FVLIW processor architecture DRAFT

Sergey Dubovyk

February 4, 2017

1 Abstract

The convectional approach to parallel computing requires quit complicated hardware design as much part of parallelization is done while executing the program. Another view at providing instruction level parallelism (ILP) guarantees that optimizations are done on compilation stage, so that each microprocessor command contains instructions for all the execution units (EU) of the system. This allows using much simpler architecture of the EUs that leads to increasing performance of the system. Generally, such approach is called VLIW (Very Long Instruction Word). However, there are still a lot of problems in field of VLIW processors which caused the current situation for this architecture. Purpose of this work is to create a command-set and an architecture of the VLIW processor that will eliminate or reduce some of these problems. **(Actually, I don‘t know which of them yet. But that‘s a research, isn‘t it? :=))**

2 Introduction

2.1 Problem

Nowadays computing involves many parallel operations, as a large part of computations is possible to do simultaneously due to no data dependence in them. There are several main approaches to providing parallel execution:

- Vector processors which simultaneously operate multiple instructions and data streams
- Task parallelism which‘s main idea is several data independent tasks and each core of processing unit cares about a single task
- Instruction-level parallelism via pipelining
- Instruction-level parallelism that is achieved with hardware methods which detect which commands can be executed simultaneously
- Instruction-level parallelism with explicit parallelism (VLIW) – each command contains instructions for all execution units

VLIW allows to simplify hardware design significantly and design cheaper processors with more IPC(Instructions per Cycle) but has number of problems:

- Low density of instructions in the memory
- Increased memory bandwidth (due to large instructions)
- High requirements for compiler to create optimized code
- Some algorithms optimal instruction set depends on input data, which can‘t be predicted by the compiler.
- Problems with backward compatibility as code created for VLIW processor with N cores can‘t be run with less number of cores on other processor.
- Inefficient in “single-thread mode” when there is a meaningful instruction only for one EU in the command

3 Ideas behind the FVLIW

In order to solve such problems as:

- Low code density
- Lack of back-compatibility

it is possible to use kind of variable length instructions – each single instruction has a signal bit which indicates if the following one depends on the result of current one or instructions before current. This allows to join independent instructions when they are executed on processors with more EU that the target machine. That is variation of the EPIC bundles idea which was used for Intel Itanium architecture.

FVLIW also provides opportunity to use a whole instruction word for immediate value in the I-type operations, that might be useful.

4 FVLIW general architecture overview

4.1 Overview

A lot of decisions were made in respect to opportunity of parallel computations: more GP registers, some restrictions to operations (e.g. one can not simultaneously execute more than one comparison operation).

The FVLIW architecture assumes, that a processor has 32 32-bit integer registers and 32 32-bit floating-point registers **(at this time only integer registers and operations because of their simplicity)**.

4.2 Register file

4.2.1 General-Purpose Registers

Integer registers \$r0 – \$r23 are available for general-purpose usage by applications.

4.2.2 Predefined registers

- \$r25
- \$r27 – Exception Program Counter
- \$r28 – Memory Base Address
- \$r29 – Instruction Pointer
- \$r30 – used as a status register for ALU operations and other events (e.g. interrupts).
  - [31:27] Exception code.
  - [26] Overflow flag
  - [25] Z(ero) flag
  - [24] N(egative) flag
  - [23:0] reserved
- \$r31 – used as a stack pointer

4.3 Memory organization

The FVLIW architecture uses flat addressing because of its simplicity and ease of implementation. This allows to simplify MMU design.
As MMIO is used for FVLIW architecture, you can find a table with address space description below (Each address corresponds to one machine word (i.e. 32 bits)):

First address	Last address	Description
0x00000000	0x00100200	Start up configuration code (stored in external ROM)
0x00100201	0x00101200	First level cache (4096 words)
0x00101201	0x00102200	Interruptions/exceptions handler (e.g. syscall)
0x00102201	0x80102200	General external RAM (10^9 words)
0x80102201	0xFFEFFFFE	Reserved
0xFFEFFFFF	0xFFEFFFFF	32 GPIO pins(specified as separate address subspace for simplicity)
0xFFF00000	0xFFF00000	GPIO pins mode
0xFFF00001	0xFFFFFFFF	MMIO devices (10^6 addresses)

Table 1: FVLIW address space

4.4 Input-Output Organization

Nowadays, there are two most popular approaches to the I/O management: x86-like, where there are two address spaces: for memory and peripheral devices or Memory-Mapped Input-Output, in which both system memory and external devices share a single address space.

