cfft1/cfft2の各命令については、 RR (read-read)モードのアドレシング・モードと共に使用しなければならない。

s_divsf	単精度除算(初期化)
s_divlf	単精度除算(ステップ)
s_divsd	倍精度除算(初期化)
s_divld	倍精度除算(ステップ)
s_mulc1	単精度複素乗算(1)
s_mulc2	単精度複素乗算(2)
s_cipf	単精度複素内積ステップ
s_cipd1	倍精度複素内積ステップ(1)
s_cipd2	倍精度複素内積ステップ(2)
s_cfft1	単精度複素FFTステップ(1)
s_cfft2	単精度複素FFTステップ(2)
s_ripf	単精度実数内積ステップ
s_ripd	倍精度実数内積ステップ
s_ripff	単精度実数倍ステップ
s_addbig1	多倍精度整数加算ステップ(1)
s_addbig2	多倍精度整数加算ステップ(2)
s_subbig1	多倍精度整数減算ステップ(1)
s_subbig2	多倍精度整数減算ステップ(2)
s_mulbig1	多倍精度整数乗算ステップ(1)
s_mulbig2	多倍精度整数乗算ステップ(2)

s_honorf　単精度ホーナーループ・ステップ
s_honord　倍精度ホーナーループ・ステップ

14.2.7. 入出力命令

FLATS2では、SVPとFLATS2間の通信のために、IODEVという共有メモリを持っている。この共有メモリを読み書きするために、以下のような命令が提供されている。これらの命令は特権命令である。命令の具体的な仕様については、命令リファレンスを参照のこと。

s_putidv　共有メモリに書き込む
s_getidv　共有メモリを読み出す

数値演算用特殊命令

使用頻度の高い数値演算操作を効率よく実行するために、FLATS2には数値演算用の専用命令(群)が用意されている。これらの命令を組み合わせて使用することにより、複素数演算、行列演算等において高いスループットを得ることができる。以下、本章では数値演算用特殊命令について概説してゆく。

特殊命令においては、オペランドは通常デフォルトで定まっている。基本的には各特殊命令にそれぞれ対応するモードがあり、それ以外のモードを指定した場合の結果は保証しない(未定義である)。命令とそのモードについては、「特殊命令の詳細」の項を参照のこと。

15. 特殊命令で用いるレジスタ

特殊命令では、通常の命令で使用するレジスタ(S, P)に加え、いくつかのスクラッチ・パッド・レジスタを使用する。これのレジスタについて、簡単にまとめておく。

15.1. Sレジスタ

Sパイプの出力を格納する。通常命令で用いるSレジスタ。通常命令を用いて読み/書き/操作することができる。SP内部形式でデータを保持するため、96ビット長。

15.2. Tレジスタ

Sパイプの出力を格納するスクラッチ・レジスタ。特殊命令のデータ移動命令を使用してアクセスする。SP内部形式でデータを保持するため、96ビット長。

15.3. Pレジスタ

Pパイプの出力を格納する。通常の命令で用いるPレジスタ。通常命令を用いて読み/書き/操作することができる。SP内部形式でデータを保持するため、96ビット長。

15.4. Qレジスタ

Pパイプの出力を格納するスクラッチ・レジスタである。特殊命令のデータ移動命令を使用してアクセスする。SP内部形式でデータを保持するため、96ビット長。

15.5. Rレジスタ

メモリからのデータを格納するスクラッチ・レジスタ。特殊命令でしかアクセスできない。メモリ形式でデータを保持するため、66ビット長である。

15.6. Uレジスタ

Rレジスタと同様、入力データを一時的に格納するスクラッチ・レジスタ。特殊命令でしかアクセスできない。メモリ形式でデータを保持するため、66ビット長である。

Sレジスタ、Pレジスタは、データのロード/ストア、通常命令のソース・オペランド/デスティネーション・オペランド、特殊命令のオペランド(ソース/デスティネーション)のいずれにも使用可能である。 TおよびQレジスタは、データのロード/ストア、特殊命令のオペランド(ソース/デスティネーション)としてのみ、使用可能である。 RおよびUレジスタは更に制限が強く、特殊命令によるデータのロード/ ストアおよび、以後の特殊命令の暗黙のオペランドとしてしか用いられることはない。

特殊命令を実行したとき、どのレジスタが変更/保存されるかは、命令/モードによってそれぞれ事情が異なる。特殊命令の使用に際してはドキュメントを参照し、充分注意して行なう必要がある。

16. 特殊命令の概説

16.1. データ移動命令

以下のデータ移動命令は、通常命令で使用する命令と全く同じものである(オプコード自体が同じ)。ただし、特殊命令で使用するレジスタにアクセスするため、モードに特殊な値を使用する(後述)。

movw	シングル・ワード整数の移動
movq	ダブル・ワード整数の移動
movf	単精度浮動小数点数の移動
movd	倍精度浮動小数点数の移動
movff	単精度複素数の移動

正確には、これらのmov系命令はそれぞれ2つのオプコードを持ち、一方(mov)はSパイプ/Sレジスタを、もう一方(mop)はPパイプ/Pレジスタを使用して移動操作を行なう。アセンブラが自動的にどちらかのオプコードを選択するので、プログラマはオプコードの相違を気にする必要はない。

これらの命令に対して、使用可能な特殊命令用オペランド(モード)は以下の通りである。

stg	メモリまたはレジスタの、TまたはQへの書き込み
tsf	メモリまたはレジスタへの、Tの読み出し
qsg	メモリまたはレジスタへの、Qの読み出し
isg	イミディエイト値の、TまたはQへの書き込み
rdr	Rの読み出し
rdu	Uの読み出し
wtr	メモリからRへの書き込み
wtu	メモリからUへの書き込み

モードにstgまたはisgを使用した場合、オプコードがmovならばTへの書き込み、 mopならばQへの書き込みが発生する。

wtr/wtuは、ソース・オペランドがメモリのとき、ソース・アドレスのメモリ内容をR/Uレジスタへ書き込む。また、rdr/rduは、ソースがレジスタの時、 R/Uから値を読みだして、データを命令のソース・オペランドとして送りだす。

16.2. 除算命令

倍精度(64bit)の浮動小数点除算を実行するための命令セット。除算開始命令(divs)と除算ステップ命令(divl)がある。

divs, divlとも、使用するモードはdivである。 divsは、被除数(Q)と除数(P)を処理して、divl命令のオペランドを準備する。 divlは、PとQを演算して除算結果の近似を行なう。単精度浮動小数点ならば5回、倍精度ならば6回で収束して、除算結果がQレジスタに残る。 divs, divlともに、オペランドはPとQである。 PもQも、命令実行にともなって変化する(破壊される)。

16.3. 複素数基本命令

単精度複素数(32bit×2 即ち COMPLEX*4)の加算(**addff**)、減算(**subff**)、実数倍(**mulff**)、乗算(**mulc1, mulc2**)が用意されている。倍精度複素数(64bit×2 即ち COMPLEX*8)の基本演算は通常命令の組合せで実行するため、特別な命令はない。

実はaddff, subff, mulffは通常命令なので、通常命令で使用可能なモードは全て使用可能である。 mulffでは、第1オペランドは単精度複素数、第2オペランドは倍精度浮動小数点数でなければならない。デスティネーションは、単精度複素数になる。 mulc1の場合、暗黙のソース・オペランドおよびデスティネーションはSとPであるが、ソースのどちらか一方をメモリ・オペランドで置き換えることができる。 mulc2は、mulc1が残した複素数の部分積(PとQ)を使用して、最終的な複素数の積を計算する。ソース・オペランドはPとQ、暗黙のデスティネーションはSであるが、デスティネーションにはメモリを指定することもできる。 mulc1とmulc2のモードには、それぞれ専用のモードを指定する。

16.4. 内積命令

内積命令は、オペランドのタイプによって以下の4種に分けられる。

単精度浮動小数点数	ripf, ripff
倍精度浮動小数点数	ripd
単精度複素数	cipf
倍精度複素数	cipd1, cipd2

ripf, ripdは、2つの実数オペランドの積をとって部分積を計算し、一方で前回の部分積を前回までの部分和に加える。同様に、cipfは2つの複素数オペランドをとり、部分積の計算と部分和の計算を並行して行なう。 cipd1, cipd2はcipfの倍精度版であるが、この2命令で2つの倍精度複素数を扱う。 ripffは、ripfと似ているが、オペランドとして2つの実数オペランドを2組とる。すなわち、ripfを2組並列に実行する。

内積命令はループ最内周で用いるための命令であり、通常の意味での内積プリミティブ(Multiply and Add)とは異なる。「Multiply and Add」は2つのオペランドの積を部分和に加えるが、内積命令においては2つのオペランドの積が部分積デスティネーション(P, Q)に書き込まれ、同時に命令開始時の部分積ソース(P, Q)が命令開始時の部分和ソース(S, T)と加算されて部分和デスティネーション(S, T)に書き込まれる。ループ内で内積命令を繰り返し実行することにより、最終的な内積を得ることができる。

16.5. FFT命令

FFT命令(**cfft1**, **cfft2**)は、単精度複素数配列上でFFT演算を行なうための特殊命令である。この2つの命令でFFTの最内周ループを構成する。使用にあたっては、あらかじめUレジスタに定数ωをロードしておき、各命令で配列要素を1つ読んではωと演算し、その一方で前回の演算結果を配列に書き戻す。

cfft1のソース・オペランドは、メモリ, R, U, T, P, Qレジスタ。デスティネーションは、メモリ, R, T, P, Qレジスタ。cfft2のソースは、メモリ, P, Q, Tレジスタ。デスティネーションは、メモリ, T, Pレジスタ。単一用途のために、極めて特化された命令であるといえる。

16.6. 多倍長整数命令

FLATS2では、32ビット整数の配列によって多倍長整数を表現し、それらに対する加算/減算/乗算用プリミティブ命令を提供している。除算については特別な命令を持たないので、加減乗算を組み合わせてエミュレートする。

加算と減算は、演算が異なるだけで手順は全く同様である。基本的にはループを用いて、多倍長数の各ワードについて2つの演算プリミティブ命令を実行する。最内周ループでは、オペランド双方の対応するニブル(ワード)と下位の加減算結果からの桁上がり(借り)を加算/減算する。以下に、各命令の動作を概観する。

第1ステップ(**addbig1**, **subbig1**)では、二つの長整数の対応する2つのニブルを加えて、部分和を生成する。得られた部分和(64ビット)はTレジスタに格納され、同時に部分和の下位32ビットがデスティネーション(対応する部分和のニブル)にストアされる。このとき、加算と並行して、次のサイクルで使用する第1オペランドをSレジスタにロードしておく。

第2ステップ(**addbig2**, **subbig2**)では、次のサイクルで使用する第2オペランドをSレジスタにロードする。またこれと並行して、第1ステップでロードした第1オペランドに、第1ステップの結果からの桁上がりを加えておく。即ち、Tレジスタの上位32ビットとSレジスタを加えてTレジスタに置く。

乗算も2つの基本プリミティブ命令を持ち、最内周ループでこれらの命令を繰り返し実行する。最内周ループでは、多倍長整数オペランドとTレジスタの乗算を行ない、乗算結果を積の部分和に加算する。 Tレジスタには、無符号単精度整数(すなわち多倍長数の1ニブル)を入れておく。外周のループで、この最内周ループを多倍長数の各ニブルについて実行することにより、多倍長整数同志の乗算結果が得られる。以下に、各命令の動作を概観する。

第1ステップ(**mulbig1**)では、メモリ・データとTレジスタを掛けてPレジスタに格納する。並行して、SレジスタとPレジスタを加えて結果をSレジスタに書き込み、同時にその演算結果の下位32ビットをメモリ(部分和)に書き戻す。第2ステップ(**mulbig2**)では、Sレジスタの上位32ビットと Pレジスタを加えて結果をSレジスタに書き込む。同時に、メモリから部分和を読んでPレジスタにロードする。

16.7. 多項式計算命令

いわゆるHonorループを実現する命令。メモリ上に係数(浮動小数点数)の配列があるとき、Sレジスタにパラメータxを入れてこの命令を繰り返し用いることにより、 xのn次の多項式の値を計算する。係数配列の精度に応じて、倍精度honor命令(honord)と単精度honor命令(honorf)の二種類が用意されている。

具体的には、

(1) SレジスタとPレジスタの値を掛けてPレジスタに格納し、

(2) Tレジスタの内容にQレジスタの内容を加え、

(3) メモリからのデータとPレジスタを掛けてQレジスタに格納する。

という3つの操作を並行して行なう。Pレジスタには係数に対応した xのべキ乗数、Qレジスタには係数と対応するべキ乗数の積、Tレジスタには多項式の部分和が残る。

17. 特殊命令の使用法

17.1. 浮動小数点数の除算

浮動小数点数の除算は、

(1) ソース・オペランドのセット・アップ、
(2) 被除数×逆数の近似値、
(3) デスティネーションへのストア、

の3ステップで実行できる。これをFLATS2の命令を用いて具体的に書くと、以下のようになる。ここではモード等については省略し、アルゴリズムの骨組みだけを簡単にまとめる。

倍精度浮動小数点数の場合

```
movd    除数, P         ; 除数のロード
movd    被除数, Q       ; 被除数のロード
divs                    ; 被除数×逆数の第1近似値
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
movd    Q, 商           ; 除算結果のストア
```

単精度浮動小数点数の場合

```
movf    除数, P         ; 除数のロード
movf    被除数, Q       ; 被除数のロード
divs                    ; 被除数×逆数の第1近似値
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
divl                    ; 除算ステップ
movf    Q, 商           ; 除算結果のストア
```

もともと除算命令は倍精度浮動小数点数用に設計されているが、単精度数でも同様の手法で計算できる。単精度オペランドはロード時に倍精度数に拡張され、除算ステップ命令でも倍精度数として処理される。結果が単精度で良い場合、除算ステップ命令による近似が倍精度より1段少なくて済む。除算結果をストアする時も、単精度数として扱えば自動的に丸めが行なわれる。

17.2. 整数の除算

FLATS2には、整数の除算命令はない。従って、なんらかのソフトウェアで商と剰余を計算しなければならない。整数除算における商と剰余の定義は、プロセッサ/言語によって異なっており、いろいろな定義が考えられるが、ここでは標準的と考えられる定義を採用する。

定義

```
i div j   ::=   Sign(i)× Sign(j)× Floor(|i/j|)
i mod j   ::=   i − j × (i div j)
```

i *mod* j の符号は、iの符号と一致する。この定義は、Pascal、Ada、VAX-11のdiv命令、68020のdivs命令、IBM S/370のD命令、unix上のf77、unix上のFlanzLispと一致している。

実現

```
S = float.d(j);     除数を浮動小数点数にする。
T = float.d(i);     被除数を浮動小数点数にする。
Q = div.d(T, S);    浮動小数点数の除算を行なう。
T = ifix.q(Q);      Tに商を入れる。
P = imuls(T, j);
S = isub(i, P);     Sに剰余を入れる。
```

このままだと、桁落ちによる誤差が残る可能性があるので、なんらかの形で修正する。以下に一例を示す。

```
while (S <= j < 0 .or. S >= j > 0) {
    if (T < 0) {
        T = T - 1;
        S = S + j;
    } else {
        T = T + 1;
        S = S - j;
    }
}
```

符号を別に計算し、絶対値で商と剰余を近似することも考えられる。

このように整数の除算は複雑な計算であるので、マクロ展開よりサブルーチン化ないしトラップルーチン化が望ましいと考えられる。

17.3. 複素数の加減乗算

倍精度複素数については、すべてライブラリで処理する。そこで以下この節では、単精度複素数の専用命令の使い方を述べる。以下、「オペランド.r」はオペランドの実部(下位ワード)、「オペランド.i」はオペランド虚部(上位ワード)を表わすこととする。

複素数の加減算

```
addff   (ソース1), (ソース2), (デスティネーション)
subff   (ソース1), (ソース2), (デスティネーション)
mulff   (ソース1), (ソース2), (デスティネーション)
```

複素数の乗算

```
movff (ソース1), P          ; P.r = x.r
                            ; P.i = x.i
mulc1 (ソース2), P          ; P.r = y.r * P.r
                            ; P.i = y.i * P.r
                            ; R.r = y.r
                            ; R.i = y.i
                            ; Q.r = R.r * P.i
                            ; Q.i = R.i * P.i
mulc2 (デスティネーション)  ; S.r = P.r - Q.i
                            ; S.i = P.i + Q.r
```

17.4. 実数配列の内積計算

ripf, ripff, ripd命令は1命令で実数内積計算の最内周ループを構成する。ソース1とソース2にはMMR形のアドレシング・モードを使用し、デスティネーションは暗黙の内にS, P, U, R (ripd, ripf)またはS, T, P, Q, R, U (ripff)である。

以下に、それぞれの動作を略述する。ここで、X, Yはアドレシング・モードで指定されたメモリ・データとする。

ripd

使い方

```
loop:   ripd  X, Y;  goto loop;
```

動作

```
S = P + S;
P = R * U;
U = X;
R = Y;
```

ripf

使い方

```
loop:   ripf  X, Y;  goto loop;
```

動作

```
S = P + S;
P = R * U;
U = X;
R = Y;
```

ripff

使い方

```
loop:   ripff  X, Y;  goto loop;
```

動作

```
S = S + P;
T = T + Q;
P = R.r * U.r;
R.r = X.r;
R.i = X.i;
U.r = Y.r;
U.i = Y.i;
Q = R.i * U.i;
```

17.5. 複素数配列の内積計算

cipfは、1命令で単精度複素数上の内積計算用最内周ループを構成する。同様に、cipd1とcipd2は、2命令で倍精度複素数上の内積計算用最内周ループを実現する。

以下に、各命令の概要を説明する。X, Y は、メモリ上の単精度複素数、.rはその実部、.iはその虚部を表わす。 X_R, X_I, Y_R, Y_I は、メモリ上の倍精度浮動小数点数で、それぞれ倍精度複素数x, yの実部、虚部を表わすとする。

cipf

使い方

```
loop:   cipf  X, Y; goto loop;
```

動作

```
P.r = R.r * U.r;
P.i = R.i * U.r;
S.r = S.r + T.r;
S.i = S.i + T.i;
R.r = X.r;
R.i = X.i;
Q.r = R.r * Y.i;
Q.i = R.i * Y.i;
T.r = P.r - Q.i;
T.i = P.i + Q.r;
U.r = Y.r;
U.i = Y.i;
```

