Zyelios CPU 10 Handbook

Black Phoenix

Русская документация доступна здесь

The Zyelios CPU is a virtual processor. It was created so the Wiremod players could include it in their contraptions, automating their contraptions and learning the basics of low-level programming, assembly programming and, with introduction of the HL-ZASM compiler, C programming.

This processor is very similar to the kind of processors all modern complex devices use. Its closest real-world analog is the `x86` architecture, but it has several major distinctions, the most of which is the use of floating-point numbers instead of integers.

It should be noted that the processor is a complex machine, and just like a real vehicle it will require some practice to master. It should be understood, however, that despite large number of possible uses of the processor it’s based on a very simple set of logic rules. Only the complex combination of these basic logic rules creates all the behaviour of the processor.

This handbook should serve as a concise documentation that covers all of the ZCPU features and quirks you might encounter during your programming. Just like a car manual for a real car, this book covers all of the controls and features of the ZCPU. This handbook assumes the reader has some basic knowledge of real-world processors.

The processor was invented as a logical step up from basic analog computers. First processors were tasked with solving mathematical equations, performing simulations, and processing large arrays of statistic data.

The program that the processor must execute is split into instructions. Each instruction is a simple operation, which somehow alters processor state, or state of any external devices. For example:

```
MOV R0,100 //Move 100 to R0
ADD R0,R1 //Add R0 with R1
```

The extra parameters specified by a programmer to alter the execution of a specific instruction by the processor are called *operands*. All the instructions in ZCPU have zero, one or two operands. The first operand is usually the *destanation* operand, and the second one is the *source* operand.

If there is some result after execution the instruction, it is written to the first operand:

```
ADD R0,200; //R0 = R0 + 200
```

A *register* is a variable inside the processor. They are divided into *general purpose* registers and the *special* (internal) registers. Programmer may use general purpose registers as temporary or intermediate storage for any sort of numeric data:

```
MOV R6,100;
MOV R7,R6;
```

The special registers are used to alter the processor execution mode.

The processor may be attached to external devices and external memory via data bus. A *data bus* is a special type of device which allows to connect several other devices together. There are two data buses in ZCPU: the `MemBus` (memory bus), and the `IOBus` (input/output bus). Any memory cell on these buses and in the processors internal memory can be accessed by an *address* - an integer value. There is no principial difference between how the two buses operate, although it is possible to use memory cells located on the `IOBus` in an easy to understand way - by addressing them as ports:

```
MOV R0,PORT0 //Read cell 0 via I/O bus
MOV [500],R0 //Write to cell 500 via memory bus
```

There are several memory models provided by the ZCPU - these specify how much internal memory the ZCPU has (it may either have some memory and ROM, like a microcontroller, or not use any internal memory at all, similarly to common processors).

There are 40 general purpose registers in the ZCPU. They are divided into the *main* and the *extended* set. The registers from the main set are called `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP` and `EBP` (named similarly to the x86 architecture registers). The registers from the extended set are called `R0`, `R1`, ..., `R31`.

You can use any of these for any arithmetics purposes (that means they only hold numerical values), apart from the `ESP` register, which is reserved as the stack pointer (see: ?? for more information on the processor stack). When programming using C syntax the `EBP` register is also used for storing stack frame pointer, and must not be altered to avoid program crash.

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. This is a simplification that differs ZCPU from any other real-world processor.

Registers are *a lot faster* than using memory. In the version 10 of the processor 32 more general purpose registers were added to speed up the program execution.

When the CPU is reset, all registers (except for `ESP`) will be set to 0. If the current ZCPU memory model includes RAM space, then `ESP` will be reset to the CPU internal RAM size minus one (e.g. `65535` for 64k memory model).

The Zyelios CPU has a special register called the instruction pointer (`IP`). `IP` points to the currently executed instruction and is incremented every time a new instruction is read.

Instructions can be either variable sized or fixed-sized, depending on the local execution mode (see: ??), so the `IP` may increment in varying steps. It is possible to set `IP` explictly by using any of the branching opcodes (see: ??)

The following is the internal execution flow for fetching and executing an instruction:

1. Fetch instruction number (so the processor would know what instruction it is going to execute, and how much operands will the instruction have)
2. If the instruction has operands, then fetch the RM byte (see `rmbyte`). If the instruction has no operands and the next byte is zero it will be skipped (this behaviour assures that the processor is backwards compatible).
3. Processor decodes instruction number and fetches segment prefix bytes (if required, this provides support for segment prefixes. see: ?? for more information on the instruction format).
4. If RM byte requires an immediate value to follow, it will be fetched too.
5. The processor executes the instruction microcode, which somehow alters the processor state (but does not touch the values which are explictly stated in the operation).
6. All values that must be written are written back to registers, memory, etc.

For the purpose of optimization, one or more of these steps might be cached by the processor and omitted from actual execution. See page ?? for more information on this behaviour.

The processor will run a specific amount of cycles per second. The amount of cycles required for executing each instruction may vary, and it is counted by the `TMR` register. Most instructions are only 1 cycle long, although internal caching or memory access may consume additional cycles.

see: ?? more information on the execution process.

When a certain exceptional event occurs (external signal, error in arithmetics, memory error, etc), the ZCPU will generate an interrupt. Interrupts are special events which will temporarily "interrupt" or change flow of the program and branch to execute the appropriate interrupt handler code. This means that whenever the ZCPU encounters an interrupt it will remember its current state, store it on the processor stack (see: ??) and change `IP` to the first opcode of the interrupt handler.

In extended mode the processor stores an interrupt table - a table which holds pointers to all interrupt handlers. If extended mode is not enabled, any interrupts called will stop the processor execution.

After the interrupt handler has finished running, it can restore the ZCPU state and the previously executed code will continue executing from where it was interrupted.

For example, if this was the stack state before an interrupt call, and current value of `IP` is 157:

```
..........
65304: ...
65306: 181
65305: -94

ESP = 65304
```

then the stack state will be altered to this after an interrupt call (and `IP` will be set to interrupt handler entrypoint):

```
..........
65304: ...
65306: 181
65305: -94
65304: 0   CS
65303: 157 IP
```

```
ESP = 65302
```

Interrupts are very similar to normal calls (see: ??), but instead of using a memory pointer an interrupt number is given. Also unlike the regular calls a special return instruction `IRET` must be used in place of `RET`. Any interrupt can be called using the `INT` instruction. For example:

```
interrupt_handler:
  ....
iret;
```

There are two types of interrupts: internal interrupts, and external interrupts. The external interrupts will push the entire ZCPU state (along with the return data internal interrupt would push) to stack before entering the interrupt handler. They can only be triggered via external interrupt pin, or by using the `EXTINT` instruction. The external interrupts require `EXTRET` instruction instead of the `IRET` to restore the processor state.

There are several static interrupts. These interrupts are executed when certain exceptional conditions happen internally. All interrupt numbers less than 32 are reserved by the processor. In addition, interrupts 0 and 1 have special meanings and are handled internally. The user program can only use the interrupts 32..255. see: ?? for more information on the error interrupts.

The following table describes all of the reserved interrupt numbers:

Number	Description
0	*Reset*
1	*Halt*
2	End of program execution
3	Division by zero

4	**Unknown opcode**
5	**Internal processor error**
6	**Stack error (overflow/underflow)**
7	**Memory read/write fault**
8	**MemBus fault**
9	**Write access violation (page protection)**
10	**Port read/write fault**
11	**Page acccess violation (page protection)**
12	**Read access violation (page protection)**
13	**General processor fault**
14	**Execute access violation (page protection)**
15	**Address space violation**
17	**Frame instruction limit (*GPU only*)**
23	**String read error (*GPU only*)**
28	***Page read access trap***
29	***Page write access trap***
30	***Page access trap***
31	***Debug trap***

The reset interrupt (0) will reset the processor state, and restart code execution from the top of memory. The halt interrupt (1) will stop execution until the processor is reset externally. The `int 1` and `int 0` instructions will work as long as the interrupts are not handled by the software (as long as they are disabled in the interrupt table, or interrupt table is not used):

```
INT 1; //Stop execution

INT 0; //Reset the processor
```

If extended mode is not enabled, *all the interrupts* will work similarly to the halt interrupt (`int 1`) when triggered. The error code will be set on the error output.

Interrupt handling can be disabled using the `IF` flag. It is not directly writeable, but it can be changed using the special instructions. *When setting the flag, the value will only become updated after the next instruction*. For example:

```
STI;         //Set interrupt flag
MOV EAX,123; //IF = 0
ADD EAX,123; //IF = 1 (flag updated)
CLI;         //Clear interrupt flag
SUB EAX,123; //IF = 0 (flag clear on this instruction)
RET;         //IF = 0
```

This assures that for example in this situation the interrupt will not occur between the `RET` and the `STI`:

```
ContextSwitch:
  CLI;

  ... thread switcher code ...

  STI;  //An interrupt will never occur between this
EXTRET; //STI instruction and the NMIRET
```

For information on using the interrupt table, consult the advanced interrupts chapter (see: ??).

There are 8 segment registers in the ZCPU. They are `CS`, `SS`, `DS`, `ES`, `GS`, `FS`, `KS` and `LS`. They are used by the ZCPU to support certain memory models. Additionally any of the 40 general purpose registers can be used as segment registers.

Every segment register can contain a 48-bit integer. This value can be used for specifying an offset for a pointer to memory. The ZCPU always uses segments when referencing memory: it translates the local address (the memory address explictly written by the user, it's important to remember that any variable or label has a certain memory address) into the absolute address (the physical address, which serves as a command to direct ZCPU towards where the real data is located).

The formula used for the address translation is:

$$AbsoluteAddress = LocalAddress + SegmentOffset$$

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

Both segment and general purpose registers can be used for segment prefixing. It is not possible to use a constant value for segment prefix. Here is an example of various ZASM syntax 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 valid
```

With some clever programming the segment prefixes can be used for quick array access, typically by using the prefix as the array pointer and the base as the index into the array:

```
MOV ES,ArrayStart
MOV EAX,ES:#0 //EAX = 10
MOV EBX,ES:#2 //EBX = 30
MOV ECX,ES:#1 //ECX = 50

MOV EAX,0
MOV EBX,EAX:#ArrayStart //EBX = 10
INC EAX
MOV EBX,EAX:#ArrayStart //EBX = 50
INC EAX
MOV EBX,EAX:#ArrayStart //EBX = 30

ArrayStart: db 10,50,30
```

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

All segment registers other than `CS` can be directly set by using the `MOV` operation. The only way to modify the `CS` is to execute `CALLF` or `JMPF` instructions (see: ?? for more

information):

```
//CS = 0
//IP = 928

JMPF 500,100;

//CS = 100
//IP = 500
```

Attempting to set `CS` directly will trigger interrupt `13:1` (general processor fault). For example:

```
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`.

The ZCPU is capable of working in 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 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.

It is important to be aware of how memory access actually works. When a memory request occurs (read or write) these are the actions that will be undertaken by the CPU:

1. Check `BusLock` register. If this register is set to 1 the request will fail.
2. Validate the address. The address must be a 48-bit integer value.
3. Fetch page that corresponds to the requested address.

4. If `EF` flag is set check if current runlevel is less or equal to target page runlevel and check if it’s possible to read/write on the page.
5. If page has `Mapped` flag set then change the address accordingly.
6. Perform the I/O operation.

This is the default address mode when all segment registers are initialized to `0`. This mode is available after the initial processor startup. Since all registers are set to zero, in this mode the code, data and stack are all located in same address space.

No segment prefixes are required in this mode. For example you can access the processor stack without specifying the `SS` prefix:

```
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
```

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 different data block
```

This will run the same routine (located at physical offset `500`) on two different sets of data. Thus the same subprogram can be reused for different sets of variables.

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. This mode 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: ?? for more information.

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*.

PUSH and POP are used to push or pop values to or from the stack, respectively. Stack overflow or underflow is 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, the RSTACK and SSTACK instructions allow the user to read or write an arbitrary position on the stack. These instructions may also trigger stack underflow or overflow interrupts. The following example illustrates the use of RSTACK and SSTACK:

```
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
```

The Zyelios CPU supports various kinds of branching: conditional or unconditional, absolute or relative. The instruction pointer (IP) points to the currently executing instruction. All branching instructions modify IP; some also modify CS (see: ??).

The simpliest type of branching is absolute unconditional. To perform an unconditional jump, JMP or JMPF are used (the latter also modifies CS).

Program subroutines can be called by using CALL or CALLF. This will save current the instruction pointer (and CS when CALLF is used) to the processor stack. The instruction pointer (and code segment if required) can be restored from the stack by using RET or RETF accordingly.

The following example illustrates creating routines and jumping:

```
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, `JMPR` (jump relative) is used. This instruction adds or subtracts `IP` by a certain amount instead of giving a specific number. For example 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. `CMP` is used to compare two values, and instructions from the following table react accordingly:

Instruction	Operation	Description
`JNE`	X <> Y	Jump if not equal
`JNZ`	X - Y <> 0	Jump if not zero
`JG`	X > Y	Jump if greater than
`JNLE`	NOT X <= Y	Jump if not less or equal
`JGE`	X >= Y	Jump if greater or equal
`JNL`	NOT X < Y	Jump if not less than
`JL`	X < Y	Jump if less than
`JNGE`	NOT X >= Y	Jump if not greater or equal
`JLE`	X <= Y	Jump if less or equal
`JNG`	NOT X > Y	Jump if not greater than
`JE`	X = Y	Jump if equal
`JZ`	X - Y = 0	Jump if zero
`CNE`	X <> Y	Call if not equal
`CNZ`	X - Y <> 0	Call if not zero
`CG`	X > Y	Call if greater than
`CNLE`	NOT X <= Y	Call if not less or equal
`CGE`	X >= Y	Call if greater or equal
`CNL`	NOT X < Y	Call if not less than
`CL`	X < Y	Call if less than
`CNGE`	NOT X >= Y	Call if not greater or equal
`CLE`	X <= Y	Call if less or equal
`CNG`	NOT X > Y	Call if not greater than

CE	X = Y	Call if equal
CZ	X - Y = 0	Call if zero
JNER	X <> Y	Jump relative if not equal
JNZR	X - Y <> 0	Jump relative if not zero
JGR	X > Y	Jump relative if greater than
JNLER	NOT X <= Y	Jump relative if not less or equal
JGER	X >= 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 >= Y	Jump relative if not greater or equal
JLER	X <= 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 `BIT`, which tests specific bits of a given value. 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
```

There are several error codes that might be generated during program execution. Each error condition generates an interrupt.

If processor is not in extended mode, code execution will halt when the error is generated. 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. If the processor is in extended mode, the corresponding interrupt will be called instead.

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 the CPU is reset. The following are error codes are defined by the current version of the processor:

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 acccess violation (page protection)
`12`	Read access violation (page protection)
`13`	General processor fault
`14`	Execute access violation (page protection)
`15`	Address space violation

Error message: STOP detected
Occurs when: `STOP`/`OPCODE 0` is executed
Cause: Abnomal 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

Error message: Unable to divide by zero
Occurs when: Second operand for `DIV` opcode is zero
Cause: User error
Result: `LADD = 1`

Error message: Unable to perform inversion on a zero
Occurs when: Calling `FINV` opcode with zero operand
Cause: User error
Result: `LADD = 2`

Error message: Unknown opcode detected in the instruction stream
Occurs when: Next executed instruction is not one of the recognized instructions
Cause: Invalid branching operation is performed (processor attempts to execute data)
Result: `LADD = OPCODE NUMBER`

Error message: Unable to execute instruction (microcode error)
Occurs when: An internal error has ocurred while executing microcode
Cause: Encountering a bug in processor
Result: `LADD = 01`

Error message: Unknown internal error
Occurs when: Never
Cause: None
Result: `LADD = 02`

Error message: Read error while fetching the instruction
Occurs when: Instruction decoder was not able to fetch all of the instruction bytes
Cause: `JMP` or `CALL` to a memory location outside of physical/logical range
Result: `LADD = 12`

Error message: Could not fetch an immediate byte for operand 1
Occurs when: Instruction decoder was not able to fetch an immediate byte for the first operand
Cause: `JMP` or `CALL` to a memory location outside of physical/logical range
Result: `LADD = 22`

Error message: Could not fetch an immediate byte for operand 2
Occurs when: Instruction decoder was not able to fetch an immediate byte for the second operand
Cause: `JMP` or `CALL` to a memory location outside of physical/logical range
Result: `LADD = 32`

Error message: Invalid RM parameter for operand 1
Occurs when: Never
Cause: None
Result: `LADD = 42`

Error message: Invalid RM parameter for operand 2
Occurs when: Never
Cause: None
Result: `LADD = 52`

Error message: Target operand not writeable
Occurs when: Never
Cause: None
Result: `LADD = 62`

Error message: Source operand not writeable (`XCHG` instruction only)
Occurs when: Never
Cause: None
Result: `LADD = 72`

Error message: Unknown opcode
Occurs when: Never
Cause: None
Result: `LADD = 82`

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

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 occured 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 occured while attempting to write a value (value out of the device address range)
Result: `LADD` equals to faulty address in memory

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`

Error message: Access violation
Occurs when: Attempting to write a value
Cause: `EF` flag set to 1, no write permission on referenced page, and referenced by the address has a smaller runlevel than the current one.
Result: `LADD` equals to faulty address in memory

Error message: Read error: unable to read a port
Occurs when: Processor attempts to read a value from a port
Cause: A failure has occured while attempting to read a value (value out of he 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 occured while attempting to write a value (value out of the device address range)
Result: `LADD` equals to faulty address in memory

Error message: Unable to set page readonly
Occurs when: `SPG` opcode is executed
Cause: Current runlevel greater than runlevel of the refernced page
Result: `LADD` points to referenced page

Error message: Unable to set page read- and writeable
Occurs when: `CPG` opcode is executed
Cause: Current runlevel greater than runlevel of the refernced page
Result: `LADD` points to referenced page

Error message: Unable to set page flag
Occurs when: `SPP` opcode is executed
Cause: Current runlevel greater than runlevel of the refernced page
Result: `LADD` points to referenced page

Error message: Unable to clear page flag
Occurs when: `CPP` opcode is executed
Cause: Current runlevel greater than runlevel of the refernced page
Result: `LADD` points to referenced page

Error message: Unable to set page runlevel
Occurs when: `SRL` opcode is executed
Cause: Current runlevel greater than runlevel of the refernced page
Result: `LADD` points to referenced page

Error message: Unable to set page mapping
Occurs when: `SMAP` opcode is executed
Cause: Current runlevel greater than runlevel of the refernced page
Result: `LADD` points to referenced page

Error message: Access violation
Occurs when: Attempting to read a value
Cause: `EF` flag set to 1, no read permission on referenced page, and referenced by the address has a smaller runlevel than the current one.
Result: `LADD` equals to faulty address in memory

Error message: Referenced page outside ROM
Occurs when: `ERPG` opcode executed
Cause: Page referenced by the instruction could not be erased, because it lies outside the processor ROM
Result: `LADD = 0`

Error message: Referenced page outside ROM
Occurs when: `WRPG` opcode executed
Cause: Page referenced by the instruction could not be written, because it lies outside the processor ROM
Result: `LADD = 0`

Error message: Referenced page outside ROM
Occurs when: `RDPG` opcode executed
Cause: Page referenced by the instruction could not be read, because it lies outside the processor ROM
Result: `LADD = 0`

Error message: Code segment write violation
Occurs when: `CS` value is being used as a target operand
Cause: An attempt is being made to set a read-only segment register. Use `CALLF` or `JMPF` instructions to modify the CS register
Result: `LADD = 1`

Error message: Unable to read interrupt table
Occurs when: Could not fetch entry from interrupt table
Cause: Invalid interrupt table location set with `LIDTR`
Result: `LADD = 2`

Error message: Invalid interrupt number
Occurs when: Interrupt outside of the acceptable boundary (0..255) is called
Cause: Misuse of the `INT` or the `EXTINT` instructions
Result: `LADD = 3`

Error message: Unprivileged external interrupt call
Occurs when: External interrupt was called while the appropriate flag was not set in the interrupt table
Cause: `6th` bit not set for the external interrupt in the interrupt table, and the external interrupt is called
Result: `LADD = 4`

Error message: Incompatible mode
Occurs when: Interrupt call in protected (compatibility) mode with less than 512 bytes of RAM
Cause: Not enough RAM to contain a complete interrupt table
Result: `LADD = 5`

Error message: Unable to push return data
Occurs when: Interrupt handler was unable to push return data onto the processor stack
Cause: Interrupt was called with no stack space available
Result: `LADD = 6`

Error message: Unable to call an interrupt
Occurs when: Interrupt is called without sufficient rights
Cause: Interrupt is called without sufficient rights to complete the jump
Result: `LADD = 7`

Error message: Unable to read page table
Occurs when: Page table is not located in valid memory range
Cause: Page table is not located in valid memory range
Result: `LADD = 8`

Error message: Attempting to execute a privileged instruction
Occurs when: Could not execute a privileged instruction due to runlevel
Cause: Calling one of the following instructios without sufficient rights: `RD`, `WD`, `SPG`, `CPG`, `STI`, `CLI`, `STP`, `CLP`, `STEF`, `CLEF`, `EXTINT`, `ERPG`, `WRPG`, `RDPG`, `LIDTR`, `EXTRET`, `IDLE`, `STD2`, `STM`, `CLM`, `CPUGET`, `CPUSET`, `CPP`, `SPP`, `SRL`, `GRL`, `SMAP`, `GMAP`
Result: `LADD = Opcode number`

Error message: Access violation
Occurs when: Execution is attempted on a memory page that does not allow execution
Cause: Invalid branching into the protected memory areas

Result: `LADD = Page number`

Error message: Access violation
Occurs when: Execution is attempted on a memory page that does not allow execution
Cause: Code execution has entered page that does not allow execution
Result: `LADD = Page number`

Error message: Address space violation
Occurs when: Memory is accessed by an invalid address
Cause: Address is not a 48-bit signed integer
Result: `LADD = Referenced invalid address`

This section describes instruction execution in complete detail. The processor will fetch instructions sequentially, decode them, and then execute.

During execution the `XEIP` register points to the instruction start address in global memory space, while the `IP` register is incremented as the instruction bytes are fetched in sequence.

The execution will also increment `TMR` register by amount of cycles that have been required to complete this instruction. The `CODEBYTES` register is incremented each time next byte is fetched by the instruction decoder (so the `CODEBYTES` register counts the total size of executable code executed so far).

