The Virtual Instruction Set (VIS)

Carl Eric Codère

October 24, 2002

Contents

1 Introduction to the VIS **5**

2 General information **6**

2.1 Register set information 6
2.2 Addressing modes 6
2.2.1 Register direct 7
2.2.2 Register indirect 7
2.2.3 Immediate addressing 7
2.2.4 Relative addressing 7
2.3 Data types 8
2.4 memory issues 8
2.4.1 memory endian 8
2.4.2 the stack 8
2.4.3 alignment 8
2.5 Instruction set encoding 9
2.5.1 register encoding 11
2.5.2 opcode encoding 11
2.5.3 instruction encoding 11

3 Instruction reference **13**

3.1 ADD - Add register with register 13
3.2 ADDC - Add register with register with carrry 13
3.3 AND - And register with register 14
3.4 ASR - Arithmetic shift right 14
3.5 BEQL - Branch relative long if equal 15
3.6 BGEL - Branch relative long if greater or equal (signed) . . . 15
3.7 BGTL - Branch relative long if greater then (signed) 15
3.8 BLEL - Branch relative long if less then (signed) 16
3.9 BLTL - Branch relative long if less then (signed) 16

3.10 BNEL - Branch relative long if not equal 17
3.11 BLSL - Branch relative if lower or same long (unsigned) . . . 17
3.12 BASL - Branch relative if above or same long (unsigned) . . . 17
3.13 BATL - Branch relative if above long (unsigned) 18
3.14 BBTL - Branch relative if below long (unsigned) 18
3.15 BRAL - Branch relative long always 19
3.16 BCCS - Branch relative short if carry clear 19
3.17 BCSS - Branch relative short if carry set 19
3.18 BEQS - Branch relative short if equal 20
3.19 BGES - Branch relative short if greater or equal (signed) . . 20
3.20 BGTS - Branch relative short if greater then (signed) 21
3.21 BLES - Branch relative short if less then (signed) 21
3.22 BLTS - Branch relative short if less then (signed) 21
3.23 BNES - Branch relative short if not equal 22
3.24 BRAS - Branch relative short always 22
3.25 BSRL - Branch to subroutine relative 22
3.26 BVSS - Branch relative short if overflow set 23
3.27 BVCS - Branch relative short if overflow clear 23
3.28 BLSS - Branch relative if lower or same short (unsigned) . . . 24
3.29 BASS - Branch relative if above or same short (unsigned) . . 24
3.30 BATS - Branch relative if above short (unsigned) 24
3.31 BBTS - Branch relative if below short (unsigned) 25
3.32 CALL - Call subroutine 25
3.33 CMP - Compare register with register 26
3.34 DIVS - Divide signed register with register 26
3.35 DIVU - Divide unsigned register with register 26
3.36 FADD - Floating point add register with register 27
3.37 FCMP - Floating point compare register with register 27
3.38 FDIV - Floating point divide register with register 27
3.39 FLDD - Floating point load double into register 28
3.40 FLDS - Floating point load single into register 28
3.41 FMOVE - Floating point move from register to register . . . 29
3.42 FMUL - Floating point multiply register with register 29
3.43 FNEG - Floating point negate register 29
3.44 FSTD - Floating point store double from register 29
3.45 FSTS - Floating point store single from register 30
3.46 FSUB - Floating point subtract register with register 30
3.47 LBZX - Load byte zero extended into register 30
3.48 LBSX - Load byte sign extended into register 31
3.49 LILO - Load 16-bit immediate into register 31

3.50 LIMS - Load immediate short into register 31
3.51 LLSX - Load long sign extended into register 31
3.52 LLZX - Load long zero extended into register 32
3.53 LSL/LSR - Logical Shift left/Shift right 32
3.54 LWSX - Load word sign extended into register 33
3.55 LWZX - Load word zero extended into register 33
3.56 MOD - Modulo register with register 33
3.57 MOVE - Move from register to register 34
3.58 MULS - Multiply signed register with register 34
3.59 MULU - Multiply unsigned register with register 34
3.60 NEG - Negate register 34
3.61 NOP - No operation 35
3.62 NOT - Not register 35
3.63 OR - Or register with register 35
3.64 ORHI - Or upper 16-bit of register with constant 36
3.65 POPL - Pop long into register from stack 36
3.66 PUSHL - Push long from register onto stack 36
3.67 RETS - Return from subroutine and deallocate 36
3.68 STB - Store byte from register 37
3.69 STL - Store long from register 37
3.70 STW - Store word from register 37
3.71 SUB - Subtract register with register 38
3.72 SUBB - Subtract with borrow register with register 38
3.73 SYSCALL - system call 38
3.74 XOR - Exclusive or register with register 39

4 simplified opcodes 40
4.1 LEAA - Load effective absolute address 40
4.2 LIMM - Load immediate into register 40

5 interrupts 41

6 internal information 42
6.1 object file format 42
6.1.1 object header 42
6.1.2 code section 44
6.1.3 data section 44
6.1.4 symbol table 44
6.1.5 relocation information 44
6.1.6 debug information 45

6.1.7 string table . . . 45
6.2 execution view . . . 46
6.2.1 code segment . . . 46
6.2.2 data segment . . . 46
6.2.3 bss segment . . . 46
6.2.4 stack . . . 46
6.2.5 unhandled exceptions . . . 47
6.3 VIS Assembler syntax . . . 47

7 System V ABI : VIS Processor supplement 48
7.1 low-level system information . . . 48
7.1.1 Machine interface . . . 48
7.1.2 Function calling sequence . . . 49
7.1.3 Operating system interface . . . 49

8 Operating system interface 50
8.1 introduction . . . 50
8.1.1 pre-defined types . . . 50
8.1.2 standard handles . . . 50
8.1.3 signal information . . . 51
8.2 POSIX system calls . . . 51
8.2.1 base system calls . . . 51
8.3 non-POSIX system calls . . . 51
8.3.1 Execute a native command (SYS_NATCMD) . . . 51
8.3.2 Execute an interpreted command (SYS_VISCMD) . . . 52
8.3.3 List opened file descriptors for process (SYS_FILES) . . . 53
8.3.4 Call native shared library routine (SYS_LIBCALL) . . . 53

9 internal architecture 57
9.1 execution . . . 57
9.2 external libraries . . . 57
9.3 code separation . . . 57
9.3.1 vm_mem.c . . . 58
9.3.2 vm_vm.c . . . 58
9.3.3 vm_int.c . . . 58
9.3.4 vm_os.c . . . 59
9.3.5 vm_posix.c . . . 59
9.3.6 vm_api.c . . . 59

Chapter 1

Introduction to the VIS

The virtual instruction set is a portable CISC like instruction set, and can be used to as a back end for a compiler to generate asm specific code to different RISC target processors, or as an interpreted language in its binary format (bytecode). The goal of the architecture set format was to create a fast and compact instruction set.