Briefly, we will review benefits of both ideas. The x86 style is as easy as possible: when programmer wants to write or read some external device he can simply set a flag $M/\overline{IO}$ to specify whether data should be written/acquired from memory or peripheral devices. In case of MMIO both memory and devices share the same address space. That requires more complex hardware within memory-management unit(MMU), but simplifies software access to peripheral devices and makes it more homogeneous.

In table below, there are examples of address spaces for both of this cases. For simplicity we took a 32-bit address width, but principle is the same for all others. Also, you can find example schematics of each solutions in the

Appendix A.

Base address	Max address	$M/\overline{IO}$ flag	Description
00000000	FFFFFFFF	0	RAM memory of available for CPU
00000000	FFFFFFFF	1	Address space for peripheral devices

Table 2: x86-like address spaces

First address	Last address	Description
00000000	F0000000	RAM address subspace available for read-write
F0000001	F0000400	GPIO addresses
F0000401	FFFFFFFF	Specific devices addresses

Table 3: Example of a memory-mapped input-output address space

So, main advantages and disadvantages of x86-like/MMIO are:

x86 Input-Output

"+" Advantages

- Extreme simplicity in implementation and programming
- Cheap in sense of transistors

"-" Disadvantages

- Requires specialized commands for communications with external devices
- Creates multiple entities for similar operations (read memory or external devices)

Memory-mapped IO

"+" Advantages

- Provides more homogeneous access to the memory and external
- Saves commands as access to external devices is done as to all other memory addresses.

"-" Disadvantages

- Requires more complex hardware solution.

So, for FVLIW architecture we will use MMIO as its features worth some more work as shown in paragraph above.

Another important thing is that FVLIW architecture supports 32 GPIO pins. They can be accessed with the following address: 0xFFEFFFFF, which is the first address of the MMIO block. Also, to set the Input/Output mode, user should specify the value of 0xFFF00000, where each bit is responsible for the corresponding bit‘s mode in the GPIO block.

4.5 Logical-Arithmetic Units

4.5.1 ALU

Each EU contains an ALU which provides such operations:

- 32-bit 2‘s complement addition
- 32-bit 2‘s complement subtraction
- 32-bit 2‘s complement division
- 32-bit 2‘s complement multiplication with result of 64 bits
- AND
- OR
- XOR
- NAND

Every of this operations will return result as well as give some additional information about it: Z(ero), N(egative) and more.

4.5.2 FPU

Will be implemented in further (0.2 alpha) version of the architecture.

4.6 Power up process

When the FVLIW processor gets power its program counter (PC) is set to 0x00000000 which is the address of the first command of external BIOS-like ROM. In the next approx. $1 * 10^6$ instructions the basic configuration of the hardware should be done as well as setting of the Base Address value (by default on start up it is 0x00000000).

If the 0x00000000 memory cell is empty, the PC is changed to the 0x00102201, which is the first address of accessible RAM. The Base address is also set to 0x00102201.

5 FVLIW instruction set

5.1 General instructions overview

Each instruction contains of a set of simple commands for each EU. In this version of the architecture we will take a look at the version with two EUs. That allows to design a legitimate VLIW architecture but without using two much hardware resources(e.g. without using very large register file which could satisfy the requirements of parallel computing on e.g. 4 or 8 EUs.). Instructions have a fixed length of 32 bits * number of execution units (i.e. 64 bits for current version).

There are instructions of the following structure available(divided by arguments source and destination of the execution result):

- Register-Register Instructions (RR-type) – both take arguments and store result into the register file.
- Register-Immediate Instructions (I-type) – one of the arguments is given as a value in the command and stores result in the register file.
- Immediate instruction (II-type) – has only two operands: target register and immediate value. A good example of such command is seti $r, {value} which assigns {value} to the $r register.
- Register-Memory Instructions (M-type) – either load data from memory or saves it to memory.
- Branching Instructions (B-type) – the body of the instruction can contain either only target address (in case of unconditional jumps or branching with precalculated conditions) or register names with data to compare and target address.

