Документация Zyelios CPU 10

Black Phoenix

4 апреля 2011 г.

Оглавление

Глава 1

Функции процессора

1.1 Вступление

Zyelios CPU - виртуальный процессор. Он был создан для для автоматизации устройств и машин, созданных с помощью Wiremod, а также для обучения основам низкоуровневого программирования, программирования на языке C (с выходом компилятора HL-ZASM).

Этот процессор по своей архитектуре и внутренней структуре очень сильно напоминает современные микроконтроллеры и процессоры общего назначения, которые используются в современных компьютерах. Он был первоначально основан на базе архитектуры `x86`, но имеет значительные отличия, например, ZCPU использует числа с плавающей точкой вместо целочисленной арифметики.

Процессор - сложная машина, и, подобно управлению настоящей машиной, требует навыков. Надо понимать, что, несмотря на большие возможности по использованию процессора, он работает по весьма простым логическим правилам. Все функции процессора созданы взаимодействием этих правил.

Данная книга служит краткой документацией по всем функциям и возможностям ZCPU, включая любые не очевидные особенности работы. Но здесь указаны лишь возможности по управлению процессором, эта книга не предназначена для вводного ознакомления с ним.

1.2 Теория

Процессоры были изобретены как следующий шаг развития аналоговых компьютеров. Первые процессоры были использованы для решения математических уравнений, расчёта физических моделей и обработки большого количества статистических данных.

Программа, которую выполняет процессор, разделена на инструкции. Каждая инструкция это простая операция, которая некоторым образом меняет состояние процессора, или любого из внешних устройств. Например:

```
MOV R0,100 //Занести 100 в регистр R0
ADD R0,R1 //Добавить R1 к R0
```

Дополнительные параметры, которые передаются в инструкцию и некоторым образом задают её поведение, называются *операндами*. У каждой инструкции может быть 0, 1 или 2 операнда. Первый операнд называется *целевым* операндом, а второй - *исходным*.

В результате выполнения инструкции результат может быть записан в первый операнд:

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

Регистром называется ячейка со значением, которая находится внутри процессора. Регистры могут быть регистрами *общего назначения* и *управляющими* (внутренними). Программист может любым образом использовать регистры общего назначения, например, для сохранения промежуточных результатов:

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

Управляющие регистры меняют состояние и режим работы процессора.

Внешние устройства и дополнительная память подключаются к процессору через шину данных/адресов. *Шина данных/адресов* - это особое устройство, которое позволяет связывать несколько других устройств вместе и производить обмен данными между ними. В процессоре ZCPU есть две шины: `MemBus` (шина внешней памяти), и `IOBus` (шина ввода/вывода по портам). Каждая ячейка памяти на этих шинах адресуется целым числом - *адресом*. Нету никакой принципиальной разницы между двумя вышеуказанными шинами процессора, но ячейки шины портов можно использовать как регистры процессора:

```
MOV R0,PORT0 //Считать ячейку 0 через шину портов
MOV [500],R0 //Записать в ячейку 500 через шину памяти
```

Шина памяти расположена на положительных адресах, а шина портов расположена в отрицательных адресах процессора, и как-бы зеркально отражена относительно шины памяти. Например:

```
MOV R0,PORT0 //Считать ячейку 0 через шину портов
MOV R0,[-1] //Тоже самое
```

В процессоре ZCPU есть поддержка нескольких моделей памяти, которые задают размер встроенной оперативной памяти, размер постоянной памяти, и тому подобное.

1.3 Регистры общего назначения

В процессоре есть 40 регистров общего назначения. Они разделяются на *основные* и *дополнительные*. Регистры основного набора - `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP` и `EBP`. Регистры дополнительного набора нумеруются таким образом: `R0`, `R1`, ..., `R31`.

Каждый регистр может хранить некоторое число, и его можно использовать для проведения арифметических операций. Но при этом регистр `ESP` используется для работы стека процессора, и изменение его значения может привести к некорректной работе программы. Если используются возможности C, то `EBP` также используется для сохранения указателя на стековый кадр функции, и его изменение вызовет ошибку программы.

Каждый регистр хранит одно 64-битное значение с плавающей точкой. Его значение может находиться в пределах от -10^{1022} до 10^{1023}, но при этом точность самих чисел ограничена 48 битами. Это упрощение отличает процессор ZCPU от реальных процессоров.

Использование регистров *намного* быстрее использования памяти. В 10 версии процессора были добавлены 32 регистра общего назначения, сформировав дополнительный набор регистров для ускорения работы программ.

При сбросе состояния процессора все регистры, кроме `ESP`, будут выставлены в 0. Если в текущей модели памяти процессора есть внутренняя встроенная память, то регистр указателя стека будет указывать на последнюю ячейку внутренней памяти процессора.

1.4 Выполнение инструкций

В процессоре ZCPU есть специальный регистр который называется указателем на инструкцию (`IP`). Регистр `IP` указывает на текущую выполняемую процессором инструкцию, и увеличивается по мере выполнения последующих инструкций.

Инструкции процессора могут быть либо переменной длины, либо постоянной длины, в зависимости от режима исполнения (см. стр 63), поэтому указатель на инструкцию `IP` может меняться с разным шагом. Можно напрямую задавать значение регистра `IP` используя операции условного и безусловного перехода (см. стр 19).

Выполнение инструкций происходит следующим образом:

1. Процессор считывает номер инструкции
2. Если у инструкции есть операнды, то процессор считывает байт селектора регистр-память (см. `rmbyte`). Если у инструкции нету операндов, и следующий байт нулевой, то он будет пропущен (это выполняется для совместимости со старыми версиями процессора).
3. Номер инструкции декодируется, и считываются дополнительные байты сегментных префиксов (если они нужны, см. стр 57 для более детальной информации о их работе).
4. Если нужно, то будут считаны дополнительные байты констант нужных для операнда.
5. Процессор выполняет микрокод инструкции, который некоторым образом меняет состояние процессора (но при этом эти изменения кешируются, и не сразу записываются в регистры).
6. Внутреннее состояние обновляется, и все записанные в кеш регистры переносяться в их реальные ячейки.

Кеширование используется для оптимизации скорости работы процессора. На странице 54 указан более детальный механизм кеширования.

Процессор выполняет постоянное количество циклов за секунду. Количество циклов необходимое для выполнения каждой отдельной инструкции может меняться. Полное количество циклов считается регистром `TMR`. Почти все инструкции всего 1 цикл в длину, но из-за кеширования и доступа к памяти это число может меняться в значительных пределах.

См. стр 35 для более детальной информации о процессе выполнения.

1.5 Прерывания

При возникновении некоторых особых событий в процессоре (например внешний сигнал, возникновение ошибки в арифметических операциях, и т.п.) процессор ZCPU сгенерирует прерывание. Прерывания - это особая функция процессора, которая временно "прерывает"его работу, изменяя выполнение программы что-бы выполнить обработчик события. При возникновении прерывания процессор заносит на свой стек адрес возврата в программу, и переходит к выполнению обработчика прерывания (см. стр 18 для более подробной информации о стеке процессора).

В расширеном режиме процессор использует таблицу прерываний, в которой хранятся указатели на все функции-обработчики событий. Если расширенный режим не включён, то любое прерывание будет полностью останавливать процессор.

Каждое прерывание задаётся его номером, а также параметром прерывания - некоторым числом, которое несёт дополнительную информацию о прерывании.

После окончания выполнения обработчика прерывания состояние процессора будет восстановлено, и программа продолжит выполнение с места где возникло прерывание.

При возникновении прерывания процессор заносит на стек адрес возврата. Например если текущее значение указателя на инструкцию `IP` есть 157, а состояние стека:

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

ESP = 65304
```

то после возникновения прерывания стек примет вид:

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

ESP = 65302
```

и указатель на инструкцию будет равен точке входа в обработчик прерываний.

Прерывания очень похожи на обычные вызовы подпрограмм (см. стр 19), но вместо использования прямого указателя процессору передаётся номер прерывания. Для возврата из обработчика нужно использовать специальную команду возврата `IRET` (вместо обычной `RET`). Любое прерывание можно вызвать вручную используя инструкцию `INT`. Пример обработчика прерываний:

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

Существует два вида прерываний: внутренние прерывания (ошибки), и внешние прерывания. Внешние прерывания сохраняют состояние всех основных регистров

процессора (вместе с данными для возврата из прерывания) перед выполнением обработчика. Они могут быть вызваны только через внешний вход для прерываний, или используя инструкцию `EXTINT`. Внешние прерывания требуют использования инструкции `EXTRET` для выхода из обработчика вместо `IRET` как для обычных прерываний.

Есть несколько стандартных прерываний. Они будут вызваны если происходит некоторое внутреннее исключение. Все прерывания с номером ниже 32 зарезервированы для обработки ошибок процессора. Также прерывания 0 и 1 имеют специальное значение, и используются для перезапуска/выключения процессора. Прерывания 32..255 доступны для использования пользовательской программы. См. стр 22 для более детальной информации о кодах ошибок (номерах прерываний ошибок).

Эта таблица содержит все коды ошибок процессора:

Номер	Описание
0	*Сброс*
1	*Отключение*
2	Конец выполнения программы
3	Деление на ноль
4	Неизвестная инструкция
5	Внутренняя ошибка процессора
6	Ошибка стека (переполнение/недополнение)
7	Ошибка записи/чтения памяти
8	Ошибка шины памяти
9	Ошибка доступа записи (защита страницы памяти)
10	Ошибка записи/чтения порта
11	Ошибка доступа к странице (защита страницы памяти)
12	Ошибка доступа чтения (защита страницы памяти)
13	Общая ошибка процессора
14	Ошибка исполнения (защита страницы памяти)
15	Выход за пределы адресного пространства
17	Ограничение количества инструкций (*только GPU*)
23	Ошибка чтения строки (*только GPU*)
28	*Обработчик доступа чтения страницы*
29	*Обработчик доступа записи страницы*
30	*Обработчик доступа к странице*
31	*Обработчик шага отладки*

Прерывание сброса (0) сбросит состояние процессора, и перезапустит выполнение

кода с начала. Прерывание отключения (1) остановит выполнение кода пока процессор не будет перезапущен извне. Инструкции `int 1` и `int 0` будут правильно работать если они не обрабатываются процессором (т.е. если они отключены в таблице прерываний, или если таблицы прерываний отключена):

```
INT 1; //Остановить выполнение

INT 0; //Перезапустить процессора
```

Если расширенный режим не включен, *все прерывания* работают так-же как и прерывание остановки (`int 1`). Код ошибки будет послан на внешний выход кода ошибки.

Обработка прерываний может быть отключена флагом прерываний `IF`. Этот регистр напрямую изменить нельзя, но его можно менять специальными инструкциями. *При выставлении флага он будет выставлен только после следующей инструкции.* Например:

```
STI;         //Установить флаг прерываний
MOV EAX,123; //IF = 0
ADD EAX,123; //IF = 1 (флаг обновлён)
CLI;         //Убрать флаг прерываний
SUB EAX,123; //IF = 0 (флаг убран)
RET;         //IF = 0
```

Это даёт возможность предотвратить ситуацию когда прерывание возникнет между инструкцией возврата и `STI`:

```
ContextSwitch:
  CLI;

  ... код ...

  STI;  //Прерывание никогда не возникнет между STI
EXTRET; //и NMIRET
```

Для более детальной информации по работе таблицы прерываний см. страницу 47.

1.6 Сегментация памяти и сегментные регистры

В ZCPU есть 8 сегментных регистров. Это `CS`, `SS`, `DS`, `ES`, `GS`, `FS`, `KS` и `LS`. Они используются процессором для работы с разными моделями памяти. Также любые из 40 регистров общего назначения могут быть использованы как сегментные регистры.

Каждый сегментный регистр может хранить 48-битное целое число. Это значение может быть использовано для задание смещения для указателей при чтении/записи в память. Процессор всегда использует сегменты при обращении к памяти: он транслирует локальный адрес (который прямо задан пользователем) в абсолютный адрес (физический адрес ячейки в памяти).

Формула для трансляции адреса?

$$AbsoluteAddress = LocalAddress + SegmentOffset$$

Пользователь может задать регистр, который должен быть использован как сегментный, задавая его как префикс к инструкции. Если сегмент не указан, то по умолчанию используется сегмент `DS`.

