Vircon32

32-BIT VIRTUAL CONSOLE

System specification

Part 3: The processor (CPU)

Document date 2023.01.22 Written by Carra

What is this?

This document is part number 3 of the Vircon32 system specification. This series of documents defines the Vircon32 system, and provides a full specification describing its features and behavior in detail.

The main goal of this specification is to define a standard for what a Vircon32 system is, and how a gaming system needs to be implemented in order to be considered compliant. Also, since Vircon32 is a virtual system, an important second goal of these documents is to provide anyone with the knowledge to create their own Vircon32 implementations.

About Vircon32

The Vircon32 project was created independently by Carra. The Vircon32 system and its associated materials (including documents, software, source code, art and any other related elements) are owned by the original author.

Vircon32 is a free, open source project in an effort to promote that anyone can play the console and create software for it. For more detailed information on this, read the license texts included in each of the available software.

About this document

Summary

Part 3 of the specification defines the console processor, or Central Processing Unit (CPU). This document will describe the behavior of this chip, its internal elements, its instruction set architecture and the execution process.

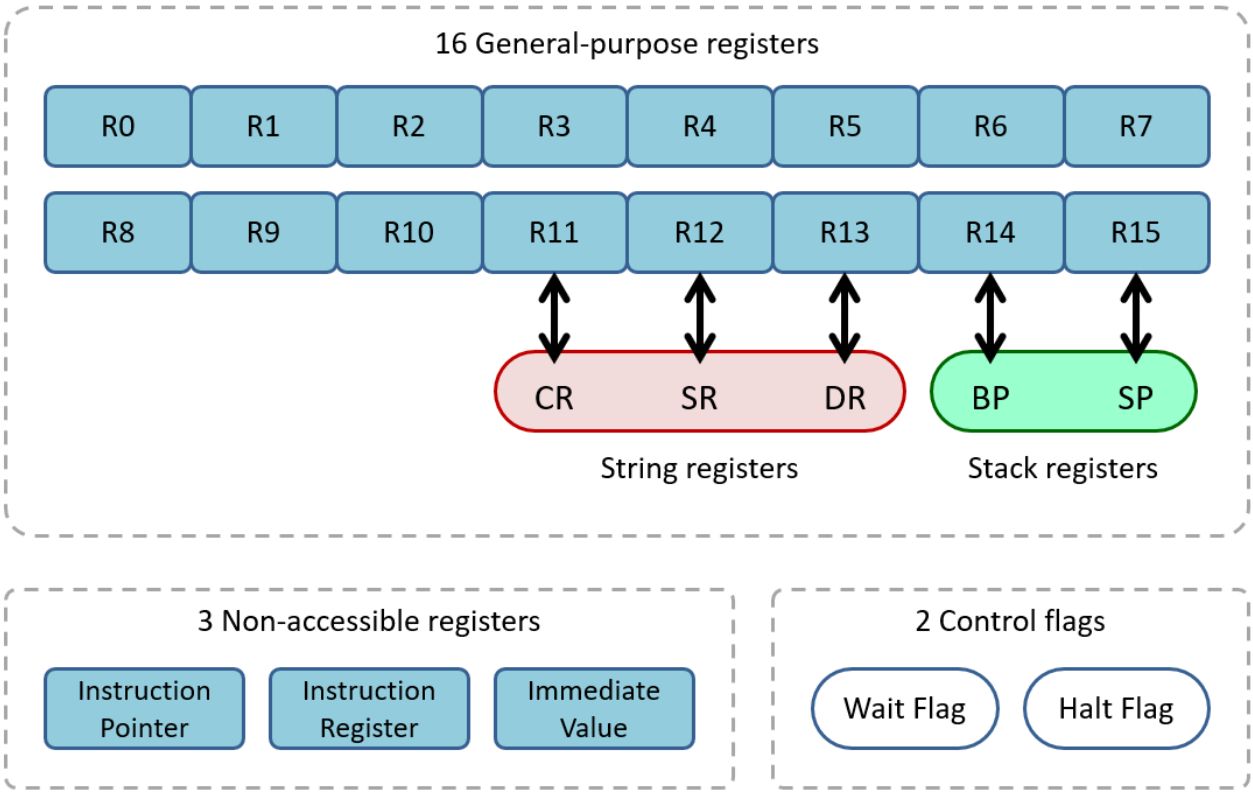
1 Introduction

The CPU is the main chip in the Vircon32 console. Its function is to read the program instructions and execute them in sequence to produce the program's described behavior. As part of this execution the CPU will also communicate with other console components and send commands to other chips to control their functions. Since the CPU is always the chip initiating communication, it is connected as master device to both the memory bus and the control bus.

This processor features a pure 32-bit architecture: as all Vircon32 components, it can only work with 32-bit data. Its instruction set is loosely based on the x86 family of processors, but adapted to Vircon32's feature set. It has also been simplified so that there are only 64 instructions. Those instructions also present fewer variants compared to other CPU designs.

1.1 CPU internal elements

This image shows an overview of the internal elements existing in the CPU. The following sections will explain each of those elements, and show how some of the general registers also receive specific functions when the CPU is performing operations relative to the stack (the area shown here in green) or string processing (the red area).



2 CPU Registers

The CPU features a set of 32-bit registers. Their function is to store any data needed for the execution. Registers can be separated in 2 different groups:

2.1 General-purpose registers

General-purpose registers are meant to support program execution, and they are the only registers that are directly accessible to the programmer. The CPU has an array of 16 general-purpose registers, which are named R0 to R15.

The last 5 of those registers are also used by some CPU features (stack and string operations) that will be covered in next chapters. Because of that, they will also receive alternative names reflecting those roles.

2.2 Internal registers

These registers are reserved for the operation of the CPU itself, and are therefore not directly accessible to the program. The 3 internal registers are:

Instruction Pointer

It holds the memory address from which the CPU will read the next program instruction.