ここでも、PとQに入っている部分積は異なったX, Yのものであることに注意されたい。従って、Tに入っているのは決して2つの複素数の積ではない。それでも、内積計算の場合は(結局ぜんぶ足してしまうので)問題はない。

cipd1, cipd2

使い方

```
loop:   cipd1  X_R, Y_R;
        cipd2  Y_I, X_I;  goto loop;
```

動作

```
cipd1:  P = R * U;
        R = X_R;
        Q = R * Y_R;
        U = Y_R;
        S = S + P;
        T = T - Q;
        goto cipd2;

cipd2:  P = X_I * U;
        U = X_I;
        Q = U * Y_I;
        U = Y_I;
        S = S + P;
        T = T + Q;
        goto cipd1;
```

ここでは、一周目で読み込んだデータが、二周目のcipd1までで部分積になり、二周目のcipd2までに部分和に足し込まれる。このように演算経路が長い場合、ループのセット・アップにそれだけ多くの命令/実行サイクルが必要であることは言うまでもない。

17.6. 複素数配列上でのFFT

cfft1, cfft2は、単精度複素数上のFFT計算ルーチンにおいて、最内周ループを構成する。

まず最初に、ここで使用する記法をまとめておく。Zはメモリ上の単精度複素数配列とし、i, jはアドレス計算時に用いるインデックス、nは最内周ループでのインデックスの増分とする。ループ内では、Rレジスタに定数(ω)を保持しておく。ここでも、xは単精度複素数の実部、iは虚部を表わすこととする。

以下に、各命令の概要を説明する。cfft1は、メモリから複素数を1つ読み込んで、Rレジスタとの積をとる。その一方で、PとTに入っている複素数の和をTに、差をデスティネーション(メモリ上)に書き込む。cfft2は、PとQに入っている部分積から複素数の積を生成し、結果をTにストアする。さらに、メモリから複素数を1つ読んでPにロードし、Tの中身をデスティネーション(メモリ上)に書き込む。この、最内周ループの様子を、以下に示す。

cfft1, cfft2

使い方

```
loop:   cfft1  Z[j], Z[j-n] ; j = j + n;
        cfft2  Z[i], Z[i-n] ; i = i + n; goto loop;
```

動作

```
cfft1:  Z[j-n].r = P.r - T.r;
        Z[j-n].i = P.i - T.i;
        T.r = P.r + T.r;
        T.i = P.i + T.i;
        P.r = Z[j].r * R.r;
        P.i = Z[j].i * R.r;
        U.r = Z[j].r;
        U.i = Z[j].i;
        Q.r = U.r * R.i;
        Q.i = U.i * R.i;
        j = j + n;
        goto cfft2;

cfft2:  Z[i-n].r = T.r;
        Z[i-n].i = T.i;
        T.r = P.r - Q.i;
        T.i = P.i + Q.r;
        P.r = Z[i].r;
        P.i = Z[i].i;
        i = i + n;
        goto cfft1;
```

ここでは、一周目で読み込んだデータが、一周目で複素乗算され、二周目で加算/減算され、メモリに書き戻される。

17.7. 多倍長整数の加減乗算

多倍長整数の加算ループ

多倍長整数の加算では、addbig1, addbig2なる2命令で、最内周ループを構成する。以下、X, Yをソース側の多倍長整数オペランド(整数配列)、PSをデスティネーション側の配列とする。

addbig1のソース・オペランドは、配列xの要素。デスティネーションは、配列PSの要素。暗黙のデスティネーションとして、SレジスタとTレジスタが使用される。 addbig2のソースは配列yの要素。暗黙のデスティネーションとして、SレジスタとTレジスタが使用される。

使い方

```
loop:   addbig1  X[i], PS[i-1]
        addbig2  Y[i]; i = i + 1; goto loop;
```

動作

```
addbig1:  T = S + T;
          PS[i-1] = T<31:0>;
          S = X[i];
          goto addbig2;

addbig2:  T = S + T<63:32>;
          S = Y[i];
          i = i + 1;
          goto addbig1;
```

単純に、二つのオペランドの対応する桁を足し、更に前のステップからの桁上がりを加えて、結果の下半分をデスティネーションにストアするだけである。

多倍長整数の減算ループ

多倍長整数の減算では、subbig1, subbig2なる2命令で、最内周ループを構成する。以下、X, Yをソース側の多倍長整数オペランド(整数配列)、PSをデスティネーション側の配列とする。

subbig1のソース・オペランドは、配列xの要素。デスティネーションは、配列PSの要素。暗黙のデスティネーションとして、SレジスタとTレジスタが使用される。 subbig2のソースは配列yの要素。暗黙のデスティネーションとして、SレジスタとTレジスタが使用される。

使い方

```
loop:   subbig1  X[i], PS[i-1]
        subbig2  Y[i]; i = i + 1; goto loop;
```

動作

```
subbig1:  T = T - S;
          PS[i-1] = T<31:0>;
          S = X[i];
          goto subbig2;
subbig2:  T = S + (sign extended)T<63:32>;
          S = Y[i];
          i = i + 1;
          goto subbig1;
```

単純に、二つのオペランドの対応する桁を減算し、更に前のステップからの桁上がりを加えて、結果の下半分をデスティネーションにストアするだけである。

多倍長整数の乗算ループ

多倍長整数の乗算では、mulbig1, mulbig2なる2命令で最内周ループを構成する。最内周ループでは、一つのワードと一つの多倍長整数の積を、部分積オペランドに加える。以下、X, Yをソース側の多倍長整数オペランド(整数配列)、 PP をデスティネーション側の配列(部分積オペランド)とする。最内周ループを使用するときは、ループ外でTレジスタに Y[i] をセットしておく。

mulbig1のソース・オペランドは、配列xの要素とTレジスタ。暗黙のデスティネーションは、SレジスタとPレジスタ。命令記述上のデスティネーションは、部分積オペランドの要素。
mulbig2のソースは部分積オペランドの要素と、SおよびPレジスタ。デスティネーションは、Sレ

ジスタとPレジスタ。

使い方

```
loop:   mulbig1  X[i], PP[i+j-1]
        mulbig2  PP[i+j];  i = i + 1; goto loop;
```

動作

```
mulbig1:   P = X[i] * T;
           S = S + P;
           PP[i+j-1] = S<31:0>;

mulbig2:   P = PP[i+j];
           S = S<63:32> + P;
           i = i + 1;
```

17.8. 多項式の計算

N次多項式の値を計算するには、honorf/honordという命令で最内周ループ(Honorループ)を形成し、係数配列に対して連続的演算を行なう。倍精度(honord)でも単精度(honorf)でも、オプコード以外は全く同じように処理するので、以下の説明では、総称としてhonor命令と呼ぶことにする。 A は、オプコードに対応する精度を持つ、ソース側の係数配列オペランド(浮動小数点配列)である。

最内周ループを使用するときは、ループ外で以下の準備をしておく。

(1) Sレジスタにパラメータ(x)をセットし、
(2) Pレジスタに1をいれ、
(3) Tレジスタは0に、
(4) Qレジスタも0に

セットする。

honorのソース・オペランドは、配列Aの要素とSレジスタ。暗黙のデスティネーションは、P,Q,R,Tレジスタ。

使い方

```
loop:   honor  A[i];  goto loop;
```

動作

```
honor:   T = T + Q;
         R = A[i];
         Q = A[i] * P;
         P = P * S;
```

Appendix B

FLATS2

Instruction Set Manual

THE FLATS2 INSTRUCTION SET

DRAFT V0.4 – November 13, 1990

Paul Spee
Computer Architecture Group
Quantum Magneto Flux Logic Project
Research Development Corporation of Japan

Revision history

v0.4	90/11/13	spec	modification K format
v0.3	90/05/14	spec	specification add, compare and branch
v0.2	90/02/01	spec	various modifications
v0.1	89/11/30	spec	various modifications
v0.0		spec	initial version

1. Addressing modes

1.1 Basic addressing modes

In this section we describe the basic addressing modes which consist of immediate data, register and memory addressing modes.

1.1.1 Immediate data

Immediate data is either treated as a 8-bit signed/unsigned integer, a 32-bit signed/unsigned integer, or a 64-bit signed/unsigned integer.

1.1.2 Registers

GV-registers

GV-registers are general purpose registers, which contain of 32 data bits and 1 address tag bit. If the address tag is clear, the GV-register contains a 32 bit integer. If the address tag is set, the GV-register contains an address (pointer).

There are 32 global GV-registers, named gr0 to gr31. There are also 32 local GV-registers, named vr0 to vr31. A new set of local registers are 'allocated' on each call. The return instruction *ret* 'restores' the old local register set.

Each even (2N) and odd (2N+1) register form a pair, called BL-pair. The BL-pair is used during address calculation. After the address calculated, it is checked against the address range as specified by the BL-pair. If the address falls outside the address range as specified by the BL-pair, and exception occurs. How this exception is handled depends on the instruction. (See the individual description of each instruction). This range checking is called BL-checking. If an odd register is specified, this register is used as base register, and the corresponding even register is used as limit register.

SP-registers

The S- and P-registers are the registers of arithmetic unit. They can be used for arithmetic operations involving integers (32- or 64-bit) and floats (32- or 64-bit).

If either S- or P-register is used as destination operand, then only one of these registers can be used as destination operand, depending on the instruction. (See the description of the individual instructions).

1.1.3 Memory addressing

Each memory address includes a BL-pair. The effective address is checked against the BL-pair; if the check fails, an exception occurs. How this exception is handled, depends on the instruction. The registers specified in the memory address are GV-registers.

Index mode

The effective address is calculated by adding the contents of the index register to the base register. The index register contains a signed integer.

Offset mode

The effective address is calculated by adding an immediate displacement to the base register.

Pointer mode
The effective address is calculated by adding an immediate displacement to the pointer register. The pointer register contains an address, that is, its address tag is set.

1.2 Side-effect

The memory addressing modes can use either of the two following side-effects.

Push side-effect
In case of the push side-effect, the effective address is written to the base register (index mode, offset mode) or the pointer register (pointer mode).

Pop side-effect
In case of the pop side-effect, the effective address is the contents of the base register (index mode, offset mode) or the pointer register (pointer mode) and the calculated address is written to the base register (index mode, offset mode) or the pointer register (pointer mode).

1.3 Special addressing modes

Two special addressing mode are available for address calculation (lea instruction).

index offset mode
The effective address is the contents of the base register plus the contents of the index register and immediate displacement.

pointer index mode
The effective address is the contents of the pointer register plus an immediate displacement and the contents of the index register.

1.4 Summary

In this section is a summary of the addressing modes for FLATS2.

Base is a GV-register which is used as the base register in the BL-pair. As such, the address tag should be set. *Index* is a GV-register which contains a signed integer (e.g. the address tag is clear). *Pointer* is a GV-register which contains an address (e.g. the address tag is set). *Displ* is a signed integer, which is either 8 bits or 32 bits.

Mode	Notation	Effective address	Side-effect
Immediate	#xxx		
GV-register	gr*nn*, vr*nn*, sp, fp		
SP-register	S, P		
Index	*base@index*	BL:*base+index*	
Index push	*base@>index*	BL:*base+index*	*base* ← *base+index*
Index pop	*base@<index*	BL:*base*	*base* ← *base+index*
Offset	*base@displ*	BL:*base+displ*	
Offset push	*base@>displ*	BL:*base+displ*	*base* ← *base+displ*
Offset pop	*base@<displ*	BL:*base*	*base* ← *base+displ*
Pointer	*base:displ(pointer)*	BL:*pointer+displ*	
	base:&address(index)	BL:*address+index*	
Pointer push	*base:>displ(pointer)*	BL:*pointer+displ*	*pointer* ← *pointer+displ*
Pointer pop	*base:<displ(pointer)*	BL:*pointer*	*pointer* ← *pointer+displ*
Index offset	*base@displ(index)*	BL:*base+index+displ*	
Pointer index	*base:displ(pointer)index*	BL:*pointer+displ+index*	

Table 1. Addressing modes

1.5 Addressing mode and instruction format

I-format/IL-format
The I-format is used for the GV-unit instructions. The operands can be either a GV-register or an immediate or a memory operand. Only one memory operand is allowed. This memory operand can not have a side-effect. The IL-format is used when the immediate is 32-bits.

$R \times R \rightarrow R$
$R \times IMM \rightarrow R$
$IMM \times R \rightarrow R$
$MEM \rightarrow R$
$R \rightarrow MEM$

J-format/JL-format

K-format/KL-format

ldb, ldh, ldw, ldq M(X,O,P) → R
stb, sth, stw, stq R → M(X,O,P)

Each instruction takes 1 cycle.

M-format/ML-format

{R/IMM} × {R} → R
{R} × {R/IMM} → R

{R/IMM} × MEM → R [1,2,5]

MEM × {R/IMM} → R [1,2,5]

{R/IMM} × {R} → MEM [2,6]
{R} × {R/IMM} → MEM [2,6]

{R/IMM} × MEM → MEM [2,3,4]

MEM × {R/IMM} → MEM [2,3,4]

MEM × MEM → R [1,3]

1. If the target operand is a GV-register, the memory operand can have offset mode only (with optional pop side-effect).
2. If a long displacement is used for either offset mode or pointer mode, an immediate cannot be used.
3. If a different BL is used, only *index* and *offset* mode can be used. The *pop* size-effect is allowed for both operands.
4. If the same BL is used, both memory operands can be either *index*, *offset* or *pointer* mode. Only one of the memory operands can have a side-effect (*pop* or *push*).
5. The source memory operand can only have pop as a side-effect.
6. The destination memory operand can only have push as a side-effect.

MOV — move scalar quantity

FORMAT mov.<src data type> <src>,<dst>

OPERATION
dst ← src
dst<atag> ← src<atag>
dst<atag'> ← src<atag'> †

CONDITION CODES unchanged

MNEMONICS

mov.l	move longword
mov.q	move quadword
mov.f	move single floating
mov.d	move double floating

OPERANDS If the source or destination operand is a GV register and the instruction is mov.q, the operand specifies an even/odd register pair.

OPCODES

M.movw	mov.l
M.movq	mov.q
M.movf	mov.f
M.movd	mov.d

NOTE All MOV operations are done using M format. (See also MOVW.)

† Move 64 data bits and two address tags (mov.q and mov.d).

MOVJ — move scalar quantity and jump

FORMAT mov.<src data type>.j <src>,<dst>,<label>

OPERATION
if exception → pc ← next instruction
if ¬ exception →
dst ← src
dst<atag> ← src<atag>
dst<atag'> ← src<atag'> †
pc ← label

CONDITION CODES unchanged

MNEMONICS
mov.l.j move longword and jump to label
mov.q.j move quadword and jump to label
mov.f.j move single floating and jump to label
mov.d.j move double floating and jump to label

OPERANDS If the source or destination operand is a GV register and the instruction is mov.q, the operand specifies an even/odd register pair.

OPCODES
M.movw mov.l.j
M.movq mov.q.j
M.movf mov.f.j
M.movd mov.d.j

NOTE All MOV operations are done using M format. (See also MOVW.)
If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† Move 64 data bits and two address tags (mov.q.j and mov.d.j).

MOVW — move word

FORMAT movw <src>,<dst>

OPERATION
dst ← src
dst<atag> ← src<atag>

CONDITION CODES unchanged

MNEMONICS movw move word

OPCODES
If the operands are a register or an immediate,
I.cpr movw R ← R/IMM
else if the operands are a register and
a memory reference without side-effect,
K.ldw movw R ← M
K.stw movw M ← R/IMM
otherwise, when both operands are memory references,
or one of the memory operands has a side-effect,
M.movw movw

NOTE Move between register and register or between register and memory is faster than in case of mov.l. (See also MOV.)

MOVWJ — move word and jump

FORMAT movw.j <src>,<dst>,<label>

OPERATION
if exception → next address
if –exception →
dst ← src
dst<atag> ← src<atag>
pc ← label

CONDITION CODES unchanged

MNEMONICS movw.j move word

OPERANDS No side-effect can be specified for the memory operand. It is not possible to specify a SP register.

OPCODES
If the operands are a register or an immediate,

I.cpr	movw.j	R ← R/IMM

else if the operands are a register and
a memory reference without side-effect,

K.ldw	movw.j	R ← M
K.stw	movw.j	M ← R/IMM

otherwise, when both operands are memory references,
or one of the memory operands has a side-effect,

M.movw	movw.j	

NOTE Move between register and register or between register and memory is faster than in case of mov.l.j. (See also MOVJ.)

MOVDW — move double word

FORMAT movdw <src>,<dst>

OPERATION
dst<31:0> ← src<31:0>
dst<atag> ← src<atag>
dst<63:32> ← src<63:32>
dst<atag'> ← src<atag'>

CONDITION CODES unchanged

MNEMONICS movdw move double word

OPERANDS If the source or destination operand is a GV register, the operand specifies an even/odd register pair.

OPCODES
If the operands are a register or an immediate,

I.cprp	movdw	R ← R/IMM

else if the operands are a register and
a memory reference without side-effect,

K.ldq	movdw	R ← M
K.stq	movdw	M ← R

otherwise, when both operands are memory references,
or one of the memory operands has a side-effect,

M.movq	movdw	

NOTE Move between register and register or between register and memory is faster than in case of mov.q. (See also MOV.)

MOVDWJ — move double word and jump

FORMAT movdw.j <src>,<dst>,<label>

OPERATION
if exception → next address
if ¬exception →
dst<31:0> ← src<31:0>
dst<atag> ← src<atag>
dst<63:32> ← src<63:32>
dst<atag'> ← src<atag'>
pc ← label

CONDITION CODES unchanged

MNEMONICS movdw.j move double word

OPERANDS If the source or destination operand is a GV register, the operand specifies an even/odd register pair.

OPCODES
If the operands are a register or an immediate,
I.cprp movdw.j R ← R/IMM
else if the operands are a register and
a memory reference without side-effect,
K.ldq movdw.j R ← M
K.stq movdw.j M ← R
otherwise, when both operands are memory references,
or one of the memory operands has a side-effect,
M.movq movdw.j

NOTE Move between register and register or between register and memory is faster than in case of mov.q.j. (See also MOVJ.)

CVT — convert signed value to different signed data type

FORMAT cvt.<src data type><dst data type> <src>,<dst>

OPERATION
dst ← conversion of (src)
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← ?

MNEMONICS
cvt.qd convert quadword to double floating
cvt.dq convert double floating to quadword
cvt.lq extend signed long integer to quad integer

OPERANDS The P register can **not** be specified as the destination operand.