Как префикс можно использовать регистры общего назначения и сегментные регистры. Невозможно использовать число как сегментный префикс. Вот несколько разных примеров синтаксиса для чтения из памяти:

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

MOV EAX,EBX:ECX   //Адрес: EBX+ECX
MOV EAX,[EBX+ECX] //Адрес: EBX+ECX
MOV EAX,[EBX:100] //Адрес: EBX+100
MOV EAX,[100:EBX] //Неправильный синтаксис
```

Можно использовать сегментные префиксы для быстрого доступа к массивам данных, если использовать префикс как указатель на массив:

```
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
```

Некоторые сегментные регистры используются процессором для особых целей, см. таблицу:

Регистр	Название	Описание
CS	Сегмент кода	Код запускается из этого сегмента
SS	Сегмент данных	Сегмент данных по умолчанию
DS	Сегмент стека	Сегмент в котором хранится стек
ES	Доп. сегмент	Пользовательский сегмент
GS	G-сегмент	Пользовательский сегмент
FS	F-сегмент	Пользовательский сегмент
KS	Ключевой сегмент	Пользовательский сегмент
LS	L-сегмент	Пользовательский сегмент

Все сегментные регистры кроме CS можно задавать напрямую инструкцией MOV. Единственный способ изменить регистр CS это выполнить инструкцию CALLF или JMPF (см. стр 19):

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

JMPF 500,100;

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

Если попытаться напрямую задать регистр CS то будет вызвана ошибка 13:1 (общая ошибка процессора). Например:

```
MOV DS,100
MOV ES,KS
MOV CS,1000 //Вызовет прерывание 13:1
```

После сброса процессора все сегментные регистры будут сброшены в 0.

1.7 Модели памяти

Процессор ZCPU поддерживает несколько разных моделей памяти. По умолчанию используется режим линейной адресации памяти, и в этом режиме находиться процессор при его старте.

Разные модели памяти требуют разного использования регистров, и дают разные возможности по выполнению кода. Большинство режимов требует включенного расширенного режима процессора для получения функций расширенной защиты памяти, задания прав доступа к страницам, отображения памяти.

Важно знать как именно происходит доступ к памяти. При обращении к памяти процессора происходят такие операции:

1. Проверка регистра `BusLock` (отключение шины)
2. Проверка адреса - адрес должен быть 48-битным целым числом.
3. Прочитать страницу, которая соответствует данному адресу.
4. Если включен флаг `EF` (расширенный режим), то проверить что-бы текущий уровень доступа был меньше или равен уровня доступа страницы, к которой идёт обращение, и проверить флаги прав доступа страницы.
5. Если у страницы выставлен флаг `Override` то войти в соответствующий режим перехвата обращения к памяти.
6. Если у страницы выставлен флаг `Remapped` то правильно обработать отображение страницы в памяти.
7. Выполнить операцию ввода/вывода.

1.7.1 Режим линейной адресации

Это модель памяти по умолчанию, когда все сегментные регистры равны `0`. Этот режим доступен сразу после запуска процессора. Поскольку все сегментные регистры равны нулю, то стек, данные, и код будут находиться в одном и том-же адресном пространстве.

В этом режиме не нужны сегментные префиксы. Например, можно напрямую обращаться к стеку:

```
MOV EAX,#0  //EAX будет равен 14 (номер инструкции MOV)
MOV #ESP,100 //Тоже PUSH 100
DEC ESP
POP EAX     //EAX будет равен 100
```

1.7.2 Режим сегментированой адресации

В этом режиме адресации сегментные регистры используются для указания отдельных блоков для стека, данных, и кода. У такого подхода есть некоторые преимущества:

- Позволяет предотвратить случайное повреждение данных/кода
- Один и тот-же код может использоваться для разных блоков данных
- Программы могут запускаться в локальном адресном пространстве

Example:

```
MOV DS,1000 //Установить первый блок данных
MOV SS,2000 //Установить первый блок стека
CALLF 0,500 //Вызвать подпрограмму (CS установлен в 500, IP в 0)

MOV DS,3000 //Установить второй блок данных
MOV SS,4000 //Установить второй блок стека
CALLF 0,500 //Вызвать ту-же подпрограмму, но с другим блоком данных
```

Это запустит туже подпрограмму (по физическому адресу `500`) используя два разных набора данных, то есть одна и та же подпрограмма может быть использована дважды для разных наборов переменных.

1.7.3 Режим отображённой памяти

Режим отображённой памяти использует возможности отображения памяти процессором для переназначения физических адресов которым соответствуют страницы. Это даёт возможность создать видимость одного целостного адресного пространства у каждого приложения. Это даёт возможность приложениям выделять блоки памяти, которые будут дополнены к адресному пространству.

См. стр 40 для более детальной информации о возможностях процессора по отображению памяти.

1.8 Стек

В процессор Zyelios CPU встроен аппаратный стек для значений, который физически находится в памяти процессора. Состояние стека описывается состоянием стекового указателя `ESP`, который указывает на следующую свободную ячейку памяти в стеке, регистра указывающего размер стеке (`ESZ`) и стекового сегментного регистра (`SS`).

ESP указывает на *следующую свободную ячейку стека*. Значения на стеке располагаются в последовательности уменьшения их адресов в памяти.

Для занесения и получения значений со стека используются инструкции PUSH и POP соответственно. При возникновении переполнения или недостатка стека будет вызвано прерывание 6:ESP (ошибка стека). Параметром прерывания будет ESP. Например:

```
MOV SS,5000   //Стек начинается по адресу 5000
MOV ESP,2999  //И будет 3000 байт в размере
              //Следующим свободным адресом на стеке будет 2999
CPUSET 9,3000 //Установить регистр ESZ (размер стека)

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

//Инструкция PUSH X это тоже самое что:
MOV SS:#ESP,X
DEC ESP

//Инструкция POP Y это тоже самое что:
INC ESP
MOV Y,SS:#ESP
```

Также можно использовать инструкции RSTACK и SSTACK для записи/чтения в произвольное место на стеке. Эти инструкции могут вызвать прерывания, которые сигнализируют ошибку стека. Вот пример использования этих инструкций:

```
RSTACK X,Y //X = MEMORY[SS+Y]
SSTACK X,Y //MEMORY[SS+X] = Y

RSTACK EAX,ESP:1 //Прочитать значение на вершине стека
RSTACK EAX,ESP:2 //Прочитать значение под вершиной

PUSH 100 //
PUSH 200 //Значение под вершиной
PUSH 300 //Значение на вершине
```

```
SSTACK ESP:2,123 //Установить значение под вершиной стека
POP EAX //EAX = 300
POP EBX //EBX = 123
POP ECX //ECX = 100
```

1.9 Подпрограммы

Zyelios CPU поддерживает несколько способов изменения течения программы: условные и безусловные переходы, абсолютные и относительные. Указатель на инструкцию (IP) указывает на инструкцию, которая сейчас выполняется. Все инструкции перехода меняют значение IP; некоторые также меняют значение сегмента CS (см. стр 13).

Самый простой тип переходов это абсолютный безусловный переход. Для того, что-бы его выполнить необходимо использовать инструкции JMP или JMPF (последняя также меняет CS).

Подпрограммы могут быть вызваны используя инструкции CALL или CALLF. Это сохранит текущий указатель инструкции (и также CS если используется CALLF) на стек процессора. Указатель инструкции (и сегмент кода) могут быть восстановлены со стека используя инструкции RET и RETF соответственно.

Вот пример для иллюстрации переходов и подпрограмм:

```
JMPF MAIN,CODE_SEGMENT //Синтаксис JMPF IP,CS

......

MAIN: //Метка
  CALL SUBROUTINE
  JMP EXIT

SUBROUTINE:
  CALL SUBROUTINE2
RET //Выход из подпрограммы

SUBROUTINE2: //Вызвано внутри подпрограммы SUBROUTINE
  ... код ...
RET
......

EXIT:
```

Также возможно совершать относительные переходы. Для этого используется инструкция JMPR (jump relative). Эта инструкция прибавляет или отнимает от значения IP некоторое число. Например:

```
JMPR +10 //Перейти на 10 байт вперёд
JMPR -10 //Перейти на 10 байт назад

JMPR LABEL-__PTR__ //Перейти к метке
                   //Метка __PTR__ указывает на текущий адрес инструкции

......

LABEL:
```

Условные переходы позволяют менять течение программы в зависимости от выполнения разных условий. Инструкция CMP используется для сравнения двух значений, и затем можно вызвать одну из указанных инструкций:

Инструкция	Операция	Описание
JNE	X <> Y	Переход если не равно
JNZ	X - Y <> 0	Переход если не ноль
JG	X > Y	Переход если больше чем
JNLE	NOT X <= Y	Переход если не меньше or равно
JGE	X >= Y	Переход если больше or равно
JNL	NOT X < Y	Переход если не меньше чем
JL	X < Y	Переход если меньше чем
JNGE	NOT X >= Y	Переход если не больше or равно
JLE	X <= Y	Переход если меньше or равно
JNG	NOT X > Y	Переход если не больше чем
JE	X = Y	Переход если равно
JZ	X - Y = 0	Переход если ноль
CNE	X <> Y	Вызов подпрограммы если не равно
CNZ	X - Y <> 0	Вызов подпрограммы если не ноль
CG	X > Y	Вызов подпрограммы если больше чем
CNLE	NOT X <= Y	Вызов подпрограммы если не меньше or равно
CGE	X >= Y	Вызов подпрограммы если больше or равно
CNL	NOT X < Y	Вызов подпрограммы если не меньше чем
CL	X < Y	Вызов подпрограммы если меньше чем

CNGE	NOT X >= Y	Вызов подпрограммы если не больше or равно
CLE	X <= Y	Вызов подпрограммы если меньше or равно
CNG	NOT X > Y	Вызов подпрограммы если не больше чем
CE	X = Y	Вызов подпрограммы если равно
CZ	X - Y = 0	Вызов подпрограммы если ноль
JNER	X <> Y	Относительный переход если не равно
JNZR	X - Y <> 0	Относительный переход если не ноль
JGR	X > Y	Относительный переход если больше чем
JNLER	NOT X <= Y	Относительный переход если не меньше or равно
JGER	X >= Y	Относительный переход если больше or равно
JNLR	NOT X < Y	Относительный переход если не меньше чем
JLR	X < Y	Относительный переход если меньше чем
JNGER	NOT X >= Y	Относительный переход если не больше or равно
JLER	X <= Y	Относительный переход если меньше or равно
JNGR	NOT X > Y	Относительный переход если не больше чем
JER	X = Y	Относительный переход если равно
JZR	X - Y = 0	Относительный переход если ноль

Существуют и другие инструкции, которые производят проверку условий, например BIT, которая тестирует состояние битов числа. Например:

```
CMP EAX,EBX
JG  LABEL1 //Переход если EAX >  EBX
JLE LABEL2 //Переход если EAX <= EBX
JE  LABEL3 //Переход если EAX  = EBX
CL  LABEL4 //Вызов подпрограммы если EAX <  EBX
CGE LABEL5 //Вызов подпрограммы если EAX >= EBX

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

1.10 Коды ошибок процессора

Есть несколько разных кодов ошибок, которые может сгенерировать процессор во время исполнения. При возникновении происходит прерывание по номеру ошибки.

Если процессор не в расширенном режиме, то выполнение кода прекращается когда возникает ошибка. При этом он выдаст код об ошибке на вывод `ERROR`. В расширенном режиме процессор попытается вызвать обработчик прерывания, который соответствует этой ошибке.

Например, для обычной ошибки памяти код на выходе может быть `7.65536` (ошибка по адресу 65536).

Вывод кода ошибки будет сброшен в ноль при сбросе состояния процессора.