Instruction Register

It stores the last read instruction, this is, the one currently being executed.

Immediate Value

If the last read instruction uses an immediate value, it is read and stored here.

3 The stack

The CPU implements a hardware stack, which is used to store and extract values following a Last-In, First-Out policy. Its primary function is to be used by call and return instructions to direct the program flow. Higher level languages will also typically use the stack to store “stack frames”, which save the language’s function call contexts. The programmer can also explicitly push and pop values from the stack.

3.1 Stack operation

The stack is located at the highest positions of RAM memory and grows downwards. Stack operation uses the last 2 general-purpose registers. In addition to their regular name, they can also be referred to as these aliases from their role in the stack:

R14 = BP (Base Pointer)
R15 = SP (Stack Pointer)

The Base Pointer points to stack's current highest position (commonly called the bottom of the stack), while the Stack Pointer keeps track of its lowest position (top of the stack).

Process for pushing to the stack

When the CPU pushes a value into the stack, it follows these steps.
In later sections, this operation will be represented as: Stack.Push(Value)

(1) Memory[StackPointer] = Value;
(2) StackPointer –= 1;

Process for popping from the stack

When the CPU pops a value from the stack into a register, it follows these steps.
In later sections, this operation will be represented as: Register = Stack.Pop( )

(1) StackPointer += 1;
(2) Register = Memory[StackPointer];

4 String operations

Some CPU instructions are designed to allow a single program instruction to operate on a set of consecutive memory addresses. Some CPUs refer to these sets of data as "strings", even if they don't represent text.

String operations use the 3 general-purpose registers right before the stack registers. In addition to their regular name, they can also be referred to as these aliases from their role in string operations:

R11 = CR (Count Register)
R12 = SR (Source Register)
R13 = DR (Destination Register)

To allow an instruction to process multiple addresses, the CPU automatically repeats that same instruction until the count register reaches zero. This way, the CPU implements a fast internal loop that allows for more efficient bulk processing. Every time the instruction

is processed, the counter is automatically decreased. In addition to that, up to 2 pointer registers (for source and destination addresses) will be automatically increased.

The way a program would use string operations is to first place the correct source and destination addresses in SR and DR, then write the number of iterations in CR, and finally use the string instruction.

5 Control flags

The CPU features 2 control flags that are used to suspend CPU operation in certain situations. These are a single bit each, and are interpreted as set when 1, and not set when 0. The meaning and use of these flags is as follows:

Halt flag

When set, it will halt CPU operation until next reset or power on.

Wait flag

When set, it will halt CPU operation until the next frame begins.

The programmer can set both of these flags for different purposes. Using the Wait flag allows programs to control their own execution speed. On the other hand, if a program has ended, it can just halt the CPU as an exit mechanism.

6 Responses to control signals

As with all components in the console, whenever a control signal is triggered the CPU will receive it and produce a response to process that event. For each of the control signals, the CPU will respond by performing the following actions:

Reset signal:

- The Halt and Wait flags are cleared to 0.
- All registers are cleared to integer value 0.
- The stack is reset by setting BP and SP to the last RAM address, i.e: 0x003FFFFF.
- Instruction pointer is set to point to the BIOS startup address, i.e: 0x10000004.

Frame signal:

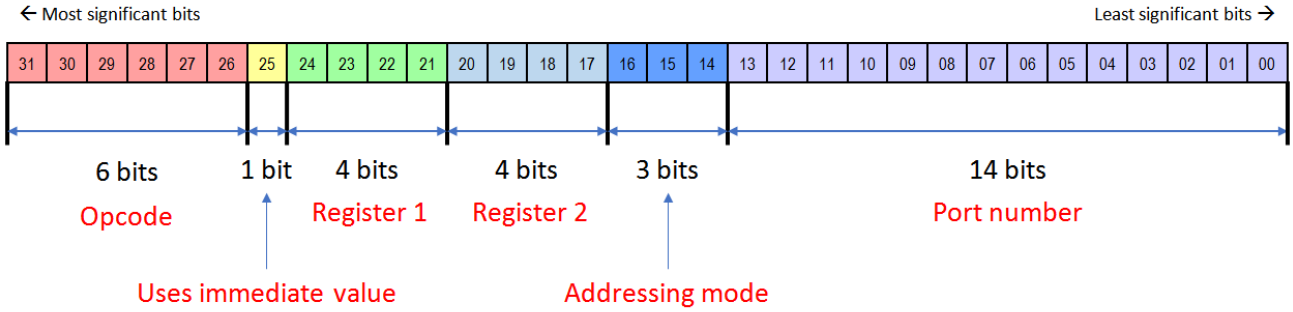
- The Wait flag is cleared to 0.

Cycle signal:

- If any of the control flags is set, the CPU will do nothing.
- Otherwise, the CPU performs a processing cycle as detailed in chapter 9.

7 Instruction format

CPU instructions are single 32-bit values that are interpreted by the CPU as a set of fields. This image shows the location and size of each field.



All of these fields are interpreted as unsigned integers. The fields “Register 1” and “Register 2”, when used by an instruction, designate a register index (0 to 15) within the array of general-purpose CPU registers.

If some field is not used by a particular instruction it should be set to zero, although this does not influence the correct processing of the instruction.

Some instructions will need the use of a second 32-bit word, called the immediate value. This is specified by setting the field called “Use immediate value” to 1. In that situation, a second word will be read from memory along with the instruction itself. This second value is stored in the Immediate Value register and will be used during instruction processing.

Note that a Vircon32 system is not required to detect ill-formed instructions. Processing any incorrect instruction will lead to implementation-dependent behavior.

8 Instruction set

The different instructions supported by the CPU are represented by an operation code (the opcode field in the instruction), which is a 6-bit unsigned integer. The Vircon32 CPU has exactly 64 different instructions, so there are no unused opcodes.