5.2 Instruction structure

5.2.1 RR-type

A structure of RR-type instructions is following:

- [31:24] 8 bits are used as instruction opcode
- [23:9] 15 bits are used for three register addresses (5 bits each, as there are 32 registers) (target, source 1, source 2)
- [8:4] instruction for ALU or FPU (used in arithmetic or comparison operations)
- [3:1] unused
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones.

5.2.2 Register-Immediate instructions

The other very similar type is Register-Immediate instructions which looks like the following:

- [31:24] 8 bits are used for opcode
- [23:14] 10 bits for addresses of two registers – one for destination and another is source for the result
- [13:2] 12 bits of the second operand‘s value
- [1] indicates if the next instruction is used for storing data for this one **(reserved)**
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones

5.2.3 Load/Store instructions

Instructions which work with memory are actually only store/load ones. They can have two variants: to work with all data and addresses stored in registers or they can take memory addresses as ”immediate” argument.

Firstly lets introduce all-registers case:

- [31:24] 8 bits are used for opcode

- [23:14] 2 register addresses are used to store the data and memory address
- [13:1] unused
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones.

And below the case with immediate value of memory address is shown:

- [31:24] 6 bits are used for opcode
- [23:19] 5 bits are used for a target/source register address
- [18:3] 16 bits allocated for a memory address, which after sign extension becomes a valid relative address (to the Base Address stored in $r28).
- [2:1] unused
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones.

Another instructions of this group are those, which set a particular value to the given register. Their structure is following:

- [31:24] 6 bits are used for opcode
- [23:19] 5 bits are used for a target register address
- [18:3] 16 bits are for the value in 2‘s complement code.
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones.

5.2.4 Branching instructions

Finally, lets look at branching instructions. There are two different approaches to conditional jumps accepted in FVLIW architecture: all the logic is done in one instruction or first command provides some computation and second one just behaves in respect with result of the first one. For example, one can use

```
jme $s1, $s2, $target
```

to check if values in $s1 and $s2 are equal and if so, jump to memory address stored in $target. However, it is also possible to use the method like following:

```
cmp $s1 , $s2
cjme $target
```

This code provides two separate operations: comparison and jump, which can be useful in some situations. Also, one can use some basic data about arithmetic operations stored by ALUs into the status register. For example, user can always know if the result of previous operation was Zero, Overflow or Negative. However, it is recommended not to use this feature as in case of several simultaneous usages of different ALUs it is undefined, what values will be in status register. Notice, that "cmp" operation belongs to the RR-type commands as it is implemented as usual subtraction with writing data about result to status register.

This is the structure of the default case of the B-type commands:

- [31:24] 8 bits bits are for opcode
- [23:21] to specify which comparison should be done
- [20:11] addresses of two registers with data to be compared
- [10:6] address of the register with target address
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones

But also, there is a following scheme:

- [31:24] 8 bits for opcode (which is actually a constant for this type of commands)
- [23:21] to specify which comparison should be done
- [20:16] address of the register with the target address
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones

And the case with immediate offset value:

- [31:24] 8 bits bits are for opcode
- [23:21] to specify which comparison should be done
- [20:5] offset value

- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones

Another important instruction is cmp, which looks like following:

- [31:24] 8 bits bits are for opcode
- [23:14] addresses of two registers with data to be compared
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones

Or with the immediate value:

- [31:24] 8 bits bits are for opcode
- [23:19] address of the register with data to be compared
- [18:3] immediate value to compare with
- [0] the last bit is used to mark if the next instruction depends on the result of this or previous ones

That is all the main instruction formats which are supported by FVLIW architecture. ***You will be able to get more detailed overview with the final version of this document.***

5.3 Instructions lists by purpose

In the following tables you will find lists of supported instructions for the FVLIW assembly language.