Chapter 2

General information

2.1 Register set information

Because of how different RISC processors implement register allocation, and for ease of use, a compromise had to be made between different RISC processors. Therefore, 32 registers are available for use in the Virtual Machine, 16 of which are integer registers, while the 16 others are floating point registers.

Table 2.1: Available registers

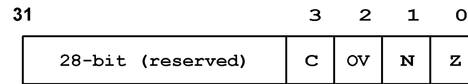
Name	Description
R0-R11	General purpose registers (32-bit)
CCR	Condition code register (32-bit)
SP	Stack pointer (32-bit)
FP	Frame or base pointer (32-bit)
PC	Program counter (32-bit)
FP0-FP15	Floating point registers (64-bit)

The CCR is used for both floating point operations and integer operations.

2.2 Addressing modes

As in general RISC architectures, memory addressing is severly limited, with a few exceptions most instructions only work in the direct register mode.

Figure 2.1: The CCR Register



C = Carry
OV = Overflow
N = Negative
Z = Zero

2.2.1 Register direct

rX →The value in the rX register is the value used

2.2.2 Register indirect

Indirect addressing mode is only allowed in a few instructions for loading and storing values to/from memory.

(rX) →Indirect register addressing, the register rX points to the memory location where to load/store.

(rX)+imm16 →Indirect register addressing with displacement, the values pointed to is equal to the value of Rx added with the sign-extended 16-bit value of imm16.

2.2.3 Immediate addressing

There are two forms of opcodes which support immediate addressing, either using a 5-bit immediate value or a 16-bit immediate value.

imm16 →The value is a signed 16-bit value

Only allowed with the `LILO` and `ORHI` opcodes.

imm5 →The value is a signed 5-bit value.

Only allowed with the `LIMS` opcode.

imm9 →The value is a signed 9-bit indicating the number of slots to deallocate from the stack.

Only allowed with the `RETS` opcode.

2.2.4 Relative addressing

This is used in the branch conditional opcodes, either using a long or a short form.

imm10 →The value is a signed 9-bit value shifted right by 1 which will be added to the Program counter (PC). imm26 →The value is a signed 25-bit, shifted right by 1, value which will be added to the Program counter (PC).

2.3 Data types

The VIS knows the following data types, which are named differently then most RISC processor implementations:

Table 2.2: Data types

Name	Description
byte	signed or unsigned 8-bit value
word	signed or unsigned 16-bit value
doubleword	signed or unsigned 32-bit value
quadword	signed or unsigned 64-bit value
single	IEEE 32-bit floating point format
double	IEEE 64-bit floating point format

2.4 memory issues

2.4.1 memory endian

The format of the operands in memory are in big-endian format. That is, the most significant byte (MSB) is stored at the lowest (or starting) address while the least significant byte (LSB) is stored at the highest (or ending) address. This is called big-endian because the big end of the scalar comes first in memory.

2.4.2 the stack

The stack works in slots instead of bytes, where each slot has a 4 byte length. Therefore the stack is always aligned to a 4 byte boundary.

2.4.3 alignment

All instructions are at least aligned on a 2 byte boundary, which means that the displacement encoding in the relative branch instructions must be

shifted left one bit to get the true displacement from the Program Counter (PC).

2.5 Instruction set encoding

Table 2.3: Instruction encoding (7-bits)

Value	opcode size	Fmt	Opcode
0	16 bits	1	BEQS
1	16 bits	1	BGES
2	16 bits	1	BGTS
3	16 bits	1	BLES
4	16 bits	1	BLTS
5	16 bits	1	BNES
6	16 bits	1	BRAS
7	16 bits	1	RETS
8	16 bits	1	BCCS
9	16 bits	1	BCSS
10	16 bits	1	BVCS
11	16 bits	1	BVSS
12	16 bits	1	BBSS
13	16 bits	1	BASS
14	16 bits	1	BATS
15	16 bits	1	BBTS
16	32 bits	8	BEQL
17	32 bits	8	BGEL
18	32 bits	8	BGTL
19	32 bits	8	BLEL
20	32 bits	8	BLTL
21	32 bits	8	BNEL
22	32 bits	8	BRAL
23	32 bits	8	BSRL
24	32 bits	1	BBSL
25	32 bits	1	BASL
26	32 bits	1	BATL
27	32 bits	1	BBTL
32	16 bits	2	ADD
33	16 bits	2	ADDC
34	16 bits	2	AND

Table 2.3: Instruction encoding (continued)

Value	opcode size	Fmt	Opcode
35	16 bits	2	ASR
36	16 bits	2	LSL
37	16 bits	2	LSR
38	16 bits	2	CMP
39	16 bits	2	SUB
40	16 bits	2	SUBB
41	16 bits	2	DIVS
42	16 bits	2	DIVU
43	16 bits	2	MOD
44	16 bits	2	MOVE
45	16 bits	2	MULS
46	16 bits	2	MULU
47	16 bits	2	NEG
48	16 bits	2	NOT
49	16 bits	2	OR
50	16 bits	2	XOR
51	16 bits	2	FADD
52	16 bits	2	FCMP
53	16 bits	2	FDIV
54	16 bits	2	FMOVE
55	16 bits	2	FMUL
56	16 bits	2	FNEG
57	16 bits	2	FSUB
64	32 bits	6	FLDD
65	32 bits	6	FLDS
72	32 bits	6	LBZX
73	32 bits	6	LBSX
74	32 bits	6	LLSX
75	32 bits	6	LLZX
76	32 bits	6	LWSX
77	32 bits	6	LWZX
80	32 bits	6	FSTD
81	32 bits	6	FSTS
88	32 bits	6	STB
89	32 bits	6	STL
90	32 bits	6	STW
112	16 bits	3	SYSCALL

