

Zylios CPU 10 Documentation Handbook

Black Phoenix

July 24, 2010

Contents

1	Processor Features	5
1.1	General Purpose Registers	5
1.2	Instruction Pointer	5
1.3	Segment Registers	6
1.4	Memory Models	7
1.4.1	Linear Address Mode	8
1.4.2	Segmented Address Mode	8
1.4.3	Mapped Memory Mode	8
1.5	Stack	9
1.6	Branching	10
1.7	Processor Error Codes	13
1.7.1	End of program execution (02)	14
1.7.2	Division By Zero (03)	15
1.7.3	Stack Error (06)	15
1.7.4	Memory Read/Write Fault (07)	16
1.7.5	MemBus Fault (08)	17
1.7.6	Port Read/Write Fault (10)	17
2	Advanced Processor Features	18
2.1	Paging System	18
2.2	Internal ROM	20
2.3	Page Table Control Opcodes	21
2.4	Bitwise Operations	21
2.5	Memory Blocks Support	21

2.6	Copying, Shifting, Swapping Large Blocks of Data	21
2.7	Stack Frame Support	21
2.8	Interrupts/Extended Mode	21
2.9	Non-Masked Interrupts	22
2.10	Memory Access Overriding	22
2.11	Internal Timer	22
2.12	Vector Extension	22
2.13	Hardware Debug Mode	22
3	Compatibility	23
3.1	Instructions	23
3.2	Processor Modes	23
3.3	Instruction Format	23
4	Instruction Format	24
4.1	Format	24
4.2	Register-Memory Modifier Byte	24
4.3	Segment Offset	24
4.4	Immediate Value	24
5	Internal Registers	25
6	Instruction Set Reference	28
6.1	ZCPU Opcode Set	28
6.2	Vector Extension Opcode Set	33
7	HL-ZASM	34
7.1	Assembler Syntax	34
7.2	Expression Generator	34
7.3	C Syntax Support	34
7.4	Special Features	34

Chapter 1

Processor Features

1.1 General Purpose Registers

There are 8 general purpose registers in the ZCPU. These registers are **EAX**, **EBX**, **ECX**, **EDX**, **ECX**, **EDX**, **ESP** and **EBP**. You can use any of these for any purpose, apart from the **ESP** register, which is reserved as the stack pointer (see 1.5 for more information on the processor stack).

Every register is a single 64-bit floating point value. It can contain any value between -10^{3000} and 10^{3000} , but the precision in digits is limited to 48 bits.

The ZCPU is capable of treating registers as 48-bit binary values, ignoring the exponent.

After every CPU reset, all registers except for **ESP** will be set to 0. **ESP** will be reset to CPU ROM size minus one (65535 by default).

1.2 Instruction Pointer

Zyelios CPU has a special register called the instruction pointer, or **IP**. **IP** points to the currently executing instruction and is incremented every time a new instruction is read.

Instructions may have variable size, so **IP** may increment in different steps. It is possible to set **IP** explicitly by using any of the branching opcodes

(see 1.6)

1.3 Segment Registers

There are 8 segment registers in the ZCPU. They are CS, SS, DS, ES, GS, FS, KS and LS.

Every segment register is a 48-bit address offset. This offset can be used for specifying an address offset for a block of data or code. The ZCPU always uses segments when referencing memory to translate local address (the address requested by user) into absolute address.

The formula used for address translation is:

$$\textit{AbsoluteAddress} = \textit{LocalAddress} + \textit{SegmentOffset}$$

The user can specify which register to use for segment offset by prefixing operand with segment name followed by a semicolon. If a segment is not specified, the processor will use DS.

You can use both segment and general purpose registers for segment prefixing. It is not possible to use a constant value for segment prefix. Here is an example of various syntaxes for reading memory:

```
MOV EAX,#EBX      //Address: DS+EBX
MOV EAX,ES:#EBX   //Address: ES+EBX
MOV EAX,[EBX]     //Address: DS+EBX
MOV EAX,[ES:EBX]  //Address: ES+EBX
MOV EAX,[ES+EBX]  //Address: ES+EBX

MOV EAX,EBX:ECX   //Address: EBX+ECX
MOV EAX,[EBX+ECX] //Address: EBX+ECX
MOV EAX,[EBX:100] //Address: EBX+100
MOV EAX,[100:EBX] //This is not a valid segment prefix syntax
```

With some clever programming the segment prefixes can be used for quick array access.