В процессоре есть такие коды ошибок:

Номер	Описание
2	Конец выполнения программы
3	Деление на ноль
4	Неизвестная инструкция
5	Внутренняя ошибка процессора
6	Ошибка стека (переполнение/недополнение)
7	Ошибка записи/чтения памяти
8	Ошибка шины памяти
9	Ошибка доступа записи (защита страницы памяти)
10	Ошибка записи/чтения порта
11	Ошибка доступа к странице (защита страницы памяти)
12	Ошибка доступа чтения (защита страницы памяти)
13	Общая ошибка процессора
14	Ошибка исполнения (защита страницы памяти)
15	Выход за пределы адресного пространства
17	Ограничение количества инструкций (*только GPU*)
23	Ошибка чтения строки (*только GPU*)

1.10.1 Окончание выполнения программы (02)

Сообщение: Обнаружена инструкция STOP

Возникает когда: Выполнена операция `STOP`

Причина: Ненормальное завершение работы программы

Следствие: Отключение процессора

Сообщение: Неизвестная инструкция
Возникает когда: Выполняется любая из инструкций перехода (`JMP`, `CALL`, и т.п.)
Причина: Переход по адресу, который не указывает на корректную инструкцию процессора
Следствие: Отключение процессора

1.10.2 Деление на ноль (03)

Сообщение: Невозможно поделить на ноль
Возникает когда: Второй операнд в инструкции `DIV` нулевой
Причина: Ошибка в коде
Следствие: `LADD = 0`

Сообщение: Невозможно найти инверсию нуля
Возникает когда: Инструкция `FINV` вызвана с нулевым операндом
Причина: Ошибка в коде
Следствие: `LADD = 1`

1.10.3 Неизвестная инструкция (04)

Сообщение: Обнаружена неизвестная инструкция в потоке выполнения
Возникает когда: Инструкция, которая выполняется процессором не есть известной инструкцией
Причина: Неверное использование инструкций перехода (попытка выполнить данные как код)0
Следствие: `LADD = номер инструкции`

1.10.4 Внутренняя ошибка процессора (05)

Сообщение: Невозможно выполнить инструкцию (ошибка в микрокоде)
Возникает когда: Произошла внутренняя ошибка при выполнении микрокода инструкции
Причина: Ошибка в логике процессора
Следствие: `LADD = 01`

Сообщение: Неизвестная внутренняя ошибка
Возникает когда: Никогда
Причина: Нету
Следствие: `LADD = 02`

Сообщение: Ошибка чтения при декодировании инструкции
Возникает когда: Декодер инструкций не смог получить все нужные байты
Причина: Переход за пределы физической/логической памяти
Следствие: `LADD = 12`

Сообщение: Ошибка чтения байта первого операнда
Возникает когда: Декодер инструкций не смог получить все нужные байты
Причина: Переход за пределы физической/логической памяти
Следствие: `LADD = 22`

Сообщение: Ошибка чтения байта второго операнда
Возникает когда: Декодер инструкций не смог получить все нужные байты
Причина: Переход за пределы физической/логической памяти
Следствие: `LADD = 32`

Сообщение: Неправильный селектор RM для первого операнда
Возникает когда: Никогда
Причина: Нету
Следствие: `LADD = 42`

Сообщение: Неправильный селектор RM для второго операнда
Возникает когда: Никогда
Причина: Нету
Следствие: `LADD = 52`

Сообщение: Невозможно записать в первый операнд

Возникает когда: Никогда
Причина: Нету
Следствие: `LADD = 62`

Сообщение: Невозможно записать во второй операнд (только для `XCHG`)
Возникает когда: Никогда
Причина: Нету
Следствие: `LADD = 72`

Сообщение: Неизвестная инструкция
Возникает когда: Никогда
Причина: Нету
Следствие: `LADD = 82`

1.10.5 Ошибка стека (06)

Сообщение: Ошибка переполнения стека
Возникает когда: Выполнена инструкция `PUSH`
Причина: Значение регистра `ESP` становиться отрицательным
Следствие: `LADD = 0`

Сообщение: Ошибка недополнения стека
Возникает когда: Выполнена инструкция `POP`
Причина: Регистр `ESP` становиться больше чем `ESZ`
Следствие: `LADD = ESZ`

Сообщение: Ошибка чтения стека
Возникает когда: Выполнена инструкция `POP`
Причина: Невозможно прочитать значение из памяти
Следствие: `LADD = ESP`

Сообщение: Выход за пределы стека
Возникает когда: Выполнена инструкция `RSTACK` или `SSTACK`
Причина: Указанное значение выходит за границы стека
Следствие: `LADD` хранит ошибочный индекс

1.10.6 Ошибка записи/чтения памяти (07)

Сообщение: Ошибка чтения: адрес не существует
Возникает когда: Процессор попытался прочитать значение из внешней памяти
Причина: К шине MemBus не подключено никакое устройство
Следствие: `LADD` равен адресу к которому произошло обращение

Сообщение: Ошибка чтения: невозможно прочитать адрес из памяти
Возникает когда: Процессор попытался прочитать значение из внешней памяти
Причина: Произошла ошибка при чтении значения из памяти внешнего устройства (или страница не доступна для чтения)
Следствие: `LADD` равен адресу к которому произошло обращение

Сообщение: Ошибка записи: адрес не существует
Возникает когда: Процессор попытался записать значение в внешнюю память
Причина: К шине MemBus не подключено никакое устройство
Следствие: `LADD` равен адресу к которому произошло обращение

Сообщение: Ошибка записи: невозможно записать в память по адресу
Возникает когда: Процессор попытался записать значение в внешнюю память
Причина: Произошла ошибка при записи значения в памяти внешнего устройства (или страница не доступна для чтения)
Следствие: `LADD` равен адресу к которому произошло обращение

1.10.7 Ошибка шины памяти (08)

Сообщение: Ошибка устройства
Возникает когда: Процессор попытался прочитать значение из внешней памяти
Причина: Автобус сломался (bus fail): устройство подключенное к шине MemBus не поддерживает высокоскоростной интерфейс
Следствие: `LADD` равен адресу к которому произошло обращение

Сообщение: Ошибка устройства
Возникает когда: Процессор пытается прочитать значение из порта

Причина: Автобус сломался (bus fail): устройство подключенное к шине IOBus не поддерживает высокоскоростной интерфейс
Следствие: `LADD = -PORT_NUMBER`

1.10.8 Ошибка доступа записи (09)

Сообщение: Ошибка доступа
Возникает когда: Попытка записать значение в память
Причина: Флаг `EF` установлен в 1, страница к которой происходит доступ не имеет разрешения на запись, и уровень доступа текущей страницы выше или равен уровню доступа запрашиваемой страницы
Следствие: `LADD` равен адресу к которому произошло обращение

1.10.9 Ошибка записи/чтения порта (10)

Сообщение: Ошибка чтения: невозможно прочитать с порта
Возникает когда: Процессор попытался прочитать значение с порта
Причина: Возникла ошибка при чтении с порта (адресу не назначено устройство)
Следствие: `LADD` равен адресу к которому произошло обращение

Сообщение: Ошибка записи: невозможно записать в порт
Возникает когда: Процессор попытался записать значение в порт
Причина: Возникла ошибка при записи в порт (адресу не назначено устройство)
Следствие: `LADD` равен адресу к которому произошло обращение

1.10.10 Ошибка доступа к странице (11)

Сообщение: Невозможно сделать страницу только для чтения
Возникает когда: Выполнена инструкций `SPG`
Причина: Текущий уровень доступа выше чем уровень доступа страницы, к которой идёт обращение
Следствие: `LADD` указывает на страницу, к которой идёт обращение

Сообщение: Невозможно сделать страницу читаемой и записываемой
Возникает когда: Выполнена инструкций `CPG`
Причина: Текущий уровень доступа выше чем уровень доступа страницы, к которой идёт обращение
Следствие: `LADD` указывает на страницу, к которой идёт обращение

Сообщение: Невозможно установить флаг страницы
Возникает когда: Выполнена инструкций `SPP`
Причина: Текущий уровень доступа выше чем уровень доступа страницы, к которой идёт обращение
Следствие: `LADD` указывает на страницу, к которой идёт обращение

Сообщение: Невозможно убрать флаг страницы
Возникает когда: Выполнена инструкций `CPP`
Причина: Текущий уровень доступа выше чем уровень доступа страницы, к которой идёт обращение
Следствие: `LADD` указывает на страницу, к которой идёт обращение

Сообщение: Невозможно изменить уровень доступа страницы
Возникает когда: Выполнена инструкций `SRL`
Причина: Текущий уровень доступа выше чем уровень доступа страницы, к которой идёт обращение
Следствие: `LADD` указывает на страницу, к которой идёт обращение

Сообщение: Невозможно сменить настройки отображения страницы
Возникает когда: Выполнена инструкций `SMAP`
Причина: Текущий уровень доступа выше чем уровень доступа страницы, к которой идёт обращение
Следствие: `LADD` указывает на страницу, к которой идёт обращение

1.10.11 Ошибка доступа чтения (12)

Сообщение: Ошибка доступа
Возникает когда: Попытка прочитать значение
Причина: Флаг `EF` установлен в 1, у страницы к которой идёт обращение нету

прав на чтение, и уровень доступа текущей страницы превышает или равен уровню доступа страницы, с которой происходит чтение
Следствие: `LADD` равен адресу к которому произошло обращение

Сообщение: Страница за пределами постоянной памяти
Возникает когда: Использование инструкции `ERPG`
Причина: Страница не могла быть стёрта, поскольку она не находиться в постоянной памяти процессора
Следствие: `LADD = 0`

Сообщение: Страница за пределами постоянной памяти
Возникает когда: Использование инструкции `WRPG`
Причина: Страница не могла быть записана, поскольку она не находиться в постоянной памяти процессора
Следствие: `LADD = 0`

Сообщение: Страница за пределами постоянной памяти
Возникает когда: Использование инструкции `RDPG`
Причина: Страница не могла быть прочитана, поскольку она не находиться в постоянной памяти процессора
Следствие: `LADD = 0`

1.10.12 Общая ошибка процессора (13)

Сообщение: Невозможно записать сегмент кода
Возникает когда: Регистр `CS` выступает в роли операнда, в который идёт запись результата
Причина: Невозможно изменить сегментный регистр напрямую. Надо использовать инструкции `CALLF` ил `JMPF`
Следствие: `LADD = 1`

Сообщение: Невозможно прочитать таблицу прерываний
Возникает когда: Не удалось получить запись из таблицы прерываний
Причина: Адрес таблицы, установленный инструкцией `LIDTR`, не указывает на корректное место в памяти
Следствие: `LADD = 2`

Сообщение: Неправильный номер прерывания
Возникает когда: Номер прерывания находиться за пределами границ (0..255)
Причина: Неправильное использование инструкций `INT` или `EXTINT`
Следствие: `LADD = 3`

Сообщение: Неразрешённый вызов внешнего прерывания
Возникает когда: У прерывания, которое вызывается извне, нету флага который разрешает вызов этого прерывания как внешнего
Причина: `6ой` бит не выставлен в флагах прерывания при вызове его как внешнего
Следствие: `LADD = 4`

Сообщение: Несовместимый режим
Возникает когда: Вызов прерывания в защищённом (совместимом) режиме с менее чем 512 байт памяти
Причина: Недостаточно памяти для хранения таблицы прерываний
Следствие: `LADD = 5`

Сообщение: Невозможно записать данные о возврате прерывания
Возникает когда: Обработчик прерывания не смог записать данные о возврате на стек процессора
Причина: Прерывание было вызвано при недостаточном месте на стеке
Следствие: `LADD = 6`

Сообщение: Невозможно вызвать прерывание
Возникает когда: Прерывание вызвано с недостаточными правами
Причина: У обработчика прерывания недостаточно прав что-бы совершить переход
Следствие: `LADD = 7`

Сообщение: Невозможно прочитать таблицу страниц
Возникает когда: Таблица страниц не находиться в доступном адресном пространстве
Причина: Таблица страниц не находиться в доступном адресном пространстве
Следствие: `LADD = 8`

Сообщение: Выполнение привилегированной инструкции
Возникает когда: Невозможно выполнить инструкцию из-за прав доступа
Причина: Вызов одной из указанных инструкций без достаточных прав: `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
Следствие: LADD = Номер инструкции

1.10.13 Ошибка исполнения (14)

Сообщение: Ошибка доступа
Возникает когда: Попытка выполнить код на странице, в которой запрещено выполнять код
Причина: Неправильная инструкция перехода
Следствие: LADD = номер страницы

Сообщение: Ошибка доступа
Возникает когда: Попытка выполнить код на странице, в которой запрещено выполнять код
Причина: Выполнение кода пересекло границу между страницами
Следствие: LADD = номер страницы

1.10.14 Выход за пределы адресного пространства (15)

Сообщение: Выход за пределы адресного пространства
Возникает когда: Доступ к памяти по неправильному адресу
Причина: Адрес не есть 48-битным целым знаковым числом
Следствие: LADD = неправильный адрес

Глава 2

Расширенные функции процессора

2.1 Расширенное управление выполнением кода

В этой секции детально описано выполнение инструкций процессором. Процессор выполняет функции последовательно, декодирует их, а затем записывает результат по нужным направлениям.

