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, поэтому было проведено некоторое 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.

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[4] 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.

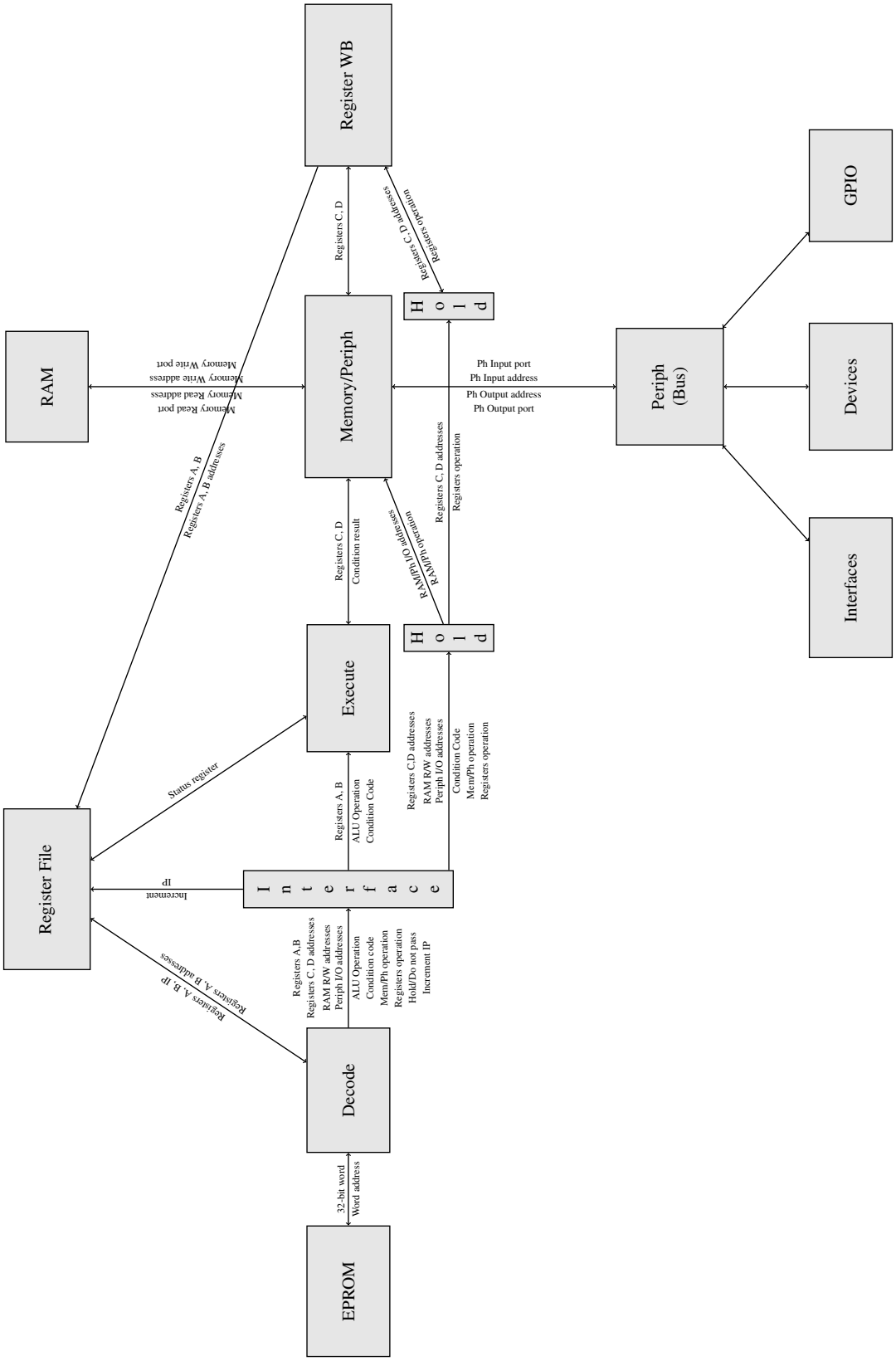


Рис. 1: Ядро (схема)

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 сигнал ошибки конвейера активируется (выставляется в единицу), и декодер приостанавливает выполнение инструкции пока сигнал не деактивируется (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