Table 2.3: Instruction encoding (continued)

Value	opcode size	Fmt	Opcode
113	16 bits	3	NOP
114	16 bits	5	LIMS
115	32 bits	7	ORHI
116	32 bits	7	LILO
120	16 bits	4	CALL
126	16 bits	4	POPL
127	16 bits	4	PUSHL

2.5.1 register encoding

Registers are encoded as a 4-bit value, and can either represent one of the general purpose registers, or one of the floating point registers, depending on the target instruction.

Table 2.4: Register encoding

Name	Encoding (4-bits)
R0-R11	internal values 0-11
SP	internal value 12
FP	internal value 13
CCR	internal value 14
PC	internal value 15
FP0-FP15	internal values 0-15

2.5.2 opcode encoding

2.5.3 instruction encoding

All instructions are encoded as either 16-bit values or 32-bit values.

Figure 2.2: 16-bit instruction encoding

Form 1

Form 2

Form 3

Form 4

Form 5

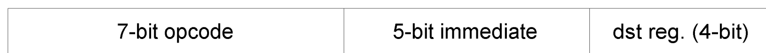


Figure 2.3: 32-bit instruction encoding

Form 6

Form 7

Form 8

Chapter 3

Instruction reference

3.1 ADD - Add register with register

Operation : Rx + Ry →Ry
Assembler syntax : ADD Rx,Ry

Description : Adds the source operand to the destination operand.
Instruction format : 2

CCR:

NV Set if the result is negative, cleared otherwise

ZF Set if the result is zero, cleared otherwise

OVF Set if an overflow occurs, cleared otherwise

CF Set if there is a carry, cleared otherwise

3.2 ADDC - Add register with register with carrry

Operation : Rx + Ry + Carry →Ry
Assembler syntax : ADDC Rx,Ry

Description : Adds the source operand plus the carry to the destination operand.
Instruction format : 2

CCR:

NV Set if the result is negative, cleared otherwise

ZF Set if the result is zero, cleared otherwise

OVF Set if an overflow occurs, cleared otherwise

CF Not affected

3.3 AND - And register with register

Operation : Rx AND Ry →Ry
Assembler syntax : AND Rx,Ry

Description: Performs the bitwise of the source operand with the destination operand and stores the result in the destination operand.
Instruction format : 2

CCR:

NV Set if the result is negative, cleared otherwise

ZF Set if the result is zero, cleared otherwise

OVF Always cleared.

CF Not affected.

3.4 ASR - Arithmetic shift right

Operation : Ry shifted right by (Rx modulo 32) →Ry; Keep MSB bit
Assembler syntax : ASR Rx,Ry

Description : Shifts all bits in Ry, Rx places to the right. This operation effectively divides a signed value by a multiple of two, without changing the sign. The carry flag can be used to round the result.
Instruction format : 2

CCR: OVF = NV XOR CF (after the result is calculated) NF = Set uf the MSB of the result is set Z = Set if the result is zero C = Set if before the shift, the LSB of the source to shift was 1, otherwise cleared.

3.5 BEQL - Branch relative long if equal

Operation : If ZF then PC + disp26 →PC
Assembler syntax : BEQL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.6 BGEL - Branch relative long if greater or equal (signed)

Operation : If (NF AND OVF) OR ((NOT NF) AND (NOT OVF)) then PC + disp26 →PC
Assembler syntax : BGEL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.7 BGTL - Branch relative long if greater then (signed)

Operation : If (NF AND OVF AND NOT ZF OR NOT NF AND NOT OF AND NOT ZF) then PC + disp26 →PC
Assembler syntax : BGTL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed

26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.8 BLEL - Branch relative long if less then (signed)

Operation : If (ZF OR NF AND NOT OVF OR NOT NF AND OVF) then PC + disp26 →PC
Assembler syntax : BLEL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.9 BLTL - Branch relative long if less then (signed)

Operation : If (NF AND NOT OVF) OR (NOT NF AND OVF) Then then PC + disp26 →PC
Assembler syntax : BLTL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.10 BNEL - Branch relative long if not equal

Operation : If NOT ZF then PC + disp26 $\rightarrow$PC
Assembler syntax : BNEL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.11 BLSL - Branch relative if lower or same long (unsigned)

Operation :
Assembler syntax : BLSL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.12 BASL - Branch relative if above or same long (unsigned)

Operation :
Assembler syntax : BASL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the

relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.13 BATL - Branch relative if above long (unsigned)

Operation :
Assembler syntax : BATL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.14 BBTL - Branch relative if below long (unsigned)

Operation :
Assembler syntax : BBTL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.15 BRAL - Branch relative long always

Operation : PC + disp26 →PC
Assembler syntax : BRAL <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 26-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 8

CCR: Not affected.

3.16 BCCS - Branch relative short if carry clear

Operation : If NOT C then PC + disp10 →PC
Assembler syntax : BCCS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.17 BCSS - Branch relative short if carry set

Operation : If C then PC + disp10 →PC
Assembler syntax : BCSS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.

Instruction format : 1

CCR: Not affected.

