Окончание. Начало в № 8'2002

Микроконтроллер для встроенного применения – NIOS. Система команд и команды, определяемые пользователем.

Часть II. Команды перехода, исключения, конвейер и команды, определяемые пользователем

Иосиф Каршенбойм

Ik@lmail.loniis.spb.su

Команды управления выполнением программы

К командам управления выполнением программы относятся команды перехода и обработки исключений:

- две команды относительного перехода (BR и BSR);
- две команды абсолютного перехода (JMP и CALL);
- две команды обработки исключений (TRET и TRAP);
- пять команд условных переходов (SKPs, SKP0, SKP1, SKPRz и SKPRnz).

Команды относительного перехода

В Nios используется две команды относительного перехода — BR и BSR. Адрес перехода вычисляется по текущему счетчику программы (то есть непосредственно от адреса команды BR) и полю команды IMM11. Работа команды BSR отличается от BR тем, что адрес возврата сохраняется в %o7. Обе команды выполняются без условий. Если необходимо выполнение условных переходов, то команде BR или BSR должна предшествовать команда типа SKP. Обе команды (BR и BSR) имеют слот задержки перехода: команда, следующая сразу после BR или BSR, выполняется после BR или BSR, но перед выполнением команды перехода в новый адрес (см. ниже «Слоты задержки перехода»). Диапазон перехода команд BR и BSR: вперед до 2048 байт или назад до 2046 байт относительно адреса команды BR или BSR.

Команды абсолютного перехода

В Nios используется две команды абсолютного перехода — JMP и CALL. Адрес перехода задается содержанием регистра общего назначения. Содержание регистра сдвигается влево на один бит и передается в PC. Команда CALL отличается от JMP тем, что адрес возврата сохраняется в %o7. Обе команды выполняются без условий. Если необходимо выполнение условных переходов, то командам JMP или CALL должна предшествовать команда типа SKP.

Обе команды JMP и CALL имеют слот задержки перехода: команда, следующая сразу после JMP и CALL, выполняется после JMP и CALL, но перед выполнением команды перехода в новый адрес.

Псевдокоманда LRET, которая является псевдонимом ассемблера для JMP %o7, традиционно используется для возврата из подпрограммы.

Команды «TRAP»

Процессор Nios имеет две команды для программной обработки исключения: TRAP и TRET. В отличие от JMP и CALL, ни TRAP, ни TRET не имеют слота задержки перехода: команда, следующая сразу после TRAP, не выполняется до окончания работы драйвера исключения. Команда, следующая за TRET, вообще не выполняется как часть операции TRET.

Условные команды

Есть пять условных команд (SKPs, SKP0, SKP1, SKPRz, и SKPRnz). Каждая из этих команд имеет обратную команду ассемблера (IFs, IF0, IF1, IFRz, и IFRnz соответственно). Каждая из этих команд проверяет внутреннее состояние CPU и затем выполняет следующую команду или не выполняет (в зависимости от результата). Работа всех пяти команд типа SKP (и их псевдонимов) идентична и отличается только условием. В каждом случае следующая за условной команда выбирается из памяти независимо от результата проверки условия. Но, в зависимости от результата проверки условия, она будет выполнена или отменена.

Несмотря на то что SKP и команды условного выражения типа IF часто используются совместно с последующей командой перехода (JMP, CALL) или с командой ветвления (BR, BSR), чтобы выполнить действие по условию, они могут использоваться и с любой другой командой. Последующее использование команды PFX (PFX немедленно после команды SKPx или IFx) представляет специальный случай, когда следующие две команды либо будут отменены, либо выполнены. Пары команд, в которых присутствуют PFX и команды, выполняемые по условию, можно назвать «атомный модуль».

Исключения

Процессор Nios позволяет иметь до 64 векторных исключений. Исключения могут быть либо глобально разрешены, либо глобально заблокированы управляющим битом IE в регистре STATUS, либо выборочно разрешены в соответствии с разрешенным приоритетом IPRI в регистре STATUS. Исключения генерируются любым из трех источников: внешние аппаратные прерывания, внутренние исключения или программные команды TRAP.

Модель обработки исключения Nios позволяет производить точную обработку всех внутренних исключений, то есть механизм передачи исключения оставляет подпрограмму обработки исключения с достаточной информацией, чтобы восстановить состояние

прерванной программы. Внутренние исключения генерируются, если команда SAVE или RESTORE вызывает выход за пределы окна регистрового файла.

Подпрограммы обработки исключений всегда выполняются в открытом окне регистра, позволяя иметь очень малое время ожидания обработки прерывания. Драйвер исключения не требует сохранять содержание регистра статуса при прерываниях вручную.

Таблица векторов исключений

Таблица векторов исключений — это набор 64 адресов драйверов исключения. Для Nios CPU-32 каждый вход — это 4 байт, а для Nios CPU-16 каждый вход — 2 байт. Базовый адрес таблицы векторов исключений может иметь перестраиваемую конфигурацию.

Когда Nios CPU обрабатывает номер исключения n, то он выбирает n-й вход из таблицы векторов исключений, удваивает выбранное значение и затем загружает результаты в PC.

Таблица векторов исключений может физически постоянно находиться в оперативной памяти или в ROM, в зависимости от аппаратной карты памяти целевой системы. Таблица векторов исключений в ROM не будет требовать инициализации во время выполнения.