```
//TMR = 170

MOV EAX,10; //1-cycle instructions
ADD EAX,EBX;

//TMR = 172
```

The instruction decoder will set `CPAGE` register to number of the current page (the page that fetched instruction is located on), and `PPAGE` to number of the page previous instruction was located on. *Current page is determined by the location of the first fetched byte of the currently executed instruction*.

When `CPAGE` and `PPAGE` values mismatch the permission check logic is triggered. Peforming a jump of any sort will run a different check logic, and will reset both `CPAGE` and `PPAGE` to page the target jump offset is located on.

The instructions can be either fixed-sized or variable-sized, see: ??.

This is the Pseudocode for the instruction decoding procedure:

```
// Calculate absolute execution address and set current page
XEIP = IP + CS
SetCurrentPage(floor(XEIP/128))

// Do not allow execution if we are not on kernel
// page, or not calling from kernel page
if (PCAP == 1) and (CurrentPage.Execute == 0) and
   (PreviousPage.RunLevel <> 0) then
  Interrupt(14,CPAGE)
end

// Reset interrupts flags
INTR = 0
if NIF <> undefined then
  IF = NIF
  NIF = undefined
end

// Fetch instruction and RM byte
```

```
Opcode = Fetch()
RM = 0
isFixedSize = false

// Check if it is a fixed-size instruction
if ((Opcode >= 2000) and (Opcode < 4000)) or
   ((Opcode >= 12000) and (Opcode < 14000)) then
  Opcode = Opcode - 2000
  isFixedSize = true
end

// Fetch RM if required
if (OperandCount > 0) or
   (Precompile_Peek() == 0) or
   (isFixedSize) then
  RM = Fetch()
end

// If failed to fetch opcode/RM then report an error
if INTR == 1 then
  IF = 1
  Interrupt(5,12)
end

// Check opcode runlevel
if (PCAP == 1) and (CurrentPage.Runlevel > RunLevel[Opcode]) then
  Interrupt(13,Opcode)
end

// Decode RM byte
dRM2 = floor(RM / 10000)
```

```
dRM1 = RM - dRM2*10000

// Default segment offsets
Segment1 = -4
Segment2 = -4

// Decode segment offsets
if Opcode > 1000 then
  if Opcode > 10000 then
    Segment2 = Fetch()

    Opcode = Opcode-10000
    if Opcode > 1000 then
      Segment1 = Fetch()

      Opcode = Opcode-1000
      Segment1 <> Segment 2
    else
      if isFixedSize then
        Fetch()
      end
    end
  else
    Segment1 = Fetch()
    Opcode = Opcode-1000
    if isFixedSize then
      Fetch()
    end
  end
elseif isFixedSize then
  Fetch()
```

```
      Fetch()
    end

    // If failed to fetch segment prefix then report an error
    if INTR == 1 then
      Interrupt(5,12)
    end

    // Check if opcode is invalid
    if opcode is not valid then
      Interrupt(4,Opcode)
    else
      // Fetch immediate values if required
      if isFixedSize then
        OperandByte1 = Fetch()
        if INTR == 1 then
          Interrupt(5,22)
        end
        OperandByte2 = Fetch()
        if INTR == 1 then
          Interrupt(5,32)
        end
      else
        if OperandCount > 0 then
          if NeedFetchByte1 then
            OperandByte1 = Fetch()
            // If failed to read the byte, report an error
            if INTR == 1 then
              Interrupt(5,22)
            end
          end
          if OperandCount > 1 then
            if NeedFetchByte2 then
              OperandByte2 = Precompile_Fetch() or 0
              // If failed to read the byte, report an error
              if INTR == 1 then
                Interrupt(5,32)
              end
            end
          end
        end
      end

      // Execute instruction
      Execute()

      // Write back the values
      if OperandCount > 0 then
        WriteBack(1)
        if OperandCount > 1 then
          WriteBack(2)
        end
      end
    end

    // Advance timers and counters
    CODEBYTES = CODEBYTES + Instruction_Size
    TMR = TMR + Instrucion_Cycles
    TIMER = TIMER + TimerDT

    // Set this page as previous (if it is executable)
    XEIP = IP + CS
```

```
SetPreviousPage(floor(XEIP/128))
```

The Zyelios CPU divides the entire accessable 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 different permission levels, with higher levels reflecting kernel-mode code and lower levels reflecting user-level code. Each runlevel is a number between 0 and 255 inclusive. Higher permission levels are those that are closer to 0. Lower permission levels are those that are closer to 255. Code executing from a page with a higher permission level can read or write pages of a lower runlevel, but the opposite cannot occur.

```
SRL 1,12 //Set page 1 runlevel to 12
//Page 1 is addresses 128..255

SPP 5,0 //Set page 5 to be readable
CPP 6,1 //Make page 6 non-writeable
CPP 7,2 //Prevent code execution on page 7
```

Runlevel 0 is a special runlevel which will ignore all permission settings. This runlevel is also capable of running several privileged instructions such as `CPUSET`. All others runlevels follow page permissions.

When a page is marked non-readable, it can only be read from a page with a higher runlevel. When a page is marked non-writeable it can only be written from a page with smaller runlevel. When a page is marked non-executable, then only code from runlevel 0 can branch into this page.

Every page can also be remapped to any other page in memory. This means that every time the processor accessess target page, the data will be read from a different physical address. Any page can be remapped, even if the page already points to valid memory.

```
MOV #130,1234 //Set cell 2 of page 1 to 1234
SMAP 0,1 //Remap page 0 to page 1
MOV EAX,#2 //Read cell 2 of page 0 (but really read from page 1)
//EAX is now set to 1234
```

Page mapping and permissions settings are stored in the *page table*. The page table is active when the processor has the extended memory mode enabled (`MF` 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 actual permission checks are being made only when `EF` flag is set to 1, and the processor is running in the extended mode.

The `PTBL` internal register is an *absolute* pointer to the table start, and `PTBE` holds the number of entires 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. Note that permission flags in the page table entry are *inverted*; 1 means the permission is restricted and 0 means that it is not.

Example on how to setup a page table:

```
PageTable:
  alloc 513*2 //512 page entries + default for 64KB of RAM

  .....

//Setup page table
CPUSET 37,PageTable //PTBL
CPUSET 38,512 //PTBE

//Enable extended memory mode
STM

//Indirect table manipulation
SMAP 0,1 //Map page 0 to 1
SPP 5,0 //Set page 5 to be readable

//Direct table manipulation
MOV ESI,PageTable; //Table offset
MOV ESI:#0,0xE0; //Set default page permissions to not allow anything
MOV ESI:#2,0; //Disable page 0
MOV ESI:#5,10; //Remap page 1 to page 10
```

The layout of the first byte of each entry is as follows:

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 external interrupt 30 (page trap) upon access
03	Page trap overrides read/write (generates external interrupts 28, 29)
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 bits 2 and/or 3. This will cause any access to this page to generate external interrupt 28 (read), 29 (write), or 30 (access) with the page number being accessed as the parameter.

If bit 2 and is set, the external interrupt 30 will be generated on memory access attempt. If bit 3 is also set then the external interrupts 28 or 29 will be generated instead of the interrupt 30, and it will be possible to override the result of the memory read/write access (see: ?? 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 `CALL`ing 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).

```
STEF //Enable extended mode
LIDTR 2048 //Interrupt table at pages 16-23

SRL 16,0 //Set runlevel of interrupts 0..31 to 0
SRL 17,1 //Set runlevel of interrupts 32..63 to 1
SRL 18,2 //Set runlevel of interrupts 64..95 to 2
SRL 19,3 //Set runlevel of interrupts 96..127 to 3
```

Current page when executing the instructions is determined by the `CPAGE` register. *The actual current page is determined by the location of the first fetched byte of the currently executed instruction*. The paging system performs permissions checks whenever `PPAGE` (page the previously executed instruction was located on) and `CPAGE` register values mismatch (which corresponds to crossing the page boundary).

Peforming a jump of any sort will run permission check logic, and will reset both `CPAGE` and `PPAGE` to page the target jump offset is located on.

It is only possible to cross page boundary or perform a jump if execute flag of the next (target) page is set to 1, or if it is set to 0 and previous page has runlevel 0.

All external access to the ZCPU inside memory will trigger the corresponding read/write permission checks. The runlevel for the external access is equal to the value of the `XTRL` register (external runlevel). By default the external memory access operations have a runlevel of 0.

To sum it up, these are all the permission and logic checks performed by the paging system:

- Boundary permission check (when execution crosses boundary between two pages)
- Jump permission check (execution jumps to a new offset)
- Interrupt table permission checks
- Read/write memory access (to any location)
- External read/write memory access (accessing CPU address space from outside)
- Read/write address remapping logic
- Page access trapping logic

The Zyelios CPU can be configured to 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 written into RAM every time the processor is reset. The ROM can be read from and written to in software.

There are three opcodes to work with the 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 ...
```

The ZCPU can work with integer values of variable bit width. It supports 8, 16, 32, and 48 bit integers, and has a complete instruction set to work with them.

There are bitwise logic instructions which work on all bits of the integer number. They are `BAND`, `BOR`, `BXOR`, `BSHL`, `BSHR`, `BNOT`. For example:

```
MOV  EAX,105 //1101001
BAND EAX,24  //0011000
//EAX = 8      0001000

BOR EAX,67   //1000011
//EAX = 75   //1001011

BXOR EAX,15  //0001111
```

```
//EAX = 68   //1000100

BSHL EAX,2
//EAX = 272  //100010000

BSHR EAX,4
//EAX = 17   //0010001

BNOT EAX
//EAX = -18  //11111111111111111101110
```

Also there are additional operations that work on separate bits of the number. They are BIT, SBIT, TBIT, CBIT.

The BIT operation tests whether specific bit of the number is set. It is possible to check result of this comparsion by using a conditional jump:

```
MOV EAX,105
BIT EAX,0
JNZ LABEL //Jump succeeds (bit is not zero)

BIT EAX,1
JNZ LABEL //Jump fails (bit is zero)
```

The SBIT and CBIT clear and set the specific bit of the number. TBIT instruction will toggle that bit:

```
MOV EAX,105 //1101001
SBIT EAX,1
//EAX = 107   1101011

CBIT EAX,6
```

```
//EAX = 43    0101011

TBIT EAX,0
//EAX = 42    0101010
```

Certain instructions in the ZCPU support using the `BLOCK` instruction before them to specify a memory block the instruction must be executed on. For example it is possible to set permissions of an entire memory block at once:

```
BLOCK 1024,8192 //A 8KB block at offset 1024
SRL 0,4 //Set runlevel of all pages in this block to 4
```

The first operand of the `BLOCK` instruction is the memory offset the block must start from (must be aligned on page boundary for the paging instructions), and the second operand is the block size (the size must be divisable by 128 if this is a block for the page permission instructions).

The `BLOCK` instruction sets the two internal registers `BlockStart`, `BlockSize`. After the instruction that supports memory blocks has executed, it will reset block size back to zero.

The block size must be non-zero for the instruction to work on that block.

It is possible that an interrupt will occur between the `BLOCK` instruction and the target instruction. It is advised against using `BLOCK` instruction inside an interrupt handler!

The following instructions support this psuedo-prefix: `SPP`, `CPP`, `SRL`, `SMAP`.

There are built-in instructions in the ZCPU for quickly moving and working with large arrays of data in an easy way.

It's possible to copy one area of memory to another using the `MCOPY` instruction:

```
mov ESI,source_data;
mov EDI,dest_data;
mcopy 5; //Copy 5 bytes

string source_data,"Apple";
string dest_data,"A quick fox";
```

After running this code "dest_data" string will be equal to "Appleck fox".

Two blocks of memory can be swapped using the `MXCHG` instruction:

```
mov ESI,source_data;
mov EDI,dest_data;
mxchg 5; //Swap 5 bytes

string source_data,"Apple";
string dest_data,"A quick fox";
```

After running this code "dest_data" string will be equal to "Appleck fox", and "source_data" will be equal to "A qui".

And it’s also possible to shift all values in block of data:

```
mov ESI,source_data;

mshift 13,2; //Shift 13 bytes by 2 cells to the right
//source_data = "quick foxA "

mshift 13,-4; //Shift 13 bytes by 4 cells to the left
//source_data = "oxA quick f"

string source_data,"A quick fox";
```

The offset by which data is shifted must be less or equal to amount of bytes that are being shifted. But it is possible to specify offsets larger than the data size, and that will reverse the direction of shift, and will not wrap the data around.

All of these instructions can work on up to `8192` bytes in a memory block.

The ZCPU processor has two extra instructions to provide stack frame support. Stack frame allows to prevent stack corruption by induvidual subroutines, and it also provides an easy way to create local variables on stack, and pass values to the subroutine via stack.

`ENTER X` instruction is same as the following code:

```
Push(EBP)
EBP = ESP + 1
ESP = ESP - X
```

It will set `EBP` register to stack frame base (value of stack pointer at function entrypoint), and add some empty space on stack for the local variables.

`LEAVE` instruction corresponds to the following code:

```
ESP = EBP - 1
EBP = Pop()
```

It will restore the stack frame base of the previous function (if it exists), and restore stack.

Interrupts work in a way similar to processor subroutines, but they can also be triggered if an error occurs while executing an instruction. In this case the instruction execution will not be completed, and the processor will attempt to handle this interrupt.

Interrupt handling is only available in extended mode of the processor. If interrupt occurs with extended mode disabled the processor will halt its execution and report error code to the external output:

```
lidtr interrupt_table; //Load interrupt table
stef; //Enable extended mode

int 40; //Handle interrupt 40
mov #0.123,10; //Handle interrupt 15 (address space violation)

clef; //Disable extended mode
int 40; //Will halt processor execution
```

The processor determines which subroutine it must call by checking the interrupt descriptor table/interrupt table. It’s a table located somewhere in the processor memory which holds addresses to all interrupt handlers, and extra flags that alter the way the interrupt is handled.

The pointer to this table must be specified using the `LIDTR` instruction. The interrupt table usually has 256 entries for 256 possible interrupts, but it may have less entries if `NIDT` (interrupt table entry count) register is changed. If processor attempts to handle an interrupt which is not in the table, it will skip it:

```
cpuset 52,32; //Only have 32 entries
lidtr interrupt_table; //Load interrupt table
stef; //Enable extended mode

int 31; //Will be handled
int 32; //Will be skipped
```

Each entry in the interrupt table has the following format:

Byte	Description

0	Interrupt handler IP
1	Interrupt handler CS (if required, see flags)
2	Reserved (must be 0)
3	Flags

There are the following flags that can be set:

Bit	Description
3	CMPR register will be set to 1 if an interrupt has occured
4	1 if interrupt should *not* set CS
5	Interrupt enabled (active)
6	This interrupt is an external interrupt
0	Interrupt handler IP

If bit 3 is set, the interrupt will set CMPR register to 1. This can be used for making simple exception handlers:

If bit 4 is set, the interrupt will not set the code segment (it sets it by default). The interrupt will only be called if bit 5 is set, if it's clear then it will be ignored (unless it's the interrupt 0 or 1, if so, then they will act as if called without extended mode enabled).

This interrupt can only be called as an external interrupt if bit 6 is set. Otherwise it will generate interrupt 13 (general processor fault) instead.

There are special rules for using the interrupt table if paging is used. see: ?? for more information on how permissions will work if that's the case.

The interrupt handler is fairly complex. The Pseudocode for the interrupt handler is listed below:

```
        // Interrupt is active, lock the bus to prevent any further read/write
        INTR = 1
        BusLock = 1

        // Set registers
        LINT = interruptNo
        LADD = interruptParameter or XEIP

        // Output the error externally
        SignalError(interruptNo,LADD)

        if IF == 1 then
          if EF == 1 then // Extended mode
            // Boundary check
            if (interruptNo < 0) or (interruptNo > 255) then
              if not cascadeInterrupt then Interrupt(13,3) end
            end

            // Check against boundaries of the interrupt table
            if interruptNo > NIDT-1 then
              if interruptNo == 0 then Reset = 1 end
              if interruptNo == 1 then Clk = 0 end
            end

            // Calculate absolute offset in the interrupt table
            interruptOffset = IDTR + interruptNo*4

            // Disable bus lock, set the current page for read operations
            BusLock = 0
            SetCurrentPage(interruptOffset)
```

```
IF = 0
INTR = 0
IP    = ReadCell(interruptOffset+0)
CS    = ReadCell(interruptOffset+1)
        ReadCell(interruptOffset+2)
FLAGS = ReadCell(interruptOffset+3)
IF = 1

if INTR == 1 then
  if not cascadeInterrupt then Interrupt(13,2) end
else
  INTR = 1
end

// Set previous page to trigger same logic as if
//  CALL-ing from a privilegied page
SetCurrentPage(XEIP)
SetPrevPage(interruptOffset)
BusLock = 1

if isExternal and (FLAGS[6] <> 1) then
  if not cascadeInterrupt then Interrupt(13,4) end
end

if FLAGS[5] == 1 then
  // Push return data
  BusLock = 0
  IF = 0
  INTR = 0
  Push(IP)
  Push(CS)
```

```
      IF = 1

      if INTR == 1 then
        if not cascadeInterrupt then Interrupt(13,6) end
      else
        INTR = 1
      end
      BusLock = 1

      // Perform a short or a long jump
      IF = 0
      INTR = 0
      if FLAGS[4] == 0
      then Jump(IP,CS)
      else Jump(IP)
      end
      IF = 1

      if INTR == 1 then
        if not cascadeInterrupt then Interrupt(13,7) end
      else
        INTR = 1
      end

      // Set CMPR
      if FLAGS[3] == 1 then
        CMPR = 1
      end
    else
      if interruptNo == 0 then
        Reset()
      end
      if interruptNo == 1 then
        Clk = 0
      end
      if FLAGS[3] == 1 then
        CMPR = 1
      end
    end
  end

  if (EF == 0) then // Normal mode
    if (interruptNo < 0) or
       (interruptNo > 255) or
       (interruptNo > NIDT-1) then
      // Interrupt not handled
      Exit()
    end
    if interruptNo == 0 then Reset = 1 end
    if interruptNo ~= 31 then Clk = 0 end
  end
end

// Unlock the bus
BusLock = 0
```

(no chapter)

(no chapter)

The ZCPU has an internal timer which can be used for precise time measurements during the code execution. The timer can also be used for triggering interrupts at precise time intervals.