При выполнении инструкции регистр `XEIP` указывает на начало инструкции, и является абсолютным адресом, в то время как регистр `IP` увеличивается на единицу при чтении каждого следующего байта, и является относительным вокруг сегмента кода.

Процесс выполнения также увеличивает регистр `TMR` на количество циклов, которые заняла данная инструкция. Регистр `CODEBYTES` хранит полное количество байт кода которые уже были прочитаны.

```
//TMR = 170

MOV EAX,10; //Инструкции длиной в 1 цикл
ADD EAX,EBX;

//TMR = 172
```

Декодер инструкций устанавливает регистр `CPAGE` в номер текущей страницы (страница, с которой была прочитана инструкция), и регистр `PPAGE` в номер страницы, на которой находилась предыдущая инструкция. *Текущая страница определяется положением первого байта инструкции, которая сейчас выполняется.*

Когда значения `CPAGE` и `PPAGE` не равны друг-другу производиться проверка прав доступа. Выполнение перехода любого рода также вызовет подобную проверку, и

сбросит значения обоих регистров CPAGE и PPAGE в номер страницы, на которую осуществлён переход.

Инструкции могут быть фиксированного, или разного размера, см. стр 63.

Вот псевдокод декодера инструкций:

```
// Вычислить абсолютный адрес выполняемой инструкции, установить текущую страницу
XEIP = IP + CS
SetCurrentPage(floor(XEIP/128))

// Не позволять выполнения если не внутри страницы ядра, или если
// вызов происходит не из страницы ядра
if (PCAP == 1) and (CurrentPage.Execute == 0) and
   (PreviousPage.RunLevel <> 0) then
  Interrupt(14,CPAGE)
end

// Сброс флагов прерываний
INTR = 0
if NIF <> undefined then
  IF = NIF
  NIF = undefined
end

// Чтение инструкции и селектора RM
Opcode = Fetch()
RM = 0
isFixedSize = false

// Проверка, является ли инструкция фиксированного размера
if ((Opcode >= 2000) and (Opcode < 4000)) or
   ((Opcode >= 12000) and (Opcode < 14000)) then
  Opcode = Opcode - 2000
  isFixedSize = true
end

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

// Если не удалось прочитать опкод/селектор RM то сообщить об ошибке
if INTR == 1 then
  IF = 1
```

```
  Interrupt(5,12)
end

// Проверка на возможность выполнения инструкции
if (PCAP == 1) and (CurrentPage.Runlevel > RunLevel[Opcode]) then
  Interrupt(13,Opcode)
end

// Декодирование селектора RM
dRM2 = floor(RM / 10000)
dRM1 = RM - dRM2*10000

// Сегментные префиксы по умолчанию (DS)
Segment1 = -4
Segment2 = -4

// Декодирование сегментных префиксов
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

// Если не удалось прочитать сегментные префиксы, сообщить об ошибке
if INTR == 1 then
```

```
  Interrupt(5,12)
end

// Проверка на правильность номера инструкции
if opcode is not valid then
  Interrupt(4,Opcode)
else
  // Прочитать дополнительные байты, если нужно
  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 INTR == 1 then
          Interrupt(5,22)
        end
      end
      if OperandCount > 1 then
        if NeedFetchByte2 then
          OperandByte2 = Precompile_Fetch() or 0
          if INTR == 1 then
            Interrupt(5,32)
          end
        end
      end
    end
  end

  // Выполнить инструкцию
  Execute()

  // Записать результат назад, если нужно
  if OperandCount > 0 then
    WriteBack(1)
    if OperandCount > 1 then
      WriteBack(2)
    end
  end
```

```
end

// Обновить таймера и счётчики
CODEBYTES = CODEBYTES + Instruction_Size
TMR = TMR + Instrucion_Cycles
TIMER = TIMER + TimerDT

// Установить страницу как предыдущую
XEIP = IP + CS
SetPreviousPage(floor(XEIP/128))
```

2.2 Система страниц

Процессор Zyelios CPU разделяет всю доступною память в страницы. Каждая страница имеет 128 байт в размере. Страницы нумеруются последовательно начиная с нуля, тоесть адреса 0 .. 127 принадлежат первой странице, 128 .. 255 второй, и так далее.

Каждая страница имеет собственную маску прав (т.е. возможность разрешения/запрета записи, чтения, выполнения) и уровень доступа.

Уровень доступа это число, которое используется для присвоения коду разных уровней доступа. Высокий уровень доступа соответствует коду уровня ядра, а нижние уровни доступа соответствуют коду пользователя. Уровень доступа - число между 0 и 255 включительно, и более высоким уровнем доступа есть тот, у которого меньшее число, ему присвоенное. Например уровень доступа 3 имеет больше прав чем уровень доступа 6.

Код, запускаемый из страницы с более низким уровнем доступа (т.е. большими правами) может читать и писать страницы более высокого уровня, но не наоборот.

```
SRL 1,12 //Установить уровень доступа страницы 1 в 12
//Страница 1 соответствует адресам 128..255

SPP 5,0 //Сделать страницу 5 читаемой
CPP 6,1 //Сделать страницу 6 не записываемой
CPP 7,2 //Предотвратить выполнение кода на странице 7
```

Уровень доступа 0 - особый уровень, который обходит все проверки прав доступа. На этом уровне также допустимо использование некоторых привилегированных инструкций, например `CPUSET`. Все остальные уровни доступа подчиняются проверкам прав.

Если страница помечена как не читаемая, то её можно прочитать только со страницы с более низким уровнем (более высокими правами), аналогично для записи.

Если страница помечена как запрещённая к выполнению, то на неё можно перейти только с уровня доступа 0.

Каждую страницу можно отобразить в другую страницу памяти. Это значит, что каждый раз как процессор будет пытаться писать или читать в эту страницу, данные на самом деле будут прочитаны с отображённой страницы. Любая страница может быть отображена.

```
MOV #130,1234 //Установить ячейку 2 страницы 1 в 1234
SMAP 0,1 //Отобразить страницу 0 в страницу 1
MOV EAX,#2 //Прочитать ячейку 2 страницы 0 (но на самом деле прочитать со страницы 1)
//EAX теперь равен 1234
```

Настройки отображения и прав доступа хранятся в *таблице страниц*. Таблица страниц активна когда процессор находиться в режиме расширенной работы с памятью (флаг `MF` установлен в 1). Если флаг обнулён, то таблица страниц будет храниться внутри процессора. При этом проверки доступа производятся лишь кода процессор находиться в расширеном режиме (флаг `EF`).

Внутренний регистр `PTBL` хранит *абсолютный* указатель на начало таблицы страниц, а в регистре `PTBE` записано количество записей в таблице. Есть возможность менять таблицу прямо во время выполнения программы.

Каждая запись в таблице имеет 2 байта в размере. Первая запись (запись номер 0) это страница по умолчанию. Все остальные записи в таблице соответствуют страницам памяти.

Если адрес, к которому идёт обращение, не имеет соответствующей записи для страницы, на которой он находиться, то он будет использовать права и флаги, которые заданы страницей по умолчанию (нулевой записью в таблице страниц). При этом будет не доступно отображение в память.

Первый байт записи хранит флаги страницы и её уровень доступа. Второй байт хранит номер страницы, в которую эта страница должна быть отображена. Флаги доступа в записи страницы *инвертированы*, то есть 1 значит запрет, а 0 значит разрешение.

Вот пример настройки таблицы страниц:

```
PageTable:
  alloc 513*2 //512 записей страниц + страница по умолчанию (64КБ памяти)

  .....

//Установить таблицу страниц
```

```
CPUSET 37,PageTable //PTBL
CPUSET 38,512 //PTBE

//Включить режим расширенной памяти
STM

//Непрямая работа с таблицей
SMAP 0,1 //Отобразить страницу 0 в 1
SPP 5,0 //Установить флаг страницы 5

//Прямое обращение к таблицу
MOV ESI,PageTable; //Сдвиг таблицы
MOV ESI:#0,0xE0; //Установить разрешения для страницы по умолчанию
MOV ESI:#2,1; //Отключить страницу 0
MOV ESI:#5,10; //Отобразить страницу 1 в 10
```

Первый байт записи в таблице:

Бит	Описание
00	Отключена ли страница? 1 если это так
01	Отображена ли страница? 1 если это так
02	Страница генерирует внешнее прерывание 30 при обращении
03	Страница генерирует внешние прерывания 28, 29
04	Зарезервированный бит
05	Доступ на чтение (0: разрешено, 1: запрещено)
06	Доступ на запись (0: разрешено, 1: запрещено)
07	Доступ на выполнение (0: разрешено, 1: запрещено)
08..15	Уровень доступа

Отключенные страницы будут вызывать ошибку памяти при попытке чтения любого адреса в них находящегося (так будто бы эта страница не находилась в адресном пространстве процессора). Отображённые страницы используют второй байт как номер *физической* страницы, в которую происходит отображение.

Возможно отлавливать доступ к странице используя биты 2 и/или 3. Это будет вызывать внешние прерывания 28 (чтение), 29 (запись), 30 (доступ) каждый раз, когда идёт обращение к странице. Номер страницы, к которой идёт доступ будет передан как параметр (для прерывания 30), и сохранён адрес обращения. Это можно использовать для переопределения операций чтения/записи (см. стр 52 для более подробной информации).

При обработке прерываний используется специальное правило. Права на вызов прерываний определяются уровнем доступа таблицы прерываний. Это значит что можно ограничить пользовательскую программу от вызова конкретных прерываний. Это работает будто-бы вызов происходил с помощью инструкции `CALL` с адреса, на котором находиться прерывание. Это даёт возможность использовать прерывания для интерфейса между двумя частями кода с разными правами доступа.

```
STEF //Включить расширенный режим
LIDTR 2048 //Таблица прерываний на страницах 16-23

SRL 16,0 //Установить уровень доступа прерываний 0..31 в 0
SRL 17,1 //Установить уровень доступа прерываний 32..63 в 1
SRL 18,2 //Установить уровень доступа прерываний 64..95 в 2
SRL 19,3 //Установить уровень доступа прерываний 96..127 в 3
```

Текущая страница при выполнении инструкций определяется регистром `CPAGE`. *Вообще текущая страница определяется только положением первого байта инструкции, которая сейчас выполняется.* Система страниц производит проверки когда регистр `PPAGE` (значение `CPAGE` для предыдущей инструкции) не совпадает по значению с `CPAGE` (т.е. происходит переход через границу страниц).

Любой переход также вызовет подобную проверку, и сбросит значения регистров `CPAGE` и `PPAGE` в номер новой страницы.

Пересечь границу между страницами, или совершить прямой переход, возможно лишь если флаги следующей страницы (целевой страницы перехода) позволяют выполнение кода на той странице, либо если предыдущая страница имеет нулевой уровень доступа.

Весь внешний доступ к памяти процессора через его шину будет производить проверки чтения/записи. Уровень доступа для внешних операций ввода/вывода определяется регистром `XTRL` (внешний уровень доступа). По умолчанию все внешние операции имеют уровень доступа 0.

Что-бы подытожить, вот те проверки, которые производятся системой страниц (но только в расширенном режиме):

- Проверка перехода границ страницы
- Проверка при прямом переходе по адресу
- Проверки на права доступа к таблице прерываний
- Доступ к памяти на чтение/запись

- Внешний доступ к памяти/шине процессора
- Логика отображения страниц при записи/чтении
- Перехват доступа к странице

2.3 Внутренняя память

Для процессора Zyelios CPU можно настроить внутреннюю оперативную и постоянную память. Это позволяет сохранять программу прямо внутри процессора. Содержимое постоянной памяти будет скопировано в оперативную память при каждом сбросе состояния процессора. В постоянную память также можно программно писать, и стирать данные.