Внешний источник аппаратного прерывания

Внешний источник может запрашивать аппаратное прерывание, передавая на вводы irq_number 6-битовый номер прерывания и одновременно устанавливая 1 на входе irq. Nios обработает исключение, если установлен бит IE и требуемый номер прерывания меньше текущего значения в поле IPRI регистра STATUS (то есть имеет более высокий приоритет). Управление передается драйверу исключения, чей номер подается на входы irq_number.

Внешняя логика, необходимая для того, чтобы подать на вход irq_number номер прерывания и установить сигнал на входе irq, автоматически генерируется программным обеспечением Nios SOPC builder и включена в периферийный модуль шины PBM вне процессора. Периферийные устройства, требующие прерывание, генерируют только один или более сигналов запроса на прерывание, которые объединяются в пределах PBM, где производится генерация сигналов Nios irq_number и устанавливается сигнал irq.

Вход Nios irq работает по уровню. Входы irq и irq_number проверяются на каждом переднем фронте синхрочастоты. Внешние источники, которые генерируют прерывания, должны установить свои сигналы на выводах irq, пока прерывание не подтверждено программным обеспечением (например, сброс запроса прерывания периферийного устройства в 0 при соответствующей записи в регистр). Прерывания, которые установлены, а затем сброшены до того, как Nios начнет их обрабатывать, игнорируются.

Внутренние источники исключения

Есть два источника внутренних исключений: переполнение окна регистра и выход за нижний предел окна регистра. Архитектура процессора Nios позволяет точную обработку для всех внутренних исключений. В каждом случае для драйвера исключения можно установить все условия для того, чтобы команды обработки исключения были успешно выполнены.

Выход за нижний предел окна регистра

Исключение, происходящее всякий раз при выходе за нижний предел окна регистра, то есть когда используется самое низкое допустимое окно регистра (CWP = LO_LIMIT) и выполняется команда SAVE. Команда SAVE перемещает CWP ниже LO_LIMIT, и %sp устанавливается как и при нормальной операции SAVE. В этом случае генерируется исключение выхода за нижний предел окна регистра, и управление передается подпрограмме обработки исключения, прежде чем следующая за SAVE команда будет выполнена.

Когда команда SAVE вызывает исключение выхода за нижний предел окна регистра, CWP декрементируется только один раз, прежде чем управление переходит к подпрограмме обработки исключения. Драйвер исключения выхода за нижний предел окна регистра считает значение CWP = LO_LIMIT — 1. Исключение выхода за нижний предел окна регистра имеет номер 1. CPU не будет обрабатывать исключение выхода за нижний предел окна регистра, если прерывания заблокированы (IE = 0) или текущее значение в IPRI меньше или равно 1.

Действие, предпринимаемое подпрограммой драйвера исключения выхода за нижний предел окна регистра, зависит от требований системы. Для систем, выполняющих большой и сложный код, драйверы выхода за нижний предел окна регистра (так же, как и выхода за верхний предел окна регистра) могут представлять файловый регистр как виртуальный файловый регистр, который простирается за пределы физического файлового регистра. Когда происходит выход за нижний предел окна регистра, драйвер выхода за нижний предел окна регистра мог бы (например) сохранять текущее содержание полного файла регистра в памяти и перезагружать CWP назад в HI_LIMIT, позволяя программе продолжить открывать окна регистра. С другой стороны, есть много встроенных систем, где пользователи могли бы желать управлять использованием стека и глубиной вложений вызовов подпрограмм. Такие системы могли бы осуществлять вывод сообщения об ошибках в драйвере выхода за нижний предел окна регистра и после этого выходить из программы.

Программист определяет характер и действия, принятые драйвером исключения выхода за нижний предел окна регистра. Комплект разработчика программного обеспечения Nios SDK включает и автоматически устанавливает по умолчанию подпрограмму, которая виртуализирует файл регистра, используя стек как временную память.

Исключение выхода за нижний предел окна регистра может быть вызвано только командой SAVE. Непосредственная запись в CWP (через команду WRCTL) значения меньшего, чем LO_LIMIT, не будет вызывать исключение выхода за нижний предел окна регистра. Выполнение команды SAVE, когда CWP уже ниже LO_LIMIT, не будет вызывать исключение выхода за нижний предел окна регистра.

Переполнение окна регистра

Исключение переполнения окна регистра происходит всякий раз, когда используется самое высокое допустимое окно регистра (CWP = HI_LIMIT) и выполняется команда RESTORE. Управление будет передано подпрограмме обработки исключения прежде, чем выполнится команда, следующая после RESTORE.

Когда исключение переполнения окна регистра принято, CWP установлен в HI_LIMIT так, будто CWP инкрементируется командой RESTORE, но затем немедленно декрементируется, как в последствие нормальной обработки исключения. Исключение переполнения окна регистра имеет номер 2.

Действие, предпринимаемое подпрограммой драйвера исключения переполнения, зависит от требований системы. Для систем, выполняющих большой и сложный код, переполнение может создавать виртуальный файл регистра, который выходит за пределы физического файла регистра. Когда переполнение происходит, драйвер переполнения мог бы, к примеру, перезагружать все содержание физического файла регистра из стека и восстанавливать CWP назад в LO_LIMIT. Есть множество систем, где пользователи могли бы желать управлять использованием стека и глубиной вложенности подпрограмм. Такие системы могли бы выводить сообщения об ошибках в драйвере переполнения окна регистра и после этого осуществлять выход из программы.

