МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский университет
«Московский институт электронной техники»

Факультет электроники и компьютерных технологий (ЭКТ)
Кафедра проектирования и конструирования интегральных микросхем

Кареев Кирилл Андреевич

Бакалаврская работа
по направлению 09.03.01 «Информатика и вычислительная техника»

"Разработка RTL-описания интегрированного микропроцессорного модуля с RISC-архитектурой"

Студент _____ Кареев К.А.

Научный руководитель,
к.т.н., доцент каф. ПКИМС _____ Гусев С.В.

Москва 2016

Содержание

Часть I

Введение

Главная цель моего дипломного проекта - создание процессора, пригодного для изучения программирования машинных кодов и общего процессоростроения. Для этого процессор должен удовлетворять следующим критериям:

- Простота работы с машинным кодом и ассемблерным представлением.
- Единая внутренняя структура.
- Минимальное количество состояний.
- Открытость RTL-описания.

Для начала следует примерить на роль такого «учебного» процессора какой-нибудь из существующих. Было проведено некоторе исследование, в результате которого были выделены следующие процессорные системы и выявлены недостатки, котоые мешают эти системам удовлетворять заданным критериям:

1. ARM Thumb1:
    - Сложность бинарного представления машинного кода (из-за упора на уменьшенный размер).
    - Работа с дробными частями машинного слова.
    - Сложность работы с ассемблерным представлением кода (следствие функциональной простоты).
2. OpenRISC 1000 (mor1kx):
    - Наличие большого количества состояний процессора.
    - Сложность RTL-описания, в основном из-за высокой функциональной развитости.

3. MIPS32:

    - Относительно сложное построение инструкции
    - Большинство реализаций не совместимы друг с другом

В результате было принято решение создать собственную процессорную систему с нестандартным набором инструкций.

Часть II

Реализация

1 Строение ядра

Ядро процессора - главная структура, в которой заключена вся логика его работы. Сюда входит конвейер, регистровый файл, оперативная память и адаптер к шине периферических устройств. Ядро процессора УП-1 обладает следующими свойствами:

- 32-битная архитектура
- Набор из 35 (заложено до 128) инструкций
- 32 РОН (Регистра общего назначения) шириной 32 бита с четырёхпортовым интерфейсом (2 чтение, 2 запись + особые линии для PC и LR)
- Регистры PC и LR (счётчик инструкций и адрес возврата) также являются общими (31 и 29 соответственно)
- Однотактовый умножитель с возможностью сохранения всего результата (2 слова)
- Однотактовый комбинированный регистр быстрого сдвига (циклический, арифметический и логический сдвиги)
- Комбинированный однотактовый полный сумматор-вычитатель.
- Раздельные шины памяти и периферический устройств
- 16 кодов условного исполнения
- Четырёхшаговая архитектура конвейера (Декодирование, Исполнение, Память/Периферия и Регистры)
- Двухпортовое однотактовое ОЗУ ёмкостью 4 КБ (1 Кс) (1- чтение, 1 - запись)

Схема построения ядра представлена на рисунке 1

Главная логика исполнения инструкций содержится в конвейере. Конвейер построен по типовой[design] для RISC процессоров пятистадийной схеме. Однако, в процессоре УП-1 отсутствует выделенная логика получения инструкций от ПЗУ, чем и объясняется наличие только четырёх стадий на схеме ядра. Стадия Decode выполняет роль декодера инструкций, а также подготавливает все необходимые данные для успешного их исполнения. Стадия Execute содержит основную вычислительную логику, а также блок вычисления условных кодов. Стадия Memory/Periph является интерфейсом между ядром и шинами памяти и периферии. Стадия Register WB сохраняет результаты исполнения и завершает конвейер.

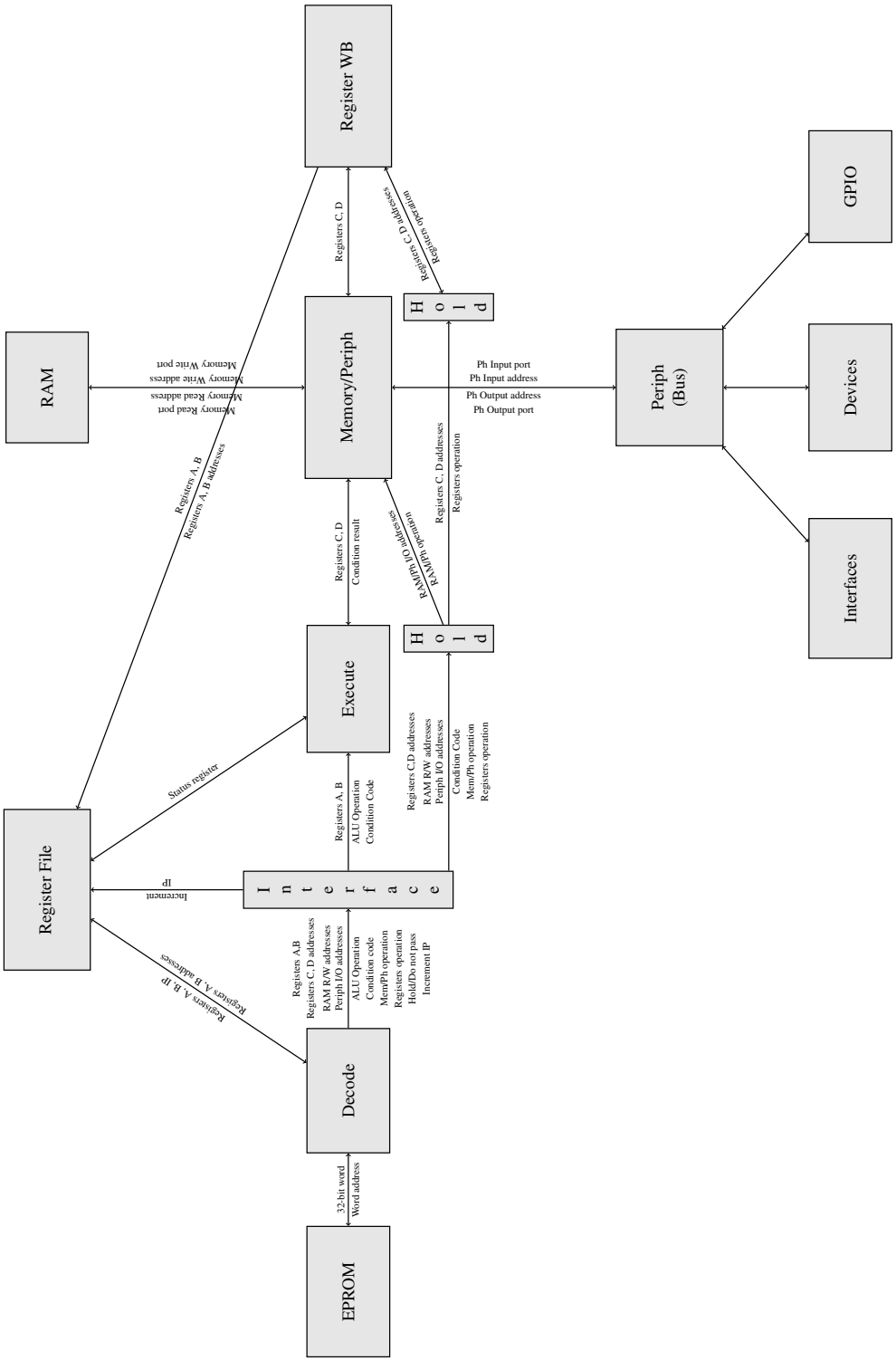


Рис. 1: Ядро (схема)

2 Конвейер

2.1 Назначение стадий

Конвейер процессора состоит из четырёх стадий, одной «невидимой» стадии и набора подстадий:

1. Decode. Получает от ПЗУ (по адресу в pc) инструкцию и подготавливает её к исполнению на остальных стадиях. Для этих целей стадия подготавливает управляющие сигналы для каждой из трёх последующих стадий и помещает их в следующую стадию. Также в этой стадии находится блок обработки ошибок конвейера, который исключает возможность чтения «не готовых» данных из регистров.

2. Interface - Вспомогательная стадия, служит для равномерного распределения сигналов по стадиям и подстадиям. Работает синхронно со стадией Decode для обеспечения наивысшей производительности. Из-за такого поведения является «невидимой»

3. Execute. В этой стадии располагается АЛУ, которое и выполняет основную часть вычислений. Также здесь происходит вычисление флагов исполнения и подготовка на основе флагов результатов исполнения услоных кодов. Управляющие сигналы для оставшихся двух стадий помещаются в подстадию Hold.

4. Memory/Periph. Данная стадия является единственной точкой входа-выхода для ОЗУ и периферийных устройств. Благодаря этому отсутствует необходимость в обработке ошибок конвейера по ОЗУ и периферии. В этой стадии происходит запись и чтение ОЗУ и периферийных регистров. Сигналы для последней стадии задерживаются на подстадии Hold

5. Register WB. Данная стадия производит запись результатов выполнения всех стадий в регистровый файл. Так как эта стадия является продуктом разделения операций чтения и записи в регистры, она также

является причиной внесения в стадию decode блока разрешения ошибок конвейера.

2.2 Стадия «Decode»

Декодер работает по следующему принципу:

1. Получает инструкцию и разделяет ещё на исполняемые части согласно схеме инструкции (см. Приложение 1)
2. Генерирует начальные управляющие сигналы для основных исполняющих блоков в соответствии с номером инструкции (АЛУ, память, регистры)
3. Производит получение содержимого регистров, указанных в инструкции, если необходимо.
4. В случае присутствия в инструкции флагов наличия мгновенных значений, производит постановку задержки исполнения, и во время этой задержки производит получение мгновенных значений из ПЗУ
5. В случае исполнения т.н. «длинных» инструкций (инструкции, занимающие больше 1 такта, например инструкции перехода) производит постановку задержки, равной времени исполнения инструкции
6. В случае присутствия ошибки конвейера, производит постановку задержки и запрещает инкремент счётчика инструкций до тех пор, пока сигнал ошибки не вернётся в единицу.

2.3 Стадия «Interface»

Интерфейс является «ширмой» между декодером и остальными стадиями.

Специальный сигнал d_pass позволяет подменить операцию, хранящуюся в нем на nop, что очень удобно для постановки всяческого рода задержек. Задержка срабатывания этой стадии подобрана таким образом, чтобы она

(стадия) срабатывала одновременно со стадией декодера, что уменьшает эффективную длину конвейера, а значит и задержку срабатывания инструкций, требующих полного сброса конвейера.

Также интерфейс распределяет управляющие сигналы по соответствующим стадиям и подстадиям.

2.4 Стадия «Execute»

Стадия исполнения производит все заявленные в наборе инструкций вычисления. Внутри этой стадии находятся два блока:

1. Блок АЛУ - основная вычислительная сила процессора.
2. Блок условного исполнения - блок, производящий вычисление условного результата (cres) исходя из входного условного кода и флагов исполнения.

Входными для данной стадии являются следующие сигналы:

- a и b - входные операнды, без изменений проводятся к АЛУ
- alu_op - управляющий кода АЛУ, проводится к нему без изменений
- st - регистр статуса - регистр, содержащий флаги исполнения. Применяется в вычислении условного результата
- cond - условный код.
- is_cond - сигнал, определяющий необходимость вычисления условного результата. В случае, когда этот сигнал равен нулю, условный результат принудительно выставляется в единицу
- write_flags - сигнал, определяющий флаги, которые будут перезаписаны текущей инструкцией

Стадия генерирует следующие сигналы:

- r1 и r2 - результаты вычислений (из АЛУ)

- n, z, c, v - флаги, сгенерированные АЛУ
- cres - условный результат
- cc - сигнал, определяющий необходимость записи флагов в регистр st

2.5 Стадия «Memory/Periph»

Эта стадия является точкой входа/выхода для операций с ОЗУ и периферийными устройствами. Управляется эта стадия специальными командными сигналами r1_op и r2_op, для каждого входного операнда свой код управления. Кроме них, также используются следующие сигналы:

1. r1 и r2 - входные операнды, приходят из стадии исполнения
2. a1 и a2 - адресные операнды, заполняются на стадии декодирования.
3. proceed - сигнал условного результата. Если он равен нулю, то командные сигналы принудительно выставляются в «сквозной NOP»
4. ram_r_line и sys_r_line - линии чтения ОЗУ и периферии соответственно.

Также эта стадия генерирует следующие сигналы:

1. m1 и m2 - выходные операнды
2. ram_w_line, sys_w_line, ram_w_addr, sys_w_addr etc. - линии управления ОЗУ и периферией соответственно

Набор команд следующий:

0: Чистый NOP. Никаких операций не производится. В выходной операнд записывается 0

1: Сквозной NOP. Входной операнд просто копируется в выходной без изменений

2: Чтение из ОЗУ по адресу a1

3: Чтение из ОЗУ по адресу a2

4: Чтение из ОЗУ по адресу в другом операнде

5: Запись в ОЗУ по адресу a1

6: Запись в ОЗУ по адресу a2

7: Запись в ОЗУ по адресу в другом операнде

8: Чтение из периферии по адресу a1

9: Чтение из периферии по адресу a2

10: Чтение из периферии по адресу в другом операнде

11: Запись в периферию по адресу a1

12: Запись в периферию по адресу a2

13: Запись в периферию по адресу в другом операнде

14: Копирует входной операнд в противоположный выходной.

2.6 Стадия «Register WB»

Данная стадия производит сохранение результата, т.е. обратную запись в регистровый файл. Эта стадия также управляется специальным командным сигналом op. Помимо него, также используются следующие сигналы:

- r1 и r2 - входные операнды.
- a1 и a2 - адреса для записи, заполняются декодером.
- proceed - сигнал условного результата. Если он равен нулю, то командный сигнал принудительно переключается в NOP

Выходные сигналы этой стадии контролируют порты записи регистрового файла.

Набор команд представлен следующим образом:

0: NOP, записи не происходит

1: Запись r1 по адресу a1

2: Запись r1 по адресу a2

3: Запись r1 по адресу в r2

4: Запись r2 по адресу a1

5: Запись r2 по адресу a2

6: Запись r2 по адресу в r1

7: Запись r1 по адресу a1 и r2 по адресу a2

7: Запись r1 по адресу a2 и r2 по адресу a1

2.7 Ошибки конвейера

Ошибки конвейера обнаруживаются специальным блоком. Принцип его действия состоит в том, чтобы проверить выходные сигналы регистровой записи каждой стадии и подстадии, обнаружить среди них сигналы активной записи и произвести сравнение адресов назначения при этих сигналах с адресами текущей инструкции в декодере. В случае совпадения сгнал ошибки конвейера выставляется в единицу, и декодер приостанавливает выполнение инструкции пока сигнал не упадёт обратно в ноль (то есть пока запись не произойдёт).

3 АЛУ

3.1 Строение АЛУ

АЛУ разделён на пять основных блоков:

1. Декодер инструкций и селектор результатов/флагов

2. Комбинированный сумматор-вычитатель

3. Комбинированный регистр быстрого сдвига-вращения
4. Полный умножитель
5. Блок побитовых инструкций

Схема соединения блоков представлена на рисунке 2

На входе АЛУ присутствуют следующие сигналы:

1. a и b - входные операнды
2. op - управляющий сигнал

АЛУ генерирует следующие сигналы:

1. q1 и q2 - выходные операнды
2. st - выходные флаги исполнения

Следует также заметить, что АЛУ является комбинаторным блоком, то есть работает без внешней синхронизации

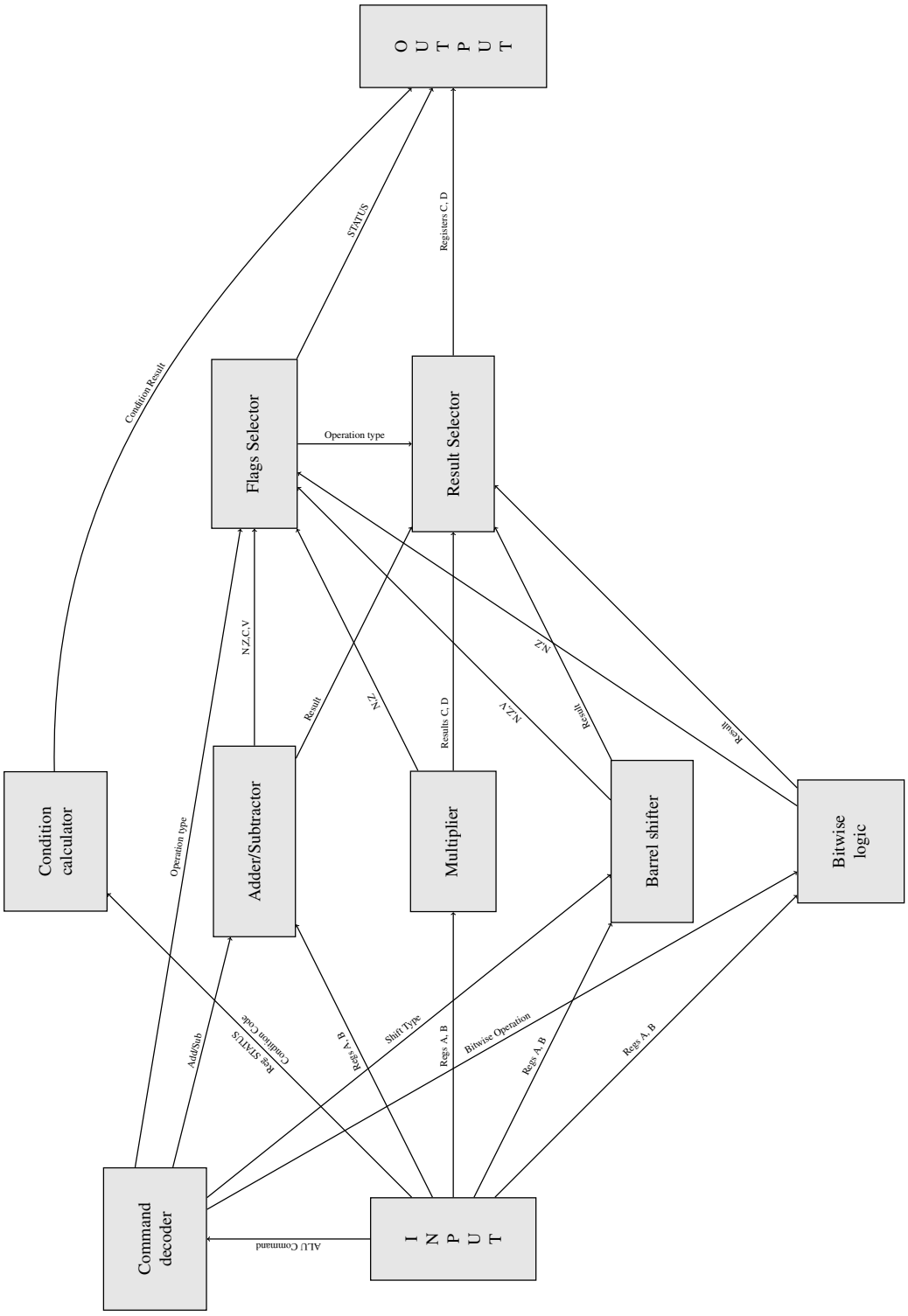


Рис. 2: АЛУ (схема)

3.2 Сумматор/Вычитатель

Сумматор-вычитатель построен по схеме сумматора с параллельным переносом [cla]. Состоит из следующих исходных блоков:

1. fa_pg - полный сумматор, модифицированный для генерации сигналов Propagate и Generate вместо сигнала переноса
2. cla4 - четырёхбитный сумматор с параллельным переносом. Состоит из четырёх модифицированных полных сумматоров и логики распространения переноса
3. cla16 - 16-битный сумматор, состоит из четырёх четырёбитных и аналогичной логики распространения переноса.
4. cla32 - 32-битный сумматор, конечный продукт, составлен из двух шестнадцатибитных и упрощённой логики распространения переноса.

При вычитании в схему вносятся следующие изменения:

1. На пути второго операнда встаёт блок побитовой инверсии
2. Сигнал нулевого переноса устанавливается в единицу

Данный блок способен генерировать все четыре флага исполнения.

3.3 Комбинированный регистр быстрого сдвига/вращения

Данный блок построен по схеме реверсивного сдвигового регистра, основанного на операции маскирования, представленной в [shifter]. Данная схема позволяет производить все возможные виды сдвигов и вращений (кроме, возможно, операций через бит переноса) за один такт. Управляется эта схема с помощью тройки сигналов {left, rotate, arith} следующим образом:

000: Логический сдвиг вправо

001: Арифметический сдвиг вправо

01X: Циклический сдвиг (вращение) вправо

100: Логический сдвиг влево

101: Арифметический сдвиг влево

11X: Циклический сдвиг (вращение) влево

Арифметический сдвиг отличается от логического тем, что сохраняет знаковый бит операнда. Также арифметический сдвиг влево может, в отличие от остальных сдвигов, генерировать флаг переполнения. Все виды сдвигов могут генерировать флаг нулевого результата

3.4 Умножитель

Данный умножитель является полным параллельным умножителем, построенным по схеме дерева Дадды [dadda]. Построением таких умножителей занимается программа MultiplierGenerator. Алгоритм построения следующий:

1. Перемножить (логическое И) каждый бит первого результата с каждым битом второго, с получением n^2 частичных произведений с разным весом.

2. Уменьшить количество частичных произведений по следующим правилам:

   (a) Взять любые три бита с одним весом и пропустить через полный сумматор. В результате получится один бит с текущим весом и один - с весом на единицу больше

   (b) Если осталось только два бита одного веса, и выходных бит с таким весом равно 1 или 2 по модулю 3, пропустить их через полусумматор, иначе - пробросить на следующий слой без изменений

   (c) Если остался только один - пробросить его на следующий слой без изменений

3. Сгруппировать результат в два числа и просуммировать обыкновенным полным сумматором.

Так как результат умножения в два раза шире его операндов, был предусмотрен механизм разделения результата на два слова и перегрузки их в два регистра.

Данный блок может выставлять флаг переполнения (при ненулевом старшем слове) и флаг нулевого результата (при нулевом младшем слове)

3.5 Блок побитовых операций

Данный блок принимает на вход один-два операнда (A и B соответственно, в зависимости от вида операции) и преобразует их согласно управляющему сигналу следующим образом:

000: $Q = \overline{A}$ (Инверсия A)

001: $Q = A \wedge B$ (A И B)

010: $Q = A \vee B$ (A ИЛИ B)

011: $Q = A \veebar B$ (A ИСКЛ. ИЛИ B)

100: $Q = \overline{A \wedge B}$ (A И-НЕ B)

101: $Q = \overline{A \vee B}$ (A ИЛИ-НЕ B)

110: $Q = \overline{A \veebar B}$ (A ИСКЛ. ИЛИ-НЕ B)

111: $Q = \overline{B}$ (Инверсия B)

Данный блок может генерировать только флаг нулевого результата

3.6 Декодер команд

Декодер команд выполняет роль объединителя всех блоков АЛУ и селектора нужного результата. В соответствии со значением сигнала alu_op будет выполняться следующая операция:

0x00: NOP - входные операнды без изменений копируются в выходные

0x01: ADD - $q_1 = a + b$, $q_2 = 0$

0x02: SUB - $q_1 = a - b$, $q_2 = 0$

0x03: CPL - $q_1 = -a$, $q_2 = 0$

0x04: MUL - $\{q_2, q_1\} = a \cdot b$

0x05: SHR[1] - $q_1 = a\ shr\ b$, $q_2 = 0$

0x06: SHL[2] - $q_1 = a\ shl\ b$, $q_2 = 0$

0x07: SAR[3] - $q_1 = a\ sar\ b$, $q_2 = 0$

0x08: SAL[4] - $q_1 = a\ sal\ b$, $q_2 = 0$

0x09: ROR[5] - $q_1 = a\ ror\ b$, $q_2 = 0$

0x0A: ROL[6] - $q_1 = a\ rol\ b$, $q_2 = 0$

0x0B: NOT - $q_1 = \overline{a}$, $q_2 = 0$

0x0C: AND - $q_1 = a \wedge b$, $q_2 = 0$

0x0D: OR - $q_1 = a \vee b$, $q_2 = 0$

0x0E: XOR - $q_1 = a \veebar b$, $q_2 = 0$

0x0F: NAND - $q_1 = \overline{a \wedge b}$, $q_2 = 0$

0x10: NOR - $q_1 = \overline{a \vee b}$, $q_2 = 0$

0x10: XNOR - $q_1 = \overline{a \veebar b}$, $q_2 = 0$

[1] Логический сдвиг вправо
[2] Логический сдвиг влево
[3] Арифметический сдвиг вправо
[4] Арифметический сдвиг влево
[5] Циклический сдвиг вправо
[6] Циклический сдвиг влево

4 Память

4.1 Виды памяти

В ядре процессора присутствует три вида памяти:

1. Регистровый файл
2. Оперативная память
3. Программная память

Самой быстрой среди них является регистровая. Программная является неперезаписываемой и в данном случае не рассматривается.

4.2 Регистровый файл

В ядре присутствует регистровый файл на 32 регистра шириной 32 бита и четырьмя портами (два порта на чтение, два - на запись).

Чтение регулируется сигналом read следующим образом:

1. Если соответствующий бит сигнала равен единице, то на эту линию асинхронно выставляется содержимое регистра по адресу, заданному на адресной линии данного порта
2. Иначе на эту линию выставляется состояние Z

Запись регулируется похожим образом, различие в том, что запись - процесс синхронный.

Также организованы слеующие внеочередные вводы-выводы:

1. Регистр 28 (st) имеет собственный ввод, вывод и сигнал записи
2. Регистр 29 (lr) имеет собственный вывод
3. Регистр 31 (pc) имеет собственный вывод и логику инкрементирования.

4.3 ОЗУ

В ядре находится двухпортовая ОЗУ немедленного действия (1 - чтение, 1 - запись). Чтение регулируется сигналом read, запись - сигналом write в манере, похожей на чтение/запись в регистровом файле. Использование z-состояния в неактивном режиме позволяет упростить объединение нескольких однотипных блоков ОЗУ при расширении памяти.

Следует также заметить, что в отличии от регистрового файла, в ОЗУ обе операции (чтение и запись) синхронные.

5 Периферия

5.1 Строение шины

Все периферические устройства в данной системе подключены к шине периферийных устройств. Она представляет собой параллельную внутреннюю шину с multidrop топологией и двумя раздельными линиями приёма/передачи - одна линия «записи», одна - «чтения». В каждой линии передаются параллельно адрес и данные, а также ассоциированный с данной линией сигнал (т.е. сигналы записи и чтения). По своему строению шина поддерживает любые MultiMaster - MultiSlave конфигурации, однако в данном процессоре единственным мастером является стадия «Memory/Periph» конвейера, а периферийные устройства являются подчинёнными. Подразумевается, что при заполнении пула устройств каждому из них (в т.ч. каждому из его регистров, если их несколько) назначается уникальный адрес, для устранения возможных коллизий на шине.

На данный момент в процессоре присутствуют следующие устройства:

- Выходной мультиплексор пинов на 4 функции
- Контроллер GPIO

5.2 Выходной мультиплексор

Данное устройство призвано обеспечить многофункциональность каждого пина процессора, путём возможности мультиплексирования на один пин до четырёх различных функций. Эта цель достигается путём назначения на каждый из четырёх входов модуля мультиплексора функции ввода (чтения с ноги) и вывода (установки уровня на ноге) и определения текущей функции ноги во внутреннем регистре.

На шину периферийных устройств, на линии чтения и записи мультиплексор выставляет два регистра, которые являются частями одного 64-битного регистра control. Младший адрес (самый младший бит равен нулю) ссответствует младшей части регистра, старший (самый младший бит равен единице) - старшей части. Каждые два бита этого регистра (начиная с самого младшего бита) управляют функцией каждой ноги, подключенной к этому мультиплексору (начиная с самой первой) следующим образом:

00: Выбор первой функции

01: Выбор второй функции

10: Выбор третьей функции

11: Выбор четвёртой фукции

Переключение функции ноги происходит незамедлительно, т.е. сразу после записи в регистр control.

В текущей версии сборки процессора присутствует 128 ног, на каждой по мультиплексору, что означает присутствие четырёх блоков выходных мультиплексоров на 32 ноги каждый.

5.3 Контроллер GPIO

Данное устройство призвано обеспечить базовый универсальный контроль над всеми пинами процессора. Эта цель достигается путём предоставления регистров, подключенных непосредственно к путям управления и считывания состояния пинов.

На шину периферийных устройств данный контроллер выставляет два регистра:

1. direction. Располагается в старшем регистре. Задаёт направление данных на пинах. Каждый бит ассоциирован с одной ногой. Значение «0» определяет ногу как «Вход», т.е. переключает её в высокоинмедансное состояние, в котором она готова для чтения; Значение «1» определяет ногу как «Выход», т.е. её состояние определяется значением в регистре value

2. value. Располагается в младшем регистре. При записи определяет состояние ноги в случае настройки её на выход; При чтении возвращает текущее состояние ноги. Каждый бит также ассоциирован с одной ногой.

Следует также заметить, что при попытке чтения ноги с состоянием «Выход» корректность и действительность возвращаемого значения не гарантируется, однако в *большинстве* случаев будет возвращено её текущее состояние.

В текущей версии сборки процессора контроллеры GPIO подключены в качестве первой функции для всех ног.

5.4 Адресация

В настоящей версии сборки процессора устройства распределены по адресам следующим образом:

00000 - 00001: Пусто (защита от случайной перезаписи)

00010 - 00011: Мультиплексор на ноги 0-31

00100 - 00101: Мультиплексор на ноги 63-32

00110 - 00111: Мультиплексор на ноги 95-64

01000 - 01001: Мультиплексор на ноги 127-96

01010 - 01011: Контроллер GPIO на первый мультиплексор (ноги 0-31)

01100 - 01101: Контроллер GPIO на второй мультиплексор (ноги 63-32)

01110 - 01111: Контроллер GPIO на третий мультиплексор (ноги 95-64)

10000 - 10001: Контроллер GPIO на четвёртый мультиплексор (ноги 127-96)

Часть III

Результаты

6 Симуляция

6.1 Средства симуляции

Симлуяция проводится средствами программы IcarusVerilog. Для тестирования были созданы две программы:

- Программа «Тест», она же тестовая программа. Была создана для проверки работоспособности всех блоков процессора. Эта программа написана таким образом, что люба ошибка, влияющая на конечный результат хотя бы одной операции вызывает существенные изменения в потоке исполнения программы, что очень легко обнаружить не прибегая к анализатору временных диаграмм, прямо в статистике работы симулятора. Такой подход меньшил время подстройки блоков процессора

- Программа «Фибоначчи». Классическая программа, призванная продемонстрировать процессы, происходящие в процессоре, а также полноту по Тьюрингу его набора инструкций. Такая программа существует для всех процессорных систем, и хорошо зарекомендовала себя для демонстрационных целей.

Программы создавались в виде отдельных модулей по принципу параллельной конструкции case. Такой метод был выбран для упрощения и оптимизации работы с разрежённым кодом, коим являются обе тестовых программы.

Результаты моделирования были представлены программой IcarusVerilog в виде дампа временных диаграмм в формате FST/ Эти диаграммы были проинспектированы и выведены в графический формат с помощью программы GTKWave. Результаты в графическом формате представлены для каждой программы в разделе «Временные диаграммы».