The `TIMER` instruction can be used to fetch the internal timer value in seconds:

```
TIMER EAX
//EAX is now equal to amount of seconds since CPU startup
```

It is possible to configure the timer to trigger external interrupt after certain amount of seconds, or a certain amount of cycles has passed by configuring one of the special registers.

`TimerMode` register controls the timer mode. If it’s set to 0 the timer will be disabled. If `TimerMode` is set to 1 the timer will use `TMR` register as counter source, and if `TimerMode` is set to 2 the timer will use the `TIMER` register as the source.

`TimerRate` sets the number of cycles, or the number of seconds that must pass before the timer will trigger. `TimerPrevTime` register stores value of the `TMR` or the `TIMER` register when timer fired the last time.

`TimerAddress` is the number of the interrupt that will be called when the timer fires. The interrupt call will be an external interrupt call.

Changing value of the `TimerMode` register will reset the `TimerPrevTime` register.

This is an example of how to setup timer:

```
CPUSET 65,90; //Trigger once per 90 cycles
CPUSET 67,40; //Trigger external interrupt #40

CPUSET 64,1;  //Enable timer to count cycles
```

It could also be initialized for an interval in seconds:

```
CPUSET 65,1.5; //Trigger once per 1.5 seconds
CPUSET 67,40;  //Trigger external interrupt #40
CPUSET 64,2;   //Enable timer to count seconds
```

If precision in intervals between timer firing is required, the timer can be reset in the interrupt caller (if required):

```
ExternalInterrupt:
  CLI; //Disable interrupts

  .....

  CPUGET EAX,29; //Read cycles counter
  ADD EAX,4;     //Account for 4 'missing' cycles
  CPUSET 66,EAX; //Write last timer fire time

  STI; //Enable interrupts
EXTRET;
```

Or alternatively it’s possible to reset it by swapping the timer mode:

```
ExternalInterrupt:
  CLI; //Disable interrupts

  .....

  CPUSET 64,1; //Restart timer
  STI; //Enable interrupts
EXTRET;
```

(using VMODE,VADD,etc)

(Using hardware debug mode)

The Zyelios CPU will cache executed microcode for faster execution. This increases code execution speed, but some penalties apply. The processor will decode instructions the first time they are encountered, and all subsequent executions will use the cached execution information.

The Zyelios CPU will cache executed microcode for faster execution. The processor will decode instructions the first time they are encountered, and all subsequent executions will use the cached execution information. The cache will provide very little benefit for code that is executed infrequently, but code that is executed frequently will run much faster.

The penalties caused by caching system mostly concern the flow of execution and the internal optimizations the processor makes:

1. **While executing a single cached block of microcode the processor uses fast-access cached registers. This means that register values are not changed until the block finishes executing.**
2. **The processor only caches instructions the first time they are decoded. It will invalidate cache if the processor itself writes or reads to the memory, but it does not invalidate cache if any other device writes to an external source.**
3. **Cached microcode blocks contain up to 8 instructions each. They will end prematurely on unconditional branches and jumps.**
4. **All reading and writing operations might be delayed by several instructions until they actually happen. There are only very specific ways to make sure your I/O operation really happens.**

Each instruction in the ZCPU begins with a single byte which identifies the instruction to be executed. If the instruction has operands, then a *RM byte* (register/memory selector byte) follows. This extra byte specifies what kind of operands are being passed to the instruction.

Instruction number may also encode information about use of segment prefix, or about

the local execution mode (see: ??). The instruction number can fall into one of the following ranges:

- `000 - 999`: variable size instructions
- `1000 - 1999`: variable size instructions with segment prefix for 1st operand
- `10000 - 10999`: variable size instructions with segment prefix for 2nd operand
- `11000 - 11999`: variable size instructions with segment prefixes for both operands
- `2000 - 2999`: fixed size instructions
- `3000 - 3999`: fixed size instructions with segment prefix for 1st operand
- `12000 - 12999`: fixed size instructions with segment prefix for 2nd operand
- `13000 - 13999`: fixed size instructions with segment prefixes for both operands

The instruction may be followed by several extra bytes that specify segment prefix, constant values, etc. The constant values are always the last bytes in the instruction data. Here’s an example of a wide variety of different instructions:

```
STEF            48
INC EAX         20, 1
MOV EAX,10      14, 1, 10
ADD EAX,ESP     10, 70001
DIV EBX,ES:ECX  10013, 290002, 4
ADD R0,#R2      10, 20822048
MOV #100,#500   14, 250025, 100, 500
MOV EAX:#50,GS  1014, 130025, 9, 50
```

The RM byte consists of two parts - the RM selector for the first and the second operand: $RM = RM_1 + 10000 \cdot RM_2$.

For example RM bytes from the example in the previous chapter could be decoded like this:

```
                RM        RM1   RM2
```

```
STEF              n/a       n/a   n/a
INC EAX           1         1     n/a
MOV EAX,10        1         1     0
ADD EAX,ESP       70001     1     7
DIV EBX,ES:ECX    290002    2     29
ADD R0,#R2        20822048  2048  2082
MOV #100,#500     250025    25    25
MOV EAX:#50,GS    130025    25    13
```

These RM selectors are supported right now:

Selector	Operand	Source/target
0	123	Constant value
1	EAX	Register EAX
2	EBX	Register EBX
3	ECX	Register ECX
4	EDX	Register EDX
5	ESI	Register ESI
6	EDI	Register EDI
7	ESP	Register ESP
8	EBP	Register EBP
9	CS	Register CS
10	SS	Register SS
11	DS	Register DS
12	ES	Register ES
13	GS	Register GS
14	FS	Register FS
15	KS	Register KS
16	LS	Register LS

17	`#EAX, ES:#EAX`	Memory cell `EAX + segment`
18	`#EBX, ES:#EBX`	Memory cell `EBX + segment`
19	`#ECX, ES:#ECX`	Memory cell `ECX + segment`
20	`#EDX, ES:#EDX`	Memory cell `EDX + segment`
21	`#ESI, ES:#ESI`	Memory cell `ESI + segment`
22	`#EDI, ES:#EDI`	Memory cell `EDI + segment`
23	`#ESP, ES:#ESP`	Memory cell `ESP + segment`
24	`#EBP, ES:#EBP`	Memory cell `EBP + segment`
25	`#123, ES:#123`	Memory cell by constant value
26	`ES:EAX`	Register `EAX + segment`
27	`ES:EBX`	Register `EBX + segment`
28	`ES:ECX`	Register `ECX + segment`
29	`ES:EDX`	Register `EDX + segment`
30	`ES:ESI`	Register `ESI + segment`
31	`ES:EDI`	Register `EDI + segment`
32	`ES:ESP`	Register `ESP + segment`
33	`ES:EBP`	Register `EBP + segment`
34	No syntax	Memory cell `EAX + constant`
35	No syntax	Memory cell `EBX + constant`
36	No syntax	Memory cell `ECX + constant`
37	No syntax	Memory cell `EDX + constant`
38	No syntax	Memory cell `ESI + constant`
39	No syntax	Memory cell `EDI + constant`
40	No syntax	Memory cell `ESP + constant`
41	No syntax	Memory cell `EBP + constant`
42	No syntax	Register `EAX + constant`
43	No syntax	Register `EBX + constant`
44	No syntax	Register `ECX + constant`

45	No syntax	Register EDX + constant
46	No syntax	Register ESI + constant
47	No syntax	Register EDI + constant
48	No syntax	Register ESP + constant
49	No syntax	Register EBP + constant
50	ES:123	Constant value plus segment
1000	PORT0	Port 0
1001	PORT1	Port 1
....	....	....
2023	PORT1023	Port 1023
2048	R0	Extended register R0
....	....	....
2079	R31	Extended register R31
2080	#R0, ES:#R0	Memory cell R0 + segment
....	....	....
2111	#R31, ES:#R31	Memory cell R31 + segment
2112	ES:R0	Extended register R0 + segment
....	....	....
2143	ES:R31	Extended register R31 + segment
2144	No syntax	Memory cell R0 + constant
....	....	....
2175	No syntax	Memory cell R31 + constant
2176	No syntax	Extended register R0 + constant
....	....	....
2207	No syntax	Extended register R31 + constant

Segment offset is specified by a byte that follows the RM byte. For example, here is how

the same instruction is encoded with different segment prefixes:

```
MOV EAX,EBX        14,20001
MOV LS:EAX,EBX     1014,20027,8
MOV EAX,LS:EBX     10014,280001,8
MOV LS:EAX,LS:EBX  11014,280027,8,8
MOV R0:EAX,EBX     1014,20027,17
MOV EAX,R0:EBX     10014,280001,17
MOV R0:EAX,R0:EBX  11014,280027,17,17
```

There can be the following segment prefixes. The negative-value prefixes are obsolete now, and are only preserved for backwards compatibility:

Value	Register	Value	Register
-02	CS	01	CS
-03	SS	02	SS
-04	DS	03	DS
-05	ES	04	ES
-06	GS	05	GS
-07	FS	06	FS
-08	KS	07	KS
-09	LS	08	LS
-10	EAX	09	EAX
-11	EBX	10	EBX
-12	ECX	11	ECX
-13	EDX	12	EDX
-14	ESI	13	ESI
-15	EDI	14	EDI
-16	ESP	15	ESP
-17	EBP	16	EBP

17	R0		
...	...		
47	R32		

The Zyelios CPU supports two machine code formats: fixed-length and variable-length. The CPU automatically determines which type is in use in order to retain binary compatibility. Several simple rules are employed for this detection.

Example of instructions in the default mode:

```
STEF            48
INC EAX         20, 1
MOV EAX,10      14, 1, 10
ADD EAX,ESP     10, 70001
DIV EBX,ES:ECX  10013, 290002, 4
ADD R0,#R2      10, 20822048
MOV #100,#500   14, 250025, 100, 500
MOV EAX:#50,GS  1014, 130025, 9, 50
```

Example of instructions encoded in fixed-size mode:

```
STEF            2048,0,-4,-4,0,0
INC EAX         2020,1,-4,-4,0,0
MOV EAX,10      2014,1,-4,-4,0,10
ADD EAX,ESP     2010,70001,-4,-4,0,0
DIV EBX,ES:ECX  12013,290002,-4,4,0,0
ADD R0,#R2      2010,20822048,-4,-4,0,0
MOV #100,#500   2014,250025,-4,-4,100,500
MOV EAX:#50,GS  3014,130025,9,-4,50,0
```

Same instructions in the ZCPU compatibility mode:

```
STEF             48, 0
INC EAX          20, 1
MOV EAX,10       14, 1, 10
ADD EAX,ESP      10, 70001
DIV EBX,ES:ECX  10013, 290002, 4
ADD R0,#R2       10, 20822048
MOV #100,#500    14, 250025, 100, 500
MOV EAX:#50,GS  1014, 130025, 9, 50
```

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 these registers using `CPUSET` and `CPUGET`.

For example:

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

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

The registers `XEIP`, `CPAGE`, `PPAGE`, `SerialNo`, `CODEBYTES`, `TimerDT`, `RAMSize` are read-only - it is not possible to change their value by any means.

Changing value of the `IP` or the `CS` register is possible, and will be handled as a far jump.

See the section 2 for description of how some of the registers are used.

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`	Extra segment
`GS`	`20`	User segment

FS	21	User segment
KS	22	Key segment
LS	23	Library segment
IDTR	24	Interrupt descriptor table pointer
CMPR	25	Comparsion result register
XEIP	26	Pointer to start of currently executed 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
PCAP	39	Processor paging system capability

RQCAP	40	Processor delayed memory request capability
PPAGE	41	Previous page ID
MEMRQ	42	Type of the memory request
RAMSize	43	Amount of internal memory
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
TimerDT	62	Current timer discrete step
MEMADDR	63	Address reqested by the memory operation
TimerMode	64	Timer mode (off, instructions, seconds)
TimerRate	65	Timer rate
TimerPrevTime	66	Previous timer fire time
TimerAddress	67	Number of external interrupt to call when timer fires

---

This section describes the primary set of opcodes available in the ZCPU. Not all of these are present in ZGPU or ZSPU though. The following primary set instructions are not available in those architectures: `SPG`, `CPG`, `HALT`, `IRET`, `STI`, `CLI`, `STEF`, `CLEF`, `EXTINT`, `ERPG`, `WRPG`, `RDPG`, `LIDTR`, `EXTRET`, `STM`, `CLM`, `SPP`, `CPP`, `SRL`, `GRL`, `SMAP`, `GMAP`.

Mnemonic: `STOP`
Encoding: `000`

Triggers interrupt #2 (equivalent to `INT 2`). Will stop the processor execution if extended mode is not enabled.

Used by the processor to detect end of program/invalid jump error, because this instruction is usually the result of invalid jump. Acts as `NOP` if interrupts are disabled.

Pseudocode:

```
Interrupt(2,0)
```

Mnemonic: `JNE/JNZ X`
Encoding: `001` `RM` `[Segment1]` `[Constant1]`

Jumps to the specified address if in previous comparsion two values were not equal to each other.

Can also be used to check if result of the previous bit operation was not zero:

```
BIT EAX,2
JNZ LABEL
```

This code will jump to a label if the second bit is set to `1`.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
if CMPR <> 0 then
  Jump(X)
end
```

Mnemonic: `JMP X`
Encoding: `002` `RM` `[Segment1]` `[Constant1]`

Unconditional jump to the specified address.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
Jump(X)
```

Mnemonic: `JG/JNLE X`
Encoding: `003 RM [Segment1] [Constant1]`

Jumps to target address if the previous comparsion resulted in greater, or not less or equal result.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
if CMPR > 0 then
  Jump(X)
end
```

Mnemonic: `JGE/JNL X`
Encoding: `004 RM [Segment1] [Constant1]`

Jumps to target address if the previous comparsion resulted in greater or equal, or not less result.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
if CMPR >= 0 then
  Jump(X)
end
```

Mnemonic: `JL/JNGE X`
Encoding: `005 RM [Segment1] [Constant1]`

Jumps to target address if the previous comparsion resulted in less, or not greater or equal result.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
if CMPR < 0 then
  Jump(X)
end
```

Mnemonic: `JLE/JNG X`
Encoding: `006 RM [Segment1] [Constant1]`

Jumps to target address if the previous comparsion resulted in less or equal, or not greater result.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
if CMPR <= 0 then
  Jump(X)
end
```

Mnemonic: `JE/JZ X`
Encoding: `007 RM [Segment1] [Constant1]`

Jumps to the specified address if in previous comparsion two values were equal to each other.

Can also be used to check if result of the previous bit operation was zero:

```
BIT EAX,2
JZ LABEL
```

This code will jump to a label if the second bit is set to `0`.

May trigger error `14` (execution access violation) if instruction execution moves into a restricted memory area when extended mode is enabled. see: ?? for information about memory protection and paging.

Pseudocode:

```
if CMPR <> 0 then
  Jump(X)
end
```

Mnemonic: `CPUID X`
Encoding: `008 RM [Segment1] [Constant1]`

The `CPUID` instruction will write various information about the processor itself into the `EAX` register. This can be used to check for the processor capabilities.

The first parameter passed into the instruction defines what kind of the information is required:

Operand	Description
0	Processor version (current version: 10.00, reported as `1000`)
1	Amount of internal RAM
2	Processor type
3	Amount of internal ROM

The result will be written into the `EAX` register. Processor type can be one of the following:

EAX	Description
0	ZCPU
1	ZGPU beta version
2	ZSPU
3	ZGPU

Pseudocode:

```
EAX = CPUID[X]
```

Mnemonic: PUSH X
Encoding: 009 RM [Segment1] [Constant1]

Pushes a value to the processor stack (see: ?? for more information on the processor stack). Will check for interrupt overflow by comparing the new stack pointer to zero.

Uses the ESP register as the stack pointer, and the ESZ register as the stack size pointer.

Example of use:

```
PUSH 10
PUSH 20

POP EAX //EAX is now 20
```

May trigger error 6 (stack overflow/underflow) if the stack pointer goes outside the allowable limits.

May trigger error 7 (memory read/write fault) if the instruction was not able to perform a memory write/read operation due to an error.

Pseudocode:

```
MEMORY[ESP+SS] = X
ESP = ESP - 1

if ESP < 0 then
  ESP = 0
  Interrupt(6,ESP)
end
```

Mnemonic: ADD X,Y
Encoding: 010 RM [Segment1] [Segment2] [Constant1] [Constant2]

Adds two values together, and writes the result to the first operand.

Pseudocode:

```
X = X + Y
```

Mnemonic: `SUB X,Y`
Encoding: `011 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Subtracts second operand from the first one, and writes the result to the first operand.

Pseudocode:

```
X = X - Y
```

Mnemonic: `MUL X,Y`
Encoding: `012 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Muliplies two values together, and writes the result to the first operand.

Pseudocode:

```
X = X * Y
```

Mnemonic: `DIV X,Y`
Encoding: `013 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Divides the first operand by the second operand, and writes the result to the first operand. Checks for the division by zero error.

May trigger error 3 (division by zero) if the second operand was equal to zero.

Will not trigger division by zero error if interrupt flag register `IF` is set to 0.

Pseudocode:

```
if Y <> 0 then
  X = X / Y
else
  Interrupt(3,0)
end
```

Mnemonic: `MOV X,Y`
Encoding: `014 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Copies the contents of the second operand to the first operand.

Pseudocode:

```
X = Y
```

Mnemonic: `CMP X,Y`
Encoding: `015 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Compares the two operands together, and remembers the result of this comparsion.

