Отладчик MDB для процессоров серии «Мультикор»

Руководство оператора

Версия 2.0

09.06.2010

Содержание.

1. Введение.

Отладчик **MDB (Multicore DeBugger) –** программа, обеспечивающая доступ к базовым аппаратным средствам отладки процессоров серии «Мультикор» (OnCD) через следующие интерфейсы:

- параллельный порт персонального компьютера (PC), работающий в режиме EPP (Enhanced Parallel Port);
- USB 2.0.

Базовые аппаратные средства отладки процессоров «Мультикор» имеют внешний JTAG-порт (IEEE 1149.1). Согласование между PC и JTAG-портом осуществляется через специализированный адаптер интерфейсов: JTAG-EPP или USB-JTAG .

Данный документ является руководством пользователя по применению отладчика для отладки программ. В документе приведена последовательность установки отладчика, режимы работы отладчика, система команд. В документе применяются следующие обозначения:

- шестнадцатеричные значения начинаются с префикса '0x', если префикс '0x' отсутствует, то значение считается десятичным;
- параметры команд отладчика, указанные в квадратных скобках, являются необязательными.

2. Установка отладчика.

Отладчик предназначен для работы в операционной системе Windows (Windows XP, Windows Vista, Windows 7).

При использовании параллельного порта компьютера его надо сконфигурировать в BIOS со следующими параметрами:

- BASE ADDRESS: 0x378
- MODE: EPP or EPP/ECP
- EPP version: 1.9

Также необходимо скопировать драйвер mcdbio.sys в директорию %windows%\system32\drivers.

При использовании адаптера USB-JTAG необходимо при подключенном адаптере запустить командный файл .\Driver\install.bat.

В настоящее время использование адаптера JTAG-EPP под ОС Windows возможно только с Windows XP.

3. Режимы работы.

Отладчик MDB может работать в следующих режимах:
- режим чтения команд из файла;
- интерактивный режим;
- режим ввода-вывода.

3.1 Режим чтения команд из файла.

Отладчик позволяет выполнять команды из файлов, при этом файлы команд могут содержать комментарии, пустые строки и вызовы команд из других файлов. Комментарии начинаются с символа '#' и продолжаются до конца строки. В одной строке файла должна находиться только одна команда. Пустые строки в файле отладчик интерпретирует как комментарии. Файл должен быть создан в текстовом формате.

3.2 Интерактивный режим.

В интерактивном режиме отладчик способен выполнять одну команду из командной строки. В интерактивном режиме работы для удобства пользователя реализованы следующие возможности:

1. Редактирование команд.

В интерактивном режиме отладчик использует библиотеку GNU readline и позволяет редактировать командную строку при помощи комбинаций клавиш.
Например:

Ctrl-w удаляет слово перед курсором.
Ctrl-a перемещает курсор в начало строки.
Ctrl-e перемещает курсор в конец строки.
Ctrl-r производит обратный поиск в истории команд (reverse incremental search).
Alt-f перемещает курсор на слово вперед.
Alt-b перемещает курсор на слово назад.

Более подробно о возможностях редактирования командной строки можно узнать из документации на библиотеку readline.

2. Command history.

Отладчик сохраняет историю команд и позволяет быстро вернуться к набранной ранее команде с помощью стрелки «вверх».

3. Tab-completion.

При нажатии на клавишу Tab, отладчик завершает набор частично набранной команды или параметра. При двойном нажатии на Tab, отображаются возможные варианты продолжения.

4. Command prediction.

Если команда набрана не полностью, то отладчик пытается дополнить команду до полного имени и потом запустить ее. Эта возможность позволяет выполнять команды, не набирая их полностью, ускоряя работу в интерактивном режиме и повышая её комфортность.

5. Command repeating.

Если осуществляется перевод строки, но команда не задана (пустая линия), то отладчик повторяет выполнение предыдущей команды. Эта особенность очень удобна при исполнении отлаживаемой программы по шагам и для вывода дампа памяти.

3.3 Режим потока ввода-вывода.

Отладчик также способен выполнять команды из стандартного потока ввода-вывода. Такие возможности отладчика позволяют использовать его как универсальный инструмент из других программ и надстроек. Такими программами могут являться интегрированная среда разработки или графический интерфейс для отладчика. Кроме того, отладчик может вызываться из языка-интерпретатора (например perl) и выполнять команды под его управлением, обеспечивая взаимодействие с нижним уровнем скрытое для пользователя. Использование отладчика из других программ позволяет реализовывать циклы, условное исполнение команд и автоматически анализировать полученные результаты. Такое применение очень удобно, например, для реализации наборов тестов. Пример программы, которая может вызывать отладчик и выполнять пользовательские процедуры, находится в приложении А.

4. Запуск отладчика.

Для запуска необходимо выполнить из командной строки команду:
./mdb.exe [ключ]

4.1 Ключи командной строки при запуске отладчика.

Отладчик распознает следующие ключи командной строки:

-h Помощь по ключам командной строки.