Some of the segment registers are used by the processor for specific tasks, as per the following table:

Register	Name	Description
CS	Code segment	Processor fetches code from this segment
SS	Data segment	Default segment for data
DS	Stack segment	Processor stack is located in this segment
ES	Extra segment	User segment
GS	G segment	User segment
FS	F segment	User segment
KS	Key segment	User segment
LS	Library segment	User segment

You can set all segment registers except CS with MOV. The only way to modify CS is to execute CALLF or JMPF (see 1.6 for more information).

Attempting to set CS will trigger interrupt 13:1 (general processor fault). The following is an example of this:

```
MOV DS,100
MOV ES,KS
MOV CS,1000 //Will generate interrupt 13:1
```

After CPU reset all segment registers will be initialized to 0.

1.4 Memory Models

The ZCPU is capable of working with several different memory modes. Linear addressing mode is the default mode, and it is the mode in use when the CPU is reset.

Different modes require different use of registers, and they provide different execution features. Each mode will usually require extended processor mode to be active, enabling some advanced memory protection features such as page permissions and memory mapping.

1.4.1 Linear Address Mode

This is the default address mode, when all segment registers are initialized to 0. In this mode the code, data, and stack are all located in same address space. In this address mode no segment prefixes are required. For example:

```
MOV EAX,#0 //EAX will be equal to 14 (MOV opcode no)
MOV #ESP,100 //Same as PUSH 100
DEC ESP
POP EAX //EAX will be equal to 100
```

1.4.2 Segmented Address Mode

This is the most common addressing mode, when segment registers are used differently. For example, the code, data and stack might all be located in different areas of memory. This has certain benefits:

- Allows to prevent accidental data/code corruption
- Same code can be used for different blocks of data
- Programs can run in local address space, not aware of different programs (for example BIOS)

Example:

```
MOV DS,1000 //Set first data block
MOV SS,2000 //Set first stack block
CALLF 0,500 //Call the routine (CS will be set to 500, IP to 0)

MOV DS,3000 //Set second data block
MOV SS,4000 //Set second stack block
CALLF 0,500 //Call the same routine, but now working on second data block
```

1.4.3 Mapped Memory Mode

Mapped memory mode uses the memory mapping features of the processor to reroute memory addresses in order to create the appearance of a single

continuous address space for the user program. It allows user programs to dynamically allocate blocks of data, which can themselves be physically located in different areas of memory. This also allows use of dynamic libraries, where single library loaded once can be used in many programs while remaining in a single place in physical memory.

See 2.1 for more information.

FIXME: add code examples

1.5 Stack

The Zyelios CPU has a built-in hardware processor stack. Stack operation is controlled by the current stack pointer register (ESP), the stack size register (ESZ) and the stack segment register (SS) Stack data is located in physical RAM.

ESP points to the *next free value on the stack*. The stack grows *down*.

You can use PUSH and POP to push values to, or pop values from, the stack. Stack overflow or underflow will be indicated by the interrupt 6:ESP (stack error). The interrupt parameter will be set to the value of ESP. For example:

```
MOV SS,5000 //Stack starts at offset 5000
MOV ESP,2999 //Stack is 3000 bytes in size
           //Next free offset in segment is 2999
CPUSET 9,3000 //Set ESZ register

PUSH 200
PUSH 100
POP EAX //EAX = 100
POP EBX //EBX = 200

//PUSH X is same as the following (but with error checks):
MOV SS:#ESP,X
DEC ESP
```

```
//POP Y is same as the following (but with error checks):  
INC ESP  
MOV Y,SS:#ESP
```

Additionally, there are `RSTACK` and `SSTACK` instruction which allow the user to read or write any value off the stack. These instructions may also trigger stack underflow or overflow interrupts. Consult the following example for an explanation:

```
RSTACK X,Y //X = MEMORY[SS+Y]  
SSTACK X,Y //MEMORY[SS+X] = Y  
  
RSTACK EAX,ESP:1 //Read stack top  
RSTACK EAX,ESP:2 //Read value under stack top  
  
PUSH 100 //  
PUSH 200 //Value under top value  
PUSH 300 //Value on top  
  
SSTACK ESP:2,123 //Set value under stack top  
POP EAX //EAX = 300  
POP EBX //EBX = 123  
POP ECX //ECX = 100
```

1.6 Branching

The Zyelios CPU supports various kinds of branching: conditional or unconditional, absolute or relative. The instruction pointer register, `IP`, points to the currently executing instruction. All branching instructions modify `IP`, and some of them modify `CS` (code segment, see 1.3)

The simplest type of branching is absolute unconditional. To perform an unconditional jump you can use the `JMP` opcode or the `JMPF` opcode (the latter of which also modifies `CS`).