6.2 Тестовая программа

6.2.1 Описание

Данная программа производит базовое тестирование всех блоков процессора. Алгоритм действий следующий:

1. Проинициализировать регистры 29 и 30 значениями 14888h и 22888h
2. Суммировать эти регистры в регистр 30
3. Суммировать 35942h и DEADBEAFh
4. Перемножить регистры 29 и 30 в них же
5. ИСКЛ. ИЛИ этих регистров с сохранением в тридцатый
6. Циклический сдвиг содержимого 30-го регистра на 11 бит в 29-й
7. Безусловный переход по адресу 132h
8. (смещение 132h)
9. Записать на шину регистры 29 и 30 в прямом и обратном порядке
10. Вызов процедуры по адресу регистре 30
11. Записать в ОЗУ содержимое регистра 30 по адресу 16
12. Переставить регистры 29 и 30
13. Любая операция (здесь, запись на шину)
14. Прочитать ОЗУ по адресу 16 в регистр 30
15. Настроить GPIO0 на чтение, GPIO1 на вывод
16. Вывести на GPIO1 единицы
17. Считать GPIO0 в регистр 30
18. (смещение 5E771E7Dh)

19. Если флаг N не стоит - переход по адресу в регистре 0
20. Иначе - возврат

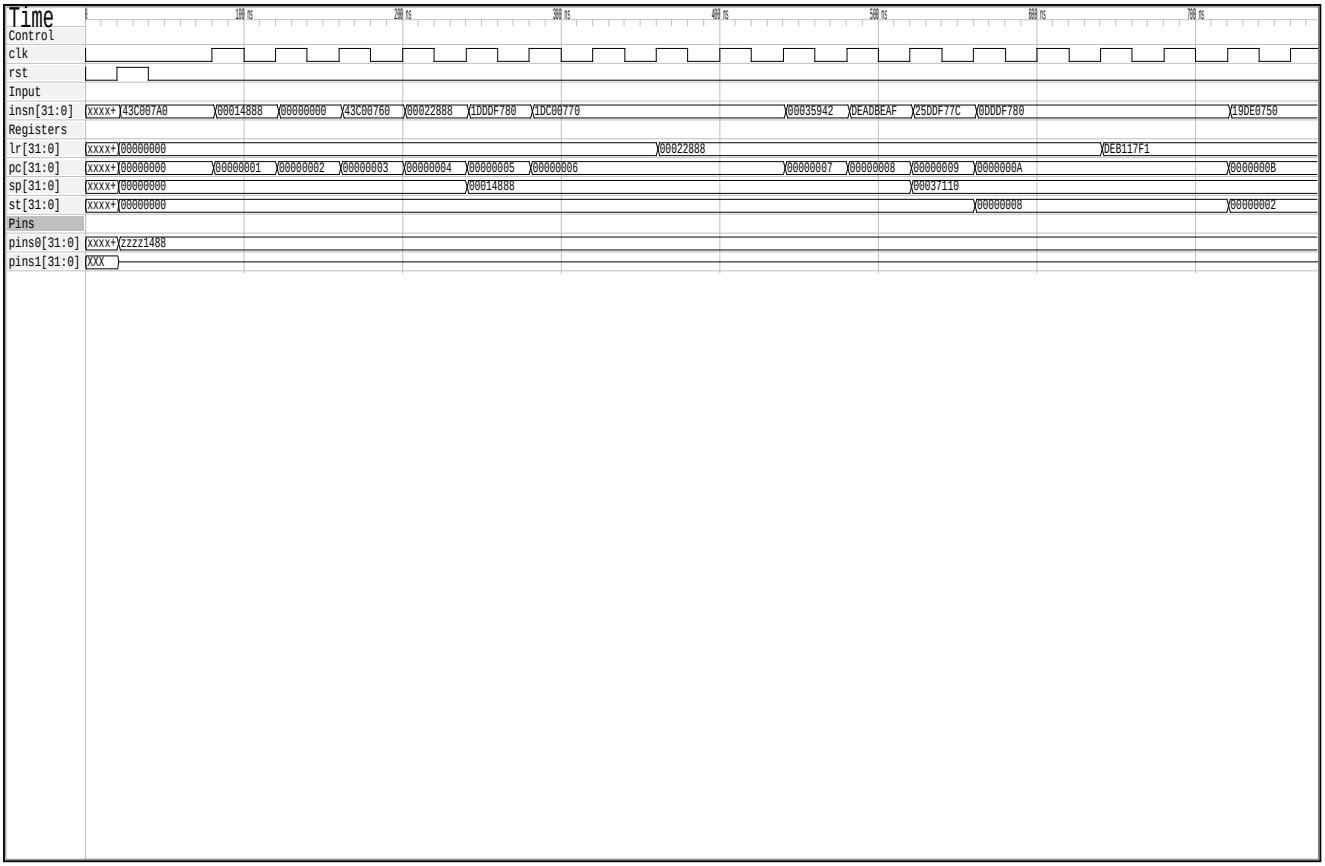
6.2.2 Исходный код

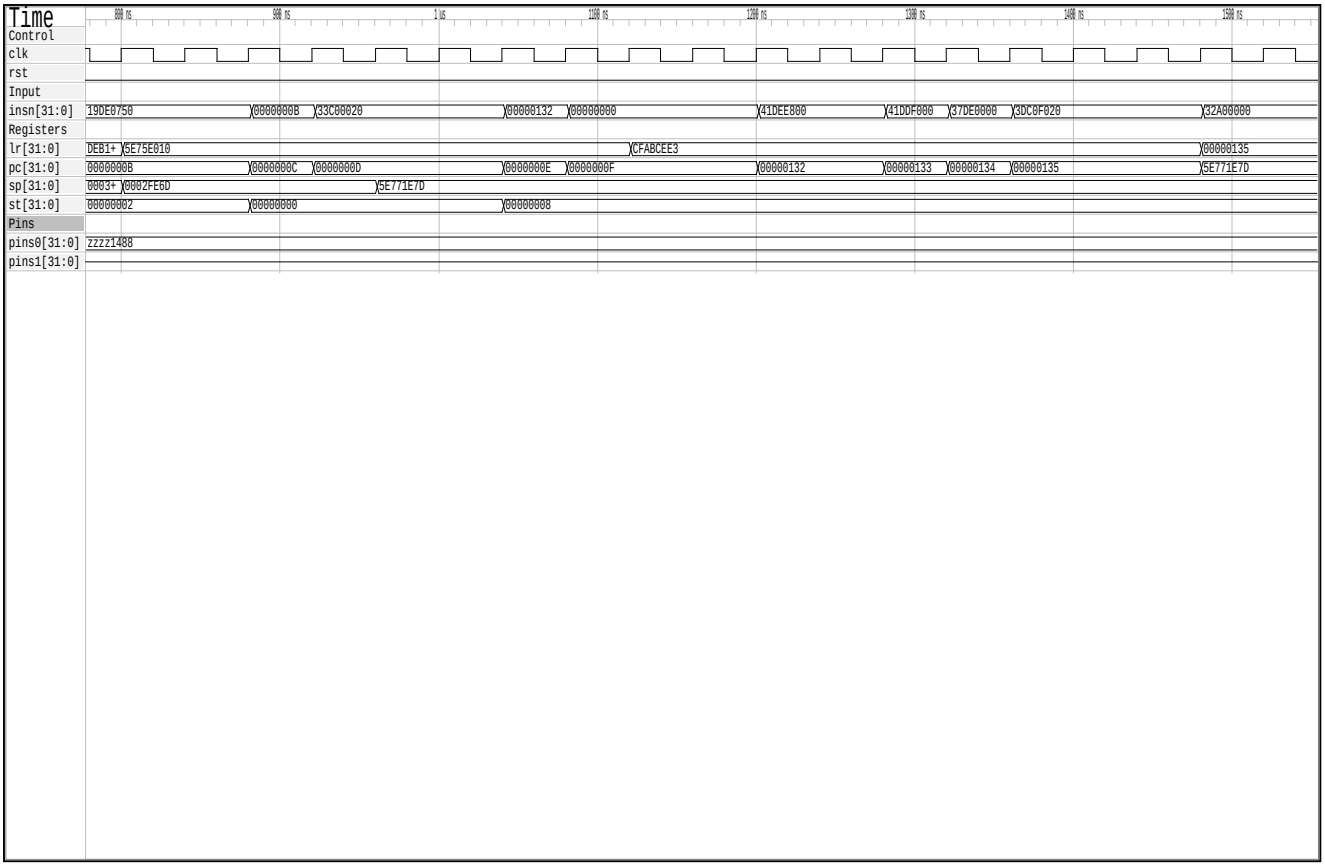
```
[0x00000000]:
movs 0x14888 -> r30
movs 0x22888 -> r29
add r29, r30 -> r30
add 0x35942, 0xDEADBEAF -> r29
mul r29, r30 -> r29, r30
xor r29, r30 -> r30
csr r30, 0x0B -> r29
br  0x132
(nop)
[0x00000132]:
out r29 -> [r30]
out r30 -> [r29]
brl r30
str r30 -> 0x10
mov r29, r30 -> r30, r29
out r30 -> [r29]
ldr 0x10 -> r30
movs 0xFFFFFFFF -> r1
out r1 -> 0x0D
out r1 -> 0x0F
out r1 -> 0x11
out r1 -> 0x0E
in  0x0A -> r30
(nop)
[0x5E771E7D]:
br_pos r0
ret_neg
```

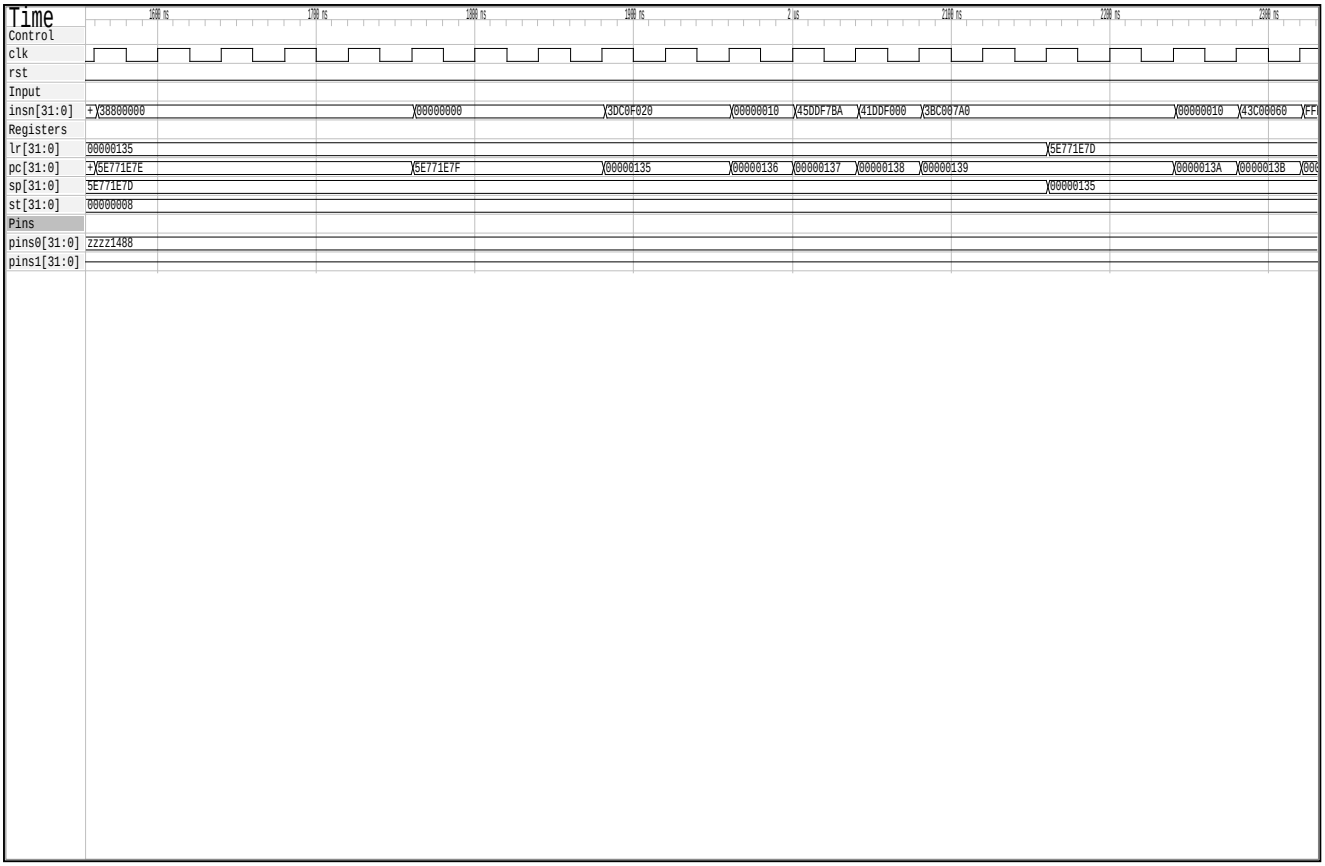
```
(nop)
```

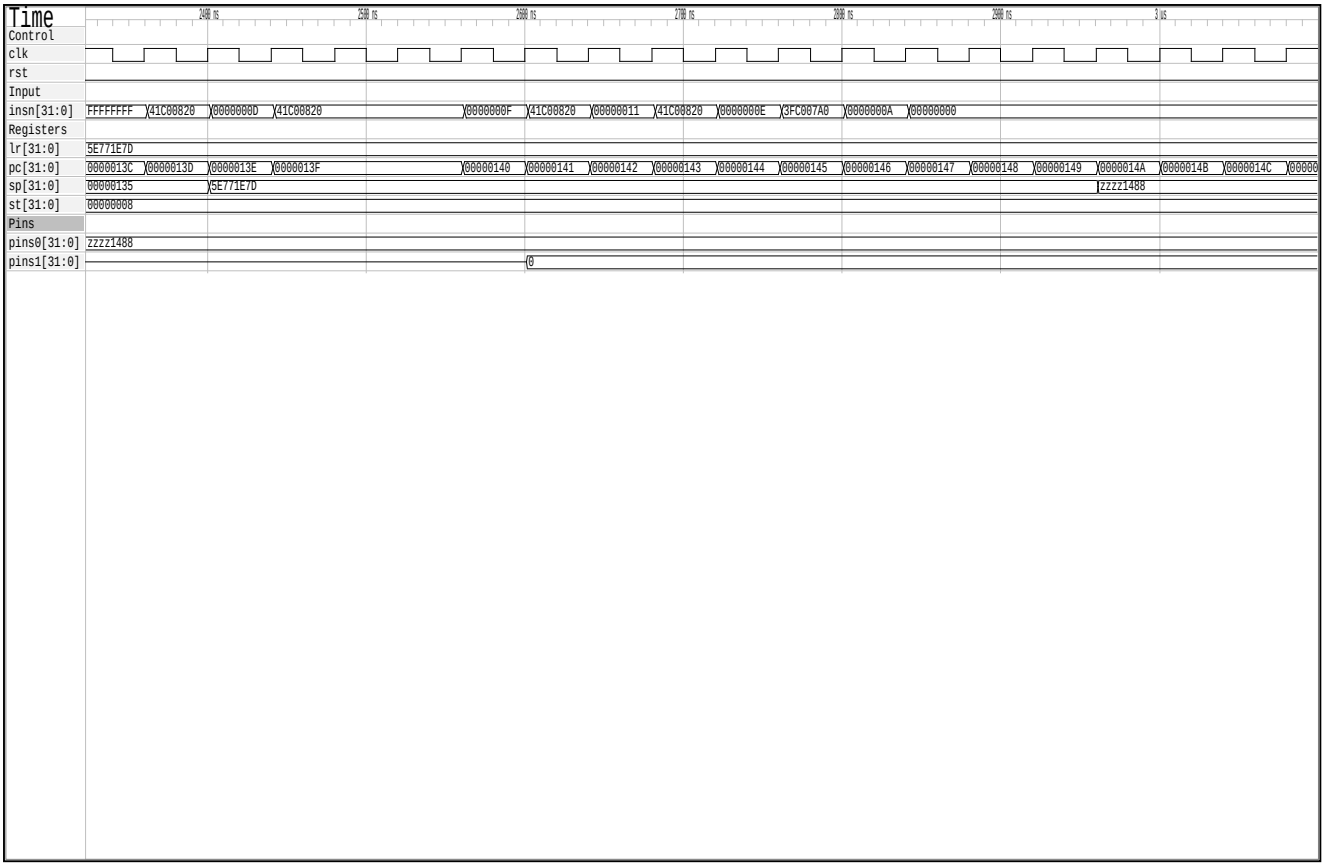
6.2.3 Временные диаграммы

На изображениях - результат выполнения тестовой программы (полностью)


Time
0
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
Control
clk
rst
Input
insn[31:0]
xxxx
43C007A0
00014888
00000000
43C00760
00022888
1DDDF780
1DC00770
00035942
DEADBEAF
25DDF77C
0DDDF780
19DE0750
Registers
lr[31:0]
xxxx
00000000
00022888
DEB117F1
pc[31:0]
xxxx
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
sp[31:0]
xxxx
00000000
00014888
00037110
st[31:0]
xxxx
00000000
00000008
00000002
Pins
pins0[31:0]
xxxx
zzzz1488
pins1[31:0]
XXX


Time
800 ns
900 ns
1 us
1100 ns
1200 ns
1300 ns
1400 ns
1500 ns
Control
clk
rst
Input
insn[31:0]
19DE0750
0000000B
33C00020
00000132
00000000
41DEE800
41DDF000
37DE0000
3DC0F020
32A00000
Registers
lr[31:0]
DEB1+
5E75E010
CFABCEE3
00000135
pc[31:0]
0000000B
0000000C
0000000D
0000000E
0000000F
00000132
00000133
00000134
00000135
5E771E7D
sp[31:0]
0003+
0002FE6D
5E771E7D
st[31:0]
00000002
00000000
00000008
Pins
pins0[31:0]
zzzz1488
pins1[31:0]


Time
1696 ns
1798 ns
1898 ns
1998 ns
2 us
2104 ns
2206 ns
2306 ns
Control
clk
rst
Input
insn[31:0]
38800000
00000000
3DC0F020
00000010
45DDF7BA
41DDF000
3BC007A0
00000010
43C00060
Registers
lr[31:0]
00000135
5E771E7D
pc[31:0]
5E771E7E
5E771E7F
00000135
00000136
00000137
00000138
00000139
0000013A
0000013B
sp[31:0]
5E771E7D
00000135
st[31:0]
00000808
Pins
pins0[31:0]
zzzz1488
pins1[31:0]


Time
2400 ns
2500 ns
2600 ns
2700 ns
2800 ns
2900 ns
3 us
Control
clk
rst
Input
insn[31:0]
FFFFFFFF
41C00820
0000000D
41C00820
0000000F
41C00820
00000011
41C00820
0000000E
3FC007A0
0000000A
00000000
Registers
lr[31:0]
5E771E7D
pc[31:0]
0000013C
0000013D
0000013E
0000013F
00000140
00000141
00000142
00000143
00000144
00000145
00000146
00000147
00000148
00000149
0000014A
0000014B
0000014C
sp[31:0]
00000135
5E771E7D
zzzz1488
st[31:0]
00000008
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
0

6.3 Программа «Фибоначчи»

6.3.1 Описание

Программа вычисляет первые 47 чисел последовательности Фибоначчи. Последовательнось Фибоначчи F задаётся следующим образом:

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = 0$$
$$F_1 = 1$$

Вычисленный член последовательности выводится на ноги чипа GPIO1 (ноги 63..31). При попытке вычисления 48-го члена последовательности (который уже не помещается в нативный 32-битный тип, а значит выставляет флаг C) программа перезапускается.

Алгоритм действий следующий:

1. Проинициализировать регистры:
   (a) Нулевой - нулями
   (b) Первый - единицами
   (c) Второй - 0h (Нулевое число Фибоначчи)
   (d) Третий - 1h (Первое число Фибоначчи)
   (e) Пятый - Ch (адрес регистра value чипа gpio1)
   (f) Шестой - 100h (смещение процедуры fib())
   (g) Седьмой - -0x03 (относительно смещение в цикле)
2. Настроить GPIO1 на вывод и вывести первое число Фибоначчи
3. (начало цикла) Вызов процедуры fib()
4. Если переполнения нет - Вывести полученное число на GPIO1
5. Если переполнения нет - Перейти в начало цикла

6. Иначе перейти в начало программы
7. (смещение 100h - fib())
8. Суммировать второй и третий регистр в четвёртый
9. Сместить третий и четвёртый регистр во второй и третий соответственно
10. Возврат

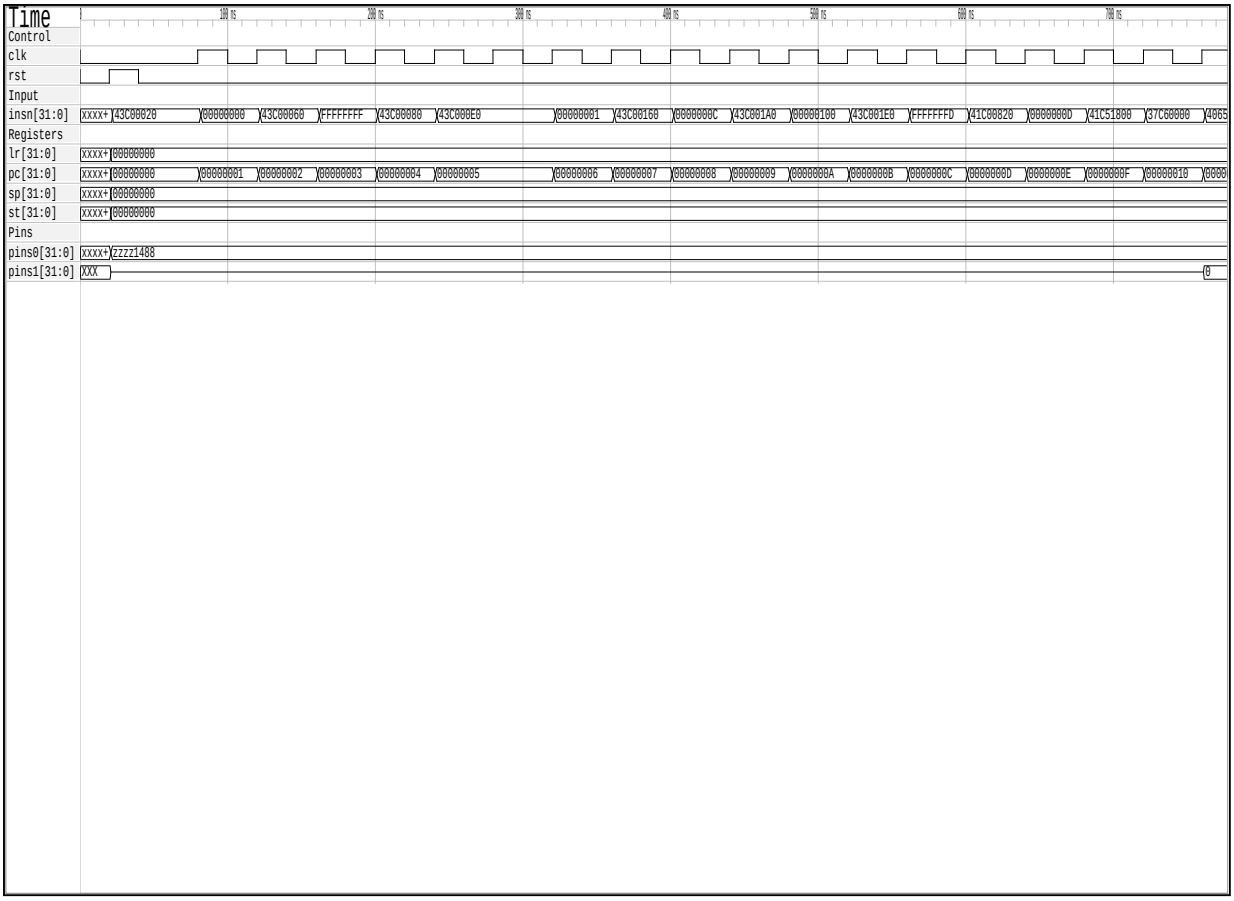
6.3.2 Исходный код

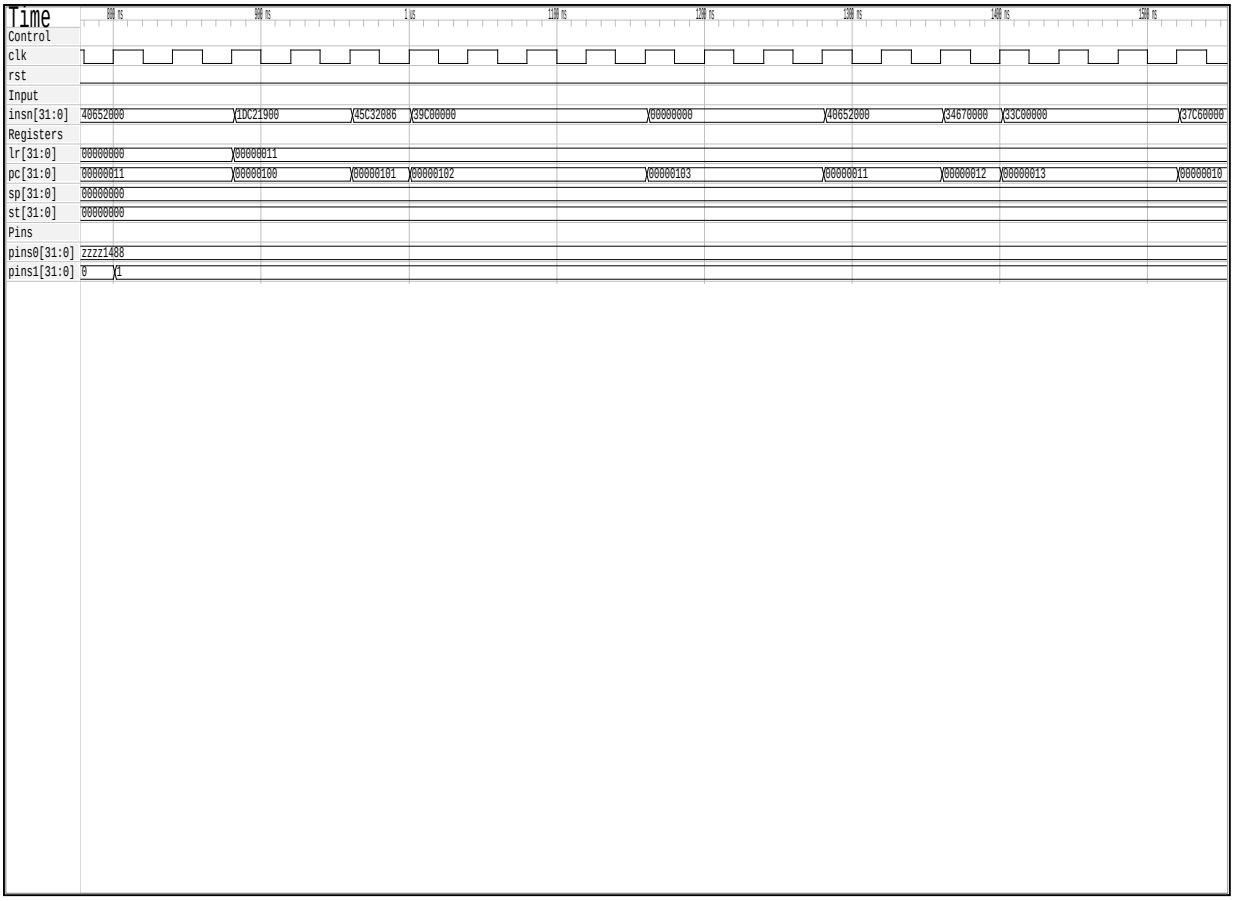
```
[0x00000000]:
movs 0x00 -> r0
movs 0xFFFFFFFF -> r1
movs r0 -> r2  //F0
movs 0x01 -> r3  //F1
movs 0x0C -> r5  //[gpio1.val]
movs 0x100 -> r6  //[fib()]
movs 0xFFFFFFFD -> r7  //-0x03
out r1 -> 0x0D
out r3 -> [r5]
brl [r6]
out1o r4 -> [r5]
rbr1o r7
br  r0
(nop)
[0x00000100]:  //r4 fib(&r2, &r3)
add r2, r3 -> r4
mov r3, r4 -> r2, r3
ret
(nop)
```

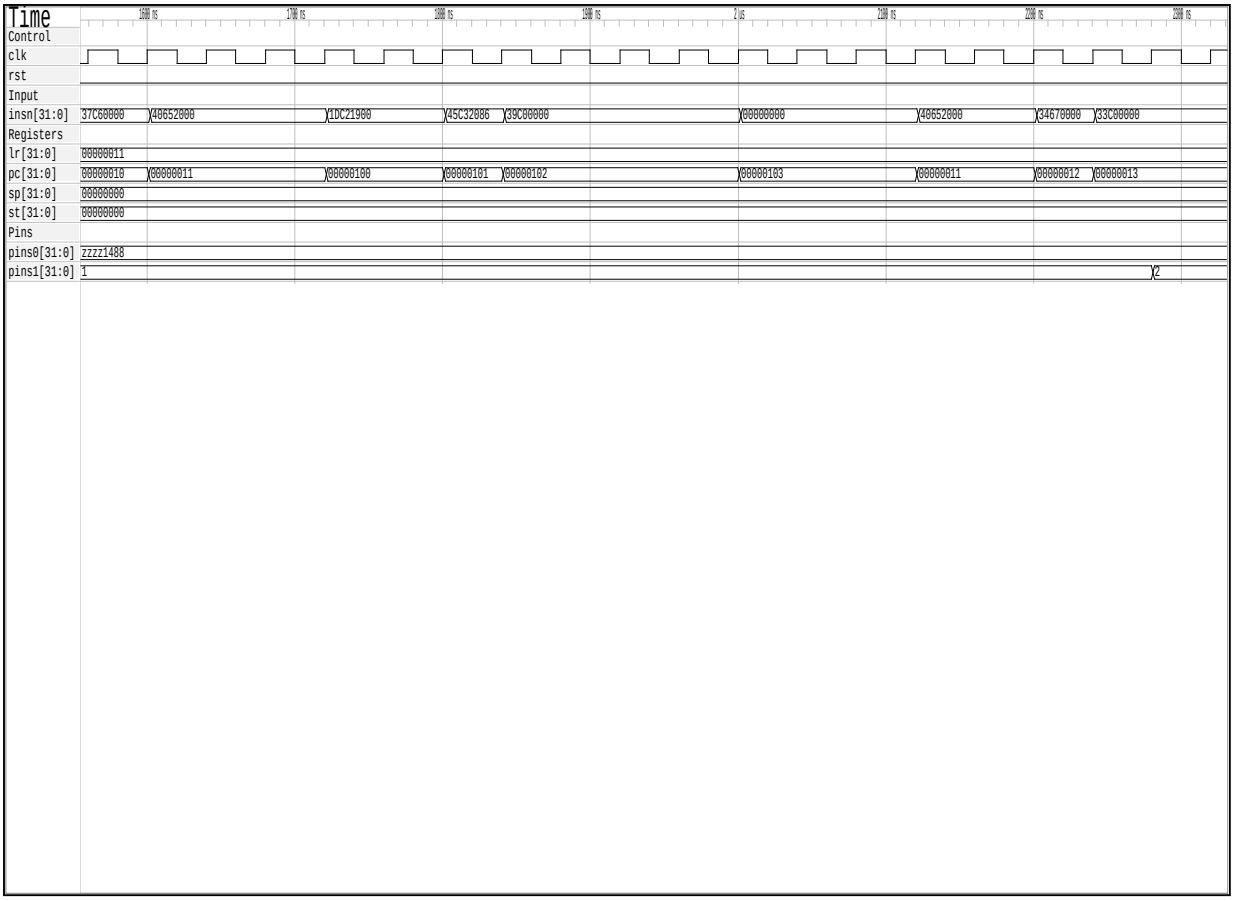
6.3.3 Временные диаграммы

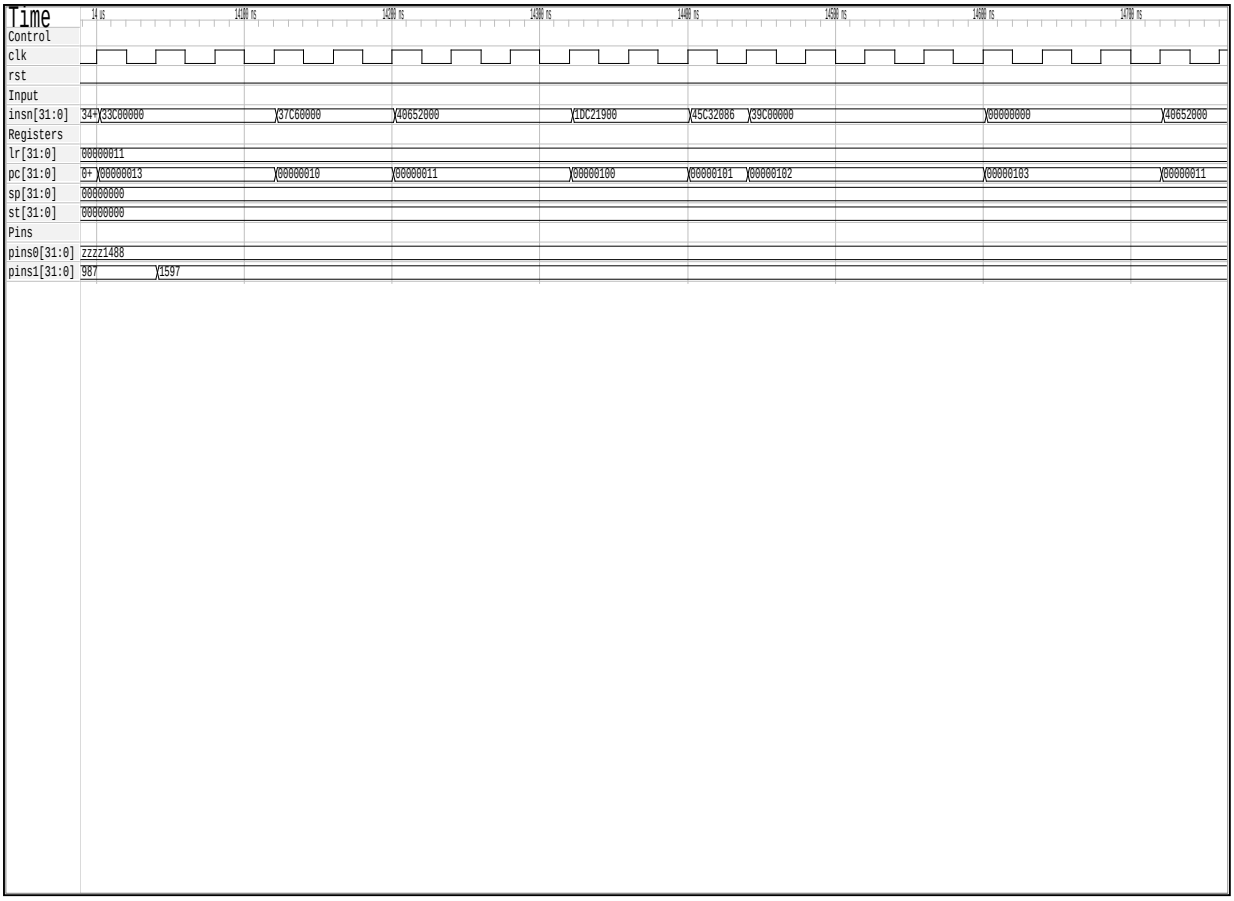
На изображениях:

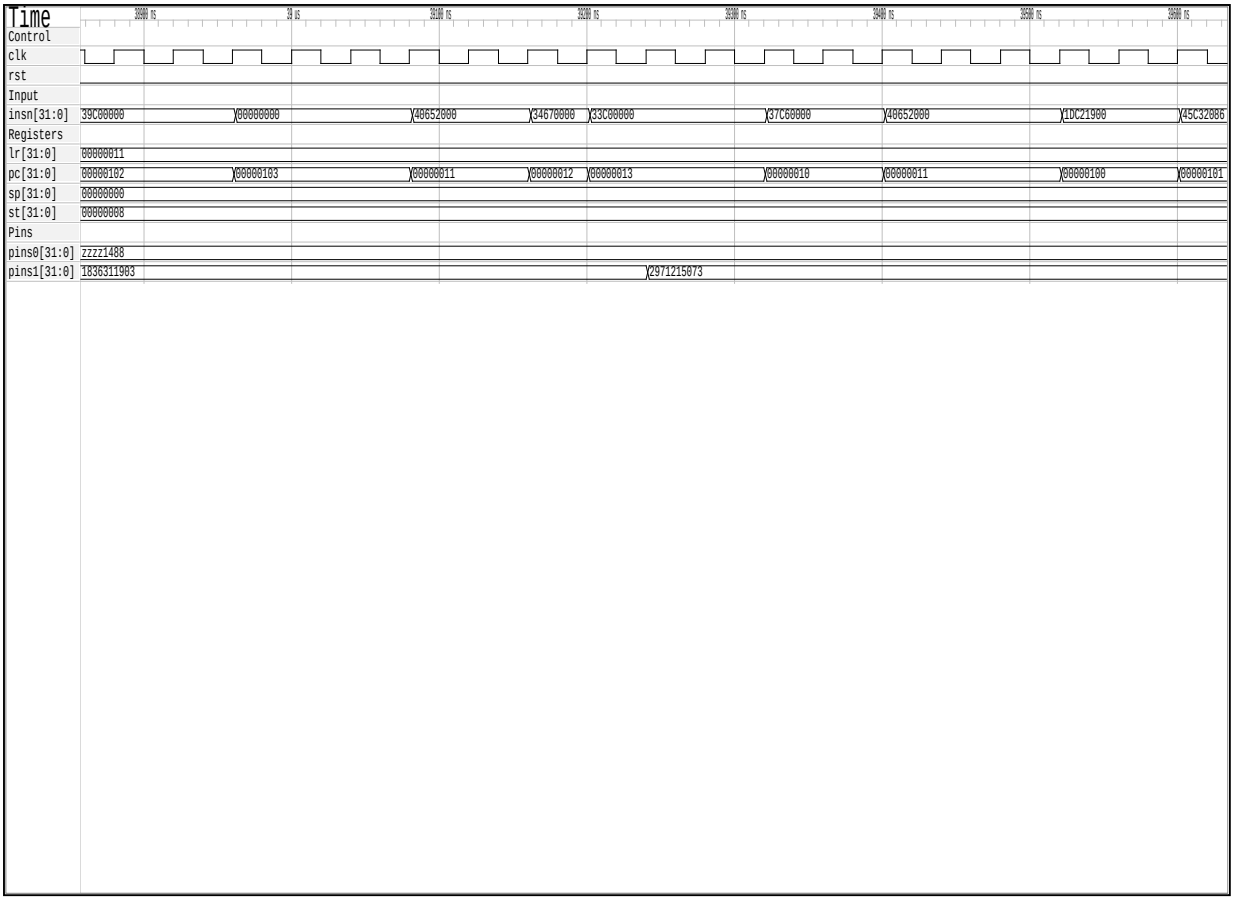
1. Инициализация
2. Первое число Фибоначчи (1)
3. Второе и третье числа Фибоначчи (1 и 2)
4. Семнадцатое число Фибоначчи (1597)
5. 47-е число Фибоначчи (2971215073)
6. Перезагрузка и переинициализация после 47-го числа
7. Первые 29 чисел Фибоначчи (обзорно)
8. 30-44 числа Фибоначчи (обзорно)
9. Перзагрузка и счёт сначала (обзорно)