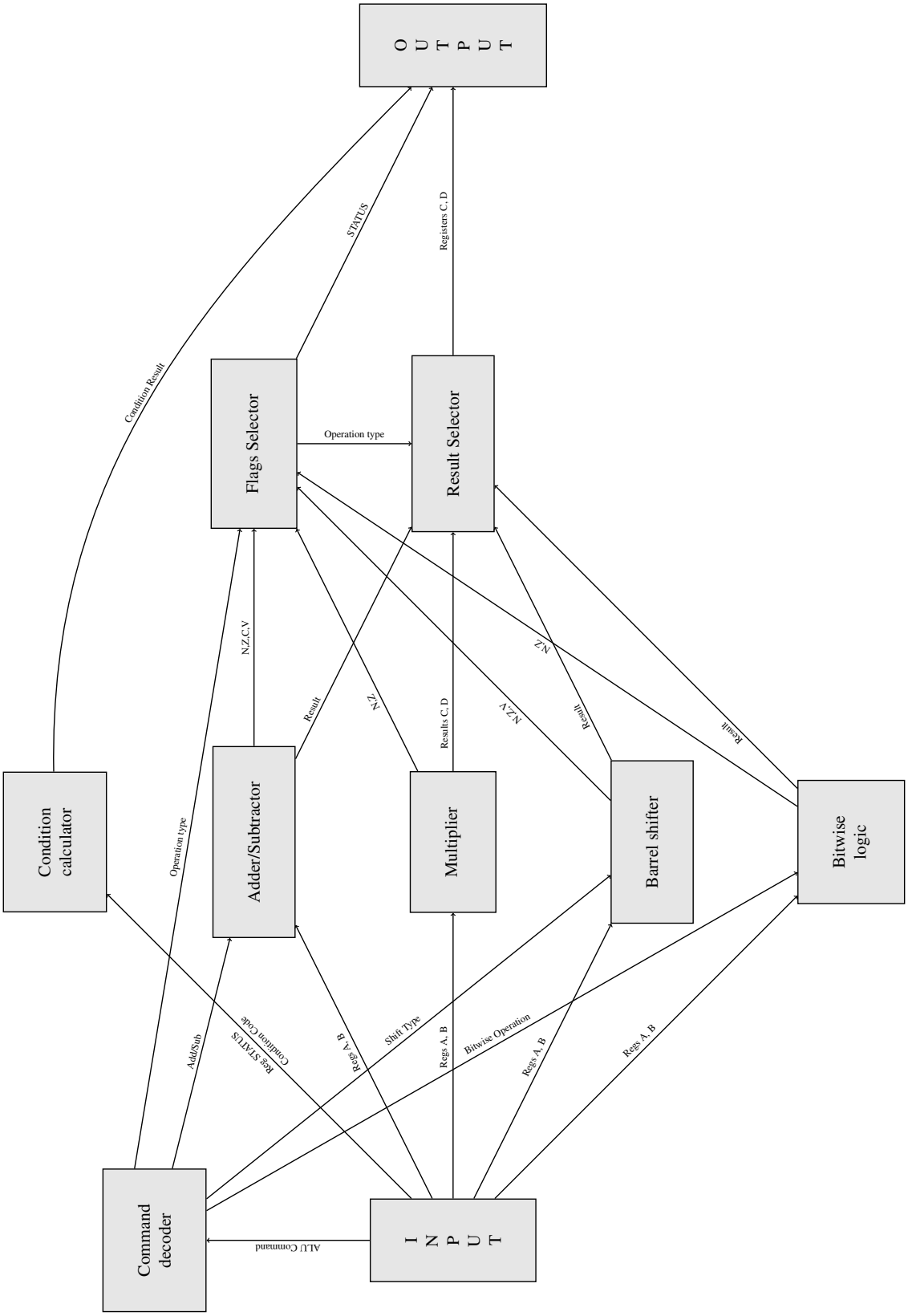


Рис. 2: АЛУ (схема)

3.2 Сумматор/Вычитатель

Сумматор-вычитатель построен по схеме сумматора с параллельным переносом[2]. 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 [3]. 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[1]. 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] - $q_1 = a\ ror\ b$, $q_2 = 0$

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

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

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 здесь 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

[6]Циклический сдвиг влево

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 регистров устройств, если таких несколько) назначается уникальный адрес.

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:

1. **00000 - 00001**: Пусто (защита от случайной перезаписи)
2. **00010 - 00011**: Мультиплексор на ноги 0-31
3. **00100 - 00101**: Мультиплексор на ноги 63-32
4. **00110 - 00111**: Мультиплексор на ноги 95-64
5. **01000 - 01001**: Мультиплексор на ноги 127-96
6. **01010 - 01011**: Контроллер GPIO на первый мультиплексор (ноги 0-31)
7. **01100 - 01101**: Контроллер GPIO на второй мультиплексор (ноги 63-32)
8. **01110 - 01111**: Контроллер GPIO на третий мультиплексор (ноги 95-64)
9. **10000 - 10001**: Контроллер GPIO на четвёртый мультиплексор (ноги 127-96)

Часть III

Результаты

1 Симуляция

1.1 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 любая 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».

1.2 Тестовая программа

1.2.1 Описание

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

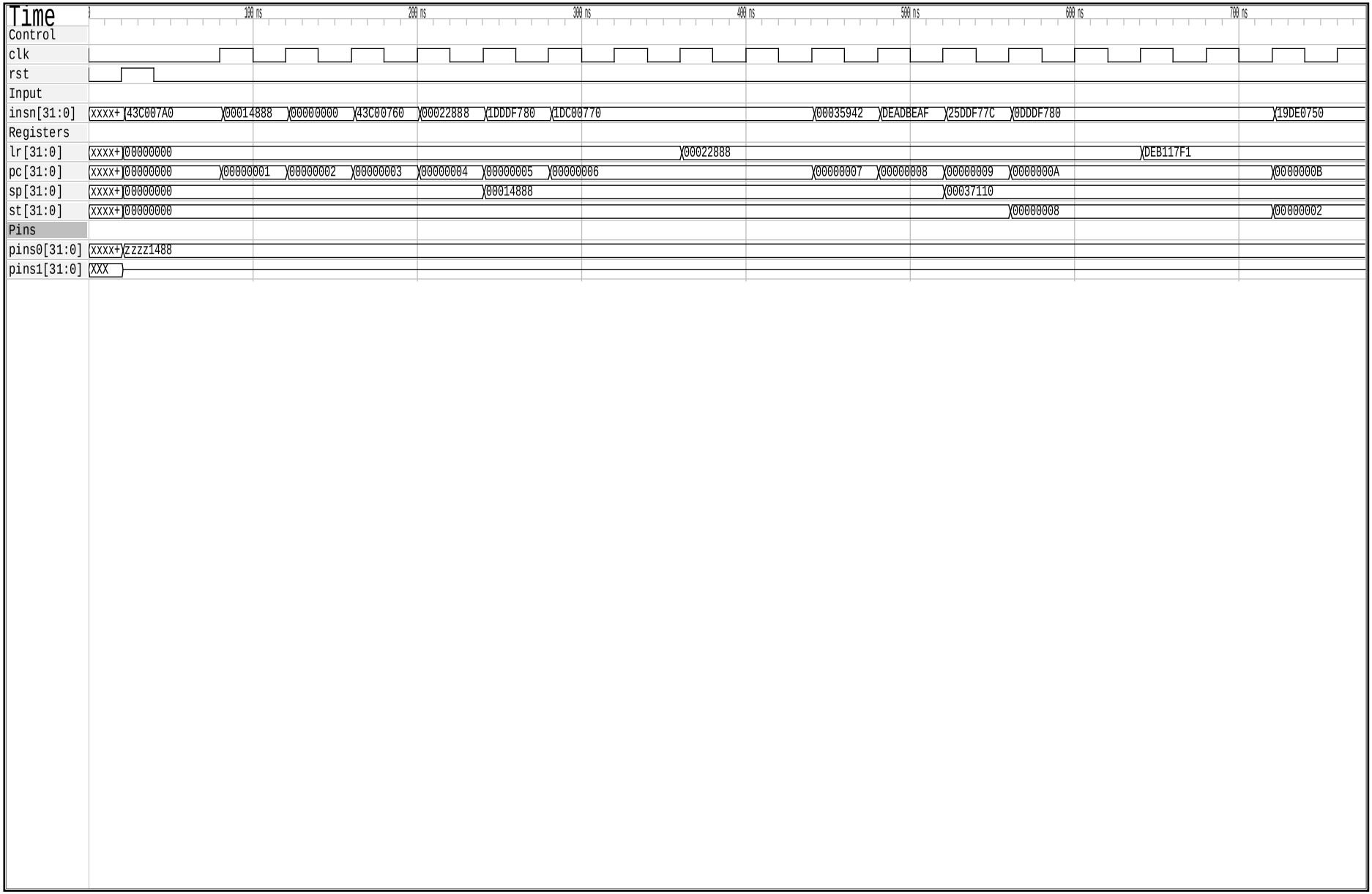
1. Проинициализировать регистры 29 и 30 значениями 14888h и 22888h
2. Суммировать эти регистры в регистр 30
3. Суммировать 35942h и DEADBEAFh
4. Перемножить регистры 29 и 30 в них же
5. ИСКЛ. ИЛИ этих регистров с сохранением в тридцатый
6. Циклический сдвиг содержимого 30-го регистра на 11 бит в 29-й
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