Всего есть три инструкции для работы с внутренней памятью: `ERPG` (стереть страницу постоянной памяти), `WRPG` (записать страницу постоянной памяти), `RDPG` (прочитать страницу постоянной памяти).

Примеры использования:

```
ERPG 4 //Стирает данные из постоянной памяти
WRPG 4 //Записывает страницу 4 в постоянную память
       //После сброса процессора она будет восстановлена
RDPG 4 //Восстановить страницу 4 из постоянной памяти

ORG 512 //Разместить переменные на странице 4
SOME_AREA:
 ... данные ...
```

2.4 Побитовые операции

ZCPU может работать с целыми числами разной ширины. Он поддерживает 8, 16, 32, и 48 битные целые числа, и имеет дополнительный набор инструкций для работы с ними: `BAND`, `BOR`, `BXOR`, `BSHL`, `BSHR`, `BNOT`. Например:

```
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
```

Также есть дополнительные операции, которые работают с отдельными битами числа: `BIT`, `SBIT`, `TBIT`, `CBIT`.

Инструкция `BIT` проверяет, установлен ли данный бит числа. Проверить результат этой операции можно используя условный переход:

```
MOV EAX,105
BIT EAX,0
JNZ LABEL //Переход удался, если бит не нулевой

BIT EAX,1
JNZ LABEL //Переход не удался, если бит нулевой
```

Инструкции `SBIT` и `CBIT` выставляют или убирают определённый бит числа. Инструкция `TBIT` меняет этот бит на противоположный:

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

CBIT EAX,6
//EAX = 43    0101011

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

2.5 Поддержка блоков в памяти

Некоторые инструкции процессора поддерживают инструкцию `BLOCK`, которая выполняется перед выполнением нужной инструкции. Она используется для указания блока, над которым будет действовать следующая инструкция, например можно задать права для целого блока памяти сразу:

```
BLOCK 1024,8192 //8-килобайтовый блок начиная с адреса 1024
SRL 0,4 //Выставить уровень доступа всего блока в 4
```

Первый операнд инструкции `BLOCK` указывает на абсолютное смещение в памяти с которого начинается блок (должен быть выровнен по краю страницы если используется для инструкций, которые работают со страницами), а второй операнд указывает на размер блока (если используется для работы со страницами, то размер тоже должен быть кратный размеру страницы).

Инструкция `BLOCK` выставляет два внутренних регистра `BlockStart`, `BlockSize`. После выполнения инструкции, которая поддерживает выставление блоков это регистры будут сброшены обратно в 0.

Размер блока должен быть отличным от нуля, иначе инструкция не будет действовать.

Есть возможность возникновения прерывания между инструкцией `BLOCK` и другой инструкцией. Поэтому желательно не использовать инструкцию `BLOCK` внутри обработчика прерывания, поскольку его состояние восстановлено не будет.

Данные инструкции поддерживают этот префикс: `SPP`, `CPP`, `SRL`, `SMAP`.

2.6 Копирование, сдвиг, обмен больших блоков данных

(MCOPY, MSHIFT, MXCHG)

2.7 Поддержка стекового кадра

(using EBP register, and ENTER/LEAVE opcodes)

2.8 Прерывания/расширенный режим

If an interrupt occurs while reading or writing to memory pointer specified by the operand the operation will ???.

The processor interrupt handler performs a variety of checks before turning the control over back to the execution.

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

```
// Если прерывание включено, запретить любой обмен данными на шине
INTR = 1
BusLock = 1

// Выставить регистры
LINT = interruptNo
LADD = interruptParameter or XEIP

// Выдать сигнал прерывания на выход процессора
SignalError(interruptNo,LADD)

if IF == 1 then
  if EF == 1 then // Расширеный режим
    // Проверка границ
    if (interruptNo < 0) or (interruptNo > 255) then
      if not cascadeInterrupt then Interrupt(13,3) end
    end

    // Проверка границ таблицы прерываний
    if interruptNo > NIDT-1 then
      if interruptNo == 0 then Reset = 1 end
      if interruptNo == 1 then Clk = 0 end
    end

    // Вычислить смещение в таблице прерываний
    interruptOffset = IDTR + interruptNo*4

    // Отключить запрет на работу шины процессора
    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

// Вызвать предыдущую страницу, правильно настроив
// состояние страниц
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
  // Записать данные о возврате
  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

  // Произвести переход
  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

  // Выставить 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 PF == 1 then // Расширенный режим (совместимости)
  // Проверка границ
  if (interruptNo < 0) or (interruptNo > 255) then
    if not cascadeInterrupt then Interrupt(13,3) end
  end

  // Проверка памяти
  if RAMSize < 512 then
    if not cascadeInterrupt then Interrupt(13,5) end
  end

  // Посчитать абсолютное смещение записи прерывания
  interruptOffset = IDTR + interruptNo*2

  if interruptOffset > RAMSize-2 then interruptOffset = RAMSize-2 end
  if interruptOffset < 0         then interruptOffset = 0 end

  interruptOffset = Memory[interruptOffset]
  interruptFlags = Memory[interruptOffset+1]
  if (interruptFlags == 32) or (interruptFlags == 96) then
    BusLock = 0
    IF = 0
    INTR = 0
    if (interruptNo == 4 ) or
       (interruptNo == 7 ) or
       (interruptNo == 9 ) or
       (interruptNo == 10) then
      Push(LADD)
    end
    if (interruptNo == 4 ) or
       (interruptNo == 31) then
      Push(LINT)
```

```
      end
      if Push(IP) and Push(XEIP) then
        Jump(interruptOffset)
      end
      IF = 1

      if INTR == 1 then
        if not cascadeInterrupt then Interrupt(13,6) end
      else
        INTR = 1
      end
      CMPR = 0
      BusLock = 1
    else
      if interruptNo == 1 then Clk = 0 end
      CMPR = 1
    end
  end

  if (PF == 0) and (EF == 0) then // Обычный режим
    if (interruptNo < 0) or (interruptNo > 255) or (interruptNo > NIDT-1) then
      // Прерывание не обработано
      Exit()
    end
    if interruptNo == 0 then Reset = 1 end
    if interruptNo ~= 31 then Clk = 0 end
  end
end

// Сделать возможным запись по шине процессора
BusLock = 0
```

2.9 Внешние прерывания

(NMI stuff)

2.10 Перехват доступа к памяти

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

2.11 Внутренний таймер

В ZCPU есть внутренний таймер, который можно использовать для точного измерения времени выполнения кода. Таймер может быть также использован для вызова прерываний с регулярным интервалом.

Инструкция `TIMER` возвращает значение внутреннего таймера в секундах:

```
TIMER EAX
//EAX теперь равен количеству секунд, прошедших с момента старта процессора
```

Можно настроить таймер так, что-бы он вызывал внешнее прерывание после некоторого количества секунд, или циклов, настроив один из специальных регистров.

Регистр `TimerMode` управляет режимом таймера. Если он установлен в 0, то таймер будет отключён. Если `TimerMode` выставлен в 1, то как источник сигнала для таймера будет использован регистр `TMR`, а если `TimerMode` установлен в 2, то таймер будет использовать регистр `TIMER`.

Регистром `TimerRate` задаётся количество циклов или количество секунд, которые должны быть отсчитаны таймером перед вызовом следующего прерывания. Регистр `TimerPrevTime` хранит предыдущее значение регистра `TMR` или `TIMER`, когда прерывание было вызвано прошлый раз.

Регистром `TimerAddress` указывается номер внешнего прерывания, которое будет вызвано когда таймер сработает.

При изменении значения регистра `TimerMode` значение `TimerPrevTime` будет сброшено.

Вот пример, как настроить таймер:

```
CPUSET 65,90; //Вызов каждые 90 циклов
CPUSET 67,40; //Вызов внешнего прерывания #40

CPUSET 64,1;  //Включить таймер
```

Также можно установить таймер на секунды:

```
CPUSET 65,1.5; //Вызов каждые 1.5 секунд
CPUSET 67,40;  //Вызов внешнего прерывания #40

CPUSET 64,2;   //Включить таймер
```

Если нужна точность в интервалах между срабатываниями ми таймера, то можно сбросить значения предыдущего времени срабатывания таймера (если нужно):

```
ExternalInterrupt:
  CLI; //Отключить прерывания

  .....

  CPUGET EAX,29; //Прочитать счётчик циклов
  ADD EAX,4;     //Добавить 4 пропущенны[ цикла
  CPUSET 66,EAX; //Записать в последнее время срабатывания таймера

  STI; //Включить прерывания
EXTRET;
```

Также можно просто перезапустить таймер:

```
ExternalInterrupt:
  CLI; //Отключить прерывания

  .....

  CPUSET 64,1; //Перезапустить таймер
  STI; //Включить прерывания
EXTRET;
```

Если интервал таймера установлен в 0, а последнее время срабатывания находится позже чем текущее время, то таймер можно использовать для единократного вызова события через некоторое время.

2.12 Векторное расширение

(using VMODE,VADD,etc)

2.13 Режим аппаратной отладки

(Using hardware debug mode)

2.14 Кеширование и оптимизации реального времени

Процессор Zyelios CPU кеширует выполняемый микрокод для более быстрой работы. Это намного повышает скорость повторного исполнения кода, но при этом первый

запуск этого блока кода будет использовать значительное количество циклов процессора. Более быстро выполняется код в циклах, которые много раз повторяются.

Некоторые ограничения, которые накладываются системой кеширования:

1. При выполнении кешированого блока микрокода процессор использует временные скоростные регистры вместо настоящих. Настоящее значение регистра будет обновлено только после выполнения блока микрокода.

2. Процессор кеширует инструкции первый раз когда они декодируются. Чтение или запись в память самим процессором могут обнулить кеш, но операции ввода-вывода других устройств в эту-же область не меняют кеша, создавая возможность выполнения несуществующего кода.

3. В каждом кешированом блоке микрокода содержится до 24 инструкций. Если происходит переход по адресу, то выполнение обрывается заранее.

4. Все операции чтения-записи могут быть задержаны на несколько циклов, или не выполнены вообще.

Глава 3

Формат инструкций

3.1 Формат

Каждая инструкция в ZCPU начинается с одного байта, который определяет, какая инструкция должна быть выполнена. Если у инструкции есть операнды, то далее следует селектор *RM* (register/memory). Этот дополнительный байт указывает, какого типа операнды используются с инструкцией.

Номер инструкции также содержит информацию про использование сегментного префикса, и про текущий режим выполнения (см. стр 63). Номер инструкции может принадлежать к одному из таких интервалов:

- `000 - 999`: инструкции переменной длины
- `1000 - 1999`: инструкции переменной длины с сегментным префиксом для 1ого операнда
- `10000 - 10999`: инструкции переменной длины с сегментным префиксом для 2ого операнда
- `11000 - 11999`: инструкции переменной длины с сегментным префиксом для обоих операндов
- `2000 - 2999`: инструкции постоянной длины
- `3000 - 3999`: инструкции постоянной длины с сегментным префиксом для 1ого операнда
- `12000 - 12999`: инструкции постоянной длины с сегментным префиксом для 2ого операнда

- 13000 - 13999: инструкции постоянной длины с сегментным префиксом для обоих операндов

Далее могут следовать несколько байтов, которые указуют сегментный префикс, константы, и т.п. Константные значения всегда последние в описании каждой инструкции. Вот пример разных инструкций:

```
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
```

3.2 Селектор Регистра-Памяти

Селектор RM состоит из двух частей - селектор для первого операнда, и второго операнда: $RM = RM_1 + 10000 \cdot RM_2$.