OPCODES
M.floatq cvt.qd
M.ifixd cvt.dq
M.ext cvt.lq

NOTE

† When the size of the destination is 64 bits, two address tags are cleared. (cvt.ld, cvt.qd, cvt.fq, cvtdq)

CVTJ **convert signed value to different signed data type and jump**

FORMAT

cvt.<src data type><dst data type>.j <src>,<dst>,<label>

OPERATION

```
if exception → pc ← next instruction
if ¬ exception →
        dst ← conversion of (src)
        dst<atag> ← 0
        dst<atag'> ← 0 †
        pc ← <label>
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← ?

MNEMONICS

cvt.qd.j	convert quadword to double floating and jump
cvt.dq.j	convert double floating to quadword and jump
cvt.lq.j	sign extend long integer to quad integer and jump

OPERANDS

The P register can **not** be specified as the destination operand.

OPCODES

M.floatq	cvt.qd.j
M.ifixd	cvt.dq.j
M.ext	cvt.lq.j

NOTE

† When the size of the destination is 64 bits, two address tags are cleared. (cvt.qd.j, cvt.dq.j)

LD **load small integer into GV register**

FORMAT

ld.<src data type> <src>,<dst>
ld.<src data type>.j <src>,<dst>,<label>

OPERATION

```
dst[31:0] ← sign-extend (src<size-1:0>)
dst<atag> ← 0
pc ← <label>    ; ld.j
```

CONDITION CODES

unchanged

MNEMONICS

ld.b	load and sign-extend byte into GV register
ld.s	load and sign-extend short into GV register
ld.b.j	load and sign-extend byte into GV register and jump
ld.s.j	load and sign-extend short into GV register and jump

OPERANDS

No side-effect can be specified for the memory operand. The register operand must be a GV register.

OPCODES

K.extb	ld.b
K.exth	ld.s

NOTE

LDJ **load small integer into GV register and jump**

FORMAT ld.<src data type>.ej <src>,<dst>,<label>

OPERATION
if exception → pc ← next address
if ¬ exception →
dst<31:0> ← sign-extend (src<size-1:0>)
dst<atag> ← 0
pc ← label

CONDITION CODES unchanged

MNEMONICS
ld.b.j load and sign-extend byte into GV register and jump
ld.s.j load and sign-extend short into GV register and jump

OPERANDS No side-effect can be specified for the memory operand. The register operand must be a GV register.

OPCODES
K.extb ld.b.j
K.exth ld.s.j

NOTE

LDZ **load unsigned small integer into GV register**

FORMAT ldz.<src data type> <src>,<dst>

OPERATION
dst<31:0> ← zero-extend (src<size-1:0>)
dst<atag> ← 0

CONDITION CODES unchanged

MNEMONICS
ldz.b load and zero-extend byte into GV register
ldz.s load and zero-extend short into GV register

OPERANDS No side-effect can be specified for the memory operand. The register operand must be a GV register.

OPCODES
K.ldb ldz.b
K.ldh ldz.s

NOTE

LDZJ — load small unsigned integer into GV register and jump

FORMAT

ldz.<src data type>.j <src>,<dst>,<label>

OPERATION

if exception → pc ← next address
if ¬ exception →
dst<31:0> ← zero-extend (src<size-1:0>)
dst<atag> ← 0
pc ← label

CONDITION CODES

unchanged

MNEMONICS

ldz.b.j load and zero-extend byte into GV register and jump
ldz.s.j load and zero-extend short into GV register and jump

OPERANDS

No side-effect can be specified for memory operand. The register operand must be a GV register.

OPCODES

K.ldb ldz.b.j
K.ldh ldz.s.j

NOTE

ST — store small integer in GV register to memory

FORMAT

st.<src data type> <src>,<dst>
st.<src data type>.j <src>,<dst>,<label>

OPERATION

dst<size-1:0> ← src<size-1:0>
dst<atag> ← 0
pc ← <label> ; st.j

CONDITION CODES

unchanged

MNEMONICS

st.b store byte in GV register to memory
st.s store short in GV register to memory
st.b.j store byte in GV register to memory and jump
st.s.j store short in GV register to memory and jump

OPERANDS

No side-effect can be specified for memory operand. The source operand must be a GV register.

OPCODES

K.stb st.b
K.sth st.s

NOTE

STJ — store small integer in GV register to memory and jump

FORMAT st.<src data type>.j <src>,<dst>,<label>

OPERATION
if exception → pc ← next address
if ¬ exception →
dst<size-1:0> ← src<size-1:0>
dst<atag> ← 0
pc ← label

CONDITION CODES unchanged

MNEMONICS
st.b.j store byte to memory, jump if error
st.s.j store short to memory, jump if error

OPERANDS No side-effect can be specified for the memory operand. The source operand must be a GV register.

OPCODES
K.stb st.b.j
K.sth st.s.j

NOTE

EXT — sign-extend GV register

FORMAT ext.<src type> <byte position>, <src>, <dst> ext.<src type>.j <byte position>, <src>, <dst>, <label>

OPERATION
if src<atag> → trap
if ¬ src<atag> →
dst<31:0> ← sign-extend (src<size-1:0>[byte position])
dst<atag> ← 0
pc ← <label> ; ext.j

CONDITION CODES unchanged

MNEMONICS
ext.b sign-extend byte in register
ext.s sign-extend short in register
ext.b.j sign-extend byte in register and jump
ext.s.j sign-extend short in register and jump

OPERANDS The source and destination operands are GV registers, while the byte position can either be specified as GV register or immediate.

OPCODES
I.extrb ext.b
I.extrh ext.s

NOTE An trap is generated, if the source operand has an address tag.

ABS — absolute value

FORMAT abs.<src data type> <src>,<dst> two operand

OPERATION
dst ← abs(src)
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS

abs.l	absolute of longword
abs.q	absolute of quadword
abs.f	absolute of single floating
abs.d	absolute of double floating

OPERANDS The P register can **not** be specified as destination operand.

OPCODES

M.absw	abs.l
M.absq	abs.q
M.absf	abs.f
M.absd	abs.d

NOTE

† A 64-bit destination operand includes two address tags.

ABSJ — absolute value and jump

FORMAT abs.<src data type>.j <src>,<dst>,<label>

OPERATION
if exception → pc ← next instruction
if ¬ exception →
dst ← abs(src)
dst<atag> ← 0
dst<atag'> ← 0 †
pc ← label

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS

abs.l.j	absolute of longword and jump
abs.q.j	absolute of quadword and jump
abs.f.j	absolute of single floating and jump
abs.d.j	absolute of double floating and jump

OPERANDS The P register can **not** be specified as destination operand.

OPCODES

M.absw	abs.l.j
M.absq	abs.q.j
M.absf	abs.f.j
M.absd	abs.d.j

NOTE

† A 64-bit destination operand includes two address tags.

NEG — negative value

FORMAT neg.<src data type> <src>,<dst> two operand

OPERATION
dst ← neg(src)
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS
neg.l negative of longword
neg.q negative of quadword

neg.f negative of single floating
neg.d negative of double floating

OPERANDS The P register can **not** be specified as destination operand.

OPCODES
M.negw neg.l
M.negq neg.q
M.negf neg.f
M.negd neg.d

NOTE

† A 64-bit destination operand includes two address tags.

NEGJ — negative value and jump

FORMAT neg.<src data type>.j <src>,<dst>,<label>

OPERATION
if exception → pc ← next instruction
dst ← neg(src)
dst<atag> ← 0
dst<atag'> ← 0 †
pc ← label

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS
neg.l.j negative of longword and jump
neg.q.j negative of quadword and jump

neg.f.j negative of single floating and jump
neg.d.j negative of double floating and jump

OPERANDS The P register can **not** be specified as destination operand.

OPCODES
M.negw neg.l.j
M.negq neg.q.j
M.negf neg.f.j
M.negd neg.d.j

NOTE

† A 64-bit destination operand includes two address tags.

NOT — complement value

FORMAT not.<src data type> <src>,<dst> two operand

OPERATION
dst ← not(src)
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS
not.l complement of longword
not.q complement of quadword

OPERANDS The P register can **not** be specified as destination operand.

OPCODES
M.notw not.l
M.notq not.q

NOTE

† A 64-bit destination operand includes two address tags.

NOT.J — complement value and jump

FORMAT not.<src data type>.j <src>,<dst>,<label>

OPERATION
if exception → pc ← next address
if ¬ exception →
dst ← not(src)
dst<atag> ← 0
dst<atag'> ← 0 †
pc ← label

CONDITION CODES
N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS
not.l.j complement of longword and jump
not.q.j complement of quadword and jump

OPERANDS The P register can **not** be specified as destination operand.

OPCODES
M.notw not.l.j
M.notq not.q.j

NOTE

† A 64-bit destination operand includes two address tags.

ADD — arithmetic add

FORMAT

add.<src data type>	<src>,<dst>	two operand
add3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

dst ← src + dst two operand
dst ← src1 + src2 three operand
dst<atag> ← 0
dst<atag´> ← 0 †

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS

add.l	add longword, two operand
add3.l	add longword, three operand
add.q	add quadword, two operand
add3.q	add quadword, three operand
add.f	add single floating, two operand
add3.f	add single floating, three operand
add.d	add double floating, two operand
add3.d	add double floating, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.addw	add.l, add3.l
M.addq	add.q, add3.q
M.addf	add.f, add3.f
M.addd	add.d, add3.d

NOTE

† A 64-bit destination operand includes two address tags.

ADDJ — arithmetic add and jump

FORMAT

add.<src data type>.j	<src>,<dst>,<label>	two operand
add3.<src data type>.j	<src1>,<src2>,<dst>,<label>	three operand

OPERATION

if exception → pc ← next instruction
if ¬ exception →
dst ← src + dst two operand
dst ← src1 + src2 three operand
dst<atag> ← 0
dst<atag´> ← 0 †
pc ← label

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS

add.l.j	add longword and jump, two operand
add3.l.j	add longword and jump, three operand
add.q.j	add quadword and jump, two operand
add3.q.j	add quadword and jump, three operand
add.f.j	add single floating and jump, two operand
add3.f.j	add single floating and jump, three operand
add.d.j	add double floating and jump, two operand
add3.d.j	add double floating and jump, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.addw	add.l.j, add3.l.j
M.addq	add.q.j, add3.q.j
M.addf	add.f.j, add3.f.j
M.addd	add.d.j, add3.d.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

ADDR — add using GV registers

FORMAT

addr <src1>,<src2>,<dst>

OPERATION

if src1<atag> | src2<atag> → trap
if ¬ src1<atag> & ¬ src2<atag> →
dst ← src1 + src2
dst<atag> ← 0

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← carry

MNEMONICS

addr add register

OPCODES

I.add addr

NOTE

SUB — arithmetic subtract

FORMAT

sub.<src data type>	<src>,<dst>	two operand
sub3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

dst ← src - dst two operand
dst ← src1 - src2 three operand
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← borrow

MNEMONICS

sub.l	subtract longword, two operand
sub3.l	subtract longword, three operand
sub.q	subtract quadword, two operand
sub3.q	subtract quadword, three operand
sub.f	subtract single floating, two operand
sub3.f	subtract single floating, three operand
sub.d	subtract double floating, two operand
sub3.d	subtract double floating, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.subw	sub.l, sub3.l
M.subq	sub.q, sub3.q
M.subf	sub.f, sub3.f
M.subd	sub.d, sub3.d

NOTE

† A 64-bit destination operand includes two address tags.

SUBJ — arithmetic subtract and jump

FORMAT

sub.<src data type>.j <src>,<dst>,<label> two operand
sub3.<src data type>.j <src1>,<src2>,<dst>,<label> three operand

OPERATION

```
if exception → pc ← next instruction
if ¬ exception →
        dst ← src - dst                 two operand
        dst ← src1 - src2               three operand
        dst<atag> ← 0
        dst<atag´> ← 0 †
        pc ← label
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← borrow

MNEMONICS

sub.l.j	subtract longword and jump, two operand
sub3.l.j	subtract longword and jump, three operand
sub.q.j	subtract quadword and jump, two operand
sub3.q.j	subtract quadword and jump, three operand
sub.f.j	subtract single floating and jump, two operand
sub3.f.j	subtract single floating and jump, three operand
sub.d.j	subtract double floating and jump, two operand
sub3.d.j	subtract double floating and jump, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.subw	sub.l.j, sub3.l.j
M.subq	sub.q.j, sub3.q.j
M.subf	sub.f.j, sub3.f.j
M.subd	sub.d.j, sub3.d.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

SUBR — subtract using GV registers

FORMAT

subr <src1>,<src2>,<dst>

OPERATION

```
if src1<atag> | src2<atag> → trap
if ¬ src1<atag> & ¬ src2<atag> →
        dst ← src1 - src2
        dst<atag> ← 0
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← overflow
C ← borrow

MNEMONICS

subr subtract register

OPCODES

I.sub subr

NOTE

MUL — signed multiply

FORMAT

mul.<src data type>	<src>,<dst>	two operand
mul3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

dst ← src * dst — two operand
dst ← src1 * src2 — three operand
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES

N ← (dst < 0) §
Z ← (dst = 0)
V ← overflow
C ← 0

MNEMONICS

mul.l	multiply longword, two operand
mul3.l	multiply longword, three operand
mul.q	multiply quadword, two operand
mul3.q	multiply quadword, three operand
mul.f	multiply single floating, two operand
mul3.f	multiply single floating, three operand
mul.d	multiply double floating, two operand
mul3.d	multiply double floating, three operand

OPERANDS

The S register can **not** be specified as destination operand.

OPCODES

M.mulw	mul.l, mul3.l
M.mulq	mul.q, mul3.q
M.mulf	mul.f, mul3.f
M.muld	mul.d, mul3.d

NOTE

Floating word in SP-register is always 64-bit floating.

† A 64-bit destination operand includes two address tags.
§ The condition codes are **not** set for integer multiply.

MUL.J — signed multiply and jump

FORMAT

mul.<src data type>.j	<src>,<dst>,<label>	two operand
mul3.<src data type>.j	<src1>,<src2>,<dst>,<label>	three operand

OPERATION

if exception → pc ← next instruction
if ¬ exception →
    dst ← src * dst — two operand
    dst ← src1 * src2 — three operand
    dst<atag> ← 0
    dst<atag'> ← 0 †
    pc ← label

CONDITION CODES

N ← (dst < 0) §
Z ← (dst = 0)
V ← overflow
C ← 0

MNEMONICS

mul.l.j	multiply longword and jump, two operand
mul3.l.j	multiply longword and jump, three operand
mul.q.j	multiply quadword and jump, two operand
mul3.q.j	multiply quadword and jump, three operand
mul.f.j	multiply single floating and jump, two operand
mul3.f.j	multiply single floating and jump, three operand
mul.d.j	multiply double floating and jump, two operand
mul3.d.j	multiply double floating and jump, three operand

OPERANDS

The S register can **not** be specified as destination operand.

OPCODES

M.mulw	mul.l.j, mul3.l.j
M.mulq	mul.q.j, mul3.q.j
M.mulf	mul.f.j, mul3.f.j
M.muld	mul.d.j, mul3.d.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is muled to the execution time.

† A 64-bit destination operand includes two address tags.
§ The condition codes are **not** set for integer multiply.

MULU — unsigned multiply

FORMAT

mulu.<src data type>	<src>,<dst>	two operand
mulu3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

dst ← src * dst — two operand
dst ← src1 * src2 — three operand
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES

N ← (dst < 0) §
Z ← (dst = 0)
V ← overflow
C ← 0

MNEMONICS

mulu.l	multiply longword, two operand
mulu3.l	multiply longword, three operand
mulu.q	multiply quadword, two operand
mulu3.q	multiply quadword, three operand

OPERANDS

The S register can **not** be specified as destination operand.

OPCODES

M.muluw	mulu.l, mulu3.l
M.muluq	mulu.q, mulu3.q

NOTE

Floating word in SP-register is always 64-bit floating.

† A 64-bit destination operand includes two address tags.
§ The condition codes are **not** set for integer multiply.

MULUJ — unsigned multiply and jump

FORMAT

mulu.<src data type>.j	<src>,<dst>,<label>	two operand
mulu3.<src data type>.j	<src1>,<src2>,<dst>,<label>	three operand

OPERATION

if exception → pc ← next instruction
if ¬ exception →
dst ← src * dst — two operand
dst ← src1 * src2 — three operand
dst<atag> ← 0
dst<atag'> ← 0 †
pc ← label

CONDITION CODES

N ← (dst < 0) §
Z ← (dst = 0)
V ← overflow
C ← 0

MNEMONICS

mulu.l.j	multiply longword and jump, two operand
mulu3.l.j	multiply longword and jump, three operand
mulu.q.j	multiply quadword and jump, two operand
mulu3.q.j	multiply quadword and jump, three operand

OPERANDS

The S register can **not** be specified as destination operand.

OPCODES

M.muluw	mulu.l.j, mulu3.l.j
M.muluq	mulu.q.j, mulu3.q.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is muled to the execution time.

† A 64-bit destination operand includes two address tags.
§ The condition codes are **not** set for integer multiply.

AND — logical and

FORMAT	and.<src data type>	<src>,<dst>	two operand
	and3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

dst ← src & dst — two operand
dst ← src1 & src2 — three operand
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

and.l	and longword, two operand
and3.l	and longword, three operand
and.q	and quadword, two operand
and3.q	and quadword, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.andw	and.l, and3.l
M.andq	and.q, and3.q

NOTE

† A 64-bit destination operand includes two address tags.

ANDJ — logical and and jump

FORMAT	and.<src data type>.j	<src>,<dst>,<label>	two operand
	and3.<src data type>.j	<src1>,<src2>,<dst>,<label>	three operand

OPERATION

if exception → pc ← next instruction
if ¬ exception →
  dst ← src & dst — two operand
  dst ← src1 & src2 — three operand
  dst<atag> ← 0
  dst<atag'> ← 0 †
  pc ← label

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

and.l.j	exclusive or longword and jump, two operand
and3.l.j	exclusive or longword and jump, three operand
and.q.j	exclusive or quadword and jump, two operand
and3.q.j	exclusive or quadword and jump, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.andw	and.l.j, and3.l.j
M.andq	and.q.j, and3.q.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

ANDR — logical and on GV register

FORMAT

andr <src1>,<src2>,<dst>

OPERATION

```
if src1<atag> | src2<atag> → trap
if ¬ src1<atag> & ¬ src2<atag> →
        dst ← src1 & src2
        dst<atag> ← 0
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

andr and register

OPCODES

Land andr

NOTE

OR — logical inclusive or

FORMAT

or.<src data type>	<src>,<dst>	two operand
or3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

```
dst ← src | dst        two operand
dst ← src1 | src2      three operand
dst<atag> ← 0
dst<atag'> ← 0 †
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

or.l	or longword, two operand
or3.l	or longword, three operand
or.q	or quadword, two operand
or3.q	or quadword, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.orw	or.l, or3.l
M.orq	or.q, or3.q

NOTE

† A 64-bit destination operand includes two address tags.

ORJ — logical inclusive or and jump

FORMAT

or.<src data type>.j	<src>,<dst>,<label>	two operand
or3.<src data type>.j	<src1>,<src2>,<dst>,<label>	three operand

OPERATION

```
if exception → pc ← next instruction
if ¬ exception →
        dst ← src | dst                          two operand
        dst ← src1 | src2                        three operand
        dst<atag> ← 0
        dst<atag´> ← 0 †
        pc ← label
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

or.l.j	inclusive or longword and jump, two operand
or3.l.j	inclusive or longword and jump, three operand
or.q.j	inclusive or quadword and jump, two operand
or3.q.j	inclusive or quadword and jump, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.orw	or.l.j, or3.l.j
M.orq	or.q.j, or3.q.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

ORR — logical inclusive or on GV registers

FORMAT

orr <src1>,<src2>,<dst>

OPERATION

```
if src1<atag> | src2<atag> → trap
if ¬ src1<atag> & ¬ src2<atag> →
        dst ← src1 | src2
        dst<atag> ← 0
```

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

orr or register

OPCODES

I.or orr

NOTE

XOR — logical exclusive or

FORMAT

xor.<src data type>	<src>,<dst>	two operand
xor3.<src data type>	<src1>,<src2>,<dst>	three operand

OPERATION

dst ← src ^ dst — two operand
dst ← src1 ^ src2 — three operand
dst<atag> ← 0
dst<atag'> ← 0 †

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

xor.l	exclusive or longword, two operand
xor3.l	exclusive or longword, three operand
xor.q	exclusive or quadword, two operand
xor3.q	exclusive or quadword, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.xorw	xor.l, xor3.l
M.xorq	xor.q, xor3.q

NOTE

† A 64-bit destination operand includes two address tags.