This instruction is used in conjunction with the conditional branching instructions (see ??), for example:

```
CMP EAX,EBX
JG  LABEL1 //Jump if EAX >  EBX
JLE LABEL2 //Jump if EAX <= EBX
JE  LABEL3 //Jump if EAX  = EBX
```

Pseudocode:

```
CMPR = X - Y
```

Mnemonic: `MIN X,Y`

Encoding: `018 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds the smaller of the two values, and writes it to the first operand. For example:

```
MOV EAX,100
MOV EBX,200
MIN EBX,EAX //Sets EBX to 100
```

Pseudocode:

```
if X > Y then
  X = Y
end
```

Mnemonic: `MAX X,Y`
Encoding: `019 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds the bigger of the two values, and writes it to the first operand. For example:

```
MOV EAX,100
MOV EBX,200
MAX EAX,EBX //Sets EAX to 200
```

Pseudocode:

```
if X < Y then
  X = Y
end
```

Mnemonic: `INC X`
Encoding: `020 RM [Segment1] [Constant1]`

Increments the target operand by one. For example:

```
MOV EAX,100
INC EAX //EAX is now 101
```

Pseudocode:

```
X = X + 1
```

Mnemonic: DEC X
Encoding: 021 RM [Segment1] [Constant1]

Decrements the target operand by one. For example:

```
MOV EAX,100
DEC EAX //EAX is now 99
```

Pseudocode:

```
X = X - 1
```

Mnemonic: NEG X
Encoding: 022 RM [Segment1] [Constant1]

Negates the target operand (changes its sign). For example:

```
MOV EAX,123
NEG EAX //EAX is now -123

MOV EBX,0
NEG EBX //EBX is now -0, signed zero
```

Pseudocode:

```
X = -X
```

Mnemonic: RAND X
Encoding: 023 RM [Segment1] [Constant1]

Generates a random value between 0.0 and 1.0. The random seed is automatically generated by the processor hardware.

Range inclusive of 0.0 and 1.0.

Pseudocode:

```
X = RANDOM(0.0,1.0)
```

Mnemonic: `LOOP X`
Encoding: `024 RM [Segment1] [Constant1]`

Performs a conditional jump to a certain label as long as ECX does not equal zero. Is usually used for creating loops in programs:

```
MOV ECX,100;
LABEL:
  <...>
LOOP ECX; //Repeats 100 times
```

Will only terminate when ECX is equal to zero.

Pseudocode:

```
if ECX <> 0 then
  ECX = ECX - 1
  IP = X
end
```

Mnemonic: `LOOPA X`
Encoding: `025 RM [Segment1] [Constant1]`

Performs a conditional jump to a certain label as long as EAX does not equal zero. Is usually used for creating

loops in programs:

```
MOV EAX,100;
LABEL:
  <...>
LOOP EAX; //Repeats 100 times
```

Will only terminate when EAX is equal to zero. Similar to the `LOOP` instruction.

Pseudocode:

```
if EAX <> 0 then
  EAX = EAX - 1
  IP = X
end
```

Mnemonic: `LOOPB X`
Encoding: `026 RM [Segment1] [Constant1]`

Performs a conditional jump to a certain label as long as EBX does not equal zero. Is usually used for creating loops in programs:

```
MOV EBX,100;
LABEL:
  <...>
LOOP EBX; //Repeats 100 times
```

Will only terminate when EBX is equal to zero. Similar to the `LOOP` instruction.

Pseudocode:

```
if EBX <> 0 then
  EBX = EBX - 1
  IP = X
end
```

Mnemonic: `LOOPD X`
Encoding: `027 RM [Segment1] [Constant1]`

Performs a conditional jump to a certain label as long as EDX does not equal zero. Is usually used for creating loops in programs:

```
MOV EDX,100;
LABEL:
  <...>
LOOP EDX; //Repeats 100 times
```

Will only terminate when EDX is equal to zero. Similar to the `LOOP` instruction.

Pseudocode:

```
if EDX <> 0 then
  EDX = EDX - 1
  IP = X
end
```

Mnemonic: `SPG X`
Encoding: `028 RM [Segment1] [Constant1]`

Makes the specific page read-only. It will clear the write flag of the page specified by the operand, and also set the read flag of the page specified by the operand. see: ?? for more information on the paging system.

Example of use:

```
SPG 1 //Set addresses 128..255 read-only
SPG 2 //Set addresses 256..511 read-only

CPG 1 //Make the 128..255 range writeable again
```

May trigger error `11` (page acccess violation) if the instruction attempts to access a page, and the target page runlevel is greater than the current page runlevel.

This instruction is privileged, meaning it can only be executed from the code page which has runlevel of 0.

In case of an error it will pass the target page number as an interrupt parameter.

Pseudocode:

```
if CurrentPage.Runlevel < Page[X].Runlevel then
  Page[X].Read = 1
  Page[X].Write = 0
else
  Interrupt(11,X)
end
```

Mnemonic: `CPG X`
Encoding: `029 RM [Segment1] [Constant1]`

Makes the specific page readable and writeable. It will set the write flag of the page specified by the operand, and also set the read flag of the page specified by the operand. see: ?? for more information on the paging system.

Example of use:

```
SPG 1 //Set addresses 128..255 read-only
SPG 2 //Set addresses 256..511 read-only

CPG 1 //Make the 128..255 range writeable again
```

May trigger error `11` (page acccess violation) if the instruction attempts to access a page, and the target page runlevel is greater than the current page runlevel.

This instruction is privileged, meaning it can only be executed from the code page which has runlevel of 0.

In case of an error it will pass the target page number as an interrupt parameter.

Pseudocode:

```
if CurrentPage.Runlevel < Page[X].Runlevel then
  Page[X].Read = 1
  Page[X].Write = 1
else
  Interrupt(11,X)
end
```

Mnemonic: `POP X`

Encoding: `030` `RM` `[Segment1]` `[Constant1]`

Pops a value off the processor stack (see: ?? for more information on the processor stack). Will check for underflow by comparing the new stack pointer to the stack size register.

Uses the `ESP` register as the stack pointer, and the `ESZ` register as the stack size pointer.

Example of use:

```
PUSH 10
PUSH 20

POP EAX //EAX is now 20
```

May trigger error `6` (stack overflow/underflow) if the stack pointer goes outside the allowable limits.

Pseudocode:

```
ESP = ESP + 1
if ESP > ESZ then
  ESP = ESZ
  Interrupt(6,ESP)
end
```

Mnemonic: `CALL X`
Encoding: `031` `RM` `[Segment1]` `[Constant1]`

Calls a subroutine of the program. The subroutine will return to this point of execution by executing the `RET` instruction.

This instruction pushes the current instruction pointer to stack, and will restore it upon executing the `RET` instruction. Any damage to the stack data might cause subroutine to fail to return to the original calling point.

The opcode may trigger stack-related errors.

For example:

```
CALL SUBROUTINE0;
CALL SUBROUTINE1;

SUBROUTINE0:
  <...>
RET

SUBROUTINE1:
  <...>
  CALL SUBROUTINE0;
RET
```

May trigger error 6 (stack overflow/underflow) if the stack pointer goes outside the allowable limits.

Pseudocode:

```
Push(IP)
if NoInterrupts then
  IP = X
end
```

Mnemonic: BNOT X
Encoding: 032 RM [Segment1] [Constant1]

Toggles all bits in the number. The number of bits affected by the operation depend on the current setting of the BPREC register (binary precision).

For example:

```
CPUSET 50,8 //Set 8-bit precision

MOV EAX,1
BNOT EAX //EAX is now 254
```

Pseudocode:

```
X = NOT X
```

Mnemonic: FINT X

Encoding: `033 RM [Segment1] [Constant1]`

Rounds down the value. Will round down to the lower integer:

```
MOV EAX,1.9
FINT EAX //EAX = 1.0

MOV EAX,4.21
FINT EAX //EAX = 4.0

MOV EAX,1520.101
FINT EAX //EAX = 1520.0
```

Pseudocode:

```
X = FLOOR(X)
```

Mnemonic: `FRND X`
Encoding: `034 RM [Segment1] [Constant1]`

Rounds the value to the nearest integer:

```
MOV EAX,1.9
FRND EAX //EAX = 2.0

MOV EAX,4.21
FRND EAX //EAX = 4.0

MOV EAX,1520.101
FRND EAX //EAX = 1520.0
```

Pseudocode:

```
X = ROUND(X)
```

Mnemonic: `FFRAC X`

Encoding: `035 RM [Segment1] [Constant1]`

Returns the fractional value of the operand:

```
MOV EAX,1.9
FRND EAX //EAX = 0.9

MOV EAX,4.21
FRND EAX //EAX = 0.21

MOV EAX,1520.101
FRND EAX //EAX = 0.101
```

Pseudocode:

```
X = FRAC(X)
```

Mnemonic: `FINV X`
Encoding: `036 RM [Segment1] [Constant1]`

Finds the inverse of the operand. Checks for the division by zero error.

May trigger error `3` (division by zero) if the second operand was equal to zero.

Will not trigger division by zero error if interrupt flag register `IF` is set to 0.

Pseudocode:

```
if X <> 0 then
  X = 1 / X
else
  Interrupt(3,1)
end
```

Mnemonic: `FSHL X`
Encoding: `038 RM [Segment1] [Constant1]`

Performs an arithmetic shift-left by multiplying the input number by two. The result might be a floating-point value:

```
MOV EAX,100
FSHR EAX //EAX = 200

MOV EAX,8
FSHR EAX //EAX = 16

MOV EAX,4.2
FSHR EAX //EAX = 8.2
```

Pseudocode:

```
X = X * 2
```

Mnemonic: `FSHR X`
Encoding: `039 RM [Segment1] [Constant1]`

Performs an arithmetic shift-right by dividing the input number by two. The result might be a floating-point value:

```
MOV EAX,100
FSHR EAX //EAX = 50

MOV EAX,8
FSHR EAX //EAX = 4

MOV EAX,4.2
FSHR EAX //EAX = 2.1
```

Pseudocode:

```
X = X / 2
```

Mnemonic: `RET`

Encoding: `040`

Returns from a subroutine previous called by `CALL` instruction. This instruction will pop a value off the stack, and use it as a return address.

It can be used for creating subroutines:

```
CALL SUBROUTINE0;
CALL SUBROUTINE1;

SUBROUTINE0:
  <...>
RET

SUBROUTINE1:
  <...>
  CALL SUBROUTINE0;
RET
```

May trigger error `6` (stack overflow/underflow) if the stack pointer goes outside the allowable limits.

Pseudocode:

```
IP = POP()
```

Mnemonic: `IRET`
Encoding: `041`

Returns from an interrupt call. This instruction is very similar to the `RET` instruction, except it restores both the code segment `CS` and the instruction pointer `IP`. It will pop both values off the stack.

The code segment that will be restored correspond to the code segment that the execution was in when the interrupt occured. It is *not* affected by the `IF` flag (interrupt flag).

For example, a typical interrupt body would be:

```
INTERRUPT_HANDLER:
  <...>
IRET;
```

Pseudocode:

```
if EF = 0 then
  IP = Pop()
end
if EF = 1 then
  CS = Pop()
  IP = Pop()
end
```

Mnemonic: `STI`
Encoding: `042`

Set interrupt flag `IF` to 1 *after the next instruction*. This will enable triggering the interrupts (without the `IF` flag interrupts are ignored).

The delay of one instruction is so it would be possible to combine `STI` instruction with an `IRET` or `EXTRET` instruction to provide an atomic interrupt handler (preventing any other interrupts from happening during this interrupt). It’s most efficiently used to prevent external interrupts from happening.

For example:

```
INTERRUPT_HANDLER:
  CLI;
  <...>
  STI;
EXTRET;
```

This instruction is privileged, meaning it can only be executed from the code page which has runlevel of 0.

Pseudocode:

```
NextIF = 1
```

Mnemonic: `CLI`

Encoding: `043`

Clears the interrupt flag `IF`. This will prevent any interrupts from being triggered, and the errors raised by the processor will be ignored. The interrupts can be turned back on using the `STI` instruction.

This instruction is privileged, meaning it can only be executed from the code page which has runlevel of 0.

Pseudocode:

```
IF = 0
```

Mnemonic: `RETF`
Encoding: `047`

Performs a return from a far subroutine call via the `CALLF` instruction. Works similarly to the `IRET` instruction, but does not check for the extended mode.

Will set code segment and instruction pointer to values popped off the stack.

Pseudocode:

```
CS = Pop()
IP = Pop()
```

Mnemonic: `STEF`
Encoding: `048`

Enable the extended mode of the processor. This mode will enable the interrupt table support, and permission checks for the paging system.

The paging system is independant of the extended mode, and can function with or without the extended mode enabled.

It’s possible to disable the extended mode by using the `CLEF` instruction.

This instruction is privileged, meaning it can only be executed from the code page which has runlevel of 0.

Pseudocode:

```
EF = 1
```

Mnemonic: `CLEF`
Encoding: `049`

Disables the extended mode which was enabled by the `STEF` instruction. This will disable interrupt table, and disable the permission checks in the paging system.

The paging system is independant of the extended mode, and can function with or without the extended mode enabled.

This instruction is privileged, meaning it can only be executed from the code page which has runlevel of 0.

Pseudocode:

```
EF = 0
```

Mnemonic: `AND X,Y`
Encoding: `050 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Performs a logical AND operation on the two operands, and writes the result back to the first operand. The result will be 1 if both operands are greater or equal to 1.

Pseudocode:

```
X = X AND Y
```

Mnemonic: `OR X,Y`
Encoding: `051 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Performs a logical OR operation on the two operands, and writes the result back to the first operand. The result will be 1 if either of the operands is greater or equal to 1.

Pseudocode:

```
X = X OR Y
```

Mnemonic: `XOR X,Y`
Encoding: `052 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Performs a logical XOR operation on the two operands, and writes the result back to the first operand. The result will be 1 if just one of operands is greater or equal to 1 (but not two of them at once).

Pseudocode:

```
X = X XOR Y
```

Mnemonic: `FSIN X,Y`
Encoding: `053 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds sine of the second operand, and writes it into the first operand.

Pseudocode:

```
X = Sin(Y)
```

Mnemonic: `FCOS X,Y`
Encoding: `054 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds cosine of the second operand, and writes it into the first operand.

Pseudocode:

```
X = Cos(Y)
```

Mnemonic: `FTAN X,Y`
Encoding: `055 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds tangent of the second operand, and writes it into the first operand.

Pseudocode:

```
X = Tan(Y)
```

Mnemonic: `FASIN X,Y`
Encoding: `056 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds arcsine of the second operand, and writes it into the first operand.

Pseudocode:

```
X = ArcSin(Y)
```

Mnemonic: `FACOS X,Y`
Encoding: `057 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds arccosine of the second operand, and writes it into the first operand.

Pseudocode:

```
X = ArcCos(Y)
```

Mnemonic: `FATAN X,Y`
Encoding: `058 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds arctangent of the second operand, and writes it into the first operand.

Pseudocode:

```
X = ArcTan(Y)
```

Mnemonic: `MOD X,Y`
Encoding: `059 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Finds a remainder after the first operand was divided by the second operand, and returns the result to the first operand. If the input is a floating-point value, then it will return the value of $X - n * Y$, where n is the quotient of X / Y, rounded toward zero to an integer.

Pseudocode:

```
X = X FMOD Y
```

Mnemonic: `BIT X,Y`
Encoding: `060 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Tests whether specific bit is set in the given number, and writes the result to the `CMPR` register. It's possible to check the result of this operation using the conditional branching opcodes (see:s ??, ??):

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

Pseudocode:

```
CMPR = Yth bit of X
```

Mnemonic: `SBIT X,Y`
Encoding: `061 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Sets specific bit of the first operand (see: ??):

```
MOV EAX,105 //1101001
SBIT EAX,1
//EAX = 107   1101011
```

Pseudocode:

```
Yth bit of X = 1
```

Mnemonic: `CBIT X,Y`
Encoding: `062 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Clears specific bit of the first operand (see: ??):

```
MOV EAX,107 //1101011
CBIT EAX,6
//EAX = 43    0101011
```

Pseudocode:

```
Yth bit of X = 0
```

Mnemonic: `TBIT X,Y`
Encoding: `063 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Toggles specific bit of the first operand (see: ??):

```
MOV EAX,43 //0101011
TBIT EAX,0
//EAX = 42   0101010
```

Pseudocode:

```
Yth bit of X = 1 - Yth bit of X
```

Mnemonic: BAND X,Y
Encoding: 064 RM [Segment1] [Segment2] [Constant1] [Constant2]

Pseudocode:

Mnemonic: BOR X,Y
Encoding: 065 RM [Segment1] [Segment2] [Constant1] [Constant2]

Pseudocode:

Mnemonic: BXOR X,Y
Encoding: 066 RM [Segment1] [Segment2] [Constant1] [Constant2]

Pseudocode:

Mnemonic: BSHL X,Y
Encoding: 067 RM [Segment1] [Segment2] [Constant1] [Constant2]

Pseudocode:

Mnemonic: BSHR X,Y
Encoding: 068 RM [Segment1] [Segment2] [Constant1] [Constant2]

Pseudocode:

Mnemonic: `JMPF X,Y`
Encoding: `069 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `EXTINT X`
Encoding: `070 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CNE X`
Encoding: `071 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CNZ X`
Encoding: `071 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CG X`

Encoding: `073 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CNLE X`
Encoding: `073 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CGE X`
Encoding: `074 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CNL X`
Encoding: `074 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CL X`
Encoding: `075 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CNGE X`
Encoding: `075 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CLE X`
Encoding: `076 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CNG X`
Encoding: `076 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CE X`
Encoding: `077 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CZ X`
Encoding: `077 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `MCOPY X`
Encoding: `078 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `MCOPY X`
Encoding: `078 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `MXCHG X`
Encoding: `079 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `MXCHG X`
Encoding: `079 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `FPWR X,Y`

Encoding: `080 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `XCHG X,Y`
Encoding: `081 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `FLN X,Y`
Encoding: `082 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `FLOG10 X,Y`
Encoding: `083 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `IN X,Y`
Encoding: `084 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `OUT X,Y`
Encoding: `085 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `FABS X,Y`
Encoding: `086 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `FSGN X,Y`
Encoding: `087 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `FEXP X,Y`
Encoding: `088 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CALLF X,Y`
Encoding: `089 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `FPI X`
Encoding: `090` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `FE X`
Encoding: `091` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `INT X`
Encoding: `092` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `TPG X`
Encoding: `093` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `FCEIL X`

Encoding: `094 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `ERPG X`
Encoding: `095 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `WRPG X`
Encoding: `096 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `RDPG X`
Encoding: `097 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `TIMER X`
Encoding: `098 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `LIDTR X`
Encoding: `099` `RM` `[Segment1]` `[Constant1]`

Pseudocode:

Mnemonic: `JNER X`
Encoding: `101` `RM` `[Segment1]` `[Constant1]`

Pseudocode:

Mnemonic: `JNZR X`
Encoding: `101` `RM` `[Segment1]` `[Constant1]`

Pseudocode:

Mnemonic: `JMPR X`
Encoding: `102` `RM` `[Segment1]` `[Constant1]`

Pseudocode:

Mnemonic: `JGR X`
Encoding: `103` `RM` `[Segment1]` `[Constant1]`

Pseudocode:

Mnemonic: `JNLER X`
Encoding: `103` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JGER X`
Encoding: `104` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JNLR X`
Encoding: `104` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JLR X`
Encoding: `105` `RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JNGER X`

Encoding: `105 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JLER X`
Encoding: `106 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JNGR X`
Encoding: `106 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JER X`
Encoding: `107 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `JZR X`
Encoding: `107 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `LNEG X`
Encoding: `108 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `EXTRET`
Encoding: `110`

Pseudocode:

Mnemonic: `IDLE`
Encoding: `111`

Pseudocode:

Mnemonic: `NOP`
Encoding: `112`

Pseudocode:

Mnemonic: `PUSHA`
Encoding: `114`

Pseudocode:

Mnemonic: `POPA`
Encoding: `115`

Pseudocode:

Mnemonic: `STD2`
Encoding: `116`

Pseudocode:

Mnemonic: `LEAVE`
Encoding: `117`

Pseudocode:

Mnemonic: `STM`
Encoding: `118`

Pseudocode:

Mnemonic: `CLM`

Encoding: `119`

Pseudocode:

Mnemonic: `CPUGET X,Y`
Encoding: `120` `RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CPUSET X,Y`
Encoding: `121` `RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `SPP X,Y`
Encoding: `122` `RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CPP X,Y`
Encoding: `123` `RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `SRL X,Y`
Encoding: `124 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `GRL X,Y`
Encoding: `125 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `LEA X,Y`
Encoding: `126 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `BLOCK X,Y`
Encoding: `127 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CMPAND X,Y`
Encoding: `128 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CMPOR X,Y`
Encoding: `129 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MSHIFT X,Y`
Encoding: `130 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `SMAP X,Y`
Encoding: `131 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `GMAP X,Y`
Encoding: `132 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `RSTACK X,Y`

Encoding: `133 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `SSTACK X,Y`
Encoding: `134 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `ENTER X`
Encoding: `135 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `VADD X,Y`
Encoding: `250 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VSUB X,Y`
Encoding: `251 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VMUL X,Y`
Encoding: `252 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VDOT X,Y`
Encoding: `253 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VCROSS X,Y`
Encoding: `254 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VMOV X,Y`
Encoding: `255 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VNORM X,Y`
Encoding: `256 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VCOLORNORM X,Y`
Encoding: `257 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `LOOPXY X,Y`
Encoding: `259 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

```
IF EDX > 0 THEN
IP = X
IF ECX > 0 THEN
ECX = ECX - 1
ELSE
EDX = EDX - 1
ECX = Y
END
END
```

Mnemonic: `MADD X,Y`
Encoding: `260 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MSUB X,Y`
Encoding: `261 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MMUL X,Y`
Encoding: `262 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MROTATE X,Y`
Encoding: `263 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MSCALE X,Y`
Encoding: `264 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MPERSPECTIVE X,Y`
Encoding: `265 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MTRANSLATE X,Y`
Encoding: `266 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MLOOKAT X,Y`
Encoding: `267 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MMOV X,Y`
Encoding: `268 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VLEN X,Y`
Encoding: `269 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `MIDENT X`
Encoding: `270 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `VMODE X`
Encoding: `273 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `VDIV X,Y`
Encoding: `295 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `VTRANSFORM X,Y`
Encoding: `296 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

HL-ZASM is a high-level assembler compiler and a programming language. It provides support for all the basic C high-level structures, while retaining compatibility with ZASM2 assembly code.