You can call a program routine by using `CALL` or `CALLF`. This will save current instruction pointer (and `CS` when `CALLF` is used) to the processor stack. The instruction pointer (and code segment if required) can be restored from stack by using `RET` or `RETF` opcode accordingly.

See example for creating routines and jumping around:

```
JMPF MAIN, CODE_SEGMENT //Syntax is JMPF IP, CS
```

```
.....
```

```
MAIN: //A label  
    CALL SUBROUTINE  
    JMP EXIT
```

```
SUBROUTINE:  
    CALL SUBROUTINE2  
RET //Exit subroutine
```

```
SUBROUTINE2: //Called inside SUBROUTINE  
    ... do something ...
```

```
RET
```

```
.....
```

```
EXIT:
```

It is possible to perform a relative jump. To do this, you need to use `JMPR` (jump relative) instruction. It will increment `IP` by a certain amount of bytes. See example:

```
JMPR +10 //Jump 10 bytes forward  
JMPR -10 //Jump 10 bytes backward
```

```
JMPR LABEL-__PTR__ //Jump to label  
                    //__PTR__ label always points to current write pointer
```

.....

LABEL :

Conditional branching allows program flow to be changed depending on certain conditions. You need to use comparison instruction **CMP** to perform branch test, and then use one of the following opcodes to change program flow depending on the branch test result:

Instruction	Operation	Description
JNE	$X \neq Y$	Jump if not equal
JNZ	$X - Y \neq 0$	Jump if not zero
JG	$X > Y$	Jump if greater than
JNLE	$\text{NOT } X \leq Y$	Jump if not less or equal
JGE	$X \geq Y$	Jump if greater or equal
JNL	$\text{NOT } X < Y$	Jump if not less than
JL	$X < Y$	Jump if less than
JNGE	$\text{NOT } X \geq Y$	Jump if not greater or equal
JLE	$X \leq Y$	Jump if less or equal
JNG	$\text{NOT } X > Y$	Jump if not greater than
JE	$X = Y$	Jump if equal
JZ	$X - Y = 0$	Jump if zero
CNE	$X \neq Y$	Call if not equal
CNZ	$X - Y \neq 0$	Call if not zero
CG	$X > Y$	Call if greater than
CNLE	$\text{NOT } X \leq Y$	Call if not less or equal
CGE	$X \geq Y$	Call if greater or equal
CNL	$\text{NOT } X < Y$	Call if not less than
CL	$X < Y$	Call if less than
CNGE	$\text{NOT } X \geq Y$	Call if not greater or equal
CLE	$X \leq Y$	Call if less or equal
CNG	$\text{NOT } X > Y$	Call if not greater than
CE	$X = Y$	Call if equal
CZ	$X - Y = 0$	Call if zero
JNER	$X \neq Y$	Jump relative if not equal
JNZR	$X - Y \neq 0$	Jump relative if not zero
JGR	$X > Y$	Jump relative if greater than
JNLER	$\text{NOT } X \leq Y$	Jump relative if not less or equal
JGER	$X \geq Y$	Jump relative if greater or equal

JNLR	NOT $X < Y$	Jump relative if not less than
JLR	$X < Y$	Jump relative if less than
JNGER	NOT $X \geq Y$	Jump relative if not greater or equal
JLER	$X \leq Y$	Jump relative if less or equal
JNGR	NOT $X > Y$	Jump relative if not greater than
JER	$X = Y$	Jump relative if equal
JZR	$X - Y = 0$	Jump relative if zero

There are other instruction which perform branch testing, such as the BIT, which allows you to check whether certain bit of a value is set. For example:

```

CMP EAX,EBX
JG LABEL1 //Jump if EAX > EBX
JLE LABEL2 //Jump if EAX <= EBX
JE LABEL3 //Jump if EAX = EBX
CL LABEL4 //Call if EAX < EBX
CGE LABEL5 //Call if EAX >= EBX

BIT EAX,4 //Test 5th bit of EAX
JZ LABEL1 //Jump if 5th bit is 0
JNZ LABEL1 //Jump if 5th bit is 1

```

1.7 Processor Error Codes

There are several error codes that might be generated during program execution. If processor is not in the extended mode, the execution will halt when error is generated. If processor is in the extended mode, the corresponding interrupt will be called.

Upon encountering an error, the processor will halt and output the error code to **ERROR** processor output. It will also emit a secondary error code in the fraction part of the **ERROR** processor output.

For example, a typical output might be 7.65536, which would indicate a memory read/write fault at address 65536.

The error code will be reset to zero when CPU is reset.