3.18 BEQS - Branch relative short if equal

Operation : If ZF then PC + disp10 →PC
Assembler syntax : BEQS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.19 BGES - Branch relative short if greater or equal (signed)

Operation : If (NF AND OVF) OR ((NOT NF) AND (NOT OVF)) then PC + disp10 →PC
Assembler syntax : BGES <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.20 BGTS - Branch relative short if greater then (signed)

Operation : If (NF AND OVF AND NOT ZF OR NOT NF AND NOT OF AND NOT ZF) then PC + disp10 →PC
Assembler syntax : BGTS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.21 BLES - Branch relative short if less then (signed)

Operation : If (ZF OR NF AND NOT OVF OR NOT NF AND OVF) then PC + disp10 →PC
Assembler syntax : BLES <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.22 BLTS - Branch relative short if less then (signed)

Operation : If (NF AND NOT OVF) OR (NOT NF AND OVF) Then then PC + disp10 →PC
Assembler syntax : BLTS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed

10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.23 BNES - Branch relative short if not equal

Operation : If NOT ZF then PC + disp10 →PC
Assembler syntax : BNES <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.24 BRAS - Branch relative short always

Operation : PC + disp10 →PC
Assembler syntax : BRAS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.25 BSRL - Branch to subroutine relative

Operation : SP - 4 →SP; PC →(SP); PC + disp26 →PC
Assembler syntax : BSRL <label>

Description: Pushes the 32-bit address on top of the stack, of the address immediately following the BSRL instruction. The relative displacement value (25-bit) is shifted left by 1 (to create a 26-bit signed displacement) and is added to the PC. Program execution continues at the new Program Counter value.
Instruction format : 8

CCR: Not affected.

3.26 BVSS - Branch relative short if overflow set

Operation : If OVF then PC + disp10 →PC
Assembler syntax : BVSS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.27 BVCS - Branch relative short if overflow clear

Operation : If NOT OVF then PC + disp10 →PC
Assembler syntax : BVCS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.28 BLSS - Branch relative if lower or same short (unsigned)

Operation :
Assembler syntax : BLSS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.29 BASS - Branch relative if above or same short (unsigned)

Operation :
Assembler syntax : BASS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.30 BATS - Branch relative if above short (unsigned)

Operation :
Assembler syntax : BATS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed

10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.31 BBTS - Branch relative if below short (unsigned)

Operation :
Assembler syntax : BBTS <label>

Description : If the specified condition is true, program execution continues at location (PC) + displacement. The displacement is a signed 10-bit value which after being sign extended to 32-bits, contains the relative distance in bytes from the current program counter to the destination program counter.
Instruction format : 1

CCR: Not affected.

3.32 CALL - Call subroutine

Operation : SP - 4 $\rightarrow$SP; PC $\rightarrow$(SP); Rx $\rightarrow$PC
Assembler syntax : CALL (Rx)

Description: Pushes the 32-bit address on top of the stack, of the address immediately following the CALL instruction. Program execution continues at the address pointed by Rx.
Instruction format : 4

CCR: Not affected.
Notes: The way to call a routine, si to get the address of the routine in a register an issue this instruction. For example:

```
LEAA MyRoutine,%r0    // Get address of routine
CALL (%r0)            // Call the routine
```

3.33 CMP - Compare register with register

Operation : Ry - Rx →CCR
Assembler syntax : CMP Rx,Ry

Description: Subtracts the source register from the destination register and sets the CCR according to the result. The destination register is not changed.
Instruction format : 2

CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected

3.34 DIVS - Divide signed register with register

Operation : Ry / Rx →Ry
Assembler syntax : DIVS Rx,Ry

Description: Does a sign divide of the 32-bit value of the source register with the 32-bit value in the destination register. The result is put in the destination register. In the case of a division by zero, the operation is not performed and the OVF flag is set in the CCR.
Instruction format : 2

CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if the divisor is zero, cleared otherwise

3.35 DIVU - Divide unsigned register with register

Operation : Ry / Rx →Ry
Assembler syntax : DIVU Rx,Ry

Description: Does an unsigned divide of the 32-bit value of the source register with the 32-bit value in the destination register. The result is put in the destination register. In the case of a division by zero, the operation is not performed and the OVF flag is set in the CCR.

Instruction format : 2

CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if the divisor is zero, cleared otherwise

3.36 FADD - Floating point add register with register

Operation : Source + Destination →Destination
Assembler syntax : ADD FPx,FPy

Description: Adds the source operand to the destination operand.
Instruction format : 2

CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if an overflow occurs, cleared otherwise

3.37 FCMP - Floating point compare register with register

Operation : Destination - Source →CCR
Assembler syntax : CMP FPx,FPy

Description: Subtracts the source register from the destination register and sets the CCR according to the result. The destination register is not changed.
Instruction format : 2

CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected

3.38 FDIV - Floating point divide register with register

Operation : Destination / Source →Destination
Assembler syntax : FDIV FPx,FPy

Description: Divides the destination by the source and puts the result back in the destination FPU register. A division by zero causes the exception handler to be invoked.
Instruction format : 2

CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected (?BUGBUG!)

3.39 FLDD - Floating point load double into register

Operation : Get operand at address →Destination
Assembler syntax : FLDD imm16(Rx),FPy

Description: Load a 64-bit IEEE double pointed to by the source into the destination floating point register. Loading an immediate into a floating point register is not allowed in the virtual machine, you have to go through a temporary memory operand tand load the constant from there.
Instruction format : 6
CCR: Not affected

3.40 FLDS - Floating point load single into register

Operation : Get operand at address round to double →Destination
Assembler syntax : FLDS imm16(Rx),FPy

Description: Load a 32-bit IEEE single pointed to by the source into the destination floating point register. Loading an immediate into a floating point register is not allowed in the virtual machine, you have to go through a temporary memory operand tand load the constant from there.
Instruction format : 6
CCR: Not affected

3.41 FMOVE - Floating point move from register to register

Operation : Source →Destination
Assembler syntax : FMOVE FPx,FPy

Description: Moves the data at the source to the destination location.
Instruction format : 2
CCR: Not affected

3.42 FMUL - Floating point multiply register with register

Operation : Source x Destination →Destination
Assembler syntax : FMUL FPx,FPy

Description: Multiplies two 32-bit floating point operands yielding a signed 32-bit result.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected (?BUGBUG?)