There are certain characters and keywords that are reserved in the HL-ZASM compiler, and cannot be used as, for example, variable names. A variable name/function name/identifier can be any alphanumeric character, underscore (`_`). In some cases the dot (`.`) symbol can be part of a label name.

The following keywords are reserved: `GOTO`, `FOR`, `IF`, `ELSE`, `WHILE`, `DO`, `SWITCH`, `CASE`, `CONST`, `RETURN`, `BREAK`, `CONTINUE`, `EXPORT`, `FORWARD`, `DB`, `ALLOC`, `SCALAR`, `VECTOR1F`, `VECTOR2F`, `UV`, `VECTOR3F`, `VECTOR4F`, `COLOR`, `VEC1F`, `VEC2F`, `VEC3F`, `VEC4F`, `MATRIX`, `STRING`, `DB`, `DEFINE`, `CODE`, `DATA`, `ORG`, `OFFSET`, `VOID`, `FLOAT`, `CHAR`, `INT48`, `VECTOR`, `PRESERVE`, `ZAP`.

The exact meaning and use of these keywords is detailed in the following chapters.

HL-ZASM supports the following common assembly syntax (Intel-like):

```
mov eax,123; //Moving constant into register
mov eax,ebx; //Moving register into register

add eax,123; //2-operand instruction
jmp 123; //1-operand instruction on constant
jmp eax; //1-operand instruction on register
```

The two operands always follow the instruction. The first one is always the destanation operand, and the second one is the source operand.

It is possible to use both the traditional ZASM memory access symbol (#) or the more common square brackets (`[]`). Both are considered valid.

The CPU also supports segment prefixes to offset memory reads (see: ?? for more information on segments). It is possible to use any general-purpose or segment register as a segment prefix to any common operand.

HL-ZASM allows for an ’inverted’ syntax, which is when a segment prefix follows the operand itself (for example ’es:eax’ instead of ’eax:es’).

When using common syntax for reading a value from memory, a segment prefix must be included inside the brackets. It is possible to use the plus sign (+) instead of a colon (`:`), but the common memory access syntax must be used.

It is also possible to use a constant value as a segment prefix for a register via inverted syntax support. It is currently not possible to access a memory cell using a constant offset by a constant, but this will be fixed later.

The following syntax for using memory access and segments is supported:

```
//ZASM2-style reading from memory
mov eax,#100;
mov eax,#eax;
mov eax,es:#100;
mov eax,es:#eax;
mov eax,eax:#100;
//mov eax,eax:#es; //Invalid: cannot use segment register as pointer

//Common-style reading from memory
mov eax,[100];
mov eax,[eax];
mov eax,[es:eax];
mov eax,[es+eax];
//mov eax,[eax:es]; //Invalid: cannot use segment register as pointer
//mov eax,[eax+es]; //Invalid: cannot use segment register as pointer

//Using constants with segments
mov eax,es:100;
mov eax,eax:100;
mov eax,100:es;
mov eax,100:eax;
//mov eax,100:200; //Invalid

//Using registers with segments
mov eax,ebx:ecx;
mov eax,es:ebx;
mov eax,ebx:es; //Invalid, but translated into the valid 'es:ebx'
//mov eax,fs:es; //Invalid, as no valid syntax is possible
```

It is possible to use constant expressions, hexadecimal values and label values in HL-ZASM:

```
//Constant integers
mov eax,255;   //EAX = 255
mov eax,0255;  //EAX = 255
mov eax,0x1FF; //EAX = 511

//Negative integers
mov eax,-255;   //EAX = -255
mov eax,-0x1FF; //EAX = -511

//Floating point numbers and expressions
mov eax,1e4;         //EAX = 10000
mov eax,1.53e-3;     //EAX = 0.00153
mov eax,1.284;       //EAX = 1.248
mov eax,2.592+3.583; //EAX = 6.175
mov eax,1e2+15;      //EAX = 115
//mov eax,1e-5+0.034;//Does not parse (bug)
mov eax,0xFF+100;    //EAX = 355

//Complex expressions
define labelname,512;
mov eax,10+15;         //EAX = 25
mov eax,10*(15+17*24); //EAX = 4230
mov eax,10+labelname*53-10*(17-labelname); //EAX = 32096
```

Labels are used to branch to specific portions of code. Each label name is actually a number with a name assigned to it. That means that variable names and label names all have valid constant values, and can be used in expressions as parameters for various macros:

```
labelname:
  jmp labelname;
  mov eax,labelname+3;

labelName: //case sensitive
  jmp labelName;
```

The `define` macro allow a label name to be assigned to any constant expression:

```
define defVar1,120;
define defVar2,50+defVar1;
//define defVar3,labelname2*10; //Does not parse, bug

labelname2:
  mov eax,defVar1; //120
  mov eax,defVar2; //170
//mov eax,defVar3; //170
```

The `db` macro is used to write specific values to memory, bypassing any preprocessing. They will be written into the code segment:

```
db 100; //Outputs 100
db 0xFF; //Ouputs 255
db labelname2+10;
db 'This is a string',0;
db 15,28,595,'string',35,29;
```

The `alloc` macro allocates space and allows it to be accessed by name, if one is given. It will be located at the current position in memory (if data and code are located in the same segment), or in a separate data segment, depending on the compiler preferences. The first parameter must either be a valid ident or a constant expression that does not start with an ident:

```
alloc var1,100,72; //alloc label,size,value, same as 'var1: db 72,72,72,...'
alloc var2,100;    //alloc label,value, same as 'var2: db 100'
alloc var3;        //alloc label, same as 'var3: db 0'
alloc 120;         //alloc size, same as 'db 0,0,0,0,0....'

define allocsize,120;
alloc 0+allocsize; //must not start with an ident name
```

There are also special macros for defining vector variables and string variables. The basic syntax is `VECTOR_MACRO NAME,DEFAULT_VALUE`. These are the examples of all available vector macros:

```
scalar name,...; //For example "scalar x,10"
vector1f name,...;
vector2f name,...;
vector3f name,...;
vector4f name,...;
vec1f name,...;
vec2f name,...;
vec3f name,...;
vec4f name,...;
uv name,...;
color name,...;
matrix name; //No default initializer
```

It’s possible to get pointer to each of the members of the vector variable:

```
vector3f vec1;
mov #vec1.x,10;
mov #vec1.y,20;
*(vec1.z) = 30;

uv texcoord1;
mov #texcoord1.u,10;
mov #texcoord1.v,20;

color col1;
mov #col1.r,10;
mov #col1.g,20;
mov #col1.b,30;
mov #col1.a,40;
```

Matrix and vector variables can be used along with the vector extension of the ZCPU:

```
matrix m1;
matrix m2;
vector4f rot,0,0,1,45;

mident m1; //Load identity matrix
mrotate m2,rot; //Load rotation matrix

mmul m1,m2; //Multiply two matrices
```

There's also an additional macro which is not directly connected to creating variables, but it allows to change the current write pointer of the program (the location in memory at which program is being written). The macro is called `ORG`:

```
//Write pointer 0
alloc 64;
//Write pointer is now 64
alloc 1;
//Write pointer is now 65

ORG 1000;
//Write pointer is now 1000
```

This macro can be used for slightly more advanced management of the code generation.

There are also two special helper macros available that can be used to simplify variable declaration in the simple program: the `DATA` and the `CODE` macros. They are used like this:

```
DATA; //Data section
  alloc var1;
  alloc var2;
  ......
  subroutine1:
    ....
  ret
  ....
CODE; //Main code section
  call subroutine1;
  call subroutine2;
  ......
```

These macros are expanded into the following code:

```
//DATA
jmp _code;

//CODE
_code:
```

HL-ZASM has a built-in expression generator. It is possible to generate complex expressions that can involve function calls, registers, variables in a similar way as they are generated in the C code.

Here are the several examples of various expressions:

```
EAX = 0 //Same as "mov eax,0"
EAX += EBX //Same as "add EAX,EBX"

main()
print("text",999)
R17 = solve("sol1",50,R5)
R2 = R0 + R1*128
c = *ptr++;

R0 = (noise(x-1, y) + noise(x+1, y) + noise(x, y-1) + noise(x, y+1)) /  8
Offset = 65536+32+MAX_DEVICES+udhBusOffset[busIndex*2]
```

Extra care must be taken while using the expression generator, since it will use the 6 general purpose registers (`EAX` .. `EDI`) as temporary storage for evaulating the expressions.

There is support for parsing constant expressions, which are reduced to a single constant value during compile time. It is possible to use the following syntax features in the expression parser:

Syntax	Description
`-X, +X`	Specify value sign, or negate the value
`&globalvar`	Pointer to a global variable (always constant)
`&stackvar`	Pointer to a stack variable (always dynamic)
`&globalvar[..]`	Pointer to an element of a global array
`&stackvar[..]`	Pointer to an element of a stack-based array
`1234`	Constant value
`"string"`	Pointer to a constant string, can be only used inside functions
`’c’`	Single character
`pointervar`	Constant pointer (or a label)
`EAX`	Register
`func(...)`	Function call
`var[...]`	Array access
`*var`	Read variable by pointer
`expr + expr`	Addition
`expr - expr`	Subtraction
`expr * expr`	Product
`expr / expr`	Division
`expr ^^expr`	Raising to power
`expr &expr`	Binary AND
`expr |expr`	Binary OR
`expr ^expr`	Binary XOR

`expr &&expr`	Logic AND
`expr ||expr`	Logic OR
`expr = expr`	Assign (returns value of left side AFTER assigning)
`expr > expr`	Greater than
`expr >= expr`	Greater or equal than
`expr == expr`	Equal
`expr <= expr`	Less or equal than
`expr < expr`	Less than
`expr++`	Increment (returns value BEFORE incrementing)
`expr--`	Decrement (returns value BEFORE decrementing)
`++expr`	Increment (returns value AFTER incrementing)
`--expr`	Decrement (returns value AFTER decrementing)

Expression generator supports expressions inside opcodes as well.

It’s possible to declare variables in HL-ZASM the same way they are declared in the C programming language. It must be noted, however, that there is a difference between variables declared with this way, and variables declared using ZASM2 macros. It will be detailed a bit more on this problem further down.

Variables may be declared either in global space (data segment), or local space (stack segment/stack frame). The variables declared in the global space are created at compile-time, while the stack-based variables are allocated runtime.

Take note that variable declarations and variable definitions are two different things. Variables are declared as shown below, but they can be defined using ZASM2 macros. Consider the following code:

```
float x;
```

```
mov x,10;  //Set variable X to 10
mov #x,10; //Same as '*(x) = 10'

scalar y;
mov y,10; //Does nothing, same as 'mov 123,10'
mov #y,10; //Sets variable Y to 10
```

There are three types supported by the compiler right now: `float` (64-bit floating point variable), `char` (64-bit floating point character code), `void` (undeclared type/no assigned behaviour). Unlike the usual compilers, all three types work the same way, and are essentially the same thing:

```
float x;
float y,z;
char* w;
char** u; //Pointer to a pointer

//In all of these examples 'a' is a pointer to a character
char * a;
char* a;
char *a;
char b,*a;
```

It’s possible to use * symbol to create pointers to variables. There is no difference on whether there are any whitespaces between the asterisk and the variable name (the asterisk belongs to the variable name).

Arrays of constant size may be declared as variables. If array is located in global scope, *all members will usually default to zero, although this is might not be the case if program is dynamically loaded by the OS*. If the array is declared in local scope, the contents of it may be undeclared. :

```
float arr[256];

arr[100] = 123;
R0 = arr[R1];
```

There is no way to declare 2-dimensional arrays right now, but there will be support for that feature at some point.

It's possible to use initializers for the arrays and the variables. It's only possible to use constant expressions as initializers for the global scope variables and arrays (there is no limit on what expressions may be used in the stack-based mode):

```
//Global scope
float x = 10;
float garr1[4] = { 0, 5, 7, 2 };
float garr2[8] = { 1, 2 }; //Missing entries filled with zeroes
float garr3[8] = { 1, 2, 0, 0, 0, 0, 0, 0 }; //Same as garr2

//Local scope
void func(float x,y) {
  float z = x + y * 128;
  float larr1[4] = { x, y, 0, 0 };
  float larr2[16] = { x, 0, y }; //Missing entries filled with zeroes
}
```

There is also support for creating variables, which are stored in the processors registers. To do this their type must be prepended with the `register` keyword. Local variables in registers work much faster than the stack-based local variables, but they cannot be an array type (they can be a pointer though):

```
void strfunc(char *str) {
  register char *ptr = str;
  *ptr++ = 'A';
}
```

There is a limit on how much local register variables there can be.

It’s possible to use C-style function declaration syntax, although only one style is currently supported:

```
return_type function_name(param_type1 name1, name2, param_type2 name3) {
  ....
  code
  ....
}
```

Parameters are defined the same way the variables are defined, see: ??. It’s possible to use constant-size arrays as variables into function, but *there is no way to actually pass them into the function right now*. It’s also possible to use sizeless array as a parameter (see examples).

Examples of declaring functions:

```
void main() { .. }
float func1() { .. }
float func2(float x, y, float z) { .. }

char strcmp(char* str1, char* str2) { .. }
float strlen(char str[]) { .. } //'char str[]' is same as 'char* str'

//no way to actually pass an array like this:
void dowork(char str[256]) { .. }
```

There are two kinds of function declarations - normal declarations, and forward declarations. Normal function declarations can be anywhere in the code, and there’s no restriction on using them from any other part of the program. Forward function declarations must always precede the function use (or they must be declared beforehand without the function body).

The forward function declarations allow to perform strict arguments check, and they allow overriding the function to have different parameter lists while having the same name. A function can be forward-declared by using the `FORWARD` keyword before the function type:

```
//Forward declarations before use
forward void func(); //First function
forward void func(float x); //Second function
forward void func(float x, *y);

func(); //Calls first function
func(10); //Calls second function

func2(); //COMPILE ERROR: function 'func2' is not declared

//Actual function bodies
forward void func() { //First function
  ...
}
forward void func(float x) { //Second function
  ...
}
forward void func(float x, *y) {
  ...
}
forward void func2() {
  ...
}
```

Unlike the forward function declarations, the normal functions can be used from anywhere in the code, but they do not provide strict argument type checks:

```
void func1() { .. }

func1();
func2(); //Works even thought it's not yet declared

void func2() { .. }
```

There’s also an additional keyword that can be used when defining functions - `EXPORT`. If this keyword is used, the function name will be preserved in the generated library (see: ?? for more informations on how to generate libraries). The compiler will also add a declaration for this function automatically in the generated library file:

```
export void func1() { .. }
void func2() { .. } //Function name will be mangled in the resulting file
```

HL-ZASM uses the `cdecl` calling convention. The function result is passed via the EAX register. If the function is not forward-declared, or if it has variable argument count, then the ECX register must be set to the parameter count:

```
R0 = func(a,b,c);

//Same as:
push c;
push b;
push a;
mov ecx,3; //Optional if forward-declared
call func;
add esp,3; //Must clean up stack
mov r0,eax; //Return result

main();
```

```
//Same as:
mov ecx,0;
call func;
```

The function will modify EAX, ECX, EBP registers (along with the ESP register), and may modify all other registers unless they are marked as preserved (see ??).

The HL-ZASM uses the ZCPU stack frame management instructions (`ENTER`, `LEAVE`) for creating a stack frame for the function. It will pass number of local variables into the `ENTER` instruction, so it will create a stack frame with pre-allocated space for local variables.

The function would generate such code:

```
void func(float x,y) {
  float z,w;
  ...
}

//Generates:
func:
  enter 2;
    ....
  leave;
  ret
```

All the access to variables on stack is done via the `RSTACK`, `SSTACK` instructions. The compiler will use `EBP:X` as the stack offset, where `EBP` is the stack frame base register, and `X` is the offset of the target value on stack. For example:

```
void func(float x,y) {
  float z,w;
}

//These values would be laying on stack
//[ 3] Y
//[ 2] X
//[ 1] Return address
//[ 0] Saved value of EBP (at function call)
//[-1] Z
//[-1] Z
```

```
//Therefore this would be valid:
rstack R0,EBP:2  //R0 = X
rstack R1,EBP:3  //R1 = Y
rstack R2,EBP:-1 //R2 = Z
rstack R3,EBP:-2 //R3 = W
```

Therefore it's possible to generate a stack trace using the following code:

```
void stack_trace() {
  char* returnAddress,savedEBP;

  R0 = EBP; //'Current' EBP
  while ((R0 > 0) && (R0 < 65535)) {
    rstack returnAddress,R0:1;
    rstack savedEBP,R0:0;

    add_to_trace(returnAddress);
    R0 = savedEBP;
  }
}
```

If the current program makes use of any ZCPU registers, it must mark them as preserved so the expression generator does not use those registers for purpose of calculating expressions.

The compiler will give out warning when unpreserved registers are being used. Right now only EAX-EDI registers must be marked as preserved (since only those are used for expression generator):

```
void func(float x,y) {
  preserve EAX, EBX;

  //Expression generator will never change EAX or EBX registers
  EAX = 123;
  EBX = x*10 + y;
}
```

The HL-ZASM compiler supports common C control structures, although right now the support is limited.