-e Включает режим expert mode (аналог команды «expert on» в интерактивном режиме.)

-l Краткий справочник по командам (аналог команды help в интерактивном режиме.)

-n Спрашивает пользователя о подключаемом устройстве.

При использовании ключа -n выводится запрос о подключаемом устройстве. Это полезно использовать, когда отладчик не может распознать подключаемое устройство или для проверки адаптера, когда никакое устройство не подключено.

Например, после выполнения команды: ./mdb.exe –n на экран выведется следующее:

Adapter version: 1

```
1: Elvis Multicore 12
2: Elvis Multicore 12 revision 1
3: unknown or not attached

Select attached device:
```

Для продолжения работы необходимо выбрать номер версии подключаемого процессора или, если устройство неизвестно или его нет, выбрать номер 3. При выборе пункта 3 многие команды отладчика отключаются.

-r Вход в режим отладки с установленным сигналом CPU reset.

Иногда процессор может находиться в состоянии зависания, при этом его невозможно перевести в состояние отладки без предварительного сброса. В таких случаях необходимо использовать ключ -r. При запуске отладчика с этим ключом выполняется следующая последовательность действий:

- инициализируется EPP (или USB) порт PC, адаптер и OnCD-модуль процессора,
- выставляется сигнал сброса процессора (CPU Reset),
- производится запрос режима отладки (DEBUG REQUEST),
- подключаются регистры OnCD (команда DEBUG ENABLE) и после перехода процессора в режим отладки снимается сигнал CPU Reset.
- сохраняется состояние процессора (сохраняются значения его регистров, кроме регистров, связанных с tlb).

Рекомендуется всегда использовать ключ -r при первом запуске отладчика.

-c cmd Выполнить одну команду и выйти из отладчика.

При использовании ключа -c отладчик выполняет заданную команду и завершается. Если команда содержит параметры, то необходимо использовать кавычки, например:

```
./mdb -c «clearmem 0xbfc00180,0x100»
```

-f file Выполнять команды из указанного файла.

При использовании ключа -f отладчик переходит в интерактивный режим и выполняет команды из указанного файла, выводя результат на экран. При использовании ключа -q, совместно с -f , на экран кроме ошибок ничего выводиться не будет. После завершения команд, отладчик остается в интерактивном режиме.

-q Quiet mode. В этом режиме отладчик ничего не выводит на экран кроме ошибок. Этот ключ может быть полезен при использовании совместно с -f.

При запуске отладчика можно применять несколько ключей, например:

```
./mdb.exe –r –e –f  example
```

где «example» - имя запускаемого файла.

-u Режим использования USB (по умолчанию используется параллельный порт).

При использовании ключей h, l, c, отладчик не переходит в интерактивный режим.

После запуска отладчик выводит версию адаптера, версию процессора, и распечатывает содержимое конвейера, затем выводит приглашение к вводу команд "mdb> ".

Например:

```
Adapter version: 1
Device: Elvis Multicore 12 revision 1
PCfetch: 0xbfc00000
PCdec:   0
PCexec:  0
PCmem: 0
IRdec:    0
mdb>
```

Далее пользователь вводит команды и отладчик выполняет их. После выполнения команды отладчик снова выводит prompt "mdb> " и ожидает ввода следующей команды.

5. Команды отладчика.

В этом разделе описываются команды отладчика, их формат и операции, которые они выполняют. Команды отладчика можно условно разделить на 3 группы:

- команды, которые

- **help [command]**

Выводит краткую справку по заданной команде или по всем командам, если команда не указана. Пример команды:

```
help tlb
```

- **? [command]**

Тоже самое, что и help.

- **run [addr]**

Переводит процессор в состояние исполнения кода (RUN), после чего производится контроль перехода процессора в режим отладки по заданным условиям, если перед запуском таковые были заданы, и после останова сохраняется состояние процессора. Сохранение состояния процессора производится с довыполнением в пошаговом режиме команд, находящихся в конвейере на момент останова, и заполнением конвейера NOP командами, и в дальнейшем для регистрации состояния процессора выполняются в пошаговом режиме требуемые для этого команды, заносимые в регистр инструкций, при PCfetch = 0xbfc00000.

Если адрес запуска не задан, то исполнение кода выполняется без изменения PCfetch с выполнением процедуры восстановления состояния процессора в пошаговом режиме, которое было в предыдущем останове (восстанавливаются значения регистров процессора, кроме регистров, связанных с tlb).

Все операции по сохранению и восстановлению состояния процессора скрыты от оператора.

Если адрес задан, то производится сброс CPU (reset), в PCfetch заносится адрес старта и процессор запускается на исполнение. Пример:

```
run 0xbfc01000
```

- **step [N] (N целое число от 1 до 65536)**

По данной команде процессор выполняет одну или N инструкций и возвращается в режим отладки. При большом N, команда может выполняться достаточно долго. Если N не указан, то выполняется одна инструкция. Перед запуском выполняется процедура восстановления состояния процессора. После останова выполняется процедура сохранения состояния процессора.