Many instructions used in Vircon32 CPU are based on x86 instructions, and they use the same names in most cases. Assembly instructions are represented using Intel syntax, both in this document and in the Vircon32 assembler program.

We can distinguish several different instruction groups attending to their purpose. The following tables list all 64 instructions grouped by their function. Note how the opcodes were also numbered taking those groups into account for easier understanding.

CPU control		
OpCode	Instruction	Short description
00	HLT	Halt CPU operation until reset
01	WAIT	Wait for next frame

Jump instructions		
OpCode	Instruction	Short description
02	JMP	Jump unconditionally
03	CALL	Call subroutine
04	RET	Return from subroutine
05	JT	Jump if True
06	JF	Jump if False

Integer comparisons		
OpCode	Instruction	Short description
07	IEQ	Integer Equal
08	INE	Integer Not Equal
09	IGT	Integer Greater Than
10	IGE	Integer Greater or Equal
11	ILT	Integer Less Than
12	ILE	Integer Less or Equal

Float comparisons		
OpCode	Instruction	Short description
13	FEQ	Float Equal
14	FNE	Float Not Equal
15	FGT	Float Greater Than
16	FGE	Float Greater or Equal
17	FLT	Float Less Than
18	FLE	Float Less or Equal

Data movement		
OpCode	Instruction	Short description
19	MOV	Move value
20	LEA	Load Effective Address of a memory position
21	PUSH	Push value on top of the stack
22	POP	Pop value from top of the stack
23	IN	Read value from an I/O port
24	OUT	Write value to an I/O port

String operations		
OpCode	Instruction	Short description
25	MOVS	Move string (Hardware memcpy)
26	SETS	Set string (Hardware memset)
27	CMPS	Compare string (Hardware memcmp)

Data conversion		
OpCode	Instruction	Short description
28	CIF	Convert Integer to Float
29	CFI	Convert Float to Integer
30	CIB	Convert Integer to Boolean
31	CFB	Convert Float to Boolean

Binary operations		
OpCode	Instruction	Short description
32	NOT	Bitwise NOT
33	AND	Bitwise AND
34	OR	Bitwise OR
35	XOR	Bitwise XOR
36	BNOT	Boolean NOT
37	SHL	Bit Shift Left

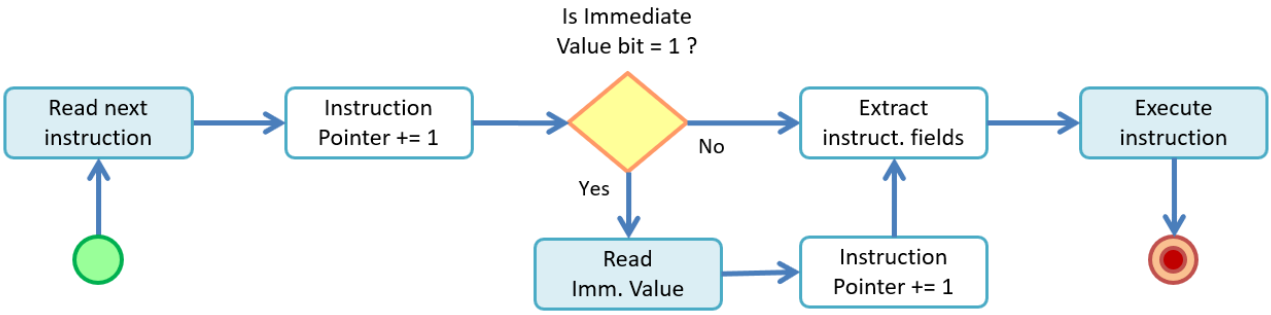
Integer arithmetic		
OpCode	Instruction	Short description
38	IADD	Integer Add
39	ISUB	Integer Subtract
40	IMUL	Integer Multiply
41	IDIV	Integer Divide
42	IMOD	Integer Modulus
43	ISGN	Integer Sign change
44	IMIN	Integer Minimum
45	IMAX	Integer Maximum
46	IABS	Integer Absolute value

Float arithmetic		
OpCode	Instruction	Short description
47	FADD	Float Add
48	FSUB	Float Subtract
49	FMUL	Float Multiply
50	FDIV	Float Divide
51	FMOD	Float Modulus
52	FSGN	Float Sign change
53	FMIN	Float Minimum
54	FMAX	Float Maximum
55	FABS	Float Absolute value

Extended float operations		
OpCode	Instruction	Short description
56	FLR	Floor (Round down)
57	CEIL	Ceiling (Round up)
58	ROUND	Round to nearest integer
59	SIN	Sine
60	ACOS	Arc Cosine
61	ATAN2	Arc Tangent from y and x
62	LOG	Natural Logarithm
63	POW	Raise to a Power

9 The CPU processing cycle

The CPU splits its program processing into cycles. The purpose of each cycle is to process a new, individual instruction from the program. A CPU processing cycle is described by the following algorithm:



The steps marked in blue are the ones where a hardware error might happen. In such case, this process would be aborted and the related error processing would take place. The actions from this processing cycle are performed as follows:

Read next instruction:

The CPU reads the memory address pointed by Instruction Pointer and stores its content in Instruction Register.

Read immediate value:

The CPU reads the memory address pointed by Instruction Pointer and stores its content in Immediate Value.

Extract instruction fields:

The value stored in Instruction Register is split into the different integer values that are encoded into the instruction bits. This step may not actually be needed if the implementation is able to work with the bit fields directly.

Execute instruction:

The CPU will choose the appropriate processing for the instruction depending on its opcode. Then it will apply one of the 64 possible cases detailed in the next section.

Note that the CPU must always honor the “Uses immediate value” field by using that bit to determine whether to read another word. This must be done even if the instruction is not well formed. Processing that kind of instruction will result in undefined behavior.

10 Instruction processing

This section provides a full specification of how each instruction is structured and its execution process. The following subsections cover all 64 possible opcodes.

