

Реализация архитектуры MIPS32 на языке HDL

Алексей Гребенников (г. Актау, Казахстан)

Статья содержит описание архитектуры MIPS32 на языке Verilog. Рассмотрены принципы действия основных блоков процессора – конвейера обработки данных, модуля умножения/деления, системы управления памятью и системного сопроцессора. Приведены примеры моделирования работы блоков процессора.

ВВЕДЕНИЕ

Архитектура MIPS (Microprocessor without Interlocked Pipeline Stages – микропроцессор без блокировок в конвейере) определяет набор инструкций и основные правила работы микропроцессора. Основной сферой применения процессоров MIPS являются встраиваемые системы. В процессе эволюции эта архитектура претерпевала изменения, и в настоящее время существует два основных подкласса семейства – MIPS32/64 и microMIPS32/64. На уровне ассемблерного исходного кода оба подкласса идентичны, но для процессоров microMIPS генерируется более компактный машинный код.

Все процессоры семейства MIPS относятся к классу RISC (Reduced Instruction Set Computer – компьютер с сокращённым набором инструкций). Достоинством этого класса является фиксированная длина инструкции и минимальный их набор, более сложные задачи обработки данных возлагаются на программное обеспечение. Кроме того, процессоры MIPS характеризуются минимальными простоями конвейера обработки данных, связанными с взаимными блокировками.

Следует различать понятия архитектуры и её реализации. Архитектура определяет набор инструкций, регистров, модель управления памятью, способы обработки исключений и некоторые другие параметры. Реализация

описывает физическое устройство, соответствующее заданной архитектуре. Например, архитектура MIPS32 регламентирует наличие блока умножения/деления. Однако алгоритм этих операций задаётся реализацией процессора. Более подробно с архитектурой MIPS можно ознакомиться в [1] или на интернет-странице www.mips.com.

В данной статье рассмотрено построение модели процессора MIPS32-4Kp (далее 4Kp) на языке Verilog и её реализация для отладочной платы SP605 на базе ПЛИС Xilinx семейства Spartan 6. Более подробно особенности процессора 4Kp описаны в [2], а также в [3], где приведено общее описание процессоров семейства 4K. На русском языке описание архитектуры MIPS32 приведено в [4].

КОНВЕЙЕРНАЯ ОБРАБОТКА ДАННЫХ

Все инструкции процессора выполняются за несколько циклов. Для ядра 4Kp определено пять циклов или стадий:

- I-stage – выборка инструкции из памяти;
- E-stage – декодирование и выполнение инструкции;
- M-stage – запись/чтение данных в память/из памяти;
- A-stage – выравнивание данных в пределах 32-битного слова;
- W-stage – запись результатов выполнения инструкций в память.

Каждый цикл выполняется за один период тактового сигнала, т.е. для выполнения одной инструкции требуется пять периодов. Соответственно, частота выполнения инструкций минимум в пять раз меньше тактовой частоты. При конвейерной обработке выборка инструкций происходит каждый период тактового сигнала без ожидания завершения выполнения предыдущей инструкции.

Допустим, в начальный момент времени процессор выбирает первую инструкцию из памяти, т.е. она находится на стадии I. В следующий момент времени первая инструкция переходит на стадию E, и одновременно происходит выборка следующей инструкции. Далее первая инструкция переходит на стадию M, вторая – на стадию E, и выбирается инструкция на стадию I. Как видно на рисунке 1, на пятом цикле конвейер будет заполнен целиком, т.е. все ступени конвейера будут одновременно обрабатывать данные для разных инструкций. В идеальном случае конвейер будет выдавать обработанную инструкцию каждый период тактового сигнала. Однако существует ряд условий, при которых непрерывная схема выполнения инструкций неприемлема. Рассмотрим фрагмент ассемблерного кода:

```
LW      $t1, 0x400($a0)
ADD     $t3, $t1, $t2
```

Первая инструкция загружает в регистр \$t1 слово из внешней памяти, а следующая за ней инструкция выполняет операцию сложения регистров \$t1 и \$t2. Поэтому для выполнения второй инструкции необходимо, чтобы первая уже была выполнена, иначе значение регистра \$t1 будет неверным. В этом случае конвейер блокируется и ожидает завершения инструкции, вызвавшей блокировку. Для примера выше инструкция ADD будет ждать на стадии E до тех пор, пока инструкция LW не закончит чтение слова из памяти на стадии M.

Архитектура MIPS предусматривает два типа блокировок:

- Stall – конвейер останавливается полностью и ожидает снятия блокировки;



Рис. 1. Работа конвейера

- Slip – останавливается только часть конвейера – ступень, которая вызвала блокировку, и все более ранние ступени. Более поздние ступени завершаются. Например, при блокировке стадии M более поздние стадии A и W могут быть завершены. При этом до снятия блокировки на стадии M конвейер вставляет пустые команды NOP на стадии A и W, которые также называются bubble.