5.3.1 Reserved

Instruction	Operands	Description
cprs		Command for coprocessor

5.3.2 Load/Store

Instruction	Operands	Description
lw	$t, $s	Loads one word (basically 32-bit) from memory at the address $s to the register $t.
lb	$t, $s	Loads one byte from memory at the address $s to the 8 most significant bits of register $t.
sw	$t, $s	Stores a word from a register $s to memory at the address stored in register $t
sb	$t, $s	Stores a byte from 8 most significant bits of a register $s to memory at the address stored in register $t
seti	$t, {val}	Stores sign extended value of {val} to the $t register.

5.3.3 Branching

Note: *"jumps to a given address" means that next value of PC is BASE ADDRESS register value + given offset.(e.g. if base address is set to default value of 0x00101201 and offset is 100, then the next PC value will be 0x101301)*

Command	Operands	Description
jmp	$t	Changes PC to the value of the $t register.
jmpi	{value}	Changes PC to the value of the {value}.
jmr	$t	Adds the value of $t register to the value of current PC to get next one.
jmri	{val}	Adds the value of {val} to the value of current PC to get next one.

cjme	\$t	Jumps to the value of \$t register Z flag is 1.
jme	\$t, \$s1, \$s2	Jumps to the value of \$t register if \$s1 and \$s2 are equal.
cjmei	{val}	Jumps to the {val} if Z flag is 1.
jmei	{val}, \$s1, \$s2	Jumps to the {val} if \$s1 and \$s2 are equal.
cjmer	\$t	Adds value of \$t to the PC if Z flag is 1.
jmer	\$t, \$s1, \$s2	Adds value of \$t register to the PC if \$s1 and \$s2 are equal.
cjmeri	{val}	Adds {val} to the PC if Z flag is 1.
jmeri	{val}, \$s1, \$s2	Adds {val} to the PC if \$s1 and \$s2 are equal.
cjne	\$t	Jumps to the value of the \$t if Z flag is 0
jne	\$t, \$s1, \$s2	Jumps to the value of the \$t if \$s1 != \$s2
cjnei	{val}, \$s1, \$s2	Jumps to {val} if Z flag is 0.
jnei	{val}, \$s1, \$s2	Jumps to {val} if \$s1 and \$s2 are not equal.
cjner	\$t	Adds value of \$t register to the PC if Z flag is 0.
jner	\$t, \$s1, \$s2	Adds value of \$t register to the PC if values of \$s1 and \$s2 are not equal.
cjneri	{val}	Adds value of {val} to the PC if Z flag is 0.
jneri	{val}, \$s1, \$s2	Adds value of {val} to the PC if values of \$s1 and \$s2 are not equal.
cjle	\$t	Changes next value of PC to the value of \$t register if N flag is 1\$s2
jle	\$t, \$s1, \$s2	Changes next value of PC to the value of \$t register if value of \$s1 is less or equal \$s2
cjlei	{val}	Adds {val} to the PC if N flag is 1.
jlei	{val}, \$s1, \$s2	Adds {val} to the PC if value of \$s1 is less or equal \$s2
cjler	\$t	Jumps to the address stored in \$t if Z flag is 1.
jler	\$t, \$s1, \$s2	Jumps to the address stored in \$t if value of \$s1≤\$s2

cjleri	{val}, $s1, $s2	Adds value of {val} to the PC if Z flag is 1.
jleri	{val}, $s1, $s2	Adds value of {val} to the PC if value of $s1≤$s2
call	$t	Puts the current next value of the PC to the top of stack and jumps to the value of $t.
calli	{val}	Puts the current next value of the PC to the top of stack and jums assigns PC to the {val}.
callr	$t	Puts the current next value of the PC to the top of stack and adds value of $t register to the PC.
callri	{val}	Puts the current next value of the PC to the top of stack and adds value of {val} to the PC.
syscall	None	Stores current PC to the EPC register and jumps to address 0x00001201 in RAM.
ret	None	Pops a value from the stack and jumps to that value

5.3.4 Integer arithmetic

Command	Operands	Description
add	\$t, \$s1, \$s2	Adds values of \$s1 and \$s2 and stores result in \$t.
addi	\$t,\$s1, {immval}	Adds value of \$s1 to {immval} and stores result in \$t.
sub	\$t, \$s1, \$s2	Subtracts value of \$s1 register from value of \$s2 and stores result in \$t.
subi	\$t,\$s1, {immval}	Subtracts value of \$s1 from {immval} and stores result in \$t.
mul	\$t, \$s1, \$s2	Multiplies values of \$s1 and \$s2. As result is 64-bits wide it is stored in two registers: with addresses \$t and \$t+1.
muli	\$t,\$s1, {immval}	Multiplies value of \$s1 with {immval} and stores result in \$t. As result is 64-bits wide it is stored in two registers: with addresses \$t and \$t+1.
div	\$t, \$s1, \$s2	Divides value of \$s1 with value of \$s2 and stores result in \$t.
divi	\$t,\$s1, {immval}	Divides value of \$s1 with {immval} and stores result in \$t.