Программист определяет характер и действия, предпринимаемые драйвером. Nios SDK автоматически устанавливает драйвер переполнения окна регистра, который виртуализирует файловый регистр, используя стек.

Исключение переполнения окна регистра может быть вызвано только командой RESTORE. Непосредственная запись в CWP (через команду WRCTL) значения большего, чем HI_LIMIT, не будет вызывать исключение переполнения окна регистра. Выполнение команды RESTORE, когда CWP уже выше HI_LIMIT, также не будет вызывать исключение переполнения окна регистра.

Прямые программные исключения (команды TRAP)

Программное обеспечение может непосредственно передавать управление драйверу исключения, используя команду TRAP. Поле IMM6 команды дает номер исключения. Команды TRAP всегда обрабатываются, независимо от установки битов IPRI или IE. Команды TRAP не имеют слота задержки. Команда, следующая немедленно после TRAP, будет выполнена после выхода из драйвера исключения.

Ссылка на команду, следующую после TRAP, будет сохранена в %o7 так, чтобы команда TRET передала управление назад команде, следующей после TRAP при завершении обработки исключения.

Последовательность обработки исключения

Когда исключение от любого из источников, упомянутых выше, будет обработано, производятся следующие действия:

1. Содержание регистра STATUS будет скопировано в регистр ISTATUS.
2. CWP декрементируется и открывает новое окно для использования подпрограммой драйвера исключения (кроме случая, когда произошло исключение выхода за нижний предел окна регистра, где CWP был уже декрементирован командой SAVE, которая и вызвала исключение).
3. IE будет установлен в 0, запрещая прерывания.
4. IPRI будет установлен с 6-битовым номером исключения.
5. Адрес следующей невыполненной команды в прерванной программе будет помещен в %o7.
6. Адрес начала драйвера исключения будет выбран из таблицы вектора исключения и записан в PC.
7. После того как драйвер исключения завершает работу, выдается команда TRET, чтобы возвратить управление к прерванной программе.

Использование окна регистра

Вся обработка исключения производится в последнем открытом окне регистра. Этот процесс уменьшает сложность и время ожидания драйверов исключения, потому что они не ответственны за поддержание аппаратной обработки процедуры исключения. Драйвер исключения может свободно использовать регистры %o0... %L7 в недавно открытом окне. Драйвер исключения не должен выполнять команду SAVE в начале обработки. Использование команд SAVE и RESTORE внутри драйверов исключения будет обсуждено позже.

Поскольку передача управления обработчику исключения всегда открывает новое окно регистра, программы должны всегда оставлять одно окно регистра доступным для исключений. Установка (настройка) LO-LIMIT в 1 гарантирует, что одно окно является доступным для исключений (значение сброса LO_LIMIT — 1). Всякий раз, когда программа выполняет команду SAVE, которая может израсходовать последнее окно регистра (CWP = 0), генерируется «trap» при выходе за нижний предел окна регистра. Драйвер выхода за нижний предел окна регистра для самого регистра будет выполнен в конечном окне (с CWP = 0).

Правильно написанное программное обеспечение никогда не будет обрабатывать исключение, когда CWP = 0. CWP может быть только 0 во время обработки исключения, и драйверы исключения должны иметь четкую защиту от перепредоставления прерываний.

Сохранение состояния: регистр ISTATUS

Когда исключение происходит, регистр STATUS будет скопирован в регистр ISTATUS. Затем регистр STATUS будет изменен (IE сброшен в 0, IPRI устанавливается в 1 и декрементируется CWP). Первоначальное содержание регистра STATUS сохраняется в регистре ISTATUS. Когда обработчик исключения возвращает управление к первоначальной программе, содержание регистра первоначальной программы STATUS будет восстановлено из ISTATUS командой TRET.

После входа в драйвер исключения прерывания будут автоматически заблокированы, так что нет никакой опасности, что данные, хранящиеся в регистре ISTATUS, будут перезаписаны последующим прерыванием или исключением. Случаи вложенных драйверов исключения (драйверы исключения, которые используют или переразрешают исключения) будут подробно обсуждены ниже.

Вложенные драйверы исключений должны явно сохранять, поддерживать и восстанавливать содержание регистра ISTATUS прежде и после предоставления последующих прерываний.

Адрес возврата

Когда происходит исключение, выполнение основной программы временно приостанавливается. Команда, которая резервировалась (то есть команда, которая выполнится, но еще не выполнялась) в прерванной программе, определяется как место возврата после обработки исключения.

Место возврата будет сохранено в %o7 (последнее открытое окно регистра в драйвере исключения) прежде, чем управление будет передано драйверу исключения. Значение, сохраненное в %o7 — адрес байта команды возврата, сдвинутое вправо на один бит. Это значение непосредственно используется как адрес команды TRET. Драйверы исключения обычно выполняют команду TRET %o7, чтобы возвратить управление прерванной программе.