Помимо блокировок, при работе конвейера наблюдается другой вид зависимости инструкций друг от друга, называемый в литературе data bypassing (обходной путь передачи данных). Рассмотрим фрагмент ассемблерного кода:

```
OR    $t3, $t1, $t2
ADD   $t5, $t4, $t3
XOR   $t7, $t5, $t3
```

Первая инструкция выполняет операцию OR для регистров \$t1 и \$t2 и помещает результат в регистр \$t3. Следующая инструкция суммирует регистры \$t4 и \$t3 и помещает результат в регистр \$t5. Но регистр \$t3 был модифицирован предыдущей инструкцией. Все математические операции выполняются на второй стадии конвейера (E), тогда как значения регистров обновляются на пятой стадии (W). Несмотря на то что на момент выполнения второй инструкции (ADD) результат выполнения первой (OR) уже известен, новая величина регистра \$t3 всё ещё находится на стадии M и будет записана непосредственно в регистр \$t3 только через два цикла (A и W). С третьей инструкцией ситуация аналогичная: значения регистров \$t5 и \$t3, необходимые для выполнения операции «исключающее ИЛИ» (XOR), всё ещё находятся на промежуточных стадиях конвейера (\$t3 – на стадии A, \$t5 – на стадии M).

Метод обходной передачи данных (data bypassing), изображённый на рисунке 2, используется во избежание простоев конвейера при взаимной зависимости регистров. Суть его заключается в том, что если инструкции требуются данные, которые уже есть в наличии, но всё ещё находятся на промежуточных стадиях конвейера, эти данные передаются непосредственно из промежуточной стадии, а не из файла регистров. Для рассмотренного выше примера инструкция ADD на стадии E берёт значение \$t4 из

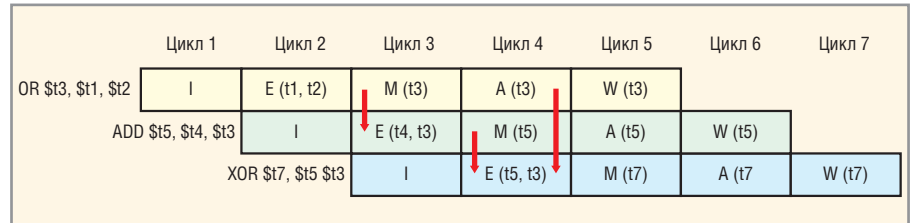


Рис. 2. Обходной путь передачи данных (data bypassing)

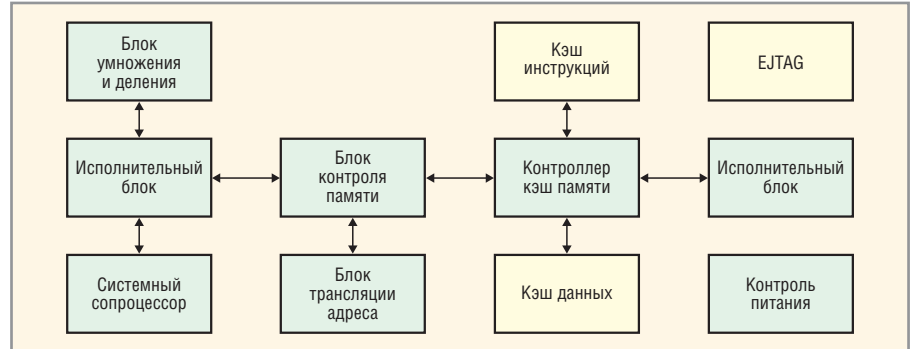


Рис. 3. Блок-схема процессора 4Кр

файла регистров, поскольку величина этого регистра не была модифицирована предыдущими командами. Значение \$t3 берётся со стадии M, на которой находится последняя обновлённая величина регистра \$t3. Для инструкции XOR значение \$t5 берётся со стадии M и значение \$3 – со стадии A.

Ещё одной особенностью работы конвейера является задержка при выполнении инструкций перехода JUMP или BRANCH. Рассмотрим ассемблерный код:

```
AND    $s1, $v1, $t2
J      0xFE00
ORI    $t2, $t3, 0x10
```

После инструкции AND выполняется безусловный переход по адресу, заданному командой J. Задержка заключается в том, что инструкция, следующая непосредственно за командой перехода (в данном случае ORI), всегда выполняется. Это связано с тем, что решение о переходе и вычисление нового адреса программного счётчика принимается на стадии E, т.е. в момент времени, когда следующая инструкция уже поступила на стадию I. Во избежание пустого цикла эта инструкция также выполняется. При генерации машинного кода задержка учитывается автоматически. Инструкция J будет размещена в бинарном файле до инструкции AND. Если такую замену произвести невозможно, ассемблер вставляет дополнительную инструкцию NOP после команды перехода.

БЛОК-СХЕМА ПРОЦЕССОРА 4Кр

Рассмотрим блок-схему процессора 4Кр, изображённую на рисунке 3. Зелёным цветом обозначены обязательные модули, жёлтым – возможные. Исполнительный блок включает в себя все стадии конвейера и выполняет основную работу по обработке данных. Умножение и деление являются наиболее сложными математическими операциями, выполняемыми процессором. Поэтому блок, выполняющий эти функции, изображён отдельно. Системный сопроцессор отслеживает возникновение исключений, внешних прерываний, контролирует трансляцию адреса и режим работы кэш-памяти (при её наличии), задаёт режим выполнения инструкций – пользовательский или привилегированный. Блок контроля памяти, наряду с блоком трансляции адреса, выполняет преобразование виртуального адреса в физический и управляет формированием шинных циклов. Блоки кэш-памяти инструкций и данных являются необязательными. Блок интерфейса с внешней шиной непосредственно формирует шинные циклы при запросе от блока контроллера памяти.