It is important to note that, except for instruction MOV, all instructions have either 1 or 2 variants. In all cases, instructions with 2 variants determine which one is used by the value of the “uses immediate value” bit.

00 Instruction HLT (Halt)

Structure and variants:

HLT

Processing actions:

HaltFlag = 1

Description:

HLT activates the CPU’s Halt flag. This will cause the CPU to stop execution until the flag is cleared at the next console power on or reset. Note that other components will keep functioning: for instance, if the SPU was playing music it will continue to do so.

01 Instruction WAIT

Structure and variants:

WAIT

Processing actions:

WaitFlag = 1

Description:

WAIT activates the CPU’s Wait flag. This will cause the CPU to pause execution until the flag is cleared when the timer signals the start of the next frame. Reset or power-on actions will also resume CPU execution, since they also cause a new frame to begin.

When the new frame begins, the CPU will resume execution following the usual order, i.e. processing the instruction directly after WAIT. Note that other components will keep functioning.

02 Instruction JMP (Jump)

Structure and variants:

(Variant 1): JMP { ImmediateValue }
(Variant 2): JMP { Register1 }

Processing actions:

(Variant 1): InstructionPointer = ImmediateValue
(Variant 2): InstructionPointer = Register1

Description:

JMP performs an unconditional jump to the address specified by its operand. After processing this instruction the CPU will continue execution at the new address.

03 Instruction CALL

Structure and variants:

(Variant 1): CALL { ImmediateValue }
(Variant 2): CALL { Register1 }

Processing actions:

(Variant 1):
Stack.Push(InstructionPointer)
InstructionPointer = ImmediateValue
(Variant 2):
Stack.Push(InstructionPointer)
InstructionPointer = Register1

Description:

CALL performs a subroutine call to the specified address. When processing this instruction, the current Instruction Pointer (that was already incremented) will be saved to the top of the stack and then it will be overwritten with the new address. Execution will continue at the new address.

04 Instruction RET (Return)

Structure and variants:

RET

Processing actions:

InstructionPointer = Stack.Pop( )

Description:

RET returns from a previously called subroutine. When processing this instruction, the current Instruction Pointer will be overwritten with the topmost value in the stack. Execution will then continue at that previously saved address.

---

05 Instruction JT (Jump if True)

Structure and variants:

(Variant 1): JT { Register1 }, { ImmediateValue }
(Variant 2): JT { Register1 }, { Register2 }

Effect:

(Variant 1): if Register1 != 0 then InstructionPointer = ImmediateValue
(Variant 2): if Register1 != 0 then InstructionPointer = Register2

Description:

JT performs a jump only if its first operand is true, i.e. non zero when taken as an integer. In that case its behavior is the same as an unconditional jump. Otherwise it has no effect.

---

06 Instruction JF (Jump if False)

Structure and variants:

(Variant 1): JF { Register1 }, { ImmediateValue }
(Variant 2): JF { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 == 0 then InstructionPointer = ImmediateValue
(Variant 2): if Register1 == 0 then InstructionPointer = Register2

Description:

JF performs a jump only if its first operand is false, i.e. zero when taken as an integer. In that case its behavior is the same as an unconditional jump. Otherwise it has no effect.

---

07 Instruction IEQ (Integer Equal)

Structure and variants:

(Variant 1): IEQ { Register1 }, { ImmediateValue }
(Variant 2): IEQ { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 == ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 == Register2 then Register1 = 1 else Register1 = 0

Description:

IEQ takes two operands interpreted as integers, and checks if they are equal. It will store the boolean result in the first operand, which is always a register.

---

08 Instruction INE (Integer Not Equal)

Structure and variants:

(Variant 1): INE { Register1 }, { ImmediateValue }
(Variant 2): INE { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 != ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 != Register2 then Register1 = 1 else Register1 = 0

Description:

INE takes two operands interpreted as integers, and checks if they are different. It will store the boolean result in the first operand, which is always a register.

---

09 Instruction IGT (Integer Greater Than)

Structure and variants:

(Variant 1): IGT { Register1 }, { ImmediateValue }
(Variant 2): IGT { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 > ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 > Register2 then Register1 = 1 else Register1 = 0

Description:

IGT takes two operands interpreted as integers, and checks if the first one is greater than the second. It will store the boolean result in the first operand, which is always a register.

10 Instruction IGE (Integer Greater or Equal)

Structure and variants:

(Variant 1): IGE { Register1 }, { ImmediateValue }
(Variant 2): IGE { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 >= ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 >= Register2 then Register1 = 1 else Register1 = 0

Description:

IGE takes two operands interpreted as integers, and checks if the first one is greater or equal to the second. It will store the boolean result in the first operand, which is always a register.

11 Instruction ILT (Integer Less Than)

Structure and variants:

(Variant 1): ILT { Register1 }, { ImmediateValue }
(Variant 2): ILT { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 < ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 < Register2 then Register1 = 1 else Register1 = 0

Description:

ILT takes two operands interpreted as integers, and checks if the first one is less than the second. It will store the boolean result in the first operand, which is always a register.

12 Instruction ILE (Integer Less or Equal)

Structure and variants:

(Variant 1): ILE { Register1 }, { ImmediateValue }
(Variant 2): ILE { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 <= ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 <= Register2 then Register1 = 1 else Register1 = 0

Description:

ILE takes two operands interpreted as integers, and checks if the first one is less or equal to the second. It will store the boolean result in the first operand, which is always a register.

---

13 Instruction FEQ (Float Equal)

Structure and variants:

(Variant 1): FEQ { Register1 }, { ImmediateValue }
(Variant 2): FEQ { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 == ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 == Register2 then Register1 = 1 else Register1 = 0

Description:

FEQ takes two operands interpreted as floats, and checks if they are equal. It will store the boolean result in the first operand, which is always a register.