Простые и сложные драйверы исключения

Архитектура процессора Nios позволяет создавать эффективные, простые драйверы исключения. Аппаратные средства выполняют сохранение состояния многих регистров еще до того, как запустится требуемый драйвер исключения. Простые драйверы исключений могут игнорировать все автоматические аспекты обработки исключения. Сложные драйверы исключения (например, вложенные) должны быть выполнены с дополнительными предосторожностями.

Простые драйверы исключения

Драйвер исключения рассматривается как простой, если он подчиняется следующим правилам:

- не переразрешает прерывания;
- не использует SAVE или RESTORE (непосредственно, либо вызывая подпрограммы, которые используют SAVE или RESTORE);
- не использует команды TRAP и не вызывает никакие подпрограммы, использующие команды TRAP;
- не изменяет содержание регистров %g0... %g7, или %i0... %i7.

Любой драйвер исключения, который подчиняется этим правилам, может не использовать специальные меры защиты ISTATUS или адреса возврата в %o7. Простой драйвер исключения не требует специального управления окном регистра или CWP.

Сложные драйверы исключения

Драйвер исключения рассматривается как сложный, если он нарушает любое из требований к простому драйверу исключения, перечисленных выше. Сложные драйверы исключения позволяют вложенную обработку исключений и выполнение более сложного кода (например, подпрограммы, имеющие команды SAVE и RESTORE). Сложный драйвер исключения имеет следующие дополнительные особенности:

- Он должен сохранить содержание ISTATUS перед перепредоставлением прерываний. Например, ISTATUS может быть сохранен в стеке.
- Он должен проверить CWP перед перепредоставлением прерываний, чтобы убедиться, что CWP расположен выше LO_LIMIT. Если CWP расположен ниже LO_LIMIT, то он должен открыть более доступные окна регистра (например, сохранить содержание файлового регистра в оперативной памяти) или сообщить об ошибке.
- Он должен переразрешить прерывания при выполнении двух вышеупомянутых условий перед выполнением любых команд SAVE или RESTORE или подпрограмм, которые выполняют команды SAVE или RESTORE.
- До возвращения управления к основной программе, в которой возникло прерывание, он должен восстановить содержание регистра ISTATUS, включая любые корректировки CWP, если окно регистра было преднамеренно сдвинуто.
- До возвращения управления к основной программе он должен восстановить содержание окна регистра в основной программе.

Конвейер

Одна операция конвейера

Nios CPU имеет конвейерную RISC-архитектуру. Работа конвейера скрыта от программного обеспечения, кроме случаев со слотами задержки перехода и случаев, когда CWP изменяется при прямой записи WRCTL.

Стадии работы конвейера включают:

- Выборка команды. Nios CPU устанавливает адрес для подсистемы памяти, откуда возвращается команда, хранящаяся в данном адресе.
- Производится декодирование команды и, если есть операнды, то они читаются из файлового регистра. Адрес перехода для команд BR и BSR вычисляется на специализированном сумматоре.
- Выполнение. Операнды и управляющие биты передаются в ALU. ALU вычисляет результат.
- Запись. Производится, когда требуется, чтобы результат ALU был записан в регистр назначения.

Слоты задержки перехода

Слот задержки перехода — это команда, немедленно следующая после BR, BSR, CALL или JMP. Слот задержки перехода выполняется после команды перехода, но перед командой, которая будет выполняться по новому адресу после перехода. Таблица 12 иллюстрирует слот задержки перехода для команды BR.

Таблица 12. Пример слота задержки команды BR Branch

```
            ...
(a)         ADD  %g2,  %g3
(b)         BR   Target
(c)         ADD  %g4,  %g5      <-- Branch Delay Slot
(d)         ADD  %g6,  %g7
            ...
            Target:
(e)         ADD  %g8,  %g9
```

После того как команда перехода (b) принята, команда (c) выполняется прежде, чем управление передается по адресу перехода (e). Последовательность выполнения вышеупомянутого фрагмента кода была бы (a), (b), (c), и (e). Команда (c) — слот задержки перехода команды (b). Команда (d) не выполняется. Большинство команд может использоваться как слот задержки перехода, кроме следующих:

BR, RSR, CALL, IF1, IFO, IFRnz, IFRz, IFs, JMP, LRET, PFX, RET, SKP1, SKPO, SKPRnz, SKPRz, SKPs, TRET, TRAP.

Прямая манипуляция CWP

Каждая команда WRCTL, изменяющая регистр STATUS (%ctl0), должна сопровождаться командой NOP.

Команды, определяемые пользователем

В версии 2.0 процессора Nios фирмой Altera произведено важное дополнение к возможностям системы — введены команды, определяемые пользователем (заказные команды). Теперь проектировщики могут ускорять критические по времени выполнения программные алгоритмы, добавляя собственные команды к системе команд Nios. Системные проектировщики могут использовать такие команды, чтобы добавить к ALU аппаратный узел собственной разработки, позволяющий обрабатывать задачи за один или несколько тактов процессора. Используя такие команды, дополнительно добавленная пользователем логика может обращаться к памяти и логике вне системы Nios (см. рис. 6).