The conditional branching can be done via the `if` construct. The `else` clause, and the

`else if` are supported. It’s possible to branch into a single expression, or into an entire block of code too:

```
//Can use blocks
if (expression) {
  ....
} else if (expression) {
  ....
} else {
  ....
}

//Can avoid using blocks:
if (expression) expression;
if (expression) expression1 else expression2;
```

HL-ZASM supports `for` loops. The syntax is:

```
for (initializer; condition; step) { ... }
```

where `initializer` is the expression that will be executed to setup the loop, `condition` is the condition that is tested on each step, and `step` is the expression executed after each step. For example:

```
float x;
for (x = 0; x < 128; x++) { .. } //Loop for X from 0 to 127
for (x = 128; x > 0; x--) { .. } //Loop for X from 128 to 1
for (;;) { .. } //Infinite loop
```

It’s possible to use the `while` loop (but no support for `do - while` loops yet):

```
while (expression) { ... }
while ((x < y) && (x > 0)) { ... }
while (1) { ... } //Infinite loop
```

The `break` keyword can be used to end the currently executed loop, for example:

```
while(1) {
  if (condition) {
    break;
  }
  ....
}
```

It’s possible to use the `continue` keyword to go on to the next step in the loop. For example:

```
float x;
for (x = 0; x < 128; x++) {
  if (x == 50) { //Skip iteration 50
    continue;
  }
}
```

It’s also possible to define labels, and then jump to those labels by defining them the same way they are defined in ZASM2, and using `GOTO`:

```
....
  goto label1; //Jumps to label1
....
  label1:
```

When using the `GOTO` it’s also possible to pass a complex expression into it:

```
goto function_entrypoints[10];
```

The HL-ZASM preprocessor supports C-style preprocessor macros. Preprocessor macros are always last on the current line (it is not possible to write two preprocessor macros on same line).

By default programs compiled with HL-ZASM have no attached runtime library, and would require rewriting all the basic routines. It is possible to link to a runtime library of a choice though. The default runtime library is called `ZCRT`, and it allows to boot up the ZCPU without any additional software.

The CRT library can be picked with the following preprocessor macro. It must be located in the first line of code, before any other code is generated, otherwise it will not work correctly (macro is case-insensitive):

```
#pragma CRT ZCRT
```

This macro will add CRT folder as one of the search paths, and include the main CRT file:

```
#pragma SearchPath lib\zcrt\
#include <zcrt\main.txt>
```

This makes it possible to use the default libraries that belong to that runtime library. see: ?? for more information on the `ZCRT` library.

It is possible to use the C style definition preprocessor macros (it is not possible to define preprocessor functions yet though). It supports the `#define`, `#ifdef`, `#elseif`, `#else`, `#endif` and the `#undef` macros:

```
#define DEF1
#define DEF2 1234

#ifdef DEF1
  func(DEF2) //same as func(1234)
  ...
#elseif DEF2
  ....
#else
  ...
#endif

#undef DEF1
```

The preprocessor supports including external files using the `#include` macro:

```
#include "filename"
```

```
#include <filename>
```

The `#include "filename"` macro will include file from the current working directory. This is the same directory the main (first) compiled source file is located in. The other version of this macro includes file relative to the base directory (`CPUChip`).

If file is not found, it will also be searched on one of the search paths.

The preprocessor also supports the ZASM2 file include syntax:

```
##include## filename
same as
#include <filename>
```

There are several special compiler commands available through the `#pragma` macro.

The `#pragma set` macro allows user to modify settings of the compiler. Example of the syntax (everything is case-sensitive):

```
#pragma set OutputResolveListing true
```

There are the following settings available:

Name	Default	Description
CurrentLanguage	HLZASM	Current compiler language. Can be HLZASM or ZASM2
CurrentPlatform	CPU	Target platform. Defines the feature set, cannot be modified
MagicValue	-700500	The magic value is used in place of an erroneous constant value
OptimizeLevel	0	Optimizer level. 0 is none, 1 is low, 2 is high. Not supported right now.
OutputCodeTree	false	Output code tree
OutputResolveListing	false	Output code listing for resolve stage
OutputFinalListing	false	Output code listing for final stage
OutputTokenListing	false	Output tokenized sourcecode
OutputBinaryListing	false	Output final binary dump as listing
OutputDebugListing	false	Output the debug data as listing
OutputToFile	false	Output listings to files instead of to the console
OutputOffsetsInListing	true	Output binary offsets in listings
OutputLabelsInListing	true	Output label names in final listing
GenerateComments	true	Generate extra comments in output listing
FixedSizeOutput	false	Output fixed-size instructions (can be toggled at any time)

SeparateDataSegment	false	Puts all variables into separate data segment Not supported right now.
GenerateLibrary	false	Generate a precompiled library. see: ??
AlwaysEnterLeave	false	Always generate the enter/leave blocks in functions
NoUnreferencedLeaves	true	Do not compile functions and variables which are not used by the program

The `#pragma language` macro can be used in place of setting the language via changing the compiler variables:

```
#pragma language zasm
#pragma set CurrentLanguage ZASM2
```

The `#pragma crt` macro can be used to attach a C runtime library. see: ?? for more information.

The `#pragma cpuname` macro is used to assign a specific name to the target processor:

```
#pragma CPUName ACPI Power Controller
```

There are several preprocessor definitions and special labels available for use by the programmer:

Bit	Description
`__PTR__`	Current write pointer
`__LINE__`	Current line number
`__FILE__`	Current file name (a string)
`__DATE_YEAR__`	Current year (at compile time)
`__DATE_MONTH__`	Current month (at compile time)
`__DATE_DAY__`	Current day (at compile time)
`__DATE_HOUR__`	Current hour (at compile time)
`__DATE_MINUTE__`	Current minute (at compile time)
`__DATE_SECOND__`	Current second (at compile time)
`__PROGRAMSIZE__`	Total size of the program in bytes
`programsize`	Total size of the program in bytes (ZASM2 compatibility macro)

no chapter

no chapter

no chapter but lots of incredible fun

Previously unknown label, variable, or function call was never declared in the current scope.

Variable or function with this name was already defined previously in the scope. Error message will point to initial definition.

Expression generator expects identifier to follow (variable/function name, etc).

There was something unexpected in the source code at that position. Can also indicate invalid expression syntax.

Unable to get pointer of the given identifier.

The ZCPU does not support the given instruction operand. Can indicate that segment prefix is used for segment register access, or for port access.

Given opcode is not supported by the current architecture.

It’s only possible to define arrays which have constant size. There is no support for declaring variable-sized arrays.

There is no support for having complex expressions as variable initializers right now.

Compiler does not support preserving registers in global scope right now.

A constant value is expected, and it must not rely on values of any unknown variables or labels.

Invalid syntax is being used for some specific language structure.

This error indicates one of the following things:

- **There are no more free registers to use due to too much local register variables allocated.**
- **The expression being generated was too complex (there are not enough unallocated registers).**
- **Internal compiler error.**

Invalid runtime library name/library is not found.

File was not found in the specified folder, and it was not found on one of the search paths.

Internal compiler error (is not an error that can be worked around).

ZGPU is a vector graphics processor, which runs a certain set of instructions to display an image on screen. It has two basic modes of operation - frame-based mode, in which the processor core will execute a certain subprogram each time the new frame must be drawn to screen, and asynchonous mode, which allows to run ZGPU instructions without being tied to screen refreshes.

Even thought ZGPU works with vector graphics the final result is being rasterized into one of two buffers. These two buffers are called the *front* buffer, and the *texture* buffer. Both buffers are 512x512 pixels in size, and can be used by the programmer. There is support for a natively vertex mode for drawing graphics, but it has certain limitations.

By default the ZGPU runs the frame-based mode, which makes use of the *front* buffer to store data, which will then

be displayed on screen. When running in asynchonous mode the graphics are drawn to one of the two buffers, and are then being output to screen.

Most of the features in the ZGPU are controlled via internal registers (see ?? for a complete list of all the registers). They are located in the register memory, which starts at addesses 63488 and ends at 65535.

The memory area between addresses 65536 and 131071 (inclusive) is reserved as an additional external memory bus. It is very slow, but it allows GPU to communicate with other devices. If several GPU's are running at the same time, the access to these memory areas will be concurrent, and so additional synchronization is required to prevent race conditions or collisions of any sort.

The simpliest program GPU can execute is the following (it makes use of the frame-based mode of execution):

```
dtest; //Output test pattern to screen
dexit; //Finish execution
```

It's possible to setup asynchonous rendering instead of frame-based rendering:

```
//Setup entrypoints
dentrypoint 0,DrawThread;
dentrypoint 4,AsyncThread;

//Disable hardware clear so drawing thread does not wipe
//the image on screen
mov #regHWClear,0;
//Set asynchronous thread frequency (speed)
mov #regAsyncFreq,200000;
```

```
//Run the thread
mov #regAsyncClk,1;
dexit;

DrawThread: //Do nothing
dexit;

AsyncThread:
  dbegin;
    dtest;
  dend;

  dvsync; //Add frame synchonization
jmp AsyncThread;
```

ZGPU makes use of the vector extension (see: ??) which allows it to work with matrices and vectors:

```
//Generate rotation, translation matrices
mrotate mRotateMatrix,vRotate;
mtranslate mTranslateMatrix,vTranslate;

//Create model matrix
mmov mModelMatrix,mRotateMatrix;
mmul mModelMatrix,mTranslateMatrix;
```

The ZGPU supports 2D and 3D graphics , which must be drawn as polygons:

```
dvxdata_2f polydata,4; //4 vertices
dvxdata_3f cubedata,12; //12 triangles

....

polydata:
db  0,  0;
db 10,  0;
db 10, 10;
db  0, 10;

cubedata:
db -1,-1,-1; //Triangle 1
db  1,-1,-1;
db  1, 1,-1;

db -1,-1,-1; //Triangle 2
db  1, 1,-1;
db -1, 1,-1;

...
```

Polygons can be drawn in both normal mode, and indexed mode. They can also be drawn solid-colored, textured, or wireframe:

```
//Load array of vertices
mov #regVertexArray,cube_varray;
dvxdata_3f cube_idxarray,12; //Draw all faces
dvxdata_3f_wf cube_idxarray,6; //Draw faces 1-3 as wireframe

//Load array of vertices with texture coords
mov #regVertexArray,cube_varray_tex;
dvxdata_3f_tex cube_idxarray,12; //Draw all faces, textured

cube_varray:
  db -1,-1,-1; //0
  db -1,-1, 1; //1
  db -1, 1,-1; //2
  db -1, 1, 1; //3
  db  1,-1,-1; //4
  db  1,-1, 1; //5
  db  1, 1,-1; //6
  db  1, 1, 1; //7

cube_idx:
  db 0,4,6;  db 0,6,2; //Face 1
  db 5,1,7;  db 1,3,7; //Face 2
  db 4,0,5;  db 0,1,5; //Face 3
  db 2,6,7;  db 3,2,7; //Face 4
  db 0,2,3;  db 1,0,3; //Face 5
  db 6,4,7;  db 4,5,7; //Face 6
```

It supports vertex buffer, which serves as temporary storage for 2D/3D data before it's rendered on screen. This allows to provide depth-sorting within the buffer, and other features:

```
//Enable vertex buffer features
denable 0; //Vertex buffer
denable 1; //ZSorting
denable 2; //Lighting

//Add commands to vertex buffer
dcolor cube_color;
dvxdata_3f cube_data,12;

//Flush vertex buffer
dvxflush;

//Disable vertex buffer and its features
ddisable 0;
```

There is support for texturing using both custom textures, and textures available externally:

```
mov #regVertexMode,1; //Enable vertex mode
mov #regTexSize,128; //Texture size
denable 5; //Enable custom texture mapping
dcolor white; //Set color to white

dtexture 2; //Pick texture #2
drectwh rect_pos,rect_size;

...
ddisable 5; //Disable custom texture mapping

dxtexture texture_name; //Pick texture
drectwh rect_pos,rect_size;

....
string texture_name,"brick/brickfloor001a";
```

ZGPU supports various 2D transformations to move shapes on screen:

```
dmove target_pos; //move to position
drotatescale 1.23,2; //Rotate by 1.23 radians, and scale up twice
drect rect_pos1,rect_pos2; //Draw rectangle around 0,0 point

....
```

```
vector2f target_pos,256,256; //Screen center

vector2f rect_pos1,-50,-50; //Two endpoints for rectangle
vector2f rect_pos2, 50, 50;
```

There is also support for performing similar transformations on the textures, independantly of the previous transformations (rotation is performed around texture centerpoint usually):

```
denable 5; //Enable custom texturing
mov #regTexRotation,1.23; //Rotate texture by 1.23 radians
mov #regTexOffsetV,0.2; //Offset V coordinates by 0.2
dvxtexpoly horizon_polygon,4;
```

The basic graphics output in GPU makes use of the few control instructions (such as DCOLOR, which changes the current drawing color), and the few drawing instructions (for example DRECT, DLINE, etc).

The basic graphics output only requires use of the frame-based drawing mode. The GPU will clear the screen to black each frame, and set the current color to black too. To draw something the color must first be set to wanted color, and then some drawing instructions must be executed:

```
dcolor white;
drect rect_point1,rect_point2;
dexit; //Program must be correctly terminated

//Compiler macros for data:
color white,255,255,255;

vec2f rect_point1,50,50;
vec2f rect_point2,100,150;
```

These are all the basic drawing instruction that can be used:

Instruction	Description
DRECT	Draw a rectangle between two endpoints
DRECTWH	Draw a rectangle at some point (first operand), with some size (second operand)
DORECT	Similar to DRECT, but draws a rectangle outline
DORECTWH	Similar to DRECTWH, but draws a rectangle outline
DCIRCLE	Draw a circle at some point (first operand), with some radius (second operand)
DLINE	Draws a line between two points. Width specified with the DSETWIDTH instruction
DVXPOLY	Draw a custom polygon

It’s possible to specify quality at which the circle is drawn:

```
dcolor white;
mov #regCircleQuality,8; //8 vertices in the circle
dcircle pos,256; //Draw a circle in middle of the screen,
                 //and covering the entire screen
dexit;

//Compiler macros for data:
color white,255,255,255;
vec2f pos,256,256;
```

It’s also possible to draw 2D polygons (each polygon may have up to 128 vertices in it):

```
dcolor white;
dvxpoly polygon_data,4;
dexit;

//Compiler macros for data:
color white,255,255,255;

polygon_data: //Polygon for a distorted rectangle
  db 50,50;
  db 190,50;
  db 120,190;
  db 50,120;
```

It’s possible to use all of these instructions to draw textured data (see: ??)

Asynchonous thread runs in parallel to the main frame-based rendering thread, but it is not synchronized to frame boundaries (while the normal frame-based mode will restart execution each time it must render a new frame). It’s possible to use both at the same time, or use just one of two.

Asynchonous thread is not active by default, but it can be started up using the following code:

```
//Setup entrypoints
dentrypoint 0,DrawThread;
dentrypoint 4,AsyncThread;
```

```
//Set asynchronous thread frequency (speed)
mov #regAsyncFreq,200000;
//Run the thread
mov #regAsyncClk,1;
dexit;

DrawThread: //Do nothing
dexit;

AsyncThread:
  ...
jmp AsyncThread;
```

Asychronous thread frequency may be set up to 1,200,000. If asynchonous thread encounters an error, and there is no specified error handler, it will simply shut down (and reset `AsyncClk` register back to 0).

It’s possible to perform rendering in asynchonous thread in two ways. There are built-in opcodes which allow to draw to texture buffer, and then copy that image back into the front buffer. They require the hardware clear feature to be disabled though:

```
mov #regHWClear,0;
....

AsyncThread:
  dbegin; //Start drawing
    ... //Drawing code of any length
  dend; //Copy the image to front buffer

  dvsync; //If rendering is too fast, it can be synchronized with frame
          //generation, making it less resource intensive
jmp AsyncThread;
```

It’s also possible to manually switch buffers for drawing. see: ?? for more information on that.

If any error is encountered during the GPU execution, it will be handled in one of the possible ways:

- In frame-based mode with no entrypoint for the error handler set the GPU will display an error screen, detailing the error code and the error address.
- In frame-based mode with entrypoint set for the error handler the GPU will jump to the error handler. There must be no error occuring in the error handler itself, or the GPU will be stuck in an infinite loop until the frame ends.

- In asynchonous mode with no error handler an error will cause the thread to halt.
- In asynchonous mode with an error handler defined the thread will jump over to that error handler. Just as with the frame-based mode, and error inside the error handler will cause an infinite loop.

Entrypoint for error handler in the frame-based mode is `3`, and entrypoint for the asynchonous thread error handler is `5`. The error code will be passed in the `LINT` internal register, and the error parameter is passed in the `LADD` register. Here's an example of how to setup an error handler in both threads:

```
//Setup entrypoints
dentrypoint 0,DrawThread;
dentrypoint 3,DrawError;
dentrypoint 4,AsyncThread;
dentrypoint 5,AsyncError;

....

DrawError:
  cpuget R0,28; //Read error parameter
  cpuget R1,27; //Read error code
  ....
dexit;

AsyncError: //Similar to DrawError
  cpuget R0,28; //Read error parameter
  cpuget R1,27; //Read error code
  ....
Stop: dvsync; jmp Stop; //Stop with infinite loop
```

GPU provides several coordinate transformations. This allows programmer to control how the screen coordinates, which are generated by the drawing instructions, are mapped to screen coordiantes. The GPU native screen size is always 512x512 pixels (size of the rasterizer buffer/front buffer).

Coordinate transformation pipe (routine) can be selected using the `DCPIPE` opcode:

```
dcpipe 2; //Select transformation pipe 2
```

These coordinate transformation pipes are supported:

Index	Description
`0`	Coordinates are unchanged

1	Screen height/width specified by `Width` and `Height` registers
2	Coordinates must be in 0..1 range
3	Coordinates must be in -1..1 range
4	All drawing is offset so point (0,0) is the screen center

Before the coordinates are mapped to the screen ones the GPU also performs additional transformations based on the values of several register. This allows for scaling, rotating, offseting the result of drawing instructions:

```
dmove offset; //Move by vector 'offset'
drotatescale 1.23,2; //Scale up twice, rotate by 1.23 radians
drect ...; //Draw something

drotatescale 0,1; //Reset rotation, scale
dmove 0; //Reset offset
```

Rotation is clockwise, argument is specified in radians. It’s possible to scale on each axis separately, see the list of internal registers at page ??.

The GPU can also perform transformations on separate vertices which are being drawn via the drawing instructions. This is usually used to provide 3D graphics support.

no chapter

no chapter

no chapter

no chapter

no chapter paramlist

no chapter

no chapter

no chapter

no chapter

no chapter

no chapter DDFRAME, DDTERRAIN

The internal registers of the ZGPU are mapped to the memory, and are available as memory locations. They can be read and written to at any time, and they control various aspects of the ZGPU operation.

All of these registers are available in the HL-ZASM compiler by prepending `reg` prefix to the registers name.

Memory offsets `63488..64511` are mapped to the IOBus (external ports). The memory offsets `65536..131071` are mapped to the MemBus, allowing for access to external devices from the GPU. There is support for both reading and writing this memory, although at very low speed.