---

14 Instruction FNE (Float Not Equal)

Structure and variants:

(Variant 1): FNE { Register1 }, { ImmediateValue }
(Variant 2): FNE { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 != ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 != Register2 then Register1 = 1 else Register1 = 0

Description:

FNE takes two operands interpreted as floats, and checks if they are different. It will store the boolean result in the first operand, which is always a register.

---

15 Instruction FGT (Float Greater Than)

Structure and variants:

(Variant 1): FGT { Register1 }, { ImmediateValue }
(Variant 2): FGT { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 > ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 > Register2 then Register1 = 1 else Register1 = 0

Description:

FGT takes two operands interpreted as floats, and checks if the first one is greater than the second. It will store the boolean result in the first operand, which is always a register.

---

16 Instruction FGE (Float Greater or Equal)

Structure and variants:

(Variant 1): FGE { Register1 }, { ImmediateValue }
(Variant 2): FGE { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 >= ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 >= Register2 then Register1 = 1 else Register1 = 0

Description:

FGE takes two operands interpreted as integers, and checks if the first one is greater or equal to the second. It will store the boolean result in the first operand, which is always a register.

---

17 Instruction FLT (Float Less Than)

Structure and variants:

(Variant 1): FLT { Register1 }, { ImmediateValue }
(Variant 2): FLT { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 < ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 < Register2 then Register1 = 1 else Register1 = 0

Description:

FLT takes two operands interpreted as floats, and checks if the first one is less than the second. It will store the boolean result in the first operand, which is always a register.

18 Instruction FLE (Float Less or Equal)

Structure and variants:

(Variant 1): FLE { Register1 }, { ImmediateValue }
(Variant 2): FLE { Register1 }, { Register2 }

Processing actions:

(Variant 1): if Register1 <= ImmediateValue then Register1 = 1 else Register1 = 0
(Variant 2): if Register1 <= Register2 then Register1 = 1 else Register1 = 0

Description:

FLE takes two operands interpreted as floats, and checks if the first one is less or equal to the second. It will store the boolean result in the first operand, which is always a register.

19 Instruction MOV (Move)

Structure and variants:

(Variant 1): MOV { Register1 }, { ImmediateValue }
(Variant 2): MOV { Register1 }, { Register2 }
(Variant 3): MOV { Register1 }, [{ ImmediateValue }]
(Variant 4): MOV { Register1 }, [{ Register2 }]
(Variant 5): MOV { Register1 }, [{ Register2 } + { ImmediateValue }]
(Variant 6): MOV [{ ImmediateValue }], { Register2 }
(Variant 7): MOV [{ Register1 }], { Register2 }
(Variant 8): MOV [{ Register1 } + { ImmediateValue }], { Register2 }

Processing actions:

(Variant 1): Register1 = ImmediateValue
(Variant 2): Register1 = Register2
(Variant 3): Register1 = Memory[ImmediateValue]
(Variant 4): Register1 = Memory[Register2]
(Variant 5): Register1 = Memory[Register2 + ImmediateValue]
(Variant 6): Memory[ImmediateValue] = Register2
(Variant 7): Memory[Register1] = Register2
(Variant 8): Memory[Register1 + ImmediateValue] = Register2

Description:

MOV copies the value indicated in its second operand into the register or memory address indicated by its first operand. MOV is the most complex instruction to process because it needs to distinguish between 8 different addressing modes.

The instruction specifies which of the 8 modes to use in its "Addressing mode" field, being the possible values interpreted as follows:

MOV Addressing modes		
Value	Destination	Source
0	Register 1	Immediate Value
1	Register 1	Register 2
2	Register 1	Memory[Immediate Value]
3	Register 1	Memory[Register 2]
4	Register 1	Memory[Register 2 + Immediate Value]
5	Memory[Immediate Value]	Register 2
6	Memory[Register 1]	Register 2
7	Memory[Register 1 + Immediate Value]	Register 2

20 Instruction LEA (Load Effective Address)

Structure and variants:

(Variant 1): LEA { Register1 }, [{ Register2 }]
(Variant 2): LEA { Register1 }, [{ Register2 } + { ImmediateValue }]

Processing actions:

(Variant 1): Register1 = Register2
(Variant 2): Register1 = Register2 + ImmediateValue

Description:

LEA takes a memory address as second operand. It stores that address (not its contents) into the register given as first operand. The most useful case is when the address is given in the form pointer + offset, since the addition is automatically performed.

21 Instruction PUSH

Structure and variants:

PUSH { Register1 }

Processing actions:

Stack.Push(Register1)

Description:

PUSH uses the CPU hardware stack to add the value contained in the given register at the top of the stack.

22 Instruction POP

Structure and variants:

POP { Register1 }

Processing actions:

Register1 = Stack.Pop( )

Description:

POP uses the CPU hardware stack to remove a value from the top of the stack and write it in the given register.

23 Instruction IN

Structure and variants:

IN { Register1 }, { PortNumber }

Processing actions:

Register1 = Port[PortNumber]

Description:

IN uses the control bus to read from an I/O port in another chip and stores the returned value in the specified register. This read request may lead to side effects depending on the specified port.

24 Instruction OUT

Structure and variants:

(Variant 1): OUT { PortNumber }, [{ ImmediateValue }]
(Variant 2): OUT { PortNumber }, { Register1 }

Processing actions:

(Variant 1): Port[PortNumber] = ImmediateValue
(Variant 2): Port[PortNumber] = Register1

Description:

OUT uses the control bus to write the specified value to an I/O port in another chip. This write request may lead to side effects depending on the specified port.

25 Instruction MOVS (Move String)

Structure and variants:

MOVS

Processing actions:

```
Memory[ DR ] = Memory[ SR ]
DR += 1
SR += 1
CR –= 1
if CR > 0 then InstructionPointer –= 1
```

Description:

MOVS copies a value from the memory address pointed by SR to the one pointed by DR (as in a supposed MOV [DR], [SR]). It then implements a local loop to repeat itself until the counter in CR reaches 0, while working on consecutive addresses.