5.3.5 Floating-point arithmetic (to be implemented)

Command	Operands	Description
fadd		
faddi		
fsub		
fsubi		
fmul		
fmuli		

5.3.6 Logic operations

Command	Operands	Description
and	\$s1, \$s2, \$t	Computes the bit-wise value of \$s1 AND \$s2 and stores it in \$t.
or	\$s1, \$s2, \$t	Computes the bit-wise value of \$s1 OR \$s2 and stores it in \$t.
xor	\$s1, \$s2, \$t	Computes the bit-wise value of \$s1 XOR \$s2 and stores it in \$t.
nand	\$s1, \$s2, \$t	Computes the bit-wise value of \$s1 NAND \$s2 and stores it in \$t.

5.3.7 Shift operations

Command	Operands	Description
shl	\$t	Shifts all bits of the register to the right. The LSB(the most right one) is stored to C flag. The MSB is set to 0.
shr	\$t	Shifts all bits of the register to the right. The LSB(the most right one) is stored to C flag. The MSB is set to 0.
ror	\$t	Rotates a value of the register right.
rol	\$t	Rotates a value of the register left.

5.4 Opcodes listing

Command	8-bit Opcode	ALU command	JUMP sub-instruction
cprs	0000 0001	—	—
lw	0000 0010	—	—
lb	0000 0011	—	—
sw	0000 0100	—	—
sb	0000 0101	—	—
seti	0000 0110	—	—
jmp	0000 0111	—	—
jmpi	0000 1000	—	—
jmr	0000 1001	—	—
jmri	0000 1010	—	—
cjme	1000 1011	—	001
jme	0000 1011	—	001
cjmei	1000 1100	—	001
jmei	0000 1100	—	001
cjmer	1000 1101	—	001
jmer	0000 1101	—	001
cjmeri	1000 1110	—	001
jmeri	0000 1110	—	001
cjle	1000 1011	—	010
jle	0000 1011	—	010
cjlei	1000 1100	—	010
jlei	0000 1100	—	010
cjler	1000 1101	—	010
jler	0000 1101	—	010
cjleri	1000 1110	—	010
jleri	0000 1110	—	010
cjne	1000 1011	—	011
jne	0000 1011	—	011
cjnei	1000 1100	—	011
jnei	0000 1100	—	011
cjner	1000 1101	—	011
jner	0000 1101	—	011
cjneri	1000 1110	—	011
jneri	0000 1110	—	011
call	0000 1111	—	—
calli	0001 0000	—	—
callr	0001 0001	—	—

callri	0001 0010	——	——
syscall	0001 0011	——	——
ret	0001 0100	——	——
add	0001 0101	00001	——
sub	0001 0101	00010	——
mul	0001 0101	00011	——
div	0001 0101	00100	——
and	0001 0101	00101	——
xor	0001 0101	00110	——
nand	0001 0101	00111	——
or	0001 0101	01000	——
addi	1001 0110	——	——
subi	1011 0110	——	——
muli	1111 0110	——	——
divi	1110 0110	——	——
shl	0001 0111	——	——
shr	0001 1000	——	——
rol	0001 1001	——	——
ror	0001 1010	——	——