Name	Address	Description
`Clk`	`65535`	Current GPU power state (if set to 0, the GPU will be shut down)
`Reset`	`65534`	Reset the GPU state
`HWClear`	`65533`	Enables or disables the hardware clear (front buffer filling to black)
`VertexMode`	`65532`	Enables or disables the vertex mode (raw vertex output instead of rasterizing)
`Halt`	`65531`	Halts the current GPU execution, and preserves image in the front buffer
`RAMReset`	`65530`	Clears the GPU RAM
`AsyncReset`	`65529`	Reset the asynchonous thread state
`AsyncClk`	`65528`	Asynchronous thread execution state
`AsyncFreq`	`65527`	Asynchronous thread frequency
`Index`	`65526`	GPU index, can be between 0 and 31
`HScale`	`65525`	Horizontal image scale (for rasterized output)

`VScale`	65524	Vertical image scale (for rasterized output)
`HWScale`	65523	Hardware image scale
`Rotation`	65522	Rotation of the rasterized image. 0 for 0 deg, 1 for 90 deg, 2 for 180 deg, 3 for 270 deg
`TexSize`	65521	Subtexture size
`TexDataPtr`	65520	Pointer to texture data for load by the GPU
`TexDataSz`	65519	Size of the texture data for load by the GPU
`RasterQ`	65518	Rasterizer quality
`TexBuffer`	65517	Buffer used for the texturing (0: front buffer, 1: texture buffer)
`Width`	65515	Screen width (resolution)
`Height`	65514	Screen height (resolution)
`Ratio`	65513	Current screen ratio (physical)
`ParamList`	65512	Pointer to list of parameters for the `DWRITEFMT` instruction, or 0 if unused
`CursorX`	65505	X coordinate of the cursor (0..1)
`CursorY`	65504	Y coordinate of the cursor (0..1)
`Cursor`	65503	Should the cursor be drawn on screen
`CursorButtons`	65502	State of the cursor buttons
`BrightnessW`	65495	Total screen brightness
`BrightnessR`	65494	R component brightness
`BrightnessG`	65493	G component brightness
`BrightnessB`	65492	B component brightness
`ContrastW`	65491	Total screen contrast
`ContrastR`	65490	R component contrast
`ContrastG`	65489	G component contrast
`ContrastB`	65488	B component contrast
`CircleQuality`	65485	Circle output quality (number of vertices). Can be between 3 and 128

`OffsetX`	65484	X offset for screen coordinates of all drawn graphics
`OffsetY`	65483	Y offset for screen coordinates of all drawn graphics
`Rotation`	65482	Rotation in radians for screen coordinates of all drawn graphics
`Scale`	65481	Scale (1 is normal scale) for screen coordinates of all drawn graphics
`CenterX`	65480	X coordinate of centerpoint of rotation (see `Rotation` register)
`CenterY`	65479	Y coordinate of centerpoint of rotation (see `Rotation` register)
`CircleStart`	65478	Circle start angle (in radians)
`CircleEnd`	65477	Circle end angle (in radians)
`LineWidth`	65476	Line width
`ScaleX`	65475	X component of the scale for screen coordinates of all drawn graphics
`ScaleY`	65474	Y component of the scale for screen coordinates of all drawn graphics
`FontHalign`	65473	Font horizontal align mode
`ZOffset`	65472	Extra Z offset for all coordinates passed into vertex pipe
`FontValign`	65471	Font vertical align mode
`CullDistance`	65470	Culling distance
`CullMode`	65469	Face culling mode (0: front, 1: back)
`LightMode`	65468	Lighting mode (0: two-side, 1: front, -1: back)
`VertexArray`	65467	Pointer to array of vertices for indexed rendering
`TexRotation`	65466	Texture rotation in radians
`TexScale`	65465	Texture scale (1 is normal)
`TexCenterU`	65464	U component of centerpoint of texture rotation
`TexCenterV`	65463	V component of centerpoint of texture rotation

TexOffsetU	65462	U offset for the texture output
TexOffsetV	65461	V offset for the texture output

Mnemonic: `DTEST`
Encoding: `200`

This opcode generates a test pattern image on the GPU screen, somewhat similar to the PAL TV test pattern. It ignores any coordinate or texture transformations, and ignores all previous color commands/settings.

The rightmost left black bar will be left transparent.

Pseudocode:

```
W = ScreenWidth
H = ScreenHeight

for bar=0,6 do
  SetColor(TEST_PATTERN_COLOR[bar])
  Rectangle(W*0.125*bar,0,W*0.125,H*0.80)
end

for gray=0,7 do
  SetColor(31*gray,31*gray,31*gray,255)
  Rectangle(W*0.125*gray,H*0.80,W*0.125,H*0.20)
end
```

Mnemonic: `DEXIT/DVSYNC`
Encoding: `201`

Finishes drawing the current frame (used only in the frame-based mode). This must be the last instruction in any program that makes use of the frame-based mode.

The execution may be terminated before this instruction if amount of cycles spent drawing the current frame exceeds total limit

It’s implementation is exactly same as that of the `IDLE` instruction (while in the frame-based mode).

Pseudocode:

```
INTR = 1
```

Mnemonic: `DCLR`
Encoding: `202`

Clears screen/current buffer by filling it with black background.

Pseudocode:

```
SetColor(0,0,0,0)
Rectangle(0,0,ScreenWidth,ScreenHeight)
```

Mnemonic: `DCLRTEX`
Encoding: `203`

Clears screen/current buffer by filling it with a texture.

If vertex texturing is enabled, it will use the texture specified by the `DTEXTURE` opcode. Otherwise it will use the texture specified by the `DXTEXTURE` opcode.

If no texture was defined, it will fill buffer with solid black color.

Pseudocode:

```
BindState()
SetColor(0,0,0,255)
Rectangle(0,0,ScreenWidth,ScreenHeight)
```

Mnemonic: `DVXFLUSH`

Encoding: `204`

Draws all the pending polygons in the vertex buffer to screen, and clears the buffer.

This instruction is used with vertex buffer enabled. It will perform Z-sorting, clipping, etc, and draw the output to screen.

Pseudocode:

```
FlushBuffer()
```

Mnemonic: `DVXCLEAR`
Encoding: `205`

Clears any pending polygons from the vertex buffer.

Pseudocode:

```
ClearBuffer()
```

Mnemonic: `DSETBUF_VX`
Encoding: `206`

Sets the current drawing target to the raw vertex output. This opcode can only be used when vertex mode is active, and it is the default target for the vertex mode.

Pseudocode:

```
SetRendertarget(2)
```

Mnemonic: `DSETBUF_SPR`
Encoding: `207`

Sets current drawing target to the texture buffer. Also known as `DBACKBUF`.

Pseudocode:

```
SetRenderTarget(1)
```

Mnemonic: `DSETBUF_FBO`
Encoding: `208`

Sets current drawing target to the front/main buffer. Also known as `DFRONTBUF`.

Pseudocode:

```
SetRenderTarget(0)
```

Mnemonic: `DSWAP`
Encoding: `209`

Copies contents of the texture buffer into the front buffer.

Pseudocode:

```
Copy(RenderTarget(1),RenderTarget(0))
```

Mnemonic: `DVXPIPE X`
Encoding: `210 RM [Segment1] [Constant1]`

Selects the current vertex pipe/vertex transformation mode. This controls the transformation that brings world-space coordinates into screen coordinates. `X` can be one of the following pipes:

- `0`: X, Y coordinates are used as the screen coordinates
- `1`: Y, Z coordinates are used as the screen coordinates
- `2`: X, Z coordinates are used as the screen coordinates
- `3`: Uses basic 3D perspective projection (Z: depth)
- `4`: Transforms X, Y coordinates with the current model matrix
- `5`: Performs 3D transformation with projection and model matrices

Pseudocode:

```
VertexPipe = X
```

Mnemonic: `DCPIPE X`
Encoding: `211 RM [Segment1] [Constant1]`

Selects the current coordinate pipe/coordinate transformation mode. This controls the transformation that converts the screen coordinates into true coordinates, which are correctly mapped to the buffer. `X` can be one of the following pipes:

- `0`: No transformation
- `1`: Mapped to screen using the `Width` and `Height` registers.
- `2`: Coordinates are transformed from 0..1 range.
- `3`: Coordinates are transformed from -1..1 range.
- `4`: Coordinates are relative to the screen center (and not the top-left corner).

Pseudocode:

```
CoordinatePipe = X
```

Mnemonic: `DENABLE X`
Encoding: `212 RM [Segment1] [Constant1]`

Enables one of the internal GPU modes/switches. `X` can be one of the following:

- `0`: Vertex buffer
- `1`: Z-Sorting for the triangles in the vertex buffer.

- 2: Flat face lighting using the internal lighting system
- 3: Front/back face culling
- 4: Distance-based culling
- 5: Texturing using the internal GPU buffers

For example:

```
//Prepare 3D drawing
mov #regCullingDistance,4.0; //Setup culling distance
denable 0; //Vertex buffer
denable 1; //ZSorting
denable 2; //Lighting
denable 3; //Face culling
denable 4; //Distance-based culling
```

Pseudocode:

```
MODE_SWITCH[X] = 1
```

Mnemonic: `DDISABLE X`
Encoding: `213 RM [Segment1] [Constant1]`

Disables one of the internal GPU modes/switches. `X` can be one of the following:

- `0`: Vertex buffer
- 1: Z-Sorting for the triangles in the vertex buffer.
- 2: Flat face lighting using the internal lighting system
- 3: Front/back face culling
- 4: Distance-based culling
- 5: Texturing using the internal GPU buffers

For example:

```
//Finish 3D drawing
ddisable 0; //Vertex buffer
ddisable 1; //ZSorting
ddisable 2; //Lighting
ddisable 3; //Face culling
ddisable 4; //Distance-based culling
```

Pseudocode:

```
MODE_SWITCH[X] = 0
```

Mnemonic: `DCLRSCR X`
Encoding: `214 RM [Segment1] [Constant1]`

Clears screen with the specified color.

Pseudocode:

```
SetColor(X)
Rectangle(0,0,ScreenWidth,ScreenHeight)
```

Mnemonic: `DCOLOR X`
Encoding: `215 RM [Segment1] [Constant1]`

Sets the current drawing color.

If the vertex buffer is enabled, it will change the color of all the following polygons in the buffer (until the next `DCOLOR` command, or end of buffer).

Pseudocode:

```
SetColor(X)
```

Mnemonic: `DTEXTURE X`
Encoding: `216 RM [Segment1] [Constant1]`

Sets a texture out of one of the internal buffers. The buffer texture data is taken from is specified by the `TexBuffer` register.

By default it will take the entire buffer as the texture, but it is possible to specify smaller subtextures of the buffer. For example, it's possible to use four 256x256 textures, or 16 128x128 textures. If it is done so, the `X` parameter will specify which subtexture must be used.

Example:

```
mov #regTexBuffer,0; //Select front buffer
mov #regTexSize,128; //128x128 subtextures
dtexture 2; //Bind subtexture #2
```

Pseudocode:

```
SetBufferTexture(X)
```

Mnemonic: `DSETFONT X`
Encoding: `217 RM [Segment1] [Constant1]`

Set the current font for all operations which output text. There are 8 fonts available:

- `0`: Letter Gothic (Lucida Console)
- `1`: Courier New
- `2`: Trebuchet
- `3`: Arial
- `4`: Times New Roman
- `5`: Coolvetica
- `6`: Akbar
- `7`: CSD

Pseudocode:

```
Font = X
```

Mnemonic: `DSETSIZE X`
Encoding: `218 RM [Segment1] [Constant1]`

Set the current font size for all operations which output text. Size can be any integer value between 4 and 200.

Pseudocode:

Mnemonic: `DMOVE X`
Encoding: `219 RM [Segment1] [Constant1]`

Set the offset for 2D drawing. This instruction will offset all screen coordinates of next rendering instructions by the given vector.

`X` must be a pointer to vector, or it can be 0. If `X` is equal to zero then offset will be removed.

Pseudocode:

```
Registers[OffsetX] = X.x
Registers[OffsetY] = X.y
```

Mnemonic: `DVXDATA_2F X,Y`
Encoding: `220 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draw a single 2D polygon with up to 128 vertices. Also known as `DVXPOLY`.

If vertex array mode is not used, `X` points to an array of polygon vertex coordinates, and `Y` specifies the total count of vertices.

In vertex array mode, `X` points to an array of indexes into the vertex array, and `Y` specifies the total count of vertices.

Pseudocode:

```
VDATA = Registers[VertexArray]
for IDX=1,MIN(128,Y) do
  if VDATA > 0 then
    VIDX = ReadCell(X+IDX-1)
    VD[IDX] = {
      x = ReadCell(VDATA+VIDX*2+0),
      y = ReadCell(VDATA+VIDX*2+1)}
  else
```

```
    VD[IDX] = {
      x = ReadCell(X+(IDX-1)*2+0),
      y = ReadCell(X+(IDX-1)*2+1)}
  end
  ComputeTextureUV(VD[IDX],VD[IDX].x/512,VD[IDX].y/512)
end
DrawToBuffer(VD)
```

Mnemonic: `DVXDATA_2F_TEX X,Y`
Encoding: `221 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draw a single textured 2D polygon with up to 128 vertices. Also known as `DVXTEXPOLY`.

If vertex array mode is not used, `X` points to an array of polygon vertex coordinates and texture coordinates, and `Y` specifies the total count of vertices.

In vertex array mode, `X` points to an array of indexes into the vertex array, and `Y` specifies the total count of vertices.

Pseudocode:

```
VDATA = Registers[VertexArray]
for IDX=1,MIN(128,Y) do
  if VDATA > 0 then
    VIDX = ReadCell(X+IDX-1)
    VD[IDX] = {
      x = ReadCell(VDATA+VIDX*4+0),
      y = ReadCell(VDATA+VIDX*4+1)}
    ComputeTextureUV(VD[IDX],
      ReadCell(VDATA+VIDX*4+2),
      ReadCell(VDATA+VIDX*4+3))
  else
    VD[IDX] = {
      x = ReadCell(X+(IDX-1)*4+0),
      y = ReadCell(X+(IDX-1)*4+1)}
    ComputeTextureUV(VD[IDX],
      ReadCell(X+(IDX-1)*4+2),
      ReadCell(X+(IDX-1)*4+3))
  end
end
DrawToBuffer(VD)
```

Mnemonic: `DVXDATA_3F X,Y`
Encoding: `222 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draw a single 3D polygon with up to 128 triangles.

If vertex array mode is not used, `X` points to an array of triangle vertex coordinates, and `Y` specifies the total count of triangles.

In vertex array mode, `X` points to an array of indexes into the vertex array, and `Y` specifies the total count of triangles.

Pseudocode:

```
VDATA = Registers[VertexArray]
for IDX=1,MIN(128,Y) do
  if VDATA > 0 then
    VIDX1 = ReadCell(X+(IDX-1)*3+0)
    VIDX2 = ReadCell(X+(IDX-1)*3+1)
    VIDX3 = ReadCell(X+(IDX-1)*3+2)
    VD[1] = {
      x = ReadCell(VDATA+VIDX1*3+0),
      y = ReadCell(VDATA+VIDX1*3+1),
      z = ReadCell(VDATA+VIDX1*3+2)}
    VD[2] = {
      x = ReadCell(VDATA+VIDX2*3+0),
      y = ReadCell(VDATA+VIDX2*3+1),
      z = ReadCell(VDATA+VIDX2*3+2)}
    VD[3] = {
      x = ReadCell(VDATA+VIDX3*3+0),
      y = ReadCell(VDATA+VIDX3*3+1),
      z = ReadCell(VDATA+VIDX3*3+2)}
  else
    VD[1] = {
      x = ReadCell(X+(IDX-1)*9+0),
      y = ReadCell(X+(IDX-1)*9+1),
      z = ReadCell(X+(IDX-1)*9+2)}
    VD[2] = {
      x = ReadCell(X+(IDX-1)*9+3),
      y = ReadCell(X+(IDX-1)*9+4),
      z = ReadCell(X+(IDX-1)*9+5)}
    VD[3] = {
      x = ReadCell(X+(IDX-1)*9+6),
      y = ReadCell(X+(IDX-1)*9+7),
      z = ReadCell(X+(IDX-1)*9+8)}
  end

  ComputeTextureUV(VD[1],0,0)
  ComputeTextureUV(VD[2],1,0)
  ComputeTextureUV(VD[3],1,1)

  DrawToBuffer(VD)
end
```

Mnemonic: `DVXDATA_3F_TEX X,Y`
Encoding: `223 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draw a single textured 3D polygon with up to 128 triangles.

If vertex array mode is not used, `X` points to an array of triangle vertex coordinates and texture coordinates, and `Y` specifies the total count of triangles.

In vertex array mode, `X` points to an array of indexes into the vertex array, and `Y` specifies the total count of triangles.

Pseudocode:

```
VDATA = Registers[VertexArray]
for IDX=1,MIN(128,Y) do
    if VDATA > 0 then
      $L VIDX1 = ReadCell(X+(IDX-1)*3+0)
      $L VIDX2 = ReadCell(X+(IDX-1)*3+1)
      $L VIDX3 = ReadCell(X+(IDX-1)*3+2)
      VD[1] = {
        x = ReadCell(VDATA+VIDX1*5+0),
        y = ReadCell(VDATA+VIDX1*5+1),
        z = ReadCell(VDATA+VIDX1*5+2),
      }
      VD[2] = {
        x = ReadCell(VDATA+VIDX2*5+0),
        y = ReadCell(VDATA+VIDX2*5+1),
        z = ReadCell(VDATA+VIDX2*5+2),
      }
      VD[3] = {
        x = ReadCell(VDATA+VIDX3*5+0),
        y = ReadCell(VDATA+VIDX3*5+1),
        z = ReadCell(VDATA+VIDX3*5+2),
      }

      ComputeTextureUV(VD[1],
        ReadCell(VDATA+VIDX1*5+3),
        ReadCell(VDATA+VIDX1*5+4))
      ComputeTextureUV(VD[2],
        ReadCell(VDATA+VIDX2*5+3),
        ReadCell(VDATA+VIDX2*5+4))
      ComputeTextureUV(VD[3],
        ReadCell(VDATA+VIDX3*5+3),
        ReadCell(VDATA+VIDX3*5+4))
    else
      VD[1] = {
        x = ReadCell(X+(IDX-1)*15+0),
        y = ReadCell(X+(IDX-1)*15+1),
        z = ReadCell(X+(IDX-1)*15+2),
      }
      VD[2] = {
        x = ReadCell(X+(IDX-1)*15+5),
        y = ReadCell(X+(IDX-1)*15+6),
        z = ReadCell(X+(IDX-1)*15+7),
      }
      VD[3] = {
        x = ReadCell(X+(IDX-1)*15+10),
        y = ReadCell(X+(IDX-1)*15+11),
        z = ReadCell(X+(IDX-1)*15+12),
      }

      ComputeTextureUV(VD[1],
        ReadCell(X+(IDX-1)*15+ 3),
        ReadCell(X+(IDX-1)*15+ 4))
      ComputeTextureUV(VD[2],
        ReadCell(X+(IDX-1)*15+ 8),
        ReadCell(X+(IDX-1)*15+ 9))
      ComputeTextureUV(VD[3],
        ReadCell(X+(IDX-1)*15+13),
        ReadCell(X+(IDX-1)*15+14))
    end

    self:Dyn_EmitInterruptCheck()
    DrawToBuffer(VD)
  end
```

Mnemonic: `DVXDATA_3F_WF X,Y`
Encoding: `224 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draw a single wireframe 3D polygon with up to 128 triangles.

If vertex array mode is not used, `X` points to an array of triangle vertex coordinates and texture coordinates, and `Y` specifies the total count of triangles.

In vertex array mode, `X` points to an array of indexes into the vertex array, and `Y` specifies the total count of triangles.