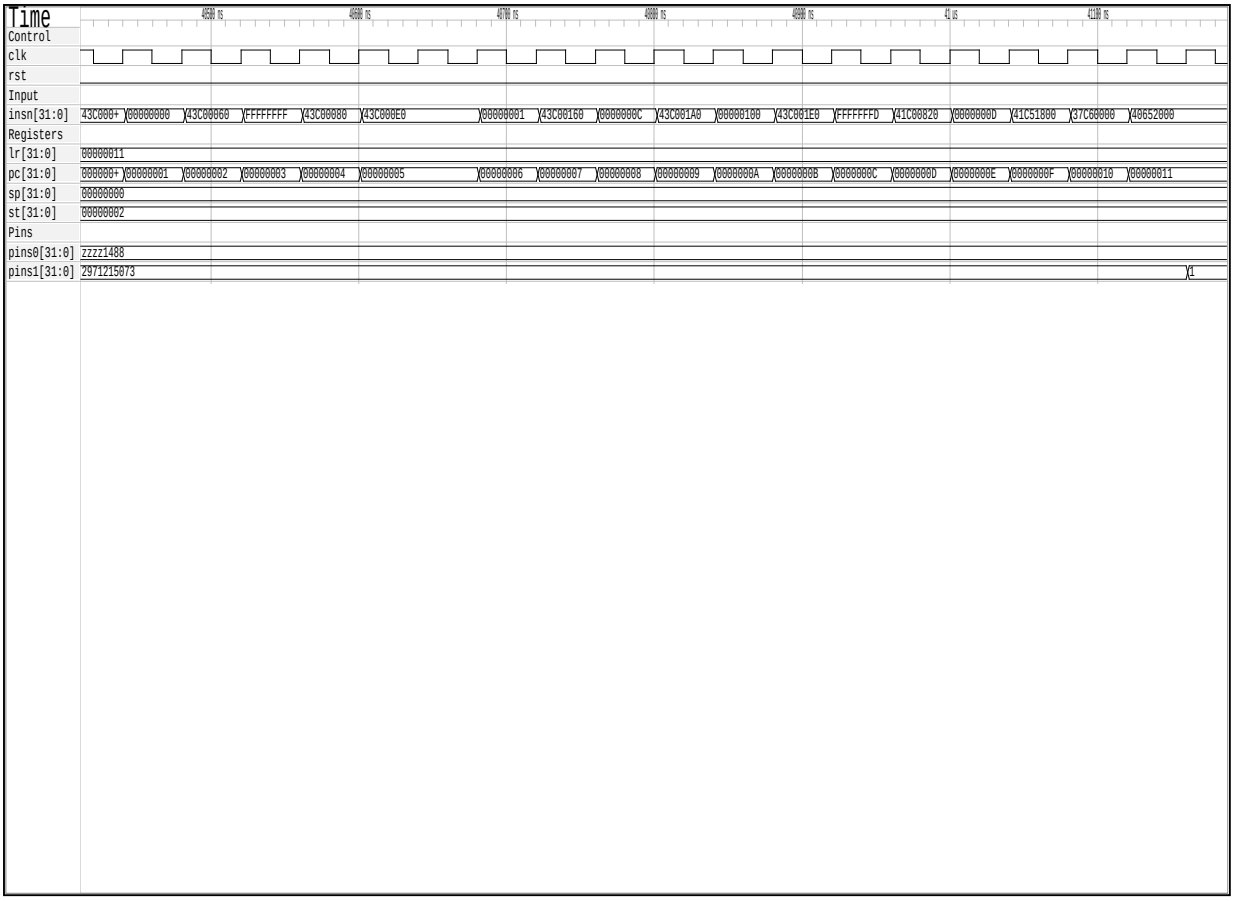

Time
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
Control
clk
rst
Input
insn[31:0]
xxxx+
43C00020
00000000
43C00060
FFFFFFFF
43C00080
43C000E0
00000001
43C00160
0000000C
43C001A0
00000100
43C001E0
FFFFFFFD
41C00820
0000000D
41C51800
37C60000
4065
Registers
lr[31:0]
pc[31:0]
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000010
sp[31:0]
st[31:0]
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
XXX

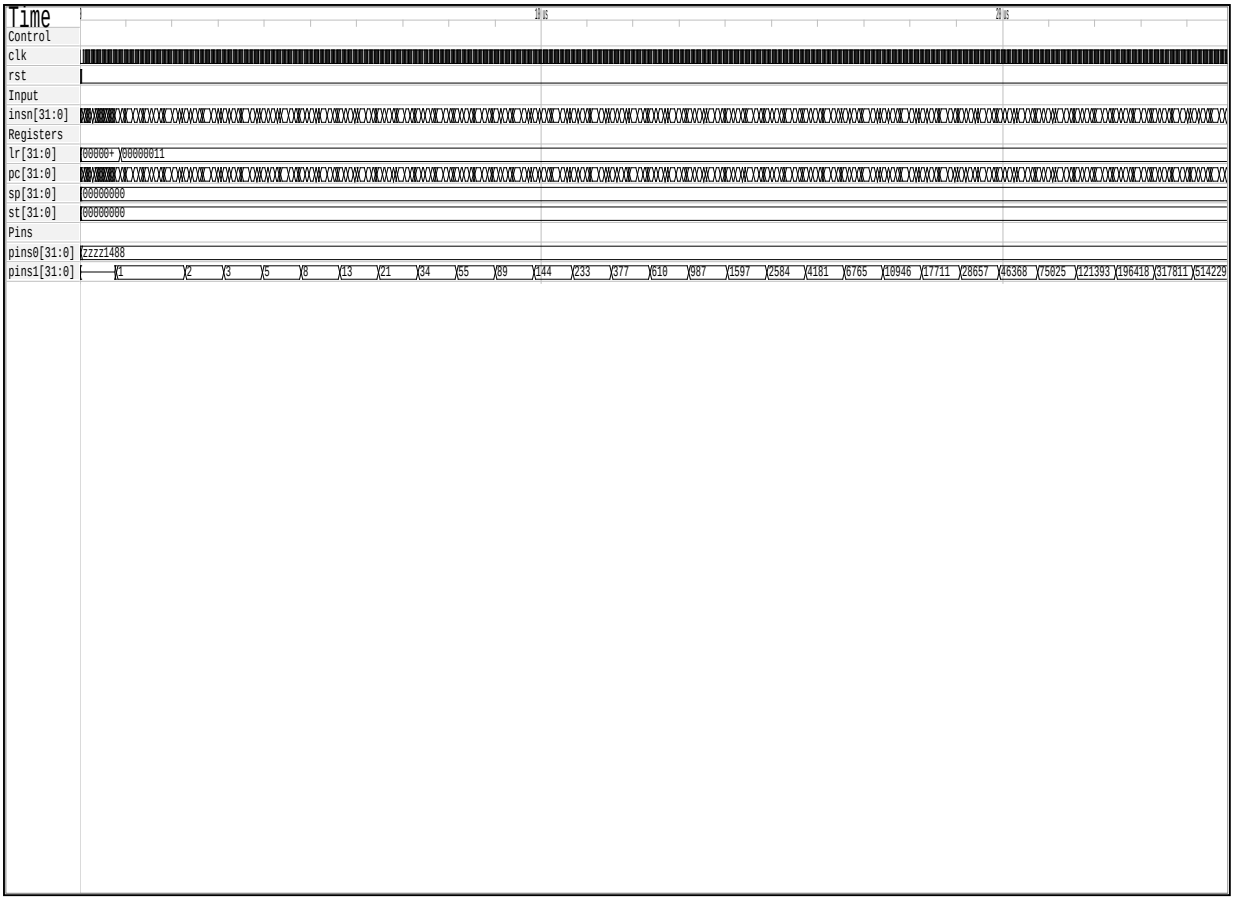

Time
800 ns
900 ns
1 us
1100 ns
1200 ns
1300 ns
1400 ns
1500 ns
Control
clk
rst
Input
insn[31:0]
40652000
1DC21900
45C32086
39C00000
00000000
40652000
34670000
33C00000
37C60000
Registers
lr[31:0]
00000000
00000011
pc[31:0]
00000011
00000100
00000101
00000102
00000103
00000011
00000012
00000013
00000010
sp[31:0]
00000000
st[31:0]
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
0
1

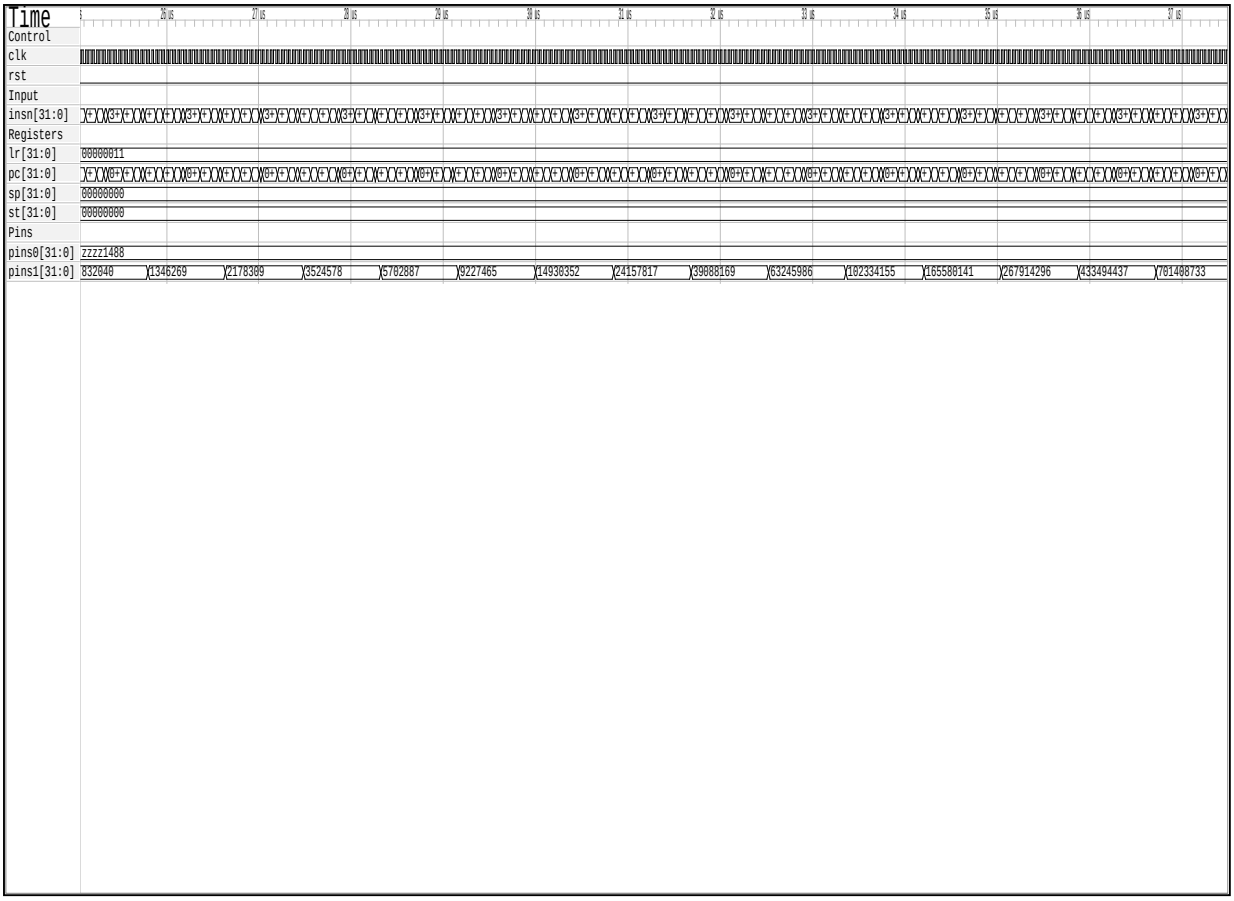

Time
1600 ns
1700 ns
1800 ns
1900 ns
2 us
2100 ns
2200 ns
2300 ns
Control
clk
rst
Input
insn[31:0]
37C60000
40652000
1DC21900
45C32086
39C00000
00000000
40652000
34670000
33C00000
Registers
lr[31:0]
00000011
pc[31:0]
00000010
00000011
00000100
00000101
00000102
00000103
00000011
00000012
00000013
sp[31:0]
00000000
st[31:0]
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
1
2

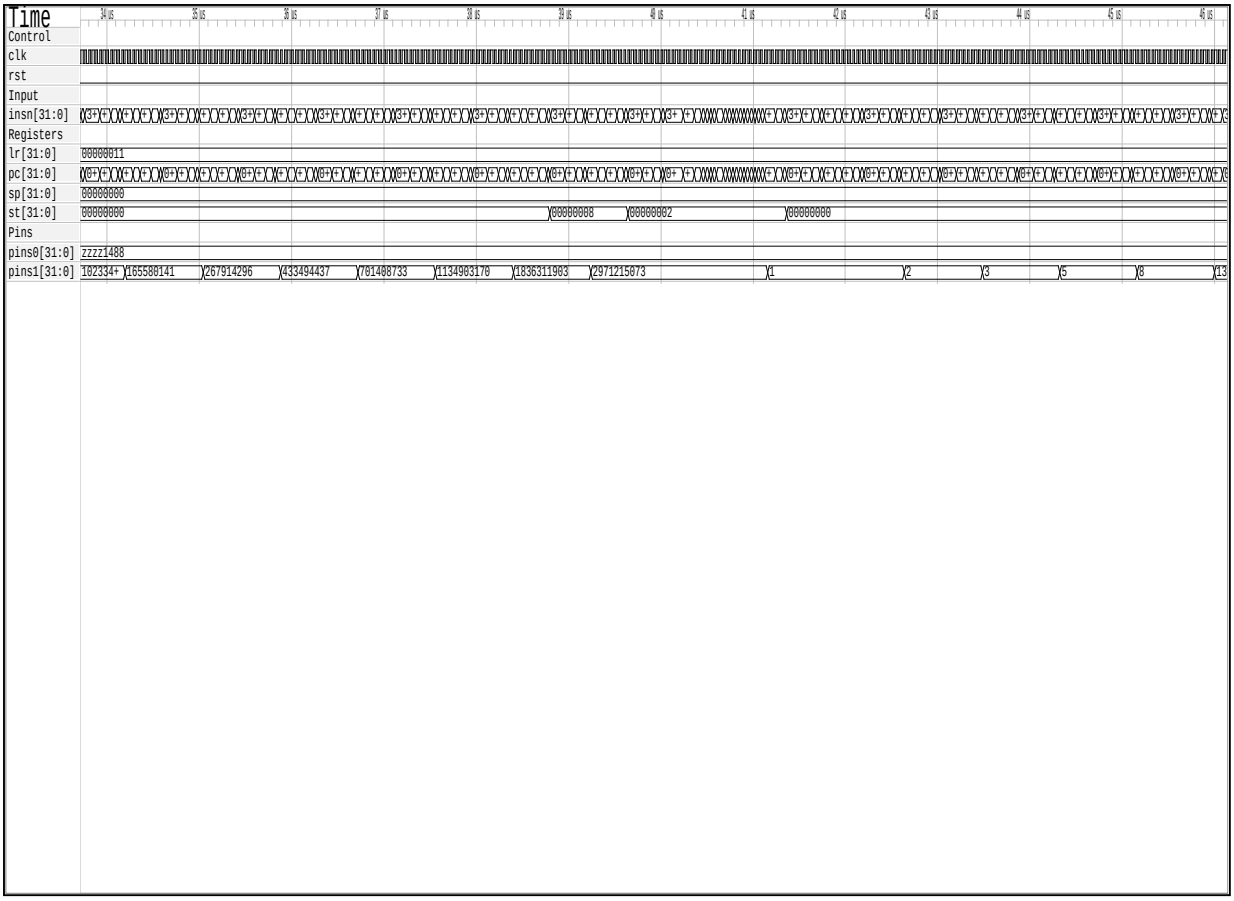

Time
14 ns
14400 ns
14600 ns
14800 ns
15000 ns
15200 ns
15400 ns
15600 ns
Control
clk
rst
Input
insn[31:0]
34+
33C00000
37C60000
40652000
1DC21900
45C32086
39C80000
00000000
40652000
Registers
lr[31:0]
00000011
pc[31:0]
0+
00000013
00000010
00000011
00000100
00000101
00000102
00000103
00000011
sp[31:0]
00000000
st[31:0]
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
987
1597


Time
Control
clk
rst
Input
insn[31:0]
39C00000
00000000
40652000
34670000
33C00000
37C60000
40652000
1DC21900
45C32086
Registers
lr[31:0]
00000011
pc[31:0]
00000102
00000103
00000011
00000012
00000013
00000010
00000011
00000100
00000101
sp[31:0]
00000000
st[31:0]
00000008
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
1836311903
2971215073


Time
Control
clk
rst
Input
insn[31:0]
Registers
lr[31:0]
pc[31:0]
sp[31:0]
st[31:0]
Pins
pins0[31:0]
pins1[31:0]
43C000+
00000000
43C00060
FFFFFFFF
43C00080
43C000E0
00000001
43C00160
0000000C
43C001A0
00000100
43C001E0
FFFFFFFD
41C00820
0000000D
41C51800
37C60000
40652000
00000011
000000+
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000010
00000011
00000000
00000002
zzzz1488
Z971215073


Time
10 us
20 us
Control
clk
rst
Input
insn[31:0]
Registers
lr[31:0]
00000011
pc[31:0]
sp[31:0]
00000000
st[31:0]
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229


Time
Control
clk
rst
Input
insn[31:0]
Registers
lr[31:0]
00000011
pc[31:0]
sp[31:0]
00000000
st[31:0]
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733


Time
Control
clk
rst
Input
insn[31:0]
Registers
lr[31:0]
00000011
pc[31:0]
sp[31:0]
00000000
st[31:0]
00000000
00000008
00000002
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
102334+
165580141
267914296
433494437
701408733
1134903170
1836311903
2971215073
1
2
3
5
8
13

7 Синтез

7.1 Средства синтезирования

Синтез был произведён с помощью открытого набора синтезаторов **YOSYS**. Синтез производился только для оценки размеров и временных характеристик процессора, актуальной загрузки на какие-либо платформы (ПЛИС) не было. Для исследования поведения размеров процессора были выбраны три цели:

1. ПЛИС Xilinx 7-Series. Такая ПЛИС содержит большое количество разноразмерных LUT (таблиц истинности), специальные блоки сумматоров и умножителей а также блоки консолидированной двухпортовой ОЗУ, которая подставляется на место блока ОЗУ процессора.

2. ПЛИС Lattice Semiconductors iCE40. Эта ПЛИС имеет упрощённую архитектуру, а именно состоит из LUT на 4 значения с присоединённым полным сумматором и отдельных блоков псевдодвухпортовой консолидированной ОЗУ. Из-за того, что блоки ОЗУ не имеют полных двух портов, подстановки их на место блока ОЗУ процессора не происходит, что влечёт резкое повышение количества использованных триггеров.

3. ASIC (заказная схема) на основе библиотеки OSU Stdcells для техпроцесса TSMC 25нм. Здесь наблюдается увеличение размеров процессора, вызванное отсутствием каких-дибо блоков стандартной оптимизации. Больше всего (> 80%) занимают блоки ОЗУ и регистровый файл, так как они набираются из отдельных триггеров и мультиплексоров.

Был произведён синтез на приведённые три цели предусмотренными для этих целей средствами YOSYS, результат был экспортирован в Verilog Netlist. Также был произведён вывод статистики синтезированных ячеек, которая и будет представлена далее.

Для целей временного анализа был проведён финтез в САПР для ПЛИС фирмы Altera **Quartus Prime**. В качестве целевой платформы была выбрана ПЛИС серии MAX10 с подходящим количеством ячеек (на основе оценки синтеза в YOSYS) и ног.

7.2 Результаты синтезирования

TODO !!!

7.3 Результаты временного анализа

TODO !!!

Часть IV

Заключение

В результате выполнения дипломной работы был создан процессор, удовлетворяющий всем начальным требованиям. Он был оттестирован с помощью симулятора сначала поблочно (на ранней стадии), потом в составе всей системы с использованием двух тестовых программ. Далее, для оценки эффективности данной реализации процессора был произведён синтез двумя различными инструментами, в ходе чего была получена информация о его площади (сложности) и временных характеристиках (максимальная рабочая частота). В таком виде система была выложена в открытый доступ.

Предполагается продолжение развития данной процессорной системы после сдачи дипломного проекта. Некоторые из краткосрочных целей:

- Реализовать часть АЛУ и инструкции для работы с числами с плавающей точкой одинарной точности (IEEE 754).
- Добавить operand forwarding в качестве меры по уменьшению задержек при ошибках конвейера.
- Добавить в ядро поддержку режима прерывания и контроллер прерываний (в периферийные устройства).
- Добавить MMU для реализации концепции Единого Адресного Пространства (отобразить ПЗУ, ОЗУ и периферию на одно адресное пространство).
- Произвести непосредственную проверку путём синтеза и загрузки в ПЛИС.

Список литературы

[dadda] Dadda L. Some schemes for parallel multipliers //Alta frequenza. – 1965. – Т. 34. – №. 5. – С. 349-356.

[cla] Lynch T., Swartzlander Jr E. E. A spanning tree carry lookahead adder //Computers, IEEE Transactions on. – 1992. – Т. 41. – №. 8. – С. 931-939.

[shifter] Pillmeier M. R., Schulte M. J., Walters III E. G. Design alternatives for barrel shifters //International Symposium on Optical Science and Technology. – International Society for Optics and Photonics, 2002. – С. 436-447.

[design] Microprocessor Design [Электронный ресурс]: электронная книга // сайт wikibooks.org – Режим доступа : https://en.wikibooks.org/wiki/Microprocessor_Design

Часть V

Приложение 1. Instruction Set Architecture

8 Введение

8.1 Общее описание

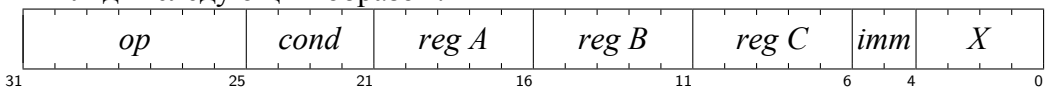
Процессор УП-1 обладает монолитной 32-битной архитектурой с типом доступа к памяти/периферии load-store, совмещённым доступом к памяти/периферии, раздельным доступом к регистрам и четырёхстадийным конвейером, что означает следующее:

- Размер любой инструкции, мгновенного значения, чтения/записи памяти/периферии, регистров и т.д. равен 32 битам
- Большинство инструкций могут работать только с регистрами (кроме операций load-store)
- В наборе есть класс инструкций, осушествляющий доступ к памяти/-периферии
- Чтение/запись в память/периферию происходит на одной и той же стадии конвейера, что исключает возможность появления ошибок конвейера (pipeline hazards)
- Чтение/запись в регистры, в свою очередь, происходят на разных стадиях конвейера (чтение на первой, запись на четвёртой), что приводит к возможности возникновения ошибок конвейера, а значит требует мер по их устранению.

8.2 Формат инструкции

Как было сказано ранее, каждая инструкция (машинное слово) имеет размер 32 бита. По строению инструкции подразделяются на два вида:

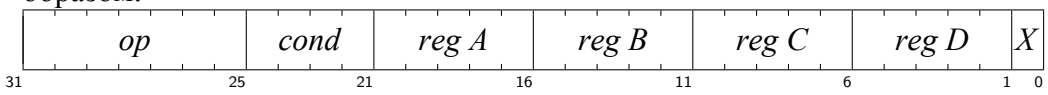
1. Инструкция с тремя и менее операндами. Такая инструкция может иметь до двух входных операндов и до одного выходного. Любой из входных операндов может быть заменён на мгновенное значение. Инструкции такого типа выглядят следующим образом:



Поля:

- (op)code - Опкод, код операции
- (cond)itional code - Код условного исполнения
- Reg A, B - входные операнды
- Reg C - выходной операнд
- (imm)ediate operation - Код подстановки мгновенного значения (см. далее)
- X - неиспользуемые биты

2. Инструкция с четырьмя операндами. Такая инструкци имеет два входных операнда и два выходных, что позволяет сполна использовать ресурсы регистрового файла (напомню, что он четырёхпортовый - два порта на чтение и два на запись). Однако, инструкция такого типа не может использовать подстановку мгновенных значений. Выглядит такая инструкция следующим образом:



Поля:

   - (op)code - Опкод, код операции
   - (cond)itional code - Код условного исполнения
   - Reg A, B - входные операнды
   - Reg C, D - выходные операнды
   - X - неиспользуемый бит

Стоит заметить, что почти все (кроме двух ????) инструкций имеют трёхоперандный формат, а значит почти все инструкции могут использовать подстановку мгновенных значений.

8.3 Условное исполнение

Каждая инструкция (кроме, пожалуй, NOP, в котором он не учитывается) имеет код условного исполнения. Такой код позволяет производить условные вычисления следующим образом:

- Если условие, связанное с условным кодом выполняется, то инструкция без изменений спускается по конвейеру, производя необходимые изменения.
- Если же такое условие не выполняется, то на стадиях записи в память/периферию/регистры эта инструкция подменяется на чистый NOP, то есть эффективно пропускается. Флаги такая инструкция также не изменяет.

Такой подход позволяет крайне эффективно организовывать условные секции в машинном коде, путём отказа от ветвления, которое требует очистки конвейера, а значит имеет задержку исполнения в 4 такта.

Условные коды работают с флагами исполнения. Таких флагов всего 4:

1. (N)egative - Отрицательный результат. Этот флаг равен самому старшему биту результата.
2. (Z)ero - Нулевой результат. Этот флаг выставляется, когда результат равен беззнаковому нулю.
3. (C)arry, или также Unsigned Overflow - беззнаковое переполнение в результате арифметической или сдвиговой операции
4. Signed o(V)erflow - знаковое переполнение в результате арифметической или сдвиговой операции

Условие исполнения задаётся четырёхбитным полем cond, которое присутствует в каждой инструкции:

0000: EQ - «Равен». Условие - Z

0001: NEQ - «Не равен». Условие - $\overline{Z}$

0010: HS - «Больше или равен беззнаковый». Условие - C

0011: LO - «Строго меньше беззнаковый». Условие - $\overline{\text{C}}$

0100: NEG - «Отрицательный». Условие - N

0101: POS - «Положительный». Условие - $\overline{N}$

0110: SOV - «Знаковое переполнение». Условие - V

0111: NSOV - «Отсутствие знакового переполнения». Условие - $\overline{V}$

1000: HI - «Строго больше беззнаковый». Условие - $C \wedge \overline{Z}$

1001: LS - «Меньше или равен беззнаковый». Условие - $\overline{C} \wedge Z$

1010: GE - «Больше либо равен знаковый». Условие - $N = V$

1011: LT - «Строго меньше знаковый». Условие - $N \neq V$

1100: GT - «Строго больше знаковый». Условие - $\overline{Z} \wedge (N = V)$

1101: LE - «Меньше либо равен знаковый». Условие - $Z \wedge (N \neq V)$

1110: AL - «Всегда». Всегда выполняется.

1111: NV - «Никогда». Никогда не выполняется.

8.4 Мгновенные значения

Инструкции трёхоперандного типа могут производить подстановку мгновенных значений на место любого из своих входных операндов. Такое поведение инструкции регулируется полем imm следующим образом:

00: Мгновенные значения отсутствуют

01: Мгновенное значение подставляется в операнд B

10: Мгновенное значение подставляется в операнд A

11: Первое мгновенное значение подставляется в операнд A, второе - в операнд B

В зависимости от значения поля imm следующие после инструкции одно/два слова будут восприняты как мгновенные значения для соответствующих операндов. Такая инструкция будет задержана на первой стадии конвейера до тех пор, пока не будут получены все необходимые мгновенные значения, что соответствует одному/двум тактам задержки.

Следует также заметить, что операции с памятью один из операндов подставляют в поле «Адрес» интерфейса, которое следует отличным от стандартных регистров A и B путём, поэтому мгновенное значение тоже будет подставлено в адрес и пройдёт мимо стадии исполнения.

8.5 Набор инструкций

Процессор УП-1 обладает достаточно большим набором инструкций, что позволяет ему быть предельно понятным для конечного пользователя. Всего в наборе содержится 35 инструкций, которые можно подразделить на следующие классы:

- Логические инструкции - or, nor, and, nand, inv, xor (logic)
- Сдвиги - арифметический, логический и циклический, влево и вправо (shift)
- Арифметические операции - сумма, разность, беззнаковое произведение, инкремент/декремент, сравнение (arith)
- Операции потока исполнения - прыжок, вызов и возврат (branch)
- Операции с ОЗУ (mem)
- Операция перемещения регистр-регистр (в т.ч двойная) и пустая операция (mov и nop)
- Операции с шиной периферических устройств (sys)

Тип инструкции задаётся значением семибитного поля opcode. Такоее поле может вместить в себя до 128 инструкций. В данный момент набор содержит 33 инструкции, представленные в сводной таблице 1

Таблица 1: ISA

Набор инструкций

№	Мнемоника	Описание	Опкод	Класс	Вид	Операнды				Данные	Флаги				Циклы
						A	B	C	D		N	Z	C	V	
1	NOP	No-op	0000000	nop	3 op	-	-	-	-	-	-	-	-	-	1
2	OR	Bitwise OR	0000001	logic	3 op	a	b	c	-	a,b -> c	0	+	0	0	1
3	NOR	Bitwise NOR	0000010	logic	3 op	a	b	c	-	a,b -> c	0	+	0	0	1
4	AND	Bitwise AND	0000011	logic	3 op	a	b	c	-	a,b -> c	0	+	0	0	1
5	NAND	Bitwise NAND	0000100	logic	3 op	a	b	c	-	a,b -> c	0	+	0	0	1
6	INV	Bitwise NOT	0000101	logic	3 op	a	-	c	-	a -> c	0	+	0	0	1
7	XOR	Bitwise XOR	0000110	logic	3 op	a	b	c	-	a,b -> c	0	+	0	0	1
8	XNOR	Bitwise XNOR	0000111	logic	3 op	a	b	c	-	a,b -> c	0	+	0	0	1
9	LSL	Logical Shift Left	0001000	shift	3 op	a	b	c	-	a,b -> c	+	+	+	0	1
10	LSR	Logical Shift Right	0001001	shift	3 op	a	b	c	-	a,b -> c	+	+	+	0	1
11	ASR	Arithmetic Shift Right	0001010	shift	3 op	a	b	c	-	a,b -> c	+	+	+	0	1
12	ASL	Arithmetic Shift Left	0001011	shift	3 op	a	b	c	-	a,b -> c	+	+	+	0	1
13	CSR	Cyclic Shift Right	0001100	shift	3 op	a	b	c	-	a,b -> c	+	+	+	0	1
14	CSL	Cyclic Shift Left	0001101	shift	3 op	a	b	c	-	a,b -> c	+	+	+	0	1
15	ADD	Add w/o carry	0001110	arith	3 op	a	b	c	-	a,b -> c	+	+	+	+	1
16	SUB	Subtract w/o carry	0001111	arith	3 op	a	b	c	-	a,b -> c	+	+	+	+	1

Набор инструкций - продолжение

17	MULL	Multiply and store low	0010000	arith	3 op	a	b	c	-	a,b -> c	0	+	+	0	1
18	MULH	Multiply and store high	0010001	arith	3 op	a	b	c	-	a,b -> c	0	+	+	0	1
19	MUL	Multiply and store both	0010010	arith	4 op	a	b	c	d	a,b -> c,d	0	+	+	0	1
20	CSG	Change Sign	0010011	arith	3 op	a	-	c	-	a -> c	+	+	+	+	1
21	INC	Increment	0010100	arith	3 op	a	-	c	-	a,+1 -> c	+	+	+	+	1
22	DEC	Decrement	0010101	arith	3 op	a	-	c	-	a,-1 -> c	+	+	+	+	1
23	CMP	Compare	0010110	arith	3 op	a	b	-	-	a,b	+	+	+	+	1
24	CMN	Compare with Negative	0010111	arith	3 op	a	b	-	-	a,-b	+	+	+	+	1
25	TST	Test	0011000	arith	3 op	a	b	-	-	a,b	0	+	0	0	1
26	BR	Branch	0011001	branch	3 op	a	-	-	-	a -> pc	-	-	-	-	4
27	RBR	Relative branch	0011010	branch	3 op	a	-	-	-	a, pc -> pc	-	-	-	-	4
28	BRL	Branch w/ Link	0011011	branch	3 op	a	-	-	-	a, pc -> pc, lr	-	-	-	-	4
29	RET	Return	0011100	branch	3 op	-	-	-	-	lr -> pc	-	-	-	-	4
30	LDR	Load from RAM	0011101	mem	3 op	a1	-	c	-	m[a1] -> c	-	-	-	-	1
31	STR	Store to RAM	0011110	mem	3 op	a1	b	-	-	b -> m[a1]	-	-	-	-	1
32	IN	Input from SYS	0011111	sys	3 op	a1	-	c	-	s[a1] -> c	-	-	-	-	1
33	OUT	Output to SYS	0100000	sys	3 op	a1	b	-	-	b -> s[a1]	-	-	-	-	1
34	MOVS	Move Single	0100001	mov	4 op	a	-	c	-	a -> c	-	-	-	-	1
35	MOV	Move Double	0100010	mov	3 op	a	b	c	d	a,b -> c,d	-	-	-	-	1

Целевые регистры:

a: Первый операнд АЛУ.

b: Второй операнд АЛУ.

c: Первый операнд записи в регистр.

d: Второй операнд записи в регистр.

a1: Первый адрес для записи в память/периферию.

pc: Program Counter, программный указатель, тж. r31. Указывает на следующую инструкцию.

lr: Link Register, адрес возврата, тж. r29. Содержит адрес возврата из процедуры.

m[x]: Содержимое ОЗУ по адресу x

s[x]: Периферийное устройство по адресу x

9 Описание

9.1 NOP

Пустая операция

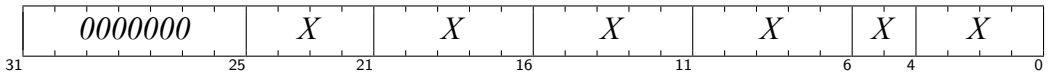


Рис. 3: Машинное представление инструкции NOP

9.1.1 Описание

No Operation, пустая инструкция
Пропускает один такт не меняя флагов исполнения

9.1.2 Флаги, затрагиваемые данной инструкцией:

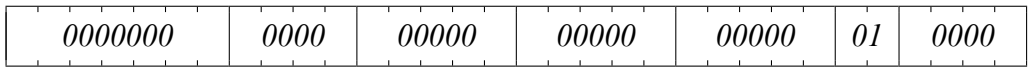
N	Z	C	V
-	-	-	-

9.1.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Не может иметь кодов исполнения
- Не может использовать мгновенные значения
- Не производит запись в регистры, память и периферические устройства.
- Не меняет потока исполнения.

9.1.4 Пример использования:

```
NOP  //       1
```

1. 

9.2 OR

Побитовое ИЛИ

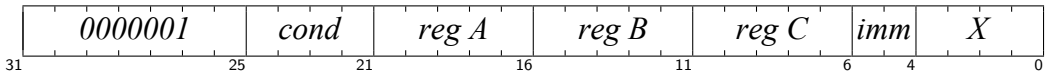


Рис. 4: Машинное представление инструкции OR

9.2.1 Описание

Производит побитовое ИЛИ двух операндов и сохраняет результат в третий

9.2.2 Флаги, затрагиваемые данной инструкцией:

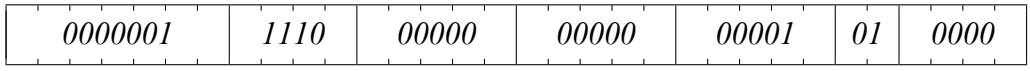
N	Z	C	V
0	+	0	0

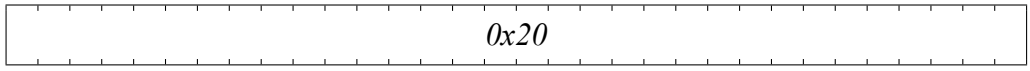
9.2.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.2.4 Пример использования:

```
OR r0, 0x20 -> r1  // r0    32          r1
```

1. 

2. 

9.3 NOR

Побитовое ИЛИ-НЕ

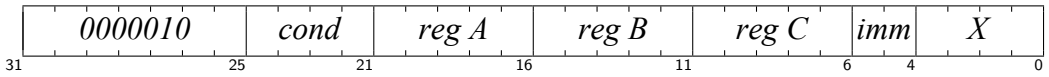


Рис. 5: Машинное представление инструкции NOR

9.3.1 Описание

Производит побитовое ИЛИ-НЕ двух операндов и сохраняет результат в третий

9.3.2 Флаги, затрагиваемые данной инструкцией:

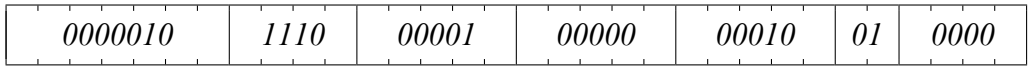
N	Z	C	V
0	+	0	0

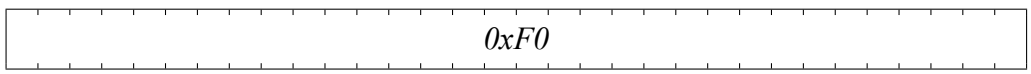
9.3.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.3.4 Пример использования:

```
NOR r1, 0xF0 -> r2  // r1   -  240          r2
```

1. 

2. 

9.4 AND

Побитовое И

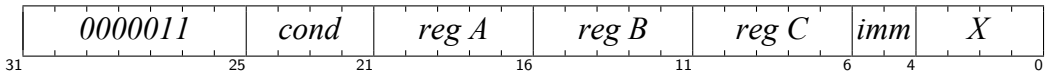


Рис. 6: Машинное представление инструкции AND

9.4.1 Описание

Производит побитовое И двух операндов и сохраняет результат в третий

9.4.2 Флаги, затрагиваемые данной инструкцией:

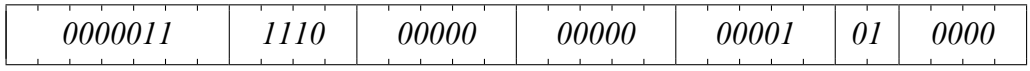
N	Z	C	V
0	+	0	0

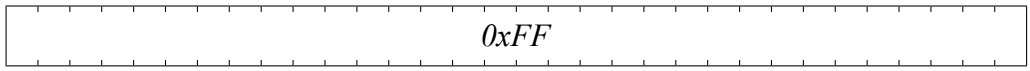
9.4.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.4.4 Пример использования:

```
AND r0, 0xFF -> r1  // r0   255          r1
```

1. 