Table 5: Binary instruction codes

6 Appendix

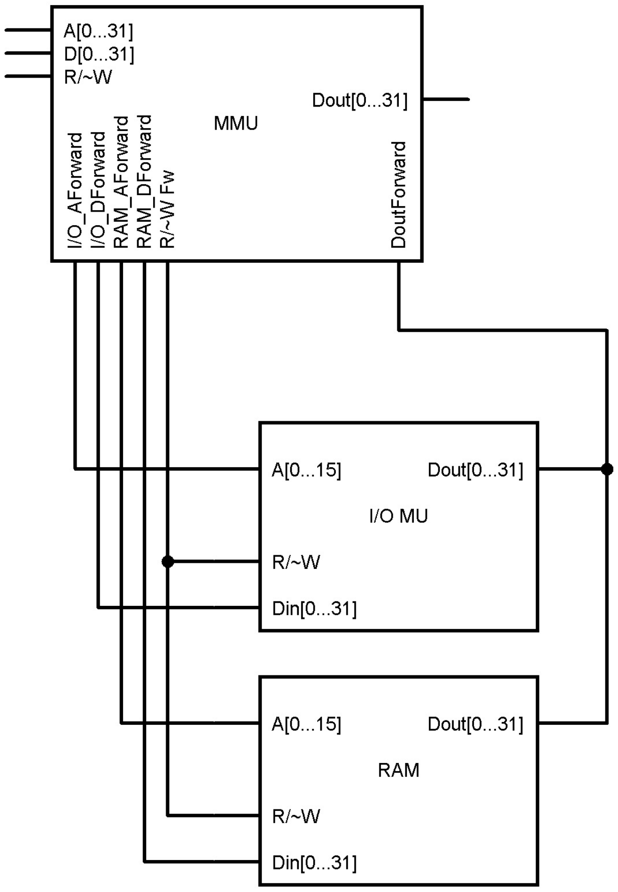

Figure 1: Simplest MMIO controller
MMU compares value of the A-bus and in respect with result of that comparison forwards data either to RAM or I/O unit.

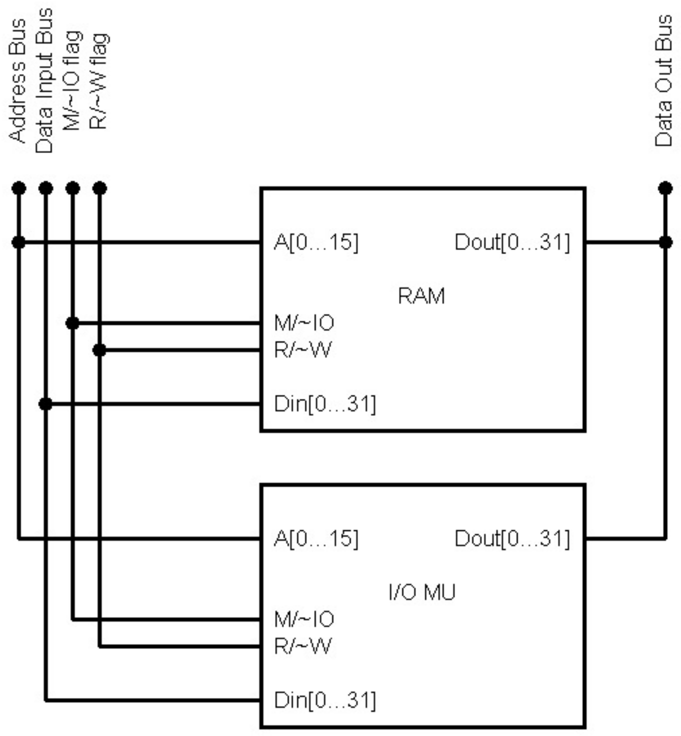

Figure 2: x86-like input/output controller
To specify which device to use, instruction must set a $M/\overline{IO}$ flag.

7 References

References

[1] Joseph A. Fisher *Very Long Instruction Word architectures and the ELI-512* ISCA '83 Proceedings of the 10th annual international symposium on Computer architecture Pages 140-150

[2] Joseph A. Fisher *Retrospective: very long instruction word architectures and the ELI-512* ISCA '98 25 years of the international symposia on Computer architecture (selected papers) Pages 34-36

[3] Carlo Galuzzi, Koen Bertels *The Instruction-Set Extension Problem: A Survey* ACM Transactions on Reconfigurable Technology and Systems (TRETS). Volume 4 Issue 2, May 2011

[4] David A. Patterson *Reduced instruction set computers* Communications of the ACM - Special section on computer architecture. Volume 28 Issue 1, Jan. 1985

List of Figures

Contents