Code	Description
02	End of program execution
03	Division by zero
04	Unknown opcode
05	Internal processor error
06	Stack error (overflow/underflow)
07	Memory read/write fault
08	MemBus fault
09	Write access violation (page protection)
10	Port read/write fault
11	Page access violation (page protection)
12	Read access violation (page protection)
13	General processor fault
14	Execute access violation (page protection)
15	Address space violation

1.7.1 End of program execution (02)

Error message: STOP detected

Occurs when: STOP/OPCODE 0 is executed

Cause: Abnormal program end

Result: None

Error message: Invalid opcode

Occurs when: Any of the branching instructions is executed (JMP, CALL, etc)

Cause: Jumping by offset that does not point to valid ZCPU instruction

Result: None

1.7.2 Division By Zero (03)

Error message: Unable to divide by zero

Occurs when: Second operand for DIV opcode is zero

Cause: User error

Result: None

Error message: Unable to inverse zero

Occurs when: Calling INV opcode with zero operand

Cause: User error

Result: None

1.7.3 Stack Error (06)

Error message: Stack overflow error

Occurs when: PUSH is executed

Cause: ESP register value becomes negative

Result: LADD = 0

Error message: Stack underflow error

Occurs when: POP is executed

Cause: ESP register is greater than ESZ

Result: LADD = ESZ

Error message: Stack read error

Occurs when: POP is executed

Cause: Unable to read value from memory

Result: LADD = ESP

Error message: Stack out of bounds error

Occurs when: RSTACK or SSTACK is executed

Cause: Requested value out of stack bounds

Result: LADD equals to requested index on stack

1.7.4 Memory Read/Write Fault (07)

Error message: Read error: address does not exist

Occurs when: Processor attempts to read a value from outside of the internal RAM

Cause: No device is attached to the MemBus

Result: LADD equals to faulty address in memory

Error message: Read error: unable to read memory location

Occurs when: Processor attempts to read a value from outside of the internal RAM

Cause: A failure has occurred while attempting to read a value (value out of the device address range)

Result: LADD equals to faulty address in memory

Error message: Write error: address does not exist

Occurs when: Processor attempts to write a value to outside of the internal RAM

Cause: No device is attached to the MemBus

Result: LADD equals to faulty address in memory

Error message: Write error: unable to write to memory location

Occurs when: Processor attempts to write a value to outside of the internal RAM

Cause: A failure has occurred while attempting to write a value (value out of the device address range)

Result: LADD equals to faulty address in memory

1.7.5 MemBus Fault (08)

Error message: MemBus device error

Occurs when: Processor attempts to read a value from outside of the internal RAM

Cause: Device currently attached to MemBus does not support hispeed interface

Result: LADD equals to faulty address in memory

Error message: IOBus device error

Occurs when: Processor attempts to read a value from a port

Cause: Device currently attached to IOBus does not support hispeed interface

Result: LADD = -PORT_NUMBER

1.7.6 Port Read/Write Fault (10)

Error message: Read error: unable to read a port

Occurs when: Processor attempts to read a value from a port

Cause: A failure has occurred while attempting to read a value (value out of the device address range)

Result: LADD equals to faulty address in memory

Error message: Write error: unable to write to a port

Occurs when: Processor attempts to write a value to a port

Cause: A failure has occurred while attempting to write a value (value out of the device address range)

Result: LADD equals to faulty address in memory

Chapter 2

Advanced Processor Features

2.1 Paging System

The Zyelios CPU divides the entire accessible memory space into pages. Every page is 128 bytes in size. Pages are numbered sequentially; This means that addresses 0, 1, ... 127 belong to page 0, addresses 128 .. 255 belong to page 1, 256 .. 383 belong to page 2 and so on.

Every page can have a separate permission mask (read permission, write permission, execute permission) and runlevel.

Runlevel is a number That can be used to divide code into high-permission (kernel) and low-permission (application) parts. The higher runlevel is, the lower permissions thread has. The runlevel value must be 0 through 255 inclusive.

Runlevel 0 is a special runlevel which will ignore all permission settings. All others runlevels follow page permissions.

When a page is marked non-readable, it can only be read from a page with smaller runlevel. When a page is marked non-writeable it can only be written from a page with smaller runlevel.

If the page is marked non-executable, then only pages with smaller runlevel can branch into this page. This means that if the runlevel of the currently executing page is smaller than the target page one can `JMP` or `CALL` into the target page.

Every page can also be remapped to any other page in memory. This means that every time the processor accesses target page, the data will be read from a different physical address. This can be used for moving resources around without applications noticing it.