- **trace [N] (N целое число от 1 до 65536)**

По данной команде процессор выполняет одну или N инструкций и возвращается в режим отладки. Данная команда в отличие от команды «step» использует аппаратно реализованный счетчик исполненных инструкций. Команда «trace» завершается практически мгновенно независимо от заданного числа инструкций, в отличие от команды «step N», так как процедуры восстановления и сохранения состояния процессора выполнятся 1 раз, а при команде step N эти процедуры выполнятся N раз. Перед запуском выполняется процедура восстановления состояния CPU. После останова выполняется процедура сохранения состояния CPU.

- **wait [sec]**

Ожидает останова процессора. Если процессор не перешел в режим отладки в течение заданного интервала времени, например, в заданной точке останова, то отладчик принудительно запрашивает режим отладки. Кроме того пользователь может нажать Ctrl-C и тогда отладчик немедленно запросит переход в режим отладки.

После перехода процессора в режим отладки, отладчик выполняет процедуры считывания состояния процессора, аналогичные командам run, trace, step.

- **sleep [sec]**

Пауза. Эта команда может быть полезна для использования в командных файлах. Пауза может быть прервана по Ctrl-C.

- **reset [cpu|oncd|epp|dsp]**

cpu - Сброс CPU. Очищается PC конвейер. PCfetch = 0xbfc00000. Процессор остается в режиме отладки. Сохраняется состояние процессора.

oncd - сброс модуля OnCD (On Chip Debug).

epp - сброс EPP порта PC и инициализация EPP_JTAG адаптера.

dsp – сброс DSP.

Если параметр не указан, то подразумевается «cpu».

- **set regname value**

Устанавливает регистр 'regname' в заданное значение. Regname может быть
регистром OnCD (oscr, pcfetch, omar, …);
регистром CPU (r1, r2, …, r31) – см. Примечания;
регистром системного сопроцессора (cp0.status, cp0.cause, …, cp0.config1, cp0.regno.sel) – см. Примечания;
vaddr - виртуальным адресом 32-х битной ячейки памяти. Адрес должен быть выровнен на границу 4-х байт.

Во время установки регистра CPU (r1, r2… r31, PCfetch, IRdec) или CP0 значение этого регистра заносится в буфер (в ранее сохраненное состояние процессора), реально это значение занесется в регистр непосредственно только перед выполнением хотя бы одного шага (т.е. происходит восстановление состояния процессора перед шагом из этого буфера).

Но регистры OnCD или память прописываются сразу после выполнения команды set regname value, причем во время обращения в память считываются необходимые

регистры CP0 для вычисления физического адреса. При некорректно заданном адресе выдается ошибка. Пример команды:

```
set 0xbfc00010 0x12344321
```

- **show regname1 regname2 ...**

Отображает содержимое перечисленных регистров. Regname может принимать такие же значения, как и в команде 'set'. Значения регистров CPU или CP0 считываются из буфера, в котором хранится состояние процессора, т.е. при этом пошаговый режим не включается. При чтении регистров OnCD так же не включается пошаговый режим, но при чтении ячейки памяти, происходит считывание некоторых регистров CP0 для вычисления физического адреса, т.е. включается пошаговый режим. При некорректно заданном адресе выдается ошибка. Пример команды:

```
show 0xbfc00010
```

- **display ["?" | regname1 regname2 …]**

Команда display устанавливает список регистров для автоматического отображения отладчиком после каждого останова процессора. Regname трактуется так же как и в командах set/show.

При указании ключа '?' команда display выводит текущий список регистров. Пример команды:

```
display Pcfetch IRdec cp0.config 0xbfc00100 r1 r2
```

- **catch ["?" | vector1 vector2 …]**

Команда catch устанавливает векторы исключений процессора в Exception Catch Register. При возникновении исключения с запрошенным вектором процессор переходит в режим отладки и сохраняется состояние процессора.

Параметр vector может принимать следующие значения:

0xbfc00000,
0x80000000,
0x80000180,
0x80000200,
0xbfc00200,
0xbfc00380,
0xbfc00400.

С ключом '?' команда catch выводит текущий список запрошенных векторов. Пример команды:

```
catch 0xbfc00000 0x80000200 0xbfc00380
```

- **tlb [flush|probe|write|read]**

Команда tlb позволяет производить сброс, инициализацию, чтение и поиск в tlb. Сначала в пошаговом режиме, скртым от оператора, считывается регистр CP0.config, проверяется бит FM. Если tlb включен (FM = 0), то выполняются следующие команды:

tlb read [index] – выводит все содержимое tlb. Если надо вывести только одну строку tlb, то в конце команды поставьте ее номер (index).

tlb flush – очищает все строки в tlb.

tlb probe vmaddr [asid] – команда показывает, есть ли такая трансляция для виртуального адреса (vmaddr), если есть, то выводит эту запись.

tlb write Index Size VPN2 G ASID PFN0 C0 D0 V0 PFN1 C1 D1 V1 – формат команды для записи строки в tlb.