Например, вот пример значений байтов RM, и как они декодируются:

```
                RM        RM1   RM2
STEF            н/о       н/о   н/о
INC EAX         1         1     н/о
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
```

Вот эти селекторы сейчас поддерживаются процессором:

Селектор	Операнд	Описание
0	123	Константное значение
1	EAX	Регистр EAX
2	EBX	Регистр EBX
3	ECX	Регистр ECX
4	EDX	Регистр EDX

5	ESI	Регистр ESI
6	EDI	Регистр EDI
7	ESP	Регистр ESP
8	EBP	Регистр EBP
9	CS	Регистр CS
10	SS	Регистр SS
11	DS	Регистр DS
12	ES	Регистр ES
13	GS	Регистр GS
14	FS	Регистр FS
15	KS	Регистр KS
16	LS	Регистр LS
17	#EAX, ES:#EAX	Ячейка памяти EAX + сегмент
18	#EBX, ES:#EBX	Ячейка памяти EBX + сегмент
19	#ECX, ES:#ECX	Ячейка памяти ECX + сегмент
20	#EDX, ES:#EDX	Ячейка памяти EDX + сегмент
21	#ESI, ES:#ESI	Ячейка памяти ESI + сегмент
22	#EDI, ES:#EDI	Ячейка памяти EDI + сегмент
23	#ESP, ES:#ESP	Ячейка памяти ESP + сегмент
24	#EBP, ES:#EBP	Ячейка памяти EBP + сегмент
25	#123, ES:#123	Ячейка памяти за константным указателём
26	ES:EAX	Регистр EAX + сегмент
27	ES:EBX	Регистр EBX + сегмент
28	ES:ECX	Регистр ECX + сегмент
29	ES:EDX	Регистр EDX + сегмент
30	ES:ESI	Регистр ESI + сегмент
31	ES:EDI	Регистр EDI + сегмент
32	ES:ESP	Регистр ESP + сегмент
33	ES:EBP	Регистр EBP + сегмент
34	No syntax	Ячейка памяти EAX + константа
35	No syntax	Ячейка памяти EBX + константа
36	No syntax	Ячейка памяти ECX + константа
37	No syntax	Ячейка памяти EDX + константа
38	No syntax	Ячейка памяти ESI + константа
39	No syntax	Ячейка памяти EDI + константа
40	No syntax	Ячейка памяти ESP + константа
41	No syntax	Ячейка памяти EBP + константа
42	No syntax	Регистр EAX + константа
43	No syntax	Регистр EBX + константа
44	No syntax	Регистр ECX + константа
45	No syntax	Регистр EDX + константа
46	No syntax	Регистр ESI + константа
47	No syntax	Регистр EDI + константа

48	No syntax	Регистр ESP + константа
49	No syntax	Регистр EBP + константа
50	ES:123	Константное значение с сегментом
1000	PORT0	Порт 0
1001	PORT1	Порт 1
....	....	....
2023	PORT1023	Порт 1023
2048	R0	Расширеный регистр R0
....	....	....
2079	R31	Расширеный регистр R31
2080	#R0, ES:#R0	Ячейка памяти R0 + сегмент
....	....	....
2111	#R31, ES:#R31	Ячейка памяти R31 + сегмент
2112	ES:R0	Extended register R0 + сегмент
....	....	....
2143	ES:R31	Extended register R31 + сегмент
2144	No syntax	Ячейка памяти R0 + константа
....	....	....
2175	No syntax	Ячейка памяти R31 + константа
2176	No syntax	Extended register R0 + константа
....	....	....
2207	No syntax	Extended register R31 + константа

3.3 Сегментные префикс

Сегменты указываются байтом, который следует за селектором RM. Например, вот несколько примеров разных префиксов:

```
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
```

Вот список доступных сегментных префиксов:

Value	Регистр	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		

3.4 Локальный режим выполнения

Процессор Zyelios CPU поддерживает два формата машинного кода: инструкции переменной длины, и инструкции постоянной длины. Процессора автоматически определяет тип машинного кода для поддержки обратной совместимости. Для этого существует насколько правил.

Пример инструкций в обычном режиме:

```
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
```

Пример инструкций в режиме фиксированого размера:

```
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
```

Пример инструкций в режиме совместимости:

```
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
```

Глава 4

Внутренние регистры

У процессора есть несколько внутренних регистров, которые используются для хранения состояния процессора, и управления его работой. Записывать и читать из этих регистров можно с помощью инструкций `CPUSET` и `CPUGET`.

Например:

```
CPUGET EAX,24 //Прочитать регистр 24 в EAX (указатель на таблицу прерываний)
CPUSET 9,EBX //Выставить регистр 9 (размер стека) в значение EBX

CPUGET EAX,1000 //Несуществующий регистр запишет 0 в EAX
```

Регистры `XEIP`, `CPAGE`, `PPAGE`, `SerialNo`, `CODEBYTES`, `TimerDT` только для чтения - их значение невозможно изменить.

Изменение регистров `IP` или `CS` будет эквивалентно безусловному переходу.

Mnemonic	Number	Description
IP	00	Указатель на инструкцию
EAX	01	Регистр общего назначения A
EBX	02	Регистр общего назначения B
ECX	03	Регистр общего назначения C
EDX	04	Регистр общего назначения D
ESI	05	Исходный индекс
EDI	06	Целевой индекс
ESP	07	Указатель стека
EBP	08	Базовый указатель
ESZ	09	Размер стека
CS	16	Сегмент кода
SS	17	Сегмент стека
DS	18	Сегмент данных
ES	19	Дополнительный сегмент

GS	20	Дополнительный сегмент
FS	21	Дополнительный сегмент
KS	22	Дополнительный сегмент
LS	23	Дополнительный сегмент
IDTR	24	Указатель на таблицу прерываний
CMPR	25	Результат операции сравнения
XEIP	26	Указатель на начало текущей инструкции
LADD	27	Код текущего прерывания
LINT	28	Номер текущего прерывания
TMR	29	Счётчик циклов
TIMER	30	Внутренний таймер
CPAGE	31	Текущий номер страницы
IF	32	Флаг состояния системы прерываний
PF	33	Флаг режима совместимости
EF	34	Флаг расширеного режима
NIF	35	Значение флага прерываний на следущем шаге
MF	36	Флаг расширеного режима работы с памятью
PTBL	37	Адрес таблицы страницы
PTBE	38	Количество записей в таблице страниц
PCAP	39	Возможность работы с системой страниц
RQCAP	40	Возможность задержаных обращений к памяти
PPAGE	41	Номер предыдущей страницы
MEMRQ	42	Тип запроса памяти
RAMSize	43	Количество внутренней памяти
External	44	Внешняя операция ввода-вывода
BusLock	45	Состояние шины
Idle	46	Сигнал о пропуске циклов до следущего сигнала синхронизации
INTR	47	Обработка прерывания
SerialNo	48	Серийный номер процессора
CODEBYTES	49	Количество запущеного кода
BPREC	50	Режим точности операций с двоичными числами
IPREC	51	Режим точности операций с целыми числами
NIDT	52	Количество записей в таблице прерываний
BlockStart	53	Начало блока
BlockSize	54	Размер блока
VMODE	55	Режим векторных операций (2: 2D, 3: 3D)
XTRL	56	Уровень доступа для внешнего доступа
HaltPort	57	Номер порта, до получения значения на котором процессор остановлен
HWDEBUG	58	Режим аппаратной отладки

DBGSTATE	59	Состояние аппаратного режима отладки
DBGADDR	60	Адрес/параметр аппаратной отладки
CRL	61	Текущий уровень доступа
TimerDT	62	Шаг таймера
MEMADDR	63	Адрес, запрашиваемый операцией связаной с памятью
TimerMode	64	Режим таймера
TimerRate	65	Частота срабатывания таймера
TimerPrevTime	66	Время предыдущего срабатывания
TimerAddress	67	Номер внешнего прерывания для вызова при срабатывании таймера

Глава 5

Список инструкций процессора

5.1 Основной набор инструкций ZCPU

В этой секции описан основной набор инструкций, доступных для использования в ZCPU. Некоторые из указанных инструкций не могут быть использованы в производных от ZCPU архитектурах (ZGPU, ZSPU). Недоступны указанные инструкции: `EXTINT`

Для некоторых инструкций указаны ошибки, которые могут быть ими вызваны, исключая ошибки которые могут быть вызваны декодированием инструкции, например если один из операндов это указатель на чтение из памяти.

5.1.1 000 STOP

Мнемоника: `STOP`
Кодирование: `000`

Вызывает прерывание #2 (тоже самое что выполнение `INT 2`). Останавливает выполнение программы процессором, если расширенный режим не включен.

Используется для обнаружения конца программы/некорректного перехода, поскольку часто является следствием некорректного перехода по адресу. Работает как `NOP` если система прерываний отключена (флаг `IF` установлен в 0).

Псевдокод:

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

5.1.2 001 JNE/JNZ

Мнемоника: `JNE/JNZ X`
Кодирование: `001 RM [Сегмент1] [Данные1]`

Осуществляет условный переход по заданному адресу если в результате предыдущего сравнения значения были не равны друг другу.

Также данная инструкция может быть использована для проверки установленности бита числа:

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

Переход будет осуществлён только когда бит равен единице.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

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

5.1.3 002 JMP

Мнемоника: `JMP X`
Кодирование: `002 RM [Сегмент1] [Данные1]`

Безусловный переход по заданному адресу.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

```
Jump(X)
```

5.1.4 001 JG/JNLE

Мнемоника: `JG/JNLE X`
Кодирование: `001 RM [Сегмент1] [Данные1]`

Осуществляет условный переход по заданному адресу если в результате предыдущего сравнения первое значение было больше второго.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

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

5.1.5 004 JGE/JNL

Мнемоника: `JGE/JNL X`
Кодирование: `004 RM [Сегмент1] [Данные1]`

Осуществляет условный переход по заданному адресу если в результате предыдущего сравнения первое значение было больше или равно второму.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

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

5.1.6 005 JL/JNGE

Мнемоника: `JL/JNGE X`
Кодирование: `005 RM [Сегмент1] [Данные1]`

Осуществляет условный переход по заданному адресу если в результате предыдущего сравнения первое значение было меньше второго.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

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

5.1.7 006 JLE/JNG

Мнемоника: `JLE/JNG X`
Кодирование: `006 RM [Сегмент1] [Данные1]`

Осуществляет условный переход по заданному адресу если в результате предыдущего сравнения первое значение было меньше или равно второму.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

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

5.1.8 007 JE/JZ

Мнемоника: `JE/JZ X`
Кодирование: `007 RM [Сегмент1] [Данные1]`

Осуществляет условный переход по заданному адресу если в результате предыдущего сравнения значения были равны.

Также даная инструкция может быть использована для проверки установленности бита числа:

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

Переход будет осуществлён только когда бит равен нулю.

Может вызвать ошибку `14` (ошибка прав на выполнение кода) если будет осуществлён переход по адресу, который находиться в защищённой области памяти при включенном расширенном режиме. См. стр 40 для более детальной информации о механизме страниц и защиты памяти.

Псевдокод:

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

5.1.9 008 CPUID

Мнемоника: `CPUID X`
Кодирование: `008 RM [Сегмент1] [Данные1]`

Инструкция `CPUID` позволяет получить информацию о текущей версии процессора, и его возможностям. Результат будет записан в регистр `EAX`.

Параметр инструкции указывает какую информацию нужно получить:

Параметр	Описание
0	Текущая версия процессора (10.00, записывается как `1000`)
1	Количество внутренней оперативной памяти в байтах
2	Тип процессора
3	Количество внутренней постоянной памяти в байтах

Результат будет записан в регистр `EAX`. Тип процессора может указывать на:

`EAX`	Описание
0	ZCPU
1	ZGPU (бета версия)
2	ZSPU
3	ZGPU

Псевдокод:

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

5.1.10 009 PUSH

Мнемоника: `PUSH X`
Кодирование: `009 RM [Сегмент1] [Данные1]`

Отсылает значение на стек процессора (см. стр 18 для более детальной информации о работе стека процессора). Также будет проведена проверка на ошибку выхода за пределы стека (проверка перехода указателя стека через ноль).

Как указатель на стек используется регистр `ESP`, а как размер стека - регистр `ESZ`.

Пример использования:

```
PUSH 10
PUSH 20

POP EAX //EAX теперь равен 20
```

Может вызвать ошибку `6` (переполнение/недополнение стека) если указатель стека выйдет за допустимые пределы.