Page mapping and permissions settings are stored in the *page table*. The page table is active when the processor is in extended mode (**EF** is set to 1). If **MF** is set to 0 the page table will be stored internally; if it is set to 1 the page table will be stored in RAM.

The **PTBL** internal register is an *absolute* pointer to the table start, and **PTBE** holds the number of entries in the table. It is possible to switch page tables during execution.

Every page table entry is 2 bytes in size. The first entry (entry number 0) is the default page. All other pages in the table correspond to memory pages.

If the address being accessed is not covered by any page listed in the page table, it will use the permissions set on the default page entry. You cannot perform memory mapping on pages which do not have entry in page table.

The first byte of every entry holds the runlevel and permission flags. The second byte holds index of page this page should be remapped to. Please note that permission flags in the page table entry are *inverted*; that is, 1 means permission is restricted, 0 means that it is not.

Here are all the bits in the first byte:

Bit	Description
00	Is page disabled? Set to 1 to disable this page
01	Is page remapped? Set to 1 to remap this page
02	Page must generate NMI 30 (page trap) upon access
03	Page trap overrides read/write
04	Reserved
05	Read permissions (0: allowed, 1: disabled)
06	Write permissions (0: allowed, 1: disabled)
07	Execute permissions (0: allowed, 1: disabled)
08 . . 15	Runlevel

Disabled pages will cause a memory fault when a read is attempted to

any address in its range (as if page wasn't in the address space). Remapped pages will use the second byte in the entry as the index for the *physical* page this page must be remapped to.

It is possible to trap page accesses by setting bit 2. This will cause any access to this page to generate NMI 28, 29, or 30 with the page number being accessed as the parameter. If bit 2 is set, then result of the memory read/write access should be overridden by trap interrupt (see 2.10 for more info)

There are special rules when handling interrupts. Permissions to call the interrupts are defined by runlevel of the interrupt table. This means you can restrict the user program from calling certain interrupts, but the user program would be able to call lower runlevel code using other (non-restricted) interrupts. You can imagine it as if you were **CALLing** your interrupt routine from the interrupt table (and not from user program). This is the way you can protect your kernel code from reading/writing/executing while retaining the ability to run the code (via calling interrupts).

2.2 Internal ROM

Zyelios CPU may have built-in ROM and RAM. This allows you to store some program code right on the processor chip. The contents of ROM will be restored into RAM every time the processor is reset, but it is possible to read and write ROM from software.

The ROM cannot be written under regular circumstances, but it is possible to overwrite certain portions of the ROM, or erase them.

There are three opcodes to work with internal ROM: **ERPG** (erase ROM page), **WRPG** (write ROM page), **RDPG** (read ROM page).

See example of use:

```
ERPG 4 //Erase some data from ROM
WRPG 4 //Write this block of data to ROM
      //After CPU reset it will be still preserved
RDPG 4 //Restore this page from ROM
```

```
ORG 512 //Put on page 4
SOME_AREA:
    ... some data ...
```

2.3 Page Table Control Opcodes

(SPG,CPG, etc)

2.4 Bitwise Operations

(BIT, SBIT, TBIT, CBIT)

2.5 Memory Blocks Support

(support for BLOCK opcode)

2.6 Copying, Shifting, Swapping Large Blocks of Data

(MCPY, MSHIFT, MXCHG)

2.7 Stack Frame Support

(using EBP register, and ENTER/LEAVE opcodes)

2.8 Interrupts/Extended Mode

(using and handling interrupts)

2.9 Non-Masked Interrupts

(NMI stuff)

2.10 Memory Access Overriding

(using page traps to override access to specific memory areas)

2.11 Internal Timer

(using internal CPU timer)

2.12 Vector Extension

(using VMODE,VADD,etc)

2.13 Hardware Debug Mode

(Using hardware debug mode)

Chapter 3

Compatibility

3.1 Instructions

3.2 Processor Modes

3.3 Instruction Format

Chapter 4

Instruction Format

4.1 Format

4.2 Register-Memory Modifier Byte

4.3 Segment Offset

4.4 Immediate Value

Chapter 5

Internal Registers

The processor has several internal registers which are used to store the internal processor state or control advanced processor features. It is possible to read or write most of the internal registers using CPUSET and CPUGET.