3.43 FNEG - Floating point negate register

Operation : 0 - Source →Destination
Assembler syntax : FNEG Rx,Ry

Description: Subtracts the source operand from zero and stores the result in the destination location.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected

3.44 FSTD - Floating point store double from register

Operation : Source →Operand at address
Assembler syntax : FSTD FPx,imm16(Ry)

Description: Stores a 64-bit IEEE double value from the source floating point register to the address pointed to by the destination.
Instruction format : 6
CCR: Not affected

3.45 FSTS - Floating point store single from register

Operation : Source rounded to single →Operand at address
Assembler syntax : FSTS FPx,imm16(Ry)

Description: Stores a 32-bit IEEE single value from the source floating point register to the address pointed to by the destination.
Instruction format : 6
CCR: Not affected

3.46 FSUB - Floating point subtract register with register

Operation : Destination - Source →Destination
Assembler syntax : FSUB FPx,FPy

Description: Subtracts the source register from the destination register and puts the result into the destination register.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected

3.47 LBZX - Load byte zero extended into register

Operation : Get operand at address →Destination
Assembler syntax : LBZX imm16(Rx),Ry

Description: Load a byte pointed to by the source into the destination register, the upper 24-bit of the destination register are set to zero.
Instruction format : 6
CCR: Not affected

3.48 LBSX - Load byte sign extended into register

Operation : Get operand at address →Destination
Assembler syntax : LBSX imm16(Rx),Ry

Description: Load a byte pointed to by the source into the destination register, the upper 24-bit of the destination register are set to the most significant bit of the lower byte of the value loaded.
Instruction format : 6
CCR: Not affected

3.49 LILO - Load 16-bit immediate into register

Operation : 16-bit constant →(zeroed-high 16-bits — constant)
Assembler syntax : LILO #imm6,Ry

Description: Zeroes the entire destination register, and then loads the 16-bit value into the low order word of the register. The value is not sign extended.
Instruction format : 7
CCR: Not affected

3.50 LIMS - Load immediate short into register

Operation : 5-bit signed constant extended to 32-bit →Destination
Assembler syntax : LIMS #imm5,Ry

Description: Loads the signed 5-bit constant into the destination register.
Instruction format : 5
CCR: Not affected

3.51 LLSX - Load long sign extended into register

Operation : Get operand at address →Destination
Assembler syntax : LLSX imm16(Rx),Ry

Description: Load a dword pointed to by the source into the destination register. This operation currently does the same thing as the LLZX

instruction, but it is here for future expansion for a 64-bit architecture, it should be used when a signed 32-bit value must be loaded into a register.
Instruction format : 6
CCR: Not affected

3.52 LLZX - Load long zero extended into register

Operation : Get operand at address →Destination
Assembler syntax : LLZX imm16(Rx),Ry

Description: Load a dword pointed to by the source into the destination register. This operation currently does the same thing as the LLSX instruction, but it is here for future expansion for a 64-bit architecture, it should be used when an unsigned 32-bit value must be loaded into a register.
Instruction format : 6
CCR: Not affected

3.53 LSL/LSR - Logical Shift left/Shift right

Operation : Ry shifted by count in Rx →Ry
Assembler syntax : LSd Rx,Ry
where d is direction, L or R
Description: The shift count is in the Rx data register modulo 32.
The LSL instruction shifts the operand to the left the number of position specified as the shift count. Zeros are shifted into the low-order bit.
The LSR instruction shifts the operand to the left the number of position specified as the shift count. Zeros are shifted into the high-order bits.
Instruction format : 2
CCR for LSL: OVF = NV XOR CF (after the result is calculated) NF = Set uf the MSB of the result is set Z = Set if the result is zero C = Set if before the shift, the MSB of the source to shift was 1, otherwise cleared.
CCR for LSR: OVF = NV XOR CF (after the result is calculated) NF = Cleared. Z = Set if the result is zero C = Set if before the shift, the MSB of the source to shift was 1, otherwise cleared.

3.54 LWSX - Load word sign extended into register

Operation : Get operand at address →Destination
Assembler syntax : LWSX imm16(Rx),Ry

Description: Load a word pointed to by the source into the destination register, the upper 16-bit of the destination register are set to the most significant bit of the lower word of the value loaded.
Instruction format : 6
CCR: Not affected

3.55 LWZX - Load word zero extended into register

Operation : Get operand at address →Destination
Assembler syntax : LWZX imm16(Rx),Ry

Description: Load a word pointed to by the source into the destination register, the upper 16-bit of the destination register are set to zero.
Instruction format : 6
CCR: Not affected

3.56 MOD - Modulo register with register

Operation : Destination MOD Source →Destination
Assembler syntax : MOD Rx,Ry

Description: Does a 32-bit modulo division and stores the result into the destination register. In the case of a modulo by zero, the operation is not performed and the OVF is set in the CCR.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if the divisor is zero, cleared otherwise

3.57 MOVE - Move from register to register

Operation : Rx →Ry
Assembler syntax : MOVE Rx,Ry

Description: Moves the data at the source to the destination location.
Instruction format : 2
CCR: Not affected

3.58 MULS - Multiply signed register with register

Operation : Rx x Ry →Ry
Assembler syntax : MULS Rx,Ry

Description: Multiplies two signed operands yielding a signed 32-bit result.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if the result does not fit in the destination register, cleared otherwise.

3.59 MULU - Multiply unsigned register with register

Operation : Rx x Ry →Ry
Assembler syntax : MULU Rx,Ry

Description: Multiplies two unsigned operands yielding an unsigned 32-bit result.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if the result does not fit in the destination register, cleared otherwise.

3.60 NEG - Negate register

Operation : 0 - Rx →Ry
Assembler syntax : NEG Rx,Ry

Description: Subtracts the source operand from zero and stores the result in the destination location.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Set if MSB of result is set. CF = Set , except if Ry is zero

3.61 NOP - No operation

Operation : None
Assembler syntax : NOP

Description: Performs no operation. The processor state, other than the program counter is unaffected.
Instruction format : 3
CCR: Not affected.