Модель процессора, рассмотренная в данной статье, не содержит кэш-памяти и интерфейса EJTAG. Блок умножения/деления включён в состав исполнительного блока в форме отдельного модуля. Все файлы модели Verilog находятся в архиве MIPS_verilog.zip. Файл верхнего уровня src.v содержит следующие компоненты:

- блок PLL (diff_clk), который преобразует входную частоту 200 МГц в частоту работы процессора 20 МГц и реализован в виде мегафункции;
- модуль верхнего уровня MIPS32 (mips_top_sorc) – содержит всю логику процессора;
- контроллер памяти (mem_ctrl_sorc). Этот тестовый блок используется вместо внешней памяти для контроля работы процессора. Включает в себя внутрикристалльную память объемом 32 Кб и логику взаимодействия с шиной процессора. При использовании внешней памяти, например DDR3 SDRAM, данный блок необходимо заменить соответствующим контроллером памяти;
- асинхронный приёмопередатчик (uart_sorc). Этот модуль подробно описан в [5].

Сначала приёмопередатчик инициализирует память процессора и затем используется для обмена данными с компьютером. Связь с платой со стороны компьютера обеспечивается программой, находящейся в архиве PC_interface_Delphi.zip. Эта программа написана в среде Delphi (Code Gear). Скорость обмена данными – 38 400 Кбит/с. Программа может работать как с бинарными, так и с текстовыми hex-данными.

Для написания тестовых программ для MIPS-процессора использовался свободно доступный в Интернете пакет cross-mipsel-binutils для операционной системы Linux. Заметим, что процессор, описанный в данной статье, имеет порядок следования байт Little Endian, поэтому и был выбран пакет mipsel. Этот пакет содержит ряд программ, таких как ассемблер, дисассемблер, редактор кода и т.д. После генерации ассемблером файла в формате elf его необходимо преобразовать в бинарный формат. Для этого использовалась программа, исходный код которой находится в файле elf2bin.c. Эта программа отображает подробно содержимое elf-файла и одновременно копирует данные сегмента .text. в отдельный файл mips.bin, который и является бинарным образом для процессора MIPS.

Рассмотрим более подробно принципы действия отдельных модулей процессора.

Исполнительный блок

В файле верхнего уровня процессора mips_top.v исполнительный блок

представлен модулем exes_core_top, исходный код которого, в свою очередь, находится в файле exes_core.v. Это – самый объёмный модуль, который включает в себя все пять ступеней конвейера, модуль обхода данных (bypass_e), модуль блокировки конвейера (hazard_e) и модуль умножения/деления (mdu_e).

I-стадия конвейера

Первая стадия конвейера I расположена в модуле istage_e (исходный файл istage.v). Задачей этой стадии является выборка инструкции из внешней памяти и управление регистром PC (Program Counter). После системного сброса регистр PC устанавливается в значение 32'hBFC00000 – виртуальный адрес первой инструкции. Разница между виртуальным и физическим адресом будет рассмотрена ниже при описании работы системного сопроцессора и модуля контроля памяти.

На выходе стадии I instr_o в начальный момент времени выставлена инструкция NOP. Стадия сразу же делает запрос на чтение инструкции по адресу 32'hBFC00000 и при отсутствии исключений или условий перехода ожидает момента времени, когда инструкция будет прочитана из внешней памяти, т.е. сигнал cmd_ready_i перейдёт в единичное состояние. На время ожидания на выходе instr_o выставляется команда NOP (код команды 32'h0):

```
case x
  ((stall_i | ex_branch_i), cmd_ready_i, branch_i)
3'b00x:
    begin
        instr_o <= 32'h0;
```

После получения сигнала готовности инструкция появляется на выходе instr_o на время одного периода тактового сигнала, и регистр PC инкрементируется на величину 4, т.е. размер 32-битного слова в байтах. Этот цикл последовательного выполнения инструкций повторяется до тех пор, пока не перейдёт в активное состояние один из сигналов в вышеописанном фрагменте кода:

- stall_i – блокировка конвейера. В этом случае конвейер останавливается. Регистр инструкций не обновляется, и указатель адреса команды PC не инкрементируется;

- ex_branch_i (exception branch input) – сигнал о том, что произошло исключение. В этом случае значение регистра PC задаётся системным сопроцессором, и до завершения чтения инструкции по этому новому адресу на выходе instr_o выставляется инструкция NOP;

- branch_i – на стадии E конвейера находится инструкция типа BRANCH или JUMP, требующая перехода регистра счётчика команд PC. В этом случае новое значение PC получается в результате вычислений на стадии E.