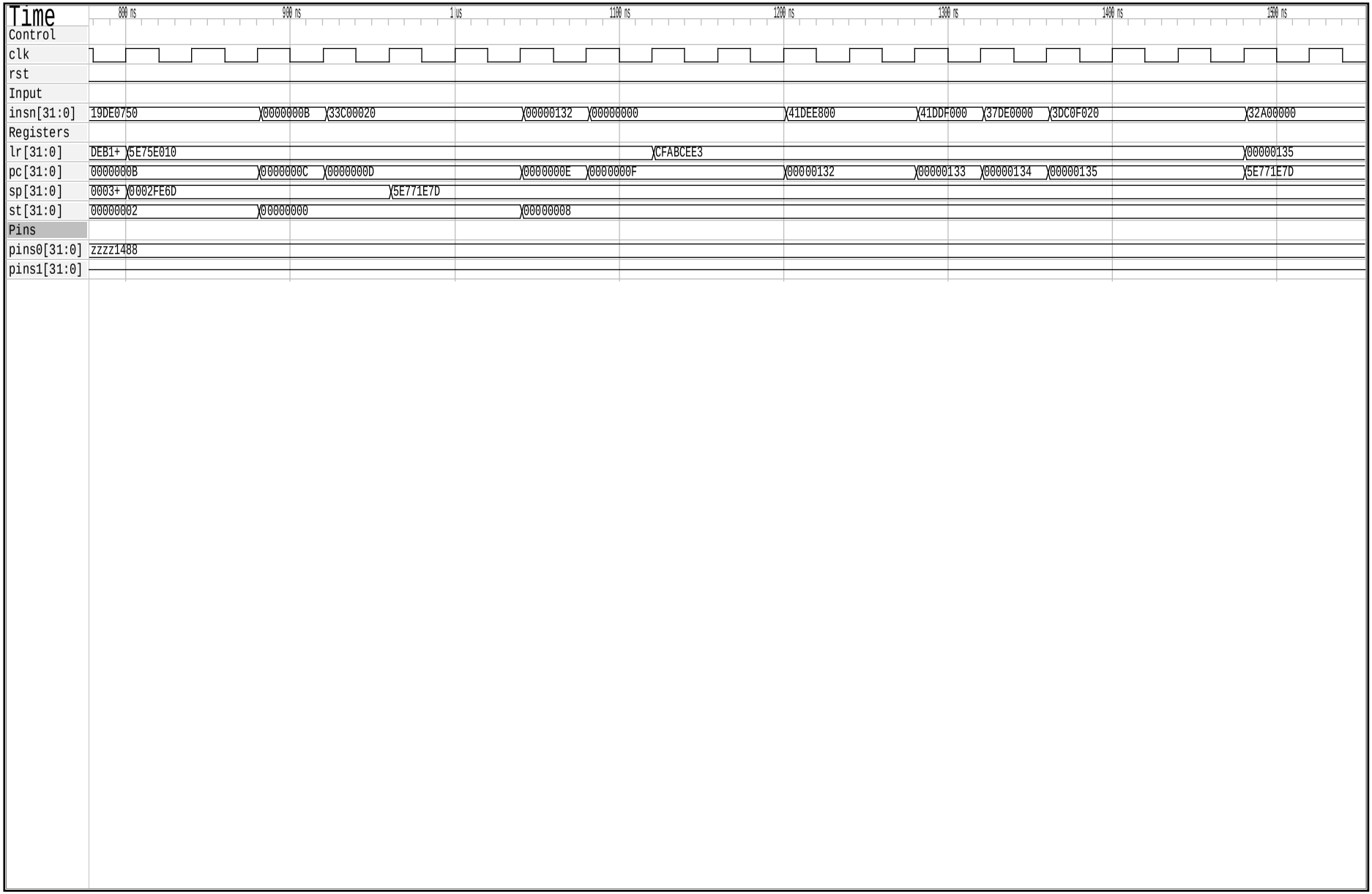
1.2.2 Исходный код

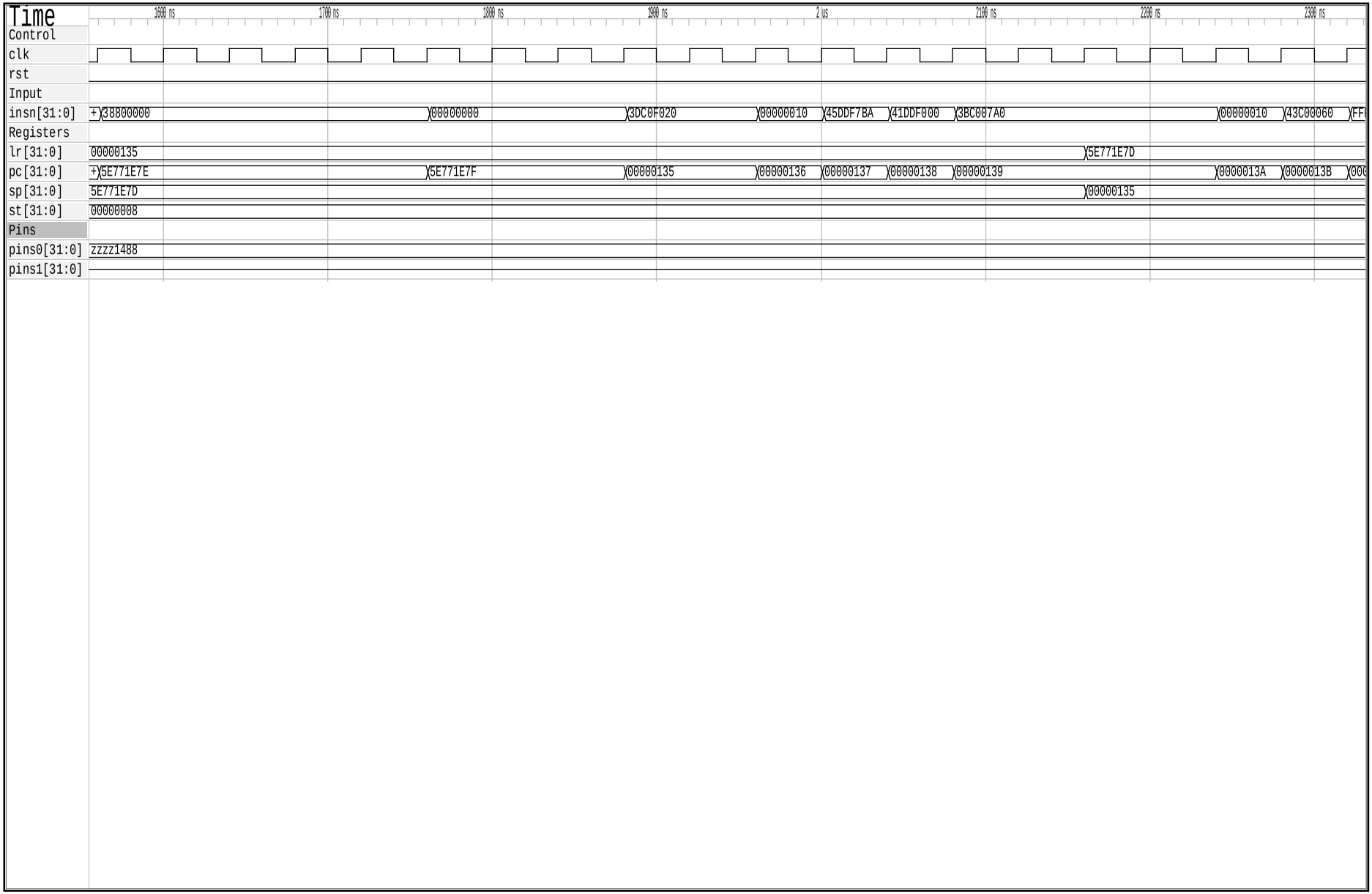
```
[0x00000000]:
movs 0x14888 -> r30
movs 0x22888 -> r29
add r29, r30 -> r30
add 0x35942, 0xDEADBEAF -> r29
mul r29, r30 -> r29, r30
xor r29, r30 -> r30
csr r30, 0x0B -> r29
br  0x132
(nop)
[0x00000132]:
out r29 -> [r30]
out r30 -> [r29]
brl r30
str r30 -> 0x10
mov r29, r30 -> r30, r29
out r30 -> [r29]
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)
```