3.62 NOT - Not register

Operation : NOT Rx →Ry
Assembler syntax : NOT Rx,Ry

Description: All bits of the source operand are complemented and stored into the destination register. The source register is not modified.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Cleared CF = Not affected

3.63 OR - Or register with register

Operation : Rx OR Ry →Ry
Assembler syntax : OR Rx,Ry

Description: Performs an inclusive or operation on the source with the destination and stores the result in the destination register.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Cleared CF = Not affected

3.64 ORHI - Or upper 16-bit of register with constant

Operation : 16-bit constant OR high 16-bit of Ry
Assembler syntax : ORHI #imm16,Ry

Description: Ors the high 16-bits of the registers. The low order 16-bits of the register are not affected by this operation.
Instruction format : 7
CCR: No affected.

3.65 POPL - Pop long into register from stack

Operation : (SP) $\rightarrow$Rx; SP + 4
Assembler syntax : POPL Rx

Description: Pulls the dword stored on top of the stack, and stores the value in the lower 32-bit of the destination register. All other bits of the destination register are unaffected.
Instruction format : 4

CCR: Not affected

3.66 PUSHL - Push long from register onto stack

Operation : SP - 4; Rx $\rightarrow$(SP)
Assembler syntax : PUSHL Rx

Description: Pushes the lower 32-bits of the register onto the stack.
Instruction format : 4

CCR: Not affected

3.67 RETS - Return from subroutine and deallocate

Operation : (SP) $\rightarrow$PC; SP + 4 + (imm9*4) $\rightarrow$SP
Assembler syntax : RETS #imm9

Description: Pulls the PC from the top of the stack and adds the number of slots to release from to the Stack pointer. The number of slots to release is the sign-extended 9-bit immediate operand multiplied by the size of a slot on the stack (4 bytes). The previous program counter is lost
Instruction format : 1
CCR: Not affected

3.68 STB - Store byte from register

Operation : Rx $\rightarrow$(Ry + imm16)
Assembler syntax : STB Rx,imm16(Ry)

Description: Stores an 8-bit value from the source register to the address pointed to by the destination which is the sum of Ry and the sign-extended immediate 16-bit constant.
Instruction format : 6
CCR: Not affected

3.69 STL - Store long from register

Operation : Rx $\rightarrow$(Ry + imm16)
Assembler syntax : STL Rx,imm16(Ry)

Description: Stores a 32-bit value from the source register to the address pointed to by the destination which is the sum of Ry and the sign-extended immediate 16-bit constant.
Instruction format : 6
CCR: Not affected

3.70 STW - Store word from register

Operation : Rx $\rightarrow$(Ry + imm16)
Assembler syntax : STW Rx,imm16(Ry)

Description: Stores a 16-bit value from the source register to the address pointed to by the destination which is the sum of Ry and the sign-extended immediate 16-bit constant.

Instruction format : 6
CCR: Not affected

3.71 SUB - Subtract register with register

Operation : Ry - Rx →Ry
Assembler syntax : SUB Rx,Ry

Description: Subtracts the source register from the destination register and puts the result into the destination register.
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected CF = Set if a borrow occurs, cleared otherwise

3.72 SUBB - Subtract with borrow register with register

Operation : Ry - Rx - CF →Ry
Assembler syntax : SUBB Rx,Ry

Description: Subtracts the source register from the destination register and the carry flag and puts the result into the destination register.
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Not affected CF = Set if a borrow occurs, cleared otherwise

3.73 SYSCALL - system call

Operation : SP - 4 →SP; PC →(SP); Call operating system
Assembler syntax : syscall

Description: Call operating system services, as described later in this reference guide. No registers are specifically saved by the hardware (except for the PC) when executing this instruction, this should either be done by the operating system or user mode code.
Instruction format : 3
CCR: Not affected

3.74 XOR - Exclusive or register with register

Operation : Rx XOR Ry →Ry
Assembler syntax : XOR Rx,Ry

Description: Performs an exclusive or operation on the source with the destination and stores the result in the destination register.
Instruction format : 2
CCR: NV = Set if the result is negative, cleared otherwise ZF = Set if the result is zero, cleared otherwise OVF = Cleared CF = Not affected

Chapter 4

simplified opcodes

These opcodes are recognized by the assembler, but actually generate a sequence of instructions instead of a single instruction.

4.1 LEAA - Load effective absolute address

Operation : Effective Address →Destination
Assembler syntax : LEAA imm32,Ry

Description: Loads the effective address into the specified destination register. This is actually the sequence `LILO` followed by `ORHI`.
CCR: Not affected

4.2 LIMM - Load immediate into register

Operation : Constant →Destination
Assembler syntax : LIMM #imm32,Ry

Description: Loads the signed 32-bit constant into the destination register. This is actually the sequence `LILO` followed by `ORHI`.
CCR: Not affected

Chapter 5

interrupts

The virtual machine has several interrupts which can be generated when hardware faults occurs. In this specification, it is unspecified where this vector table resides.

Possible interrupt vectors are shown in table (5.1). Each process can independently set the interrupt vectors to specific handles without affecting the interrupt vectors of other processes.

Table 5.1: Interrupt vectors

Vector Number	Description
0	Reset initial PC (unsupported)
1	Illegal instruction
2	Invalid memory access
3	Illegal bus accesss
4	Syscall handler (unsupported)
5	STEP Handler (unsupported)
6	Breakpoint 0 handler (unsupported)
7	Breakpoint 1 handler (unsupported)
8	Breakpoint 2 handler (unsupported)
9	Breakpoint 3 handler (unsupported)
10..31	Reserved (unsupported)

Chapter 6

internal information

6.1 object file format

The object file format is a similar to COFF, but in a simplified format. The next figure gives general information on the format of the file. All the data in the COFF file is stored in big-endian format.