XORJ — logical exclusive or and jump

FORMAT

xor.<src data type>.j	<src>,<dst>,<label>	two operand
xor3.<src data type>.j	<src1>,<src2>,<dst>,<label>	three operand

OPERATION

if exception → pc ← next instruction
if ¬ exception →
  dst ← src ^ dst — two operand
  dst ← src1 ^ src2 — three operand
  dst<atag> ← 0
  dst<atag'> ← 0 †
  pc ← label

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

xor.l.j	exclusive or longword and jump, two operand
xor3.l.j	exclusive or longword and jump, three operand
xor.q.j	exclusive or quadword and jump, two operand
xor3.q.j	exclusive or quadword and jump, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.xorw	xor.l.j, xor3.l.j
M.xorq	xor.q.j, xor3.q.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

XORR — logical exclusive or on GV register

FORMAT

xorr <src1>,<src2>,<dst>

OPERATION

if src1<atag> | src2<atag> → trap
if ¬ src1<atag> & ¬ src2<atag> →
dst ← src1 ^ src2
dst<atag> ← 0

CONDITION CODES

N ← (dst < 0)
Z ← (dst = 0)
V ← 0
C ← 0

MNEMONICS

xorr exclusive or on register

OPCODES

I.xor xorr

NOTE

LSL/ASL — logical/arithmetic shift left

FORMAT

lsl.<src data type>	<cnt>,<dst>	two operand
lsl3.<src data type>	<src>,<cnt>,<dst>	three operand
asl.<src data type>	<cnt>,<dst>	two operand
asl3.<src data type>	<src>,<cnt>,<dst>	three operand

OPERATION

dst<31:0> ← dst<31-cnt:0> # 0<cnt-1:0> two operand
dst<31:0> ← src<31-cnt:0> # 0<cnt-1:0> three operand
dst<atag> ← 0
dst<atag´> ← 0 †

CONDITION CODES

unchanged

MNEMONICS

lsl.l	shift left longword, two operand
lsl3.l	shift left longword, three operand
lsl.q	shift left quadword, two operand
lsl3.q	shift left quadword, three operand
asl.l	shift left longword, two operand
asl3.l	shift left longword, three operand
asl.q	shift left quadword, two operand
asl3.q	shift left quadword, three operand

OPERANDS

The P register can not be specified as destination operand.

OPCODES

M.lslw lsl.l, lsl3.l, asl.l, asl3.l
M.lslq lsl.q, lsl3.q, asl.q, asl3.q

NOTE

Asl is equivalent to lsl.

† A 64-bit destination operand includes two address tags.

LSLJ/ASLJ — logical/arithmetic shift left and jump

FORMAT

lsl.<src data type>.j	<cnt>,<dst>,<label>	two operand
lsl3.<src data type>.j	<src>,<cnt>,<dst>,<label>	three operand
asl.<src data type>.j	<cnt>,<dst>,<label>	two operand
asl3.<src data type>.j	<src>,<cnt>,<dst>,<label>	three operand

OPERATION

```
if exception → pc ← next instruction
if ¬ exception →
      dst<31:0> ← dst<31-cnt:0> # 0<cnt-1:0>   two operand
      dst<31:0> ← src<31-cnt:0> # 0<cnt-1:0>   three operand
      dst<catag> ← 0
      dst<catag'> ← 0 †
      pc ← label
```

CONDITION CODES unchanged

MNEMONICS

lsl.l.j	shift left longword, two operand and jump
lsl3.l.j	shift left longword, three operand and jump
lsl.q.j	shift left quadword, two operand and jump
lsl3.q.j	shift left quadword, three operand and jump
asl.l.j	shift left longword, two operand and jump
asl3.l.j	shift left longword, three operand and jump
asl.q.j	shift left quadword, two operand and jump
asl3.q.j	shift left quadword, three operand and jump

OPERANDS The P register can **not** be specified as destination operand.

OPCODES

M.lslw	lsl.l.j, lsl3.l.j, asl.l.j, asl3.l.j
M.lslq	lsl.q.j, lsl3.q.j, asl.q.j, asl3.q.j

NOTE Asl is equivalent to lsl. If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

LSR — logical shift right

FORMAT

lsr.<src data type>	<cnt>,<dst>	two operand
lsr3.<src data type>	<src>,<cnt>,<dst>	three operand

OPERATION

```
dst<31:0> ← 0<cnt-1:0> # dst<31:cnt>   two operand
dst<31:0> ← 0<cnt-1:0> # src<31:cnt>   three operand
dst<catag> ← 0
dst<catag'> ← 0 †
```

CONDITION CODES unchanged

MNEMONICS

lsr.l	logical shift right longword, two operand
lsr3.l	logical shift right longword, three operand
lsr.q	logical shift right quadword, two operand
lsr3.q	logical shift right quadword, three operand

OPERANDS The P register can **not** be specified as destination operand.

OPCODES

M.lsrw	lsr.l, lsr3.l
M.lsrq	lsr.q, lsr3.q

NOTE

† A 64-bit destination operand includes two address tags.

LSRJ — logical shift right and jump

FORMAT

lsr.<src data type>.j <cnt>,<dst>,<label> two operand
lsr3.<src data type>.j <src>,<cnt>,<dst>,<label> three operand

OPERATION

if exception → pc ← next instruction
if ¬ exception →
dst<31:0> ← 0<cnt-1:0> # dst<31:cnt> two operand
dst<31:0> ← 0<cnt-1:0> # src<31:cnt> three operand
dst<atag> ← 0
dst<atag´> ← 0 †
pc ← label

CONDITION CODES

unchanged

MNEMONICS

lsr.l.j logical shift right longword, two operand and jump
lsr3.l.j logical shift right longword, three operand and jump
lsr.q.j logical shift right quadword, two operand and jump
lsr3.q.j logical shift right quadword, three operand and jump

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.lsrw lsr.l.j, lsr3.l.j
M.lsrq lsr.q.j, lsr3.q.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

ASR — arithmetic shift right

FORMAT

asr.<src data type> <cnt>,<dst> two operand
asr3.<src data type> <src>,<cnt>,<dst> three operand

OPERATION

<dst>[31:0] ← (-<dst>[31])[cnt-1:0] # <dst>[31:cnt] two operand
<dst>[31:0] ← (-<src>[31])[cnt-1:0] # <src>[31:cnt] three operand
<dst>[atag] ← 0
<dst>[atag´] ← 0 †

CONDITION CODES

unchanged

MNEMONICS

asr.l arithmetic shift right longword, two operand
asr3.l arithmetic shift right longword, three operand
asr.q arithmetic shift right quadword, two operand
asr3.q arithmetic shift right quadword, three operand

OPERANDS

The P register can **not** be specified as destination operand.

OPCODES

M.asrw asr.l, asr3.l
M.asrq asr.q, asr3.q

NOTE

† A 64-bit destination operand includes two address tags.

ASRJ — arithmetic shift right and jump

FORMAT

```
asr.<src data type>.j    <cnt>,<dst>,<label>          two operand
asr3.<src data type>.j   <src>,<cnt>,<dst>,<label>    three operand
```

OPERATION

```
if exception → pc ← next instruction
if ¬ exception →
        dst<31:0> ← (-dst<31>)<cnt-1:0> # dst<31:cnt>     two operand
        dst<31:0> ← (-src<31>)<cnt-1:0> # src<31:cnt>     three operand
        dst<atag> ← 0
        dst<atag'> ← 0 †
        pc ← label
```

CONDITION CODES unchanged

MNEMONICS

asr.l.j	arithmetic shift right longword, two operand and jump
asr3.l.j	arithmetic shift right longword, three operand and jump
asr.q.j	arithmetic shift right quadword, two operand and jump
asr3.q.j	arithmetic shift right quadword, three operand and jump

OPERANDS The P register can **not** be specified as destination operand.

OPCODES

M.asrw	asr.l.j, asr3.l.j
M.asrq	asr.q.j, asr3.q.j

NOTE If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

† A 64-bit destination operand includes two address tags.

BITSET — set bit

FORMAT

```
bitset    <bitpos>, <dst>
bitset.j  <bitpos>, <dst>, <label>
bitsetr   <bitpos>, <src>, <dst>
```

OPERATION

```
dst ← src                    ; bitsetr
dst<bitpos> ← 1
pc ← <label>    ; bitset.j
```

CONDITION CODES cc ← dst

MNEMONICS

bitset	destination is memory
bitset.j	set bit and jump
bitsetr	destination is GV register

OPERANDS

	<bitpos>	<src>	<dst>
bitset	imm/s1		s3(s2)
bitset.j	imm/s1		s3(s2)
bitsetr	imm/s1	s2	s3

The bit position maybe specified by GV register or immediate. The destination memory operand cannot take any side-effect.

OPCODES

K.bitset	
I.bitsetr	

NOTE

BITCLR — clear bit

FORMAT

```
bitclr   <bitpos>, <dst>
bitclr.j <bitpos>, <dst>, <label>
bitclrr  <bitpos>, <src>, <dst>
```

OPERATION

```
dst ← src                ; bitclrr
dst<bitpos> ← 0
pc ← <label>    ; bitclr.j
```

CONDITION CODES

cc ← dst

MNEMONICS

bitclr	destination is memory
bitclr.j	clear bit and jump
bitclrr	destination is GV register

OPERANDS

	<bitpos>	<src>	<dst>
bitclr	imm/s1		s3(s2)
bitclr.j	imm/s1		s3(s2)
bitclrr	imm/s1	s2	s3

The bit position maybe specified by GV register or immediate. The destination memory operand cannot take any side-effect.

OPCODES

K.bitclr
I.bitclrr

NOTE

BITCHG — change bit

FORMAT

```
bitchg   <bitpos>, <dst>
bitchg.j <bitpos>, <dst>, <label>
bitchgr  <bitpos>, <src>, <dst>
```

OPERATION

```
dst ← src                ; bitchgr
dst<bitpos> ← ¬ dst<bitpos>
pc ← <label>             ; bitchg.j
```

CONDITION CODES

cc ← dst

MNEMONICS

bitchg	destination is memory
bitchg.j	change bit and jump
bitchgr	destination is GV register

OPERANDS

	<bitpos>	<src>	<dst>
bitchg	imm/s1		s3(s2)
bitchg.j	imm/s1		s3(s2)
bitchgr	imm/s1	s2	s3

The bit position maybe specified by GV register or immediate. The destination memory operand cannot take any side-effect.

OPCODES

K.bitchg
I.bitchgr

NOTE

BITTST — test bit

FORMAT

```
bittst   <bitpos>, <dst>
bittst.j <bitpos>, <dst>, <label>
bittstr  <bitpos>, <src>, <dst>
```

OPERATION

```
dst ← src                    ; bittstr
Z ← dst<bitpos>
pc ← <label>    ; bittst.j
```

CONDITION CODES

Z ← dst<bitpos> (others unchanged)

MNEMONICS

bittst	destination is memory
bittst.j	test bit and jump
bittstr	destination is GV register

OPERANDS

	<bitpos>	<src>	<dst>
bittst	imm/s1		s3(s2)
bittst.j	imm/s1		s3(s2)
bittstr	imm/s1	s2	s3

The bit position maybe specified by GV register or immediate. The destination memory operand cannot take any side-effect.

OPCODES

K.bittst
I.bittstr

NOTE

BITCNT — count bits

FORMAT

```
bitcnt   <src>, <dst>
bitcnt.j <src>, <dst>, <label>
bitcntr  <src>, <dst>
```

OPERATION

```
dst ← 0
for i = 0 to 31 do i = i +1
        if src<i> → dst ← dst + 1
pc ← <label>    ; bitcnt.j
```

CONDITION CODES

cc ← dst

MNEMONICS

bitcnt	source is memory
bitcnt.j	count bits and jump
bitcntr	source is GV register

OPERANDS

	<src>	<dst>
bitcnt	s3(s1)	t
bitcnt.j	s3(s1)	t
bitcntr	s2	s3

The destination memory operand cannot take any side-effect.

OPCODES

K.bitcnt
I.bitcntr

NOTE

BITFND — find bit

FORMAT

```
bitfnd   <src>, <dst>
bitfnd.j <src>, <dst>, <label>
bitfndr  <src>, <dst>
```

OPERATION

dst ← bit position first non-zero bit

CONDITION CODES

Z ← not found (others unchanged)

MNEMONICS

```
bitfnd   source is memory
bitfnd.j find bit and jump
bitfndr  source is GV register
```

OPERANDS

	<src>	<dst>
bitfnd	s3(s1)	t
bitfnd.j	s3(s1)	t
bitfndr	s2	s3

The destination memory operand cannot take any side-effect.

OPCODES

K.bitfnd
I.bitfndr

NOTE

BITREV — reverse bits

FORMAT

```
bitrev <src>, <dst>
```

OPERATION

dst<i> ← src<31 - i> pc ← <label> ; bitrev.j

CONDITION CODES

cc ← dst

MNEMONICS

```
bitrev reverse bits
```

OPERANDS

	<src>	<dst>
bitrev	s3(s1)	t

OPCODES

M.bitrev

NOTE

LEA — load effective address

FORMAT lea <src>,<dst>

OPERATION
dst ← effective address (src)
dst<atag> ← valid address ? 1 : 0

CONDITION CODES unknown

MNEMONIC lea load effective address

OPERANDS The source operand is a memory address. The special addressing modes *index offset* and *pointer index* can be used. The destination operand is a GV register.

OPCODES
I.lea load effective address
K.mkea lead effective address (*index offset, pointer index*)

NOTE

MKBL — make new BL-pair

FORMAT
mkbl <src1>,<src2>,<dst>
mkbl.j <src1>,<src2>,<dst>,<label>

OPERATION
GV[dst] ← effective address (src1)
GV[dst]<atag> ← 1
GV[dst^1] ← effective address (src2)
GV[dst^1]<atag> ← 1
pc ← <label> ; mkbl.j

CONDITION CODES unknown

MNEMONICS mkbl make new BL-pair

OPERANDS Source1 and source2 both specify an address. No side-effect is allowed. The extended addressing mode *index offset*(XO) can be used. The destination operand is a GV register pair (even/odd register). If <dst> is even, the <dst>,<dst+1> pair is used; if <dst> is odd, the <dst>,<dst-1> pair is used.

OPCODES
K.lbl
K.mkbl create new BL-pair

NOTE

ALLOC — allocate space

FORMAT alloc <base>,<src1>,<src2>,#<offset>,<dst>

OPERATION
M[GV[base]] ← GV[src1]
M[GV[base]+4] ← GV[src2]
GV[dst] ← GV[base]
GV[base] ← GV[base] + #offset

CONDITION CODES unchanged

MNEMONICS
alloc allocate space

OPCODES
KL.alloc allocate space

NOTE

CMP — arithmetic compare

FORMAT cmp.<src data type> <src1>,<src2>

OPERATION cc ← <src1> - <src2>

CONDITION CODES cc ← <src1> - <src2>

MNEMONICS
cmp.l compare longwords
cmp.q compare quadwords
cmp.f compare single floating operands
cmp.d compare double floating operands

OPERANDS

OPCODES
M.cmpw cmp.l
M.cmpq cmp.q
M.cmpf cmp.f
M.cmpd cmp.d

NOTE

CMPJ — arithmetic compare and jump

FORMAT

cmp.<src data type>.j <src1>,<src2>,<label>

OPERATION

if exception → pc ← next instruction
if ¬ exception →
    cc ← src1 - src2
    pc ← label

CONDITION CODES

cc ← src1 - src2

MNEMONICS

cmp.l.j	compare longwords and jump
cmp.q.j	compare quadwords and jump
cmp.f.j	compare single floating operands and jump
cmp.d.j	compare double floating operands and jump

OPERANDS

OPCODES

M.cmpw	cmp.l.j
M.cmpq	cmp.q.j
M.cmpf	cmp.f.j
M.cmpd	cmp.d.j

NOTE

If | pc - <label> | ≥ 128, then an extra cycle is added to the execution time.