Основным отличием перехода по исключению и по инструкции условного или безусловного перехода является то, что в первом случае отсутствует задержка выполнения, тогда как во втором случае до фактического перехода по новому адресу выполняется ещё одна инструкция по старому адресу, т.е. обеспечивается условие задержки (branch delay slot).

E-стадия конвейера

Вторая стадия конвейера E расположена в модуле estage_e (исходный файл estage.v). Эта стадия в свою очередь состоит из двух частей – файла регистров e_regfile (исходный файл regfile.v) и модуля арифметико-логического устройства e_alu (исходный файл alu.v).

Файл регистров состоит из 32 регистров общего назначения. Нулевой регистр r0 всегда равен нулю, тридцать первый регистр r31 неявно используется в качестве регистра назначения в инструкциях JAL, BLTZAL, BLTZALL, BGEZAL, BGEZALL. Во всех других инструкциях r31 может быть использован как регистр общего назначения. Кроме того, модуль e_regfile содержит мультиплексор обхода данных, аналогичный стадиям M и A. Это необходимо, когда записываемый регистр на стадии W требуется для вычислений на стадии E. В этом случае новое значение регистра одновременно подаётся на выходы rs_o и rt_o и записывается в соответствующую ячейку reg_bank[rda_i], где rda_i – адрес обновляемого регистра. Заметим, что при написании программ на ассемблере используются специальные обозначения регистров. Например, регистры \$t0 – \$t7 соответствуют номерам \$8 – \$15, \$s0 – \$s7 – номерам \$16 – \$23. Более подробно основы ассемблера для MIPS рассмотрены в [6].

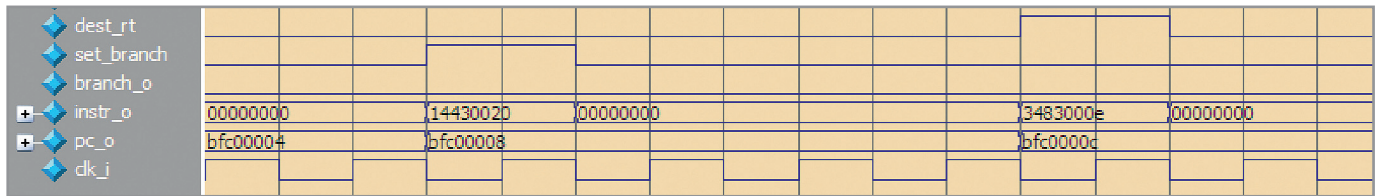


Рис. 4. Симуляция выполнения инструкции BNE

Модуль арифметико-логического устройства `e_alu` декодирует большинство инструкций и производит арифметические вычисления. Декодирование осуществлялось на основе информации в [7]. Рассмотрим более подробно формат инструкций процессора.

Все инструкции имеют фиксированную длину 32 бита и делятся на три основных класса:

- I-type (Immediate). В этом случае шесть старших бит слова задают код операции, затем 10 бит определяют регистры `rs` и `rt` – по пять бит на каждый регистр. Младшие 16 бит – это величина константы;
- J-type (Jump) – команды безусловного перехода. Старшие шесть бит задают код операции, а оставшиеся 26 – адрес перехода;
- R-type (Register) – регистровые инструкции без констант. Старшие шесть бит – это код операции, затем 15 бит – величины регистров `rs`, `rt` и `rd` (по пять бит на каждый регистр); следующие пять бит – величина сдвига `sa` для операций сдвига; и младшие шесть бит – это дополнительный код операции (`funct`). Более подробно форматы инструкций описаны в [3].

В модуле `e_alu` для декодирования каждой инструкции выделен отдельный провод. Например:

```
wire a_add = (op_i == `OP_SPE-
CIAL) && (funct_i == `SP_ADD);
wire a_lb = (op_i == `OP_LB);
```

В этих двух примерах для декодирования операции сложения (`add`) необходимо знать величины шести старших бит (`op_i`) и шести младших (`funct_i`). Тогда как для операции загрузки байта (`lb`) достаточно декодировать поле `op_i`.

Для некоторых инструкций требуется 16-битная константа, расширенная до 32 бит с учётом знака. Эта константа задаётся выражением:

```
wire [31:0] sign_exts_offset =
{{16{offset_i[15]}}, offset_i[15:0]
};
```

Затем в модуле `e_alu` задаются некоторые атрибуты команд:

- `set_branch` – этот атрибут принимает единичное значение для всех команд, которые потенциально вызывают скачок программного счётчика (группы `jump` и `branch`);
- `set_load` – равен единице для всех команд чтения данных из памяти;
- `set_store` – равен единице для всех команд записи данных в память;
- `set_rtype` – равен единице для всех команд, которые записывают результат операции в регистр `rd` или `rt`, т.е. для инструкций R-type и I-type;
- `oflow_instr` – атрибут для обозначения инструкций, которые могут вызывать переполнение;
- `dest_rt` – атрибут для инструкций, которые записывают результат в регистр `rt`.

Некоторые инструкции устанавливают сразу несколько атрибутов. Например, команда `bgezal` устанавливает одновременно линии `set_branch` и `set_rtype`.