Пример вывода результатов команд tlb flush и tlb read:

```
mdb> tlb flush
16 entries cleared
mdb> tlb read
MMU Type: Standard TLB, 16 entries
Index Size    VPN2        G ASID    PFN0        C0 D0 V0    PFN1        C1 D1 V1
--------------------------------------------------------------------------------
--------------
    0     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    1     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    2     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    3     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    4     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    5     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    6     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    7     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    8     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
    9     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
   10     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
   11     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
   12     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
   13     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
   14     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
   15     4KB 8000000000  n  0x00 0000000000  3    n    n  0000000000   3   n    n
```

Во время чтения или записи в tlb отладчик сразу же в пошаговом режиме, скрытом от оператора, производит чтение или запись регистров CP0, связанных с tlb (состояние конвейера процессора постоянно будет изменяться, после выполнения команд tlb). Примеры команд:

```
tlb write 3 1mb 0xff000000 y 0x21 0x1fc00000 2 n n 0x1fd00000 2 n n
tlb probe 0xff001000 0x21
```

- **halt**

Выполняет останов процессора (переводит процессор в режим отладки). Сохраняет состояние процессора.

- **source filename**

Исполняет команды из указанного файла (отладчик переходит в режим чтения команд из файла).

- **load filename**

Загружает программу в память из текстового файла. Адрес памяти берется из файла. Пример содержимого файла:

```
bfc06538 bfc00074 bfc00080 bfc00ff0     //bfc00060
1000ffec 00000000                       //bfc00090
```

Во время загрузки ведется анализ адреса памяти, куда идет запись так же как и для команды set vaddr value. Во время загрузки проверяется ее корректность. Загрузка файла происходит поэтапно по 4 слова, сначала 4 слова грузятся, потом считываются и сравниваются с первоначальными четырьмя, затем следующие 4 и т.д. При несовпадении выдается ошибка. Пример команды:

```
load program.txt
```

- **save filename [address [size]]**

Сохраняет содержимое памяти в формате совместимом с командой load. Пример команды:

```
save file.txt 0xbfc00000,0x1000
```

- **loadbin filename [address]**

Загружает содержимое файла filename в память по адресу address. Если параметр address не указан, то используется адрес 0xbfc00000 (состояние PC после выполнения reset). Проверка на корректность ведется как и при команде load.

- **savebin filename [address [size]]**

Сохраняет size байт памяти с адреса address в файл filename. Если параметр size не задан, то сохраняется вся память с адреса address и до границы 0xbfc00000 + 128 * 1024. Если address и size не заданы, то сохраняется вся память (address = 0xbfc00000, size = 128 * 1024).

- **loadelf filename**

Загружает содержимое ELF файла в память. Записывает значение entry point в PCfetch. После успешного выполнения этой команды отладчик позволяет использовать имена символов из ELF файла в командах show/set/bp/wp.

- **dump [address [size]]**

Выводит шестнадцатеричный дамп памяти с адреса address. Если address не задан, то дамп выводится с адреса следующего за последним распечатанным словом при предыдущем выполнении этой команды. size - размер в байтах, должен быть кратен 4. Если параметр size не задан, то используется значение size от предыдущей команды. По умолчанию значение size равно 256. Команда dump так же как и команда show vaddr использует пошаговый режим для вычисления физического адреса. Пример команды:

```
dump 0xbfc00000 0x200
```

- **clearmem [address [size]]**

Обнуляет size байт памяти с адреса address. Если параметры не указаны, то обнуляются 128kb с адреса 0xbfc00000. Так же как и для команды set vaddr вычисляется физический адрес. Пример команды:

```
Clearmem 0xbfc00000 0x1000
```

- **bp # vm_addr|clear [match=ne|eq|gt|lt] [count=N] [both|any]**

bp - break point. Устанавливает точку останова по адресу PC.
- номер точки останова, может быть 0 или 1.
vm_addr - адрес останова.
clear - команда выключения точки останова.
Значение match определяет диапазон адресов срабатывания для точки останова.
ne - not equal (не равно)
eq - equal (равно)
gt - great than (больше чем)
lt - less than (меньше чем)

Значение count определяет сколько точек останова должно быть пропущено, прежде чем производить останов. Если указан флаг both, то значение count уменьшается при одновременном срабатывании обоих точек останова (флаг необходимо устанавливать, если задается диапазон адресов). Если указан флаг any, то count декрементируется при совпадении адреса с любой из точек останова. Например:

`bp 0 0xbfc00100 mattch=eq count=0 any` – устанавливает точку останова по PC=0xbfc00100. Когда произойдет останов процессора, этот адрес окажется в конвейере на стадии декодирования. Если заказанный адрес является адресом Delay Slot, то останов произойдет, когда он уже будет в конвейере на стадии исполнения. Для задания диапазона адресов останова надо использовать два break point, например:

```
bp 0 0xbfc00200 match=gt count=0 both
bp 1 0xbfc00300 match=lt count=10 both
```

После выполнения этих 2-х команд, останов произойдет, после того как PC 10 раз попадет в диапазон указанных адресов. В count будет записано последнее значение, т.е. 10.