Может вызвать ошибку `7` (ошибка чтения/записи памяти) если инструкция не смогла выполнить необходимую запись в память.

Псевдокод:

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

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

5.1.11 010 ADD

Мнемоника: `ADD X,Y`

Кодирование: `010 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Находит сумму двух значений, и записывает результат в первый операнд.

Псевдокод:

```
X = X + Y
```

5.1.12 011 SUB

Мнемоника: `SUB X,Y`

Кодирование: `011 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Отнимает второй операнд от первого, и записывает результат в первый операнд.

Псевдокод:

```
X = X - Y
```

5.1.13 012 MUL

Мнемоника: `MUL X,Y`
Кодирование: `012 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Множит два значения между собой, и записывает результат в первый операнд.

Псевдокод:

```
X = X * Y
```

5.1.14 013 DIV

Мнемоника: `DIV X,Y`
Кодирование: `013 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Делит первый операнд на второй, и записывает результат в первый операнд. Также проверяет второй операнд на равность нулю.

Может вызвать ошибку `3` (деление на ноль) если второй операнд равен нулю.

Ошибка будет проигнорирована если отключены прерывания.

Псевдокод:

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

5.1.15 014 MOV

Мнемоника: `MOV X,Y`
Кодирование: `014 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Копирует содержимое второго операнда в первый.

Псевдокод:

```
X = Y
```

5.1.16 015 CMP

Мнемоника: `CMP X,Y`
Кодирование: `015 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Сравнивает два значения, и запоминает результат этого сравнения.

Эта инструкция используется вместе с инструкциями условного перехода (см. стр 19), например:

```
CMP EAX,EBX
JG  LABEL1 //Переход, если EAX >  EBX
JLE LABEL2 //Переход, если EAX <= EBX
JE  LABEL3 //Переход, если EAX  = EBX
```

Псевдокод:

```
CMPR = X - Y
```

5.1.17 018 MIN

Мнемоника: `MIN X,Y`
Кодирование: `018 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает меньшее из двух значений в первый операнд, например:

```
MOV EAX,100
MOV EBX,200
MIN EBX,EAX //Записывает 100 в EBX
```

Псевдокод:

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

5.1.18 019 MAX

Мнемоника: `MAX X,Y`
Кодирование: `019 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает большее из двух значений в первый операнд, например:

```
MOV EAX,100
MOV EBX,200
MAX EAX,EBX //Записывает 200 в EAX
```

Псевдокод:

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

5.1.19 020 INC

Мнемоника: `INC X`
Кодирование: `020 RM [Сегмент1] [Данные1]`

Увеличивает операнд на единицу, например:

```
MOV EAX,100
INC EAX //EAX теперь равен 101
```

Псевдокод:

```
X = X + 1
```

5.1.20 021 DEC

Мнемоника: `DEC X`
Кодирование: `021 RM [Сегмент1] [Данные1]`

Уменьшает операнд на единицу, например:

```
MOV EAX,100
DEC EAX //EAX теперь равен 99
```

Псевдокод:

```
X = X - 1
```

5.1.21 022 NEG

Мнемоника: `NEG X`
Кодирование: `022 RM [Сегмент1] [Данные1]`

Меняет знак операнда на противоположный. Например:

```
MOV EAX,123
NEG EAX //EAX теперь равен -123

MOV EBX,0
NEG EBX //EBX теперь равен -0, нуль со знаком
```

Псевдокод:

```
X = -X
```

5.1.22 023 RAND

Мнемоника: `RAND X`
Кодирование: `023 RM [Сегмент1] [Данные1]`

Аппаратно генерирует случайное число в интервале от 0.0 до 1.0, включительно.

Псевдокод:

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

5.1.23 024 LOOP

Мнемоника: `LOOP X`
Кодирование: `024 RM [Сегмент1] [Данные1]`

Выполняет условный переход по заданному адресу если регистр ECX не равен нулю. Обычно используется для создания циклов с конечным числом повторений:

```
MOV ECX,100;
LABEL:
  <...>
LOOP ECX; //Повторяет 100 раз
```

Цикл будет остановлен только когда ECX станет равным нулю.

Псевдокод:

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

5.1.24 025 LOOPA

Мнемоника: `LOOPA X`
Кодирование: `025 RM [Сегмент1] [Данные1]`

Выполняет условный переход по заданному адресу если регистр EAX не равен нулю. Обычно используется для создания циклов с конечным числом повторений:

```
MOV EAX,100;
LABEL:
  <...>
LOOP EAX; //Повторяет 100 раз
```

Цикл будет остановлен только когда EAX станет равным нулю. Аналогичен инструкции `LOOP`.

Псевдокод:

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

5.1.25 026 LOOPB

Мнемоника: `LOOPB X`
Кодирование: `026 RM [Сегмент1] [Данные1]`

Выполняет условный переход по заданному адресу если регистр EBX не равен нулю. Обычно используется для создания циклов с конечным числом повторений:

```
MOV EBX,100;
LABEL:
  <...>
LOOP EBX; //Повторяет 100 раз
```

Цикл будет остановлен только когда EBX станет равным нулю. Аналогичен инструкции `LOOP`.

Псевдокод:

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

5.1.26 027 LOOPD

Мнемоника: `LOOPD X`
Кодирование: `027 RM [Сегмент1] [Данные1]`

Выполняет условный переход по заданному адресу если регистр EDX не равен нулю. Обычно используеться для создания циклов с конечным числом повторений:

```
MOV EDX,100;
LABEL:
  <...>
LOOP EDX; //Повторяет 100 раз
```

Цикл будет остановлен только когда EDX станет равным нулю. Аналогичен инструкции `LOOP`.

Псевдокод:

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

5.1.27 028 SPG

Мнемоника: `SPG X`
Кодирование: `028 RM [Сегмент1] [Данные1]`

Делает заданную страницу доступной только для чтения. Инструкция убирает флаг возможности записи, и выставляет флаг возможности чтения страницы указанной операндом. См. стр 40 для более детальной информации о системе страниц.

Например:

```
SPG 1 //Сделать адреса 128..255 доступными только для чтения
SPG 2 //Сделать адреса 256..511 доступными только для чтения

CPG 1 //Сделать адреса 128..255 доступными для чтения и записи
```

Может вызвать ошибку `11` (ошибка доступа к странице) если инструкция обращается к странице, к которой нету прав доступа (уровень доступа текущей страницы больше чем уровень доступа страницы, над которой производится операция).

В случае ошибки номер страницы, настройки которой менялись инструкцией, будет передан как параметр прерывания (регистр `LADD`).

Псевдокод:

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

5.1.28 029 CPG

Мнемоника: `CPG X`
Кодирование: `029 RM [Сегмент1] [Данные1]`

Делает заданную страницу доступной для чтения и записи. Инструкция выставляет флаг возможности записи, и возможности чтения страницы указанной операндом. См. стр 40 для более детальной информации о системе страниц.

Например:

```
SPG 1 //Сделать адреса 128..255 доступными только для чтения
SPG 2 //Сделать адреса 256..511 доступными только для чтения

CPG 1 //Сделать адреса 128..255 доступными для чтения и записи
```

Может вызвать ошибку `11` (ошибка доступа к странице) если инструкция обращается к странице, к которой нету прав доступа (уровень доступа текущей страницы больше чем уровень доступа страницы, над которой производится операция).

В случае ошибки номер страницы, настройки которой менялись инструкцией, будет передан как параметр прерывания (регистр `LADD`).

Псевдокод:

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

5.1.29 030 POP

Мнемоника: `POP X`
Кодирование: `030 RM [Сегмент1] [Данные1]`

Убирает значение с вершины стеке процессора, и записывает его в операнд (см. стр 18 для более подробной информации о работе стека процессора). Проверяет ошибку недополнения стека сравнивая значение нового указателя стека, и регистра размера стека.

Использует регистр `ESP` как указатель стека, и регистр `ESZ` для размера стека.

Пример использования:

```
PUSH 10
PUSH 20

POP EAX //EAX теперь равен 20
```

Может вызвать ошибку `6` (переполнение/недополнение стека) если указатель стека выйдет за допустимые пределы.

Псевдокод:

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

5.1.30 031 CALL

Мнемоника: `CALL X`
Кодирование: `031 RM [Сегмент1] [Данные1]`

Вызывает подпрограмму. Выполнение основной программы будет продолжено из этого самого момента после возвращения из подпрограммы используя инструкцию `RET`.

Эта инструкция сохраняет текущий указатель на инструкцию `IP` на стек (адрес возврата), и восстанавливает его при вызове соответствующей инструкции `RET`. Повреждение или нарушение работы стека вызовет ошибку при возврате из функции.

Данная инструкция может вызвать ошибку стека, если на стеке недостаточно места для сохранения адреса возврата.

Например:

```
CALL SUBROUTINE0;
CALL SUBROUTINE1;

SUBROUTINE0:
RET

SUBROUTINE1:
  CALL SUBROUTINE0;
RET
```

Может вызвать ошибку `6` (переполнение/недополнение стека) если указатель стека выйдет за допустимые пределы.

Псевдокод:

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

5.1.31 032 BNOT

Мнемоника: `BNOT X`
Кодирование: `032 RM [Сегмент1] [Данные1]`

Меняет все биты числа на противоположные. Количество бит которые будут затронуты зависит от текущего значения регистра `BPREC` (точность двоичных операций).

Например:

```
CPUSET 50,8 //Выставить 8-битную точность

MOV EAX,1
BNOT EAX //EAX теперь равен 254
```

Псевдокод:

```
X = NOT X
```

5.1.32 033 FINT

Мнемоника: `FINT X`
Кодирование: `033 RM [Сегмент1] [Данные1]`

Округляет значение вниз (до меньшего целого числа):

```
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
```

Псевдокод:

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

5.1.33 034 FRND

Мнемоника: FRND X
Кодирование: 034 RM [Сегмент1] [Данные1]

Округляет значение за правилом округления (до более близкого целого числа):

```
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
```

Псевдокод:

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

5.1.34 035 FFRAC

Мнемоника: FFRAC X
Кодирование: 035 RM [Сегмент1] [Данные1]

Возвращает дробную часть операнда:

```
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
```

Псевдокод:

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

5.1.35 036 FINV

Мнемоника: `FINV X`
Кодирование: `036 RM [Сегмент1] [Данные1]`

Находит значение, обратное операнду. Проверяет на возникновение ошибки деления на ноль.

Может вызвать ошибку `3` (деление на ноль) если второй операнд равен нулю.

Ошибка деления на ноль вызвана не будет если регистр `IF` (флаг прерываний) выставлен в 0.

Псевдокод:

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

5.1.36 038 FSHL

Мнемоника: `FSHL X`
Кодирование: `038 RM [Сегмент1] [Данные1]`

Производит арифметический сдвиг влево домножая заданное число на два. В результате может быть получено число с плавающей точкой:

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

MOV EAX,8
FSHR EAX //EAX = 16

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

Псевдокод:

```
X = X * 2
```

5.1.37 039 FSHR

Мнемоника: `FSHR X`
Кодирование: `039 RM [Сегмент1] [Данные1]`

Производит арифметический сдвиг вправо деля заданное число на два. В результате может быть получено число с плавающей точкой:

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

MOV EAX,8
FSHR EAX //EAX = 4

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

Псевдокод:

```
X = X / 2
```

5.1.38 040 RET

Мнемоника: `RET`
Кодирование: `040`

Возвращается из подпрограммы вызванной инструкцией `CALL`. Эта инструкция убирает с вершины стека адрес возврата, и записывает его в регистр `IP` (указатель на инструкцию).