Для операций, которые могут вызывать переполнение, необходимо производить соответствующую проверку:

```
assign ioverflow_o =
oflow_instr_buf &
(carry_out^carry_in);
```

В процессоре MIPS32 используется арифметика дополнения до двух. Переполнение возникает в том случае, когда «исключающее ИЛИ» бита переноса в бит 32 (`carry_in`) и бита переноса из бита 32 (`carry_out`) равны единице, как это видно из вышеприведённого выражения. Но флаг переполнения устанавливается только для тех инструкций, которые могут вызвать переполнение, – только если `oflow_instr_buf` равен единице (`oflow_instr_buf` – это сигнал `oflow_inst`, сдвинутый на половину периода тактового сигнала, чтобы все сигналы в выражении изменялись на положительном фронте синхросигнала).

Все арифметические операции выполняются на положительном фронте синхросигнала, т.е. в блоке `always @`

(posedge `clk_i`). Рассмотрим работу модулей `estage_e` и `istage_e` на примере фрагмента программы:

```
BNE $2,$3,0x80
ORI $3,$4,0xE
BEQ $17,$18,0x40
ADDI $3,$2,0xF
```

Первая инструкция сравнивает значения регистров 2 и 3 и в случае их неравенства увеличивает величину программного счётчика на `0x80`. Фрагмент симуляции выполнения этой инструкции приведён на рисунке 4, где `pc_o` – это программный счётчик, который является выходным сигналом стадии I конвейера, `instr_o` – прочитанная из памяти инструкция – также является выходным сигналом стадии I.

В примере на рисунке 4 инструкция `BNE $2,$3,0x80` расположена в памяти по адресу `0 BFC00004` и в машинном коде представлена числом `0x14430020`. Так как процессор читает инструкции из внешней памяти без кэширования, цикл чтения занимает четыре периода тактового сигнала. В периоды ожидания внешней инструкции процессор вставляет команды `NOP` (`0x00000000`) на выход стадии I – регистр `instr_o`. После появления команды `BNE` на выходе `instr_o` программный счётчик принимает своё следующее значение, в данном случае инкрементируется на 4 (`0xBFC00008`), и стадия I инициирует процесс чтения новой инструкции из внешней памяти.

В то же самое время стадия E начинает выполнение уже прочитанной инструкции. Как видно из рисунка 4, команда `BNE` устанавливает линию `set_branch` в единичное значение на один период. Но условие сравнения не выполняется (не показанные на рисунке регистры \$2 и \$3 равны), поэтому в следующем цикле чтения инструкции программный счётчик вновь инкрементируется на четыре, а не принимает значение, заданное в команде `BNE`.

Команда `ORI $3,$4,0xE` расположена в памяти по адресу `0 BFC00008` и кодируется числом `0x3483000E`. Эта команда

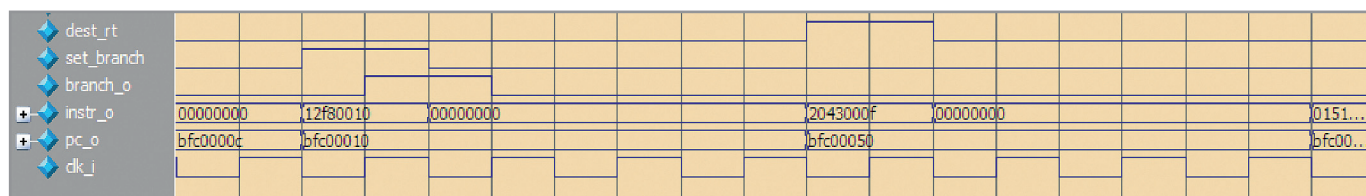


Рис. 5. Симуляция выполнения инструкции BEQ

выполняет операцию логического «И» для регистра \$4 и константы 0xE и помещает результат в регистр \$3. Поскольку адрес результирующего регистра задан в поле rt, во время выполнения этой инструкции линия dest_rt устанавливается в единичное значение.

Продолжение процесса симуляции показано на рисунке 5. Инструкция BEQ \$17,\$18,0x10 расположена в памяти по адресу 0xBFC0000C и кодируется числом 0x12F80010. На отрицательном фронте синхросигнала на стадии E в модуле e_alu линия set_branch устанавливается в единичное значение. Затем проверяется условие равенства регистров \$17 и \$18. В данном примере эти регистры равны, поэтому на следующем положительном фронте выходная линия стадии E branch_o устанавливается в единичное значение. Это – сигнал для стадии I, что непрерывное изменение программного счётчика должно быть прервано и следующее значение pc_o будет установлено командой BEQ.

Для вычисления новой величины программного счётчика необходимо сложить смещение, заданное в команде, с адресом инструкции в слоте задержки (следующей за командой перехода). В данном случае получится $0xBFC00010 + 0x40 = 0xBFC00050$. Заметим, что команда в слоте задержки выполняется независимо от истинности условия. В рассматриваемом примере после инструкции BEQ выполняется команда ADDI \$3,\$2,0xF (которая кодируется числом 0x2043000F), и затем происходит скачок с адреса 0xBFC00010 на адрес 0xBFC00050.