6.1.1 object header

This part of the file gives information on where to locate the different sections of the file, as well as give version information on the file format. It also indicates if this an executable file or an object file.

```
struct vis_header
{
  uint8 magic[5];       /* magic value */
  uint8 reserved;       /* reserved for future usage */
  uint8 version;        /* major version of file format */
  uint8 revision;       /* minor revision of file format */
  int32 codeoffset;     /* offset from start of file into code section */
  int32 codesize;       /* number of bytes of executable code */
  int32 dataoffset;     /* offset from start of file into data section */
  int32 datasize;       /* number of bytes of initialized data (optionnaly padded) */
  int32 bsssize;        /* number of bytes in BSS section */
  int32 offsettosyms;   /* offset from start of file into sym table (0 = none) */
  int32 nrsyms;         /* number of symbols */
  int32 offsettorelocs; /* offset from start of file into relocation info */
  int32 nrrelocs;       /* number of relocation entries */
```

Figure 6.1: Object file overview

FILE HEADER
TEXT SECTION (optional)
DATA SECTION (optional)
SYMBOL TABLE (optional)
RELOCATION INFORMATION (optional)
DEBUG INFORMATION (optional)
STRING TABLE (optional)

```
  int32 stroffset;      /* offset into string table */
  int32 strsize;        /* size of string table in bytes */
  int32 debugoffset;    /* offset from start of file into debug section (0 = none) */
  int32 debugentries;   /* number of debug entries */
  int32 dbgstroffset;   /* offset into debug string table */
  int32 dbgstrsize;     /* size of debug string table */
  int32 entry;          /* program entry point offset into code section */
  int32 base;           /* start of load image virtual address */
} _vis_header_ PACKED;
```

6.1.2 code section

Contains all the executable code of the virtual machine. In executable files, all relocatable code must have been resolved to physical addresses.

6.1.3 data section

Contains all the initialized data for the executable file or object module.

6.1.4 symbol table

Contains information on all the symbols present and defined in the executable file or object file, be them local or global.

The symbol table section also gives information on the location of the symbols in the different sections. Each symbol table entry has the following format (packed structure):

```
 struct internal_sym {
  int32 s_offset;      /* offset from start of section */
  int32 s_size;        /* size of data in bytes (excludes alignment) */
  uint8 s_section;     /* Section of symbol : 0=text, 1=data, 2=bss  */
  uint8 s_global;      /* <>0 if the symbol is globally visible      */
  int32 s_data;        /* index into string table                    */
  int32 s_length;      /* length of string in string table           */
 }
```

6.1.5 relocation information

In executable files, this section should be empty, since all relocatable instructions have been resolved. In object files, this gives information on all

instructions which reference non-defined symbols which must be resolved at link time.

Each entry represents information on how to relocate opcodes in the current object file (packed structure):

```
struct internal_reloc {
  int32   r_offset;      /* instruction offset from start of section   */
  uint8   r_type;        /* relocation type : 1=9bit, 2=25bit, 3=32bit */
  uint8   r_solved;      /* symbol is resolved (<>0)                   */
  int32   r_data;        /* index into string table                    */
  uint16  r_length;      /* length of string in string table           */
}
```

6.1.6 debug information

This section contains debug information in the stabs debug format if the module was compiled with debug information. Otherwise, this section is not present in the file.

Each entry represents an entry in debug format. Each entry has the following format (packed structure):

```
struct internal_nlist {
  uint32 n_strx;     /* index into string table       */
  uint8  n_type;     /* type of debug information     */
  uint8  n_other;    /* other information             */
  uint16 n_desc;     /* description field             */
  int32  n_value;    /* value of symbol               */
}
```

The following debug symbols are currently supported:

N_SO : Name of the source file N_FUN : Routine definition start / end N_LSYM : Local symbol to a routine N_RSYM : Register symbol N_PSYM : Parameter to a routine N_LBRAC : Start of a lexical blocks N_RBRAC : End of a lexical block

6.1.7 string table

This contains all strings which are pointed to by the symbol table and relocation information section.

6.2 execution view

6.2.1 code segment

The code segment start is address *0x1000000* of the process.

In this version of this architecture this currently cannot be changed. The entry point of the program is specified in the executable header, and will be used by the operating system, as the entry point of the application.

6.2.2 data segment

The initialized data segment should follow the code segment in memory.

6.2.3 bss segment

The BSS segment should be allocated after the data segment in memory, it is also to note that the BSS segment is not zero-filled, contrary to most other architectures. The BSS section does not exist physically in the object file, it is created at load time using thec parameters specified in the object file header.

6.2.4 stack

Contrary to most architectures, the stack area must be setup by the application at startup. and the stack pointer should be set to point at the start of the stack area.

```
.text
_main:
   leaa   STACK,%sp
   lilo   #256,%r0
   add    %r0,%sp

.data
STACK_SIZE:
    .long 256
.bss
.comm STACK,256
```

6.2.5 unhandled exceptions

All exceptions which are unhandled by the applications will be caught by the operating system, which will display an error message in english, and abort the application with a status code equal to the signal number.

6.3 VIS Assembler syntax

The entry point of the program should have the `_main` symbol. The assembler is case-sensitive on symbols, but not on opcodes.

Chapter 7

System V ABI : VIS Processor supplement

7.1 low-level system information

7.1.1 Machine interface

Data representation

Within this specification, the term word refers to a 16-bit object, the term long refers to a 32-bit object, and the term doublelong refers to a 64-bit object.

Table 7.1: Scalar types

Type	C	sizeof	alignment (bytes)	vis
integral	char	1	1	byte
integral	unsigned char	1	1	byte
integral	short	2	2	word
integral	int	4	4	long
integral	long	4	4	long
integral	long long	8	8	N/A
pointer	any	4	4	long
floating-point	float	4	4	single
floating-point	double	8	8	double

7.1.2 Function calling sequence

Registers and the stack frame

7.1.3 Operating system interface

Process Initialization

The entry point receives three parameters in registers, where R0 gives the argument count `argc`, and R1 points to `argv`, an array of null-terminated strings. The values are volatile, and should be copied to temporary areas at startup if they wish to be used.