For example:

```
CPUGET EAX,24 //Read register 24 into EAX (interrupt descriptor table pointer)
CPUSET 9,EBX //Set register 9 (stack size) to EBX
```

```
CPUGET EAX,1000 //Invalid register will set EAX to 0
```

Mnemonic	Number	Description
IP	00	Instruction pointer
EAX	01	General purpose register A
EBX	02	General purpose register B
ECX	03	General purpose register C
EDX	04	General purpose register D
ESI	05	Source index
EDI	06	Destanation index
ESP	07	Stack pointer
EBP	08	Base pointer
ESZ	09	Stack size
CS	16	Code segment
SS	17	Stack segment
DS	18	Data segment

ES	19	
GS	20	
FS	21	
KS	22	Key segment
LS	23	Library segment
IDTR	24	Interrupt descriptor table pointer
CMPR	25	Comparison result register
XEIP	26	Pointer to start of current instruction
LADD	27	Current interrupt code
LINT	28	Current interrupt number
TMR	29	Instruction/cycle counter
TIMER	30	Internal precise timer
CPAGE	31	Current page number
IF	32	Interrupts enabled flag
PF	33	Protected mode flag
EF	34	Extended mode flag
NIF	35	Next cycle interrupt enabled flag state
MF	36	Extended memory mapping flag
PTBL	37	Page table offset
PTBE	38	Page table number of entries
PPAGE	41	Previous page ID
External	44	External I/O operation
BusLock	45	Is bus locked for read/write
Idle	46	Should CPU skip some cycles
INTR	47	Handling an interrupt
SerialNo	48	Processor serial number
CODEBYTES	49	Amount of bytes executed so far
BPREC	50	Binary precision level
IPREC	51	Integer precision level
NIDT	52	Number of interrupt descriptor table entries
BlockStart	53	Start offset of the block
BlockSize	54	Block size
VMODE	55	Vector mode (2: 2D, 3: 3D)
XTRL	56	Runlevel for external memory access
HaltPort	57	Halt until this port changes value
HWDEBUG	58	Hardware debug mode active
DBGSTATE	59	Hardware debug mode state
DBGADDR	60	Hardware debug mode address/parameter

CRL	61	Current runlevel
TimerMode	64	Timer mode (off, instructions, seconds)
TimerRate	65	Timer rate
TimerPrevTime	66	Previous timer fire time
TimerAddress	67	Number of NMI to call when timer fires

Chapter 6

Instruction Set Reference

6.1 ZCPU Opcode Set

Mnemonic	Opcode	Operands	Description/psuedo-code
	000	0	Trigger interrupt 2
JNE	001	1	IF CMPR \neq 0 THEN IP = X END
JNZ	001	1	IF CMPR \neq 0 THEN IP = X END
JMP	002	1	IP = X
JG	003	1	IF CMPR $>$ 0 THEN IP = X END
JNLE	003	1	IFNOT CMPR \leq 0 THEN IP = X END
JGE	004	1	IF CMPR \geq 0 THEN IP = X END
JNL	004	1	IFNOT CMPR $<$ 0 THEN IP = X END
JL	005	1	IF CMPR $<$ 0 THEN IP = X END
JNGE	005	1	IFNOT CMPR \geq 0 THEN IP = X END
JLE	006	1	IF CMPR \leq 0 THEN IP = X END
JNG	006	1	IFNOT CMPR $>$ 0 THEN IP = X END
JE	007	1	IF CMPR = 0 THEN IP = X END
JZ	007	1	IF CMPR = 0 THEN IP = X END
CPUID	008	1	EAX = CPUID[X]
PUSH	009	1	MEMORY[ESP+SS] = X; ESP = ESP - 1
ADD	010	2	X = X + Y
SUB	011	2	X = X - Y
MUL	012	2	X = X * Y
DIV	013	2	IF Y = 0 THEN INTERRUPT(3,0) ELSE X = X / Y END