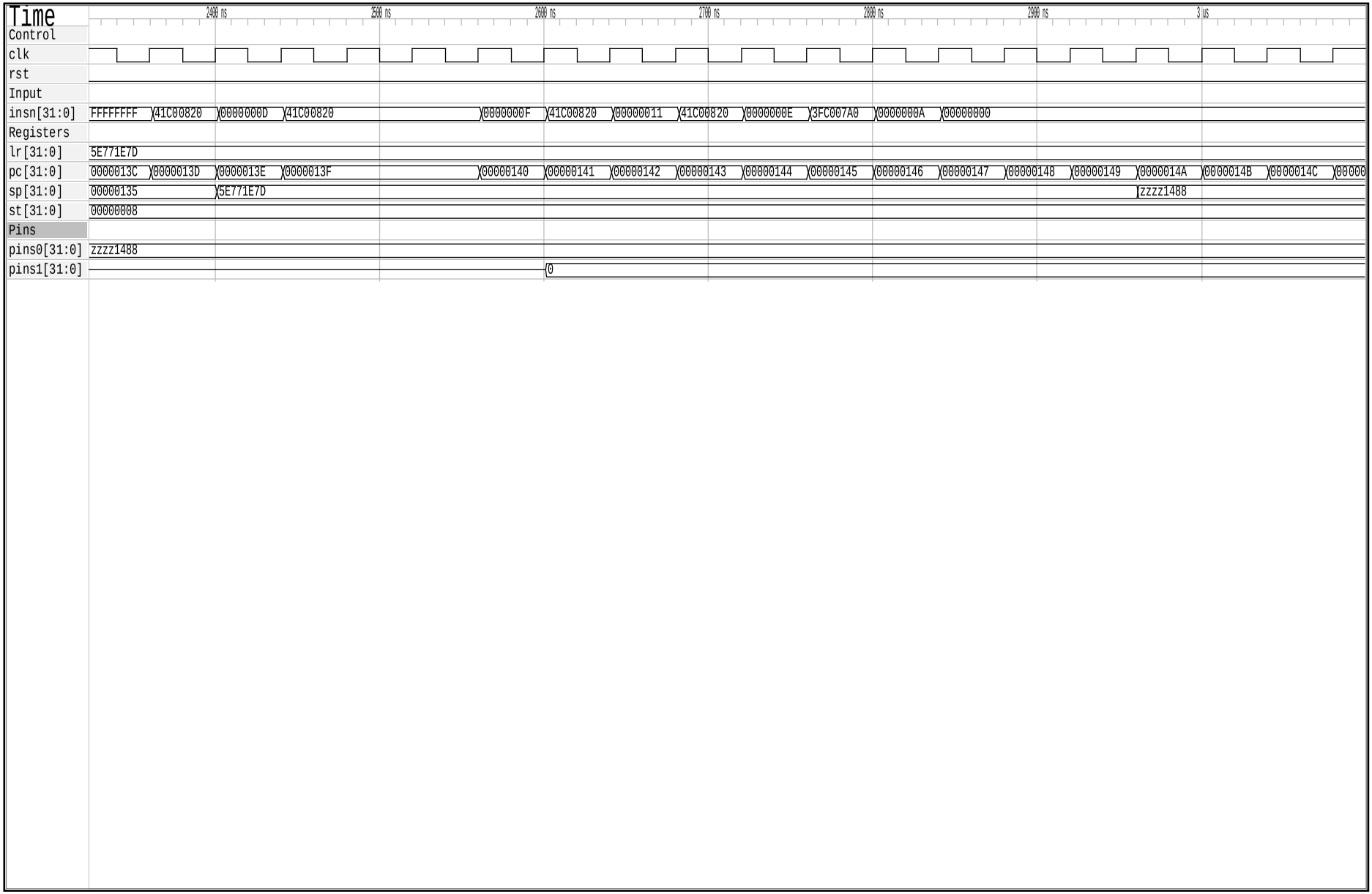
1.2.3 Временные диаграммы

На изображениях - результат выполнения тестовой программы (полностью)


Time
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]
00000000
00022888
DEB117F1
pc[31:0]
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
sp[31:0]
00014888
00037110
st[31:0]
00000008
00000002
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]
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
1600 ns
1700 ns
1800 ns
1900 ns
2 us
2100 ns
2200 ns
2300 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]
00000008
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

1.3 Программа «Фибоначчи»