Note that even when called with a value of CR of zero or less, MOVS will always perform the described loop at least once.

This instruction is the only way in Vircon32 CPU to directly copy values from 2 places in memory without going through a register.

26 Instruction SETS (Set String)

Structure and variants:

SETS

Processing actions:

```
Memory[ DR ] = SR
DR += 1
CR –= 1
if CR > 0 then InstructionPointer –= 1
```

Description:

SETS copies the value in SR to the address pointed by DR (as in a MOV [DR], SR). It then implements a local loop to repeat itself until the counter in CR reaches 0, while writing to consecutive addresses.

Note that even when called with a value of CR of zero or less, SETS will always perform the described loop at least once.

27 Instruction CMPS (Compare String)

Structure and variants:

CMPS { Register1 }

Processing actions:

Register1 = Memory[DR] – Memory[SR]
if Register1 != 0 then end processing
DR += 1
SR += 1
CR –= 1
if CR > 0 then InstructionPointer –= 1

Description:

CMPS takes as a reference the compares the value in the address pointed by DR and compares it with the one pointed by SR, by subtracting. It then implements a local loop to repeat itself until the counter in CR reaches 0, while reading consecutive addresses.

The comparison result will be stored in the specified register, and will be zero when equal, positive when some value at [DR] was greater, and negative when some value in [SR] was greater.

Note that even when called with a value of CR of zero or less, CMPS will always perform the described loop at least once.

28 Instruction CIF (Convert Integer to Float)

Structure and variants:

CIF { Register1 }

Processing actions:

Register1 = (float)Register1

Description:

CIF interprets the specified register a as an integer value. Then converts that value to a float representation and stores the result back in the same register.

Note that, due to the limited precision of the float representation, high enough values of a 32-bit integer will result in a precision loss when represented as a float.

29 Instruction CFI (Convert Float to Integer)

Structure and variants:

CFI { Register1 }

Processing actions:

Register1 = (integer)Register1

Description:

CFI interprets the specified register a as a float value. Then converts that value to an integer representation and stores the result back in the same register. Conversion is not done through rounding, but instead by truncating (the fractional part is discarded).

Note that, due to the much greater range of the float representation, high enough values of a float will result in a precision loss when represented as a 32-bit integer.

30 Instruction CIB (Convert Integer to Boolean)

Structure and variants:

CIB { Register1 }

Processing actions:

if Register1 != 0 then Register1 = 1

Description:

CIB interprets the specified register a as an integer value. Then converts that value to its standard boolean representation and stores the result back in the same register. This means that all non-zero values will be converted to 1.

31 Instruction CFB (Convert Float to Boolean)

Structure and variants:

CFB { Register1 }

Processing actions:

if Register1 != 0.0 then Register1 = 1 else Register1 = 0

Description:

CFB interprets the specified register a as a float value. Then converts that value to either 0 (for float value 0.0), or 1 (for any other value) and stores it back in that register.

32 Instruction NOT

Structure and variants:

NOT { Register1 }

Processing actions:

Register1 = NOT Register1

Description:

NOT performs a binary 'not' by inverting all of the bits in the specified register.

33 Instruction AND

Structure and variants:

(Variant 1): AND { Register1 }, { ImmediateValue }
(Variant 2): AND { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = Register1 AND ImmediateValue
(Variant 2): Register1 = Register1 AND Register2

Description:

AND performs a binary 'and' between each pair of respective bits in the 2 specified operands. The result is stored in the first of them, which is always a register.

34 Instruction OR

Structure and variants:

(Variant 1): OR { Register1 }, { ImmediateValue }
(Variant 2): OR { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = Register1 OR ImmediateValue
(Variant 2): Register1 = Register1 OR Register2

Description:

OR performs a binary 'or' between each pair of respective bits in the 2 specified operands. The result is stored in the first of them, which is always a register.

35 Instruction XOR

Structure and variants:

(Variant 1): XOR { Register1 }, { ImmediateValue }
(Variant 2): XOR { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = Register1 XOR ImmediateValue
(Variant 2): Register1 = Register1 XOR Register2

Description:

XOR performs a binary 'exclusive or' between each pair of respective bits in the 2 specified operands. The result is stored in the first of them, which is always a register.

36 Instruction BNOT (Boolean NOT)

Structure and variants:

BNOT { Register1 }

Processing actions:

if Register1 == 0 then Register1 = 1 else Register1 = 0

Description:

BNOT interprets the specified register as a boolean and then converts it to the opposite boolean value. This is equivalent to first using CIB and then inverting bit number 0.

37 Instruction SHL (Shift Left)

Structure and variants:

(Variant 1): SHL { Register1 }, { ImmediateValue }
(Variant 2): SHL { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = Register1 << ImmediateValue
(Variant 2): Register1 = Register1 << Register2

Description:

SHL performs an bit shift to the left in the specified register. The second operand is taken as an integer number of positions to shift. Shifting 0 positions has no effect, while negative values result in shifting right. The shift type is logical: in shifts left, overflow is

discarded and zeroes are introduced as least significant bits. In shifts right, underflow is discarded and zeroes are introduced as most significant bits.

38 Instruction IADD (Integer Add)

Structure and variants:

(Variant 1): IADD { Register1 }, { ImmediateValue }
(Variant 2): IADD { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 += ImmediateValue
(Variant 2): Register1 += Register2

Description:

IADD interprets both of its operands as integers and performs an addition. The result is stored in the first operand, which is always a register. Overflow bits are discarded.

39 Instruction ISUB (Integer Subtract)

Structure and variants:

(Variant 1): ISUB { Register1 }, { ImmediateValue }
(Variant 2): ISUB { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 –= ImmediateValue
(Variant 2): Register1 –= Register2

Description:

ISUB interprets both of its operands as integers and performs a subtraction. The result is stored in the first operand, which is always a register. Overflow bits are discarded.