2. 

9.5 NAND

Побитовое И-НЕ

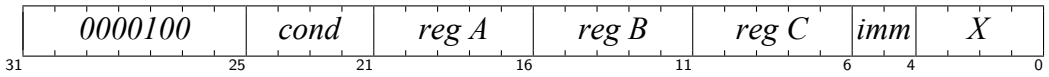


Рис. 7: Машинное представление инструкции NAND

9.5.1 Описание

Производит побитовое И-НЕ двух операндов и сохраняет результат в третий

9.5.2 Флаги, затрагиваемые данной инструкцией:

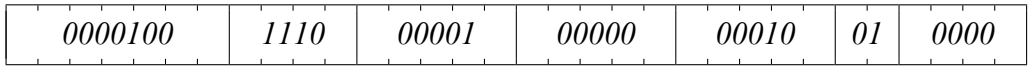
N	Z	C	V
0	+	0	0

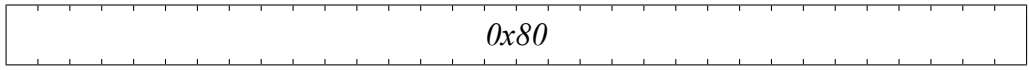
9.5.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.5.4 Пример использования:

```
NAND r1, 0x80 -> r2  // r1  -  128          r2
```

1. 

2. 

9.6 INV

Побитовая инверсия

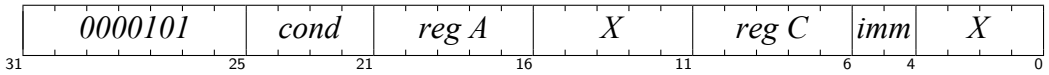


Рис. 8: Машинное представление инструкции INV

9.6.1 Описание

Инвертирует содержимое операнда и сохраняет результат во второй.

9.6.2 Флаги, затрагиваемые данной инструкцией:

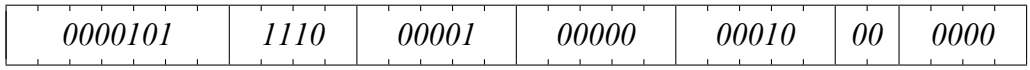
N	Z	C	V
0	+	0	0

9.6.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

9.6.4 Пример использования:

```
INV r1 -> r2  //          r1          r2
```

1. 

9.7 XOR

Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ

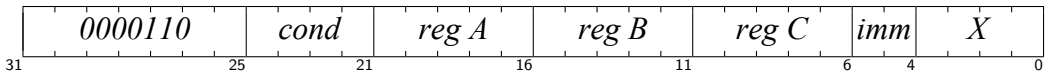


Рис. 9: Машинное представление инструкции XOR

9.7.1 Описание

Производит побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ двух операндов и сохраняет результат в третий

9.7.2 Флаги, затрагиваемые данной инструкцией:

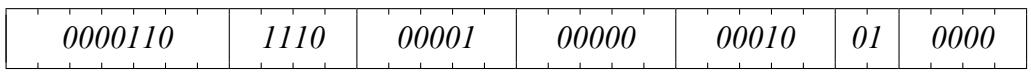
N	Z	C	V
0	+	0	0

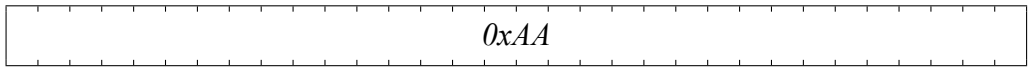
9.7.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.7.4 Пример использования:

```
XOR r1, 0xAA -> r2  // r1    .    170          r2
```

1. 

2. 

9.8 XNOR

Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ

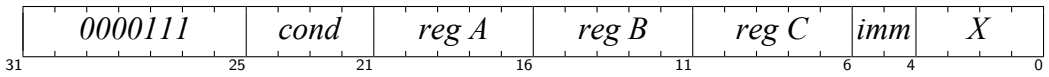


Рис. 10: Машинное представление инструкции XNOR

9.8.1 Описание

Производит побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ двух операндов и сохраняет результат в третий

9.8.2 Флаги, затрагиваемые данной инструкцией:

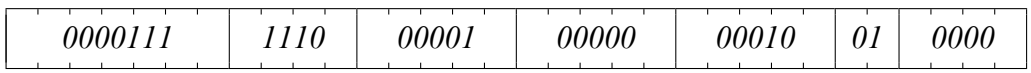
N	Z	C	V
0	+	0	0

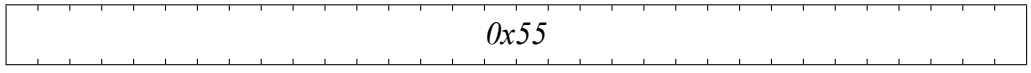
9.8.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.8.4 Пример использования:

```
XNOR r1, 0x55 -> r2  // r1   .   -  85          r2
```

1. 

2. 

9.9 LSL

Логический сдвиг влево

Рис. 11: Машинное представление инструкции LSL

9.9.1 Описание

Сдвигает содержимое первого операнда влево на количество бит, соответствующее младшим пяти битам второго операнда и сохраняет результат в третий операнд. Операция аналогична беззнаковому делению на два в степени второй операнд с округлением вниз.

9.9.2 Флаги, затрагиваемые данной инструкцией:

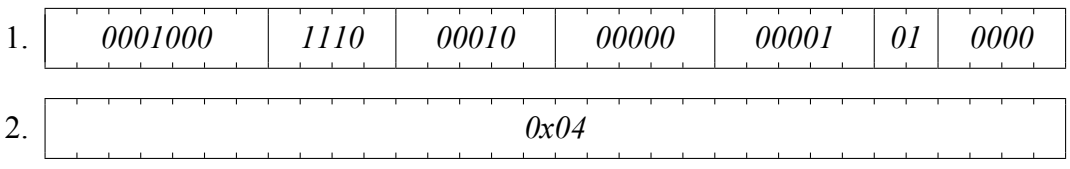
N	Z	C	V
+	+	+	0

9.9.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.9.4 Пример использования:

```
LSL r2, 0x04 -> r1  //      r2       4              r1
```



9.10 LSR

Логический сдвиг вправо

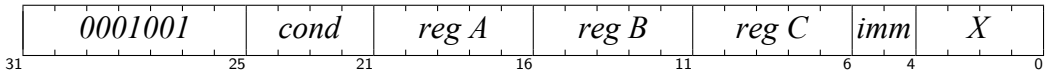


Рис. 12: Машинное представление инструкции LSR

9.10.1 Описание

Сдвигает содержимое первого операнда вправо на количество бит, соответствующее младшим пяти битам второго операнда и сохраняет результат в третий операнд. Операция аналогична беззнаковому умножению на два в степени второй операнд.

9.10.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	0

9.10.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.10.4 Пример использования:

```
LSR r2, r0 -
> r1  //      r2       (r0)            r1
```

1.

0001001	1110	00010	00000	00001	00	0000

9.11 ASR

Арифметический сдвиг вправо

Рис. 13: Машинное представление инструкции ASR

9.11.1 Описание

Сдвигает содержимое первого операнда вправо на количество бит, соответствующее младшим пяти битам второго операнда, сохраняя и распространяя при этом самый старший бит (знак) и сохраняет результат в третий операнд. Операция аналогична знаковому умножению на два в степени второй операнд .

9.11.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	0

9.11.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.11.4 Пример использования:

```
ASR r2, r0 -> r1  //                  r2   (r0)
                  //            r1
```

1.

0001010	1110	00010	00000	00001	00	0000

9.12 ASL

Арифметический сдвиг влево

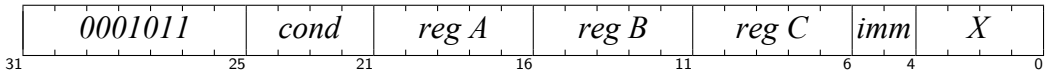


Рис. 14: Машинное представление инструкции ASL

9.12.1 Описание

Сдвигает содержимое первого операнда влево на количество бит, соответствующее младшим пяти битам второго операнда, сохраняя при этом самый старший бит (знак) и сохраняет результат в третий операнд. Операция аналогична знаковому делению на два в степени второй операнд с округлением вниз.

9.12.2 Флаги, затрагиваемые данной инструкцией:

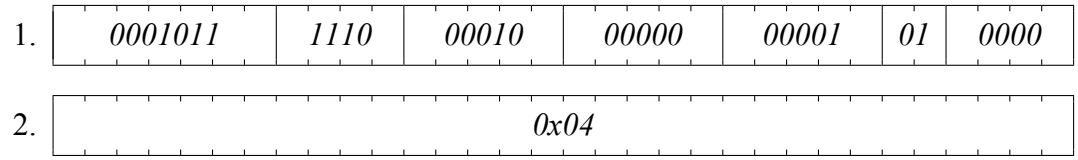
N	Z	C	V
+	+	+	0

9.12.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.12.4 Пример использования:

```
ASL r2, 0x04 -> r1  //      r2
                    // 4            r1
```



9.13 CSR

Циклический сдвиг вправо

Рис. 15: Машинное представление инструкции CSR

9.13.1 Описание

Циклически сдвигает (вращает) содержимое первого операнда вправо на количество бит, соответствующее младшим пяти битам второго операнда и сохраняет результат в третий операнд.

9.13.2 Флаги, затрагиваемые данной инструкцией:

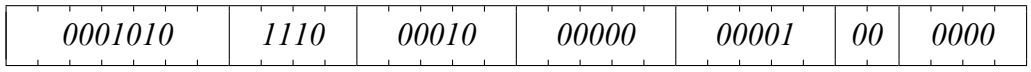
N	Z	C	V
+	+	+	0

9.13.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.13.4 Пример использования:

```
SR r2, r0 -> r1  //            r2   (r0)
                 //              r1
```

1. 

9.14 CSL

Арифметический сдвиг влево

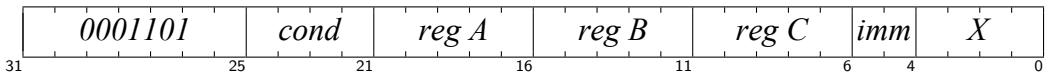


Рис. 16: Машинное представление инструкции CSL

9.14.1 Описание

Циклически сдвигает содержимое первого операнда влево на количество бит, соответствующее младшим пяти битам второго операнда и сохраняет результат в третий операнд.

9.14.2 Флаги, затрагиваемые данной инструкцией:

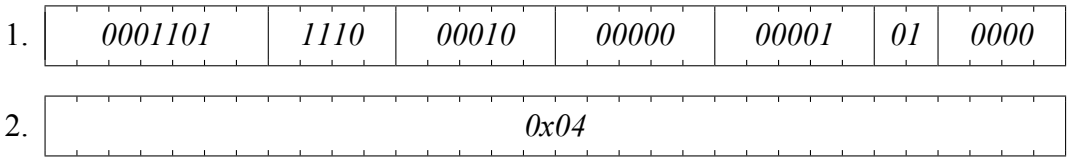
N	Z	C	V
+	+	+	0

9.14.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.14.4 Пример использования:

```
CSL r2, 0x04 -> r1  //           r2
                    // 4              r1
```



9.15 ADD

Сложение

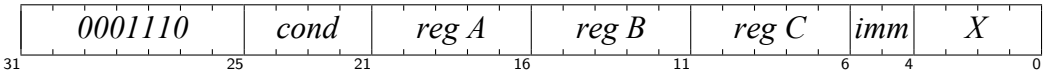


Рис. 17: Машинное представление инструкции ADD

9.15.1 Описание

Суммирует первый и второй операнд и сохраняет сумму в третий. Поддерживает отрицательные числа в дополнительном коде (two's complement, дополнение к двойке). В случае отрицательного результата, он также будет представлен в дополнительном коде.

9.15.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

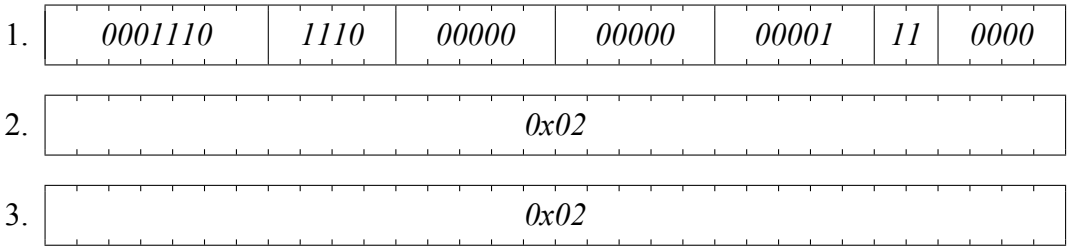
9.15.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Производит запись в регистр.

- Не меняет потока исполнения.

9.15.4 Пример использования:

```
ADD 0x02, 0x02 -> r1  //      2   2          r1
```



9.16 SUB

Вычитание

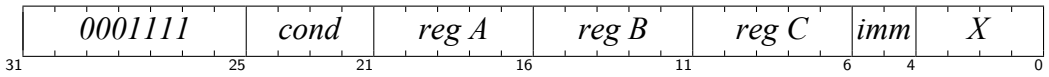


Рис. 18: Машинное представление инструкции SUB

9.16.1 Описание

Вычитает второй операнд из первого и сохраняет разность в третий. Поддерживает отрицательные числа в дополнительном коде (two's complement, дополнение к двойке). В случае отрицательного результата, он также будет представлен в дополнительном коде.

9.16.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

9.16.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.

- Может использовать до двух мгновенных значений.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.16.4 Пример использования:

```
SUB r1, r2 -> r3  //      r2   r1          r3
```

1. | *0001111* | *1110* | *00001* | *00010* | *00011* | *00* | *0000* |

9.17 MULL

Умножение (32-битная версия)

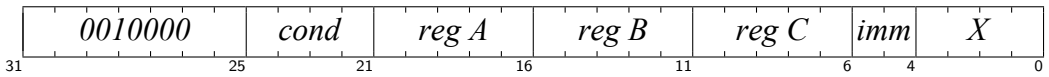


Рис. 19: Машинное представление инструкции MULL

9.17.1 Описание

Производит умножение первого и второго операнда и сохраняет младшее слово в третий. Эквивалентна 32-битному умножению. При ненулевом старшем слове выставляется флаг C. Знак ???

9.17.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	+	0

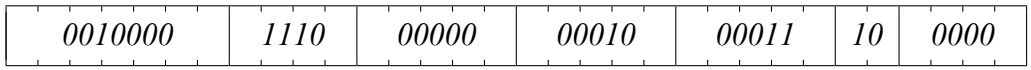
9.17.3 Свойства инструкции:

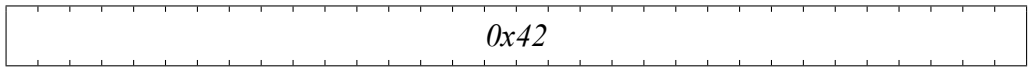
- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.

- Производит запись в регистр.
- Не меняет потока исполнения.

9.17.4 Пример использования:

```
MULL 0x42, r2 -> r3  //      0x42   r2          r3
```

1. 

2. 

9.18 MULH

Умножение с сохранением старшего слова.

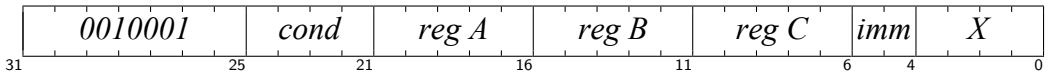


Рис. 20: Машинное представление инструкции MULH

9.18.1 Описание

Производит умножение первого и второго операнда и сохраняет старшее слово в третий. При ненулевом старшем слове выставляется флаг C.

9.18.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	+	0

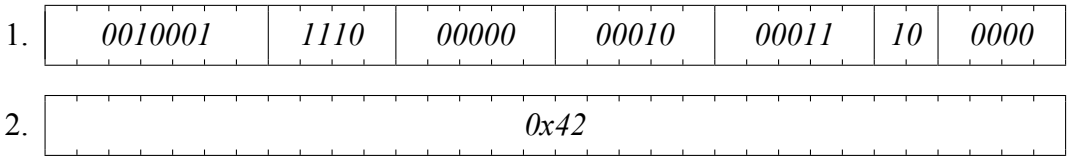
9.18.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.

- Производит запись в регистр.
- Не меняет потока исполнения.

9.18.4 Пример использования:

```
MULH 0x42, r2 -> r3  //       0x42   r2
                     //            r3
```



9.19 MUL

Умножение (полная версия)

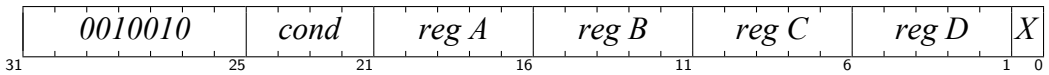


Рис. 21: Машинное представление инструкции MUL

9.19.1 Описание

Производит умножение первого и второго операнда и сохраняет младшее слово в третий, старшее - в четвёртый. При ненулевом старшем слове выставляется флаг C. Операция беззнаковая, т.е. знаки входных операндов никак не учитываются.

9.19.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	+	0

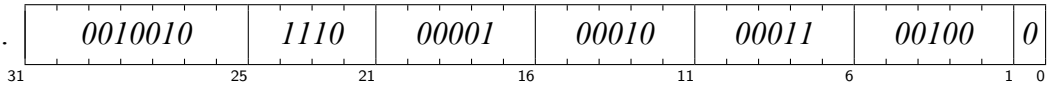
9.19.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.

- Не может использовать мгновенные значения.
- Производит две записи в регистр.
- Не меняет потока исполнения.

9.19.4 Пример использования:

```
MUL r1, r2 -> r3, r4 //       r1   r2
                     //          r3,       r4
```



9.20 CSG

Изменить знак

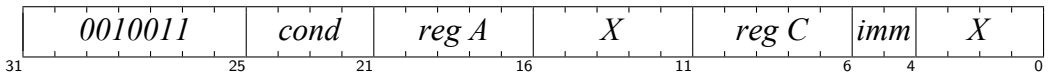


Рис. 22: Машинное представление инструкции CSG

9.20.1 Описание

Изменяет знак содержимого операнда 1 на противоположный (в дополнительном коде) и сохраняет во второй.

9.20.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

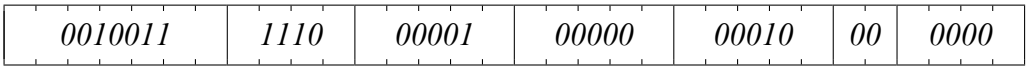
9.20.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение

- Производит запись в регистр.
- Не меняет потока исполнения.

9.20.4 Пример использования:

```
CSG r1 -> r2  //              r1          r2
```

1. 

9.21 INC

Инкремент

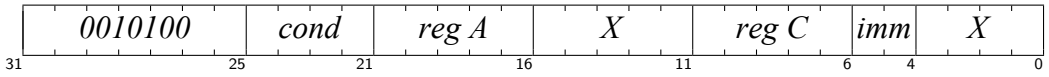


Рис. 23: Машинное представление инструкции INC

9.21.1 Описание

Инкрементирует, то есть увеличивает на единицу содержимое первого операнда и сохраняет во второй.

9.21.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

9.21.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

9.21.4 Пример использования:

```
INC r1 -> r2  //        r1            r2
```

1. | *0010100* | *1110* | *00001* | *00000* | *00010* | *00* | *0000* |

9.22 DEC

Декремент

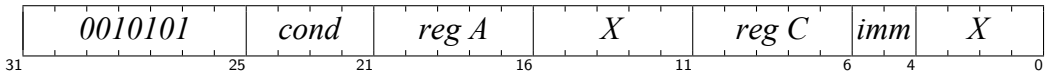


Рис. 24: Машинное представление инструкции DEC

9.22.1 Описание

Декрементирует, то есть уменьшает на единицу содержимое первого операнда и сохраняет во второй.

9.22.2 Флаги, затрагиваемые данной инструкцией:

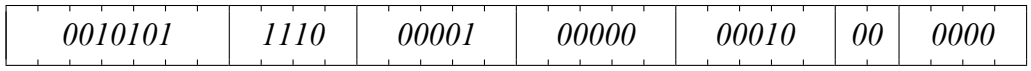
N	Z	C	V
+	+	+	+

9.22.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

9.22.4 Пример использования:

```
DEC r1 -> r2  //        r1            r2
```

1. 

9.23 CMP

Сравнение

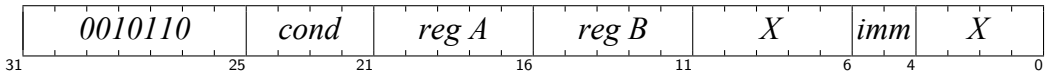


Рис. 25: Машинное представление инструкции CMP

9.23.1 Описание

Производит сравнение двух операндов и выставляет флаги исполнения в соответствии с ним. Эквивалентна разности первого операнда со вторым без сохранения результата. Все условные коды поименованы относительно результата этой инструкции.

9.23.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

9.23.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Не производит запись в регистры, память и периферические устройства.
- Не меняет потока исполнения.

9.23.4 Пример использования:

```
 MP r1, 0x01   //       r1 c 1
DEC_EQ r1 -> r1 //         -
```

1. *0010110* | *1110* | *00001* | *00000* | *00000* | *01* | *0000*
2. *0x01*
3. *0010101* | *0000* | *00001* | *00000* | *00001* | *00* | *0000*

9.24 CMN

Сравнение с обратным знаком

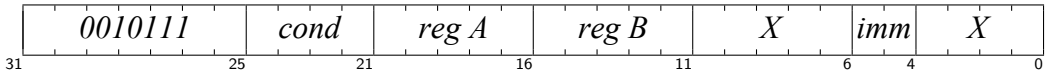


Рис. 26: Машинное представление инструкции CMN

9.24.1 Описание

Производит сравнение первого операнда с вторым операндом с обращённым знаком и выставляет флаги исполнения в соответствии с ним. Эквивалентна сумме операндов без сохранения результата. Все условные коды поименованы относительно результата этой инструкции.

9.24.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

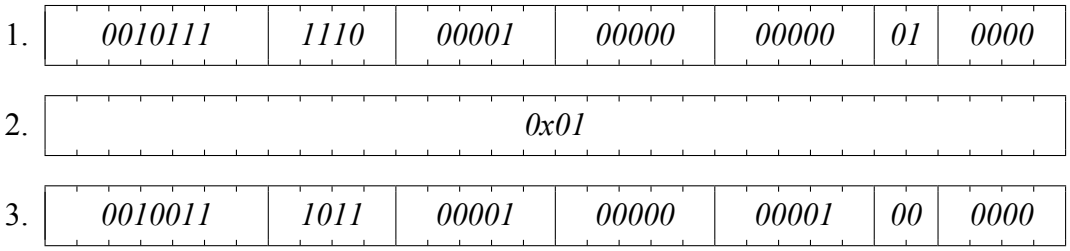
9.24.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.

- Не производит запись в регистры, память и периферические устройства.
- Не меняет потока исполнения.

9.24.4 Пример использования:

```
MN r1, 0x00   //       r1 c 0
CSG_LT r1 -> r1 //              -
```



9.25 TST

Проверка («И»)

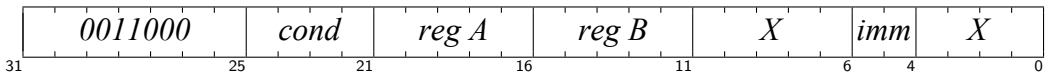


Рис. 27: Машинное представление инструкции TST

9.25.1 Описание

Производит побитовое И двух операндов и выставляет флаг Z в зависимости от результата. Подходит для быстрой проверки по битовой маске (см. пример)

9.25.2 Флаги, затрагиваемые данной инструкцией:

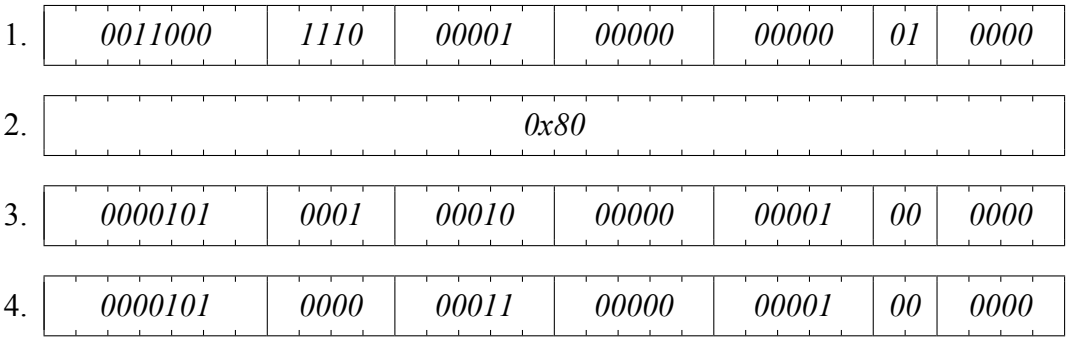
N	Z	C	V
0	+	0	0

9.25.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать до двух мгновенных значений.
- Не производит запись в регистры, память и периферические устройства.
- Не меняет потока исполнения.

9.25.4 Пример использования:

```
TST r1, 0x80   //              r1
INV_NEQ r2 -> r1 //           -          r2   r1
INV_EQ r3 -> r1  //            r3   r1
```



9.26 BR

Прямой переход

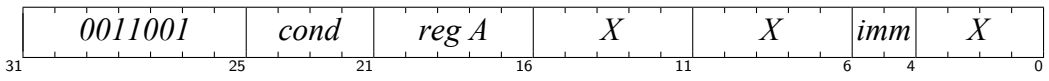


Рис. 28: Машинное представление инструкции BR

9.26.1 Описание

Производит прямой переход по адресу в операнде.

9.26.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

9.26.3 Свойства инструкции:

- Исполнение занимает 4 такта.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр pc.
- Изменяет поток исполнения.

9.26.4 Пример использования:

```
TST r0, 0x80000000 //                r0 32-
BR_NEQ 0x00000000   //            -            0
```

1. | *0011000* | *1110* | *00000* | *00000* | *00000* | *01* | *0000* |
2. | *0x80000000* |
3. | *0011001* | *0001* | *00000* | *00000* | *00000* | *10* | *0000* |
4. | *0x00000000* |

9.27 RBR

Относительный переход

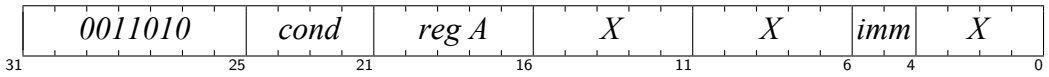


Рис. 29: Машинное представление инструкции RBR

9.27.1 Описание

Производит переход по смещению в операнде относительно счётчика инструкций. Подходит для реализации последовательного сравнения с константой (конструкции типа case)

9.27.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

9.27.3 Свойства инструкции:

- Исполнение занимает 4 такта.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр pc.
- Изменяет поток исполнения.

9.27.4 Пример использования:

```
ADD r0, 0x4000 -> r0    //         r0       4000h
RBR r0                  //              r0
```

1. | *0001110* | *1110* | *00000* | *00000* | *00000* | *01* | *0000* |
2. | *0x4000* |
3. | *0011010* | *1110* | *00000* | *00000* | *00000* | *00* | *0000* |

9.28 BRL

Переход с сохранением адреса возврата

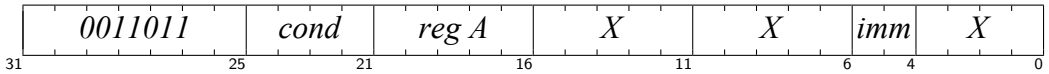


Рис. 30: Машинное представление инструкции BRL

9.28.1 Описание

Сохраняет текущий адрес в lr и производит прямой переход по адресу в первом операнде. Подходит для реализации вызовов процедур.

9.28.2 Флаги, затрагиваемые данной инструкцией:

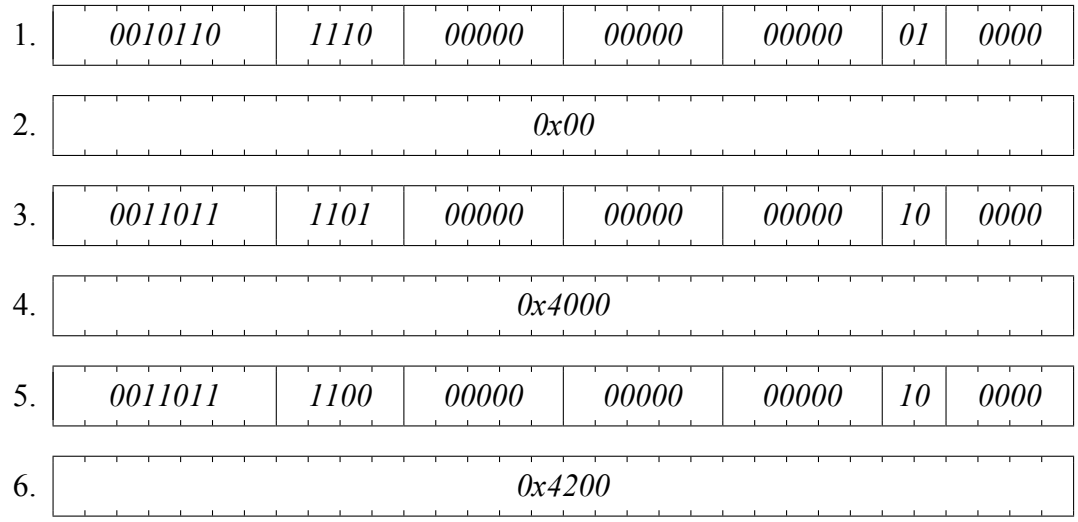
N	Z	C	V
-	-	-	-

9.28.3 Свойства инструкции:

- Исполнение занимает 4 такта.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистры pc и lr.
- Изменяет поток исполнения.

9.28.4 Пример использования:

```
CMP r0, 0x00            //      r0   0
BRL_LE 0x4000           //                -
            0x4000
BRL_GT 0x4200           //                  0x4200
```



9.29 RET

Возврат

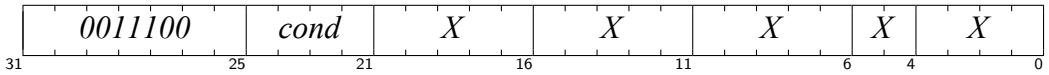


Рис. 31: Машинное представление инструкции RET

9.29.1 Описание

Производит прямой переход по адресу, сохранённому в lr. Предназначена для организации возврата из процедур.

9.29.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

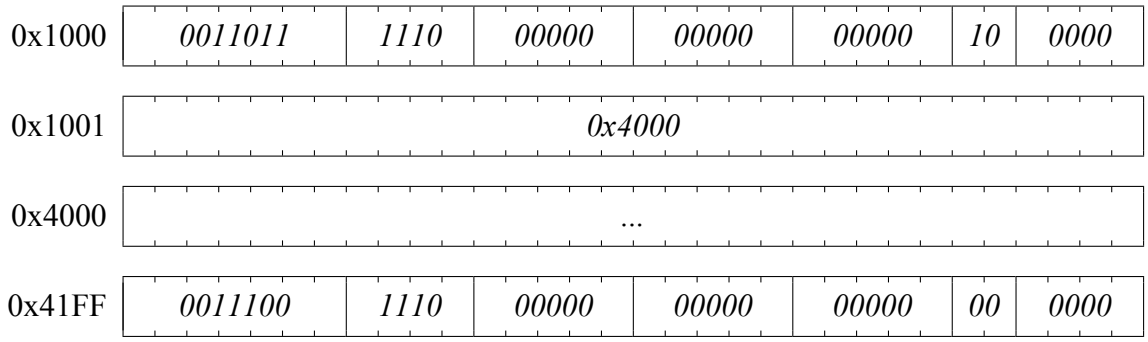
9.29.3 Свойства инструкции:

- Исполнение занимает 4 такта.
- Может исполняться условно.
- Не может использовать мгновенные значения.

- Производит запись в регистр pc.
- Изменяет поток исполнения.

9.29.4 Пример использования:

```
0x1000: BRL 0x4000      //              0x4000
0x4000: .....           //
0x41FF: RET             //          -      ( 0x1002)
```



9.30 LDR

Чтение из ОЗУ

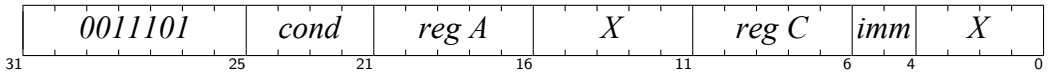


Рис. 32: Машинное представление инструкции LDR

9.30.1 Описание

Читает содержимое ОЗУ по адресу в первом операнде и сохраняет его во второй операнд

9.30.2 Флаги, затрагиваемые данной инструкцией:

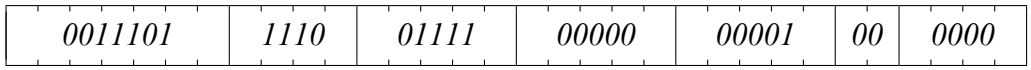
N	Z	C	V
-	-	-	-

9.30.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

9.30.4 Пример использования:

```
LDR r15 -> r1  //                r15          r1
```

1. 

9.31 STR

Запись в ОЗУ

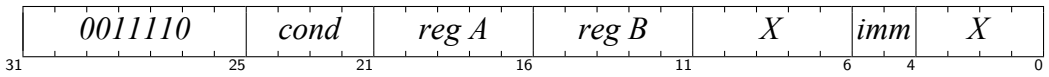


Рис. 33: Машинное представление инструкции STR

9.31.1 Описание

Записывает содержимое второго операнда в ОЗУ по адресу в первом операнде.

9.31.2 Флаги, затрагиваемые данной инструкцией:

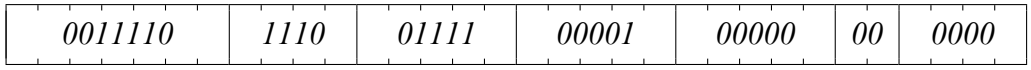
N	Z	C	V
-	-	-	-

9.31.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр и ОЗУ.
- Не меняет потока исполнения.

9.31.4 Пример использования:

```
STR r15, r1  //     r1         r15
```

1. 

9.32 IN

Чтение из периферийного регистра

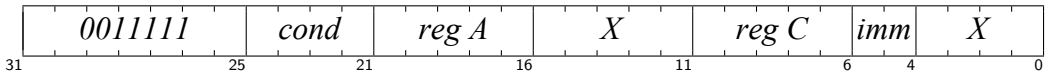


Рис. 34: Машинное представление инструкции IN

9.32.1 Описание

Читает содержимое периферийного регистра, находящегося по адресу в первом операнде и сохраняет его во второй операнд

9.32.2 Флаги, затрагиваемые данной инструкцией:

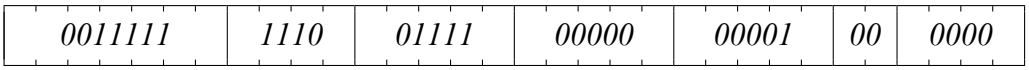
N	Z	C	V
-	-	-	-

9.32.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

9.32.4 Пример использования:

```
IN r15 -> r1  //                    r15          r1
```

1. 

9.33 OUT

Запись в периферийный регистр

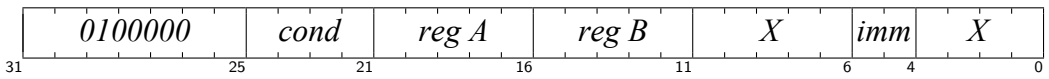


Рис. 35: Машинное представление инструкции STR

9.33.1 Описание

Записывает содержимое второго операнда в периферийный регистр, находящийся по адресу в первом операнде.

9.33.2 Флаги, затрагиваемые данной инструкцией:

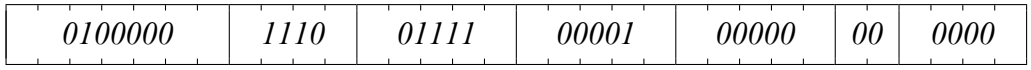
N	Z	C	V
-	-	-	-

9.33.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр и периферию.
- Не меняет потока исполнения.