1.3.1 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

1.3.2 Исходный код

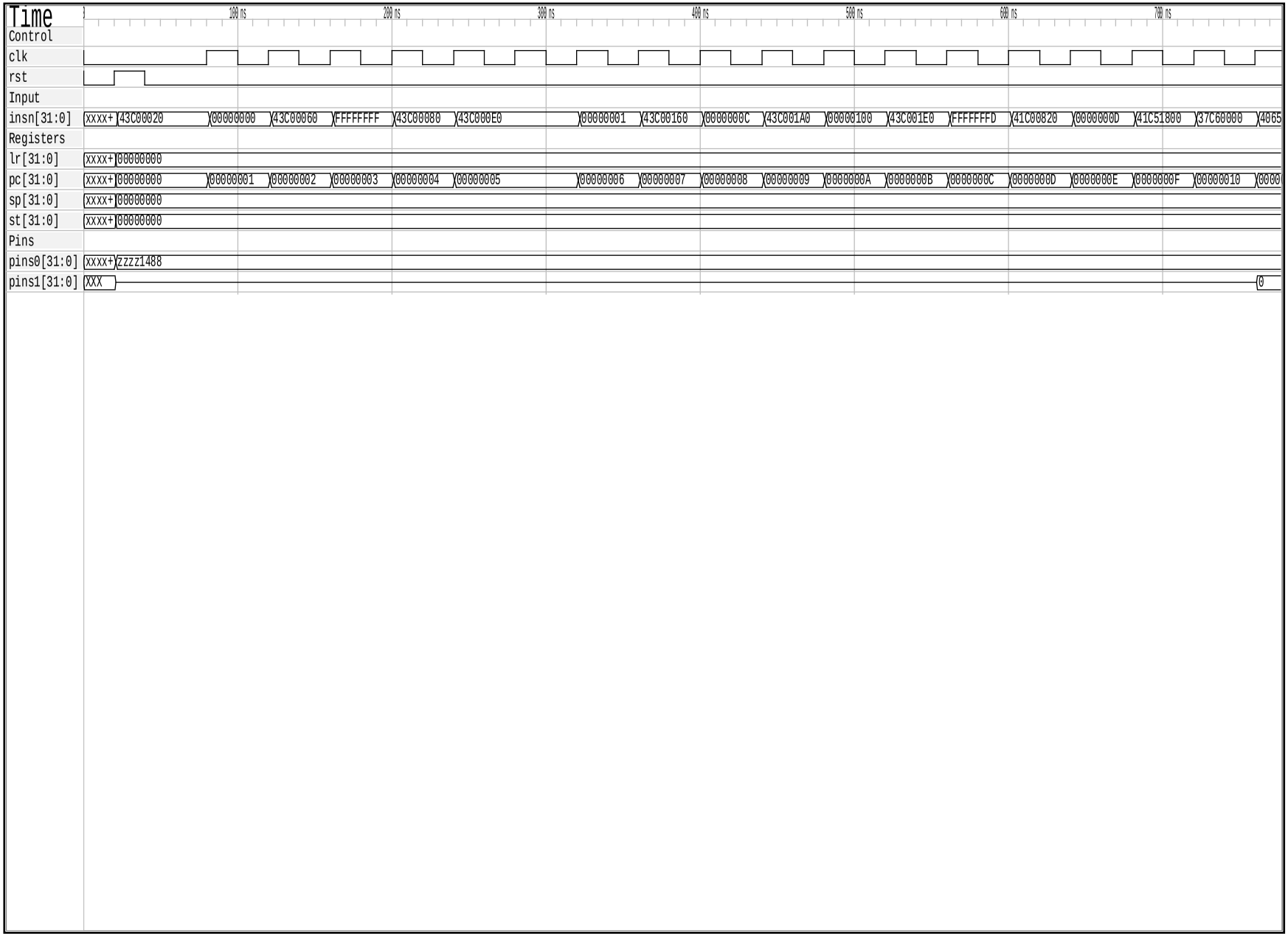
```
[0x00000000]:
movs 0x00 -> r0
movs 0xFFFFFFFF -> r1
movs r0 -> r2  //F0
movs 0x01 -> r3  //F1
movs 0x0C -> r5  //[gpio1.val]
movs 0x100 -> r6  //[fib()]
movs 0xFFFFFFFD -> r7  //-0x03
out r1 -> 0x0D
out r3 -> [r5]
brl [r6]
out_lo r4 -> [r5]
rbr_lo r7
br  r0
(nop)
[0x00000100]:  //r4 fib(&r2, &r3)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