BRS — branch on signed integer operation

FORMAT

brs <greater>,<equal>,<less>
b<cc> <label>

OPERATION

if (¬ V & ¬ Z & N) | (V & ¬ Z & ¬ N) → pc ← less
if Z → pc ← equal
if (¬ V & ¬ Z & ¬ N) | (V & ¬ Z & N) → pc ← greater

CONDITION CODES

unchanged.

MNEMONICS

brs	three way branch on condition
blt	branch on less
ble	branch on less equal
beq	branch on equal
bne	branch on not equal
bgt	branch on greater
bge	branch on greater equal

OPCODES

J.bcc	branch address
J.jcc	jump address

NOTE

The overflow condition is ignored. If the overflow is important, use brv.

BRU — branch on unsigned integer operation

FORMAT

```
bru     <greater>,<equal>,<less>
bu<cc>  <label>
```

OPERATION

if (¬ C & ¬ Z & N) | (C & ¬ Z & ¬ N) → pc ← less
if Z → pc ← equal
if (¬ C & ¬ Z & ¬ N) | (C & ¬ Z & N) → pc ← greater

CONDITION CODES unchanged.

MNEMONICS

bru	three way branch on condition
bult	branch on less
bule	branch on less equal
bueq	branch on equal
bune	branch on not equal
bugt	branch on greater
buge	branch on greater equal

OPCODES

J.bcc	branch address
J.jcc	jump address

NOTE The carry is ignored; if the carry is important, use brc.

BRV — branch on signed integer operation and overflow

FORMAT

```
brv     <greater>,<equal>,<less>,<overflow>
b<cc>   <label>
```

OPERATION

if (¬ V & ¬ Z & N) → pc ← less
if Z → pc ← equal
if (¬ V & ¬ Z & ¬ N) → pc ← greater
if V → pc ← overflow

CONDITION CODES unchanged.

MNEMONICS

brv	three way branch on condition
bvs	branch on overflow set
bvc	branch on overflow clear

OPCODES

J.bcc	branch address
J.jcc	jump address

NOTE

BRC — **branch on unsigned integer operation and carry**

FORMAT	brc <label1>, <label2>, <label3>, <label4> b<cc> <label>

OPERATION	if (C & N) → pc ← <label1>
	if (C & ¬ N) → pc ← <label2>
	if (¬ C & N) → pc ← <label3>
	if (¬ C & ¬ N) → pc ← <label4>

CONDITION CODES	unchanged.
MNEMONICS	brc four way branch on condition bcc branch on carry clear bcs branch on carry set
OPCODES	J.bcc branch address J.jcc branch address
NOTE	

JMP — **jump to label**

FORMAT	jmp <label>
OPERATION	pc ← label
CONDITION CODES	unchanged.
MNEMONICS	jmp
OPCODES	J.jump
NOTE	

CMPBR — compare and branch on signed/unsigned integer operation

FORMAT

```
cmp.brs     <src1>,<src2>,<greater>,<equal>,<less>
cmp.bru     <src1>,<src2>,<greater>,<equal>,<less>
cmp.b<cc>   <src1>,<src2>,<label>
```

OPERATION

src2 - src1
if src1 < src2 → pc ← less
if src1 = src2 → pc ← equal
if src1 > src2 → pc ← greater

CONDITION CODES

unchanged.

MNEMONICS

cmp.brs	compare and three way branch on signed condition
cmp.bru	compare and three way branch on unsigned condition
cmp.blt	compare and branch on signed less
cmp.ble	compare and branch on signed less equal
cmp.beq	compare and branch on signed equal
cmp.bne	compare and branch on signed not equal
cmp.bgt	compare and branch on signed greater
cmp.bge	compare and branch on signed greater equal
cmp.bult	compare and branch on unsigned less
cmp.bule	compare and branch on unsigned less equal
cmp.bueq	compare and branch on unsigned equal
cmp.bune	compare and branch on unsigned not equal
cmp.bugt	compare and branch on unsigned greater
cmp.buge	compare and branch on unsigned greater equal

OPCODES

J.cb	branch address
J.cj	jump address

NOTE

ACBR — add, compare and branch on signed/unsigned integer operation

FORMAT

```
ac.brs      <chk>,<src1>,<src2>,<greater>,<equal>,<less>
ac.bru      <chk>,<src1>,<src2>,<greater>,<equal>,<less>
ac.b<cc>    <chk>,<src1>,<src2>,<label>
ac.bra      <chk>,<src1>,<src2>,<label>
```

OPERATION

src2 ← src1 + src2; next
if chk < src2 → pc ← less
if chk = src2 → pc ← equal
if chk > src2 → pc ← greater

CONDITION CODES

cc ← src1 + src2

MNEMONICS

ac.brs	add, compare and three way branch on signed condition
ac.bru	add, compare and three way branch on unsigned condition
ac.blt	add, compare and branch on signed less
ac.ble	add, compare and branch on signed less equal
ac.beq	add, compare and branch on signed equal
ac.bne	add, compare and branch on signed not equal
ac.bgt	add, compare and branch on signed greater
ac.bge	add, compare and branch on signed greater equal
ac.bult	add, compare and branch on unsigned less
ac.bule	add, compare and branch on unsigned less equal
ac.bueq	add, compare and branch on unsigned equal
ac.bune	add, compare and branch on unsigned not equal
ac.bugt	add, compare and branch on unsigned greater
ac.buge	add, compare and branch on unsigned greater equal
ac.bra	add, (compare) and branch always

OPCODES

J.acb	branch address
J.acj	jump address

NOTE

SCBR — subtract, compare and branch on signed/unsigned integer operation

FORMAT

```
sc.brs   <chk>,<src1>,<src2>,<greater>,<equal>,<less>
sc.bru   <chk>,<src1>,<src2>,<greater>,<equal>,<less>
sc.b<cc> <chk>,<src1>,<src2>,<label>
sc.bra   <chk>,<src1>,<src2>,<label>
```

OPERATION

src2 ← src2 - src1; next
if chk < src2 → pc ← less
if chk = src2 → pc ← equal
if chk > src2 → pc ← greater

CONDITION CODES

cc ← src2 - src1

MNEMONICS

sc.brs	subtract, compare and three way branch on signed condition
sc.bru	subtract, compare and three way branch on unsigned condition
sc.blt	subtract, compare and branch on signed less
sc.ble	subtract, compare and branch on signed less equal
sc.beq	subtract, compare and branch on signed equal
sc.bne	subtract, compare and branch on signed not equal
sc.bgt	subtract, compare and branch on signed greater
sc.bge	subtract, compare and branch on signed greater equal
sc.bult	subtract, compare and branch on unsigned less
sc.bule	subtract, compare and branch on unsigned less equal
sc.bueq	subtract, compare and branch on unsigned equal
sc.bune	subtract, compare and branch on unsigned not equal
sc.bugt	subtract, compare and branch on unsigned greater
sc.buge	subtract, compare and branch on unsigned greater equal
sc.bra	subtract, (compare) and branch always

OPCODES

J.scb	branch address
J.scj	jump address

NOTE

CMPABR — compare two addresses and branch

FORMAT

```
cmpa.bcc  <src1>,<src2>,<label>
cmpa.br <src1>,<src2>,<equal label>,<not equal label>
```

OPERATION

if src1 = src2 → pc ← <equal label>
if src1 <> src2 → pc ← <not equal label>

CONDITION CODES

unchanged

MNEMONICS

cmpa.br	compare two addresses and branch
cmpa.beq	compare two addresses and branch if equal
cmpa.bne	compare two addresses and branch if not equal

OPERANDS

The operands are either a GV register or an immediate.

OPCODES

Leqa	compare two addresses

NOTE

BRA — branch on address tag

FORMAT bra <src>,<has-no-tag-label>,<has-tag-label>

OPERATION
if src<atag> → pc ← has-tag-label
if ~src<atag> → pc ← has-no-tag-label

CONDITION CODES unchanged

MNEMONICS bra branch on address tag

OPERANDS

OPCODES
J.adrb
J.adrj

NOTE

CMPBL — compare BL and branch on B/in/L/out

FORMAT
cmp.bl <bl>,<src>,<base-label>,<in-label>,<limit-label>,<out-label>
cmp.bl<c> <bl>,<src>,<label>

OPERATION
if <src> >= <bl>.base and <src> <= <bl>.limit → pc ← in-label
if <src> < <bl>.base or <src> > <bl>.limit → pc ← out-label
if <src> == <bl>.base → pc ← base-label
if <src> == <bl>.limit → pc ← limit-label

CONDITION CODES unknown

MNEMONICS

cmp.blr	compare src against BL-pair and branch four way
cmp.bli	compare src against BL-pair and branch on *in*
cmp.blo	compare src against BL-pair and branch on *out*
cmp.bll	compare src against BL-pair and branch equal *limit*
cmp.blb	compare src against BL-pair and branch equal *base*
cmp.blur	compare unsigned against BL-pair and branch four way
cmp.blui	compare unsigned against BL-pair and branch on *in*
cmp.bluo	compare unsigned against BL-pair and branch on *out*
cmp.blul	compare unsigned against BL-pair and branch equal *limit*
cmp.blub	compare unsigned against BL-pair and branch equal *base*

OPERANDS Both BL-register pair and source register are GV-registers.

OPCODES
J.cb
J.cj

NOTE

ACBL — add, compare BL and branch on B/in/L/out

FORMAT

ac.bl <bl>,<src>,<dst>,<base-label>,<in-label>,<limit-label>,<out-label>
ac.bl<c> <bl>,<src>,<dst>,<label>

OPERATION

<dst> ← <dst> + <src>; next
if <dst> >= <bl>.base && <dst> <= <bl>.limit → pc ← in-label
if <dst> < <bl>.base || or <dst> > <bl>.limit → pc ← out-label
if <dst> == <bl>.base → pc ← base-label
if <dst> == <bl>.limit → pc ← limit-label

CONDITION CODES

unknown

MNEMONICS

ac.blr	add, compare against BL-pair and branch four way
ac.bli	add, compare against BL-pair and branch on *in*
ac.blo	add, compare against BL-pair and branch on *out*
ac.bll	add, compare against BL-pair and branch equal *limit*
ac.blb	add, compare against BL-pair and branch equal *base*
ac.blur	add, compare unsigned against BL-pair and branch four way
ac.blui	add, compare unsigned against BL-pair and branch on *in*
ac.bluo	add, compare unsigned against BL-pair and branch on *out*
ac.blul	add, compare unsigned against BL-pair and branch equal *limit*
ac.blub	add, compare unsigned against BL-pair and branch equal *base*

OPERANDS

The BL-register pair, source and destination register are all GV-registers.

OPCODES

J.acb
J.acj

NOTE

SCBL — subtract, compare BL and branch on B/in/L/out

FORMAT

ac.bl <bl>,<src>,<dst>,<base-label>,<in-label>,<limit-label>,<out-label>
ac.bl<c> <bl>,<src>,<dst>,<label>

OPERATION

<dst> ← <dst> - <src>; next
if <dst> >= <bl>.base and <dst> <= <bl>.limit → pc ← in-label
if <dst> < <bl>.base or <dst> > <bl>.limit → pc ← out-label
if <dst> == <bl>.base → pc ← base-label
if <dst> == <bl>.limit → pc ← limit-label

CONDITION CODES

unknown

MNEMONICS

sc.blr	subtract, compare against BL-pair and branch four way
sc.bli	subtract, compare against BL-pair and branch on *in*
sc.blo	subtract, compare against BL-pair and branch on *out*
sc.bll	subtract, compare against BL-pair and branch equal *limit*
sc.blb	subtract, compare against BL-pair and branch equal *base*
sc.blur	subtract, compare unsigned against BL-pair and branch four way
sc.blui	subtract, compare unsigned against BL-pair and branch on *in*
sc.bluo	subtract, compare unsigned against BL-pair and branch on *out*
sc.blul	subtract, compare unsigned against BL-pair and branch equal *limit*
sc.blub	subtract, compare unsigned against BL-pair and branch equal *base*

OPERANDS

The BL-register pair, source and destination register are all GV-registers.

OPCODES

J.scb
J.scj

NOTE

CALL — call subroutine

FORMAT	call <src>
OPERATION	(cfp+1)->vps0 ← pc (cfp+1)->vps1<29:6> ← cfp (cfp+1)->vps1<5:0> ← pisw<5:0> cfp ← cfp+1 pc ← src
MODE	user
CONDITION CODES	unchanged
MNEMONICS	call call subroutine
OPERANDS	Immediate or register
OPCODES	I.call
NOTE	

RET — return from subroutine

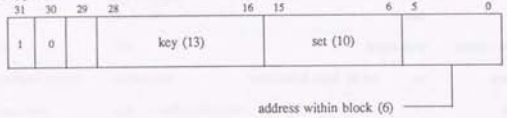
FORMAT	ret
OPERATION	pc ← cfp->vps0 pisw<5:0> ← cfp->vps1<5:0> cfp ← cfp->vps1<29:6>
MODE	user
CONDITION CODES	unchanged
MNEMONICS	ret return from subroutine
OPERANDS	
OPCODES	I.ret
NOTE	

1.6 Memory management

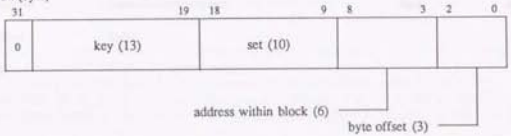
1.6.1 Memory management resources

Virtual memory address

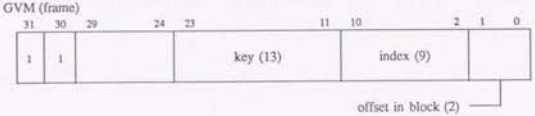
(1) IM (quad)



(2) DM (byte)

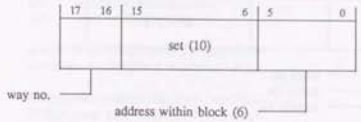


(3) GVM (frame)

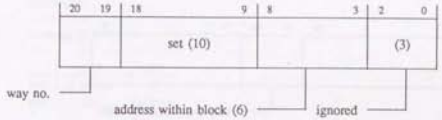


Physical address

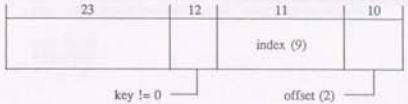
(1) IM



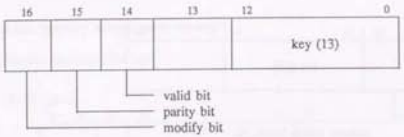
(2) DM



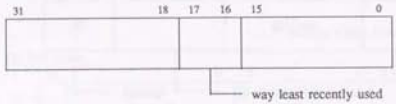
(3) GVM



Key



Lru



1.6.2 Memory management instructions

PLD — **load from data memory using physical address**

FORMAT

pld <physical address>,<dst>
pst <src>,<physical address>

OPERATION

dst<31:0> ← DM[physical address]<31:0>
dst<atag> ← DM[physical address]<atag>

DM[physical address]<31:0> ← src<31:0>
DM[physical address]<atag> ← src<atag>

MODE

privileged

CONDITION CODES

unchanged

OPERANDS

	<src>	<dst>
pld	s1+imm	s3
pst	s1	s2+imm

MNEMONICS

pld load contents of DM into register using physical address
pst store source into DM memory using physical address

OPCODES

I.pld (I.pld)
I.pst (I.pst)

NOTE

PLDIM **load from instruction memory using physical address**

FORMAT
pldim <physical address>,<dst>
pstim <src>,<physical address>

OPERATION
dst<63:0> ← IM[physical address<17:0>]
IM[physical address<17:0>] ← src<63:0>

MODE
privileged

CONDITION CODES
unchanged

MNEMONICS
pldim load contents of IM into register using physical address
pstim store source into IM memory using physical address

OPERANDS

	<src>	<dst>
pldim	s1	s3
pstim	s1	s2

OPCODES
I.pldim (I.pldim)
I.pstim (I.pldim)

NOTE

LDIM **load from instruction memory using virtual address**

FORMAT
ldim <virtual address>,<dst>
stim <src>,<virtual address>

OPERATION
if ¬virtual address<atag> ∨ virtual address<31:30> ¬ 10 ⇒ trap *ill_op*
if virtual address<atag> ∧ virtual address<31:30> = 10 ⇒
dst<63:0> ← IM[virtual address<29:3>]
IM[virtual address<29:3>] ← src<63:0>

MODE
privileged

CONDITION CODES
unchanged

MNEMONICS
ldim load contents of IM into register using virtual address
stim store source into IM memory using virtual address

OPERANDS

	<src>	<dst>
ldim	s1	s3
stim	s2	s1

OPCODES
I.ldim (I.ldim)
I.stim (I.stim)

NOTE

PSTCFP — store physical address in cfp

FORMAT pstcfp <src>

MODE privileged

OPERATION cfp ← src<29:6>

CONDITION CODES unchanged

MNEMONICS pstcfp store physical address in *cfp*

OPERANDS

	<src>
pstcfp	s1

OPCODES I.pstcfp (I.pstcfp)

NOTE

LDKEY — load contents of DM key memory into register

FORMAT
ldkey <physical address>,<dst>
ldimkey <physical address>,<dst>
ldgvkey <dst>

OPERATION
dst ← DMKEY[physical address.set]
dst ← IMKEY[physical address.set]
dst ← GVKEY[cfp.index]

MODE privileged

CONDITION CODES unchanged

OPERANDS

	<src>	<dst>
ldkey	s1+imm	s3
ldimkey	s1	s3
ldgvkey		s3†

MNEMONICS
ldkey load DM key into register
ldimkey load IM key into register
ldgvkey load GV key into register

OPCODES
I.ldkey (I.ldkey)
I.ldimkey (I.pldim)
I.ldgvkey (I.frrr)

NOTE

† The destination operand of 'ldgvkey' must be a global GV register.