9.33.4 Пример использования:

```
OUT r15, r1  //      r1                  r15
```

1. 

9.34 MOVS

Копирование регистра

Рис. 36: Машинное представление инструкции MOVS

9.34.1 Описание

Копирует содержимое первого операнда во второй.

9.34.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

9.34.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

9.34.4 Пример использования:

```
MOVS r0 -> r15  //          r0   r15
```

1.

0100001	*1110*	*00000*	*00000*	*01111*	*00*	*0000*

9.35 MOV

Копирование двух регистров

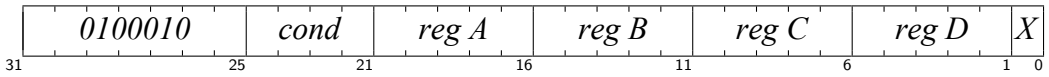


Рис. 37: Машинное представление инструкции MOV

9.35.1 Описание

Копирует содержимое первого операнда в третий, а содержимое второго - в четвёртый.

9.35.2 Флаги, затрагиваемые данной инструкцией:

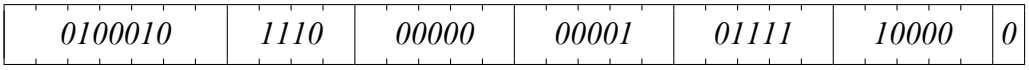
N	Z	C	V
-	-	-	-

9.35.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.
- Не может использовать мгновенные значения.
- Производит запись в регистр.
- Не меняет потока исполнения.

9.35.4 Пример использования:

```
MOV r0, r1 -> r15, r16  //          r0   r15, r1   r16
```

1. 

Часть VI

Приложение 2. Исходный код

10 Структура

Данный проект содержит в себе две части:

1. Процессор УП-1.
2. MultiplierGenerator.

10.1 Процессор УП-1

RTL-описание процессора с RISC-архитектурой. Язык описания - Verilog (синтезируемая часть стандарта).

Проекту принадлежат следующие файлы:

1. adder.v - Сумматор с параллельным переносом
2. alu.v - Арифметико-логическое устройство
3. execute.v - Стадия «Execute» конвейера
4. gpio.v - Периферийное устройство «Контроллер GPIO»
5. gpio_mux.v - Периферийное устройство «Выходной мультиплексор»
6. insn_decoder.v - Стадия «Decode» конвейера
7. memory_op.v - Стадия «Memory/Periph» конвейера
8. pipeline_interface.v - Стадия «Interface» конвейера
9. ram.v - ОЗУ процессора
10. register_wb.v - Стадия «Register WB» конвейера
11. regs.v - Регистровый файл процессора

12. shift.v - Комбинированный регистр быстрого сдвига/вращения
13. test_periph_assembly.v - Модуль верхнего уровня для периферических устройств
14. test_pipeline_assembly.v - Модуль верхнего уровня для конвейера
15. test_processor_assembly.v - Модуль верхнего уровня для процессорной системы
16. main.v - Главный тестовый модуль процессора, с двумя тестовыми программами

10.2 MultiplierGenerator

Генератор умножителей по схеме Дадды. Язык программирования - C++ (стандарт C++14).

Проекту принадлежат следующие файлы:

1. Gate.hpp - Главная логика сборки умножителей и необходимые для этого примитивы (заголовочный файл с кодом).
2. Main.cpp - Точка входа приложения. Главная логика работы приложения, а именно порядок приёма аргументов и консольный интерфейс.
3. testcase.v - Схема для тестирования сгенерированных умножителей. Язык описания - Verilog.

11 Исходные коды

11.1 Процессор УП-1

11.1.1 adder.v

```
1  `timescale 1 ns / 100 ps
2
3  module fa_pg(a, b, cin, s, p, g);
4      input a, b, cin;
```

```
5
6       output wire s, p, g;
7
8       wire w1;
9
10      xor x1(p, a, b);
11      xor x2(s, p, cin);
12
13      and a1(g, a, b);
14  //    or #1 o1(p, a, b);
15  endmodule
16
17  module cla_4(a, b, cin, s, pg, gg);
18      input [3:0] a;
19      input [3:0] b;
20      output wire [3:0] s;
21
22      input cin;
23      output wire pg, gg;
24
25      wire [3:0] p;
26      wire [3:0] g;
27      wire [2:0] c;
28
29      fa_pg fa0(a[0], b[0],  cin, s[0], p[0], g[0]);
30      fa_pg fa1(a[1], b[1], c[0], s[1], p[1], g[1]);
31      fa_pg fa2(a[2], b[2], c[1], s[2], p[2], g[2]);
32      fa_pg fa3(a[3], b[3], c[2], s[3], p[3], g[3]);
33
34      assign c[0] = g[0] | p[0]&cin;
35      assign c[1] = g[1] | g[0]&p[1] | cin&p[0]&p[1];
36      assign c[2] = g[2] | g[1]&p[2] | g[0]&p[1]&p[2] | cin&p[0]&p[1]&p[2];
37
38      assign pg = p[0]&p[1]&p[2]&p[3];
39      assign gg = g[3] | g[2]&p[3] | g[1]&p[3]&p[2] | g[0]&p[3]&p[2]&p[1];
40      //assign cout = gg | cin&pg;
41  endmodule
42
43  module cla_16(a, b, cin, s, pg, gg);
44      input [15:0] a;
45      input [15:0] b;
46      output wire [15:0] s;
47
48      input cin;
49      output wire pg, gg;
50
51      wire [3:0] p;
52      wire [3:0] g;
53      wire [2:0] c;
```

```
54
55      cla_4 cla0(a[3:0],     b[3:0],  cin,   s[3:0], p[0], g[0]);
56      cla_4 cla1(a[7:4],     b[7:4], c[0],   s[7:4], p[1], g[1]);
57      cla_4 cla2(a[11:8],   b[11:8], c[1],  s[11:8], p[2], g[2]);
58      cla_4 cla3(a[15:12], b[15:12], c[2], s[15:12], p[3], g[3]);
59
60      assign c[0] = g[0] | p[0]&cin;
61      assign c[1] = g[1] | g[0]&p[1] | cin&p[0]&p[1];
62      assign c[2] = g[2] | g[1]&p[2] | g[0]&p[1]&p[2] | cin&p[0]&p[1]&p[2];
63
64      assign pg = p[0]&p[1]&p[2]&p[3];
65      assign gg = g[3] | g[2]&p[3] | g[1]&p[3]&p[2] | g[0]&p[3]&p[2]&p[1];
66      //assign cout = gg | cin&pg;
67  endmodule
68
69  module cla_32(a, b, cin, s, cout);
70      input [31:0] a;
71      input [31:0] b;
72      output wire [31:0] s;
73
74      input cin;
75      output wire cout;
76
77      wire [3:0] p;
78      wire [3:0] g;
79      wire [2:0] c;
80
81      cla_16 cla0(a[15:0],   b[15:0],  cin,   s[15:0], p[0], g[0]);
82      cla_16 cla1(a[31:16], b[31:16], c[0],  s[31:16], p[1], g[1]);
83
84      assign c[0] = g[0] | p[0]&cin;
85      assign cout = g[1] | g[0]&p[1] | cin&p[0]&p[1];
86  endmodule
87
88  module cla_64(a, b, cin, s, cout);
89      input [63:0] a;
90      input [63:0] b;
91      output wire [63:0] s;
92
93      input cin;
94      wire pg, gg;
95      output wire cout;
96
97      wire [3:0] p;
98      wire [3:0] g;
99      wire [2:0] c;
100
101     cla_16 cla0(a[15:0],   b[15:0],  cin,  s[15:0], p[0], g[0]);
102     cla_16 cla1(a[31:16], b[31:16], c[0], s[31:16], p[1], g[1]);
```

```
103     cla_16 cla2(a[47:32], b[47:32], c[1], s[47:32], p[2], g[2]);
104     cla_16 cla3(a[63:48], b[63:48], c[2], s[63:48], p[3], g[3]);
105
106     assign c[0] = g[0] | p[0]&cin;
107     assign c[1] = g[1] | g[0]&p[1] | cin&p[0]&p[1];
108     assign c[2] = g[2] | g[1]&p[2] | g[0]&p[1]&p[2] | cin&p[0]&p[1]&p[2];
109
110     assign pg = p[0]&p[1]&p[2]&p[3];
111     assign gg = g[3] | g[2]&p[3] | g[1]&p[3]&p[2] | g[0]&p[3]&p[2]&p[1];
112     assign cout = gg | cin&pg;
113 endmodule
```

11.1.2 alu.v

```
1  `include "mult.v"
2  `include "adder.v"
3  `include "shift.v"
4
5  module addsub_32(q, a, b, sub, ov, sov, z);
6      input [31:0] a, b;
7      input sub;
8
9      output wire [31:0] q;
10     output ov, sov, z;
11
12     wire [31:0] bm = sub ? ~b : b;
13
14     cla_32 cla0(a, bm, sub, q, ov);
15
16     assign z = ~|q;
17
18     assign sov = (!a[31]) && (!b[31]) ? ov : (a[31] && b[31] ? ~q[31] : 1'b0);
19
20 endmodule
21
22 module mul_32(q1, q2, ov, z, a, b);
23     input [31:0] a, b;
24
25     output wire [31:0] q1, q2;
26     output ov, z;
27
28     wire [63:0] q;
29     assign q1 = q[31:0];
30     assign q2 = q[63:32];
31
32     mult_32 m0( a, b, q);
33
34     assign ov = |(q2);
35     assign z = ~|({q2, q1});
```

```
36  endmodule
37
38  module bitwise_32(q, z, a, b, op);
39      input [31:0] a, b;
40      input [2:0] op;
41
42      output reg [31:0] q;
43      output wire z;
44
45      assign z = ~|q;
46
47      always @* begin
48          case(op)
49              3'b000: q = ~ a; //NOT A
50              3'b001: q = a & b; // A AND B
51              3'b010: q = a | b; // A OR B
52              3'b011: q = a ^ b; // A XOR B
53              3'b100: q = ~(a & b); // A NAND B
54              3'b101: q = ~(a | b); // A NOR B
55              3'b110: q = ~(a ^ b); // A XNOR B
56              3'b111: q = ~ b; // NOT B (placeholder)
57          endcase
58      end
59  endmodule
60
61  module alu32_2x2(q0, q1, st, a, b, op);
62      input [31:0] a, b;
63      output reg [31:0] q0, q1;
64      //wire [31:0] q0, q1;
65
66      input [7:0] op;
67      output reg [3:0] st; //0 - V, 1 - C, 2 - Z, 3 - N
68
69      wire [31:0] addsub;
70      reg [31:0] addsub_a, addsub_b;
71      wire addsub_z, addsub_ov, addsub_sov;
72      reg subtract;
73      addsub_32 as0(addsub, addsub_a, addsub_b, subtract, addsub_ov, addsub_sov,
            addsub_z);
74      wire [3:0] addsub_st = {addsub[31], addsub_z, addsub_ov, addsub_sov};
75
76
77      wire [31:0] shift;
78      wire shift_z, shift_ov;
79      reg rotate, left, arithmetic;
80      bshift_32 sh0(shift, shift_ov, shift_z, a, b[4:0], rotate, left, arithmetic);
81      wire [3:0] shift_st = {shift[31], shift_z, shift_ov, 1'b0};
82
83      wire [31:0] mull, mulh;
```

```
84      wire mul_z, mul_ov;
85      mul_32 mul0(mull, mulh, mul_ov, mul_z, a, b);
86      wire [3:0] mul_st = {1'b0, mul_z, mul_ov, 1'b0};
87
88      wire [31:0] bws;
89      wire bws_z;
90      reg [2:0] b_op;
91      bitwise_32 bw0(bws, bws_z, a, b, b_op);
92      wire [3:0] bws_st = {1'b0, bws_z, 2'b0};
93
94
95      always @* begin
96          case(op)
97          8'h00: begin //NOP
98              q0 = a;
99              q1 = b;
100
101             st = 4'b0;
102         end
103         8'h01: begin //ADD
104             subtract = 0;
105             addsub_a = a;
106             addsub_b = b;
107             q0 = addsub;
108             q1 = 32'b0;
109
110             st = addsub_st;
111         end
112         8'h02: begin //SUB
113             subtract = 1;
114             addsub_a = a;
115             addsub_b = b;
116             q0 = addsub;
117             q1 = 32'b0;
118
119             st = addsub_st;
120         end
121         8'h03: begin //CPL
122             subtract = 1;
123             addsub_a = 32'b0;
124             addsub_b = a;
125             q0 = addsub;
126             q1 = 32'b0;
127
128             st = addsub_st;
129         end
130         8'h04: begin //MUL
131             q0 = mull;
132             q1 = mulh;
```

```
133
134             st = mul_st;
135         end
136         8'h05: begin //SHR
137             rotate = 0;
138             left = 0;
139             arithmetic = 0;
140             q0 = shift;
141             q1 = 32'b0;
142
143             st = shift_st;
144         end
145         8'h06: begin // SHL
146             rotate = 0;
147             left = 1;
148             arithmetic = 0;
149             q0 = shift;
150             q1 = 32'b0;
151
152             st = shift_st;
153         end
154         8'h07: begin // SAR
155             rotate = 0;
156             left = 0;
157             arithmetic = 1;
158             q0 = shift;
159             q1 = 32'b0;
160
161             st = shift_st;
162         end
163         8'h08: begin // SAL
164             rotate = 0;
165             left = 1;
166             arithmetic = 1;
167             q0 = shift;
168             q1 = 32'b0;
169
170             st = shift_st;
171         end
172         8'h09: begin // ROR
173             rotate = 1;
174             left = 0;
175             arithmetic = 0;
176             q0 = shift;
177             q1 = 32'b0;
178
179             st = shift_st;
180         end
181         8'h0A: begin // ROL
```

```
182             rotate = 1;
183             left = 1;
184             arithmetic = 0;
185             q0 = shift;
186             q1 = 32'b0;
187 
188             st = shift_st;
189         end
190         8'h0B: begin //NOT
191             b_op = 0;
192             q0 = bws;
193             q1 = 32'b0;
194 
195             st = bws_st;
196         end
197         8'h0C: begin //AND
198             b_op = 1;
199             q0 = bws;
200             q1 = 32'b0;
201 
202             st = bws_st;
203         end
204         8'h0D: begin //OR
205             b_op = 2;
206             q0 = bws;
207             q1 = 32'b0;
208 
209             st = bws_st;
210         end
211         8'h0E: begin //XOR
212             b_op = 3;
213             q0 = bws;
214             q1 = 32'b0;
215 
216             st = bws_st;
217         end
218         8'h0F: begin //NAND
219             b_op = 4;
220             q0 = bws;
221             q1 = 32'b0;
222 
223             st = bws_st;
224         end
225         8'h10: begin //NOR
226             b_op = 5;
227             q0 = bws;
228             q1 = 32'b0;
229 
230             st = bws_st;
```

```
231             end
232             8'h11: begin //XNOR
233                 b_op = 6;
234                 q0 = bws;
235                 q1 = 32'b0;
236 
237                 st = bws_st;
238             end
239             /*default: begin //invalid
240                 q0 = 32'bz;
241                 q1 = 32'bz;
242                 st = 4'bz;
243             end*/
244         endcase
245         end
246 endmodule
```

11.1.3 execute.v

```
1  `timescale 1 ns / 100 ps
2 
3  `include "alu.v"
4 
5  module cond_calc(cr, cc, n, z, c, v);
6      input [3:0] cc;
7      input n, z, c, v;
8 
9      output reg cr;
10 
11     always @* begin
12         case(cc)
13             4'b0000: cr = z == 1'b1; //EQ - equal
14             4'b0001: cr = z == 1'b0; //NEQ - not equal
15             4'b0010: cr = c == 1'b1; //HS - higher or same (unsigned)
16             4'b0011: cr = c == 1'b0; //LO - strictly lower (unsigned)
17             4'b0100: cr = n == 1'b1; //NEG - negative
18             4'b0101: cr = n == 1'b0; //POS - positive
19             4'b0110: cr = v == 1'b1; //SOV - signed overflow
20             4'b0111: cr = v == 1'b0; //NSOV - no signed overflow
21             4'b1000: cr = (c == 1'b1) && (z == 1'b0); //HI - strictly higher (
                   unsigned)
22             4'b1001: cr = (c == 1'b0) || (z == 1'b1); //LS - lower or same (
                   unsigned)
23             4'b1010: cr = n == v; //GE - greater or equal (signed)
24             4'b1011: cr = n != v; //LT - strictly less (signed)
25             4'b1100: cr = (z == 1'b0) && (n == v); //GT - strictly greater (
                   signed)
26             4'b1101: cr = (z == 1'b1) || (n != v); //LE - lower or equal (signed)
27             4'b1110: cr = 1'b1; //AL - always
```

```
28              4'b1111: cr = 1'b0; //NV - never
29          endcase
30      end
31  endmodule
32
33  module status_register_adaptor(st, stwr, n, z, c, v, cc);
34      input n, z, c, v;
35      input cc;
36
37      output [31:0] st;
38      output stwr;
39
40      assign stwr = cc;
41      assign st[3:0] = {n, z, c, v};
42      assign st[31:4] = 28'b0;
43  endmodule
44
45  module execute_stage_passthrough(qm_a1, qm_a2, qm_r1_op, qm_r2_op, qr_a1, qr_a2,
        qr_op, m_a1, m_a2, m_r1_op, m_r2_op, r_a1, r_a2, r_op, clk, rst);
46      input [31:0] m_a1, m_a2; //(mem_op)
47      input [3:0] m_r1_op, m_r2_op; //(mem_op)
48
49      input [4:0] r_a1, r_a2; //(reg_wb)
50      input [3:0] r_op; //(reg_wb)
51
52      input clk, rst;
53
54      output reg [31:0] qm_a1, qm_a2; //(mem_op)
55      output reg [3:0] qm_r1_op, qm_r2_op; //(mem_op)
56
57      output reg [4:0] qr_a1, qr_a2; //(reg_wb)
58      output reg [3:0] qr_op; //(reg_wb)
59
60      always @(posedge clk or posedge rst) begin
61          if(rst) begin
62              qm_a1 <= 32'b0; qm_a2 <= 32'b0;
63              qm_r1_op <= 4'b0; qm_r2_op <= 4'b0;
64              qr_a1 <= 5'b0; qr_a2 <= 5'b0;
65              qr_op <= 4'b0;
66          end
67          else begin
68              qm_a1 <= m_a1; qm_a2 <= m_a2;
69              qm_r1_op <= m_r1_op; qm_r2_op <= m_r2_op;
70              qr_a1 <= r_a1; qr_a2 <= r_a2;
71              qr_op <= r_op;
72          end
73      end
74  endmodule
75
```

```
76
77
78  module execute(r1, r2, cres, n, z, c, v, cc, a, b, alu_op, is_cond, cond,
        write_flags, st, swp, clk, rst);
79      input [31:0] a, b; //operands
80      input [31:0] st; //status register
81      input [7:0] alu_op; // alu operation
82      input is_cond; //is a conditional command signal
83      input [3:0] cond; //cc
84      input [3:0] write_flags; //write n/z/c/v
85      input swp; //swap ops?
86      input clk, rst;
87
88      output reg [31:0] r1, r2; //results, sync
89      output wire n, z, c, v; //flags, async
90      output wire cc; //write flags, async
91      output reg cres; //conditional results, sync
92
93      wire [31:0] ra = swp ? b : a;
94      wire [31:0] rb = swp ? a : b;
95
96      wire [31:0] alu_q1, alu_q2;
97      wire alu_n, alu_z, alu_c, alu_v;
98      wire [7:0] alu_op;
99      alu32_2x2 alu0(alu_q1, alu_q2, {alu_n, alu_z, alu_c, alu_v}, ra, rb, alu_op);
100
101     wire cond_n = st[3], cond_z = st[2], cond_c = st[1], cond_v = st[0];
102     wire cond_res;
103     cond_calc cond0(cond_res, cond, cond_n, cond_z, cond_c, cond_v);
104
105     assign cc = (write_flags != 4'b0) && (is_cond && cond_res);
106     assign n = write_flags[3] ? alu_n : cond_n;
107     assign z = write_flags[2] ? alu_z : cond_z;
108     assign c = write_flags[1] ? alu_c : cond_c;
109     assign v = write_flags[0] ? alu_v : cond_v;
110
111     always @(posedge clk or posedge rst) begin
112         if(rst) begin
113             r1 <= 31'b0;
114             r2 <= 31'b0;
115             cres <= 1'b0;
116         end
117         else begin
118             r1 <= alu_q1;
119             r2 <= alu_q2;
120             if(is_cond) cres <= cond_res;
121             else cres <= 1'b1;
122         end
123     end
```

```
124 endmodule
```

11.1.4 gpio.v

```
1  `timescale 1 ns / 100 ps
2
3  module gpio(gpio_out, gpio_in, gpio_dir, addr, sys_w_addr, sys_r_addr, sys_w_line
       , sys_r_line, sys_w, sys_r, rst, clk);
4      //control signals
5      input [31:0] gpio_in;
6      output wire [31:0] gpio_out;
7      output wire [31:0] gpio_dir;
8
9      //address, constant
10     input [31:0] addr;
11
12     //peripheral bus
13     input [31:0] sys_w_addr;
14     input [31:0] sys_r_addr;
15     input [31:0] sys_w_line;
16     output reg [31:0] sys_r_line;
17     input sys_w;
18     input sys_r;
19
20     //generic
21     input clk;
22     input rst;
23
24     //control regs
25     reg [31:0] direction; // 1 for out, 0 for in
26     reg [31:0] value; //default
27
28     assign gpio_out = value;
29     assign gpio_dir = direction;
30
31     always @(posedge clk or posedge rst) begin
32         if(rst) begin
33             direction <= 32'b0;
34             value <= 32'b0;
35             sys_r_line <= 32'bz;
36         end
37         else begin
38             #1;
39             if(sys_r) begin //read requested
40                 if(sys_r_addr[31:1] == addr[31:1]) begin //if r addr is same
41                     if(sys_r_addr[0]) begin //high part, direction
42                         sys_r_line <= direction;
43                     end else begin //low part, read value
44                         sys_r_line <= gpio_in;
```

```
45                      end
46                  end else begin
47                      sys_r_line = 32'bz; //don't scramble other devices
48                  end
49              end else begin
50                  sys_r_line = 32'bz; //minimize power consumption
51              end
52              if(sys_w) begin //write requested
53                  if(sys_w_addr[31:1] == addr[31:1]) begin //if w addr is same
54                      if(sys_w_addr[0]) begin //high part, direction
55                          direction <= sys_w_line;
56                      end else begin //low part, write value
57                          value <= sys_w_line;
58                      end
59                  end
60              end
61          end
62      end
63  endmodule
```

11.1.5 gpio_mux.v

```
1   `timescale 1 ns / 100 ps
2
3   module gpio_mux(pins, func0_in, func1_in, func2_in, func3_in, func0_out,
        func1_out, func2_out, func3_out, func0_dir, func1_dir, func2_dir, func3_dir,
        addr, sys_w_addr, sys_r_addr, sys_w_line, sys_r_line, sys_w, sys_r, rst, clk)
        ;
4       inout [31:0] pins;
5
6       //functions
7       //output signals
8       input [31:0] func0_out;
9       input [31:0] func1_out;
10      input [31:0] func2_out;
11      input [31:0] func3_out;
12
13      //input signals
14      output wire [31:0] func0_in;
15      output wire [31:0] func1_in;
16      output wire [31:0] func2_in;
17      output wire [31:0] func3_in;
18
19      //direction signals, 1 - out, 0 - in
20      input [31:0] func0_dir;
21      input [31:0] func1_dir;
22      input [31:0] func2_dir;
23      input [31:0] func3_dir;
24
```

```
25      //address, constant
26      input [31:0] addr;
27
28      //peripheral bus
29      input [31:0] sys_w_addr;
30      input [31:0] sys_r_addr;
31      input [31:0] sys_w_line;
32      output reg [31:0] sys_r_line;
33      input sys_w;
34      input sys_r;
35
36      //generic
37      input clk;
38      input rst;
39
40      //pin control register;
41      reg [63:0] control;
42
43      //generate muxes for every pin
44      genvar i;
45      generate
46      for(i = 0; i < 32; i = i + 1) begin : pin_mux
47          wire [1:0] pin_control = control[(i*2 + 1):(i*2)];
48          wire pin_out = pin_control == 0 ? func0_out[i] : (pin_control == 1 ?
                func1_out[i] : (pin_control == 2 ? func2_out[i] : func3_out[i]));
49          wire pin_dir = pin_control == 0 ? func0_dir[i] : (pin_control == 1 ?
                func1_dir[i] : (pin_control == 2 ? func2_dir[i] : func3_dir[i]));
50          assign pins[i] = pin_dir == 1 ? pin_out : 1'bz;
51          assign func0_in[i] = pin_dir == 1 ? pin_out : pins[i];
52          assign func1_in[i] = pin_dir == 1 ? pin_out : pins[i];
53          assign func2_in[i] = pin_dir == 1 ? pin_out : pins[i];
54          assign func3_in[i] = pin_dir == 1 ? pin_out : pins[i];
55      end
56      endgenerate
57
58      always @(posedge clk or posedge rst) begin
59          #1;
60          if(rst) begin
61              control = 64'b0;
62          end
63          else begin
64              if(sys_r) begin //read requested
65                  if(sys_r_addr[31:1] == addr[31:1]) begin //if r addr is same
66                      if(sys_r_addr[0]) begin //high part
67                          sys_r_line <= control[63:32];
68                      end else begin //low part
69                          sys_r_line <= control[31:0];
70                      end
71                  end else begin
```

```
72                      sys_r_line = 32'bz; //don't scramble other devices
73                  end
74              end else begin
75                  sys_r_line = 32'bz; //minimize power consumption
76              end
77              if(sys_w) begin //write requested
78                  if(sys_w_addr[31:1] == addr[31:1]) begin //if w addr is same
79                      if(sys_w_addr[0]) begin //high part
80                          control[63:32] <= sys_w_line;
81                      end else begin //low part
82                          control[31:0] <= sys_w_line;
83                      end
84                  end
85              end
86          end
87      end
88  endmodule
```