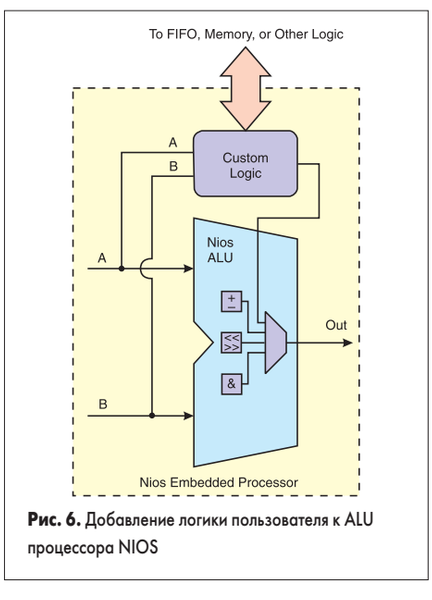

Рис. 6. Добавление логики пользователя к ALU процессора NIOS

Используя заказные команды, системные проектировщики могут уменьшать сложную последовательность стандартных команд, применяя вместо них единственную команду, которая аппаратными средствами выполняет те же действия. Системные проектировщики могут использовать эту особенность для ряда прикладных программ, например, оптимизировать программные циклы для обрабатываемого по алгоритмам DSP цифрового сигнала, обработки заголовка пакета и для прикладных программ со сложными вычислениями. Более того, как уже говорилось [1], применение заказных команд позволит реализовывать мультимикропроцессорные устройства, где ведущий процессор запускает на выполнение процесс в ведомом через обращение к нему заказной командой. Здесь необходимо отметить основное отличие микропроцессорного устройства, реализованного в FPGA от стандартного микропроцессора. В стандартном микропроцессоре скорость обработки задачи зависит только от скорости работы самого микропроцессора, ресурсов микропроцессора, таких, как разрядность шин, число регистров, наличие умножителей и т. д. и эффективности программного кода. При этом обычно все ресурсы микропроцессора участвуют в решении задачи последовательно, а данные из одного «элемента вычислительного процесса» переносятся в другой «элемент вычислительного процесса» под управлением программы, которая выполняется при данном решении задачи. Для микропроцессорного устройства, реализованного в FPGA, ресурс — это логические ячейки в

Таблица 13. Команды пользователя: код операции, тип и формат

Код операции	Тип	Формат
USR0	RR	Ra <– Ra op Rb
USR1	Rw	Ra <– Ra op %r0
USR2	Rw	Ra <– Ra op %r0
USR3	Rw	Ra <– Ra op %r0
USR4	Rw	Ra <– Ra op %r0

Таблица 14. Описание формата записи команд для 32-битной версии

Notation	Meaning
X ← Y	X is written with Y
∅ ← e	Expression e is evaluated, and the result is discarded
RA	One of the 32 visible registers, selected by the 5-bit a-field of the instruction word
RB	One of the 32 visible registers, selected by the 5-bit b-field of the instruction word
RP	One of the 4 pointer-enabled (P-type) registers, selected by the 2-bit p-field of the instruction word
iMMn	An n-bit immediate value, embedded in the instruction word
K	The 11 -bit value held in the K register, (K can only be set by a PFX instruction)
0xnn.mm	Hexadecimal notation (decimal points not significant added for clarity)
X : Y	Bitwise-concatenation operator. For example: (0x12 : 0x34) = 0x1234
{e1, e2}	Conditional expression. Evaluates to e2 if previous instruction was PFX, e1 otherwise
σ(X)	X after being sign-extended into a full register-sized signed integer
X[n]	The nth bit of X (n = 0 means LSB)
X[n..m]	Consecutive bits n through m of X
C	The C (carry) flag in the STATUS register
CTLk	One of the 2047 control registers selected by K
PC	(Program Counter) Byte address of currently executing instruction.
X >> n	The value X after being right-shifted n bit positions
X<<n	The value X after being left-shifted n bit positions
^{b_n}X	The n^{th} byte (8-bit field) within the full-width value X. ^{b0}X = X[7..0], ^{b1}X = X[15..8], ^{b2}X = X[23..16], and ^{b3}X = X[31..24]
^{h_n}X	The n^{th} half-word (16-bit field) within the full-width value X. ^{h0}X = X[15..0]. ^{h1}X = X[31..16]
X & Y	Bitwise logical AND
X | Y	Bitwise logical OR
X ⊕ Y	Bitwise logical exclusive OR
~X	Bitwise logical NOT (one's complement)
IXI	The absolute value of X (that is, –X if (X < 0), X otherwise).
Mem32[X]	The aligned 32-bit word value stored in external memory, starting at byte address X
align32(X)	X & 0xFF.FF.FF.FC, which is the integer value X forced into full-word alignment via truncation
VECBASE	Byte address of Vector #0 in the interrupt vector table (VECBASE is configurable)