Pseudocode:

```
VDATA = Registers[VertexArray]
for IDX=1,MIN(128,Y) do
  if VDATA > 0 then
    VIDX1 = ReadCell(X+(IDX-1)*3+0)
    VIDX2 = ReadCell(X+(IDX-1)*3+1)
    VIDX3 = ReadCell(X+(IDX-1)*3+2)
    VD[1] = {
      x = ReadCell(VDATA+VIDX1*3+0),
      y = ReadCell(VDATA+VIDX1*3+1),
      z = ReadCell(VDATA+VIDX1*3+2)}
    VD[2] = {
      x = ReadCell(VDATA+VIDX2*3+0),
      y = ReadCell(VDATA+VIDX2*3+1),
      z = ReadCell(VDATA+VIDX2*3+2)}
    VD[3] = {
      x = ReadCell(VDATA+VIDX3*3+0),
      y = ReadCell(VDATA+VIDX3*3+1),
      z = ReadCell(VDATA+VIDX3*3+2)}
  else
    VD[1] = {
      x = ReadCell(X+(IDX-1)*9+0),
      y = ReadCell(X+(IDX-1)*9+1),
      z = ReadCell(X+(IDX-1)*9+2)}
    VD[2] = {
      x = ReadCell(X+(IDX-1)*9+3),
      y = ReadCell(X+(IDX-1)*9+4),
      z = ReadCell(X+(IDX-1)*9+5)}
    VD[3] = {
      x = ReadCell(X+(IDX-1)*9+6),
      y = ReadCell(X+(IDX-1)*9+7),
      z = ReadCell(X+(IDX-1)*9+8)}
  end

  ComputeTextureUV(VD[1],0,0)
  ComputeTextureUV(VD[2],1,0)
  ComputeTextureUV(VD[3],1,1)

  DrawToBuffer(VD,WIREFRAME)
end
```

Mnemonic: `DRECT X,Y`
Encoding: `225 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draws a single rectangle. `X` is a pointer to vector which specifies the top-left vertex, and `Y` is a pointer to vector which specifies the bottom-right vertex.

Pseudocode:

```
VD[1] = {
  x = ReadCell(X+0),
  y = ReadCell(X+1)}
VD[2] = {
  x = ReadCell(Y+0),
  y = ReadCell(X+1)}
VD[3] = {
  x = ReadCell(Y+0),
  y = ReadCell(Y+1)}
VD[4] = {
  x = ReadCell(X+0),
  y = ReadCell(Y+1)}

ComputeTextureUV(VD[1],0,0)
ComputeTextureUV(VD[2],1,0)
ComputeTextureUV(VD[3],1,1)
ComputeTextureUV(VD[4],0,1)

DrawToBuffer(VD)
```

Mnemonic: `DCIRCLE X,Y`
Encoding: `226 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draws a circle or a sector with specific radius and angles.

Pseudocode:

```
R = Y
SIDES = clamp(ReadCell(65485),3,64)
START = ReadCell(65478)
END = ReadCell(65477)
STEP = (END-START)/SIDES
VEC = ReadVector2f(X)

for IDX=1,SIDES do
  VD[1] = {
    x = VEC.x + R*sin(START+STEP*(IDX+0)),
    y = VEC.y + R*cos(START+STEP*(IDX+0))}
  VD[2] = {
    x = VEC.x,
    y = VEC.y}
  VD[3] = {
    x = VEC.x + R*sin(START+STEP*(IDX+1)),
    y = VEC.y + R*cos(START+STEP*(IDX+1))}

  ComputeTextureUV(VD[1],0,0)
  ComputeTextureUV(VD[2],1,0)
  ComputeTextureUV(VD[3],1,1)

  DrawToBuffer(VD)
end
```

Mnemonic: `DLINE X,Y`
Encoding: `227 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draws a line between two points specified by the vectors `X` and `Y`.

Pseudocode:

```
  DrawLine(ReadVector2f($1),ReadVector2f($2))
```

Mnemonic: `DRECTWH X,Y`
Encoding: `228 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draws a single rectangle. `X` is a pointer to vector which specifies the top-left corner coordinates, and `Y` is a pointer to vector which specifies the Rectangle size.

Pseudocode:

```
  VD[1] = {
    x = ReadCell(X+0),
    y = ReadCell(X+1)}
  VD[2] = {
    x = ReadCell(X+0)+ReadCell(Y+0),
    y = ReadCell(X+1)}
  VD[3] = {
    x = ReadCell(X+0)+ReadCell(Y+0),
    y = ReadCell(X+1)+ReadCell(Y+1)}
  VD[4] = {
    x = ReadCell(X+0),
    y = ReadCell(X+1)+ReadCell(Y+1)}

  ComputeTextureUV(VD[1],0,0)
  ComputeTextureUV(VD[2],1,0)
  ComputeTextureUV(VD[3],1,1)
  ComputeTextureUV(VD[4],0,1)

  DrawToBuffer(VD)
```

Mnemonic: `DORECT X,Y`
Encoding: `229 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draws an outline of a rectangle. `X` is a pointer to vector which specifies the top-left vertex, and `Y` is a pointer to vector which specifies the bottom-right vertex.

The line width can be specified with the `DSETWIDTH` instruction.

Pseudocode:

```
  VD[1] = {
    x = ReadCell(X+0),
    y = ReadCell(X+1)}
  VD[2] = {
    x = ReadCell(Y+0),
    y = ReadCell(X+1)}
  VD[3] = {
    x = ReadCell(Y+0),
    y = ReadCell(Y+1)}
  VD[4] = {
    x = ReadCell(X+0),
    y = ReadCell(Y+1)}

  DrawLine(VD[1],VD[2])
  DrawLine(VD[2],VD[3])
  DrawLine(VD[3],VD[4])
  DrawLine(VD[4],VD[1])
```

Mnemonic: `DTRANSFORM2F X,Y`
Encoding: `230 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Transforms a 2D vector using the projection and the modelview matrices.

Pseudocode:

Pseudocode:

Mnemonic: `DTRANSFORM3F X,Y`
Encoding: `231 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Transforms a 3D vector using the projection and the modelview matrices.

Pseudocode:

Mnemonic: `DSCRSIZE X,Y`
Encoding: `232 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Sets the current screen size.

Pseudocode:

```
Registers[Width] = X
Registers[Height] = Y
```

Mnemonic: `DROTATESCALE X,Y`
Encoding: `233 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Rotates and scales coordinates of all following graphics instructions. The default centerpoint of rotation is (0,0) which can be changed using the `CenterX` and the `CenterY` registers.

Pseudocode:

```
Registers[Rotation] = X
Registers[Scale] = Y
```

Mnemonic: `DORECTWH X,Y`
Encoding: `234 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Draws an outline of a rectangle. `X` is a pointer to vector which specifies the top-left corner coordinates, and `Y` is a pointer to vector which specifies the Rectangle size.

The line width can be specified with the `DSETWIDTH` instruction.

Pseudocode:

```
  VD[1] = {
    x = ReadCell(X+0),
    y = ReadCell(X+1)}
  VD[2] = {
    x = ReadCell(X+0)+ReadCell(Y+0),
    y = ReadCell(X+1)}
  VD[3] = {
    x = ReadCell(X+0)+ReadCell(Y+0),
    y = ReadCell(X+1)+ReadCell(Y+1)}
  VD[4] = {
    x = ReadCell(X+0),
    y = ReadCell(X+1)+ReadCell(Y+1)}

  DrawLine(VD[1],VD[2])
  DrawLine(VD[2],VD[3])
  DrawLine(VD[3],VD[4])
  DrawLine(VD[4],VD[1])
```

Mnemonic: `DCULLMODE X,Y`
Encoding: `235 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Sets the current culling mode and lighting mode.

`X` sets the culling mode:

- **`0`: front face culling**
- **`1`: back face culling**

`Y` sets the lighting mode:

- **`0`: double-side lighting**
- **`1`: front side lighting**
- **`-1`: back side lighting**

Pseudocode:

```
Register[CullMode] = X
Register[LightMode] = X
```

Mnemonic: `DPIXEL X,Y`
Encoding: `238 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Outputs a single pixel to screen. `X` is a pointer to vector which specifies coordinates on screen (can be non-integer, which will cause anti-aliasing effect), and `Y` is a pointer to color of the pixel.

Pseudocode:

```
SetPixel(X,Y)
```

Mnemonic: `DWRITE X,Y`

Encoding: `240 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Writes a null-terminated string to screen. `X` is a pointer to vector that specifies the position of string on screen, and `Y` is the pointer to the first character of the string.

Pseudocode:

```
TEXT = VM:ReadString(Y)
FontWrite(X,TEXT)
```

Mnemonic: `DWRITEI X,Y`
Encoding: `241 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Writes a integer value to screen. `X` is a pointer to vector that specifies the position of string on screen, and `Y` is the value that must be drawn on screen.

Pseudocode:

```
FontWrite(X,Integer(Y))
```

Mnemonic: `DWRITEF X,Y`
Encoding: `242 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Writes a floating-point value to screen. `X` is a pointer to vector that specifies the position of string on screen, and `Y` is the value that must be drawn on screen.

Pseudocode:

```
FontWrite(X,Y)
```

Mnemonic: `DENTRYPOINT X,Y`
Encoding: `243 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Sets one of the GPU entrypoints. Each entrypoint corresponds to a specific function, there are the following entrypoints available right now:

Pseudocode:

Mnemonic: `DSETLIGHT X,Y`
Encoding: `244 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Sets parameters of one of the 8 lights supported by the GPU. `X` **is the light index (0..7), and** `Y` **points to the following data structure:**

```
LightData:
  vector4f position,<x>,<y>,<z>,0;
  vector4f color,<r>,<g>,<b>,<brightness>;
```

Light brightness is usually set to 1, but can vary.

Pseudocode:

```
if (X < 0) or (X > 7) then
  Interrupt(19,0)
else
  Lights[X] = {
    Position = ReadVector4f(Y+0),
    Color    = ReadVector4f(Y+4)}
end
```

Mnemonic: `DGETLIGHT X,Y`
Encoding: `245 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Reads light data for one of the 8 lights supported by the GPU. `X` **is the light index (0..7), and** `Y` **points to the following data structure, which will be filled with light data:**

```
LightData:
  vector4f position,<x>,<y>,<z>,0;
  vector4f color,<r>,<g>,<b>,<brightness>;
```

Pseudocode:

```
N/A
```

Mnemonic: `DWRITEFMT X,Y`
Encoding: `246 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Writes a formatted string to screen. `X` points to vector that specifies the position of string on screen, and `Y` is the string that must be drawn on screen.

Variables used in the string format must follow the string data. If `ParamList` register is set, then variables used in the string format start at that offset.

Pseudocode:

```
N/A
```

Mnemonic: `DWRITEFIX X,Y`
Encoding: `247 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Writes a fixed-point value to screen. `X` is a pointer to vector that specifies the position of string on screen, and `Y` is the value that must be drawn on screen.

Pseudocode:

```
N/A
```

Mnemonic: `DTEXTWIDTH X,Y`
Encoding: `248 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Returns the width of string using the current font, and writes it to `X`.

Pseudocode:

Mnemonic: `DTEXTHEIGHT X,Y`
Encoding: `249 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Returns the height of string using the current font, and writes it to `X`.

Pseudocode:

Mnemonic: `MLOADPROJ X`
Encoding: `271 RM [Segment1] [Constant1]`

Loads given matrix into the GPU projection matrix. `X` points to the matrix.

Pseudocode:

```
ProjectionMatrix = ReadMatrix(X)
```

Mnemonic: `MREAD X`
Encoding: `272 RM [Segment1] [Constant1]`

Reads GPU model matrix. `X` points to the matrix into which model matrix will be written.

Pseudocode:

```
WriteMatrix(X,ModelMatrix)
```

Mnemonic: `DT X`
Encoding: `274 RM [Segment1] [Constant1]`

Returns time passed since last frame (works only in frame-based mode).

Pseudocode:

```
X = TimerDT
```

Mnemonic: `DSHADE X`
Encoding: `276 RM [Segment1] [Constant1]`

Shades the current color by specific amount. `X` is the shading value. A value between 0 and 1 will make the color darker, a value of 1 will not change the current color, while value higher than 1 will make color brighter.

There is no normalization, so values outside of 0..1 range might generate weird colors.

Pseudocode:

```
Color.x = Color.x*X
Color.y = Color.y*X
Color.z = Color.z*X
SetColor(Color)
```

Mnemonic: `DSETWIDTH X`
Encoding: `277 RM [Segment1] [Constant1]`

Sets line width.

Pseudocode:

```
Register[LineWidth] = X
```

Mnemonic: `MLOAD X`
Encoding: `278 RM [Segment1] [Constant1]`

Loads given matrix into the GPU model matrix. `X` points to the matrix.

Pseudocode:

```
ModelMatrix = ReadMatrix(X)
```

Mnemonic: `DSHADENORM X`
Encoding: `279 RM [Segment1] [Constant1]`

Shades the current color by specific amount. `X` is the shading value. A value between 0 and 1 will make the color darker, a value of 1 will not change the current color, while value higher than 1 will make color brighter.

The resulting color is normalized, so it's possible to use values outside of the 0..1 range.

Pseudocode:

```
Color.x = Clamp(Color.x*X,0,255)
Color.y = Clamp(Color.y*X,0,255)
Color.z = Clamp(Color.z*X,0,255)
SetColor(Color)
```

Mnemonic: `DDFRAME X`
Encoding: `280 RM [Segment1] [Constant1]`

Draws a framed rectangle. `X` points to the following data structure:

```
FrameData:
  vector2f position,<x>,<y>;
  vector2f size,<w>,<h>;
  vector4f info,<shadow>,<highlight>,<face>,<border size>;
```

The `info` entry stores pointers to colors that must be used in rendering.

Pseudocode:

```
V1 = ReadVector2f(X+0)
V2 = ReadVector2f(X+2)
V3 = ReadVector4f(X+4)

CSHADOW    = ReadVector3f(V3.x)
CHIGHLIGHT = ReadVector3f(V3.y)
CFACE      = ReadVector3f(V3.z)

VD1[1] = {
  x = V3.w + V1.x,
  y = V3.w + V1.y}
VD1[2] = {
  x = V3.w + V1.x + V2.x,
  y = V3.w + V1.y}
VD1[3] = {
  x = V3.w + V1.x + V2.x,
  y = V3.w + V1.y + V2.y}
VD1[4] = {
  x = V3.w + V1.x,
  y = V3.w + V1.y + V2.y}

VD2[1] = {
  x = -V3.w + V1.x,
  y = --V3.w + V1.y}
VD2[2] = {
  x = -V3.w + V1.x + V2.x,
  y = -V3.w + V1.y}
VD2[3] = {
  x = -V3.w + V1.x + V2.x,
  y = -V3.w + V1.y + V2.y}
VD2[4] = {
  x = -V3.w + V1.x,
  y = -V3.w + V1.y + V2.y}

VD3[1] = {
  x = V1.x,
  y = V1.y}
VD3[2] = {
  x = V1.x + V2.x,
  y = V1.y}
VD3[3] = {
  x = V1.x + V2.x,
  y = V1.y + V2.y}
VD3[4] = {
  x = V1.x,
  y = V1.y + V2.y}

ComputeTextureUV(VD1[1],0,0)
ComputeTextureUV(VD1[2],1,0)
ComputeTextureUV(VD1[3],1,1)
ComputeTextureUV(VD1[4],0,1)

ComputeTextureUV(VD2[1],0,0)
ComputeTextureUV(VD2[2],1,0)
ComputeTextureUV(VD2[3],1,1)
ComputeTextureUV(VD2[4],0,1)

ComputeTextureUV(VD3[1],0,0)
ComputeTextureUV(VD3[2],1,0)
ComputeTextureUV(VD3[3],1,1)
ComputeTextureUV(VD3[4],0,1)

SetColor(CSHADOW)
DrawToBuffer(VD1)
SetColor(CHIGHLIGHT)
DrawToBuffer(VD2)
SetColor(CFACE)
DrawToBuffer(VD3)
```

Mnemonic: `DRASTER X`
Encoding: `283 RM [Segment1] [Constant1]`

Set raster quality.

Pseudocode:

```
Registers[RasterQ] = X
```

Mnemonic: DDTERRAIN X
Encoding: 284 RM `[Segment1] [Constant1]`

Draw 3D terrain.

Pseudocode:

```
W = ReadCell(X+0)
H = ReadCell(X+1)
R = clamp(floor(ReadCell(X+2)),0,16)
U = ReadCell(X+3)
V = ReadCell(X+4)

MinX = clamp(floor(W/2 + U - R),1,W-1)
MinY = clamp(floor(H/2 + V - R),1,H-1)
MaxX = clamp(floor(W/2 + U + R),1,W-1)
MaxY = clamp(floor(H/2 + V + R),1,H-1)

for X=MinX,MaxX do
  for Y=MinY,MaxY do
    XPOS = X - W/2 - U - 0.5
    YPOS = Y - H/2 - U - 0.5

    if (X > 0) and (X <= W-1) and (Y > 0) and (Y <= H-1) and (XPOS^2+YPOS^2 <= R^2) then
      Z1 = ReadCell(X+16+(Y-1)*W+(X-1)
      Z2 = ReadCell(X+16+(Y-1)*W+(X-0)
      Z3 = ReadCell(X+16+(Y-0)*W+(X-0)
      Z4 = ReadCell(X+16+(Y-0)*W+(X-1)

      VD[1] = { x = XPOS,   y = YPOS,   y = Z1 }
      VD[2] = { x = XPOS+1, y = YPOS,   y = Z2 }
      VD[3] = { x = XPOS+1, y = YPOS+1, y = Z3}

      ComputeTextureUV(VD[1],0,0)
      ComputeTextureUV(VD[2],1,0)
      ComputeTextureUV(VD[3],1,1)
      DrawToBuffer(VD)

      VD[1] = { x = XPOS,   y = YPOS,   y = Z1}
      VD[2] = { x = XPOS,   y = YPOS+1, y = Z4}
      VD[3] = { x = XPOS+1, y = YPOS+1, y = Z3}

      ComputeTextureUV(VD[1],0,0)
      ComputeTextureUV(VD[2],0,1)
      ComputeTextureUV(VD[3],1,1)
      DrawToBuffer(VD)
    end
  end
end
```

Mnemonic: `DMULDT X,Y`
Encoding: `294 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Multiplies `Y` by time-step and writes it into `X`. Used in frame-based mode to provide smooth animations.

Pseudocode:

```
X = Y * TimerDT
```

Mnemonic: `DBEGIN`
Encoding: `298`

Starts asynchonous drawing. Used only in asynchronous thread.

Pseudocode:

```
SetRenderTarget(1)
```

Mnemonic: `DEND`
Encoding: `299`

Ends asynchonous drawing, and outputs the drawn image to screen.

Pseudocode:

```
FlushBuffer()
Copy(1,0)
SetRenderTarget(2)
```

Mnemonic: `DXTEXTURE X`
Encoding: `303 RM [Segment1] [Constant1]`

Binds a predefined texture. `X` points to the string that contains texture name. If `X` is equal to 0, texture will be unbound.

Pseudocode:

```
if X > 0 then
  NAME = VM:ReadString(X)
  SetTexture(NAME)
else
  SetTexture(0)
end
```

ZSPU is a programmable sequencer and sound synthesizer. It works with waveforms and channels. Each waveform is an unique sound, there can be up to 128 different waveforms loaded, while each channel corresponds to one of the output channels. It’s possible to play just one waveform through each channel, although it’s possible to use same waveform for many channels.

Each channel has separate pitch, volume settings, and supports hardware ADSR/LFO.

no chapter

no chapter

Mnemonic: `CHRESET X`
Encoding: `320 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CHSTART X`
Encoding: `321 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `CHSTOP X`
Encoding: `322 RM [Segment1] [Constant1]`

Pseudocode:

Mnemonic: `WSET X,Y`
Encoding: `330 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHWAVE X,Y`
Encoding: `331 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHLOOP X,Y`
Encoding: `332 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHVOLUME X,Y`
Encoding: `333 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHPITCH X,Y`
Encoding: `334 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHMODT X,Y`
Encoding: `335 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHMODA X,Y`
Encoding: `336 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHMODF X,Y`
Encoding: `337 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `CHADSR X,Y`
Encoding: `338 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

Mnemonic: `WLEN X,Y`
Encoding: `339 RM [Segment1] [Segment2] [Constant1] [Constant2]`

Pseudocode:

This document was translated from LaTeX by HeVeA.