Обычно используется для создания подпрограмм:

```
CALL SUBROUTINE0;
CALL SUBROUTINE1;

SUBROUTINE0:
  <...>
RET

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

Может вызвать ошибку `6` (переполнение/недополнение стека) если указатель стека выйдет за допустимые пределы.

Псевдокод:

```
IP = POP()
```

5.1.39 041 IRET

Мнемоника: `IRET`
Кодирование: `041`

Данная инструкция возвращается из выполнения обработчика прерывания. Она работает таким-же образом как и инструкция `RET`, только восстанавливает и сегмент кода`CS`, и указатель на инструкцию `IP`. Оба значения будут сняты со стека.

Восстановленное значение сегмента кода соответствует сегменту, в котором выполнялся код на момент возникновения прерывания. Данная инструкция не зависит от флага прерываний `IF`.

Например тело обработчика прерывания может быть вот таким:

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

Псевдокод:

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

5.1.40 042 STI

Мнемоника: `STI`
Кодирование: `042`

Устанавливает флаг прерываний `IF` в 1 *после следующей инструкции*. Это включит возможность обработки прерываний (без этого прерывания будут пропускаться).

Задержка в одну инструкцию создана для того, что-бы можно было скомбинировать инструкцию `STI` вместе с `IRET` или `EXTRET` для обеспечения корректного выхода из обработчика прерывания (что-бы прерывание не возникло до того, как будет выход из обработчика).

Например:

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

Данная инструкция привилегированна, то есть она может быть исполнена лишь когда текущий уровень доступа равен 0.

Псевдокод:

```
NextIF = 1
```

5.1.41 043 CLI

Мнемоника: `CLI`
Кодирование: `043`

Очищает флаг прерываний `IF`. Это предотвратит вызов любых прерываний, они будут просто игнорироваться. Обработку прерываний снова можно включить используя инструкцию `STI`.

Данная инструкция привилегированна, то есть она может быть исполнена лишь когда текущий уровень доступа равен 0.

Псевдокод:

```
IF = 0
```

5.1.42 047 RETF

Мнемоника: `RETF`
Кодирование: `047`

Производит дальний возврат из подпрограммы, которая была вызвана с помощью инструкции `CALLF`. Работает схожим образом с инструкцией `IRET`.

Значения текущего сегмента кода и указателя инструкций будут взяты с вершины стека.

Псевдокод:

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

5.1.43 048 STEF

Мнемоника: `STEF`
Кодирование: `048`

Включает расширенный режим процессора. В этом режиме включается поддержка таблицы прерываний и проверок связанных с защитой памяти для страниц.

Система страниц не зависит от работы расширенного режима, и может работать с выключенным защищённым режимом.

Возможно отключить расширенный режим с помощью инструкции `CLEF`.

Данная инструкция привилегированна, то есть она может быть исполнена лишь когда текущий уровень доступа равен 0.

Псевдокод:

```
EF = 1
```

5.1.44 049 CLEF

Мнемоника: `CLEF`
Кодирование: `049`

Выключает расширенный режим, который был включен инструкцией `STEF`. Это отключит таблицу прерываний и проверки прав доступа при обращении к страницам памяти.

Система страниц не зависит от работы расширенного режима, и может работать с выключенным защищённым режимом.

Данная инструкция привилегированна, то есть она может быть исполнена лишь когда текущий уровень доступа равен 0.

Псевдокод:

```
EF = 0
```

5.1.45 050 AND

Мнемоника: `AND X,Y`
Кодирование: `050 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Производит логическую операцию И. В первый операнд будет записана единица если оба операнда больше или равны единице, иначе будет записан ноль.

Псевдокод:

```
X = X AND Y
```

5.1.46 051 OR

Мнемоника: `OR X,Y`
Кодирование: `051 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Производит логическую операцию ИЛИ. В первый операнд будет записана единица если один из операндов больше или равен единице, иначе будет записан ноль.

Псевдокод:

```
X = X OR Y
```

5.1.47 052 XOR

Мнемоника: `XOR X,Y`
Кодирование: `052 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Производит логическую операцию исключающего ИЛИ. В первый операнд будет записана единица если один из операндов больше или равен единице, но только один, иначе будет записан ноль.

Псевдокод:

```
X = X XOR Y
```

5.1.48 053 FSIN

Мнемоника: `FSIN X,Y`

Кодирование: `053 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает синус второго операнда в первый операнд.

Псевдокод:

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

5.1.49 054 FCOS

Мнемоника: `FCOS X,Y`

Кодирование: `054 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает косинус второго операнда в первый операнд.

Псевдокод:

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

5.1.50 055 FTAN

Мнемоника: `FTAN X,Y`
Кодирование: `055 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает тангенс второго операнда в первый операнд.

Псевдокод:

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

5.1.51 056 FASIN

Мнемоника: `FASIN X,Y`

Кодирование: `056 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает арксинус второго операнда в первый операнд.

Псевдокод:

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

5.1.52 057 FACOS

Мнемоника: `FACOS X,Y`

Кодирование: `057 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает арккосинус второго операнда в первый операнд.

Псевдокод:

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

5.1.53 058 FATAN

Мнемоника: `FATAN X,Y`

Кодирование: `058 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Записывает арктангенс второго операнда в первый операнд.

Псевдокод:

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

5.1.54 059 MOD

Мнемоника: `MOD X,Y`
Кодирование: `059 RM [Сегмент1] [Сегмент2] [Данные1] [Данные2]`

Находит остаток от деления первого операнда на второй. Если на входе число с плавающей точкой, то записано будет число $X - n * Y$, где n это результат деления of X/Y округлённый в сторону нуля.

Псевдокод:

```
X = X FMOD Y
```

Глава 6

HL-ZASM

6.1 Описание

HL-ZASM - это высокоуровневый ассемблер и язык програмирования. Он предоставляет поддержку основных высокоуровневых структур C, но при этом является обратно-совместимым с кодом ассемблера ZASM2.

6.2 Ключевые слова

Некоторые символы и ключевые слова зарезервированы компилятором HL-ZASM и не могут быть использованы как имена переменных. Название переменной может состоять из латинских буков, цифер, нижнего подчёркивания (`_`) и, в некоторых случаях, точки (`.`) в названиях меток.

Вот список всех зарезервированых ключевых слов: `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`.

Описание назначения каждого ключевого слова описано в следующих главах.

6.3 Синтаксис ассемблера

6.3.1 Базовый синтаксис

HL-ZASM поддерживает вот такой синтаксис (Intel-подобный):

```
mov eax,123; //Записать константу в регистр
```

```
mov eax,ebx; //Перенести значение из регистра в регистр

add eax,123; //Инструкция с 2мя операндами
jmp 123; //Инструкция с 1им операндом (константой)
jmp eax; //Инструкция с 1им операндом (регистром)
```

За названием инструкции всегда следует один или два операнда. Если их два, то первый это всегда целевой операнд (в который будет записан результат), а второй это исходный операнд (из которого будет взято значение).

6.3.2 Доступ к памяти и сегменты

Для доступа к памяти можно использовать синтаксис ZASM (#) или более широкоиспользуемый синтаксис с квадратными скобками ([]). Оба синтаксиса являются правильными в HL-ZASM.

Процессор поддерживает сегментные префиксы для смещения адреса при обращении к памяти (см. стр 13 для более подробного обяснения). Как префикс можно использовать любой сегментный регистр или регистр общего назначения, и для любого операнда.

В HL-ZASM возможнен 'инвертированый' синтаксис, когда сегментный префикс следует за самим операндом (например 'eax:es' вместо 'es:eax').

Если используется обычный синтаксис для чтения значения из памяти (со скобками) то сегментный префикс должен находится внутри скобок. Вместо двоеточия (:) можно использовать знак прибавления (+), но только в обычном синтаксисе обращения к памяти.

Также можно использовать константу как префикс к регистру, но такой операнд на самом деле использует инвертированый синтаксис сегментного префикса. На данный момент невозможно использовать константный префикс к константе, а также использовать сегментный префикс к сегменту.

Пример поддерживаемого синтаксиса обращения к памяти:

```
//Синтаксис чтения из памяти ZASM2
mov eax,#100;
mov eax,#eax;
mov eax,es:#100;
mov eax,es:#eax;
mov eax,eax:#100;
//mov eax,eax:#es; //Неправильный синтаксис: невозможно использовать
                   //сегментный регистр как указатель
```

```
//Обычный синтаксис чтения из памяти
mov eax,[100];
mov eax,[eax];
mov eax,[es:eax];
mov eax,[es+eax];
//mov eax,[eax:es]; //Неправильный синтаксис: невозможно использовать
                    //сегментный регистр как указатель
//mov eax,[eax+es]; //Неправильный синтаксис: невозможно использовать
                    //сегментный регистр как указатель

//Использование констант с сегментным префиксом
mov eax,es:100;
mov eax,eax:100;
mov eax,100:es;
mov eax,100:eax;
//mov eax,100:200; //Неправильный синтаксис

//Использование регистров с сегментным префиксом
mov eax,ebx:ecx;
mov eax,es:ebx;
mov eax,ebx:es; //Инвертированый синтаксис
//mov eax,fs:es; //Неправильный синтаксис, такого операнда нет
```

6.3.3 Константы

Можно использовать константные выражения, шестнадцатеричные числа и значения меток в HL-ZASM:

```
//Целые числа
mov eax,255;   //EAX = 255
mov eax,0255;  //EAX = 255
mov eax,0x1FF; //EAX = 511

//Отрицательные числа
mov eax,-255;   //EAX = -255
mov eax,-0x1FF; //EAX = -511

//Числа с плавающей точкой и выражения
```

```
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;//Не парсится (ошибка в компиляторе)
mov eax,0xFF+100;     //EAX = 355

//Сложные выражения
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
```

6.3.4 Метки и переменные

Метки используются для создания возможности перехода между отдельными частями кода. Каждая метка это на самом деле число у которого есть название. При этом это число равно указателю в памяти в том месте, куда указывает метка. Все метки это просто константы, которые могут быть использованы в константных выражениях:

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

Макрос `define` позволяет привязать к названию метки любое константное значение:

```
define defVar1,120;
define defVar2,50+defVar1;
//define defVar3,labelname2*10; //Не парсится (ошибка компилятора)

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

Макрос `db` используется для записи данных в память, без обработки. Они будут записаны вместе с кодом. Данные, записаные таким образом могут быть использованы программой, но нужно быть осторожным, что-бы процессор не попытался выполнить эти данные как код:

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

Макрос `alloc` выделяет некоторое место, и позволяет обращаться к этому пространству за именем, если оно указано. Это место будет создано в текущем положении в памяти (если данные и код находятся в одном сегменте), или в отдельном сегменте данных (в зависимости от настроек компилятора). Первым параметром макроса должно быть либо название переменной, либо константное выражение

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

define allocsize,120;
alloc 0+allocsize; //должно быть выражением
```

Также существуют макросы для определения векторных переменных и текстовых строчек. Синтаксис для этого выражения `VECTOR_MACRO NAME,DEFAULT_VALUE`. Вот пример всех доступных векторных макросов:

```
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:
```

6.4 Expression Generator

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.

6.5 Declaring Variables

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.

6.6 Declaring Functions

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 page **??**. 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 page **??** 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
```

6.7 Function Calling

6.7.1 Calling Convention

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 **??**).

6.7.2 Stack Frame

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;
  }
}
```

6.7.3 Preserving Registers

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;
}
```

6.8 Control Structures

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];
```

6.9 Preprocessor

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).

6.9.1 C Runtime Library Macros

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 page ?? for more information on the ZCRT library.

6.9.2 Definition And Conditional Macros

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
```

6.9.3 File Inclusion Macros

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>
```

6.9.4 Special Compiler Commands

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 page **??**
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 page **??** for more information.

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

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

6.9.5 Preprocessor Definitions

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)

6.10 Advanced Features

6.10.1 Generating Libraries

no chapter

6.10.2 Optimizer

no chapter

6.10.3 ZCRT Library Reference

no chapter but lots of incredible fun

6.11 List Of Errors

6.11.1 General Compiler Errors

Undefined label

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

Variable redefined

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

Identifier expected

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

Expression expected, got ...

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

Ident ... is not a variable/pointer/array

Unable to get pointer of the given identifier.

Invalid instruction operand

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

Undefined opcode

Given opcode is not supported by the current architecture.

Array size must be constant

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

Cannot have expressions in global initializers

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

Can only zap/preserve registers inside functions/local blocks

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

... must be constant

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

Expected ... got ... instead

Invalid syntax is being used for some specific language structure.

Out of free registers

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.

Unable to include CRT library

Invalid runtime library name/library is not found.

Cannot open file

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

Internal error ...

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