Chapter 8

Operating system interface

8.1 introduction

The operating system interface is a subset of the POSIX.1-1996 Interface. The following sections describes some information which can help using the operating system services. For the exact interface information consult the furnished C header files.

8.1.1 pre-defined types

The following pre-defined types are used to denote operand sizes.

Table 8.1: Pre-defined C types

Name	Description
int64	signed value (64-bit)
int32	signed value (32-bit)
int16	signed value (16-bit)

8.1.2 standard handles

The standard file descriptors used for user input and output are pre-defined (shown in table (8.2)), and are always accessible, i.e they can never be closed.

Table 8.2: Pre-defined file descriptors

Name	Value	Description
STDIN	0	Standard input stream
STDOUT	1	Standard output stream
STDERR	2	Standard error stream

8.1.3 signal information

The different hardware signals which can be generated, as well as their numeric values are defined in table (8.3). A brief description of their usage is also provided.

8.2 POSIX system calls

All POSIX interface routines are called via the `syscall` assembler instruction. The format of all parameters is shown in table (8.4).

In other words, all parameters are passed left to right in registers, starting from register R1. The return value is stored in register R0.

8.2.1 base system calls

The following lists the system call numbers, as defined by this specification:

8.3 non-POSIX system calls

In addition to the system calls available, additional non-POSIX system calls are also available to interface to native based programs and shared libraries.

Additional syscall values are shown in table (8.6).

8.3.1 Execute a native command (SYS_NATCMD)

synopsis

```
int system(const char* cmd);
```

Table 8.3: Pre-defined file descriptors

Name	Value	Description
SIGABRT	1	Not supported
SIGALRM	2	Not supported
SIGFPE	3	Not supported
SIGHUP	4	Not supported
SIGILL	5	Invalid hardware instruction
SIGINT	6	Not supported
SIGKILL	7	Not supported
SIGPIPE	8	Not supported
SIGQUIT	9	Not supported
SIGSEGV	10	Detection of invalid memory reference
SIGTERM	11	Termination signal
SIGUSR1	12	Not supported
SIGUSR2	13	Not supported
SIGCHLD	14	Not supported
SIGCONT	15	Not supported
SIGSTOP	16	Not supported
SIGTSTP	17	Not supported
SIGTTIN	18	Not supported
SIGTTOU	19	Not supported
SIGBUS	20	Not supported
SIGSTEP	21	Signal generated when using ptrace syscall with single stepping
SIGTRAP	22	Signal generated when breakpoint with ptrace syscall is true

description

8.3.2 Execute an interpreted command (SYS_VISCMD)

synopsis

```
int vissystem(const char* cmd);
```

Table 8.4: Syscall interface entry

Register nr.	Description
R0	System call number to do
R1..Rn	Parameters for the syscall, from left-to-right, based on the C declaration.

description

8.3.3 List opened file descriptors for process (SYS_FILES)

synopsis

```
int sys_files(char** fnames);
```

description

This system call can be used to retrieve the names of all opened files by this process. The value returned is a pointer to an array of null terminated strings, ending with a NULL character. The process always has the standard I/O handles oppened.

returns

Upon successfull execution, this routine returns 0, otherwise it returns one of the POSIX defined error codes.

errors

Currently, only EACCES can be returned in the case the `uid` of this process does not permit obtaining this information.

8.3.4 Call native shared library routine (SYS_LIBCALL)

synopsis

```
int sys_libcall(int handle, char *name,  paramstr_t *params);
```

description

This routine is used to call a dynamically loaded native shared library. `handle` is the handle to the opened shared library. `name` is the name of

the routine to call and `params` is a structure containing the parameters to pass to the shared library routine.

The parameter structure is as follows:

```
struct {
  int nr;          // Number of parameters of the routine
  char* params;    // Array of data in TLV format
}
```

The only possible parameter types which can be passed to this routine are as follows:

DATA_SIGNED These represent signed integral data, which can currently have their length field set to 1, 2, 4 or 8 bytes.

DATA_UNSIGNED These represent unsigned integral data, which can currently have their length field set to 1, 2, 4 or 8 bytes.

DATA_FLOAT These represent floating point values, which can currently have their length field set to 4 or 8 bytes.

DATA_POINTER These represent a pointer (typed or untyped), which always have a length field of 4 bytes.

DATA_VOID This represent an empty parameter.

returns

Upon successfull execution, this routine returns 0, otherwise it returns one of the POSIX defined error codes.

errors

Currently, only EACCES can be returned in the case the `uid` of this process does not permit calling this operating system service.

Table 8.5: System call numbers

Name	Value
SYS_EXIT	0x00
SYS_SIGACTION	0x01
SYS_UNAME	0x02
SYS_TIME	0x03
SYS_GETENV	0x04
SYS_SYSCONF	0x05
SYS_OPENDIR	0x06
SYS_READDIR	0x07
SYS_CLOSEDIR	0x08
SYS_CHDIR	0x09
SYS_GETCWD	0x0A
SYS_OPEN	0x0B
SYS_CREAT	0x0C
SYS_MKDIR	0x0D
SYS_RMDIR	0x0E
SYS_RENAME	0x0F
SYS_STAT	0x10
SYS_FSTAT	0x11
SYS_CHMOD	0x12
SYS_ACCESS	0x13
SYS_FCHMOD	0x14
SYS_UTIME	0x15
SYS_FTRUNCATE	0x16
SYS_FPATHCONF	0x17
SYS_READ	0x18
SYS_WRITE	0x19
SYS_FSYNC	0x1A
SYS_LSEEK	0x1B
SYS_CLOSE	0x1C
SYS_PATHCONF	0x1D

Table 8.6: Additional system calls

Define	Syscall Nr.	C Prototype
SYS_NATCMD	0x83	int system(char* cmd);
SYS_VISCMD	0x84	int vissystem(char* cmd);
SYS_FILES	0x85	int sys_files(char** files);
SYS_LIBOPEN	0x80	int void *dlopen(const char *file, int mode);
SYS_LIBCALL	0x82	int
SYS_LIBCLOSE	0x81	int