40 Instruction IMUL (Integer Multiply)

Structure and variants:

(Variant 1): IMUL { Register1 }, { ImmediateValue }
(Variant 2): IMUL { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 *= ImmediateValue
(Variant 2): Register1 *= Register2

Description:

IMUL interprets both of its operands as integers and performs a multiplication. The result is stored in the first operand, which is always a register. Overflow bits are discarded.

41 Instruction IDIV (Integer Divide)

Structure and variants:

(Variant 1): IDIV { Register1 }, { ImmediateValue }
(Variant 2): IDIV { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 /= ImmediateValue
(Variant 2): Register1 /= Register2

Description:

IDIV interprets both of its operands as integers and performs a division. The result is stored in the first operand, which is always a register.

42 Instruction IMOD (Integer Modulus)

Structure and variants:

(Variant 1): IMOD { Register1 }, { ImmediateValue }
(Variant 2): IMOD { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = Register1 mod ImmediateValue
(Variant 2): Register1 = Register1 mod Register2

Description:

IMOD interprets both of its operands as integers and performs a division. The remainder of that division is stored in the first operand, which is always a register.

43 Instruction ISGN (Integer Sign change)

Structure and variants:

ISGN { Register1 }

Processing actions:

Register1 = –Register1

Description:

ISGN interprets the operand register as an integer and inverts its sign.

44 Instruction IMIN (Integer Minimum)

Structure and variants:

(Variant 1): IMIN { Register1 }, { ImmediateValue }
(Variant 2): IMIN { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = min(Register1, ImmediateValue)
(Variant 2): Register1 = min(Register1, Register2)

Description:

IMIN interprets both of its operands as integers. It then takes the minimum of both values and stores it in the first operand, which is always a register.

45 Instruction IMAX (Integer Maximum)

Structure and variants:

(Variant 1): IMAX { Register1 }, { ImmediateValue }
(Variant 2): IMAX { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = max(Register1, ImmediateValue)
(Variant 2): Register1 = max(Register1, Register2)

Description:

IMAX interprets both of its operands as integers. It then takes the maximum of both values and stores it in the first operand, which is always a register.

46 Instruction IABS (Integer Sign change)

Structure and variants:

IABS { Register1 }

Processing actions:

Register1 = abs(Register1)

Description:

IABS interprets the operand register as an integer and takes its absolute value.

47 Instruction FADD (Float Add)

Structure and variants:

(Variant 1): FADD { Register1 }, { ImmediateValue }
(Variant 2): FADD { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 += ImmediateValue
(Variant 2): Register1 += Register2

Description:

FADD interprets both of its operands as floats and performs an addition. The result is stored in the first operand, which is always a register.

48 Instruction FSUB (Float Subtract)

Structure and variants:

(Variant 1): FSUB { Register1 }, { ImmediateValue }
(Variant 2): FSUB { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 –= ImmediateValue
(Variant 2): Register1 –= Register2

Description:

FSUB interprets both of its operands as floats and performs a subtraction. The result is stored in the first operand, which is always a register.

49 Instruction FMUL (Float Multiply)

Structure and variants:

(Variant 1): FMUL { Register1 }, { ImmediateValue }
(Variant 2): FMUL { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 *= ImmediateValue
(Variant 2): Register1 *= Register2

Description:

FMUL interprets both of its operands as floats and performs a multiplication. The result is stored in the first operand, which is always a register.

50 Instruction FDIV (Float Divide)

Structure and variants:

(Variant 1): FDIV { Register1 }, { ImmediateValue }
(Variant 2): FDIV { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 /= ImmediateValue
(Variant 2): Register1 /= Register2

Description:

FDIV interprets both of its operands as floats and performs a division. The result is stored in the first operand, which is always a register.

51 Instruction FMOD (Float Modulus)

Structure and variants:

(Variant 1): FMOD { Register1 }, { ImmediateValue }
(Variant 2): FMOD { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = fmod(Register1, ImmediateValue)
(Variant 2): Register1 = fmod(Register1, Register2)

Description:

FMOD interprets both of its operands as floats and performs a division. It then takes the remainder of that division when the result's fractional part is discarded and stores it in the first operand, which is always a register.

52 Instruction FSGN (Float Sign change)

Structure and variants:

FSGN { Register1 }

Processing actions:

Register1 = –Register1

Description:

FSGN interprets the operand register as a float and inverts its sign.

53 Instruction FMIN (Float Minimum)

Structure and variants:

(Variant 1): FMIN { Register1 }, { ImmediateValue }
(Variant 2): FMIN { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = min(Register1, ImmediateValue)
(Variant 2): Register1 = min(Register1, Register2)

Description:

FMIN interprets both of its operands as floats. It then takes the minimum of both values and stores it in the first operand, which is always a register.

54 Instruction FMAX (Float Maximum)

Structure and variants:

(Variant 1): FMAX { Register1 }, { ImmediateValue }
(Variant 2): FMAX { Register1 }, { Register2 }

Processing actions:

(Variant 1): Register1 = max(Register1, ImmediateValue)
(Variant 2): Register1 = max(Register1, Register2)

Description:

FMAX interprets both of its operands as floats. It then takes the maximum of both values and stores it in the first operand, which is always a register.

55 Instruction FABS (Float Sign change)

Structure and variants:

FABS { Register1 }

Processing actions:

Register1 = abs(Register1)

Description:

FABS interprets the operand register as a float and takes its absolute value.

56 Instruction FLR (Floor)

Structure and variants:

FLR { Register1 }

Processing actions:

Register1 = floor(Register1)

Description:

FLR interprets the operand register as a float and rounds it downwards to an integer value. Note that the result is not converted to an integer, but is still a float.