STKEY — store key into DM key memory

FORMAT

stkey <src>,<physical address>
stimkey <src>,<physical address>
stgvkey

OPERATION

DMKEY[physical address.set].key ← src.key
DMKEY[physical address.set].valid ← src.valid
DMKEY[physical address.set].modify ← src.modify
IMKEY[physical address.set].key ← src.key
IMKEY[physical address.set].valid ← src.valid
IMKEY[physical address.set].modify ← src.modify
GVKEY[cfp.index].key ← cfp.key
GVKEY[cfp.index].modify ← *false*
GVKEY[cfp.index].valid ← *true*

MODE

privileged

CONDITION CODES

unchanged

OPERANDS

	<src>	<dst>
stkey	s3	s1+imm
stimkey	s1	s2
stgvkey		

MNEMONICS

stkey store key into DM key memory
stimkey store key into IM key memory
stgvkey store key into GV key memory

OPCODES

I.stkey (I.stkey)
I.stimkey (I.stimkey)
I.stgvkey (I.frrr)

NOTE

RMKEY — remove key from DM key memory

FORMAT

rmkey <src>,<physical address>
rmimkey <src>,<physical address>
rmgvkey

OPERATION

DMKEY[physical address<17:6>].key ← src.key
DMKEY[physical address<17:6>].valid ← src.valid
IMKEY[physical address<17:6>].key ← src.key
IMKEY[physical address<17:6>].valid ← src.valid
GVKEY[cfp.set].key ← cfp.key
GVKEY[cfp.set].valid ← *false*

MODE

privileged

CONDITION CODES

unchanged

OPERANDS

	<src>	<dst>
rmkey	s3	s1+imm
rmimkey	s1	s2
rmgvkey		

MNEMONICS

rmkey invalidate key in DM key memory
rmimkey invalidate key in IM key memory
rmgvkey invalidate key in GV key memory

OPCODES

I.rmkey (I.stkey)
I.rmimkey (I.stimkey)
I.rmgvkey (I.frrr)

NOTE

LDLRU — **load LRU of data memory using virtual address**

FORMAT

ldlru <virtual address>,<dst>
stlru <src>,<virtual address>

OPERATION

dst<31:0> ← LRU[virtual address<31:0>]
LRU[virtual address<31:0>] ← src<31:0>

MODE

privileged

CONDITION CODES

unchanged

OPERANDS

	<src>	<dst>
ldlru	s1+imm	s3
stlru	s3	s1+imm

MNEMONICS

ldlru load DM LRU into register using virtual address
stlru store DM LRU using virtual address

OPCODES

I.ldlru (I.ldkey)
I.stlru (I.stkey)

NOTE

The address tag of the virtual address is not checked.

LDIMLRU — **load LRU of instruction memory using virtual address**

FORMAT

ldimlru <virtual address>,<dst>
stimlru <src>,<virtual address>

OPERATION

dst<31:0> ← IMLRU[virtual address<31:0>]
IMLRU[virtual address<31:0>] ← src

MODE

privileged

CONDITION CODES

unchanged

OPERANDS

	<src>	<dst>
ldimlru	s1+imm	s3
stimlru	s3	s1+imm

MNEMONICS

ldimlru load IM LRU into register using virtual address
stimlru store IM LRU using virtual address

OPCODES

I.ldimlru (I.ldkey)
I.ldimlru (I.stkey)

NOTE

The address tag of the virtual address is not checked.

1.7 Exception handling

Trap identifier
Currently, 26 of 32 possible trap identifiers have been assigned.

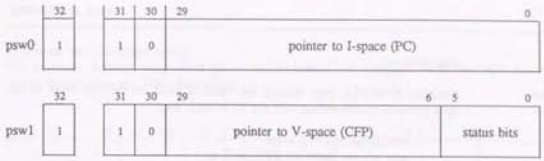
```
RESET       0   /* occurs on cold boot */
HARDWARE    1   /* can be enhanced */
PU_ARITH    4   /* arithmetic error in Multiplier */
SM_ARITH    5   /* arithmetic error in Rounding */
GV_ERROR    7   /* memory error (GV unit) */
ILLOPR      8   /* illegal operand (can be enhanced) */
LISP_SVC    9   /* lisp service request */
GV_ARITH    10  /* arithmetic error in GV unit */
ADRCAL      11  /* outside BL range */
ADRBND      12  /* alignment error */
DM_ERROR    13  /* DM: memory error */
DM_MLTHIT   14  /* DM: multiway hit */
PF_DM       15  /* DM: page fault */
SU_ARITH    17  /* arithmetic error in ALU */
TRACE       18  /* generated when trace bit on */
SVP_NMI     19  /* generated when SVP writes #1 into iodev address 128 */
SLF_NMI     20  /* generated by 'ipnmi' instruction (both receive NMI) */
IM_ERROR    22  /* IM: memory error */
IM_MLTHIT   23  /* IM: multi-way hit */
PF_IM       24  /* IM: page fault */
ILLINST     25  /* can be enhanced */
PRVVIO      26
PF_GV       27  /* gv memory page fault */
SYSCALL     29  /* generated by 'trap' instruction */
SLF_INT     30  /* generated by 'ipint' instruction (both receive INT) */
SVP_INT     31  /* generated when SVP writes #1 into iodev address 129 */
```

The trap identifier can be obtained from the *processor internal status word* (pisw).

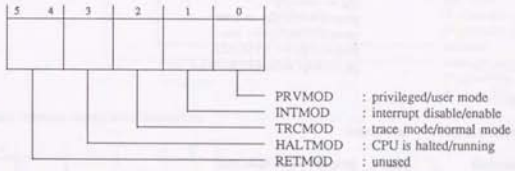
Trap vectors
The trap vector table is a jump table starting from IM address 0 (0x80000000) for machine 0 and IM address 64 (0x80000040) for machine 1 and is indexed by the trap identifier.

Cold boot
On cold boot, a RESET trap occurs.

Program status word
The program status word stored when a exception occurs is identical to the program status word saved on a subroutine call. The location is fixed, however.



Program status word status bits



Trap stack frame
When a trap occurs, the program counter, current frame pointer and status bits (program status word) are saved in a local frame, indexed by the trap identifier. Machine 0 uses the frames 0 to 31 and machine 1 uses the frames 32 to 63.
Two effective addresses are saved in two consecutive words in DM memory again indexed by the trap identifier. Machine 0 users the locations 0 to 63 in DM and machine 1 uses the locations 64 to 127.

TRAP trap

FORMAT trap <src>

OPERATION Generate SYSCALL trap, writing the value of <src> in the high word of the data memory corresponding with the SYSCALL trap.

```
DM[SYSCALL*2] ← src
new_cfp ← frame for SYSCALL trap
new_cfp->vps0 ← pc
new_cfp->vps1<29:6><- cfp
new_cfp->vps1<5:0><- pisw<5:0>
cfp ← new_cfp
pisw<PRVMOD> ← true
pisw<INTMOD> ← true
pisw<12:8> ← SYSCALL
pc ← trap vector[SYSCALL]
```

MODE user

CONDITION CODES unchanged

MNEMONICS trap generate SYSCALL trap

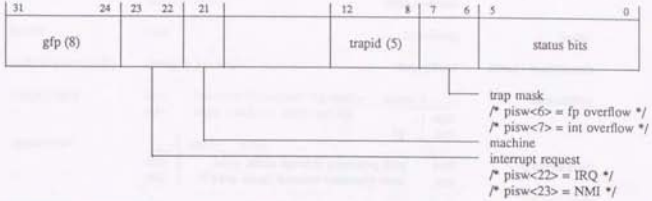
OPERANDS Currently, only an immediate operand is implemented.

OPCODES I.trap

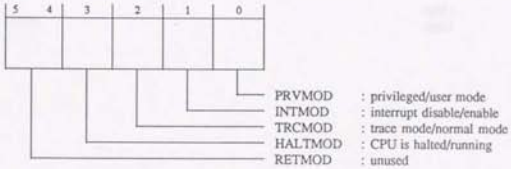
NOTE
Use the 'rei' instruction to return from an trap.

1.8 Processor status

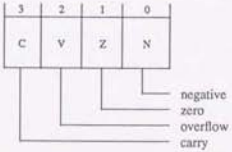
Processor internal status word
The processor internal status word (pisw) can be loaded into a register using the 'ldps' instruction. It can be stored using the 'stps' instruction.



Processor internal status word status bits



Condition code register
The 'ldcc' and 'stcc' instructions operate upon the condition code register. The condition code register is not saved on subroutine entry and exception.



LDPS — load processor internal status word

FORMAT

ldps <dst>
stps <src>

OPERATION

psiw → <dst>
<src> → psiw

MODE

privileged

CONDITION CODES

unchanged

OPERANDS

	<src>	<dst>
ldps		s3
stps	s1	

MNEMONICS

ldps load processor internal status word
stps store processor internal status word

OPERANDS

Source and destination operand are GV-registers.

OPCODES

I.ldps
I.stps

NOTE

LDCC — load condition code register

FORMAT

ldcc <dst>
stcc <src>

OPERATION

ccr → <dst>
<src> → ccr

MODE

user

CONDITION CODES

changed by stcc

MNEMONICS

ldcc load condition code register
stcc store condition code register

OPERANDS

	<src>	<dst>
ldcc		s3
stcc	s1	

OPCODES

I.ldcc
I.stcc

NOTE

LDCFP — load cfp into register

FORMAT

```
ldcfp   <dst>
stcfp   <src>
```

MODE user

OPERATION

dst ← 11 # cfp<29:6> # 0<5:0>

if (¬src<atag>) ∧ (src<31:30> ≠ 11) → trap
if src<atag> ∧ (src<31:30> = 11) → cfp ← src<29:6>

CONDITION CODES unknown

OPERANDS

	<src>	<dst>
ldcfp		s3
stcfp	s1	

MNEMONICS

ldcfp	load *cfp* into register
stcfp	store address in *cfp*

OPCODES

Ldcfp
Lstcfp

NOTE

IPINT — inter-processor interrupt request

FORMAT

```
ipint
ipnmi
```

OPERATION

```
ipint:
if ¬other.pisw<INTMOD>
        other.exception SLF_INT

ipnmi:
other.exception SLF_NMI
pisw.nmi ← true
```

MODE privileged

CONDITION CODES unchanged

MNEMONICS

ipint	interrupt request on other processor
ipnmi	non-maskable interrupt request on other processor

OPCODES

Lipint
Lipnmi

NOTE

LAS — load and store for mutual exclusion

FORMAT las <src register>, <memory>, <dst register>

OPERATION
<dst register> ← <memory>;
<memory> ← <src register>

MODE user

CONDITION CODES unchanged

OPERANDS

	<src>	<memory>	<dst>
las	s1	s3(s2)	t

MNEMONICS las load and store for mutual exclusion

OPCODES K.las

NOTE

SETA — set address tag

FORMAT seta <src>,<dst>

OPERATION
if src.b ≤ ea(src) ≤ src.l && ea<atag> ⇒
dst ← ea(src)
dst<atag> ← 1
if src.b > ea(src) ‖ ea(src) > src.l ‖ ¬ ea<atag> ⇒
dst ← ea(src)
dst<atag> ← 0

USER user

CONDITION CODES unchanged

MNEMONICS seta set address tag

OPERANDS The source operand must be a 'memory address' (*pointer mode* notation). The destination operand must be a GV-register.

OPCODES I.seta set address tag

NOTE The address tag of the source is not checked. If the effective address is not a legal address, the address is moved to the destination register with its address tag cleared.

KSETA — set address tag

FORMAT	kseta <src>,<dst>
OPERATION	dst ← effective address(src) dst<atag> ← 1
USER	privileged
CONDITION CODES	unchanged
MNEMONICS	kseta set address tag
OPERANDS	The source operand can be a register or an immediate. The destination operand must be a GV-register.
OPCODES	I.kseta set address tag
NOTE	The address tag of the source is not checked. If the effective address is not a legal address, the address is moved to the destination register with its address tag cleared.

GETIDV — get device memory

FORMAT	getidv <src>, <dst> putidv <src>
OPERATION	dst ← DEVICE_MEMORY[src<39:32>] ; getidv DEVICE_MEMORY[src<39:32>] ← src<15:0> ; putidv
MODE	priviledged
CONDITION CODES	unchanged
MNEMONICS	getidv get device memory putidv put device memory
OPERANDS	The source operand is 64-bit immediate, memory, or SP register.
OPCODES	M.getidv M.putidv
NOTE	When the SVP writes into location DEVICE_MEMORY[128], a SVP_NMI interrupt is generated. When the SVP writes into location DEVICE_MEMORY[129], a SVP_INT interrupt is generated.

INDEX

C

CONTENTS

LIST OF TABLES

Appendix C

Logic Design and Simulation Tools for FLATS2

(*in Japanese*)

FLATS2の設計支援環境

市川　周一

新技術事業団
後藤磁束量子情報プロジェクト
構成法グループ

1990年8月

要旨

本稿では、数値・記号処理用計算機FLATS2の設計支援環境およびシミュレーション環境について報告する。FLATS2の論理記述およびシミュレーションには、核としてハードウェア記述言語DDLを採用した。DDLコンパイラ以外の開発環境については、今回全てを独自に開発した。DDLは、数式処理計算機FLATSの開発に用いられた環境(PLS)の一部である。PLSもそれ自体で完結した優れたシステムではあるが、バッチ処理指向であるため、試行錯誤を要する開発には不向きである。そのため今回の設計/シミュレーションには、DDLを記述言語として採用しながらも、新たに「対話性」を重視した実行環境を作成することにした。これがDID (DDL Interactive Debugger)である。DIDでは、DDLソース・レベルの記号(信号名)を使用して、対話的にデバッグ作業を行なうことができる。このため従来のPLSと比較して、より開発指向のシステムになっている。また、DDLのソース・レベルでは従来のPLSシステムと互換性があるため、検証段階では従来システムによるバッチ処理を行なうこともできる。

1　はじめに

FLATS2は、理化学研究所、新技術事業団、三井造船(株)、三井造船システム技研(株)の4者が共同開発した数値・記号処理用計算機である。そのアーキテクチャと設計目標については既に報告済みであるので[1, 2, 3]、本稿ではFLATS2の論理設計およびシミュレーション環境について報告することとする。

FLATS2 は 10 万ゲート級の計算機であるため、その論理設計にあたっては充分な検討と検証が行なわれなくてはならない。そのため、計算機を使用した設計支援とシミュレーションが必須である。実際の製作にあたっては、三井造船(株)が市販の CAD システム (Daisy) を使用して、実装レベルのシミュレーションを行なうことになった。

しかし実装レベルの検証が行なわれる以前に、アーキテクチャの検討と論理レベルでの検証が必要であることは言うまでもない。当初、論理レベルの検証を省いて実装レベルの検証に統一しようという努力がなされたが、CAD の容量・速度、設計手順などの点で行き詰まった。そこで論理レベルのプロトタイプを別に作り、アーキテクチャと論理の設計開発・改良を行なうことにした。

検討の結果、数式処理計算機 FLATS [4, 5] の開発に使用した環境 PLS (Portable Logic Simulation System)[6] を復元することになった。全く独自の開発環境を作成することは開発行程の面で不可能であり、市販のシステムを導入することは予算面および実績の点で問題があった。通常のプログラム言語 (C, FORTRAN など) を使用すれば、記述性や可搬性の面で不安が残る。PLS ならば理化学研究所、東京大学、三井造船(株)で使用された実績がある。また復元の結果、理化学研究所と三井造船(株)で再び使用可能になった。従って、PLS は共同開発の環境として最低限の条件は備えていた。

ところが PLS は、ゲート・レベルのハードウェア記述から実ハードウェアの保守にまで対応する汎用システムである。そのため今回の使用目的にも対応しうるものの、最適な環境とも言い難い。実際 PLS はそれ自体で完結した優れたシステムではあるが、ファイルを介して多種のユーティリティが結合したバッチ指向のシステムである。今回の目的についても、バッチ的処理しか期待できない。これは、PLS が PLS 開発当時の計算機使用環境に合わせて設計されているため当然であるともいえる。

FLATS のように開発から検証・保守までを一つの環境で行なうなら、PLS を使用するのが適当であると考えられる。しかし今回行なうのは論理レベルのプロトタイピングだけである。この場合には、作業が試行錯誤をともなう性質のものであるため、対話的で修正時のターン・アラウンド・タイムが短い環境こそが望ましいといえる。

そこで FLATS2 の論理記述およびシミュレーションには、PLS の中からハードウェア記述言語 DDL だけを取り出して採用した。DDL には現実に FLATS の設計に使用された実績があり、汎用のハードウェア記述言語として FLATS2 の設計にも適合すると考えたからである。そして、それ以外の開発環境については今回全てを独自

に開発することとし、特に「対話性」を重視した実行環境を作成することにした。これが DID (DDL Interactive Debugger) である。

DID では、DDL ソース・レベルの記号 (信号名) を使用して、対話的にシンボリック・デバッグ作業を行なうことができる。このため従来の PLS と比較して、より開発指向のシステムになっている。ユーザは任意のメモリ・イメージを使用して、内部状態や信号線の状態を適宜確認しながらクロックを進め、対話的デバッグ作業を行なうことができる。入力の変更や内部状態の変更、シミュレーションの再試行は容易であるから、ユーザは試行と推論を繰り返しながら論理設計やマイクロコードのバグを発見し修正することができる。

また、DDL のソース・レベルでは PLS システムと互換性があるため、検証段階では従来システムを使用したバッチ処理を行なうこともできる。これによって、従来のソフトウェア資産を有効に活用することができる。DID は PLS の弱点を補完するシステムであって、PLS とは並行使用すべきシステムである。

2 FLATS2 設計支援環境の目的と方針

ここで、今回開発した設計支援環境の目的をまとめておく。

- **立ち上げ期間を短くすること。**

  実装設計・製作に結果を反映させるため、プロトタイピングを迅速に行なう必要がある。

- **共同開発のグループ内で互換性を保つこと。**

- **ハードウェアの論理設計および論理シミュレーションが可能であること。**

- **実装設計とその検証は別の CAD で行なう。**

- **マイクロコードの開発・修正作業が、マイクロ・アーキテクチャの開発と密に結合して、かつ容易に行えること。**

- **シンボリックで対話的な環境を持つこと。**

  その効果は、ソフトウェア開発に於けるデバッガの重要性からみて明らかである。

これらの目的をふまえて、以下の決定を行なった。

- PLS 全体を使用するのではなく、DDL 言語のソース・レベルで互換性を保つ。

  このために、DDL コンパイラ (ddlc) の移植を行なう。これによってコンパイラを新規開発する手間が省ける。移植の労力も小さくはないが、新技術事業団にもなんらかの支援環境が必要であったため、不可避の作業ではあった。

  また、ddlc を移植することにより、ソースコード・レベルで PLS との互換性が保証される。これにより、共同開発者内で役割に応じて環境を使い分けることが可能になる。

- UNIX システム上で実現する。

  全ての共同研究者が、UNIX マシンを持っていた。最近では計算機の性能が向上したため、UNIX ワークステーションでも計算能力の問題は少ない (FLATS 設計時に用いた H8700 や VAX-11/780 より高速かつ大容量になっている)。