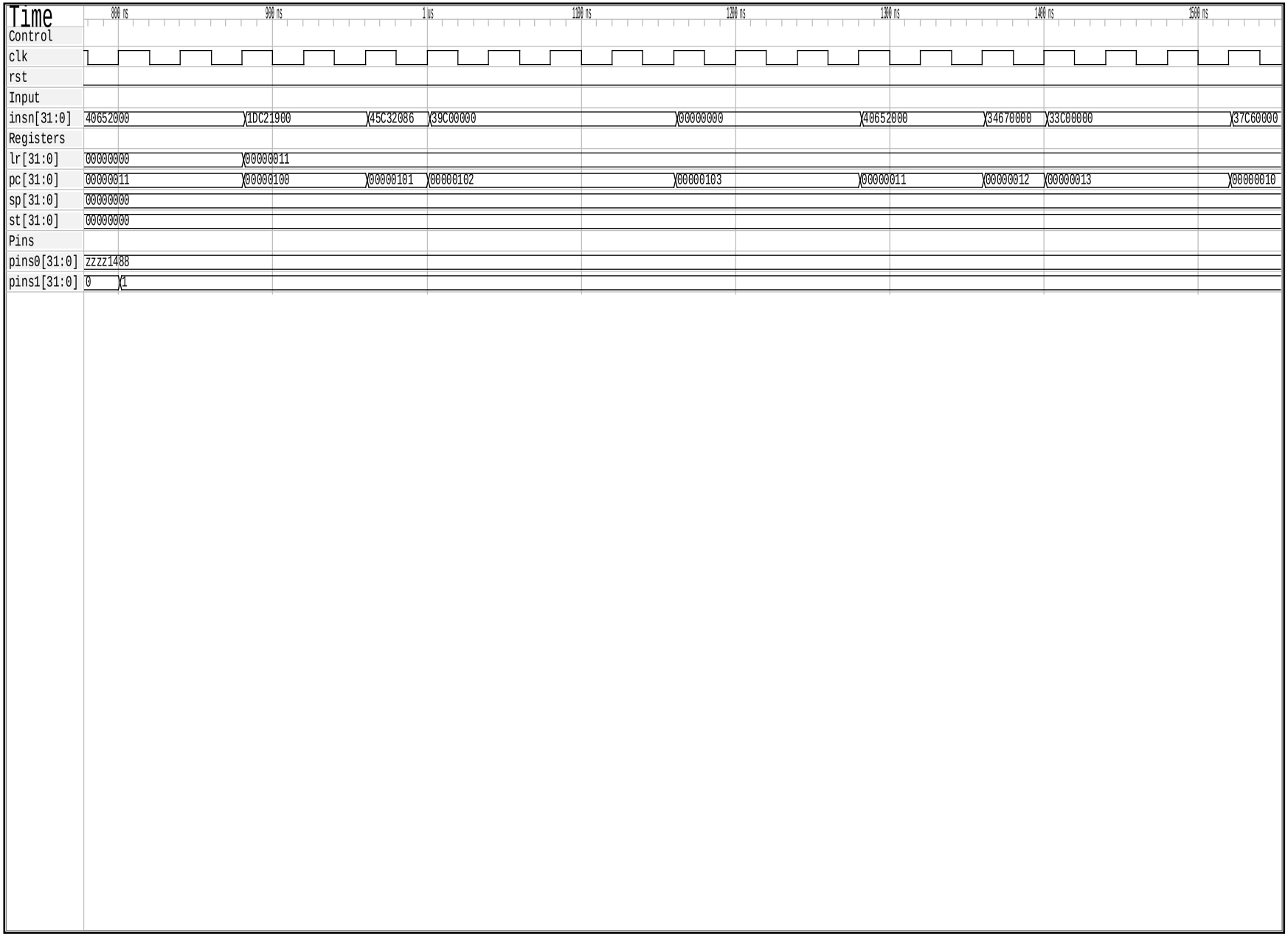
1.3.3 Временные диаграммы

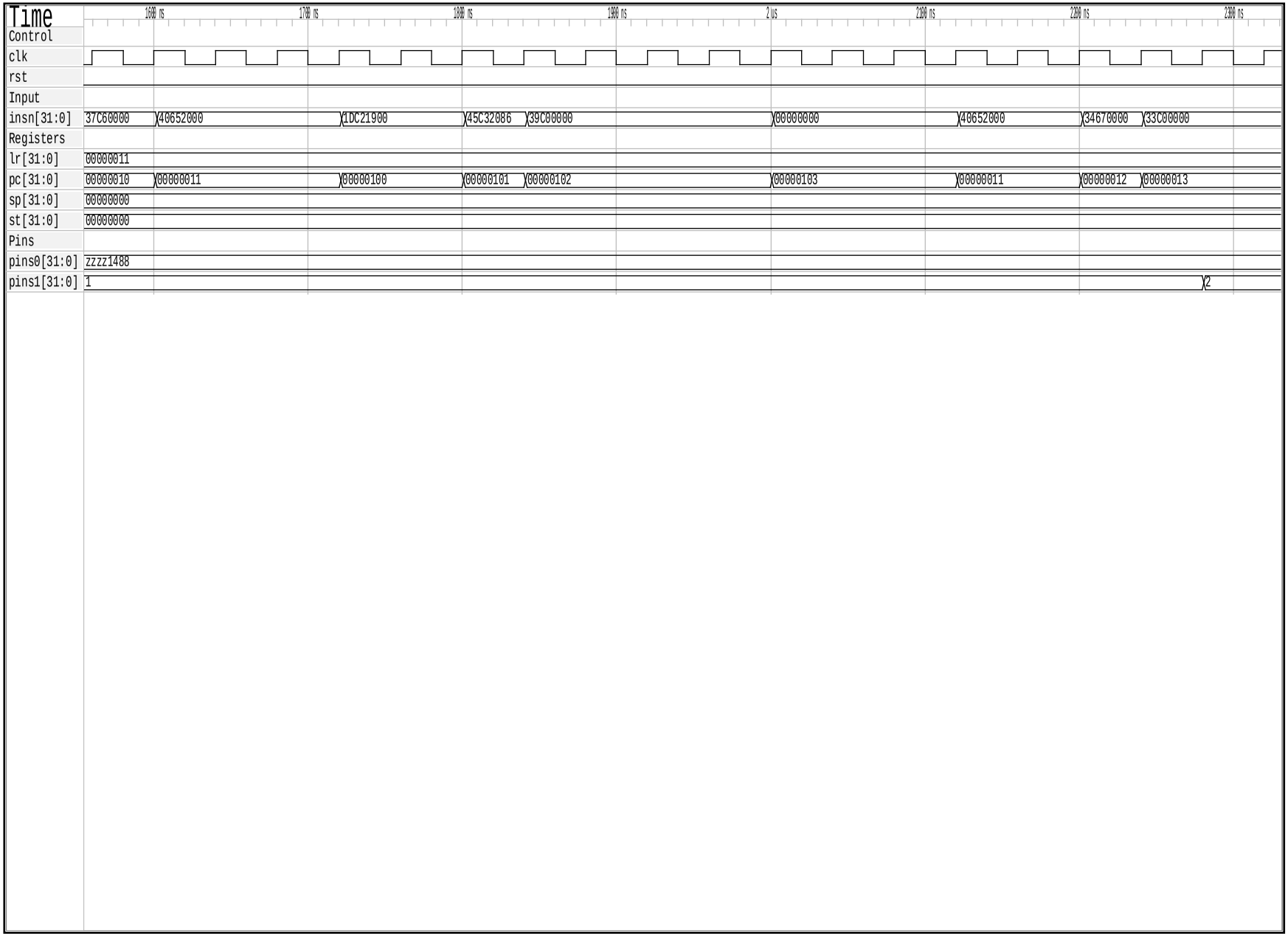
На изображениях:

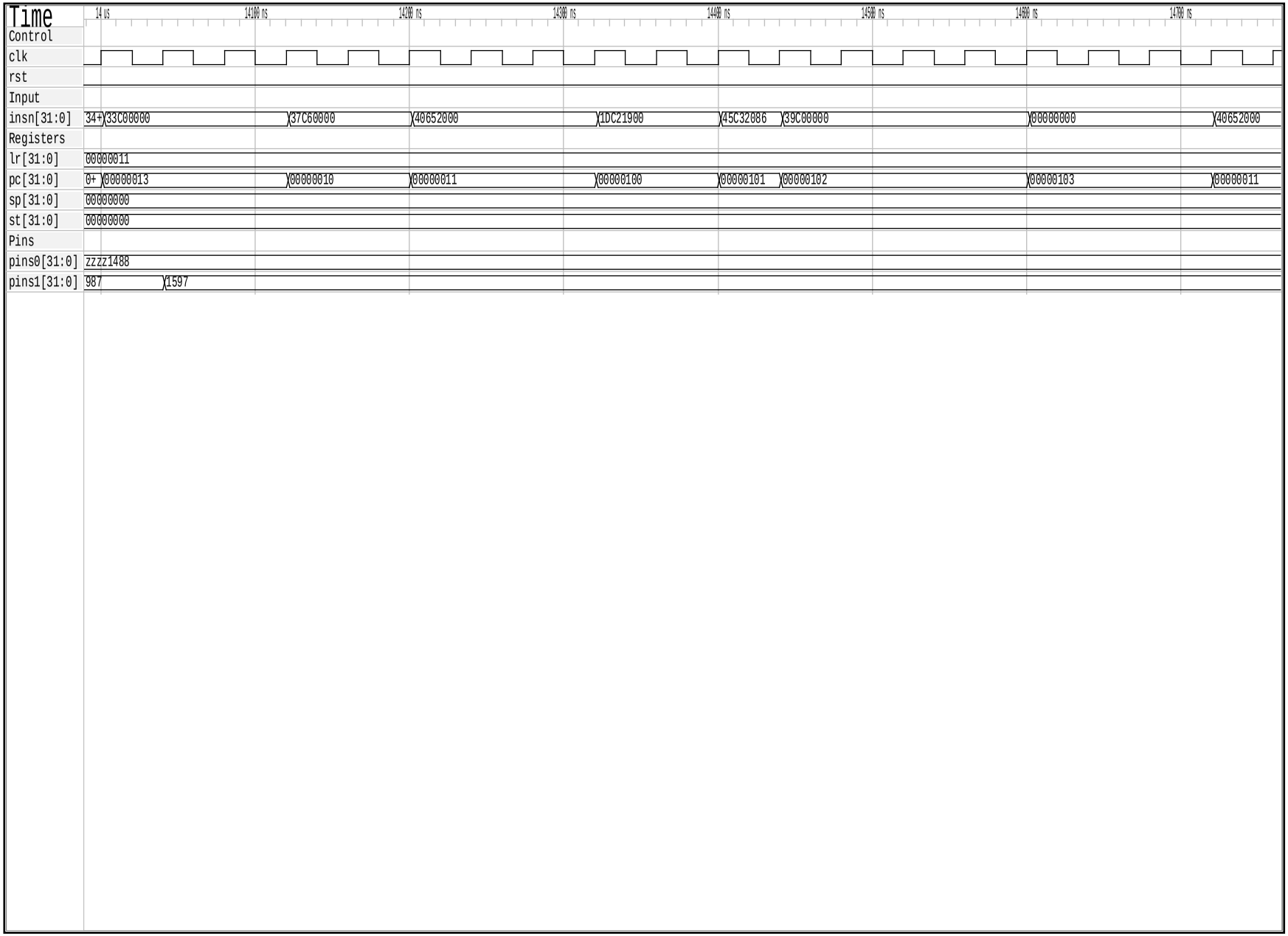
1. Инициализация
2. Первое число Фибоначчи (1)
3. Второе и третье числа Фибоначчи (1 и 2)
4. Семнадцатое число Фибоначчи (1597)

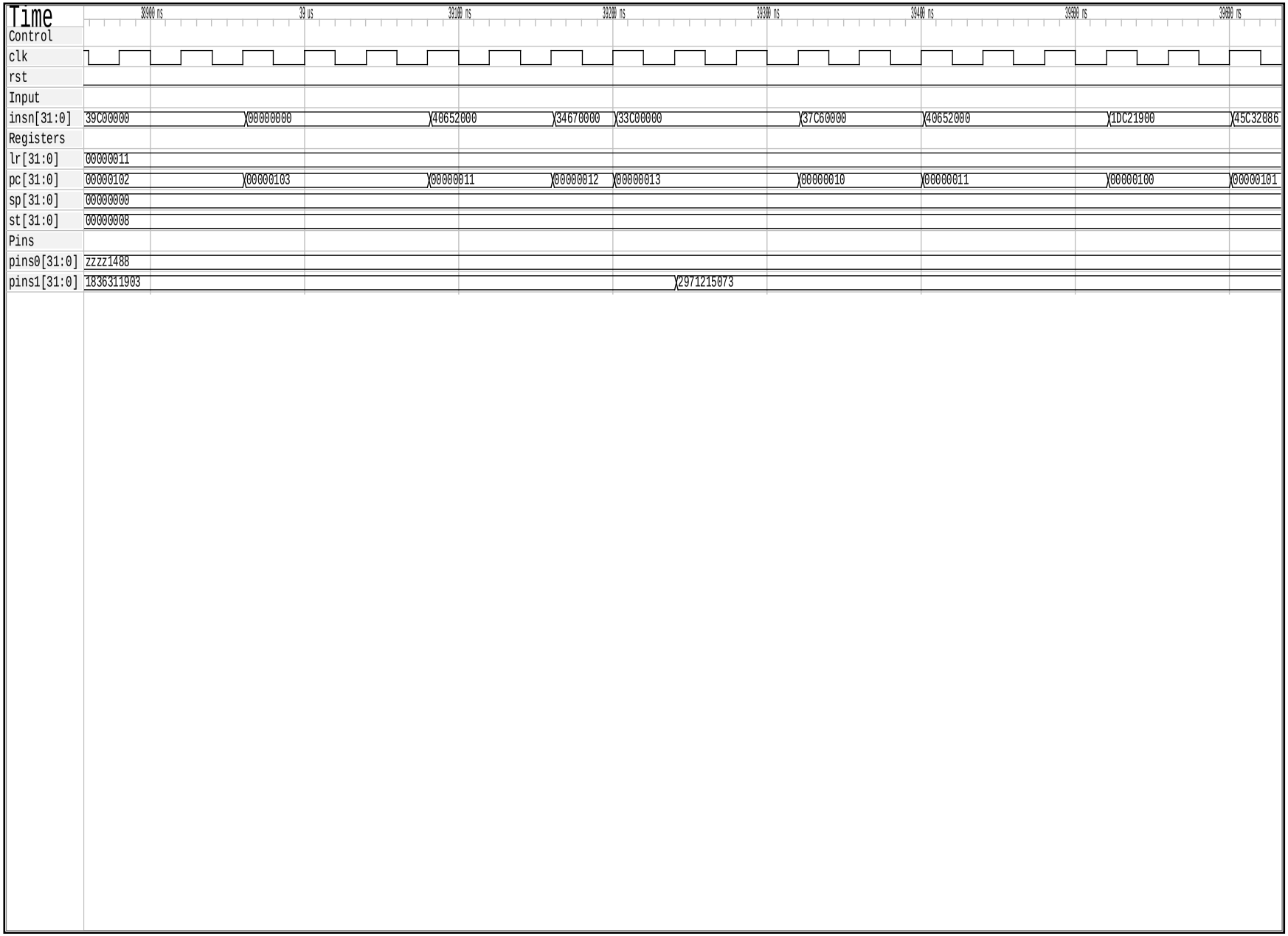
5. 47-е число Фибоначчи (2971215073)
6. Перезагрузка и переинициализация после 47-го числа
7. Первые 29 чисел Фибоначчи (обзорно)
8. 30-44 числа Фибоначчи (обзорно)