Таблица 15. Формат команд для 32-битной версии

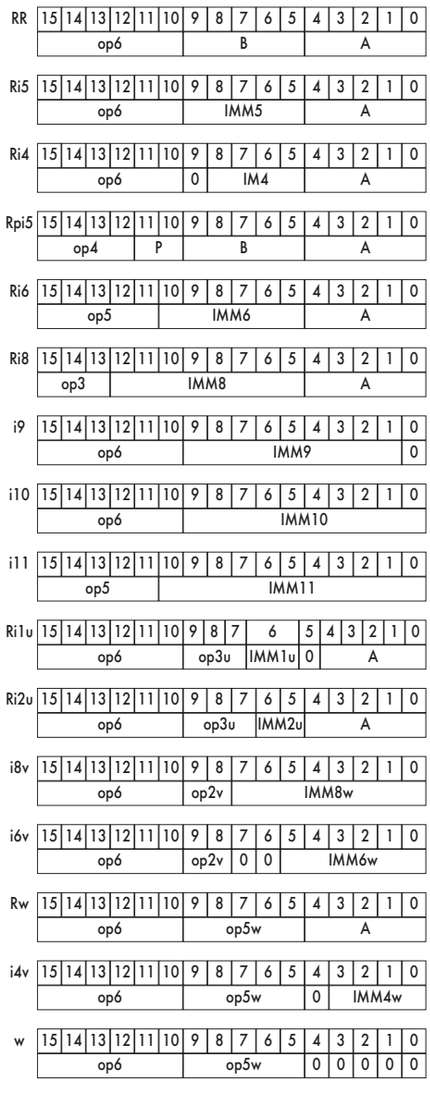

Таблица 16.1. Коды операций команд для 32-битной версии

Opcod	Mnemonic	Format	Summary
000000	ADD	RR	RA ← RA + RB Flags affected: N, V, C, Z
000001	ADDI	Ri5	RA ← RA+(0×00.00 : K : IMM5) Flags affected: N, V, C, Z
000010	SUB	RR	RA ← RA – RB Flags affected: N, V, C, Z
000011	SUBI	Ri5	RA ← RA – (0×00.00 : K : IMM5) Flags affected: N, V, C, Z
000100	CMP	RR	∅ ← RA – RB Flags affected: N, V, C, Z
000101	CMPI	Ri5	∅ ← RA – (0×00.00 : K : IMM5) Flags affected: N, V, C, Z
000110	LSL	Ri5	RA ← (RA<<RB[4..0]), Zero-fill from right
000111	LSLI	Ri5	RA ← (RA<<IMM5), Zero-fill from right
001000	LSR	RR	RA ← (RA>>RB[4..0]), Zero-fill from left
001001	LSRI	Ri5	RA ← (R>>IMM5), Zero-fill from left
001010	ASR	RR	RA ← (RA>>RB[4..0]), Fill from left withRA [31]
001011	ASRI	Ri5	RA ← (RA>>IMM5), Fill from left withRA [31]
001100	MOV	RR	RA ← RB
001101	MOVI	Ri5	RA ← (0×00.00 : K : IMM5)
001110	AND	RR Ri5	RA ← RA & {RB, (0×00.00 : K : IMM5)} Flags affected: N, Z
001111	ANDN	RR, Ri5	RA ← RA &~({RB, (0×00.00 : K : IMM5)}) Flags affected: N, Z
010000	OR	RR Ri5	RA ← RA | {RB, (0×00.00 : K : IMM5)} Flags affected: N, Z
010001	XOR	RR Ri5	RA ← RA ⊕ {RB, (0×00.00 : K : IMM5)} Flags affected: N, Z
010010	BGEN	Ri5	RA ← 2^{IMM5}
010011	EXT8D	RR	RA (0×00.00.00 : ^{bn}RA) where *n* = RB[1..0]
010100	SKP0	Ri5	Skip next instruction if: (RA [IMM5] ==0)
010101	SKP1	Ri5	Skip next instruction if: (RA [IMM5] ==1)
010110	LD	RR	RA ← Mem32 [align32(RB + (σ(K) × 4))]

Таблица 16.2. Коды операций команд для 32-битной версии

Opcode	Mnemonic	Format	Summary
010111	ST	RR	Mem32 [align32(RB + (σ(K) × 4))] ← RA
011000	STS8S	i10	^{bn}Mem32 [align32(%sp + IMM 10)] ← ^{bn}%r0 where *n* = IMM10[1..0]
011001	STS16S	i9	^{hn}Mem32 [align32(%sp + IMM9 × 2)] ← ^{hn}%r0 where *n* – IMM9[0]
011010	EXT16D	RR	RA ← (0×00.00 : ^{hn}RA) where *n* = RB[1]
011011	MOVHI	Ri5	^{h1}RA ← (K : IMM5), ^{h0}RA unaffected
011100	USRO	RR	RA ← RA <user-defined operation> RB
011101000	EXT8S	Ri1u	RA ← (0×00.00.00 : ^{bn}RA) where *n* = IMM2u
011101001	EXT16S	Ri1u	RA ← (0×00.00 : ^{hn}RA) where *n* = IMM1u
011101010			Unused
011101011			Unused
011101100	ST8S	Ri1u	^{bn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{bn}%r0 where *n* = IMM2u
011101101	ST16S	Ri1u	^{hn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{hn}%r0 where *n* = IMM1u
01111 000	SAVE	i8v	CWP ← CWP – 1; %sp ← %fp – (IMM8V × 4) If (old-CWP == LO_LIMIT) then TRAP #1
0111100100	TRAP	i6v	ISTATUS ← STATUS; IE ← 0; CWP ← CWP – 1; I PRI ← IMM6v; %r15 ← ([PC + 2) >>1); PC ← Mem32 [VECBASE + (IMM6v × 4)] × 2
01111100000	NOT	Rw	RA ← ~RA
01111100001	NEG	Rw	R A ← 0 – RA
01111100010	ABS	Rw	RA ← |RA|
01111100011	SEXT8	Rw	RA ← σ(^{b0}RA)
01111100100	SEXT16	Rw	RA ← σ(^{h0}RA)
01111100101	RLC	Rw	C ← msb (RA); RA ← (RA << 1) : C Flag affected: C
01111100110	RRC	RW	C ← RA[0]; RA ← C : (RA >> 1) Flag affected: C
01111100111			Unused
01111101000	SWAP	Rw	RA ← ^{h0}RA:^{h1}RA
01111101001	USR1	Rw	RA ← RA <user-defined operation> R0
01111101010	USR2	Rw	RA ← RA <user-defined operation> R0
01111101011	USR3	RW	RA ← RA <user-defined operation> R0
01111101100	USR4	Rw	RA ← RA <user-defined operation> R0
01111101101	RESTORE	w	CWP ← CWP + 1; if (old-CWP == HI_LIMIT) then TRAP #2
01111101110	TRET	Rw	PC ← (RA × 2); STATUS ← ISTATUS