  UNIX 上には C/FORTRAN77/ デバッガ / 画面エディタ等の開発環境が充分に揃っているため、新たに必要となるものはない。使用者もこの環境に慣れているため、新たな環境を導入するより立ち上がりが早くなり、保守作業も自分で行なうことが可能になる。

  また、UNIX は対話指向で小規模プロトタイプ作成に適した環境である (という定評がある)。これはまさに、今回の目的に適合している。

- ツール群は自分で作成し、自分で保守する。

  PLS システムの作成者[1]は FLATS2 プロジェクトのメンバーでないため、いずれにせよ保守は自分で行なわねばならない。ならば今回の目的によく適合したツールを自分で作ってしまえば、後の保守作業も容易になる。

- より低水準なハードウェア・レベル・シミュレータや、より高度なアーキテクチャ・レベル・シミュレータに発展する可能性も考慮しておく。

これらの決定をもとに、FLATS2 の論理設計支援環境を構築した。次章以下で、その実現について具体的に述べる。

[1]樹水謙多郎氏。東京大学理学部情報科学科 (現 電機通信大学)。

3 FLATS2 設計支援環境の概要

図 1 に、現在の設計支援環境のあらましを示す。

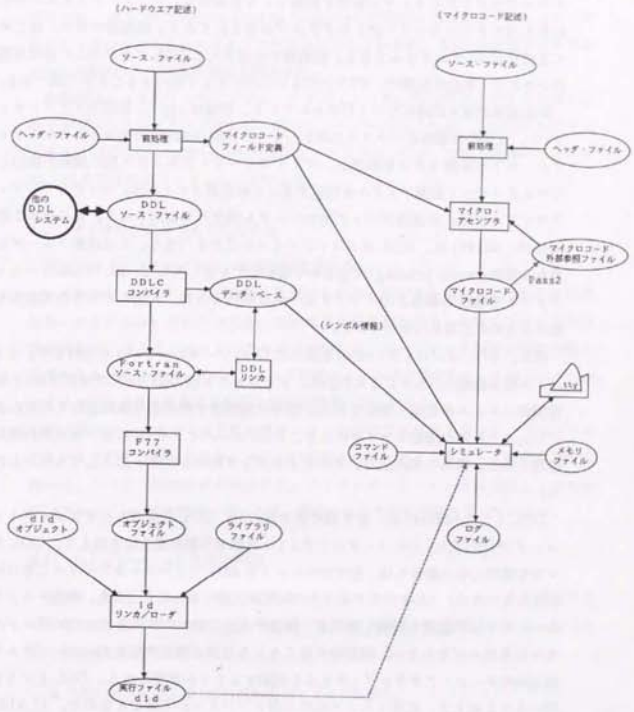


図 1 FLATS2 の設計支援環境

3.1 ハードウェア記述の処理手順

まず、ハードウェア記述とその処理手順を簡単に説明する。

ハードウェアの記述は、基本的にはDDL言語を用いて行なう。しかし、DIDシステムではプリプロセッサの使用を前提としているので、ソース・ファイルの中には多くのプリプロセッサ・ディレクティブが含まれており、前処理の後に、はじめて正式なDDLプログラムとなる。前処理で生成されるDDLプログラムの書式は既存システムと完全に互換で、PLS上でもコンパイルして実行することが可能である。

前処理は現在4段階に分けて行なわれている。初段はcpp (C言語用プリプロセッサ)で、文字列の置換等のマクロ処理を行なう。次段(ddlmt)ではマイクロコード・フィールドの定義マクロを処理し、マイクロコード・アセンブラ用の情報を抽出してマイクロコード定義ファイルを作成する。この定義ファイルは、マイクロコード・アセンブルの他、対話的デバッグ時のシンボル情報にも使用される。その後の2段(ddlct, ddlft)は、DDLソース・ファイルの成形を行なう。この成形フェーズは単なる清書(pretty printing)であるので省略しても差し支えないが、DDLソース・ファイルを他所の環境とのインタフェースに用いるため、人間にとっての可読性も高めるために成形を行なっている。

次に、DDLコンパイラ(ddlc)を用いてDDLソースのコンパイルを行なう。コンパイルは各論理ユニットごとに行われ、オブジェクト(FORTRANプログラム)と各種情報ファイルが生成・更新される。信号の属性表や外部信号参照表はそれぞれオブジェクトと別の情報ファイルになり、DDLコンパイラやDDLリンカの実行時に参照される。信号の属性表は、そのままデバッグ時のシンボル表としても用いられる。

DDLリンカ(ddlnk)は、信号属性表を用いて、DDLの生成したオブジェクト・コード(FORTRANソース・プログラム)と外部信号参照表を書き換える。DDLリンカを使用しない場合には、全てのユニットをddlcで2パス・コンパイルしなければならないため、ddlcでのコンパイル時間が大変に長くなる。また、あるユニットのハードウェア要素を変更したとき、関連する全てのユニットをまとめて再コンパイルしなければならない。開発時にはこうした状況が頻発するため、ハードウェア修正時のターン・アラウンド・タイムを短縮することが重要になる。DDLリンカを用いることにより、当該ユニット以外の再コンパイルが不要になるので、FLATS2シミュレータの生成時間が大きく短縮されて、設計変更のターン・アラウンド・タイムも大きく短縮される。このことは、システム全体の有用性・使用感にも大きく貢献する。

こうして全ユニットのコンパイルとリンクが終了したら、生成されたFORTRANプログラムをFORTRANコンパイラ(BSD UNIXのf77)でコンパイルする。

DDL記述上の各論理ユニットはコンパイルされるとSUBROUTINEになるため、実行イメージを作るためには主プログラム(main)とリンクする必要がある。PLSシステムでは、SCL (Simulation Control Language)で記述したシミュレーション・プログラムがコンパイルされて主プログラムになっていたが、DIDでは主プログラム自体に会話的コマンド解釈機能を持たせた。

各論理ユニットのリロケータブル・オブジェクトを、実行時ライブラリと共にこの主オブジェクトにリンクすれば、実行イメージが生成される。この主プログラム部をdid (DDL Interactive Debugger)主プログラムと呼ぶ。完成した実行イメージが、FLATS2のdidシミュレータである。

3.2 マイクロプログラムの処理手順

次に、マイクロプログラムの処理手順を述べる。

マイクロコード・アセンブラは、汎用のものが入手できなかったので独自に作成した。マイクロコードについても、cppによる前処理を行なった後にアセンブル作業を行なう。マイクロコードのアセンブル時には、ハードウェア記述の前処理時に生成されたマイクロコード・フィールド定義ファイルが必要である。このファイルが、マイクロプログラムにおける変数の型定義情報を含んでいる。

FLATS2のマイクロコードにはアドレスの間接参照が存在するため、アセンブルを2パスで行なう必要がある。パス1で生成された、マイクロコード外部参照表を用いて、パス2で外部参照を解決する。マイクロコード・リンカは作成していないが、アセンブルの所要時間が短いため、現在のところ特に問題になっていない。

3.3 シミュレーションの実行手順

最後に、didシミュレータの実行について述べる。didシミュレータは実行ファイルなので、UNIXのコマンド・レベルで起動すればよい。起動されると、以下の順で初期化を行なう。

- DDLのシンボル表(信号属性表ファイル)を読む。
- マイクロコード・シンボルの表(マイクロコード・フィールド定義ファイル)を読む。
- マイクロコードを読み込んで、制御記憶を初期化する。

- メモリ・イメージを読んで、命令記憶・データ記憶・レジスタに値を書き込む。
- ハードウェア(各論理ユニットに対応するサブルーチン)を初期化する。
- シミュレータの初期化ファイルを読んで、実行する。

  初期化が終了すると対話モードに入り、プロンプトを出してユーザの入力(コマンド)を待つ。もちろん、入出力をファイルに対して行なうことも可能である。コマンド等については、後で述べる。

ユーザはdidのコマンド・レベルで各種のコマンドを使用して、対話的な論理シミュレーションを行なうことができる。この方法は、誤りの原因を究明するときに特に有効である。実際、didのインタフェースはdbx(UNIX上のソース・コード・デバッガ)に似せて作られている。使用感覚も、dbxによるソフトウェアのデバッグと似ている(dbxの方がずっと高級な機能を持つが)。

全体にわたって、UNIXのツールを活用したために開発の手間を大きく省略することができた。

4 FLATS2 設計支援環境の実現

前章で述べた設計支援環境のインプリメンテーションについて、個別にもう少し具体的に紹介する。ddlcの移植は佐藤三久[2]と市川が、その他全てのユーティリティは市川が設計、開発を行なった。

4.1 ddlc (DDL コンパイラ)

オリジナルは、東京大学の清水謙多郎氏[3]によって開発された。市川と佐藤は、そのVAX-VMS版をBSD UNIXのFORTRAN77上に移植した。移植作業のほとんどは、佐藤によって行なわれた。

内部仕様書がなかったため、移植作業は手探りの状態で行なわれた。VMS上のddlcはFORTRANで記述されており、ソースコードで約5500行(147KB)であった。これをUNIXに移植したところ、6600行(FORTRAN) + 700行(C)程度になった。増えているのは原型をコメントとして残しているせいもあるが、新たに書き足した部分も多い(Cの700行など)。

[2]新技術事業団 後藤磁束量子情報プロジェクト 構成法グループ

[3]現 電機通信大学

FORTRANプログラムであるから、一応の移植作業はすぐ終了した。しかし移植にともなうバグの除去には、その後も多くの労力が費やされた。発生した問題点のうち主なものを以下に列挙する。

- FORTRANの仕様の相違

  まず、論理演算子(.AND.等)のとりうるオペランド・タイプが異なっていた。また、式の各項の評価順序がVMSとUNIXで逆だったため、副作用のある式で副作用の順序が狂ってしまった。

- 入出力の仕様の相違

  これは、どちらかといえばOSの仕様の相違であった。また、UNIX上のツールと適合させるため、積極的な改良も行なった。

- FORTRAN実行時ライブラリの相違

  ddlc内部の表を書き換えるだけでは対応できなかったので、大幅な修正が行なわれた。

  また、ddlcの実行時ルーチンもUNIX用に新たに作成された。当初はFORTRAN77で記述されていたが、結局C言語で書き直して軽量化した。現在Cで150行程度の、小さい関数の集合である。

4.2 ddlnk (DDL リンカ)

DDLでは、外部信号の参照は、ユニット名と信号タイプに応じて定まったCOMMONブロックを介して行なわれる。あるユニットの記述(DDLソース)を修正すると、新しい信号の付加・信号属性の変更等によって、信号の属するCOMMONブロック名やブロック内オフセットが変わってしまうことがある。その場合、関連する全てのユニットに影響が及ぶ。

ddlcを実行すると、オブジェクトが生成されると同時に、信号属性表が更新される。その属性表には、各信号のタイプやブロック内オフセットの情報が含まれている。また、各ユニットは自分が参照している外部信号名とその内部名などを外部信号参照表として公開しているので、それらを用いて各ユニットの外部参照を正しく修正することができる。

ddlnkは、信号属性表を参照しながら、ユニットのオブジェクトと外部信号参照表を正しく書き直す。プログラムはC言語で書かれており、700行程度のものである。

信号属性表、外部参照表、DDL オブジェクトなどの外部仕様書がなかったため、ddlc のソースや実験結果を元に、独自に作成した。

ddlc のコンパイル・オプションによっても、2 パス・コンパイルによるリンクを行なうことができるが、実行時間は 1 パス・コンパイルの 3 倍近くになる。ddlnk の実行時間は、コンパイル時間と比べれば無視できる程度 (1 パス・コンパイルの数分の一) である。

4.3 muasm (マイクロコード・アセンブラ)

プリプロセッサ (ddlmt) が作成するマイクロコード・フィールド定義ファイルを使って、マイクロコードの各フィールドに指定された値を埋め込み、マイクロコードのオブジェクト・ファイルを作成する。マイクロコードの記法は単純なものだが、シンボルも右辺値に指定することが可能であり、cpp 等の併用によって定数の置換も可能であるので、実用上充分な記述性・可読性が得られる。

他にも、大域値 (default) の設定、アセンブル番地指定、コメント挿入等、通常必要と思われる機能は組み込まれている。C 言語で 540 行程度の小さなユーティリティである。

4.4 did (did 主プログラム)

前述の通り、コンパイルされた各論理ユニットを統合し、ユーザにシンボリックで対話的デバッグ環境を提供する。基本的なコマンドには、以下のようなものがある。

- 信号線の状態表示 (print, dump)

  DDL シンボル名を評価して、現在の値を表示する。オプションにより、奇数の変更も可能である。正規表現を使用して信号名を指定することも可能である。

- 信号線の状態設定 (set)

  信号に値を設定する。代入操作。

- メモリのロード (load)、ダンプ (/)

  メモリは、print や set コマンドで裸のハードウェアとしてアクセスすることもできるが、より抽象的に、アーキテクチャ上のメモリとしてもアクセスできる。マイクロコードや命令、データの再ロードにも用いる。

- 信号線のトレース (trace, delete)

  各クロック実行終了時に、指定された信号 (表現) を表示する。

- 信号線の定義照会 (what, whatis)

  信号名を指定して、その型定義情報を表示する。正規表現も使用できる。

- クロック (clock)

  ハードウェア・ユニットにクロック信号を与えて、論理動作を実行して内部状態を進める。

- 論理ユニットの接続 (attach)、切離し (detach)

  指定したユニットへのクロック供給を開始・終了する。切り離されたユニットの内部状態は、保存される。ハードウェアの部分的デバッグに使用する。

- UNIX とのインタフェース (edit, sh, ...)

  did 内部から、UNIX の機能を呼び出す。エディタで DDL ソースを修正、再コンパイルしたり、マイクロコードの変更・作成、命令・データの編集を行なうことができる。

- 簡単なマクロ展開 (alias)

  コマンドの別名定義、略称登録が可能。

- コマンド・ファイルの読み込みと実行 (source)

  テスト手順をファイルに作っておくと、シミュレーションの再試行が極めて簡単になる。

- 内部状態の退避と復元 (save, restore)

  各論理ユニットの内部状態をファイルに退避したり、ファイルから復元したりする。

- 入出力ファイル / プロセスのオープンとクローズ (open, close)

  save, restore で用いるファイルの指定。FLATS2 の入出力をシミュレートする外部プロセスの実行開始と終了。

必要に応じて did のコマンドを増やすことは容易である。現在 C で 2800 行程度のプログラムになっている。

4.5 プリプロセッサ等 (ddlct, ddlmt, ddlft)

3つ合わせても500行(C言語)に満たないが、このような小さなユーティリティを柔軟に結合することも、迅速なプロトタイプ作成に貢献する。この他、UNIXのツール(cpp、エディタ等)、ライブラリ(文字列操作、正規表現パッケージ、入出力)を最大限に活用することで開発労力を低減し、早期に実用化することができた。

5 FLATS2のシミュレーション

FLATS2のシミュレーション作業の実際について、簡単に触れておく。以下に記す「実行時間」は、SUN-3/60を平均的負荷の下に使用している場合のものである。

5.1 ハードウェア記述

現在ソースとヘッダを合わせて約10000行余りである。必要に応じて、より実装レベルに近い記述に変えて正確なシミュレーションを行なうこともできる。現状では、全てのユニットをコンパイルして実行イメージを生成するには約12分を要する[4]が、1ユニットのみを変更する場合は、DDLリンカの使用によってターン・アラウンド・タイムが著しく短縮される。論理記述は全て市川によって行なわれた。

5.2 マイクロコード記述

現在ソースとヘッダを合わせて23000行余りである。実機で使用しているマイクロ・コードは、didで使用しているものを機械的に変換して、ハードウェアにダウンロードしたものである。マシン全体では、幅400ビット余り(深さ1Kビット)になる。マイクロコードは全て市川が記述・開発した。

全体で21の制御記憶(ファイル)に分かれているが、これら全てを2パスでアセンブルしても4分程度で終了する。ddlmtとマイクロコード定義ファイルによってハードウェア記述と機械的に連動しているため、保守の手間は大きく低減されている。

5.3 シミュレーション作業

現在didシミュレータの実行ファイルは320KB、実行中のイメージは10MB弱である。立ち上げ(コマンド投入からプロンプト出力まで)に10秒程度を要するが、コマンドの実行時には遅延を感じない。全ユニットを接続したシミュレーションにおいても、1クロックの実行は0.6秒程度である(トレース出力を行なわない場合)。対話的デバッグ環境としては、充分使用に耐える反応時間であると思われる。

[4] ddlcに5分、ddlnkに1分、f77に6分。

6 おわりに

FLATS2の開発支援環境作成については、既にPLSが使用可能であったため、PLSが不得意とする部分を補完する形をとった。また、機能を拡張するよりも保守と開発の労力を低減し、ターン・アラウンド・タイムや反応時間の短いコンパクトなシステムを、短い期間で立ち上げることに主眼を置いた。その目的の大部分は果たしたと思われる。このシステムは、FLATS2の設計と並行して開発されたが、開発開始から6カ月後には既に実用的に使用されていた。さらに、FLATS2の開発期間(3年)全体を通して使用され、初期には論理とマイクロコードの開発・デバグに、その後は実装レベル・シミュレータとの比較に使用された。

しかし、DIDという設計支援環境を独立したシステムとして評価するならば、まだまだ非力で未熟なシステムと言える。そこで、以下に今後の課題として補強が望ましい点を列挙しておく。

6.1 did コマンドの強化

プログラム機能、特に式の評価とプログラム制御(if/for/while等)を付けることが望ましい。同じコマンドを繰り返し使用することも多いので、コマンドの履歴(history)およびその編集機能もあるとよい。

UNIXには優れたユーザ・インタフェースを提供するユーティリティ/ライブラリが数多く存在するので、それらを利用したい。特に、X-Windowなどのマルチ・ウィンドウ環境を使用することが考えられる。

6.2 効率の向上

現状でも一応実用的な性能を得ているが、より一層のチューン・アップが望まれる。現在、問題と知りながら放置している点の一つに、DDLプログラムのコンパイル時の最適化がある。ddlcは移植して動かしただけなので、なんら最適化の試みは行なっていない。原理的にはFORTRANコンパイラに最適化を任せることも可能かも知れないが、実際にはコンパイル時間がかかりすぎて実用にならない。論理記述上の意味的な情報は、FORTRANプログラムになるとほとんど失われてしまうので、ddlcレベルで最適化することが望ましい。

また、現状ではddlcのコンパイル作業が遅くて、論理変更時のターン・アラウンド・タイム短縮のネックになっているが、これもddlcの改良によって解決できるかも知れない。ddlcを全て、yacc/lex,[5] C言語などで書き直すことも考えられる。

6.3 役割分担

ここで、FLATS2の開発支援環境という観点に立ち戻って考えてみる。このシステムやPLS以外に、どの様なシステムが必要であろうか。少なくとも、次の2点は解決されなければならない。

- ハードウェアのオン・ライン・デバッグと保守の環境
- バッチ処理による論理/実装設計の検証

FLATS2の実装設計は、三井造船(株)によって、PLSによるバッチ処理と市販の論理CAD(Daisy)を用いて行なわれた。ハードウェアのデバグ・保守については、FLATS2に組み込まれたscan I/Oを用いてプロセッサの内部状態を読みだし、PLSによるシミュレーション結果と照合するという方法によって行なわれた。さらに、上記の方法で発見されたバグの原因を調査するために、対話的に内部状態やメモリ内容を読み出したり、クロックを与えたりするプログラムが開発された。このプログラム(SCAN)については、三井造船(株)の資料を参照して頂きたい。

[5] UNIXに附属するコンパイラ作成支援ユーティリティ

参考文献

[1] 市川・後藤："FLATS2のアーキテクチャと論理設計"、理研シンポジウム「第4回ジョセフソン・エレクトロニクス」予稿集、pp.1-9、和光市、1987.