`bp 0 clear` – выключает первую точку останова.

- **wp # vm_addr|clear [access=r|w|rw] [match=ne|eq|gt|lt] [count=N] [both|any]**

wp - watch point. Устанавливает точку останова по обращению в память.

- номер точки останова, может быть 0 или 1.
vm_addr - адрес останова (физический?).
clear - команда выключения точки останова.
Значение access определяет режим доступа для которого определяется точка останова.
r - read (чтение)
w - write (запись)
rw - read/write (чтение и запись)

Значение match определяет диапазон адресов срабатывания для точки останова.
ne - not equal (не равно)
eq - equal (равно)
gt - great than (больше чем)
lt - less than (меньше чем)
`wp 0 0xbfc00100 access=rw mattch=eq count=0 any` – устанавливает watch point по доступу к памяти. Останов произойдет, если будет обращение в память по записи или чтению, по указанному адресу. Команда, отработавшая обращение к памяти будет на стадии memory. Если в момент останова на стадии декодирования в PC конвейере будет находиться delay slot, то останов произойдет еще на одну стадию позже.

Значение count определяет сколько точек останова должно быть пропущено, прежде чем производить останов. Если указан флаг both, то значение count уменьшается при одновременном срабатывании обоих точек останова (флаг необходимо устанавливать, если задается диапазон адресов). Если указан флаг any, то count декрементируется при совпадении адреса с любой из точек останова.

Команды bp и wp используют одни и те же аппаратные ресурсы. Нельзя одновременно задать break point и watch point с одинаковым номером. Кроме того значение count и значение флага both/any является общим для обеих точек останова.

- **expert [on|off]**

Включить/Выключить расширенный режим (expert mode). В этом режиме доступны низкоуровневые команды полезные для отладки процессора. После выполнения этих команд состояние процессора не снимается, поэтому конвейер не портится. При чередовании команд expert mode и не expert mode результат непредсказуем. Использовать данный режим необходимо с крайней аккуратностью. Рекомендуется при использовании расширенного режима прописать в регистр OnCD OSCR значение 0x220, выполнив следующую команду:

```
eset OSCR 0x220
```

Expert mode включает следующие команды:
epp
jtag
eset
eshow
estep
erun
etrace
pipeline

- **pipeline**

Выводит текущее состояние PC конвейера. Эта команда работает только в экспертном режиме.

Для вывода значений конвейера на экран пользуйтесь командой display.

```
epp read data|address
```

Прочитать 1 байт из EPP порта. Пример команды:

```
epp read data
epp write data|address=value
```

Записать 1 байт в EPP порт. Пример команды:

```
epp write address=0x25
```

- **jtag command**

Записывает команду в IR регистр JTAG порта и выводит предыдущее содержимое IR регистра.

Возможные команды:
debug_request
debug_enable
extest
sample
bypass
Пример команды:

```
jtag debug_enable
```

- **eset regname value**

Устанавливает регистр 'regname' в заданное значение, regname может быть:

- регистром OnCD (oscr, pcfetch, omar, ...);
- физическим адресом памяти;

Пример команды:

```
eset 0x1fc00100=0x12344321
```

- **eshow regname1 regname2 ...**

Отображает содержимое заданных регистров, regname может быть:

- регистром OnCD (oscr, pcfetch, omar, ...);
- физическим адресом памяти;

Пример команды:

```
eshow 0x1fc00100
```

- **estep [-f] [-p]**

Делает один шаг, т.е. продвигает конвейер PC на одну стадию, выполняя следующую последовательность действий:

1. Считывается OSCR.
2. Если установлен флаг '-f', то в OSCR записывается флаг сброса конвейера.
3. Если установлен флаг ‘-p’, то в IRd устанавливается флаг db_push.
4. Выдается команда запуска CPU на один шаг (инструкцию).
5. Печатает состояние конвейера. Состояние конвейера не портится.

- **erun [-f] [-p]**

1. Считывается OSCR.
2. Если установлен флаг '-f', то в OSCR записывается флаг сброса конвейера.
3. Если установлен флаг ‘-p’, то в IRd устанавливается флаг db_push.
4. Выдается команда запуска CPU.
5. Отладчик сканирует OSCR и ожидает режим отладки. Если нажать Ctrl-C, то отладчик выдает DEBUG_REQUEST, сканирует IR, выдает DEBUG_ENABLE.
6. Печатает состояние конвейера. Состояние конвейера не портится.

- **etrace [-f] [-p] [N]**

1. Устанавливается OTC = N - 1. Допустимое значение N: 0 - 0x10000.
2. В OSCR записывается флаг trace enable(TME).

Если установлен флаг '-f', то в OSCR также записывается флаг сброса конвейера (MPE).