М-, А- и W-стадии конвейера

Третья стадия конвейера М расположена в модуле mstage_e (исходный файл mstage.v). На этой стадии происходит обмен данными с памятью: чтение для операций типа LOAD и запись для операций типа STORE. Процессор может работать с байтами (8 бит), полусловами (16 бит) и словами (32 бита). В зависимости от размерности обмена данных устанавливается регистр

byte_mask_o[3:0], где каждому разряду соответствует байт слова. Например, при чтении первого байта слова этот регистр принимает значение 4'b0001, при чтении слова – 4'b1111. Для операций записи SWL (Store Word Left) и SWR (Store Word Right, запись по невыровненному адресу) производится подготовка данных в соответствии с описанием этих инструкции [7]. Эта стадия может блокировать конвейер при взаимозависимости данных, как это описано выше на примере команды LW. Команды, не обменивающиеся данными с памятью, проходят данную стадию транзитом.

Четвёртая стадия конвейера А расположена в модуле astage_e (исходный файл astage.v). Основная задача этой стадии – выравнивание данных, прочитанных из памяти. Данные из памяти всегда читаются 32-битными словами, поэтому необходима дополнительная обработка прочитанного слова. Например, для команды LBU (Load Byte Unsigned) обнуляются 24 старших бита 32-битного слова. Для команды LB (Load Byte) младший байт рассматривается как знаковое число, поэтому знак (седьмой бит) распространяется на старшие 24 бита. Эта стадия также определяет, требуется ли запись данных для текущей инструкции в файл регистров. Для инструкций, не требующих обновления файла регистров (переход и запись данных), эта стадия не выполняет никаких действий.

Модуль стадии W, расположенный в файле wstage.v, фактически не выполняет никаких действий и включён в модель, чтобы дополнить число стадий до пяти. Единственной задачей этой стадии является запись данных в файл регистров; её выполняет сам модуль файла регистров (модуль e_regfile на стадии E).

Блок умножения и деления

Блок умножения и деления расположен в модуле exes_core, его исходный код представлен в файле mdv.v. Эти две операции требуют значительно больше времени на выполнение и

в версии процессора 4Кр блокируют конвейер до своего полного завершения. Умножение и деление замещаются сложением и вычитанием соответственно. Различные команды требуют от 32 до 34 периодов тактового сигнала и декодируются непосредственно модулем mdv. Все операнды представлены 32-битными словами. Для чисел со знаком используется арифметика дополнения до двух. Результат записывается в один или два 32-битных регистра в зависимости от команды. Отрицательные числа предварительно преобразуются в положительные, и затем, после завершения операции, корректируется знак. Чтобы преобразовать число в кодировке дополнения до двух из положительного в отрицательное или наоборот, необходимо из нулевого регистра вычесть заданное число, при этом перенос игнорируется:

```
if (rt_i[31] && signed_mul)
begin
multiplier <= 32'h0 - rt_i;
rt_sign <= 1'b1;
end
```

В этом фрагменте кода проверяется самый старший бит слова для знаковых операций умножения. Если бит равен единице, т.е. число отрицательное, оно преобразуется в положительную форму и устанавливается регистр знака rt_sign, который после завершения умножения используется для корректировки знака произведения:

```
if ((rt_sign^rs_sign) && signed_mul)
{reg_hi_o,reg_lo_o} <= 64'h0 -
(product + multiplicand);
```

Если знаки перемножаемых чисел различны, то результат будет отрицательным и должен быть преобразован в отрицательное число. Рассмотрим более подробно процесс умножения.

Входные данные представлены 32-битными умножаемым и умножителем на входах rs_i и rt_i. Умножаемое расширяется до 64 бит:

```

multiplicand <= {32'h0,rs_i};
multiplier <= rt_i;

```

Результат умножения записывается в 64-битный регистр, называемый product. Непосредственно процесс умножения выполняется за 32 шага, по количеству бит в слове. На каждом шаге выполняется ряд операций:

```

if (multiplier[0])
product <= product + multipli-
cand;
multiplicand <= multiplicand <<
1;
multiplier <= multiplier >> 1;

```

Если нулевой бит множителя (multiplier) равен единице, то к результату (product) прибавляется умножаемое (multiplicand). Затем умножаемое сдвигается на один бит влево, а множитель – на один бит вправо. Процесс повторяется 32 раза. Окончательное произведение после корректировки знака записывается в соответствующие регистры. Для команд MULT и MULTU – это пара 32-битных регистров reg_hi_o и reg_lo_o. Для команды MUL 32 младших бита результата, т.е. регистр reg_lo_o,

записываются в регистр общего назначения, адрес которого задаётся в команде, и затем передаются на стадию A конвейера через регистр rda_o. При этом модуль mdu также сигнализирует стадии A о необходимости записи в регистр общего назначения линией mul_o.

Команды MADD и MADDU задают умножение с накоплением. Результат умножения из регистров reg_hi_o и reg_lo_o складывается с предыдущим значением этих регистров, которое хранится на стадии A конвейера. MSUB и MSUBU – аналогичные команды, но результат умножения вычитается из предыдущего значения регистров.