[2] Ichikawa: "A Study on a Cyclic Pipeline Computer: FLATS2"、修士論文、東京大学理学部情報科学科、1987.

[3] 後藤・市川・他："FLATS2のアーキテクチャ"、情報処理学会第35回全国大会(1987)、6C-4.

[4] Goto, Soma, et al:"FLATS: A Machine for Symbolic and Algebraic Manipulation", The Second Int'l Symp. on Symbolic and Algebraic Computation by Computers, Inada and Soma Ed., World Scientific.

[5] 平木："数式処理計算機の研究"、博士論文、東京大学理学部情報科学科、1986.

[6] Shimizu:"A Portable Logic Simulation System", RIMS Symposia on Software Science and Engineering, Lecture Notes in Computer Science, Springer-Verlag (to be published). (Also in Shimizu:"High-Performance Computer Architecture", Dr.Dis., Univ. of Tokyo, 1984.)

NAME

ddlc − DDL compiler

SYNOPSYS

ddlc SystemName filename linkopt listopt checkopt logopt

DESCRIPTION

ddlcは、DDL言語のソース・プログラムをFORTRANのプログラムに変換する。

清水謙多郎氏(東京大学)によるVAX/VMS版を、佐藤三久(新技術開発事業団)がSun3/160(UNIX)上に移植。市川周一(新技)が、デバッグおよびVAX/ULTRIXに移植。

OPTIONS

linkopt

十進数で、0〜7の値を指定する。指定できる機能は、ファシリティ・テーブルの一時出力(F)、ファシリティ・テーブルのマージ(M)、ファシリティ・テーブルの入力(E)、リンカの起動(L)の4つである。ただし現状では、Lは意味を持たない。

0	----
1	F---
2	FM--
3	FM-L
4	--E-
5	F-E-
6	FME-
7	FMEL

listopt 0か1を指定する。1ならば、行番号を付加したソース・リストおよび、名前表を出力する。

checkopt

0か1を指定する。1ならば、ハードウェアに対するR/Wアクセスのチェックを行なう。

logopt 実装設計オプション。0か1を指定する。1ならば、 LOGOUTファイルを生成する(現在使用不能)。

FILES

通常、DDLソース・ファイルのファイル名拡張子には、「.txt」を用いる。

DDLプログラムの実行には、実行時ライブラリが必要である。現在のライブラリは「ddllib.o」である。

SEE ALSO

"DDL• システム・コマンド VAX/VMS"、清水謙多郎

"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第5回ジョセフソン・エレクトロニクス』

NAME

ddlct − cuts the extra headers of DDL source out

SYNOPSYS

ddlct StartPattern filename...

DESCRIPTION

ddlctはファイルを修正して、StartPattrenで指定されるパターンに一致する行までを削除する。これにより、前処理に起因する無意味なヘッダを除去することができる。

FILES

PR*zzzzzz*

ワーク・ファイル。カレント・ディレクトリにとられる。 *zzzzzz*は十進6桁で表わしたプロセス番号である。正常終了すれば、renameされてfilenameになる。

SEE ALSO

ddlft(1),
"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第5回ジョセフソン・エレクトロニクス』

BUGS

一行の文字数が255を越えると、中断して異常終了する。

DIAGNOSTICS

pattern not found

StartPatternに一致する行が発見できなかった。ファイルは変更されない。

NAME

ddlft − change the case of DDL source text

SYNOPSYS

ddlft [−l|u] filename...

DESCRIPTION

ddlftは、DDL言語のソース・ファイルを読み込んで小文字／大文字の変換を行い、対象ファイル自体を書き換える。コメントの内部は変更しない。

これによって

(1) 可読性を向上し、

(2) 大文字しか受け付けないDDLシステムへの移植を可能にする。

OPTIONS

−l ソースをすべて小文字に変換する。

−u ソースをすべて大文字に変換する。（デフォルト）

FILES

ddlft.tmp.*pid*

ワーク・ファイル。カレント・ディレクトリに生成され、正常終了時にはfilenameにrenameされる。

SEE ALSO

ddlct(1),
"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第5回ジョセフソン・エレクトロニクス』

NAME

ddlnk – DDL linker

SYNOPSYS

ddlnk [**-C**] [**-e** *FacFile*] UnitName...

DESCRIPTION

DDLオブジェクト・ファイルのリンカ。

システムの信号属性表(*FacFile*)を参照しながら、DDLコンパイラ *ddlc*(1) がパス1で生成したオブジェクトと外部信号参照表を書き換える。これによって、パス1で不明だった外部信号への参照が解決する。

OPTIONS

-C チェック・オプション。リンク時に外部信号の型との整合性を検査する。異常があれば通知して、中断する。

-e *FacFile*
システムの信号属性表のファイル名を指定する。現在デフォルトはないので、必ず指定すること！
(どこがオプションなんだ？)

UnitName
リンクする論理ユニット名。ファイル名ではないので、注意すること！

FILES

UnitName.f
DDLオブジェクト・ファイル。 *ddlc*(1) の生成した、FORTRANプログラム・ファイル。

UnitName.ert
論理ユニットUnitNameの外部信号参照表。ertはExternal Reference Tableの略。 *ddlc*(1) がパス1で生成する。

SRCddlnk*pid*
ワーク・ファイル。正常終了すれば、ソース・ファイルにrenameされる。

ERTddlnk*pid*
ワーク・ファイル。正常終了すれば、外部信号参照表ファイルにrenameされる。

SEE ALSO

ddlc(1),
"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第5回ジョセフソン・エレクトロニクス』

NAME

ddlmt – extract the microcode field definition

SYNOPSYS

ddlmt [**-o** *outfile*] [**-s**|**S**] [**-t**|**T**] filename...

DESCRIPTION

ddlmtは、ハードウェア記述ファイルを読み込んで、マイクロコード・フィールドの定義を処理するプリ・プロセッサである。

ddlmtは、

(1) ソース・ファイル内のマイクロコード定義をDDL言語の構文に合わせて書換え、ソースに埋め込む。
(s/S オプション参照)

(2) マイクロコード定義の情報を、別ファイルに書き出す。
(t/T オプション参照)

OPTIONS

-s ソース・ファイル保存モード。ソース・ファイルを書き換えない。(デフォルト)

-S ソース・ファイル修正モード。ソース・ファイルを書き換える。

-t マイクロコード定義ファイルを生成しない。

-T マイクロコード定義ファイルを生成する。(デフォルト)

FILES

filename.mtb
マイクロコード定義ファイル。ソース・ファイル名に拡張子がある場合は、その拡張子が *.mtb* に置き換わる。

ddlmt.tmp.*pid*
-Sオプション使用時に使用されるワーク・ファイル。カレント・ディレクトリに生成され、正常終了すればfilenameにrenameされる。

SEE ALSO

muasm(1),
"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第5回ジョセフソン・エレクトロニクス』

BUGS

Too Long Line
現在、一行のバッファが255文字なので、ソース・ファイルがこれを越えるような長い行を含むと、中断して異常終了する。

NAME

did − DDL Interactive Debugger

SYNOPSYS

did [−e *FACfile*] [−m *MTBfile*] [WCSfile...] [MEMfile...]

DESCRIPTION

did主プログラム。

コンパイルした各論理ユニットを統合し、 DDLの信号名とマイクロコード・フィールド名に基づいた、シンボリック・デバッグ環境を提供する。

OPTIONS

−e *FACfile*
シミュレートするシステムの、信号属性表ファイルを指定する。 *FACfile*は、FACility table fileの略。

−m *MTBfile*
シミュレートするシステムの、マイクロコード・フィールド定義ファイルを指定する。 *MTBfile*は、Microcode field definition TaBle fileの略。

WCSfile
たち上げ時にロードする、マイクロコードのオブジェクト・ファイル名。ファイル名は *muasm*(1) で生成されるとおり、*Unit_Wcs*.micでなくてはならない。

MEMfile
たち上げ時にロードする、メモリ・イメージのファイル名。ファイル名は、{dm|im|gvm}.mem でなければならない。

COMMANDS

まず、以下で用いる表現を簡潔に定義しておく。

信号 ::= *SymbolName*["[" *ix* [":" *iy*] "]"] ["<" *bx* [":" *by*] ">"]
*SymbolName*は、信号属性表またはマイクロコード・フィールド定義表に含まれるシンボル名。
"[", ":", "]", "<", ">" は、それぞれの文字自体を示す。
*ix*は、開始インデックスを示す右辺値。
*iy*は、終了インデックスを示す右辺値。
*bx*は、開始ビット位置を示す右辺値。
*by*は、終了ビット位置を示す右辺値。

疑似シンボルとして、DDL言語の使用するCOMMONデータ・ブロックを使用することもできる。COMMONブロックのシンボル名は、以下の形式をとる：
%<ユニット名><q|v|s|m>
末尾がqの疑似シンボルは、そのユニットの全レジスタを表わす。同様に、vはベクター(縦束)、sはスカラー(信号線)、 mはメモリを表わす。

右辺値 ::= 信号 | 定数値

定数値 ::= 基数 ' 数を表わす文字列 '

基数 ::= d | D | o | O | x | X | b | B | f | F
bまたはBは2進数、dまたはDは10進数、oまたはOは8進数、xまたはXは16進数、 fまたはFは浮動小数点数を指定するとき用いる。

以下に、didのコマンド・モードで使用できるコマンドをまとめる。

print [−基数] 信号
信号の値を評価し、指定された基数で表示する。基数が指定されない場合、デフォルトは１６進である。

display [−基数] 正規表現
正規表現に一致する信号名を捜し、評価して、値を指定された基数で表示する。基数のデフォルトは16進。正規表現の解釈と判定には *re_comp*(3), *re_exec*(3) を使用するので、記法についてはこれらのマニュアルを参照の事。

開始番地 , 終了番地/{g|v|d|i}[, 基数]
指定されたメモリの、開始番地から終了番地までを、指定された基数で表示する。番地は、右辺値で指定する。基数のデフォルトは16進。dはDM、iはIM、gやvはGVMを意味する。

開始番地/{g|v|d|i}[, 回数] [, 基数]
上に同じ。ただし、終了番地の代わりに回数 (表示するバイト数)を指定する。回数は、右辺値で指定する。

set 信号 = 右辺値
信号に値を代入する。

clock [クロック数]
システムのクロックを、指定されたクロック数だけ進める。クロック数が省略された場合、１であるとみなされる。クロック数に０を指定すると、接続中のユニットの内部状態が初期化される。

trace [−基数] [信号]
信号のトレースを宣言する。各クロック終了時に、指定された信号を、指定された基数で表示する。基数のデフォルトは16進。信号が指定されなかった場合は、現在トレースが指定されている信号を一覧にして表示する。

delete {信号 | 番号}
信号のトレースを解除する。traceの一覧に示される「番号」か、解除したい信号を指定する。

whatis 信号
信号の定義を表示する。信号が真のハードウェアでなく、別名による指定だった場合(例えばマイクロのフィールド)、その本体を表示する。

what 正規表現
whatisと性格同様であるが、**what**では、正規表現に一致するシンボルの定義を表示する。

attach [ユニット名]
指定されたユニットをシミュレーションに接続する。接続されると、クロックによって内部状態が変わるようになる。ユニットが指定されなかった場合、現在の接続状況が表示される。

detach ユニット名
指定したユニットを、シミュレーションから切り離す。切り離した後は、ユニットの内部状態が保存される。

alias [*alias*名 [*alias*表現]]
コマンドの別名を登録する。 *alias*表現を省略すると、*alias*名の別名が消去される。 alias名、alias表現ともに省略された場合、現在使用できる別名が一覧として表示される。

load ファイル名
メモリ・イメージをファイルからロードする。ファイル名の拡張子が「.mic」であった場合、マイクロコードのオブジェクト・ファイルとしてロードを行なう。拡張子が「.mem」であった場合、メモリ(imかdmかgvm)のイメージとしてロードする。ファイル名の形式については、OPTIONS項とFILES項を参照の事。

save [−*mode* ファイル名] 信号名
現在の内部状態を、バイナリ形式でファイルにダンプする。 *mode*は、ファイルをオープンする時のモードで、 fならば新規書き込み(w)、aならば追加書き込み(a)。 −*mode*でファイル名を指定しなかった場合、 openコマンドの−saveオプションで設定したファイルが使用される。

restore [−c クロック番号] [−f ファイル名]
saveコマンドで保存された、内部状態のバイナリ・ダンプを読み込んで、内部状態を再設定する。−cオプションでクロック番号を指定した場合、指定したクロック時の内部状態がリストアされる。クロック番号の指定が無い場合、最初に読みだしたレコードのクロック番号を指定したのと同じになる。 −fオプションでファイル名を指定しなかった場合、openコマンドの−restoreオプションで設定したファイルが使用される。

open [−save ファイル名] [−restore ファイル名] [−svp ファイル名]
−saveは、saveコマンドで使用する、デフォルト・ファイルを設定する。 −restoreは、restoreコマンドで使用する、デフォルト・ファイルを設定する。 −svpは、SVPをエミュレートするプロセスを指定する。したがって、ここでのファイル名はパイプを使用する。openされたファイルは、closeコマンドでクローズされるまで、 openされたまま残る。

close [−save] [−restore] [−svp]
openコマンドでオープンしたファイルを、クローズする。パイプ(プロセス)の場合は、SIGKILLでkillされる。

edit [ファイル名]
指定されたファイルをエディットする。起動されるエディタは、環境変数EDITORで指定できる。デフォルトは/usr/ucb/viである。ファイル名を指定しなければ、引数なしでエディタが立ち上がる。

source ファイル名
didコマンドをファイルから読み込んで実行する。

readfac ファイル名
信号属性表ファイルを読み込む。

readmtb ファイル名
マイクロコード・フィールド定義ファイルを読み込む。

help [項目]
(未実装)

sh [引数…]
シェルを起動する。起動されるシェルは、環境変数SHELLで指定できる。デフォルトは/bin/shである。指定した引数は、そのままシェルに引き渡される。

![引数…]
shと全く同じ。ただ、！と引数の間に句切り文字がなくてもよい。

#[引数…]
コメント。何もしない。つまり、この行を無視する。

quit
didを終了する。

コマンド入力時の注意を２つ。

- 「/」と「!」はコマンド・シンタックス上の句切り文字なので、引数の一部として含むことができない。

これらを含む文字列を引数に指定したい場合、

(1) ＼(バックスラッシュ)を前につけて、その文字をエスケープするか、

(2) ' (シングルクォート)を文字列の前後につけて、文字列をエスケープする

必要がある。「 ' 」を使った場合、閉じ「 ' 」がなければ行末までエスケープされる。

- EOF文字(一般には ^D (コントロール+D)) を打つと、「quit」したのと同じことになる。

FILES

.didinit did初期化コマンド・ファイル。
didは、立ち上がるとまずホーム・ディレクトリ($HOME) を捜し、.didinitがあるとそのファイルに書かれたコマンドを実行する。 $HOMEが設定されていなかったり、$HOMEに.didinitがなかった時は、カレント・ディレクトリを捜す。

filename.mem
メモリ・イメージ・ファイル。
*filename*部は、最初に.dm, im, gvmのいずれか、次にピリオド、その後に任意の文字列、という形式の文字列。

filename.mic
マイクロコード・ファイル。
*filename*部は、最初にユニット名、次に下線(アンダースコア) 、その後にWCS名、という形式の文字列。

SEE ALSO

ddlc(l), ddlmt(l), ddlnk(l), muasm(l),
"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第５回ジョセフソン・エレクトロニクス』

BUGS

現在まったくシグナルはとっていないので、^Cや^\等でシグナルを送るとdidが終了してしまう。

NAME

muasm – microcode assembler

SYNOPSYS

muasm [**–f** *TabFile*] [**–e** *ExtFile*] [**–s**] filename...

DESCRIPTION

ddlmt(l) の生成したマイクロコード・フィールド定義ファイルを参照して、マイクロコードのアセンブルを行なう。

muasm(l) は、自分の出力するマイクロコード・ダンプを読み込んで、外部シンボル表として使用することができる。従って、特にマイクロコード・リンカといったものはない。 *muasm*(l) を使って、2パス・アセンブルすることで外部シンボルの参照を解決する。

OPTIONS

–f *TabFile*
フィールド定義ファイルを指定する。
(現在デフォルトはないので、必ず指定する必要がある)

–e *ExtFile*
パス2において、必要なら、外部シンボル参照表を指定する。 *ExtFile*には、パス1で生成されたマイクロコード・ダンプ・ファイルを使用すればよい。

–s サイレント・モード。警告を抑止する。
特に、パス1で外部シンボルが解決できないときに使用する。

FILES

Unit_Wcs.mic
マイクロコード・ダンプ・ファイル。
現在は、このファイルがオブジェクトを兼ねる。
*Unit*には、ソース・ファイルで$unitに代入した名前が、
*Wcs*には、ソース・ファイルで$wcsに代入した名前が、
使用される。

SEE ALSO

ddlmt(l),
"FLATS2の論理設計"、市川・佐藤・他、理研シンポジウム『第5回ジョセフソン・エレクトロニクス』