Таблица 16.3. Коды операций команд для 32-битной версии

Opcode	Mnemonic	Format	Summary
01111101111			Unused
01111110000	ST8D	Rw	^{bn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{bn}%r0 where *n* = RA[1..0]
01111110001	ST16D	Rw	^{hn}Mem32 [align32(RA + (σ(K) × 4))] ← ^{hn}%r0 where *n* = RA[1]
01111110010	FILL8	Rw	%r0 ← (^{b0}RA : ^{b0}RA : ^{b0}RA : ^{b0}RA)
01111110011	FLL16	Rw	%r0 ← (^{h0}RA : ^{h0}RA)
01111110100	MSTEP	Rw	if (%r0[31] == 1) then %r0 ← (%r0 << 1) + RA else %r0 ← (%r0<<1)
01111110101	MUL	Rw	%r0 ← (%r0 & 0x0000.ffff) × (RA & 0x0000.ffff)
01111110110	SKPRZ	Rw	Skip next instruction if: (RA == 0)
01111110111	SKPS	Mw	Skip next instruction if condition encoded by IMM4w is true
01111111000	WRCTL	Rw	CTLk ← RA
01111111001	RDCTL	Rw	RA ← CTLk
01111111010	SKPRNZ	Rw	Skip next instruction if: (RA ! = 0)
01111111011			Unused
01111111100			Unused
01111111101			Unused
01111111110	JIV1P	Rw	PC ← (RA × 2)
01111111111	CALL	Rw	R15 ← ([PC + 4) >> 1); PC ← (RA × 2)
100000	BR	i11	PC ← PC + ([σ(IMM11) + 1) × 2)
100001			Unused
100010	BSR	i11	PC ← PC + C[σ(IMM11) + 1) × 2); %r15 ← ([PC + 4)>>1)
10011	PFX	i11	K ← IMM11 (K set to zero after next instruction)
1010	STP	RPi5	Mem32[align32(RP + (σ(K : IMM5) × 4))] ← RA
1011	LDP	RPi5	RA ← Mem32 [align32(RP + (σ(K : IMM5) × 4))]
110	STS	Ri8	Mem32[align32(%sp + (IMM8 × 4))] ← RA
111	LDS	Ri8	RA ← Mem32 [align32(%sp + (IMM8 × 4))]

FPGA, и формально совершенно не важно, что из этого ресурса организовано и как именно этот ресурс связан с микропроцессором — как отдельный вычислительный узел, например умножитель с накоплением, или как микропрограммный автомат — аппаратный сопроцессор, реализующий конкретную процедуру вычислений, например FFT или FIR. Мало того, когда пользователь определил ресурс, требуемый для решения специфичных аппаратных задач пользователя, все остальные ресурсы микросхемы FPGA могут быть задействованы для реализации вычислительных узлов и аппаратных сопроцессоров.

Далее необходимо отметить еще одну особенность реализации вычислительного процесса в FPGA. При реализации вычислений на аппаратном сопроцессоре данные передаются из одного «элемента вычислительного процесса» в другой аппаратно, не требуя затрат времени от основного процессора. При обработке массивов данных все элементы сопроцессора участвуют в вычислении одновременно, как стадии конвейера обработки данных. Сам же аппаратный сопроцессор может быть настроен на решение требуемой задачи пользователя — разрядность шин, число ступеней конвейера, тип умножителя и т. д. И есть еще одна возможность в FPGA, применив которую, можно значительно повысить скорость обработки задач для многоканальных вычислений. Так, например, при обработке многоканального потока HDLC возможно реализовать аппаратный сопроцессор и дополнительный стек данных для хранения временных результатов вычислений. Загрузка или выгрузка результатов вычислений из аппаратного сопроцессора потребует только одну команду, а сами вычисления для обработки данных в канале потребуют тоже только одну команду от основного процессора. Всего на обработку одного канала потребуется до 10 тактов синхрочастоты основного процессора (1 такт — загрузить сопроцессор состоянием вычислений на предыдущем этапе для данного канала, 8 тактов — обработка байта данных потока HDLC, 1 такт — сохранить результат вычислений сопроцессора для данного канала). Обобщая, можно сказать, что аппаратные сопроцессоры с возможностью их пе-