11.1.6 insn_decoder.v

```
1   `timescale 1 ns / 100 ps
2
3   //fixed version
4
5   /*module insn_type_lookup(type, opcode);
6       input [6:0] opcode;
7       output [2:0] type;
8
9       always @(a or b) begin
10          case(opcode) //full_case parallel_case
11              0: type <= 0;
12              1: type <= 0;
13              //...
14          endcase
15      end
16  endmodule*/
17
18  module reg_hazard_checker(ex_hazard, mem_hazard, reg_hazard, ex_r1_a, ex_r2_a,
        ex_r_op, ex_proceed, mem_r1_a, mem_r2_a, mem_r_op, mem_proceed, reg_r1_a,
        reg_r2_a, reg_write, dec_r1_addr, dec_r2_addr, dec_r_read);
19      output wire ex_hazard;
20      output wire mem_hazard;
21      output wire reg_hazard;
22
23      input [4:0] ex_r1_a, ex_r2_a;
24      input [3:0] ex_r_op;
25      input ex_proceed;
26
27      input [4:0] mem_r1_a, mem_r2_a;
```

```
28      input [3:0] mem_r_op;
29      input mem_proceed;
30
31      input [4:0] reg_r1_a, reg_r2_a;
32      input [1:0] reg_write;
33
34      input [4:0] dec_r1_addr, dec_r2_addr;
35      input [1:0] dec_r_read;
36
37      wire dec_r1_read_comp = dec_r_read[0];
38      wire dec_r2_read_comp = dec_r_read[1];
39
40      wire ex_r1_op_comp = (ex_r_op == 1) || (ex_r_op == 2) || (ex_r_op == 3);
41      wire ex_r2_op_comp = (ex_r_op == 4) || (ex_r_op == 5) || (ex_r_op == 6);
42      wire ex_r1r2_op_comp = (ex_r_op == 7) || (ex_r_op == 8);
43
44      wire ex_r1_comp = (ex_r1_a == dec_r1_addr);
45      wire ex_r2_comp = (ex_r2_a == dec_r2_addr);
46      wire ex_r1r2_comp = (ex_r1_a == dec_r2_addr);
47      wire ex_r2r1_comp = (ex_r2_a == dec_r1_addr);
48
49      wire ex_hazard_r1 = ((ex_r1_op_comp || ex_r1r2_op_comp) && ex_r1_comp &&
            dec_r1_read_comp);
50      wire ex_hazard_r2 = ((ex_r2_op_comp || ex_r1r2_op_comp) && ex_r2_comp &&
            dec_r2_read_comp);
51      wire ex_hazard_r1r2 = ((ex_r1_op_comp || ex_r1r2_op_comp) && ex_r1r2_comp &&
            dec_r2_read_comp);
52      wire ex_hazard_r2r1 = ((ex_r2_op_comp || ex_r1r2_op_comp) && ex_r2r1_comp &&
            dec_r1_read_comp);
53
54      assign ex_hazard =  (ex_hazard_r1 || ex_hazard_r2 || ex_hazard_r1r2 ||
            ex_hazard_r2r1) && ex_proceed;
55
56      wire mem_r1_op_comp = (mem_r_op == 1) || (mem_r_op == 2) || (mem_r_op == 3);
57      wire mem_r2_op_comp = (mem_r_op == 4) || (mem_r_op == 5) || (mem_r_op == 6);
58      wire mem_r1r2_op_comp = (mem_r_op == 7) || (mem_r_op == 8);
59
60      wire mem_r1_comp = (mem_r1_a == dec_r1_addr);
61      wire mem_r2_comp = (mem_r2_a == dec_r2_addr);
62      wire mem_r1r2_comp = (mem_r1_a == dec_r2_addr);
63      wire mem_r2r1_comp = (mem_r2_a == dec_r1_addr);
64
65      wire mem_hazard_r1 = ((mem_r1_op_comp || mem_r1r2_op_comp) && mem_r1_comp &&
            dec_r1_read_comp);
66      wire mem_hazard_r2 = ((mem_r2_op_comp || mem_r1r2_op_comp) && mem_r2_comp &&
            dec_r2_read_comp);
67      wire mem_hazard_r1r2 = ((mem_r1_op_comp || mem_r1r2_op_comp) && mem_r1r2_comp
            && dec_r2_read_comp);
68      wire mem_hazard_r2r1 = ((mem_r2_op_comp || mem_r1r2_op_comp) && mem_r2r1_comp
```

```
        && dec_r1_read_comp);
69
70      assign mem_hazard =  (mem_hazard_r1 || mem_hazard_r2 || mem_hazard_r1r2 ||
            mem_hazard_r2r1) && mem_proceed;
71
72      wire reg_r1_write_comp = reg_write[0];
73      wire reg_r2_write_comp = reg_write[1];
74
75      wire reg_r1_comp = (reg_r1_a == dec_r1_addr);
76      wire reg_r2_comp = (reg_r2_a == dec_r2_addr);
77      wire reg_r1r2_comp = (reg_r1_a == dec_r2_addr);
78      wire reg_r2r1_comp = (reg_r2_a == dec_r1_addr);
79
80      wire reg_hazard_r1 = (reg_r1_write_comp && reg_r1_comp && dec_r1_read_comp);
81      wire reg_hazard_r2 = (reg_r2_write_comp && reg_r2_comp && dec_r2_read_comp);
82      wire reg_hazard_r1r2 = (reg_r1_write_comp && reg_r1r2_comp &&
            dec_r2_read_comp);
83      wire reg_hazard_r2r1 = (reg_r2_write_comp && reg_r2r1_comp &&
            dec_r1_read_comp);
84
85      assign reg_hazard =  reg_hazard_r1 || reg_hazard_r2 || reg_hazard_r1r2 ||
            reg_hazard_r2r1;
86
87  endmodule
88
89  module insn_decoder( e_a, e_b, e_alu_op, e_is_cond, e_cond, e_write_flags, e_swp,
        m_a1, m_a2, m_r1_op, m_r2_op, r_a1, r_a2, r_op, d_pass, d_pcincr, r_r1_addr,
        r_r2_addr, r_read, word, r1, r2, hazard, rst, clk);
90      output reg [31:0] e_a, e_b;
91      output reg [7:0] e_alu_op;
92      output reg [3:0] e_cond;
93      output reg [3:0] e_write_flags;
94      output reg e_swp;
95      output reg e_is_cond;
96
97      output reg [31:0] m_a1, m_a2;
98      output reg [3:0] m_r1_op, m_r2_op;
99
100     output reg [4:0] r_a1, r_a2;
101     output reg [3:0] r_op;
102
103     output reg d_pass;
104     output reg d_pcincr;
105
106     output reg [4:0] r_r1_addr, r_r2_addr;
107     output reg [1:0] r_read;
108
109     input [31:0] word;
110     input [31:0] r1, r2;
```

```
111     input hazard;
112     input rst, clk;
113
114     reg [7:0] state1;
115     reg fetch;
116     reg reg_fetch;
117     reg [3:0] delay_counter;
118     reg [2:0] imm_action; // 000 - nop, 001 - imm1 -> b, 010 - imm1 -> a, 011 {
            imm1, imm2} -> {a,b}, 100 - nop? 101..111 - as 001..011 but a ~ m_a2, b ~
             m_a1
119     //reg [1:0] imm_counter;
120     reg [7:0] old_state1_imm;
121     reg old_pass_imm, old_fetch_imm, old_pcincr_imm;
122     reg [1:0] r_to_mem; //00 a,b; 01 m1, b; 10 a, m2; 11 m1, m2
123     reg [7:0] old_state1_hz;
124     reg old_pass_hz, old_fetch_hz, old_pcincr_hz;
125     reg set_delay;
126
127     reg [6:0] opcode;
128     reg [3:0] cond;
129 //    reg [1:0] imm;
130     reg [4:0] reg_a_addr, reg_b_addr;
131     reg [4:0] reg_c_addr, reg_d_addr;
132     reg stage1,stage2, stage3, stage4;
133
134     always @(posedge clk or posedge rst) begin
135         #1;
136         if(rst) begin
137             e_a <= 31'b0; e_b <= 31'b0;
138             e_alu_op <= 8'b0; //NOP
139             e_cond <= 4'b0;
140             e_write_flags = 4'b0;
141             e_swp <= 1'b0; e_is_cond <= 1'b0;
142
143             m_a1 <= 31'b0; m_a2 <= 31'b0;
144             m_r1_op <= 4'b0; m_r2_op <= 4'b0; //clean NOP
145
146             r_a1 <= 5'b0; r_a2 <= 5'b0;
147             r_op <= 4'b0; //NOP;
148             d_pass <= 1'b0; d_pcincr <= 1'b1;
149             r_r1_addr <= 5'b0; r_r2_addr <= 5'b0;
150             r_read <= 2'b0;
151             state1 <= 0; fetch <= 1; reg_fetch <= 0;
152             old_pass_imm <= 0; old_fetch_imm <= 0; old_pcincr_imm <= 0;
                    old_state1_imm <= 0;
153             old_pass_hz <= 0; old_fetch_hz <= 0; old_pcincr_hz <= 0;
                    old_state1_hz <= 0;
154             set_delay <= 0;
155             opcode <= 0;
```

```
156             delay_counter <= 4'b0;
157             imm_action <= 3'b0;
158             r_to_mem <= 0;
159             stage1 <= 0;stage2 <= 0;stage3 <= 0; stage4 <= 0;
160         end
161         else begin
162             /*case(state1)
163                 0: begin opcode = word[31:25];
164                         cond <= word[24:21];
165                         reg_a_addr <= word[20:16]; reg_b_addr <= word[15:11];
                                reg_c_addr <= word[10:5]; reg_d_addr <= word[4:0];
166                         imm <= word[4:3];
167 
168                     state1 <= 1;
169                     state2 <= opcode;
170                 end
171                 1:
172             endcase
173             //state 1 is for decoding
174             //state 2 is for opcode setup
175             //state 3 is for additional operations
176             case(state2)
177                 0: begin //nop
178                     e_alu_op <= 0; e_cond <= 0; e_write_flags <= 0; e_is_cond <=
                            0;
179                     m_r1_op <= 4'b0; m_r2_op <= 4'b0;
180                     r_op <= 0; r_read <= 0; d_pass <= 1 d_pcincr <= 1;
181                     state1 <= 0;
182                 end
183                 1: begin //or
184                     e_alu_op <= 8'h0D; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1;
185                     m_r1_op <= 4'b0; m_r2_op <= 4'b0;
186                     r_op <= 2; //if respective imm r_read = 0, d_pass = 0,
                            d_pcincr = 1;
187             */
188             if(fetch) begin
189                 opcode = word[31:25];
190                 cond <= word[24:21];
191                 reg_a_addr <= word[20:16]; reg_b_addr <= word[15:11]; reg_c_addr
                        <= word[10:6]; reg_d_addr <= word[5:1];
192                 imm_action <= {1'b0, word[5:4]};
193                 state1 <= opcode;
194                 #1;
195                 d_pcincr <= 1;
196                 d_pass <= 1;
197                 reg_fetch <= 1;
198             end
199             stage1 <= 1;
```

```
200         end
201     end
202
203     always @(posedge stage1) begin
204             #0.1;
205             case(state1)
206                 //logic
207                 0: begin //nop
208                     e_alu_op <= 0; e_cond <= 0; e_write_flags <= 0; e_is_cond <=
                            0; //alu nop, not conditional, no flags
209                     m_r1_op <= 4'b0; m_r2_op <= 4'b0; //memory clean nop
210                     r_op <= 0; //register write nop
211                     r_read <= 0; //register read none
212                     r_to_mem <= 0;//register read to a,b
213                     imm_action <= 3'b000; //no imm in this insn
214                 end
215                 1: begin //or
216                     e_alu_op <= 8'h0D; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu or, conditional, all flags
217                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
218                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
219                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
220                     r_to_mem <= 0;//register read to a,b
221                 end
222                 2: begin //nor
223                     e_alu_op <= 8'h10; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu nor, conditional, all flags
224                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
225                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
226                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
227                     r_to_mem <= 0;//register read to a,b
228                 end
229                 3: begin //and
230                     e_alu_op <= 8'h0C; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu and, conditional, all flags
231                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
232                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
233                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
234                     r_to_mem <= 0;//register read to a,b
235                 end
236                 4: begin //nand
237                     e_alu_op <= 8'h0F; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu nand, conditional, all flags
238                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
239                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
240                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
```

```
                3; //register read both
241             r_to_mem <= 0;//register read to a,b
242         end
243         5: begin //inv
244             e_alu_op <= 8'h0B; e_cond <= cond; e_write_flags <= 4'hF;
                    e_is_cond <= 1; //alu not, conditional, all flags
245             m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
246             r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
247             r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
248             r_to_mem <= 0;//register read to a,b
249             imm_action[0] <= 0; //no imm for b in this insn
250         end
251         6: begin //xor
252             e_alu_op <= 8'h0E; e_cond <= cond; e_write_flags <= 4'hF;
                    e_is_cond <= 1; //alu xor, conditional, all flags
253             m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
254             r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
255             r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                    3; //register read both
256             r_to_mem <= 0;//register read to a,b
257         end
258         7: begin //xnor
259             e_alu_op <= 8'h11; e_cond <= cond; e_write_flags <= 4'hF;
                    e_is_cond <= 1; //alu xnor, conditional, all flags
260             m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
261             r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
262             r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                    3; //register read both
263             r_to_mem <= 0;//register read to a,b
264         end
265         //shifts
266         8: begin //lsl
267             e_alu_op <= 8'h06; e_cond <= cond; e_write_flags <= 4'hF;
                    e_is_cond <= 1; //alu shl, conditional, all flags
268             m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
269             r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
270             r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                    3; //register read both
271             r_to_mem <= 0;//register read to a,b
272         end
273         9: begin //lsr
274             e_alu_op <= 8'h05; e_cond <= cond; e_write_flags <= 4'hF;
                    e_is_cond <= 1; //alu shr, conditional, all flags
275             m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
276             r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
277             r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                    3; //register read both
278             r_to_mem <= 0;//register read to a,b
279         end
```

```
280                 10: begin //asr
281                     e_alu_op <= 8'h07; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu sar, conditional, all flags
282                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
283                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
284                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
285                     r_to_mem <= 0;//register read to a,b
286                 end
287                 11: begin //asl
288                     e_alu_op <= 8'h08; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu sal, conditional, all flags
289                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
290                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
291                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
292                     r_to_mem <= 0;//register read to a,b
293                 end
294                 12: begin //csr
295                     e_alu_op <= 8'h09; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu ror, conditional, all flags
296                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
297                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
298                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
299                     r_to_mem <= 0;//register read to a,b
300                 end
301                 13: begin //csl
302                     e_alu_op <= 8'h0A; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu rol, conditional, all flags
303                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
304                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
305                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
306                     r_to_mem <= 0;//register read to a,b
307                 end
308                 //arithmetics
309                 14: begin //add
310                     e_alu_op <= 8'h01; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu add, conditional, all flags
311                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
312                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
313                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
314                     r_to_mem <= 0;//register read to a,b
315                 end
316                 15: begin //sub
317                     e_alu_op <= 8'h02; e_cond <= cond; e_write_flags <= 4'hF;
                            e_is_cond <= 1; //alu sub, conditional, all flags
```

```
318                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
319                 r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
320                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
321                 r_to_mem <= 0;//register read to a,b
322             end
323             16: begin //mull
324                 e_alu_op <= 8'h04; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu mul, conditional, all flags
325                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
326                 r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
327                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
328                 r_to_mem <= 0;//register read to a,b
329             end
330             17: begin //mulh
331                 e_alu_op <= 8'h04; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu mul, conditional, all flags
332                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
333                 r_op <= 4; r_a1 <= reg_c_addr; // register write d to a1
334                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
335                 r_to_mem <= 0;//register read to a,b
336             end
337             18: begin //mul
338                 e_alu_op <= 8'h04; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu mul, conditional, all flags
339                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
340                 r_op <= 7; r_a1 <= reg_c_addr; r_a2 <= reg_d_addr; //
                        register write c,d to a1,a2
341                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
342                 r_to_mem <= 0;//register read to a,b
343                 imm_action <= 3'b000; //no imm in this insn
344             end
345             19: begin //csg
346                 e_alu_op <= 8'h03; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu cpl, conditional, all flags
347                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
348                 r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
349                 r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
350                 r_to_mem <= 0;//register read to a,b
351                 imm_action[0] <= 0; //no imm for b in this insn
352             end
353             20: begin //inc
354                 e_alu_op <= 8'h01; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu add, conditional, all flags
355                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
356                 r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
```

```
357                 r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
358                 r_to_mem <= 0;//register read to a,b
359                 e_b <= 1; //force b operand to be 1
360                 imm_action[0] <= 0; //no imm for b in this insn
361             end
362             21: begin //dec
363                 e_alu_op <= 8'h02; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu sub, conditional, all flags
364                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
365                 r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
366                 r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
367                 r_to_mem <= 0;//register read to a,b
368                 e_b <= 1; //force b operand to be 1
369                 imm_action[0] <= 0; //no imm for b in this insn
370             end
371             22: begin //cmp
372                 e_alu_op <= 8'h02; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu sub, conditional, all flags
373                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
374                 r_op <= 0; // register write nop
375                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
376                 r_to_mem <= 0;//register read to a,b
377             end
378             23: begin //cmn
379                 e_alu_op <= 8'h01; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu add, conditional, all flags
380                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
381                 r_op <= 0; // register write nop
382                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
383                 r_to_mem <= 0;//register read to a,b
384             end
385             24: begin //tst
386                 e_alu_op <= 8'h0C; e_cond <= cond; e_write_flags <= 4'hF;
                        e_is_cond <= 1; //alu and, conditional, all flags
387                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
388                 r_op <= 0; // register write nop
389                 r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                        3; //register read both
390                 r_to_mem <= 0;//register read to a,b
391             end
392             //branches
393             25: begin //br
394                 e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                        e_is_cond <= 1; //alu nop, conditional, no flags
395                 m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
396                 r_op <= 1; r_a1 <= 31; // register write to pc
397                 r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
```

```
398                     r_to_mem <= 0;//register read to a,b
399                     imm_action[0] <= 0; //no imm for b in this insn
400                     //delay!
401                     //set_delay <= 1;
402                     fetch <= 0; d_pcincr <= 0;
403                     state1 <= 130;
404                     delay_counter <= 3;
405                 end
406                 26: begin //rbr
407                     e_alu_op <= 8'h01; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu add, conditional, no flags
408                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
409                     r_op <= 1; r_a1 <= 31; // register write to pc
410                     r_r2_addr <= reg_a_addr; r_r1_addr <= 31; r_read <= 3; //
                            register read both, first - pc
411                     r_to_mem <= 0;//register read to a,b
412                     imm_action[0] <= 0; //no imm for b in this insn
413                     //delay!
414                     //set_delay <= 1;
415                     fetch <= 0; d_pcincr <= 0;
416                     state1 <= 130;
417                     delay_counter <= 3;
418                 end
419                 27: begin //brl
420                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
421                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
422                     r_op <= 7; r_a1 <= 31; r_a2 <= 29; // register write a,b to
                            pc,lr
423                     r_r1_addr <= reg_a_addr; r_r2_addr <= 31; r_read <= 3; //
                            register read both, second - pc
424                     r_to_mem <= 0;//register read to a,b
425                     imm_action[0] <= 0; //no imm for b in this insn
426                     //delay!
427                     //set_delay <= 1;
428                     fetch <= 0; d_pcincr <= 0;
429                     state1 <= 130;
430                     delay_counter <= 3;
431                 end
432                 /*27: begin //rbl, can't implement now (need hook in register_wb)
433                     e_alu_op <= 8'h01; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu add, conditional, no flags
434                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
435                     r_op <= 1; r_a1 <= 31 // register write to pc
436                     r_r2_addr <= reg_a_addr; r_r1_addr <= 31; r_read <= 2; //
                            register read both, first - pc
437                     imm_action[0] <= 0; //no imm for b in this insn
438                     //delay!
439                 end*/
```

```
440                 28: begin //ret
441                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
442                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
443                     r_op <= 1; r_a1 <= 31; // register write to pc
444                     r_r1_addr <= 29; r_read <= 1; //register read first - lr
445                     r_to_mem <= 0;//register read to a,b
446                     imm_action <= 3'b000; //no imm in this insn
447                     //delay!
448                     //set_delay <= 1;
449                     fetch <= 0; d_pcincr <= 0;
450                     state1 <= 130;
451                     delay_counter <= 3;
452                 end
453                 29: begin //ldr
454                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
455                     m_r1_op <= 2; m_r2_op <= 1; //memory read c from a1
456                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
457                     r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
458                     r_to_mem <= 2'b01;//register read to m1, b
459                     imm_action[0] <= 0; //no imm for b in this insn
460                     imm_action[2] <= 1; //imm goes into m
461                 end
462                 30: begin //str
463                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
464                     m_r1_op <= 1; m_r2_op <= 5; //memory write d to a1
465                     r_op <= 0; // register write nop
466                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
467                     r_to_mem <= 2'b01;//register read to m1, b
468                     imm_action[0] <= 0; //no imm for b in this insn
469                     imm_action[2] <= 1; //imm goes into m
470                 end
471                 //ldrc
472                 //strc
473                 //needs more elaborate management of operands (3, but have only
                        2, perhaps use imm ?
474
475                 //push
476                 //pop
477                 //one of this needs advanced management in memory_op stage
478                 //or make as in x86 - pop only decrements, not returning result
479
480                 31: begin //in
481                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
482                     m_r1_op <= 4'b1000; m_r2_op <= 4'b1; //sys read c from a1
```

```
483                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
484                     r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
485                     r_to_mem <= 2'b01;//register read to m1, b
486                     imm_action[0] <= 0; //no imm for b in this insn
487                     imm_action[2] <= 1; //imm goes into m
488                 end
489                 32: begin //out
490                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
491                     m_r1_op <= 4'b1; m_r2_op <= 4'b1011; //sys write d to a1
492                     r_op <= 0; // register write nop
493                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
494                     r_to_mem <= 2'b01;//register read to m1, b
495                     imm_action[0] <= 0; //no imm for b in this insn
496                     imm_action[2] <= 1; //imm goes into m
497                 end
498
499                 //ini
500                 //outi
501                 //needs more elaborate management of operands (3, but have only
                        2, perhaps use imm ?
502
503                 33: begin //movs
504                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
505                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
506                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
507                     r_r1_addr <= reg_a_addr; r_read <= 1; //register read first
508                     r_to_mem <= 0;//register read to a,b
509                     imm_action[0] <= 0; //no imm for b in this insn
510                 end
511                 34: begin //mov
512                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, all flags
513                     m_r1_op <= 4'b1; m_r2_op <= 4'b1; //memory passthrough nop
514                     r_op <= 7; r_a1 <= reg_c_addr; r_a2 <= reg_d_addr; //
                            register write c,d to a1,a2
515                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
                            3; //register read both
516                     r_to_mem <= 0;//register read to a,b
517                     imm_action <= 3'b000; //no imm in this insn
518                 end
519                 /*28: begin //ldr
520                     e_alu_op <= 8'h00; e_cond <= cond; e_write_flags <= 4'h0;
                            e_is_cond <= 1; //alu nop, conditional, no flags
521                     m_r1_op <= 4'b0011; m_r2_op <= 4'b1; //memory read c from a2
522                     r_op <= 1; r_a1 <= reg_c_addr; // register write c to a1
523                     r_r1_addr <= reg_a_addr; r_r2_addr <= reg_b_addr; r_read <=
```

```
                    3; //register read both
524                 r_to_mem <= 2'b10;//register read to a,m2
525             end*/
526
527             128: begin //get first imm
528                 if(imm_action == 3'b001) e_b <= word;
529                 else if(imm_action == 3'b010 || imm_action == 3'b011) e_a <=
                        word;
530                 else if(imm_action == 3'b110 || imm_action == 3'b111) m_a1 <=
                         word;
531                 else if(imm_action == 3'b101) m_a2 <= word;
532             end
533             129: begin //get second imm
534                 if(imm_action == 3'b011) e_b <= word;
535                 else if(imm_action == 3'b111) m_a2 <= word;
536             end
537             130: begin //delay
538                 fetch <= 0; d_pass <= 0; d_pcincr <= 0;
539                 if(delay_counter > 0) delay_counter<=delay_counter-1;
540                 #0;
541                 if(delay_counter == 0) begin
542                     fetch <= 1; /*d_pass <= 1;*/ d_pcincr <= 1;
543                     state1 <= 0;
544                 end
545             end
546             131: begin //hazard hold
547                 #0;
548                 if(!hazard) begin
549                    d_pcincr <= old_pcincr_hz;
550                    d_pass <= old_pass_hz;
551                    reg_fetch <= 1;
552                    fetch <= old_fetch_hz;
553                    state1 <= old_state1_hz;
554                 end
555             end
556             //132: begin //branch pipeline purge
557             default: begin
558                 fetch <= 1;
559                 state1 <= 0;
560             end
561         endcase
562         #0;
563         if(set_delay) begin
564             fetch <= 0; d_pcincr <= 0;
565             state1 <= 130;
566             set_delay <= 0;
567         end
568         stage1 <= 0;
569         stage2 <= 1;
```

```
570             /*@(posedge stage2) begin
571             if(imm_action != 3'b100 && imm_action != 3'b000) begin //imm fetch
                    procedure
572                 if(state1 != 128 && state1 != 129) begin //just got insn
573                     if(imm_action[1]) begin //imm for r1
574                         r_read[0] <= 0; //don't read r1
575                     end
576                     if(imm_action[0]) begin //imm for r2
577                         r_read[1] <= 0;  //don't read r2
578                     end
579                     old_state1_imm <= state1; //save state
580                     old_pass_imm <= d_pass;
581                     old_fetch_imm <= fetch;
582                     old_pcincr_imm <= d_pcincr;
583                     d_pass <= 0; //don't issue insn
584                     fetch <= 0; //don't decode insn
585                     reg_fetch <= 0; //don't fetch regs
586                     d_pcincr <= 1; //increment pc
587                     state1 <= 128; //fetch first imm
588                 end
589                 else if(state1 == 128) begin //first imm fetched
590                     if(imm_action == 3'b011 || imm_action == 3'b111) begin //need
                            to fetch second imm
591                         d_pass <= 0; //don't issue insn
592                         fetch <= 0; //don't decode insn
593                         reg_fetch <= 0; //don't fetch regs
594                         d_pcincr <= 1; //increment pc
595                         state1 <= 129; //fetch second imm
596                     end
597                     else begin //don't need to fetch second imm
598                         state1 <= old_state1_imm; //restore state
599                         d_pass <= old_pass_imm; //restore issue
600                         fetch <= old_fetch_imm; //restore fetch
601                         d_pcincr <= old_pcincr_imm; //restore incr pc
602                         reg_fetch <= 1; //fetch regs
603                         imm_action <= 3'b000; //don't fetch imm
604                     end
605                 end
606                 else if(state1 == 129) begin //second imm fetched
607                         state1 <= old_state1_imm; //restore state
608                         d_pass <= old_pass_imm; //restore issue
609                         fetch <= old_fetch_imm; //restore fetch
610                         d_pcincr <= old_pcincr_imm; //restore incr pc
611                         reg_fetch <= 1; //fetch regs
612                         imm_action <= 3'b000; //don't fetch imm
613                 end
614             end
615             #0;
616             if(hazard && reg_fetch) begin //hazard op
```

```
617                 old_pcincr_hz <= d_pcincr;
618                 old_pass_hz <= d_pass;
619                 old_fetch_hz <= fetch;
620                 old_state1_hz <= state1;
621                 d_pcincr <= 0;
622                 d_pass <= 0;
623                 fetch <= 0;
624                 reg_fetch <= 0;
625                 state1 <= 131;
626             end
627             #0;
628             if(reg_fetch) begin //reg fetch procedure
629                 if(r_read[0]) begin
630                     if(r_to_mem[0]) m_a1 <= r1;
631                     else e_a <= r1;
632                 end
633                 if(r_read[1]) begin
634                     if(r_to_mem[1]) m_a2 <= r2;
635                     else e_b <= r2;
636                 end
637                 reg_fetch <= 0;
638             end
639             stage2 <= 0;
640             end*/
641     end
642
643     always @(posedge stage2) begin
644         #0;
645             if(imm_action != 3'b100 && imm_action != 3'b000) begin //imm fetch
                    procedure
646                 if(state1 != 128 && state1 != 129) begin //just got insn
647                     if(imm_action[1]) begin //imm for r1
648                         r_read[0] <= 0; //don't read r1
649                     end
650                     if(imm_action[0]) begin //imm for r2
651                         r_read[1] <= 0;  //don't read r2
652                     end
653                     old_state1_imm <= state1; //save state
654                     old_pass_imm <= d_pass;
655                     old_fetch_imm <= fetch;
656                     old_pcincr_imm <= d_pcincr;
657                     d_pass <= 0; //don't issue insn
658                     fetch <= 0; //don't decode insn
659                     reg_fetch <= 0; //don't fetch regs
660                     d_pcincr <= 1; //increment pc
661                     state1 <= 128; //fetch first imm
662                 end
663                 else if(state1 == 128) begin //first imm fetched
664                     if(imm_action == 3'b011 || imm_action == 3'b111) begin //need
```

```
                        to fetch second imm
665                     d_pass <= 0; //don't issue insn
666                     fetch <= 0; //don't decode insn
667                     reg_fetch <= 0; //don't fetch regs
668                     d_pcincr <= 1; //increment pc
669                     state1 <= 129; //fetch second imm
670                 end
671                 else begin //don't need to fetch second imm
672                     state1 <= old_state1_imm; //restore state
673                     d_pass <= old_pass_imm; //restore issue
674                     fetch <= old_fetch_imm; //restore fetch
675                     d_pcincr <= old_pcincr_imm; //restore incr pc
676                     reg_fetch <= 1; //fetch regs
677                     imm_action <= 3'b000; //don't fetch imm
678                 end
679             end
680             else if(state1 == 129) begin //second imm fetched
681                     state1 <= old_state1_imm; //restore state
682                     d_pass <= old_pass_imm; //restore issue
683                     fetch <= old_fetch_imm; //restore fetch
684                     d_pcincr <= old_pcincr_imm; //restore incr pc
685                     reg_fetch <= 1; //fetch regs
686                     imm_action <= 3'b000; //don't fetch imm
687             end
688         end
689         stage2 <= 0;
690         stage3 <= 1;
691     end
692
693     always @(posedge stage3) begin
694         #0;
695             if(hazard && reg_fetch) begin //hazard op
696                 old_pcincr_hz <= d_pcincr;
697                 old_pass_hz <= d_pass;
698                 old_fetch_hz <= fetch;
699                 old_state1_hz <= state1;
700                 d_pcincr <= 0;
701                 d_pass <= 0;
702                 fetch <= 0;
703                 reg_fetch <= 0;
704                 state1 <= 131;
705             end
706             #0;
707             if(reg_fetch) begin //reg fetch procedure
708                if(r_read[0]) begin
709                    if(r_to_mem[0]) m_a1 <= r1;
710                    else e_a <= r1;
711                end
712                if(r_read[1]) begin
```

```
713                     if(r_to_mem[1]) m_a2 <= r2;
714                     else e_b <= r2;
715                 end
716                 reg_fetch <= 0;
717             end
718             stage3 <= 0;
719     end
720
721 endmodule
```

11.1.7 memory_op.v

```
1  `timescale 1 ns / 100 ps
2
3  module memory_op_stage_passthrough(q_a1, q_a2, q_op, q_proceed, a1, a2, op,
       proceed, clk, rst);
4      input [4:0] a1, a2; //(reg_wb)
5      input [3:0] op; //(reg_wb)
6      input proceed;
7
8      input clk, rst;
9
10     output reg [4:0] q_a1, q_a2; //(reg_wb)
11     output reg [3:0] q_op; //(reg_wb)
12     output reg q_proceed;
13
14     always @(posedge clk or posedge rst) begin
15         if(rst) begin
16             q_a1 <= 5'b0; q_a2 <= 5'b0;
17             q_op <= 4'b0;
18             q_proceed <= 1'b0;
19         end
20         else begin
21             q_a1 <= a1; q_a2 <= a2;
22             q_op <= op;
23             q_proceed <= proceed;
24         end
25     end
26 endmodule
27
28 module memory_op( m1, m2, ram_w_addr, ram_r_addr, ram_w, ram_r, ram_w_line,
       sys_w_addr, sys_r_addr, sys_w, sys_r, sys_w_line, r1, r2, a1, a2, r1_op,
       r2_op, ram_r_line, sys_r_line, proceed, clk, rst);
29     input [31:0] r1, r2; //inputs
30     input [31:0] a1, a2; //memory addresses
31
32     input [3:0] r1_op, r2_op; //operation codes
33
34     input [31:0] ram_r_line, sys_r_line; // read lanes
```

```
35
36      input proceed; //conditional code test result
37
38      input clk, rst;
39
40      output wire [31:0] m1, m2; //outputs
41
42      output reg [31:0] ram_w_addr, sys_w_addr; //write addresses
43      output reg [31:0] ram_r_addr, sys_r_addr; //read addresses
44
45      output reg [31:0] ram_w_line, sys_w_line; //write lanes
46
47      output reg ram_w, sys_w, ram_r, sys_r; //read/write signals
48
49      wire [3:0] r1_op_inner, r2_op_inner;
50
51      assign r1_op_inner = proceed ? r1_op : 4'b0;
52      assign r2_op_inner = proceed ? r2_op : 4'b0;
53
54      reg [31:0] r1_inner, r2_inner; //copies of inputs delayed by 1 clk, to cope
            with problem of mux delay, which don't allows inputs to descend pipeline
55      //procedural continuous assignments aren't stable in IcarusVerilog, so use
            explicit muxes
56      reg [2:0] m1_select, m2_select;
57      assign m1 = (m1_select == 0 ? 32'b0 : (m1_select == 1 ? r1_inner : (m1_select
             == 2 ? r2_inner : (m1_select == 3 ? ram_r_line : (m1_select == 4 ?
            sys_r_line : 32'hAAAAAAAA)))));
58      assign m2 = (m2_select == 0 ? 32'b0 : (m2_select == 1 ? r1_inner : (m2_select
             == 2 ? r2_inner : (m2_select == 3 ? ram_r_line : (m2_select == 4 ?
            sys_r_line : 32'hAAAAAAAA)))));
59
60      always @(posedge clk or posedge rst) begin
61          if(rst) begin
62              ram_w_addr <= 32'b0; ram_r_addr <= 32'b0;
63              sys_w_addr <= 32'b0; sys_r_addr <= 32'b0;
64              ram_w_line <= 32'b0; sys_w_line <= 32'b0;
65              ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
66              m1_select <= 0; m2_select <= 0;
67              r1_inner <= 32'b0; r2_inner <= 32'b0;
68          end
69          else begin
70              //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
71              //#0;
72              ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
73              case(r1_op_inner)
74                  0: begin //clean NOP
75                      m1_select <= 0; //force m1 = 32'b0;
76                      //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
77                      end
```

```
78                  1: begin //passthrough NOP
79                      m1_select <= 1; //force m1 = r1;
80                      //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
81                      end
82                  2: begin //load from memory address a1
83                      m1_select <= 3;  //force m1 = ram_r_line;
84                      ram_r_addr <= a1;
85                      ram_r <= 1'b1;
86                      //ram_w <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
87                      end
88                  3: begin //load from memory address a2
89                      m1_select <= 3;  //force m1 = ram_r_line;
90                      ram_r_addr <= a2;
91                      ram_r <= 1'b1;
92                      //ram_w <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
93                      end
94                  4: begin //load from memory address r2
95                      m1_select <= 3;  //force m1 = ram_r_line;
96                      ram_r_addr <= r2;
97                      ram_r <= 1'b1;
98                      //ram_w <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
99                      end
100                 5: begin //write to memory address a1
101                     m1_select <= 1; //force m1 = r1;
102                     ram_w_line <= r1;
103                     ram_w_addr <= a1;
104                     ram_w <= 1'b1;
105                     //ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
106                     end
107                 6: begin //write to memory address a2
108                     m1_select <= 1; //force m1 = r1;
109                     ram_w_line <= r1;
110                     ram_w_addr <= a2;
111                     ram_w <= 1'b1;
112                     //ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
113                     end
114                 7: begin //write to memory address r2
115                     m1_select <= 1; //force m1 = r1;
116                     ram_w_line <= r1;
117                     ram_w_addr <= r2;
118                     ram_w <= 1'b1;
119                     //ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
120                     end
121                 8: begin //load from sys address a1
122                     m1_select <= 4; //force m1 = sys_r_line;
123                     sys_r_addr <= a1;
124                     sys_r <= 1'b1;
125                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_w <= 1'b0;
126                     end
```

```
127                 9: begin //load from sys address a2
128                     m1_select <= 4; //force m1 = sys_r_line;
129                     sys_r_addr <= a2;
130                     sys_r <= 1'b1;
131                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_w <= 1'b0;
132                     end
133                 10: begin //load from sys address r2
134                     m1_select <= 4; //force m1 = sys_r_line;
135                     sys_r_addr <= r2;
136                     sys_r <= 1'b1;
137                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_w <= 1'b0;
138                     end
139                 11: begin //write to sys address a1
140                     m1_select <= 1; //force m1 = r1;
141                     sys_w_line <= r1;
142                     sys_w_addr <= a1;
143                     sys_w <= 1'b1;
144                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0;
145                     end
146                 12: begin //write to sys address a2
147                     m1_select <= 1; //force m1 = r1;
148                     sys_w_line <= r1;
149                     sys_w_addr <= a2;
150                     sys_w <= 1'b1;
151                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0;
152                     end
153                 13: begin //write to sys address r2
154                     m1_select <= 1; //force m1 = r1;
155                     sys_w_line <= r1;
156                     sys_w_addr <= r2;
157                     sys_w <= 1'b1;
158                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0;
159                     end
160                 14: begin //swap regs
161                     m1_select <= 2; //force m1 = r2;
162                    // ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
163                     end
164             endcase
165
166             case(r2_op_inner)
167                 0: begin //clean NOP
168                     m2_select <= 0; //force m2 = 32'b0;
169                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
170                     end
171                 1: begin //passthrough NOP
172                     m2_select <= 2; //force m2 = r2;
173                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
174                     end
175                 2: begin //load from memory address a1
```

```
176                     m2_select <= 3; //force m2 = ram_r_line;
177                     ram_r_addr <= a1;
178                     ram_r <= 1'b1;
179                     //ram_w <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
180                     end
181                 3: begin //load from memory address a2
182                     m2_select <= 3; //force m2 = ram_r_line;
183                     ram_r_addr <= a2;
184                     ram_r <= 1'b1;
185                     //ram_w <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
186                     end
187                 4: begin //load from memory address r1
188                     m2_select <= 3; //force m2 = ram_r_line;
189                     ram_r_addr <= r1;
190                     ram_r <= 1'b1;
191                     //ram_w <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
192                     end
193                 5: begin //write to memory address a1
194                     m2_select <= 2; //force m2 = r2;
195                     ram_w_line <= r2;
196                     ram_w_addr <= a1;
197                     ram_w <= 1'b1;
198                     //ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
199                     end
200                 6: begin //write to memory address a2
201                     m2_select <= 2; //force m2 = r2;
202                     ram_w_line <= r2;
203                     ram_w_addr <= a2;
204                     ram_w <= 1'b1;
205                     //ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
206                     end
207                 7: begin //write to memory address r1
208                     m2_select <= 2; //force m2 = r2;
209                     ram_w_line <= r2;
210                     ram_w_addr <= r1;
211                     ram_w <= 1'b1;
212                     //ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
213                     end
214                 8: begin //load from sys address a1
215                     m2_select <= 4; //force m2 = sys_r_line;
216                     sys_r_addr <= a1;
217                     sys_r <= 1'b1;
218                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_w <= 1'b0;
219                     end
220                 9: begin //load from sys address a2
221                     m2_select <= 4; //force m2 = sys_r_line;
222                     sys_r_addr <= a2;
223                     sys_r <= 1'b1;
224                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_w <= 1'b0;
```

```
225                     end
226                 10: begin //load from sys address r1
227                     m2_select <= 4; //force m2 = sys_r_line;
228                     sys_r_addr <= r1;
229                     sys_r <= 1'b1;
230                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_w <= 1'b0;
231                     end
232                 11: begin //write to sys address a1
233                     m2_select <= 2; //force m2 = r2;
234                     sys_w_line <= r2;
235                     sys_w_addr <= a1;
236                     sys_w <= 1'b1;
237                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0;
238                     end
239                 12: begin //write to sys address a2
240                     m2_select <= 2; //force m2 = r2;
241                     sys_w_line <= r2;
242                     sys_w_addr <= a2;
243                     sys_w <= 1'b1;
244                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0;
245                     end
246                 13: begin //write to sys address r1
247                     m2_select <= 2; //force m2 = r2;
248                     sys_w_line <= r2;
249                     sys_w_addr <= r1;
250                     sys_w <= 1'b1;
251                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0;
252                     end
253                 14: begin //swap regs
254                     m2_select <= 1; //force m2 = r1;
255                     //ram_w <= 1'b0; ram_r <= 1'b0; sys_r <= 1'b0; sys_w <= 1'b0;
256                     end
257             endcase
258             r1_inner <= r1;
259             r2_inner <= r2;
260         end
261     end
262 endmodule
```

11.1.8 pipeline_interface.v

```
1  `timescale 1 ns / 100 ps
2
3  module pipeline_interface(
4      qe_a, qe_b, qe_alu_op, qe_is_cond, qe_cond, qe_write_flags, qe_swp, qm_a1,
           qm_a2, qm_r1_op, qm_r2_op, qr_a1, qr_a2, qr_op, qd_pcincr,
5      e_a, e_b, e_alu_op, e_is_cond, e_cond, e_write_flags, e_swp, m_a1, m_a2,
           m_r1_op, m_r2_op, r_a1, r_a2, r_op, d_pass, d_pcincr, clk, rst);
6      input [31:0] e_a, e_b;
```

```
7       input [7:0] e_alu_op;
8       input [3:0] e_cond;
9       input [3:0] e_write_flags;
10      input e_swp;
11      input e_is_cond;
12
13      input [31:0] m_a1, m_a2;
14      input [3:0] m_r1_op, m_r2_op;
15
16      input [4:0] r_a1, r_a2;
17      input [3:0] r_op;
18
19      input d_pass;
20      input d_pcincr;
21
22      input clk, rst;
23
24      output reg [31:0] qe_a, qe_b;
25      output reg [7:0] qe_alu_op;
26      output reg [3:0] qe_cond;
27      output reg [3:0] qe_write_flags;
28      output reg qe_swp;
29      output reg qe_is_cond;
30
31      output reg [31:0] qm_a1, qm_a2;
32      output reg [3:0] qm_r1_op, qm_r2_op;
33
34      output reg [4:0] qr_a1, qr_a2;
35      output reg [3:0] qr_op;
36
37      output reg qd_pcincr;
38
39      reg test;
40
41      initial begin
42          test <= 1'b0;
43      end
44      always @(posedge clk or posedge rst) begin
45          if(rst) begin
46              qe_a = 31'b0; qe_b = 31'b0;
47              qe_alu_op = 8'b0; //NOP
48              qe_cond = 4'b0;
49              qe_write_flags = 4'b0;
50              qe_swp = 1'b0; qe_is_cond = 1'b0;
51
52              qm_a1 = 31'b0; qm_a2 = 31'b0;
53              qm_r1_op = 4'b0; qm_r2_op = 4'b0; //clean NOP
54
55              qr_a1 = 5'b0; qr_a2 = 5'b0;
```

```
56              qr_op = 4'b0; //NOP;
57              test = ~test;
58              qd_pcincr = 1'b1;
59          end
60          else begin
61              `ifdef INTERFACE_STAGE_NO_DELAY
62                  #3;
63              `endif
64              if(!d_pass) begin // insert clean NOP
65                  qe_a = 31'b0; qe_b = 31'b0;
66                  qe_alu_op = 8'b0; //NOP
67                  qe_cond = 4'b0;
68                  qe_write_flags = 4'b0;
69                  qe_swp = 1'b0; qe_is_cond = 1'b0;
70
71                  qm_a1 = 31'b0; qm_a2 = 31'b0;
72                  qm_r1_op = 4'b0; qm_r2_op = 4'b0; //clean NOP
73
74                  qr_a1 = 5'b0; qr_a2 = 5'b0;
75                  qr_op = 4'b0; //NOP;
76                  test <= ~test;
77              end
78              else begin //pass args & signals down to the pipeline
79                  qe_a = e_a; qe_b = e_b;
80                  qe_alu_op = e_alu_op;
81                  qe_cond = e_cond;
82                  qe_write_flags = e_write_flags;
83                  qe_swp = e_swp; qe_is_cond = e_is_cond;
84
85                  qm_a1 = m_a1; qm_a2 = m_a2;
86                  qm_r1_op = m_r1_op; qm_r2_op = m_r2_op;
87
88                  qr_a1 = r_a1; qr_a2 = r_a2;
89                  qr_op = r_op;
90              end
91              qd_pcincr = d_pcincr;
92          end
93      end
94  endmodule
```