3. Если установлен флаг '-p', то в IRd устанавливается флаг db_push.
4. Выдается команда запуска CPU.
5. Отладчик сканирует OSCR и ожидает режим отладки. Если нажать Ctrl-C, то отладчик выдает DEBUG_REQUEST, сканирует IR, выдает DEBUG_ENABLE.
6. Печатает состояние конвейера. Состояние конвейера не портится.

- **drun [addr]**

Запускает DSP с указанного адреса.

- **dstep [N]**

Исполняет одну или N DSP инструкций.

Команды drun, dstep так же как и команды run и step снимают и восстанавливают состояние процессора.

- **dbp addr**

Устанавливает точку останова в DSP. Возможна только одна точка останова.

- **exit**

Запуск процессора и выход из отладчика. Полностью аналогична команде **run**, но вместо ожидания останова процессора, выходит из отладчика.

- **quit**

Выход из отладчика.

6. Встроенные функции.

Для отладки процессора или проверки его работоспособности некоторых блоков предусмотрен ряд тестовых функций:

- **testmem [vaddr [size [passno]]]**

vaddr - адрес начала тестирования памяти.

Тестирует size памяти.

passno – количество повторов теста

Эта команда создает массив, заполненный случайными данными, равный объему size. Прописывает этот массив в память, потом считывает память и запоминает считанное. После сравнения изначального массива с полученным выдает сообщение Test passed, если массивы равны, или Test fall (тест заканчивается, независимо от passno), выводя при этом 10 первых ячеек памяти, в которых были обнаружены ошибки (0xbfc01000 writen 0xdf34c234 readed 0x00000000 – пример ошибки), и общее количество ошибок. Если указан passno, то тест повторяется указанное число раз, причем каждый раз заново создается массив с произвольным кодом.

- **memwrite [vaddr [size [pattern]]]**

pattern – код, прописываемый в память

Команда прописывает size памяти кодом pattern, начиная с адреса vaddr.

- **testoncd [passno]**

Записывает и считывает регистры OnCD доступные для записи. Сообщает об ошибках.

- **testir**

Команда в цикле подает debug_request и выводит предыдущее значение регистра команд IR OnCD. Выход из цикла по Ctrl-C.

7. Примечания.

При доступе к памяти, отладчик производит преобразование адресов из виртуальных в физические. Поэтому пользователь должен использовать виртуальные адреса.

Команды run, step, trace, wait, halt сохраняют состояние CPU после его останова. При этом нарушается содержимое регистров общего назначения и конвейера CPU. По этой

причине нельзя смешивать обычные команды и команды расширенного режима (expert mode). Надо быть очень внимательным при переходе от команд расширенного режима к обычным и наоборот. При запуске CPU командами run, step, trace отладчик производит корректное восстановление состояния CPU. Если в программе-обработчике исключений команда в delay slot вызывает исключение, то при отладке программы по шагам будет происходить возврат на начало обработчика исключения сразу же на следующем шаге.

При входе в отладчик состояние процессора не изменяется. Это позволяет пользователю выйти из отладчика и войти в него снова, чтобы продолжить отладку. Тем не менее, пользоваться этой возможностью нужно с большой осторожностью. Если процессор был остановлен после команд run, step, trace, wait, halt, то его состояние нарушено процедурами сохранения состояния. При выполнении команд run, step, trace отладчик восстанавливает состояние процессора. Но если пользователь вышел из отладчика и вошел снова, то отладчик не сможет корректно восстановить состояние CPU.

Большинство команд отладчика могут быть прерваны по Ctrl-C. Если команда в это время ожидает останова CPU, то отладчик запрашивает режим отладки(DEBUG_REQUEST) и после останова CPU выполняет действия предусмотренные исполняемой командой.

Если команда была введена с ошибками, то отладчик выведет формат этой команды, которого необходимо придерживаться.

Список регистров процессора, который отображает отладчик:

cp0.20	dsp.ac1.0	dsp.r0.0	dsp.r2.3	dsp.r31.2	qstr
cp0.21	dsp.ac1.1	dsp.r0.1	dsp.r20.0	dsp.r31.3	r0
cp0.22	dsp.ac1.2	dsp.r0.2	dsp.r20.1	dsp.r4.0	r1
cp0.7	dsp.ac1.3	dsp.r0.3	dsp.r20.2	dsp.r4.1	r10
cp0.badvaddr	dsp.at0	dsp.r1.0	dsp.r20.3	dsp.r4.2	r11
cp0.cacheerr	dsp.at1	dsp.r1.1	dsp.r21.0	dsp.r4.3	r12
cp0.cause	dsp.at2	dsp.r1.2	dsp.r21.1	dsp.r5.0	r13
cp0.compare	dsp.at3	dsp.r1.3	dsp.r21.2	dsp.r5.1	r14
cp0.config	dsp.ccr.0	dsp.r10.0	dsp.r21.3	dsp.r5.2	r15
cp0.config0	dsp.ccr.1	dsp.r10.1	dsp.r22.0	dsp.r5.3	r16
cp0.config1	dsp.ccr.2	dsp.r10.2	dsp.r22.1	dsp.r6.0	r17
cp0.config2	dsp.ccr.3	dsp.r10.3	dsp.r22.2	dsp.r6.1	r18
cp0.config3	dsp.cntr	dsp.r11.0	dsp.r22.3	dsp.r6.2	r19
cp0.context	dsp.csh	dsp.r11.1	dsp.r23.0	dsp.r6.3	r2
cp0.count	dsp.csl	dsp.r11.2	dsp.r23.1	dsp.r7.0	r20
cp0.datahi	dsp.dcsr	dsp.r11.3	dsp.r23.2	dsp.r7.1	r21
cp0.datalo	dsp.dt0	dsp.r12.0	dsp.r23.3	dsp.r7.2	r22
cp0.debug	dsp.dt1	dsp.r12.1	dsp.r24.0	dsp.r7.3	r23
cp0.depc	dsp.dt2	dsp.r12.2	dsp.r24.1	dsp.r8.0	r24
cp0.desave	dsp.dt3	dsp.r12.3	dsp.r24.2	dsp.r8.1	r25
cp0.entryhi	dsp.i0	dsp.r13.0	dsp.r24.3	dsp.r8.2	r26
cp0.entrylo0	dsp.i1	dsp.r13.1	dsp.r25.0	dsp.r8.3	r27
cp0.entrylo1	dsp.i2	dsp.r13.2	dsp.r25.1	dsp.r9.0	r28
cp0.epc	dsp.i3	dsp.r13.3	dsp.r25.2	dsp.r9.1	r29
cp0.errctl	dsp.i4	dsp.r14.0	dsp.r25.3	dsp.r9.2	r3
cp0.errorepc	dsp.i5	dsp.r14.1	dsp.r26.0	dsp.r9.3	r30
cp0.index	dsp.i6	dsp.r14.2	dsp.r26.1	dsp.sar	r31
cp0.lladdr	dsp.i7	dsp.r14.3	dsp.r26.2	dsp.sp	r4
cp0.pagemask	dsp.it0	dsp.r15.0	dsp.r26.3	dsp.sr	r5
cp0.perfcnt	dsp.it1	dsp.r15.1	dsp.r27.0	dsp.ss	r6
cp0.prid	dsp.it2	dsp.r15.2	dsp.r27.1	hi	r7
cp0.random	dsp.it3	dsp.r15.3	dsp.r27.2	IRdec	r8
cp0.status	dsp.la	dsp.r16.0	dsp.r27.3	itcount	r9
cp0.taghi	dsp.lc	dsp.r16.1	dsp.r28.0	itcsr	rtcount

cp0.taglo	dsp.m0	dsp.r16.2	dsp.r28.1	itperiod	rtcsr
cp0.watchhi	dsp.m1	dsp.r16.3	dsp.r28.2	itscale	rtperiod
cp0.watchlo	dsp.m2	dsp.r17.0	dsp.r28.3	lo	uart.ier
cp0.wired	dsp.m3	dsp.r17.1	dsp.r29.0	maskr	uart.iir
csr	dsp.m4	dsp.r17.2	dsp.r29.1	OBCR	uart.lcr
dsp.a0	dsp.m5	dsp.r17.3	dsp.r29.2	OMAR	uart.lsr
dsp.a1	dsp.m6	dsp.r18.0	dsp.r29.3	OMBC	uart.mcr
dsp.a2	dsp.m7	dsp.r18.1	dsp.r3.0	OMDR	uart.msr
dsp.a3	dsp.mt0	dsp.r18.2	dsp.r3.1	OMLR0	uart.rbr
dsp.a4	dsp.mt1	dsp.r18.3	dsp.r3.2	OMLR1	uart.thr
dsp.a5	dsp.mt2	dsp.r19.0	dsp.r3.3	OSCR	wtcount
dsp.a6	dsp.mt3	dsp.r19.1	dsp.r30.0	OTC	wtcsr
dsp.a7	dsp.pc	dsp.r19.2	dsp.r30.1	PCdec	wtperiod
dsp.ac0.0	dsp.pdnr.0	dsp.r19.3	dsp.r30.2	PCexec	wtscale
dsp.ac0.1	dsp.pdnr.1	dsp.r2.0	dsp.r30.3	PCfetch	
dsp.ac0.2	dsp.pdnr.2	dsp.r2.1	dsp.r31.0	PCmem	
dsp.ac0.3	dsp.pdnr.3	dsp.r2.2	dsp.r31.1	PCwb	

Приложение А. Пример программы, вызывающей отладчик.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void wait_prompt(void);
void cmd_exec(const char *);
void source(const char *);
void print_out(void);
void mdb_test(void);

#define MDB_PATH "./mdb"
#define MAX_LEN 256
#define MAX_LINES 1024

FILE *rpipe, *wpipe;
int outc;
char *outv[MAX_LINES];
int
main(int argc, char **argv)
{     int wpipefd[2], rpipefd[2];
      pid_t mdb_pid;
      int mdb_status;
      char *mdb_argv[] = { "mdb", "-r", NULL };