9. Перзагрузка и счёт сначала (обзорно)

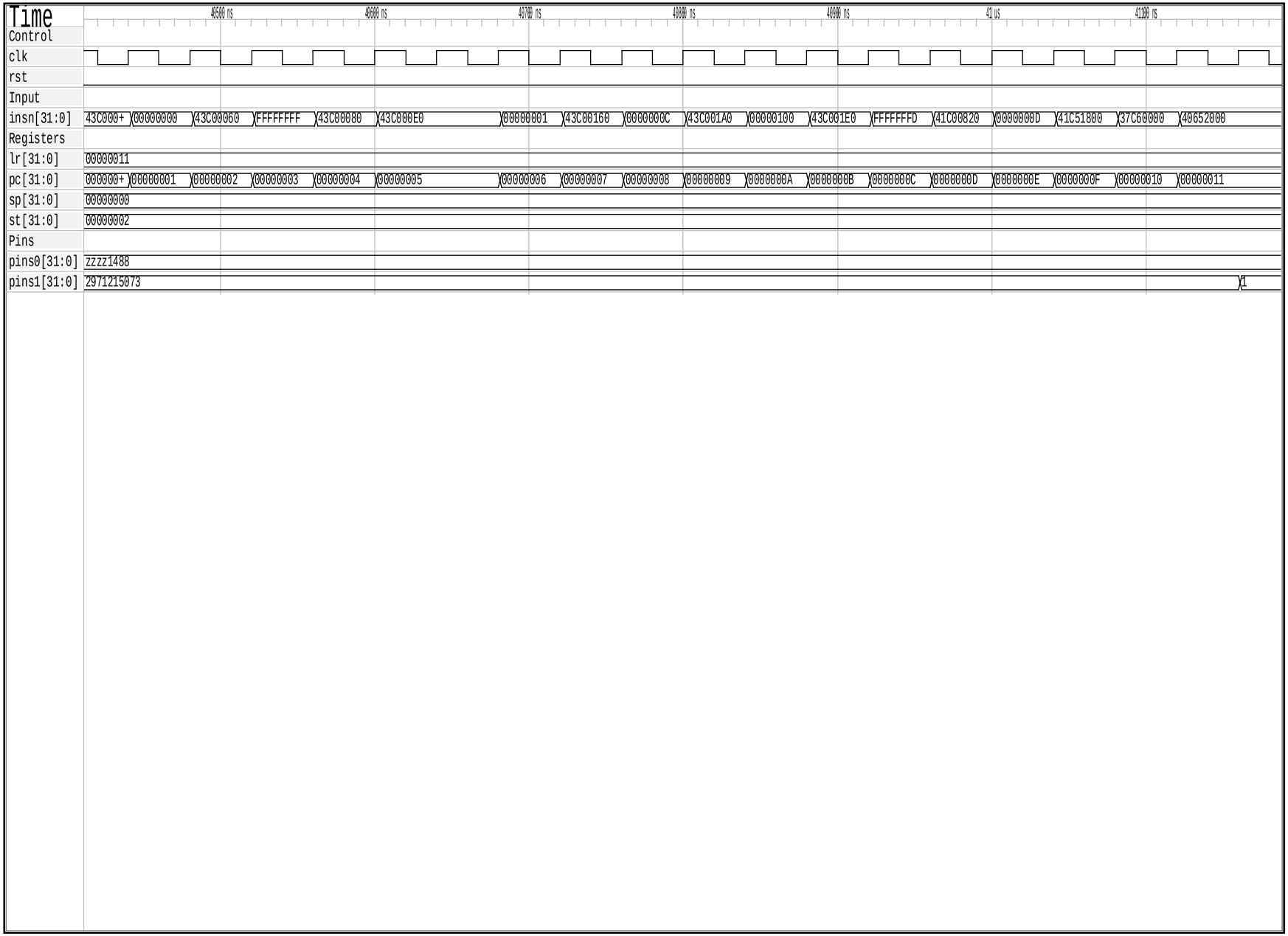

Time
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
Control
clk
rst
Input
insn[31:0]
xxxx+
43C00020
00000000
43C00060
FFFFFFFF
43C00080
43C000E0
00000001
43C00160
0000000C
43C001A0
00000100
43C001E0
FFFFFFFD
41C00820
0000000D
41C51800
37C60000
4065
Registers
lr[31:0]
xxxx+
00000000
pc[31:0]
xxxx+
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000010
0000
sp[31:0]
xxxx+