Рассмотрим более подробно операции деления DIV и DIVU для знаковых и беззнаковых чисел соответственно. Как и для операций умножения, деление производится для положительных чисел. Отрицательные числа преобразуются в положительные перед началом деления, и затем корректируется знак результата деления. Перед началом деления необходимо инициализировать соответствующие данные. Регистр rs_i после корректировки по зна-

ку загружается в регистр остатка remainder. Размерность регистра remainder равна 64 бита, поэтому старшие 32 бита заполняются нулями:

```
remainder <= {32'h0,rs_signed};
```

Регистр rt_i также после корректировки знака загружается в верхнюю часть 64-битного регистра делителя divisor. В момент инициализации производится сдвиг вправо на один бит:

```

divisor <=
{1'b0,rt_signed,31'h0};

```

Регистр частного quotient имеет размерность 32 бита и в начале цикла деления инициализируется нулевым значением. Заметим, что все эти операции выполняются на положительном фронте синхросигнала. Затем на отрицательном фронте синхросигнала производится вычитание остатка и делителя; результат записывается в промежуточный 64-битный регистр:

```
remainder_temp <= remainder - di-
visor;
```

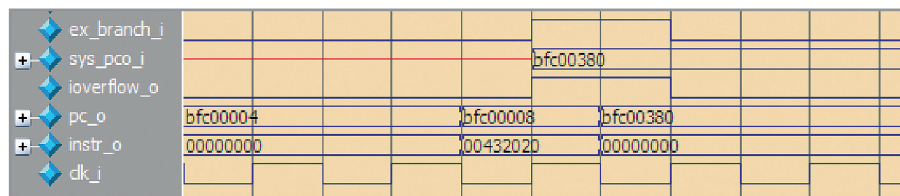


Рис. 6. Симуляция выполнения инструкции ADD

После этого на положительном фронте синхросигнала выполняется основной цикл деления, который повторяется 32 раза:

```
begin
if (!remainder_temp[63])
begin
quotient[31:0] <= {quo-
tient[30:0],1'b1};
remainder <= remainder_temp;
divisor <= divisor >> 1;
end
else
begin
quotient[31:0] <= {quo-
tient[30:0],1'b0};
divisor <= divisor >> 1;
end
```

Проверяется значение старшего бита результата вычитания, которое производилось на отрицательном фронте. Если этот бит равен нулю, то в младший бит частного записывается единица, и регистр частного сдвигается на один бит влево. Регистр остатка remainder обновляется временным значением remainder_temp, чтобы провести операцию вычитания на следующем отрицательном фронте синхросигнала. Делитель сдвигается на один бит вправо.

Если старший бит регистра remainder_temp после вычитания равен единице, то в младший бит частного записывается ноль и частное также сдвигается на один бит влево. Но регистр остатка в этом случае не обновляется, и в следующей операции вычитания используется его старое значение. Регистр делителя в этом случае также сдвигается на один бит вправо. После завершения деления 32 младших бита остатка записываются в регистр reg_hi_o, а частное – в регистр reg_lo_o.

СИСТЕМНЫЙ СОПРОЦЕССОР

Системный сопроцессор расположен в модуле mips_top, его исходный код представлен в файле sys_cr.v. В функции сопроцессора входит отслеживание исключений, внешних пре-

рываний, контроль трансляции адреса, а также определение режима работы (привилегированный или пользовательский). Сопроцессор контролируется набором регистров, доступ к которым осуществляется командами MFC0 (чтение регистров) и MTC0 (запись). Помимо этого, сопроцессор декодирует команды ERET – возврат из прерывания – и SYSCALL – системный вызов – и отслеживает все нереализованные инструкции. Более подробно значения всех битов регистров описаны в [3].

Сопроцессор определяет два уровня исключений. При возникновении исключительной ситуации в нормальном режиме работы адрес возврата записывается в регистр r14_erc, а при возникновении исключения уже в процессе обработки другого исключения адрес возврата записывается в регистр r30_erroverc; при этом устанавливается бит ERL регистра статуса r12_status. Адрес вектора прерывания выбирается в зависимости от значения бита BEV регистра статуса и бита IV регистра причины r13_cause.

Рассмотрим фрагмент симуляции инструкции ADD \$4,\$2,\$3, изображённый на рисунке 6. Эта инструкция расположена в памяти по адресу 0xBFC00004 – значение регистра pc_o на стадии I конвейера – и закодирована числом 0x00432020 – значение instr_o также на стадии I. Регистры \$2 и \$3 до выполнения операции ADD (сложение с учётом знака) были инициализированы значениями 0x7FFFFFFF и 0x10. В момент времени, когда на стадии E в модуле alu начинается выполнение операции сложения, фиксируется исключение по переполнению – сигнал ioverflow_o модуля alu. Этот сигнал затем передаётся системным сопроцессором на стадию I по линии ex_branch_i. Одновременно с этим сопроцессор формирует адрес вектора прерывания, который виден на стадии I конвейера как сигнал sys_pco_i и в данном примере равен 0xBFC00380. Соответственно, на стадии I непрерывное изменение указателя инструкций pc_o заменяется скачком по вектору прерывания: вместо подготовленного

адреса 0xBFC00008, следующего за инструкцией ADD, переход происходит по адресу 0xBFC00380.