      --argc;
      ++argv;


      if (pipe(rpipefd) != 0) {
            perror("pipe");
            exit(1);
      }

      if (pipe(wpipefd) != 0) {
            perror("pipe");
            exit(1);
      }

      mdb_pid = fork();

      switch (mdb_pid) {
      case 0:
            /* Child process */
            close(wpipefd[1]);      /* close write fd */
```

```
                close(rpipefd[0]);      /* close read fd */
                close(0);               /* close stdin */
                dup2(wpipefd[0], 0);    /* assotiate stdin with pipe */
                close(wpipefd[0]);      /* close duplicated fd */
                close(1);               /* close stdout fd */
                dup2(rpipefd[1], 1);    /* assotiate stdout with pipe */
                close(rpipefd[1]);      /* close duplicated fd */

                execvp(MDB_PATH, mdb_argv);
                perror("execvp");
                exit(1);
                /* NOTREACHED */
        case -1:
                perror("fork");
                exit(1);
                /* NOTREACHED */
        default:
                /* Parent process */
                close(wpipefd[0]);      /* close read fd */
                close(rpipefd[1]);      /* close write fd */

                rpipe = fdopen(rpipefd[0], "r");        /* Read pipe end */
                if (rpipe == NULL) {
                        perror("fdopen");
                        exit(1);
                }
                wpipe = fdopen(wpipefd[1], "w");    /* Write pipe end */
                if (wpipe == NULL) {
                        perror("fdopen");
                        exit(1);
                }
                /* Set line buffered mode for wpipe */
                setvbuf(wpipe, NULL, _IOLBF, 0);

                mdb_test();

                close(rpipefd[0]);      /* close pipe */
                close(wpipefd[1]);      /* close pipe */
                wait(&mdb_status);
        }
}

void
wait_prompt()
{   int c, len;
    char line[MAX_LEN];

    for (; outc > 0;) {
            free(outv[--outc]);
            outv[outc] = NULL;
```

```
	}
	line[0] = '\0';
	len = 0;
	while((c = getc(rpipe)) != EOF) {
		if (len + 2 < MAX_LEN) {
			line[len++] = (char)c;
			line[len] = '\0';
		} else {
			fprintf(stderr, "Line too long: %s\n", line);
			exit(1);
		}
		if (c == '\n') {
			if (outc < MAX_LINES) {
				outv[outc] = strdup(line);
				if (outv[outc] == NULL) {
					perror("strdup");
					exit(1);
				}
				outc++;
			} else {
				fprintf(stderr, "Too many output lines\n");
				exit(1);
			}
			line[0] = '\0';
			len = 0;
		}
		if (!strcmp(line, "mdb> "))
			return;
	}
}

void
source(const char* filename)
{	FILE *fp;
	char line[MAX_LEN], *p;

	fp = fopen(filename, "r");
	if (fp == NULL) {
		perror("fopen");
		exit(1);
	}
	while (fgets(line, sizeof(line), fp) != NULL) {
		if ((p = strchr(line, '\n')) == NULL) {
			fprintf(stderr, "Input line too long.\n");
			exit(1);
		} else
			*p = '\0';
		cmd_exec(line);
	}
	if (ferror(fp)) {
```

```
                perror("fgets");
                exit(1);
        }
        fclose(fp);
        return;
}

void
cmd_exec(const char *cmd)
{       int c, len;
        char line[MAX_LEN];
        char ncmd[MAX_LEN];

        if (snprintf(ncmd, sizeof(ncmd), "%s\n", cmd) >= sizeof(ncmd)) {
                fprintf(stderr, "Command too long: %s\n", cmd);
                exit(1);
        }
        fprintf(wpipe, "%s", ncmd);             /* Supply command to mdb */

        for (; outc > 0;) {
                free(outv[--outc]);
                outv[outc] = NULL;
        }
        line[0] = '\0';
        len = 0;
        while((c = getc(rpipe)) != EOF) {
                if (len + 2 < sizeof(line)) {
                        line[len++] = (char)c;
                        line[len] = '\0';
                } else {
                        fprintf(stderr, "Line too long: %s\n", line);
                        exit(1);
                }
                if (c == '\n') {
                        if (strcmp(line, ncmd)) {
                                if (outc < MAX_LINES) {
                                        outv[outc] = strdup(line);
                                        if (outv[outc] == NULL) {
                                                perror("strdup");
                                                exit(1);
                                        }
                                        outc++;
                                } else {
                                        fprintf(stderr,
                                            "Too many output lines\n");
                                        exit(1);
                                }
                        }
                        line[0] = '\0';
                        len = 0;
```

```
                }
                if (!strcmp(line, "mdb> "))
                        return;
        }
}

void
print_out()
{       int i;
        for (i = 0; i < outc; i++) {
                printf("%s", outv[i]);
        }
}

void
mdb_test()
{процедуры пользователя}
```