Таблица 17. Псевдокоманды GNU Compiler/Assembler

Псевдо-команда	Эквивалентная команда	Примечание
LRET	JMP %o7	LRET не имеет операндов
RET	JMP %i7	RET не имеет операндов
NOP	MOV %g0, %g0	NOP не имеет операндов
IF0 %rA, IMM5	SKP1 %rA, IMM5	
IF1 %rA, IMM5	SKP0 %rA, IMM5	
IFRZ %rA	SKPRNZ %rA	
IFRNZ %rA	SKPRZ %rA	
IFS cc_c	SKPS cc_nc	
IFS cc_nc	SKPS cc_c	
IFS cc_z	SKPS cc_nz	
IFS cc_nz	SKPS cc_z	
IFS cc_mi	SKPS cc_pl	
IFS cc_pl	SKPS cc_mi	
IFS cc_ge	SKPS cc_lt	
IFS cc_lt	SKPS cc_ge	
IFS cc_le	SKPS cc_gt	
IFS cc_gt	SKPS cc_le	
IFS cc_v	SKPS cc_nv	
IFS cc_nv	SKPS cc_v	
IFS cc_ls	SKPS cc_hi	
IFS cc_hi	SKPS cc_ls	

Таблица 19. Пример организации регистрового файла

(Internal Register File)	CWP=6 (HI_LIMIT)						
Reg[120..127]	%i0..%i7						
Reg[112..119]	%L0..%L7	CWP=5					
Reg[104..111]	%o0..%o7	%io..%i7					
Reg[96..103]		%L0.. %L7	CWP=4				
Reg[88..95]		%o0..%o7	%i0..%i7				
Reg[80..87]			%L0..%L7	CWP=3			
Reg[72..79]			%o0..%o7	%i0..%i7			
Reg[64..71]				%L0,.%L7	CWP=2		
Reg[56..,63]				%o0..%o7	%io..%i7	CWP=1 (LOW_LIMIT)	
Reg[48..55]					%L0..%L7		
Reg[40..47]					%o0..%o7	%i0..%i7	CWP=0 (Underflow or TRAP)
Reg[32..39]						%L0..%L7	
Reg[24..31]						%o0..%o7	%i0..%i7
Reg[16..23]							%L0..%L7
Reg[8..15]							%o0..%o7
Reg[0..7]	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7	%g0..%g7

Restore ←

Save →

This shows the smallest Nios register file, which is 128 registers. Larger files have more register windows.

Register Groups for CWP=0	
%r24..%r31	aka %io..%i7
%r16..%r23	aka %LO..%L7
%r8..%r15	aka %oO..%o7
%r0..%r7	aka %gO..%g7

резагрузки при переключении задач значительно повышают скорость отклика при переключении с задачи на задачу и эффективность работы микропроцессора.

Для конфигурации Nios CPU фирмой Altera разработан Мастер конфигурации, к которому обращается SOPC Builder. Он обеспечивает графический интерфейс для определения конфигурации пяти заказных команд процессора Nios.

Мастер конфигурации Nios интегрирует заказные логические блоки с процессором Nios ALU при формировании Nios. Он также создает программные макрокоманды в C/C++ и ассемблере, обеспечивая программный доступ к этим заказным логическим блокам. Пользователь должен задать название (имя) макрокоманды. Если заказная команда комбинаторная, то число циклов синхрочастоты, необходимых для исполнения команды, установлено в 1. Если заказная команда требует несколько циклов, то пользователь должен указать требуемое число циклов синхрочастоты.

Код операции, тип и формат заказных команд приведены в таблице 13. Заказная команда USR0 берет содержимое двух регистров общего назначения Ra и Rb и выполняет над ними операцию, определяемую логическим блоком пользователя. Результат этой операции сохраняется в регистре Ra. Код операции для команды USR0 — RR.

Заказные команды USR1... USR4 берут содержимое двух регистров общего назначения Ra и %r и выполняют над ними операцию, определяемую логическим блоком пользователя. Результаты этих операций так же сохраняются в регистре Ra. Код операции для команд USR1... USR4 — Rw.

Более подробно заказные команды описаны в документации [10].

Таблица 14 содержит краткое описание синтаксиса команд для 32-битной версии, а таблица 15 — формат команд для 32-битной версии. Коды операций команд приведены в таблице 16.

Команды, описание которых приведено в таблице 17, генерируются компилятором и совместимы с ассемблером.

Таблица 18. Операторы NIOS

Операторы	Описание	Операция
%lo(x)	Извлекает младшие 5 бит из слова x	x & 0x0000001f
%hi(x)	Извлекает биты 15..5 из слова x	(x >> 5) & 0x000007ff
%xlo(x)	Извлекает биты 20..16 из слова x	(x >> 16) & 0x0000001f
%xhi(x)	Извлекает биты 31..21 из слова x	(x >> 21) & 0x000007ff
x@h	Адрес половины слова x	x >> 1

Операторы, описание которых приведено в таблице 18, могут быть использованы с константами и символическими адресами и совместимы с ассемблером и линкером.

В таблице 19 показан пример организации регистрового файла для 128 регистров. Показано движение окна при выполнении команд SAVE и RESTORE.

Литература

1. И. Каршенбойм, Н. Семенов. Микропрограммные автоматы на базе специализированных ИС // Chip News. 2000. № 7.
2. Altera™. Nios an188 Custom Instructions.