57 Instruction CEIL (Ceiling)

Structure and variants:

CEIL { Register1 }

Processing actions:

Register1 = ceil(Register1)

Description:

CEIL interprets the operand register as a float and rounds it upwards to an integer value. Note that the result is not converted to an integer, but is still a float.

58 Instruction ROUND

Structure and variants:

ROUND { Register1 }

Processing actions:

Register1 = round(Register1)

Description:

ROUND interprets the operand register as a float and rounds it to the closest integer value. Note that the result is not converted to an integer, but is still a float.

59 Instruction SIN (Sine)

Structure and variants:

SIN { Register1 }

Processing actions:

Register1 = sin(Register1)

Description:

SIN interprets the operand register as a float and calculates the sine of that value. The sine function will interpret its argument in radians.

60 Instruction ACOS (Arc Cosine)

Structure and variants:

ACOS { Register1 }

Processing actions:

Register1 = acos(Register1)

Description:

ACOS interprets the operand register as a float and calculates the arc cosine of that value. The result is given in radians, in the range [0, pi].

61 Instruction ATAN2 (Arc Tangent 2)

Structure and variants:

ATAN2 { Register1 }, { Register2 }

Processing actions:

Register1 = atan2(Register1, Register2)

Description:

ATAN2 interprets both operand registers as floats and calculates the angle of a vector such that Vx = Register2 and Vy = Register1. The result is stored in the first operand register and will be given in radians, in the range [-pi, pi]. The origin of angles is located at (Vx > 0, Vy = 0) and angles grow when rotating towards (Vx = 0, Vy > 0).

62 Instruction LOG (Logarithm)

Structure and variants:

LOG { Register1 }

Processing actions:

Register1 = log(Register1)

Description:

LOG interprets the operand register as a float and calculates the logarithm base e of that value.

63 Instruction POW (Power)

Structure and variants:

POW { Register1 }, { Register2 }

Processing actions:

Register1 = pow(Register1, Register2)

Description:

POW interprets both operand registers as floats and calculates the result of raising the first operand to the power of the second operand. The result is stored in the first operand register.

11 Detecting hardware errors

The execution of a program can encounter some situations that the hardware is unable to process, such as a division by zero. In Vircon32 those situations are always reached as a consequence of CPU operation, so the processor must the component responsible for detecting error situations and triggering the corresponding hardware error mechanism.

The following subsections list all the possible situations where a hardware error might occur, and explains the different criteria the CPU will use to detect each type of error.

11.1 Bus errors

These are the possible errors that may happen when the CPU makes a request to one of the 2 communication buses. Their detection by the CPU is automatic, since the bus itself will respond to the request with a success/failure indicator.

Invalid memory read

A read request to the memory bus has failed.

Invalid memory write

A write request to the memory bus has failed.

Invalid port read

A read request to the control bus has failed.

Invalid port write

A write request to the control bus has failed.

11.2 Stack errors

In its normal operation, the Stack Pointer should only point to valid address positions within RAM memory. That range is between 0x00000000 and 0x003FFFFF. Program errors or an abnormally large call stack can cause situations where the stack is not able to keep functioning normally, and the CPU will trigger a hardware error as a response. There can be the following 2 error situations:

Stack overflow

After a push operation, if SP becomes negative it is considered that a stack overflow happened. At this point the stack has grown down to the end of RAM memory and cannot keep growing.

Stack underflow

After a pop operation, if SP becomes greater than its initial value of 0x003FFFFF (last word in RAM memory) it is considered that a stack underflow happened. At this point the stack cannot keep decreasing since it has been depleted.

11.3 Mathematical errors

In some of the CPU instructions that perform mathematical operations, there is a common cause for error when the received arguments fall outside of the function domain. When this kind of situation is detected, instruction processing will be aborted to trigger the corresponding error. The different cases that may be found where a mathematical function is not defined are the following:

Division error

This error can happen in 4 different instructions: division and modulus, in both of their integer and float versions. It happens when the second argument (the divisor) is zero.

Arc Cosine error

This instruction cannot be processed if its argument is not between -1.0 and +1.0 (both ends are included in the valid range).

Arc Tangent 2 error

This instruction cannot be processed when both received arguments are zero.

Logarithm error

This instruction cannot be processed if its received argument is less or equal to zero.

Power error

This instruction cannot be processed in the case where the first argument (base) is negative, and the second (exponent) has not an integer value.

12 Processing hardware errors

When the CPU detects one of the situations described in the previous section it will need to trigger the corresponding hardware error processing. The last stages of the error processing will be done by the BIOS error handler routine, but before that routine can be called, the CPU will need to prepare some information for it.

The preparations needed for the BIOS to process an error consist of identifying the type of error with an error code, and saving the internal registers.

12.1 Error codes

When a hardware error happens, the type of error situation that was detected is represented by a numeric code. The possible error codes are listed in this table:

Hardware error codes	
Code	Error type
0	Invalid memory read
1	Invalid memory write
2	Invalid port read
3	Invalid port write
4	Stack overflow
5	Stack underflow
6	Division error
7	Arc Cosine error
8	Arc Tangent 2 error
9	Logarithm error
10	Power error

12.2 CPU response to an error

When the CPU triggers a hardware error, it will take the following actions:

- Write the error code in R0.
- Copy Instruction Pointer into R1.
- Copy Instruction Register into R2.
- Copy Immediate Value into R3.
- Set BP and SP to the last position in RAM (i.e. address 0x003FFFFF).
- Set Instruction Pointer to 0x10000000 (i.e. BIOS error handler address).
- Resume normal execution from that address.

After that the CPU will just keep executing the BIOS error handler until it eventually finishes its tasks, which are displaying error information on screen and halting execution.

Note that the stack is reset, discarding any previous call contexts. This is needed since the error may have been caused by the stack itself. Also, the previous context will become unnecessary anyway after the BIOS gains control.

(End of part 3)