Адрес возврата при этом запоминается в регистре r14_erc. После завершения работы обработчика исключения инструкция ERET передаёт стадии I конвейера сохранённую величину программного счётчика из регистра r14_erc, восстанавливая процесс выполнения прерванной программы.

Блок контроля памяти

Блок контроля памяти представлен модулем mmu_core_top в модуле верхнего уровня процессора mips_top, исходный код находится в файле mmu_core.v. Основной задачей модуля является трансляция виртуального адреса в физический и обеспечение интерфейса с блоком управления внешней памятью.

Различают несколько сегментов памяти. Для программ в пользовательском режиме существует только один сегмент useg с диапазоном виртуальных адресов 0 – 0x7FFFFFFF, т.е. нижние 2 Гб памяти. Для привилегированных программ доступно всё адресное пространство, разделённое на следующие сегменты:

- kuseg 0 – 0x7FFFFFFF
- kseg0 0 80000000 – 0x9FFFFFFF
- kseg1 0 A0000000 – 0xBFFFFFFF
- kseg2 0 C0000000 – 0xDFFFFFFF
- kseg3 0 E0000000 – 0xFFFFFFF

Для сегментов kseg2 и kseg3 виртуальный адрес всегда равен физическому, сегменты kseg0 и kseg1 совмещены в физическом пространстве и начинаются с адреса 0. Трансляция пользовательского сегмента зависит от значения бита ERL регистра статуса. При нулевом значении этого бита сегмент useg/kuseg начинается с адреса 0x4000000, а при ERL = 1 сегмент useg/kuseg совмещается с сегментами kseg0/kseg1, т.е. начинается с нулевого адреса. Более подробно механизм трансляции адреса описан в [2].

Модуль mmu_core транслирует адрес и для данных, и для команд. Например, для данных этот процесс выглядит следующим образом:

```
case x
({va_i[31:30],r12_status_i[2]})
3'b0x1: pha_d <= va_i;
3'b0x0: pha_d <= {(va_i[31:28] +
4'h4),va_i[27:0]};
3'b10x: pha_d <=
{3'b000,va_i[28:0]};
```

```
3'b11x: pha_d <= va_i;
Endcase
```

Два старших бита виртуального адреса задают сегмент, а `r12_status_i[2]` – это бит ERL. Две первые линии оператора условия `case` задают трансляцию адреса сегмента `useg/kuseg`, которая зависит от значения бита ERL. Третья линия – для сегментов `kseg0` и `kseg1`, четвёртая линия – преобразование адресов для сегментов `kseg2` и `kseg3`. Для команд схема трансляции аналогична.

В данной модели процессора реализована модель памяти фон Неймана, в которой сегменты кода и данных совмещены. Поэтому физические адреса инструкций и данных мультиплексируются:

```
assign pha_o = (data_access) ?
pha_d : pha_ins;
```

В зависимости от типа доступа, модуль `mtu_core` выдаёт соответствующий физический адрес. Доступ данных имеет более высокий приоритет по сравнению с доступом команд, поэтому при обмене данными с внешней па-

мятью сначала читаются или записываются данные, а затем читается следующая инструкция.

Преобразованный физический адрес поступает на вход блока интерфейса с внешней памятью, представленного модулем `biu_top` в файле верхнего уровня процессора `mips_top`. Исходный код блока интерфейса находится в файле `biu.v`. Этот блок реализован в соответствии с временными диаграммами, приведёнными в [2] в разделе описания шинного интерфейса.

ЗАКЛЮЧЕНИЕ

Программируемая логика активно вытесняет жёсткую логику в различных областях электроники. Не являясь исключением и такие сложные устройства, как процессоры, которые на ПЛИС, как правило, реализуются в форме IP-ядер в составе более сложных систем на кристалле.

В данной статье была рассмотрена модель процессора MIPS32, написанная на языке Verilog и реализованная для ПЛИС семейства Xilinx Spartan 6. Эта модель является учебным примером, позволяющим рассмот-

реть принципы функционирования процессоров семейства MIPS32. Пример в дополнительных материалах к статье представляет собой простейшую систему на кристалле, включающую в себя MIPS-процессор, UART и контроллер памяти. Файлы архивов, упомянутые в статье, находятся в соответствующем разделе сайта журнала (soel.ru).

ЛИТЕРАТУРА

1. MIPS Architecture for programmers. Volume I-A: Introduction to the MIPS32 Architecture. MIPS Technologies, Inc.
2. MIPS32-4K Processor Core Datasheet. MIPS Technologies, Inc.
3. MIPS32-4K Processor Core Family Software User's Manual. MIPS Technologies, Inc.
4. Микросхема интегральная 1892ВМ2Я. Руководство пользователя. ГУП НПП «Элвис».
5. Гребенников А. HDL-реализация асинхронного приёмопередатчика. Современная электроника. 2011. № 4.
6. MIPS Assembly Language Programmer's Guide. ASM-01-DOC.
7. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set. MIPS Technologies, Inc.