11.1.9 ram.v

```
1   `timescale 1 ns / 100 ps
2
3   module ram(r_addr, w_addr, r_line, w_line, read, write, wrdy, rrdy, exc, clk);
4       input [31:0] r_addr;
5       input [31:0] w_addr;
6       input [31:0] w_line;
7       input        read;
```

```
8       input          write;
9       input          clk;
10
11      output [31:0] r_line;
12      reg    [31:0] r_line;
13      output         exc;
14      reg            exc;
15      output         wrdy, rrdy;
16      reg            wrdy, rrdy;
17
18      //memory
19      parameter mem_size = 1024; //4kb, 4b/w
20
21      reg [31:0] mem [mem_size:0];
22
23      integer i;
24
25      /*initial begin
26          for(i = 0; i < mem_size; i=i+1) begin
27              mem[i] = 32'b0;
28          end
29          r_line = 32'b0;
30          exc = 1'b0;
31          wrdy = 1'b0;
32          rrdy = 1'b0;
33      end*/
34
35      always @(posedge clk) begin
36          if(wrdy) wrdy <= 1'b0;
37          if(rrdy) rrdy <= 1'b0;
38
39          if(read & !rrdy ) begin
40              if(r_addr >= mem_size) begin
41                  r_line <= 32'b0;
42                  exc <= 1'b1;
43              end
44              else begin
45                  r_line <= mem[r_addr];
46                  rrdy <= 1'b1;
47                  exc <= 1'b0;
48              end
49          end
50          else r_line <= 32'bz;
51
52          if(write && !wrdy) begin
53              if(w_addr >= mem_size) exc <= 1'b1;
54              else begin
55                  mem[w_addr] <= w_line;
56                  wrdy <= 1'b1;
```

```
57                  exc <= 1'b0;
58              end
59          end
60      end
61  endmodule
62
63  module emb_ram(r_addr, w_addr, r_line, w_line, read, write, exc, clk);
64      input [31:0] r_addr;
65      input [31:0] w_addr;
66      input [31:0] w_line;
67      input        read;
68      input        write;
69      input        clk;
70
71      output [31:0] r_line;
72      reg    [31:0] r_line;
73      output        exc;
74      reg           exc;
75
76      //memory
77      parameter mem_size = 1024; //4kb, 4b/w
78
79      reg [31:0] mem [mem_size:0];
80
81      integer i;
82
83      /*initial begin
84          for(i = 0; i < mem_size; i=i+1) begin
85              mem[i] = 32'b0;
86          end
87          r_line = 32'b0;
88          exc = 1'b0;
89      end*/
90
91       always @(posedge clk) begin //??????????
92          #1;
93          if(read) begin
94              if(r_addr >= mem_size) begin
95                  r_line <= 32'b0;
96                  exc <= 1'b1;
97              end
98              else begin
99                  r_line <= mem[r_addr];
100                 exc <= 1'b0;
101             end
102         end
103         else r_line <= 32'bz;
104
105         if(write) begin
```

```
106             if(w_addr >= mem_size) exc <= 1'b1;
107             else begin
108                 mem[w_addr] <= w_line;
109                 exc <= 1'b0;
110             end
111         end
112     end
113 endmodule
```

11.1.10 register_wb.v

```
1  `timescale 1 ns / 100 ps
2
3  module register_wb( write, wr1, wr2, wa1, wa2, r1, r2, a1, a2, op, proceed, clk,
       rst);
4      input [31:0] r1, r2;
5      input [4:0] a1, a2;
6
7      input [3:0] op;
8
9      input proceed;
10
11     input clk, rst;
12
13     output reg [31:0] wr1, wr2;
14     output reg [4:0] wa1, wa2;
15     output reg [1:0] write;
16
17     wire [3:0] inner_op;
18
19     assign inner_op = proceed ? op : 4'b0;
20
21     always @(posedge clk or posedge rst) begin
22         if(rst) begin
23             wr1 <= 32'b0; wr2 <= 32'b0;
24             wa1 <= 5'b0; wa2 <= 5'b0;
25             write <= 2'b00;
26         end
27         else begin
28             write <= 2'b00;
29             case(inner_op)
30                 0: write <= 2'b00; //NOP
31                 1: begin //write r1 to addr a1
32                     wr1 <= r1;
33                     wa1 <= a1;
34                     write <= 2'b01;
35                     end
36                 2: begin //write r1 to addr a2
37                     wr1 <= r1;
```

```
38                      wa1 <= a2;
39                      write <= 2'b01;
40                      end
41                  3: begin //write r1 to addr r2
42                      wr1 <= r1;
43                      wa1 <= r2[4:0];
44                      write <= 2'b01;
45                      end
46                  4: begin //write r2 to addr a1
47                      wr1 <= r2;
48                      wa1 <= a1;
49                      write <= 2'b01;
50                      end
51                  5: begin //write r2 to addr a2
52                      wr1 <= r2;
53                      wa1 <= a2;
54                      write <= 2'b01;
55                      end
56                  6: begin //write r2 to addr r1
57                      wr1 <= r2;
58                      wa1 <= r1[4:0];
59                      write <= 2'b01;
60                      end
61                  7: begin //write r2, r1 to a2, a1
62                      wr1 <= r1; wr2 <= r2;
63                      wa1 <= a1; wa2 <= a2;
64                      write <= 2'b11;
65                      end
66                  8: begin //write r1, r2 to a2, a1
67                      wr1 <= r1; wr2 <= r2;
68                      wa1 <= a2; wa2 <= a1;
69                      write <= 2'b11;
70                      end
71              endcase
72          end
73      end
74  endmodule
```

11.1.11 regs.v

```
1   `timescale 1 ns / 100 ps
2
3   module reg32_2x2_pc(rd0, rd1, ra0, ra1, wa0, wa1, wd0, wd1, read, write, clk, rst
        , lrout, spout, stout, pcout, stin, stwr, pcincr);
4       parameter addrsize = 5;
5       parameter regsnum = 32;
6
7       input [addrsize-1:0] ra0, ra1;
8       input [addrsize-1:0] wa0, wa1;
```

```
9
10      input [31:0] wd0, wd1;
11
12      input [1:0] read, write;
13
14      input clk, rst;
15
16      output wire [31:0] rd0, rd1;
17
18      reg [31:0] regs [regsnum-1:0];
19
20      output wire [31:0] lrout, spout, stout, pcout;
21      input [31:0] stin;
22      input stwr, pcincr;
23
24      assign pcout = regs[31];
25      assign lrout = regs[29];
26      assign spout = regs[30];
27      assign stout = regs[28];
28
29      assign rd0 = regs[ra0];
30      assign rd1 = regs[ra1];
31
32      always @(posedge clk or posedge rst) begin
33          #1;
34          if(rst) begin
35              /*rd0 <= 0;
36              rd1 <= 0;*/
37              regs[0] <= 32'b0;
38              regs[28] <= 32'b0;
39              regs[29] <= 32'b0;
40              regs[30] <= 32'b0;
41              regs[31] <= 32'b0;
42          end
43          else begin
44          //if(read[0]) rd0 <= regs[ra0];
45          //if(read[1]) rd1 <= regs[ra1];
46
47              if(write[0]) regs[wa0] <= wd0;
48              if(write[1]) regs[wa1] <= wd1;
49
50              if(stwr) regs[28] <= stin;
51              if(pcincr) regs[31] <= regs[31] + 1;
52          end
53
54      end
55  endmodule
```

11.1.12 shift.v

```
1  `timescale 1 ns / 100 ps
2
3  /*module fr(a, q);
4      input  [2:0] a;
5      output [2:0] q;
6
7      assign q[0] = a[0];
8      assign q[2] = ((~a[0])&a[2])^(a[0]&a[1]);
9      assign q[1] = ((~a[0])&a[1])^(a[0]&a[2]);
10 endmodule
11
12 module fe(a, q);
13     input  [1:0] a;
14     output [1:0] q;
15
16     assign q[0] = a[0];
17     assign q[1] = a[0]^a[1];
18 endmodule
19
20 module rev_shift_4(I, O, S);
21     input  [3:0] I;
22     input  [1:0] S;
23     output [3:0] O;
24
25     wire wfe[7:0];
26
27     fe fe0({I[0], 1'b0}, wfe[1:0]);
28     fe fe1({I[1], 1'b0}, wfe[3:2]);
29     fe fe2({I[2], 1'b0}, wfe[5:4]);
30     fe fe1({I[3], 1'b0}, wfe[7:6]);
31
32     wire grb0[3:0];
33     wire sgrb[3:0];
34     wire wfr[3:0];
35
36     fr fr0({S[0],wfe[1:0]}, {sgrb[0], grb0[0], wfr[0]});
37     fr fr1({S[0],wfe[2:3]}, {sgrb[1], grb0[1], wfr[1]});
38     fr fr2({S[0],wfe[5:4]}, {sgrb[2], grb0[2], wfr[2]});
39     fr fr3({S[0],wfe[7:6]}, {sgrb[3], grb0[3], wfr[3]});
40
41     wire ssgrb[1:0];
42
43     fr fr4({S[1],wfr[1:0]},{ssgrb[0], O[1:0]});
44     fr fr5({S[1],wfr[3:2]},{ssgrb[1], O[3:2]});
45 endmodule*/
46
47 module right_shift_rot_32(y, a, b, rotate, sra, sla);
48     input [31:0] a;
49     input [4:0] b;
```

```
50
51      output[31:0] y;
52
53      input rotate, sra, sla;
54
55      wire sgnr = sra ? a[31] : 1'b0;
56
57  //stage 1, b[4] - 16-bit shift/rot
58      wire [31:0] st1;
59      wire [15:0] r1;
60      //rot section
61      assign r1 = rotate ? a[15:0] : (sgnr ? 16'hffff : 16'h0);
62      //shift section
63      assign st1[31:16] = b[4] ? r1 : a[31:16];
64      assign st1[15:0] = b[4] ? a[31:16] : a[15:0];
65
66  //stage 2, b[3] - 8-bit shift/rot
67      wire [31:0] st2;
68      wire [7:0] r2;
69      //rot section
70      assign r2 = rotate ? st1[7:0] : (sgnr ? 8'hff : 8'h0);
71      //shift section
72      assign st2[31:24] = b[3] ? r2 : st1[31:24];
73      assign st2[23:0] = b[3] ? st1[31:8] : st1[23:0];
74  //stage 3, b[2] - 4-bit shift/rot
75      wire [31:0] st3;
76      wire [3:0] r3;
77      //rot section
78      assign r3 = rotate ? st2[3:0] : (sgnr ? 4'hf : 4'h0);
79      //shift section
80      assign st3[31:28] = b[2] ? r3 : st2[31:28];
81      assign st3[27:0] = b[2] ? st2[31:4] : st2[27:0];
82  //stage 4, b[1] - 2-bit shift/rot
83      wire [31:0] st4;
84      wire [1:0] r4;
85      //rot section
86      assign r4 = rotate ? st3[1:0] : (sgnr ? 2'b11 : 2'b00);
87      //shift section
88      assign st4[31:30] = b[1] ? r4 : st3[31:30];
89      assign st4[29:0] = b[1] ? st3[31:2] : st3[29:0];
90  //stage 5, b[0] - 1-bit shift/rot
91      wire r5;
92      wire sgnl;
93      //rot section
94      assign r5 = rotate ? st4[0] : sgnr;
95      //shift section
96      assign y[31] = b[0] ? r5 : st4[31];
97      assign {y[30:1], sgnl} = b[0] ? st4[31:1] : st4[30:0];
98      assign y[0] = sla ? a[0] : sgnl;
```

```
99
100 endmodule
101
102 module right_rot_32(y, a, b);
103     input [31:0] a;
104     input [4:0] b;
105
106     output [31:0] y;
107
108 //stage 1, b[4] - 16-bit rot
109     wire [31:0] st1;
110
111     assign st1[31:16] = b[4] ? a[15:0] : a[31:16];
112     assign st1[15:0] = b[4] ? a[31:16] : a[15:0];
113 //stage 2, b[3] - 8-bit rot
114     wire [31:0] st2;
115
116     assign st2[31:24] = b[3] ? st1[7:0] : st1[31:24];
117     assign st2[23:0] = b[3] ? st1[31:8] : st1[23:0];
118 //stage 3, b[2] - 4-bit rot
119     wire [31:0] st3;
120
121     assign st3[31:28] = b[2] ? st2[3:0] : st2[31:28];
122     assign st3[27:0] = b[2] ? st2[31:4] : st2[27:0];
123 //stage 4, b[1] - 2-bit rot
124     wire [31:0] st4;
125
126     assign st4[31:30] = b[1] ? st3[1:0] : st3[31:30];
127     assign st4[29:0] = b[1] ? st3[31:2] : st3[29:0];
128 //stage 5, b[0] - 1-bit rot
129
130     assign y[31] = b[0] ? st4[0] : st4[31];
131     assign y[30:0] = b[0] ? st4[31:1] : st4[30:0];
132 endmodule
133
134 module drev_32(q, a, e);
135     input [31:0] a;
136
137     output [31:0] q;
138     input e;
139
140     genvar i;
141     generate for(i = 0; i < 32; i = i + 1) begin : drev_mixer
142         assign q[i] = e ? a[31-i] : a[i];
143     end
144     endgenerate
145 endmodule
146
147 module fmask_32(q, a);
```

```
148     input [4:0] a;
149     output [31:0] q;
150     reg [31:0] q;
151 
152     always @* begin
153         case(a)
154             5'h00: q = 32'b11111111111111111111111111111111;
155             5'h01: q = 32'b01111111111111111111111111111111;
156             5'h02: q = 32'b00111111111111111111111111111111;
157             5'h03: q = 32'b00011111111111111111111111111111;
158             5'h04: q = 32'b00001111111111111111111111111111;
159             5'h05: q = 32'b00000111111111111111111111111111;
160             5'h06: q = 32'b00000011111111111111111111111111;
161             5'h07: q = 32'b00000001111111111111111111111111;
162             5'h08: q = 32'b00000000111111111111111111111111;
163             5'h09: q = 32'b00000000011111111111111111111111;
164             5'h0A: q = 32'b00000000001111111111111111111111;
165             5'h0B: q = 32'b00000000000111111111111111111111;
166             5'h0C: q = 32'b00000000000011111111111111111111;
167             5'h0D: q = 32'b00000000000001111111111111111111;
168             5'h0E: q = 32'b00000000000000111111111111111111;
169             5'h0F: q = 32'b00000000000000011111111111111111;
170             5'h10: q = 32'b00000000000000001111111111111111;
171             5'h11: q = 32'b00000000000000000111111111111111;
172             5'h12: q = 32'b00000000000000000011111111111111;
173             5'h13: q = 32'b00000000000000000001111111111111;
174             5'h14: q = 32'b00000000000000000000111111111111;
175             5'h15: q = 32'b00000000000000000000011111111111;
176             5'h16: q = 32'b00000000000000000000001111111111;
177             5'h17: q = 32'b00000000000000000000000111111111;
178             5'h18: q = 32'b00000000000000000000000011111111;
179             5'h19: q = 32'b00000000000000000000000001111111;
180             5'h1A: q = 32'b00000000000000000000000000111111;
181             5'h1B: q = 32'b00000000000000000000000000011111;
182             5'h1C: q = 32'b00000000000000000000000000001111;
183             5'h1D: q = 32'b00000000000000000000000000000111;
184             5'h1E: q = 32'b00000000000000000000000000000011;
185             5'h1F: q = 32'b00000000000000000000000000000001;
186             default: q = 32'b00000000000000000000000000000000;
187         endcase
188     end
189 endmodule
190 
191 module ovf_32(q, a, f, sla);
192     input [31:0] f;
193     input [31:0] a;
194     input sla;
195 
196     output q;
```

```
197
198     wire [30:0] aexp = a[31] ? 31'h7FFFFFF : 31'h00000000;
199
200     wire w1 = |((aexp^a[30:0])&(~(f[31:1])));
201
202     assign q = sla&w1;
203 endmodule
204
205 module zmask_32(q, a, sla);
206     input [31:0] a;
207     input sla;
208
209     output [31:0] q;
210
211     assign q[0] = sla | a[31];
212
213     genvar i;
214     generate for(i = 1; i < 32; i = i + 1) begin : zmask_mixer
215         assign q[i] = sla ? a[32-i] : a[31-i];
216     end
217     endgenerate
218 endmodule
219
220 module tblock_32(q, a, sgn, p, sla, sra);
221     input [31:0] a;
222     input [31:0] p;
223     input sgn, sla, sra;
224
225     output [31:0] q;
226
227     wire [30:0] s = (sra&sgn) ? 31'h7FFFFFF : 31'h00000000;
228
229     assign q[0] = a[0]&(~sla) | sla&sgn;
230     assign q[31:1] = a[31:1]&p[31:1] | s&(~p[31:1]);
231 endmodule
232
233 module bshift_32(q, ov, z, a, b, rotate, left, arith);
234     input [31:0] a;
235     input[4:0] b;
236     input rotate, left, arith;
237
238     output [31:0] q;
239     output ov, z;
240
241     wire [31:0] am;
242     drev_32 dr0(am, a, left);
243
244     wire [31:0] ym;
245     right_rot_32 rr0(ym, am, b);
```

```
246
247     wire sra = (~rotate)&(~left)&arith;
248     wire sla = (~rotate)&(left)&arith;
249
250     wire[31:0] f;
251     fmask_32 f0(f, b);
252
253     wire [31:0] p;
254     assign p = rotate ? 32'hFFFFFFFF : f;
255
256     wire [31:0] t;
257     tblock_32 t0(t, ym, a[31], p, sla, sra);
258
259     drev_32 dr1(q, t, left);
260
261     wire [31:0] zm;
262     zmask_32 z0(zm, p, sla);
263
264     assign z = ~|(zm&am);
265
266     ovf_32 ov0(ov, a, f, sla);
267 endmodule
```

11.1.13 test_periph_assembly.v

```
1  `timescale 1 ns / 100 ps
2
3  `include "gpio_mux.v"
4  `include "gpio.v"
5
6  module test_periph_assembly(pins, sys_w_addr, sys_r_addr, sys_w_line, sys_r_line,
       sys_w, sys_r, rst, clk);
7      inout [127:0] pins; //our system will have 128 pins
8
9      //peripheral bus
10     input [31:0] sys_w_addr;
11     input [31:0] sys_r_addr;
12     input [31:0] sys_w_line;
13     output wire [31:0] sys_r_line;
14     input sys_w;
15     input sys_r;
16
17     //generic
18     input clk;
19     input rst;
20
21     /*devices registry
22      * 1. address
23      *  00000 - 00001 - not assigned (guard band) (0x00 - 0x01)
```

```
24          *  00010 - 00011 - gpio_mux pins 31:0 (0x02 - 0x03)
25          *  00100 - 00101 - gpio_mux pins 63:32 (0x04 - 0x05)
26          *  00110 - 00111 - gpio_mux pins 95:64 (0x06 - 0x07)
27          *  01000 - 01001 - gpio_mux pins 127:96 (0x08 - 0x09)
28          *  01010 - 01011 - gpio chip 1 (31:0) (0x0A - 0x0B)
29          *  01100 - 01101 - gpio chip 2 (63:32) (0x0C - 0x0D)
30          *  01110 - 01111 - gpio chip 3 (95:64) (0x0E - 0x0F)
31          *  10000 - 10001 - gpio chip 4 (127:96) (0x10 - 0x11)
32          * ------------------------------------------------
33          * 2. pins
34          *  all pins have gpio chip as function 0
35          * ------------------------------------------------
36          */
37
38      wire [31:0] g0_out, g1_out, g2_out, g3_out;
39      wire [31:0] g0_in, g1_in, g2_in, g3_in;
40      wire [31:0] g0_dir, g1_dir, g2_dir, g3_dir;
41      gpio chip0(g0_out, g0_in, g0_dir, 32'hA, sys_w_addr, sys_r_addr, sys_w_line,
            sys_r_line, sys_w, sys_r, rst, clk);
42      gpio chip1(g1_out, g1_in, g1_dir, 32'hC, sys_w_addr, sys_r_addr, sys_w_line,
            sys_r_line, sys_w, sys_r, rst, clk);
43      gpio chip2(g2_out, g2_in, g2_dir, 32'hE, sys_w_addr, sys_r_addr, sys_w_line,
            sys_r_line, sys_w, sys_r, rst, clk);
44      gpio chip3(g3_out, g3_in, g3_dir, 32'h10, sys_w_addr, sys_r_addr, sys_w_line,
             sys_r_line, sys_w, sys_r, rst, clk);
45
46      //here comes all other peripherals
47
48      wire [31:0] mx0_f0_out, mx0_f1_out, mx0_f2_out, mx0_f3_out;
49      wire [31:0] mx1_f0_out, mx1_f1_out, mx1_f2_out, mx1_f3_out;
50      wire [31:0] mx2_f0_out, mx2_f1_out, mx2_f2_out, mx2_f3_out;
51      wire [31:0] mx3_f0_out, mx3_f1_out, mx3_f2_out, mx3_f3_out;
52
53      wire [31:0] mx0_f0_in, mx0_f1_in, mx0_f2_in, mx0_f3_in;
54      wire [31:0] mx1_f0_in, mx1_f1_in, mx1_f2_in, mx1_f3_in;
55      wire [31:0] mx2_f0_in, mx2_f1_in, mx2_f2_in, mx2_f3_in;
56      wire [31:0] mx3_f0_in, mx3_f1_in, mx3_f2_in, mx3_f3_in;
57
58      wire [31:0] mx0_f0_dir, mx0_f1_dir, mx0_f2_dir, mx0_f3_dir;
59      wire [31:0] mx1_f0_dir, mx1_f1_dir, mx1_f2_dir, mx1_f3_dir;
60      wire [31:0] mx2_f0_dir, mx2_f1_dir, mx2_f2_dir, mx2_f3_dir;
61      wire [31:0] mx3_f0_dir, mx3_f1_dir, mx3_f2_dir, mx3_f3_dir;
62
63      gpio_mux mx0(pins[31:0], mx0_f0_in, mx0_f1_in, mx0_f2_in, mx0_f3_in,
            mx0_f0_out, mx0_f1_out, mx0_f2_out, mx0_f3_out, mx0_f0_dir, mx0_f1_dir,
            mx0_f2_dir, mx0_f3_dir, 32'h2, sys_w_addr, sys_r_addr, sys_w_line,
            sys_r_line, sys_w, sys_r, rst, clk);
64      gpio_mux mx1(pins[63:32], mx1_f0_in, mx1_f1_in, mx1_f2_in, mx1_f3_in,
            mx1_f0_out, mx1_f1_out, mx1_f2_out, mx1_f3_out, mx1_f0_dir, mx1_f1_dir,
```

```
        mx1_f2_dir, mx1_f3_dir, 32'h4, sys_w_addr, sys_r_addr, sys_w_line,
        sys_r_line, sys_w, sys_r, rst, clk);
65      gpio_mux mx2(pins[95:64], mx2_f0_in, mx2_f1_in, mx2_f2_in, mx2_f3_in,
        mx2_f0_out, mx2_f1_out, mx2_f2_out, mx2_f3_out, mx2_f0_dir, mx2_f1_dir,
        mx2_f2_dir, mx2_f3_dir, 32'h6, sys_w_addr, sys_r_addr, sys_w_line,
        sys_r_line, sys_w, sys_r, rst, clk);
66      gpio_mux mx3(pins[127:96], mx3_f0_in, mx3_f1_in, mx3_f2_in, mx3_f3_in,
        mx3_f0_out, mx3_f1_out, mx3_f2_out, mx3_f3_out, mx3_f0_dir, mx3_f1_dir,
        mx3_f2_dir, mx3_f3_dir, 32'h8, sys_w_addr, sys_r_addr, sys_w_line,
        sys_r_line, sys_w, sys_r, rst, clk);
67
68      //here comes function assignments
69      assign g0_in = mx0_f0_in, mx0_f0_out = g0_out, mx0_f0_dir = g0_dir;
70      assign g1_in = mx1_f0_in, mx1_f0_out = g1_out, mx1_f0_dir = g1_dir;
71      assign g2_in = mx2_f0_in, mx2_f0_out = g2_out, mx2_f0_dir = g2_dir;
72      assign g3_in = mx3_f0_in, mx3_f0_out = g3_out, mx3_f0_dir = g3_dir;
73  endmodule
```

11.1.14 test_pipeline_assembly.v

```
1   `timescale 1 ns / 100 ps
2
3   `include "execute.v"
4   `include "memory_op.v"
5   `include "register_wb.v"
6   `include "pipeline_interface.v"
7   `include "insn_decoder.v"
8   `include "regs.v"
9
10  /*module test_pipeline_assembly(e_a, e_b, e_alu_op, e_is_cond, e_cond,
        e_write_flags, e_swp, m_a1, m_a2, m_r1_op, m_r2_op, r_a1, r_a2, r_op, pass,
        pcincr, clk, rst);
11      input [31:0] e_a, e_b;
12      input [4:0] e_ra1, e_ra2;
13      input [3:0] e_rop;
14      input [7:0] e_alu_op;
15      input [3:0] e_cond;
16      input [3:0] e_write_flags;
17      input e_swp;
18      input e_is_cond;
19      input [31:0] m_a1, m_a2;
20      input [3:0] m_r1_op, m_r2_op;
21      input [4:0] r_a1, r_a2;
22      input [3:0] r_op;
23      input pass;
24      input pcincr;*/
25
26  // BEWARE:
27  // general rule for continuous assignment statements
```

```
28 // you can use continuous assignment in instantiation (e.g. wire a = b;) only if
       a - input and b - output
29 // if we got reverse situation, we must provide good continuous assignment below
       ( assign b = a )
30 // "continuous assignment is not bidirectional; it have dataflow directed from
       rvalue to lvalue"
31
32 module test_pipeline_assembly(ram_w_addr, ram_r_addr, ram_w_line, ram_read,
       ram_write, sys_w_addr, sys_r_addr, sys_w_line, sys_read, sys_write, lr, sp,
       pc, st, word, ram_r_line, sys_r_line, clk, rst);
33     input [31:0] word;
34
35     input clk, rst;
36
37     output wire [31:0] ram_w_addr, ram_r_addr;
38     output wire [31:0] ram_w_line;
39     input  [31:0] ram_r_line;
40     output wire ram_read, ram_write;
41
42     output wire [31:0] sys_w_addr, sys_r_addr;
43     output wire [31:0] sys_w_line;
44     input  [31:0] sys_r_line;
45     output wire sys_read, sys_write;
46
47     output wire [31:0] lr, sp, pc, st;
48
49     /*wire [31:0] ram_w_addr, ram_r_addr;
50     wire [31:0] ram_w_line, ram_r_line;
51     wire ram_read, ram_write, ram_exception;
52     emb_ram ram0(.r_addr = ram_r_addr, .w_addr = ram_w_addr, .r_line = ram_r_line
           , .w_line = ram_w_line, .read = ram_read, .write = ram_write, .exc =
           ram_exception, .clk = clk);*/
53
54     wire [31:0] reg_a, reg_b, reg_c, reg_d; //input
55     wire [4:0] reg_a_a, reg_a_b, reg_a_c, reg_a_d; //input
56     wire [1:0] reg_read, reg_write; //input
57
58     wire [31:0] reg_lr, reg_sp, reg_pc; //output
59     wire [31:0] reg_stin, reg_stout; //input, output
60     wire reg_stwr; //input
61     wire reg_pcincr; //input
62     reg32_2x2_pc rf0(reg_a, reg_b, reg_a_a, reg_a_b, reg_a_c, reg_a_d, reg_c,
           reg_d, reg_read, reg_write, clk, rst, reg_lr, reg_sp, reg_stout, reg_pc
           , reg_stin, reg_stwr, reg_pcincr);
63
64
65     wire [31:0] e_a, e_b; //output
66     wire [7:0] e_alu_op; //output
67     wire [3:0] e_cond; //output
```

```
68      wire [3:0] e_write_flags; //output
69      wire e_swp; //output
70      wire e_is_cond; //output
71
72      wire [31:0] m_a1, m_a2; //output
73      wire [3:0] m_r1_op, m_r2_op; //output
74
75      wire [4:0] r_a1, r_a2; //output
76      wire [3:0] r_op; //output
77
78      wire d_pass; //output
79      wire d_pcincr; //output
80
81      wire [4:0] r_r1_a, r_r2_a; //output
82      assign reg_a_a = r_r1_a, reg_a_b = r_r2_a;
83      wire [1:0] r_read; //output
84      assign reg_read = r_read;
85
86      wire [31:0] d_word = word; //input
87      wire [31:0] d_r1 = reg_a, d_r2 = reg_b; //input
88      wire d_hazard; //input
89      insn_decoder dec0(e_a, e_b, e_alu_op, e_is_cond, e_cond, e_write_flags, e_swp
            , m_a1, m_a2, m_r1_op, m_r2_op, r_a1, r_a2, r_op, d_pass, d_pcincr,
            r_r1_a, r_r2_a, r_read, d_word, d_r1, d_r2, d_hazard, rst, clk);
90
91
92      wire [31:0] pi_e_a, pi_e_b; //output
93      wire [7:0] pi_e_alu_op; //output
94      wire [3:0] pi_e_cond; //output
95      wire [3:0] pi_e_write_flags; //output
96      wire pi_e_swp; //output
97      wire pi_e_is_cond; //output
98
99      wire [31:0] pi_m_a1, pi_m_a2; //output
100     wire [3:0] pi_m_r1_op, pi_m_r2_op; //output
101
102     wire [4:0] pi_r_a1, pi_r_a2; //output
103     wire [3:0] pi_r_op; //output
104
105     wire pi_d_pcincr; //output
106     assign reg_pcincr = pi_d_pcincr;
107
108     pipeline_interface pi0(
109     pi_e_a, pi_e_b, pi_e_alu_op, pi_e_is_cond, pi_e_cond, pi_e_write_flags,
            pi_e_swp, pi_m_a1, pi_m_a2, pi_m_r1_op, pi_m_r2_op, pi_r_a1, pi_r_a2,
            pi_r_op, pi_d_pcincr,
110     e_a, e_b, e_alu_op, e_is_cond, e_cond, e_write_flags, e_swp, m_a1, m_a2,
            m_r1_op, m_r2_op, r_a1, r_a2, r_op, d_pass, d_pcincr, clk, rst);
111
```

```
112
113     wire [31:0] ex_a = pi_e_a, ex_b = pi_e_b; //operands  //input
114     wire [31:0] ex_st = reg_stout; //status register  //input
115     wire [7:0] ex_alu_op = pi_e_alu_op; // alu operation  //input
116     wire ex_is_cond = pi_e_is_cond; //is a conditional command signal  //input
117     wire [3:0] ex_cond = pi_e_cond; //cc  //input
118     wire [3:0] ex_write_flags = pi_e_write_flags; //write n/z/c/v  //input
119     wire ex_swp = pi_e_swp; //swap ops?  //input
120
121     wire [31:0] ex_r1, ex_r2; //results, sync  //output
122     wire ex_n, ex_z, ex_c, ex_v; //flags, async  //output
123     wire ex_cc; //write flags, async  //output
124     wire ex_cres; //conditional results, sync  //output
125     execute ex0(ex_r1, ex_r2, ex_cres, ex_n, ex_z, ex_c, ex_v, ex_cc, ex_a, ex_b,
            ex_alu_op, ex_is_cond, ex_cond, ex_write_flags, ex_st, ex_swp, clk, rst)
            ;
126
127
128     wire sr_n = ex_n, sr_z = ex_z, sr_c = ex_c, sr_v = ex_v; //input
129     wire sr_cc = ex_cc; //input
130
131     wire [31:0] sr_st;  //output
132     assign reg_stin = sr_st;
133     wire sr_stwr; //output
134     assign reg_stwr = sr_stwr;
135     status_register_adaptor sr0(sr_st, sr_stwr, sr_n, sr_z, sr_c, sr_v, sr_cc);
136
137
138     wire [31:0] ex_m_a1, ex_m_a2; //(mem_op) //output
139     wire [3:0] ex_m_r1_op, ex_m_r2_op; //(mem_op) //output
140
141     wire [4:0] ex_r_a1, ex_r_a2; //(reg_wb) //output
142     wire [3:0] ex_r_op; //(reg_wb) //output
143     execute_stage_passthrough exh0(ex_m_a1, ex_m_a2, ex_m_r1_op, ex_m_r2_op,
            ex_r_a1, ex_r_a2, ex_r_op, pi_m_a1, pi_m_a2, pi_m_r1_op, pi_m_r2_op,
            pi_r_a1, pi_r_a2, pi_r_op, clk, rst);
144
145
146     wire [31:0] mop_r1 = ex_r1, mop_r2 = ex_r2; //inputs  //input
147     wire [31:0] mop_a1 = ex_m_a1, mop_a2 = ex_m_a2; //memory addresses  //input
148
149     wire [3:0] mop_r1_op = ex_m_r1_op, mop_r2_op = ex_m_r2_op; //operation codes
            //input
150
151     wire [31:0] mop_ram_r_line = ram_r_line, mop_sys_r_line = sys_r_line; // read
            lanes  //input
152
153     wire mop_proceed = ex_cres; //conditional code test result  //input
154
```

```
155     wire [31:0] mop_m1, mop_m2; //outputs  //output
156
157     wire [31:0] mop_ram_w_addr, mop_sys_w_addr; //write addresses  //output
158     assign ram_w_addr = mop_ram_w_addr, sys_w_addr = mop_sys_w_addr;
159     wire [31:0] mop_ram_r_addr, mop_sys_r_addr; //read addresses  //output
160     assign ram_r_addr = mop_ram_r_addr, sys_r_addr = mop_sys_r_addr;
161
162     wire [31:0] mop_ram_w_line, mop_sys_w_line; //write lanes  //output
163     assign ram_w_line = mop_ram_w_line, sys_w_line = mop_sys_w_line;
164
165     wire mop_ram_w, mop_sys_w, mop_ram_r, mop_sys_r; //read/write signals  //
            output
166     assign ram_write = mop_ram_w, sys_write = mop_sys_w, ram_read = mop_ram_r,
            sys_read = mop_sys_r;
167     memory_op mop0( mop_m1, mop_m2, mop_ram_w_addr, mop_ram_r_addr, mop_ram_w,
            mop_ram_r, mop_ram_w_line, mop_sys_w_addr, mop_sys_r_addr, mop_sys_w,
            mop_sys_r, mop_sys_w_line, mop_r1, mop_r2, mop_a1, mop_a2, mop_r1_op,
            mop_r2_op, mop_ram_r_line, mop_sys_r_line, mop_proceed, clk, rst);
168
169     wire [4:0] mop_r_a1, mop_r_a2; //(reg_wb)  //output
170     wire [3:0] mop_r_op; //(reg_wb)  //output
171     wire mop_proceed2;  //output
172     memory_op_stage_passthrough moph0(mop_r_a1, mop_r_a2, mop_r_op, mop_proceed2,
             ex_r_a1, ex_r_a2, ex_r_op, ex_cres, clk, rst);
173
174
175     wire [31:0] rwb_r1 = mop_m1, rwb_r2 = mop_m2;  //input
176     wire [4:0] rwb_a1 = mop_r_a1, rwb_a2 = mop_r_a2;  //input
177
178     wire [3:0] rwb_op = mop_r_op;  //input
179
180     wire rwb_proceed = mop_proceed2;  //input
181
182     wire [31:0] rwb_wr1, rwb_wr2;  //output
183     assign reg_c = rwb_wr1, reg_d = rwb_wr2;
184     wire [4:0] rwb_wa1, rwb_wa2;  //output
185     assign reg_a_c = rwb_wa1, reg_a_d = rwb_wa2;
186     wire [1:0] rwb_write;  //output
187     assign reg_write = rwb_write;
188     register_wb rwb0( rwb_write, rwb_wr1, rwb_wr2, rwb_wa1, rwb_wa2, rwb_r1,
            rwb_r2, rwb_a1, rwb_a2, rwb_op, rwb_proceed, clk, rst);
189
190     wire ex_hazard;
191     wire reg_hazard;
192     wire mem_hazard;
193     reg_hazard_checker hz0(ex_hazard, mem_hazard, reg_hazard, ex_r_a1, ex_r_a2,
            ex_r_op, ex_cres, mop_r_a1, mop_r_a2, mop_r_op, mop_proceed2, rwb_wa1,
            rwb_wa2, rwb_write, r_r1_a, r_r2_a, r_read);
194 `ifdef RWB_STAGE_HAZARD
```

```
195     assign d_hazard = ex_hazard || reg_hazard || mem_hazard;
196 `else
197     assign d_hazard = ex_hazard || mem_hazard;
198 `endif
199
200     assign lr = reg_lr;
201     assign pc = reg_pc;
202     assign st = reg_stout;
203     assign sp = reg_sp;
204
205 endmodule
```