MOV	014	2	$X = X / Y$
CMP	015	2	$CMPR = X - Y$
	016	2	RESERVED
	017	2	RESERVED
MIN	018	2	$X = \text{MIN}(X, Y)$
MAX	019	2	$X = \text{MAX}(X, Y)$
INC	020	1	$X = X + 1$
DEC	021	1	$X = X - 1$
NEG	022	1	$X = -X$
RAND	023	1	X set to random value between 0 and 1
LOOP	024	1	IF ECX <> 0 THEN IP = X END
LOOPA	025	1	IF EAX <> 0 THEN IP = X END
LOOPB	026	1	IF EBX <> 0 THEN IP = X END
LOOPD	027	1	IF EDX <> 0 THEN IP = X END
SPG	028	1	Make page X readonly
SPG	028	1	IF CURRENT_PAGE.RUNLEVEL < PAGE[X].RUNLEVEL THEN PAGE[X].READ = 1 PAGE[X].WRITE = 0 ELSE INTERRUPT(11, X) END
CPG	029	1	Make page X readable and writeable
CPG	029	1	IF CURRENT_PAGE.RUNLEVEL < PAGE[X].RUNLEVEL THEN PAGE[X].READ = 1 PAGE[X].WRITE = 1 ELSE INTERRUPT(11, X) END
POP	030	1	$X = \text{MEMORY}[\text{ESP} + \text{SS}] ; \text{ESP} = \text{ESP} + 1$
CALL	031	1	IF PUSH(IP) THEN IP = X END
BNOT	032	1	$X = \text{BINARY NOT } X$
FINT	033	1	$X = \text{FLOOR}(X)$
FRND	034	1	$X = \text{ROUND}(X)$
FFRAC	035	1	$X = X - \text{FLOOR}(X)$
FINV	036	1	IF X = 0 THEN INTERRUPT(3, 0) ELSE X = 1 / X END
FSHL	038	1	$X = X * 2$

FSHR	039	1	$X = X / 2$
RET	040	0	IP = POP()
IRET	041	0	EF = 0: IP = POP() EF = 1: CS = POP(); IP = POP()
STI	042	0	IF = 1
CLI	043	0	IF = 0
	044	0	RESERVED
	045	0	RESERVED
	046	0	RESERVED
RETF	047	0	CS = POP(); IP = POP()
STEF	048	0	EF = 1
CLEF	049	0	EF = 0
FAND	050	2	$X = X \text{ AND } Y$
FOR	051	2	$X = X \text{ OR } Y$
FXOR	052	2	$X = X \text{ XOR } Y$
FSIN	053	2	$X = \text{SIN } Y$
FCOS	054	2	$X = \text{COS } Y$
FTAN	055	2	$X = \text{TAN } Y$
FASIN	056	2	$X = \text{ASIN } Y$
FACOS	057	2	$X = \text{ACOS } Y$
FATAN	058	2	$X = \text{ATAN } Y$
MOD	059	2	$X = X \text{ MOD } Y$
BIT	060	2	CMPR = X[Y]
SBIT	061	2	X[Y] = 1
CBIT	062	2	X[Y] = 0
TBIT	063	2	X[Y] = NOT BIT(X,Y)
BAND	064	2	$X = X \text{ BAND } Y$
BOR	065	2	$X = X \text{ BOR } Y$
BXOR	066	2	$X = X \text{ BXOR } Y$
BSHL	067	2	$X = X \text{ BSHL } Y$
BSHR	068	2	$X = X \text{ BSHR } Y$
JMPF	069	2	CS = Y; IP = X
NMIINT	070	1	NMIINTERRUPT(X)
CNE	071	1	IF CMPR <> 0 THEN CALL(X) END
CNZ	071	1	IF CMPR <> 0 THEN CALL(X) END
	072	1	RESERVED
CG	073	1	IF CMPR > 0 THEN CALL(X) END
CNLE	073	1	IFNOT CMPR <= 0 THEN CALL(X) END

CGE	074	1	IF CMPR >=0 THEN CALL(X) END
CNL	074	1	IFNOT CMPR < 0 THEN CALL(X) END
CL	075	1	IF CMPR < 0 THEN CALL(X) END
CNGE	075	1	IFNOT CMPR >= 0 THEN CALL(X) END
CLE	076	1	IF CMPR <= 0 THEN CALL(X) END
CNG	076	1	IFNOT CMPR > 0 THEN CALL(X) END
CE	077	1	IF CMPR = 0 THEN CALL(X) END
CZ	077	1	IF CMPR = 0 THEN CALL(X) END
MCOPY	078	1	Copy X bytes from absolute pointer ESI to absolute pointer EDI
MCOPY	078	1	FOR I = 1 TO X DO MEMORY[EDI] = MEMORY[ESI]; ESI++; EDI++; END
MXCHG	079	1	Swap X bytes between two blocks at absolute pointer ESI and absolute pointer EDI
MXCHG	079	1	FOR I = 1 TO X DO TEMP = MEMORY[EDI]; MEMORY[EDI] = MEMORY[ESI]; MEMORY[ESI] = TEMP; ESI++; EDI++; END
FPWR	080	2	X = X xor Y
XCHG	081	2	X,Y = Y,X
FLOG	082	2	X = LOG(Y)
FLOG10	083	2	X = LOG10(Y)
IN	084	2	X = PORT[Y]
OUT	085	2	PORT[X] = Y
FABS	086	2	X = ABS(Y)
FSGN	087	2	X = SIGN(Y)
FEXP	088	2	X = EXP(Y)
CALLF	089	2	IF PUSH(CS) AND PUSH(IP) THEN CS = Y; IP = X; END
FPI	090	1	X = PI
FE	091	1	X = E
INT	092	1	INTERRUPT(X)
TPG	093	1	Tests whether page X is readable. Sets CMPR to 1 on success, 0 on failure.
FCEIL	094	1	X = CEIL(X)
ERPG	095	1	Erases page X from internal ROM
WRPG	096	1	Writes page X to internal ROM
RDPG	097	1	Reads page X from internal ROM
TIMER	098	1	X = TIMER
LIDTR	099	1	IDTR = X