11.1.15 test_processor_assembly.v

```
1  `timescale 1 ns / 100 ps
2
3  `define INTERFACE_STAGE_NO_DELAY
4  `define RWB_STAGE_HAZARD
5
6
7  `include "test_pipeline_assembly.v"
8  `include "test_periph_assembly.v"
9  `include "ram.v"
10
11 // BEWARE:
12 // general rule for continuous assignment statements
13 // you can use continuous assignment in instantiation (e.g. wire a = b;) only if
       a - input and b - output
14 // if we got reverse situation, we must provide good continuous assignment below
       ( assign b = a )
15 // "continuous assignment is not bidirectional; it have dataflow directed from
       rvalue to lvalue"
16
17 module test_processor_assembly(lr, sp, st, pc, pins,  insn, clk, rst);
18     input [31:0] insn;
19     input clk, rst;
20
21     output wire [31:0] lr, sp, st, pc; //special registers
22     inout [127:0] pins; //device pins
23
24     wire [31:0] ram_w_addr, ram_r_addr; //input
25     wire [31:0] ram_w_line, ram_r_line; //input, output
26     wire ram_read, ram_write, ram_exception; //output
27     emb_ram ram0(ram_r_addr, ram_w_addr, ram_r_line, ram_w_line, ram_read,
           ram_write, ram_exception, clk);
28
29     wire [31:0] core_word = insn; //input
30     wire [31:0] core_ram_w_addr, core_ram_r_addr; //output
31     assign ram_w_addr = core_ram_w_addr, ram_r_addr = core_ram_r_addr;
```

```
32      wire [31:0] core_ram_w_line; //output
33      assign ram_w_line = core_ram_w_line;
34      wire [31:0] core_ram_r_line = ram_r_line; //input
35      wire core_ram_read, core_ram_write; //output
36      assign ram_read = core_ram_read, ram_write = core_ram_write;
37
38      wire [31:0] core_sys_w_addr, core_sys_r_addr; //output
39      wire [31:0] core_sys_w_line; //output
40      wire [31:0] core_sys_r_line; //input
41      wire core_sys_read, core_sys_write; //output
42
43      wire [31:0] core_lr, core_sp, core_pc, core_st; //output
44      assign lr = core_lr, sp = core_sp, pc = core_pc, st = core_st;
45      test_pipeline_assembly core0(core_ram_w_addr, core_ram_r_addr,
            core_ram_w_line, core_ram_read, core_ram_write, core_sys_w_addr,
            core_sys_r_addr, core_sys_w_line, core_sys_read, core_sys_write, core_lr,
             core_sp, core_pc, core_st,  core_word, core_ram_r_line, core_sys_r_line,
             clk, rst);
46
47      test_periph_assembly periph0(pins, core_sys_w_addr, core_sys_r_addr,
            core_sys_w_line, core_sys_r_line, core_sys_write, core_sys_read, rst, clk
            );
48
49  endmodule
```

11.1.16 main.v

```
1   `timescale 1 ns / 100 ps
2
3   `include "test_processor_assembly.v"
4
5   module test_rom(word, addr);
6           input [31:0] addr;
7
8           output wire [31:0] word;
9
10          reg [31:0] insn;
11          assign word = insn;
12
13          always @(addr) begin
14                  #1;
15                  case(addr)
16                          /*32'h0: begin //(mov)nop reg 29 to reg 30
17                              insn[31:25] <= 00; insn[24:21] <= 4'b1110; insn
                                    [20:16] <= 29; insn[15:11] <= 0; insn[10:6] <=
                                    30; insn[5:1] <= 0; insn[0] <= 0;
18                          end*/
19                          32'h0: begin //movs imm to reg 30 (sp)
20                              insn[31:25] <= 33; insn[24:21] <= 4'b1110; insn
```

```
                    [20:11] <= 0; insn[10:6] <= 30; insn[5:1] <= 5'
                    b10000; insn[0] <= 0;
21                  end
22                  32'h1: begin
23                      insn <= 32'h14888;
24                  end
25                  32'h3: begin //movs imm to reg 29 (lr)
26                      insn[31:25] <= 33; insn[24:21] <= 4'b1110; insn
                        [20:11] <= 0; insn[10:6] <= 29; insn[5:1] <= 5'
                        b10000; insn[0] <= 0;
27                  end
28                  32'h4: begin
29                      insn <= 32'h22888;
30                  end
31                  32'h5: begin //add 29 and 30 to 30
32                      insn[31:25] <= 14; insn[24:21] <= 4'b1110; insn
                        [20:16] <= 29; insn[15:11] <= 30; insn[10:6] <=
                        30; insn[5:1] <= 5'b00000; insn[0] <= 0;
33                  end
34                  32'h6: begin //add imm1 and imm2 to 29
35                      insn[31:25] <= 14; insn[24:21] <= 4'b1110; insn
                        [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 29;
                         insn[5:1] <= 5'b11000; insn[0] <= 0;
36                  end
37                  32'h7: begin
38                      insn <= 32'h35942;
39                  end
40                  32'h8: begin
41                      insn <= 32'hDEADBEAF;
42                  end
43                  32'h9: begin //mul 29 and 30 to 29 and 30
44                      insn[31:25] <= 18; insn[24:21] <= 4'b1110; insn
                        [20:16] <= 29; insn[15:11] <= 30; insn[10:6] <=
                        29; insn[5:1] <= 30; insn[0] <= 0;
45                  end
46                  32'hA: begin //xor 29 and 30 to 30
47                      insn[31:25] <= 6; insn[24:21] <= 4'b1110; insn[20:16]
                         <= 29; insn[15:11] <= 30; insn[10:6] <= 30; insn
                        [5:1] <= 00; insn[0] <= 0;
48                  end
49                  32'hB: begin //csr 30 by imm to 29
50                      insn[31:25] <= 12; insn[24:21] <= 4'b1110; insn
                        [20:16] <= 30; insn[15:11] <= 0; insn[10:6] <=
                        29; insn[5:1] <= 5'b01000; insn[0] <= 0;
51                  end
52                  32'hC: begin
53                      insn <= 11;
54                  end
55                  32'hD: begin //branch to imm
```

```
56                          insn[31:25] <= 25; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 0;
                                insn[5:1] <= 5'b10000; insn[0] <= 0;
57                          end
58                          32'hE: begin
59                              insn <= 32'h132;
60                          end
61                          32'h132: begin //out 29 to 30
62                              insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 30; insn[15:11] <= 29; insn[10:6] <=
                                0; insn[5:1] <= 0; insn[0] <= 0;
63                          end
64                          32'h133: begin //out 30 to 29
65                              insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 29; insn[15:11] <= 30; insn[10:6] <=
                                0; insn[5:1] <= 0; insn[0] <= 0;
66                          end
67                          32'h134: begin //brl to 30
68                              insn[31:25] <= 27; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 30; insn[15:11] <= 0; insn[10:6] <= 0;
                                 insn[5:1] <= 5'b00000; insn[0] <= 0;
69                          end
70                          32'h135: begin //str to imm from 30
71                              insn[31:25] <= 30; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 0; insn[15:11] <= 30; insn[10:6] <= 0;
                                 insn[5:1] <= 5'b10000; insn[0] <= 0;
72                          end
73                          32'h136: begin
74                              insn <= 16;
75                          end
76                          32'h137: begin //mov 29, 30 to 30, 29
77                              insn[31:25] <= 34; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 29; insn[15:11] <= 30; insn[10:6] <=
                                30; insn[5:1] <= 29; insn[0] <= 0;
78                          end
79                          32'h138: begin //out 30 to 29
80                              insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 29; insn[15:11] <= 30; insn[10:6] <=
                                0; insn[5:1] <= 0; insn[0] <= 0;
81                          end
82                          32'h139: begin //ldr from imm to 30
83                              insn[31:25] <= 29; insn[24:21] <= 4'b1110; insn
                                [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 30;
                                 insn[5:1] <= 5'b10000; insn[0] <= 0;
84                          end
85                          32'h13A: begin
86                              insn <= 16;
87                          end
88                          32'h13B:  begin //movs imm to r1
```

```
89                      insn[31:25] <= 33; insn[24:21] <= 4'b1110; insn
                            [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 1;
                            insn[5:1] <= 5'b10000; insn[0] <= 0;
90                  end
91                  32'h13C: begin
92                      insn <= 32'hFFFFFFFF;
93                  end
94                  32'h13D: begin //out to imm from r1
95                      insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                            [20:16] <= 0; insn[15:11] <= 1; insn[10:6] <= 0;
                            insn[5:1] <= 5'b10000; insn[0] <= 0;
96                  end
97                  32'h13E: begin
98                      insn <= 32'hD;
99                  end
100                 32'h13F: begin //out to imm from r1
101                     insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                            [20:16] <= 0; insn[15:11] <= 1; insn[10:6] <= 0;
                            insn[5:1] <= 5'b10000; insn[0] <= 0;
102                 end
103                 32'h140: begin
104                     insn <= 32'hF;
105                 end
106                 32'h141: begin //out to imm from r1
107                     insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                            [20:16] <= 0; insn[15:11] <= 1; insn[10:6] <= 0;
                            insn[5:1] <= 5'b10000; insn[0] <= 0;
108                 end
109                 32'h142: begin
110                     insn <= 32'h11;
111                 end
112                 32'h143: begin //out to imm from r1
113                     insn[31:25] <= 32; insn[24:21] <= 4'b1110; insn
                            [20:16] <= 0; insn[15:11] <= 1; insn[10:6] <= 0;
                            insn[5:1] <= 5'b10000; insn[0] <= 0;
114                 end
115                 32'h144: begin
116                     insn <= 32'hE;
117                 end
118                 32'h145: begin //in from imm to 30
119                     insn[31:25] <= 31; insn[24:21] <= 4'b1110; insn
                            [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 30;
                             insn[5:1] <= 5'b10000; insn[0] <= 0;
120                 end
121                 32'h146: begin
122                     insn <= 32'hA;
123                 end
124                 32'h5E771E7D: begin //br_pos to 0
125                     insn[31:25] <= 25; insn[24:21] <= 4'b0101; insn
```

```
                                [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 0;
                                insn[5:1] <= 5'b00000; insn[0] <= 0;
126                         end
127                         32'h5E771E7E: begin //ret_neg
128                             insn[31:25] <= 28; insn[24:21] <= 4'b0100; insn
                                    [20:16] <= 0; insn[15:11] <= 0; insn[10:6] <= 0;
                                    insn[5:1] <= 5'b00000; insn[0] <= 0;
129                         end
130                         default: begin
131                             insn <= 32'b0;
132                         end
133                 endcase
134         end
135 endmodule
136
137 module fib32_rom(word, addr);
138         input [31:0] addr;
139         output reg [31:0] word;
140
141         always @* begin
142                 #1;
143                 case(addr)
144                         32'h0: begin //movs 0x00 -> r0
145                                 word[31:25] = 33; word[24:21] = 4'b1110; word
                                        [20:16] = 0; word[15:11] = 0; word[10:6] = 0;
                                         word[5:1] = 5'b10000; word[0] = 0;
146                         end
147                         32'h1: begin
148                                 word = 32'h0;
149                         end
150                         32'h2: begin //movs 0xFFFFFFFF -> r1
151                                 word[31:25] = 33; word[24:21] = 4'b1110; word
                                        [20:16] = 0; word[15:11] = 0; word[10:6] = 1;
                                         word[5:1] = 5'b10000; word[0] = 0;
152                         end
153                         32'h3: begin
154                                 word = 32'hFFFFFFFF;
155                         end
156                         32'h4: begin //movs r0 -> r2
157                                 word[31:25] = 33; word[24:21] = 4'b1110; word
                                        [20:16] = 0; word[15:11] = 0; word[10:6] = 2;
                                         word[5:1] = 5'b00000; word[0] = 0;
158                         end
159                         32'h5: begin //movs 0x01 -> r3
160                                 word[31:25] = 33; word[24:21] = 4'b1110; word
                                        [20:16] = 0; word[15:11] = 0; word[10:6] = 3;
                                         word[5:1] = 5'b10000; word[0] = 0;
161                         end
162                         32'h6: begin
```

```
163                     word = 32'h1;
164                 end
165                 32'h7: begin //movs 0x0C -> r5
166                     word[31:25] = 33; word[24:21] = 4'b1110; word
                            [20:16] = 0; word[15:11] = 0; word[10:6] = 5;
                             word[5:1] = 5'b10000; word[0] = 0;
167                 end
168                 32'h8: begin
169                     word = 32'hC;
170                 end
171                 32'h9: begin //movs 0x100 -> r6
172                     word[31:25] = 33; word[24:21] = 4'b1110; word
                            [20:16] = 0; word[15:11] = 0; word[10:6] = 6;
                             word[5:1] = 5'b10000; word[0] = 0;
173                 end
174                 32'hA: begin
175                     word = 32'h100;
176                 end
177                 32'hB: begin //movs -0x03 -> r7
178                     word[31:25] = 33; word[24:21] = 4'b1110; word
                            [20:16] = 0; word[15:11] = 0; word[10:6] = 7;
                             word[5:1] = 5'b10000; word[0] = 0;
179                 end
180                 32'hC: begin
181                     word = 32'hFFFFFFFD;
182                 end
183                 32'hD: begin //out r1 -> 0x0D
184                     word[31:25] = 32; word[24:21] = 4'b1110; word
                            [20:16] = 0; word[15:11] = 1; word[10:6] = 0;
                             word[5:1] = 5'b10000; word[0] = 0;
185                 end
186                 32'hE: begin
187                     word = 32'hD;
188                 end
189                 32'hF: begin //out r3 -> [r5]
190                     word[31:25] = 32; word[24:21] = 4'b1110; word
                            [20:16] = 5; word[15:11] = 3; word[10:6] = 0;
                             word[5:1] = 5'b00000; word[0] = 0;
191                 end
192                 32'h10: begin //brl [r6]
193                     word[31:25] = 27; word[24:21] = 4'b1110; word
                            [20:16] = 6; word[15:11] = 0; word[10:6] = 0;
                             word[5:1] = 5'b00000; word[0] = 0;
194                 end
195                 32'h11: begin //out_lo r4 -> [r5]
196                     word[31:25] = 32; word[24:21] = 4'b0011; word
                            [20:16] = 5; word[15:11] = 4; word[10:6] = 0;
                             word[5:1] = 5'b00000; word[0] = 0;
197                 end
```

```
198                             32'h12: begin //rbr_lo pc+r7
199                                     word[31:25] = 26; word[24:21] = 4'b0011; word
                                            [20:16] = 7; word[15:11] = 0; word[10:6] = 0;
                                             word[5:1] = 5'b00000; word[0] = 0;
200                             end
201                             32'h13: begin //br r0
202                                     word[31:25] = 25; word[24:21] = 4'b1110; word
                                            [20:16] = 0; word[15:11] = 0; word[10:6] = 0;
                                             word[5:1] = 5'b00000; word[0] = 0;
203                             end
204                             //fib()
205                             32'h100: begin //add r2, r3 -> r4
206                                     word[31:25] = 14; word[24:21] = 4'b1110; word
                                            [20:16] = 2; word[15:11] = 3; word[10:6] = 4;
                                             word[5:1] = 5'b00000; word[0] = 0;
207                             end
208                             32'h101: begin //mov r3,r4 -> r2, r3
209                                     word[31:25] = 34; word[24:21] = 4'b1110; word
                                            [20:16] = 3; word[15:11] = 4; word[10:6] = 2;
                                             word[5:1] = 3; word[0] = 0;
210                             end
211                             32'h102: begin //ret
212                                     word[31:25] = 28; word[24:21] = 4'b1110; word
                                            [20:16] = 0; word[15:11] = 0; word[10:6] = 0;
                                             word[5:1] = 5'b00000; word[0] = 0;
213                             end
214                             default: begin //nop
215                                     word = 32'h0;
216                             end
217                     endcase
218             end
219     endmodule
220
221     //assembly test
222     module main();
223         wire [31:0] insn;
224         wire [31:0] lr, sp, st, pc;
225         wire [31:0] pins0, pins1, pins2, pins3;
226
227         reg clk;
228         reg rst;
229
230         test_processor_assembly proc0(lr, sp, st, pc, {pins3, pins2, pins1, pins0},
                insn, clk, rst);
231
232         fib32_rom rom0(insn, pc);
233
234         assign pins0[15:0] = 16'h1488;
235
```

```
236     initial begin
237         //insn = 32'b0; //nop
238         clk = 0;
239         rst = 0;
240         $dumpfile("dump.fst");
241         $dumpvars(0);
242         $dumpon;
243     end
244     always begin
245         integer i;
246         //reset
247         rst = 0;
248         #20;
249         rst = 1;
250         #20;
251         rst = 0;
252         #20;
253 
254         //clock 128 times
255         for(i =0; i < 1024+128; i++) begin
256             #20;
257             clk = 1;
258             #20;
259             clk = 0;
260         end
261         //finish
262         $dumpflush;
263         $finish;
264     end
265 endmodule
266 
267 
268 //shifter test
269 function [31:0] rotr;
270     input[31:0] a;
271     input[4:0] b;
272     rotr = ( a >> b) | (a << ((-b) & 31));
273 endfunction
274 
275 function [31:0] rotl;
276     input[31:0] a;
277     input[4:0] b;
278     rotl = ( a << b) | (a >> ((-b) & 31));
279 endfunction
280 
281 function [31:0] sal;
282     input[31:0] a;
283     input[4:0] b;
284     sal[30:0] = (a[30:0] << b);
```

```
285     sal[31] = a[31];
286 endfunction
287
288 function [31:0] sar;
289     input[31:0] a;
290     input[4:0] b;
291
292     integer x;
293     x = a;
294     //if(a[31]) x = -x;
295     sar = x >>> b;
296 endfunction
```

11.2 MultiplierGenerator

11.2.1 Gate.hpp

```
1  #ifndef GATE_HPP_INCLUDED
2  #define GATE_HPP_INCLUDED
3
4  #include <string>
5  #include <iostream>
6  #include <atomic>
7  #include <vector>
8  #include <cmath>
9  #include <stdexcept>
10
11 using namespace std;
12
13 class Gate{
14 public:
15     virtual void genWire(std::ostream& out) = 0;
16     virtual void genInst(std::ostream& out) = 0;
17     virtual string name() = 0;
18     virtual unsigned int count() = 0;
19
20     virtual ~Gate() {};
21 };
22
23 class InputGate: public Gate{
24     string vnm;
25     unsigned int vref;
26 public:
27     InputGate(): vnm(), vref(0) {};
28     InputGate(std::string vname, unsigned int vnumber): vnm(vname), vref(vnumber)
          {};
29
30     void genWire(std::ostream& out) override{
31         return;
```

```
32      }
33
34      void genInst(std::ostream& out) override{
35          return;
36      }
37
38      string name() override{
39          return (vnm + "[" + to_string(vref)+"]");
40      }
41
42      unsigned int count() override{ return 0;}
43  };
44
45  class OutputGate: public Gate{
46      string vnm;
47      unsigned int vref;
48      Gate* in;
49  public:
50      OutputGate(): vnm(), vref(0), in(nullptr) {};
51      OutputGate(std::string vname, unsigned int vnumber, Gate* in1): vnm(vname),
            vref(vnumber), in(in1) {};
52
53      void genWire(std::ostream& out) override{
54          out << "\tassign␣" << vnm << "[" << vref << "]" << "␣=␣" << in->name() <<
                ";\n";
55      }
56
57      void genInst(std::ostream& out) override{
58          return;
59      }
60
61      string name() override{
62          return (vnm + "[" + to_string(vref)+"]");
63      }
64      unsigned int count() override{ return 0;}
65  };
66
67  class ANDGate: public Gate{
68      unsigned int nref;
69      Gate *in1, *in2;
70
71      static atomic_uint cnt;
72  public:
73      ANDGate(): nref(cnt++), in1(nullptr), in2(nullptr) {};
74      ANDGate(Gate* in1, Gate* in2): nref(cnt++), in1(in1), in2(in2) {};
75
76      void genWire(std::ostream& out) override{
77          out << "\twire␣wand_" << nref << ";\n";
78      }
```

```
79
80      void genInst(std::ostream& out) override{
81          out << "\tand␣#1␣and_" << nref << "(␣wand_" << nref <<",␣" << in1->name()
                << ",␣" << in2->name() << ");\n";
82      }
83
84      string name() override{
85          return ("wand_" + to_string(nref));
86      }
87      unsigned int count() override{ return 1;}
88  };
89
90  class FAProvider: public Gate{
91      unsigned int nref;
92      Gate *in1, *in2, *in3;
93
94      static atomic_uint cnt;
95  public:
96      FAProvider(): nref(cnt++), in1(nullptr), in2(nullptr), in3(nullptr) {};
97      FAProvider(Gate* a, Gate* b, Gate* _cin): nref(cnt++), in1(a), in2(b), in3(
            _cin) {};
98
99      void genWire(std::ostream& out) override{
100         out << "\twire␣wfa_s_" << nref << ",␣wfa_cout_" << nref << ";\n";
101     }
102
103     void genInst(std::ostream& out) override{
104         out << "\tfa␣fa_" << nref << "(␣" << in1->name() << ",␣" << in2->name()
                << ",␣" << in3->name() << ",␣wfa_s_" << nref << ",␣wfa_cout_" << nref
                 << ");\n";
105     }
106
107     string name() override{
108         ///It is only provider, not node
109         return string();
110     }
111
112     unsigned int count() override{ return 5;}
113
114     unsigned int getRef(){ return nref; }
115 };
116
117 class FANode_S: public Gate{
118     FAProvider* p;
119 public:
120     FANode_S(): p(nullptr) {};
121     FANode_S(FAProvider* prov): p(prov) {};
122
123     void genWire(std::ostream& out) override{
```

```
124         ///It's only node, not provider
125         return;
126     }
127 
128     void genInst(std::ostream& out) override{
129         ///It's only node, not provider
130         return;
131     }
132 
133     string name() override{
134         return ("wfa_s_" + to_string(p->getRef()));
135     }
136 
137     unsigned int count() override{ return 0;}
138 };
139 
140 class FANode_Cout: public Gate{
141     FAProvider* p;
142 public:
143     FANode_Cout(): p(nullptr) {};
144     FANode_Cout(FAProvider* prov): p(prov) {};
145 
146     void genWire(std::ostream& out) override{
147         ///It's only node, not provider
148         return;
149     }
150 
151     void genInst(std::ostream& out) override{
152         ///It's only node, not provider
153         return;
154     }
155 
156     string name() override{
157         return ("wfa_cout_" + to_string(p->getRef()));
158     }
159 
160     unsigned int count() override{ return 0;}
161 };
162 
163 class HAProvider: public Gate{
164     unsigned int nref;
165     Gate *in1, *in2;
166 
167     static atomic_uint cnt;
168 public:
169     HAProvider(): nref(cnt++), in1(nullptr), in2(nullptr) {};
170     HAProvider(Gate* a, Gate* b): nref(cnt++), in1(a), in2(b) {};
171 
172     void genWire(std::ostream& out) override{
```

```
173             out << "\twire␣wha_s_" << nref << ",␣wha_c_" << nref << ";\n";
174         }
175
176         void genInst(std::ostream& out) override{
177             out << "\tha␣ha_" << nref << "(␣" << in1->name() << ",␣" << in2->name()
                    << ",␣wha_s_" << nref << ",␣wha_c_" << nref << ");\n";
178         }
179
180         string name() override{
181             ///It is only provider, not node
182             return string();
183         }
184
185         unsigned int count() override{ return 2;}
186
187         unsigned int getRef(){ return nref; }
188     };
189
190     class HANode_S: public Gate{
191         HAProvider* p;
192     public:
193         HANode_S(): p(nullptr) {};
194         HANode_S(HAProvider* prov): p(prov) {};
195
196         void genWire(std::ostream& out) override{
197             ///It's only node, not provider
198             return;
199         }
200
201         void genInst(std::ostream& out) override{
202             ///It's only node, not provider
203             return;
204         }
205
206         string name() override{
207             return ("wha_s_" + to_string(p->getRef()));
208         }
209
210         unsigned int count() override{ return 0;}
211     };
212
213     class HANode_C: public Gate{
214         HAProvider* p;
215     public:
216         HANode_C(): p(nullptr) {};
217         HANode_C(HAProvider* prov): p(prov) {};
218
219         void genWire(std::ostream& out) override{
220             ///It's only node, not provider
```

```
221             return;
222         }
223
224         void genInst(std::ostream& out) override{
225             ///It's only node, not provider
226             return;
227         }
228
229         string name() override{
230             return ("wha_c_" + to_string(p->getRef()));
231         }
232
233         unsigned int count() override{ return 0;}
234     };
235
236     atomic_uint ANDGate::cnt;
237     atomic_uint HAProvider::cnt;
238     atomic_uint FAProvider::cnt;
239
240     void gen_mult(std::ostream& out, unsigned int opsz){
241         vector<unsigned int> gen_seq;
242         gen_seq.push_back(2);
243         while(gen_seq.back() < opsz){
244             unsigned int cur_seq = gen_seq.back();
245             gen_seq.push_back( (unsigned int)(floor(3.0*cur_seq/2.0)) );
246         }
247         gen_seq.pop_back();
248
249         vector<Gate*>* wg = new vector<Gate*> [2*opsz];
250         vector<Gate*> ins1;
251         vector<Gate*> ins2;
252         vector<Gate*> used;
253         vector<Gate*> outs;
254
255         for(unsigned int i = 0; i < opsz; i++){
256             ins1.push_back(new InputGate("a", i));
257             ins2.push_back(new InputGate("b", i));
258         }
259
260         cout << "Input␣generation\n";
261         for(unsigned int i = 0; i < opsz; i++){
262             for(unsigned int j = 0; j < opsz; j++){
263                 wg[i+j].push_back(new ANDGate(ins1[i], ins2[j]));
264             }
265         }
266         for(unsigned int i = 0; i < 2*opsz - 1; i++) cout << "␣Weight␣" << i << ",␣
                length␣" << wg[i].size() << "\n";
267         cout << "\n\n";
268
```

```
269     unsigned int i = 1;
270     while(gen_seq.size() > 0){ ///Reduce vectors towards ready-to-use entities (
            auto generate last adders layer)
271         unsigned int cur = gen_seq.back();
272         gen_seq.pop_back();
273         cout << "\nLayer " << i << ", target " << cur << "\n";
274         i++;
275 
276         for(unsigned int w = 0; w < 2*opsz - 1; w++){
277             cout << " Weight " << w << ", length " << wg[w].size() << "\n";
278             if(wg[w].size() > cur){
279                 vector<Gate*>& gs = wg[w];
280                 vector<Gate*>& ngs = wg[w+1];
281                 unsigned int s = gs.size();
282                 while(s > cur){
283                     if((s - cur) >= 2){ ///Insert Full Adder
284                         Gate* a = gs[0];
285                         Gate* b = gs[1];
286                         Gate* _cin = gs[2];
287                         gs.erase(gs.begin(), gs.begin()+3);
288                         FAProvider* fa = new FAProvider(a, b, _cin);
289                         gs.push_back(new FANode_S(fa));
290                         ngs.push_back(new FANode_Cout(fa));
291                         used.push_back(a), used.push_back(b), used.push_back(_cin
                                ), used.push_back(fa);
292                         s -= 2;
293                         cout << "  Inserted Full Adder, now " << s << "\n";
294                     }
295                     else if((s - cur) == 1){ ///Insert Half Adder
296                         Gate* a = gs[0];
297                         Gate* b = gs[1];
298                         gs.erase(gs.begin(), gs.begin()+2);
299                         HAProvider* ha = new HAProvider(a, b);
300                         gs.push_back(new HANode_S(ha));
301                         ngs.push_back(new HANode_C(ha));
302                         used.push_back(a), used.push_back(b), used.push_back(ha);
303                         s -= 1;
304                         cout << "  Inserted Half Adder, now " << s << "\n";
305                     }
306                     else if((s - cur) == 0){ ///Connect to next layer
307                         s -= 0;
308                         cout << "  Passed to next layer, now" << s << "\n";
309                     }
310                     else throw runtime_error("Bad condition in place #1");
311                 }
312             }
313         }
314     }
315     cout << "\n\n";
```

```
316     ///Check if we're have good vectors
317     if(wg[0].size() != 1) throw runtime_error("First␣vector␣have␣" + to_string(wg
            [0].size()) + "␣entities␣in␣it␣instead␣of␣1");
318     for(unsigned int i = 1; i < 2*opsz - 1; i++)
319         if(wg[i].size() != 2) throw runtime_error("Vector␣" + to_string(i) + "␣
                have␣size␣" + to_string(wg[i].size()) + "␣instead␣of␣2␣after␣
                reduction");
320     if(wg[2*opsz - 1].size() != 0) throw runtime_error("Last␣(fill)␣vector␣have␣"
             + to_string(wg[2*opsz].size()) + "␣entities␣in␣it␣instead␣of␣0");
321
322     ///Add last two layers
323     cout << "Outputs␣layer,␣target␣1" << endl;
324     for(unsigned int w = 0; w < 2*opsz; w++){
325         vector<Gate*>& gs = wg[w];
326         vector<Gate*>& ngs = wg[w+1];
327         unsigned int s = gs.size();
328         cout << "␣Weight␣" << w << ",␣length␣" << s << endl;
329         if(s == 2){
330             Gate* a = gs[0];
331             Gate* b = gs[1];
332             gs.erase(gs.begin(), gs.begin()+2);
333             HAProvider* ha = new HAProvider(a, b);
334             gs.push_back(new HANode_S(ha));
335             ngs.push_back(new HANode_C(ha));
336             used.push_back(a), used.push_back(b), used.push_back(ha);
337             s -= 1;
338             cout << "␣␣Inserted␣Half␣Adder,␣now␣" << s << endl;
339         } else if( s == 3){
340             Gate* a = gs[0];
341             Gate* b = gs[1];
342             Gate* _cin = gs[2];
343             gs.erase(gs.begin(), gs.begin()+3);
344             FAProvider* fa = new FAProvider(a, b, _cin);
345             gs.push_back(new FANode_S(fa));
346             ngs.push_back(new FANode_Cout(fa));
347             used.push_back(a), used.push_back(b), used.push_back(_cin), used.
                    push_back(fa);
348             s -= 2;
349             cout << "␣␣Inserted␣Full␣Adder,␣now␣" << s << endl;
350         } else if (s == 1){
351             cout << "␣␣Passed␣to␣the␣outputs␣layer" << endl;
352         }
353     }
354     cout << "\n\n";
355
356     ///Generate outputs
357     for(unsigned int i = 0; i < 2*opsz; i++){
358         Gate* ow = wg[i][0];
359         wg[i].clear();
```

```
360             outs.push_back(new OutputGate("m", i, ow));
361             used.push_back(ow);
362         }
363 
364         unsigned long int gates_number = 0;
365         for(Gate* i: used) gates_number += i->count();
366 
367         cout << "Approx. gates count: " << gates_number << "\n" << endl;
368 
369         ///Generate rtl representation
370         for(Gate* i: used) i->genWire(out);
371         out << "\n";
372         for(Gate* i: used) i->genInst(out);
373         out << "\n";
374         for(Gate* i: outs) i->genWire(out);
375         for(Gate* i: outs) i->genInst(out);
376 
377         ///Cleanup
378         delete [] wg;
379         for(Gate* i: ins1) delete i;
380         for(Gate* i: ins2) delete i;
381         for(Gate* i: used) delete i;
382         for(Gate* i: outs) delete i;
383     }
384 
385     void gen_incls(std::ostream& out){
386         ///generate full adder
387         out << "module fa(a,b,cin, s, cout);\n";
388         out << "\tinput a;\n";
389         out << "\tinput b;\n";
390         out << "\tinput cin;\n";
391         out << "\n";
392         out << "\toutput s;\n";
393         out << "\toutput cout;\n";
394         out << "\n";
395         out << "\twire w1, w2, w3;\n";
396         out << "\n";
397         out << "\txor #1 x1(w1, a, b);\n";
398         out << "\txor #1 x2(s, w1, cin);\n";
399         out << "\n";
400         out << "\tand #1 a1(w2, a, b);\n";
401         out << "\tand #1 a2(w3, w1, cin);\n";
402         out << "\tor #1 o1(cout, w2, w3);\n";
403         out << "endmodule\n";
404         out << "\n";
405 
406         ///generate half adder
407         out << "module ha(a,b,s,c);\n";
408         out << "\tinput a;\n";
```

```
409         out << "\tinput␣b;\n";
410         out << "\n";
411         out << "\toutput␣s;\n";
412         out << "\toutput␣c;\n";
413         out << "\n";
414         out << "\txor␣#1␣x(s,␣a,␣b);\n";
415         out << "\tand␣#1␣an(c,␣a,␣b);\n";
416         out << "endmodule\n";
417         out << "\n";
418     }
419
420     void gen_module_decl(std::ostream& out, unsigned int opsz){
421         out << "module␣mult_" << opsz << "(a,␣b,␣m);\n";
422         out << "\tinput␣[" << (opsz-1) << ":0]␣a;\n";
423         out << "\tinput␣[" << (opsz-1) << ":0]␣b;\n";
424         out << "\n";
425         out << "\toutput␣[" << (2*opsz-1) << ":0]␣m;\n";
426         out << "\n\n";
427     }
428
429     void gen_module_end(std::ostream& out){
430         out << "\n";
431         out << "endmodule\n";
432         out << "\n";
433     }
434
435     void gen_header(std::ostream& out, unsigned int opsz){
436         out << "//␣This␣file␣is␣generated␣with␣MultiplierGenerator␣from␣CPU32␣project
                .\n";
437         out << "//␣(c)␣DeD␣MorozZz\n";
438         out << "//␣This␣is␣" << opsz << "x" << opsz << "␣bits␣parallel␣multiplier,␣
                Dadda␣tree␣design.\n";
439         out << "\n";
440     }
441
442     #endif // GATE_HPP_INCLUDED
```

11.2.2 Main.cpp

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4
5   #include "Gate.hpp"
6
7   using namespace std;
8
9   int main(int argc, char** argv){
10      if(argc != 3){
```

```
11          cout << "Dadda Tree Multiplier Verilog representation generator.\n\tUsage
                :\n\t" << argv[0] << " <opsz> <outfile>" << endl;
12          return 0;
13      }
14
15      ofstream out(argv[2]);
16      if(!out.is_open()){
17          cout << "Can't open outfile" << endl;
18          return -1;
19      }
20
21      unsigned int opsz = atoi(argv[1]);
22
23      gen_header(out, opsz);
24
25      gen_incls(out);
26
27      gen_module_decl(out, opsz);
28
29      gen_mult(out, opsz);
30
31      gen_module_end(out);
32
33      out.close();
34
35      return 0;
36  }
```

11.2.3 testcase.v

```
1   `timescale 1 ns / 10 ps
2
3   `include "test.v"
4
5   module main();
6       parameter s = 32;
7       parameter mx = 1 << s;
8       parameter dl = 64;
9       reg [s-1:0] a;
10      reg [s-1:0] b;
11
12      wire [2*s-1:0] m;
13
14      reg [s:0] i;
15      reg [s:0] j;
16
17      mult_8 mult(a, b, m);
18
19      initial begin
```

```
20          a = 0;
21          b = 0;
22          $dumpfile("dump.fst");
23          $dumpvars(0);
24          $dumpon;
25      end
26
27      always begin
28          for(i = 0; i < mx; i++ ) begin
29              for(j = 0; j < mx; j++) begin
30                  a = i[s-1:0];
31                  b = j[s-1:0];
32                  #dl;
33                  if(m != a*b) $display("Multiply error: %x*%x = ? %x", a, b, m);
34              end
35          end
36          a = 0;
37          b = 0;
38          #dl;
39          if(m != 0) $display("Multiply error *: 0*0 = ? 0");
40          $dumpflush;
41          $finish;
42      end
43  endmodule
```

12 Метрики кода

12.1 Процессор УП-1

В таблице 2 представлены метрики кода проекта процессора УП-1. Файл mult.v в основной расчёт (без скобок) не берётся, т.к. он сгенерирован программой из проекта MultiplierGenerator. Число в скобках отображает метрики с включением сгенерированного mult.v.

Файл	Язык	Пустых строк	Комментариев	Строк кода
mult.v (GENERATED)	Verilog	- (17)	- (3)	- (4123)
insn_decoder.v	Verilog	43	148	530
main.v	Verilog	20	11	265
memory_op.v	Verilog	18	33	211
alu.v	Verilog	44	6	197
shift.v	Verilog	57	53	158
test_pipeline_assembly.v	Verilog	51	24	130
execute.v	Verilog	25	0	102
adder.v	Verilog	29	3	82
ram.v	Verilog	17	18	78
pipeline_interface.v	Verilog	18	0	77
gpio_mux.v	Verilog	10	9	69
register_wb.v	Verilog	9	0	66
gpio.v	Verilog	7	5	51
test_periph_assembly.v	Verilog	12	20	41
regs.v	Verilog	15	4	37
test_processor_assembly.v	Verilog	12	5	32
ВСЕГО	Verilog	387 (404)	339 (342)	2126 (6249)

Таблица 2: Метрики кода проекта CPU32

12.2 MultiplierGenerator

В таблице 3 представлены метрики кода проекта генератора уможителей Дадды

Файл	Язык	Пустых строк	Комментариев	Строк кода
Gate.hpp	C++	71	17	354
testcase.v	Verilog	8	0	36
Main.cpp	C++	12	0	24
ВСЕГО	C++	83	17	378
	Verilog	8	0	36
	ВСЕ	91	17	414

Таблица 3: Метрики кода проекта MultiplierGenerator