	100	1	RESERVED
JNER	101	1	IF CMPR <> 0 THEN IP = IP+X END
JNZR	101	1	IF CMPR <> 0 THEN IP = IP+X END
JMPR	102	1	IP = IP+X
JGR	103	1	IF CMPR > 0 THEN IP = IP+X END
JNLER	103	1	IFNOT CMPR <= 0 THEN IP = IP+X END
JGER	104	1	IF CMPR >=0 THEN IP = IP+X END
JNLR	104	1	IFNOT CMPR < 0 THEN IP = IP+X END
JLR	105	1	IF CMPR < 0 THEN IP = IP+X END
JNGER	105	1	IFNOT CMPR >= 0 THEN IP = IP+X END
JLER	106	1	IF CMPR <= 0 THEN IP = IP+X END
JNGR	106	1	IFNOT CMPR > 0 THEN IP = IP+X END
JER	107	1	IF CMPR = 0 THEN IP = IP+X END
JZR	107	1	IF CMPR = 0 THEN IP = IP+X END
LNEG	108	1	X = 1-CLAMP(X,0,1))
	109	1	RESERVED
NMIRET	110	0	NMIRESTORE()
IDLE	111	0	IDLE = 1
NOP	112	0	Does nothing
	113	0	RESERVED
PUSHA	114	0	Push all GP registers
POPA	115	0	Pop all GP registers
STD2	116	0	Enable hardware debug mode
LEAVE	117	0	ESP = EBP - 1; EBP = POP()
STEP	118	0	MF = 1
CLEP	119	0	MF = 0
CPUGET	120	2	X = CPU_REGISTER[Y]
CPUSET	121	2	CPU_REGISTER[X] = Y
LEA	126	2	Loads address of second operand. Returns second operand if it is not a memory reference
BLOCK	127	2	BLOCK_START = X; BLOCK_END = Y
CMPAND	128	2	IF CMPR <> 0 THEN CMPR = X - Y END
CMPOR	129	2	IF CMPR = 0 THEN CMPR = X - Y END
MSHIFT	130	2	Shift X bytes by Y positions (positive shifts left) at absolute pointer ESI
RSTACK	133	2	X = MEMORY[SS+Y]
SSTACK	134	2	MEMORY[SS+X] = Y
ENTER	135	1	PUSH(EBP); EBP = ESP + 1; ESP = ESP - X

6.2 Vector Extension Opcode Set

Mnemonic	Opcode	Operands	Description/psuedo-code
VADD	250	2	$X = X + Y$
VSUB	251	2	$X = X - Y$
VMUL	252	2	$X = X * \text{SCALAR } Y$
VDOT	253	2	$X = X \cdot Y$
VCROSS	254	2	$X = X \times Y$
VMOV	255	2	$X = Y$
VNORM	256	2	$X = \text{NORMALIZE}(Y)$
VCOLORNORM	257	2	$X = \text{COLOR_NORMALIZE}(Y)$
MADD	260	2	$X = X + Y$
MSUB	261	2	$X = X - Y$
MMUL	262	2	$X = X * Y$
MROTATE	263	2	$X = \text{ROT}(Y)$
MSCALE	264	2	$X = \text{SCALE}(Y)$
MPERSPECTIVE	265	2	$X = \text{PERSP}(Y)$
MTRANSLATE	266	2	$X = \text{TRANS}(Y)$
MLOOKAT	267	2	$X = \text{LOOKAT}(Y)$
MMOV	268	2	$X = Y$
VLEN	269	2	$X = \text{Sqrt}(Y \cdot Y)$
MIDENT	270	1	$X = \text{IDENTITY}()$
VMODE	273	1	Vector mode = Y
VDIV	295	2	$X = X / Y$
VTRANSFORM	296	2	$X = X * Y$

Chapter 7

HL-ZASM

7.1 Assembler Syntax

7.2 Expression Generator

7.3 C Syntax Support

7.4 Special Features