00000000
st[31:0]
xxxx+
00000000
Pins
pins0[31:0]
xxxx+
zzzz1488
pins1[31:0]
XXX
0

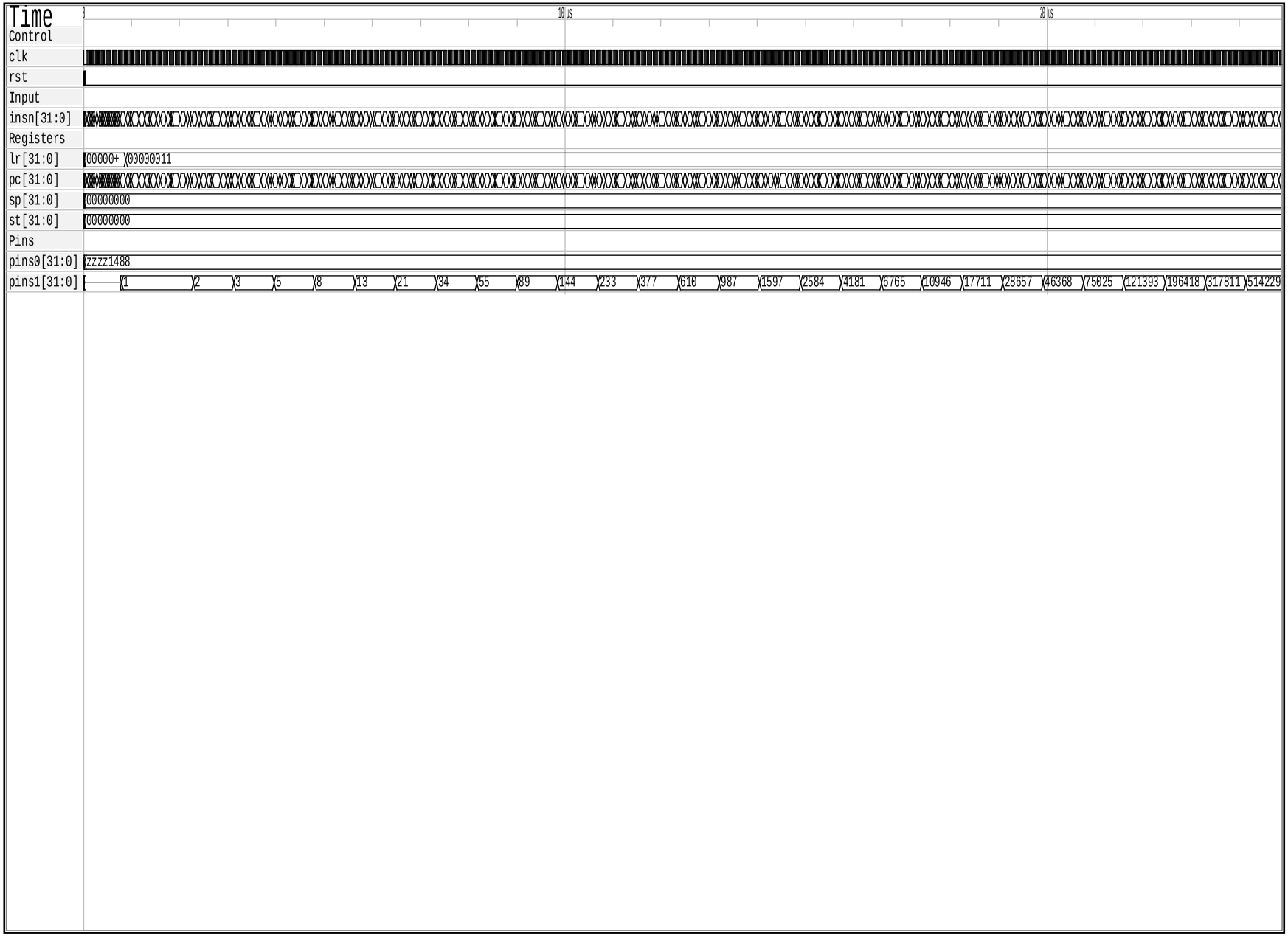

Time
800 ns
900 ns
1 us
1100 ns
1200 ns
1300 ns
1400 ns
1500 ns
Control
clk
rst
Input
insn[31:0]
40652000
1DC21900
45C32086
39C00000
00000000
40652000
34670000
33C00000
37C60000
Registers
lr[31:0]
00000000
00000011
pc[31:0]
00000011
00000100
00000101
00000102
00000103
00000011
00000012
00000013
00000010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

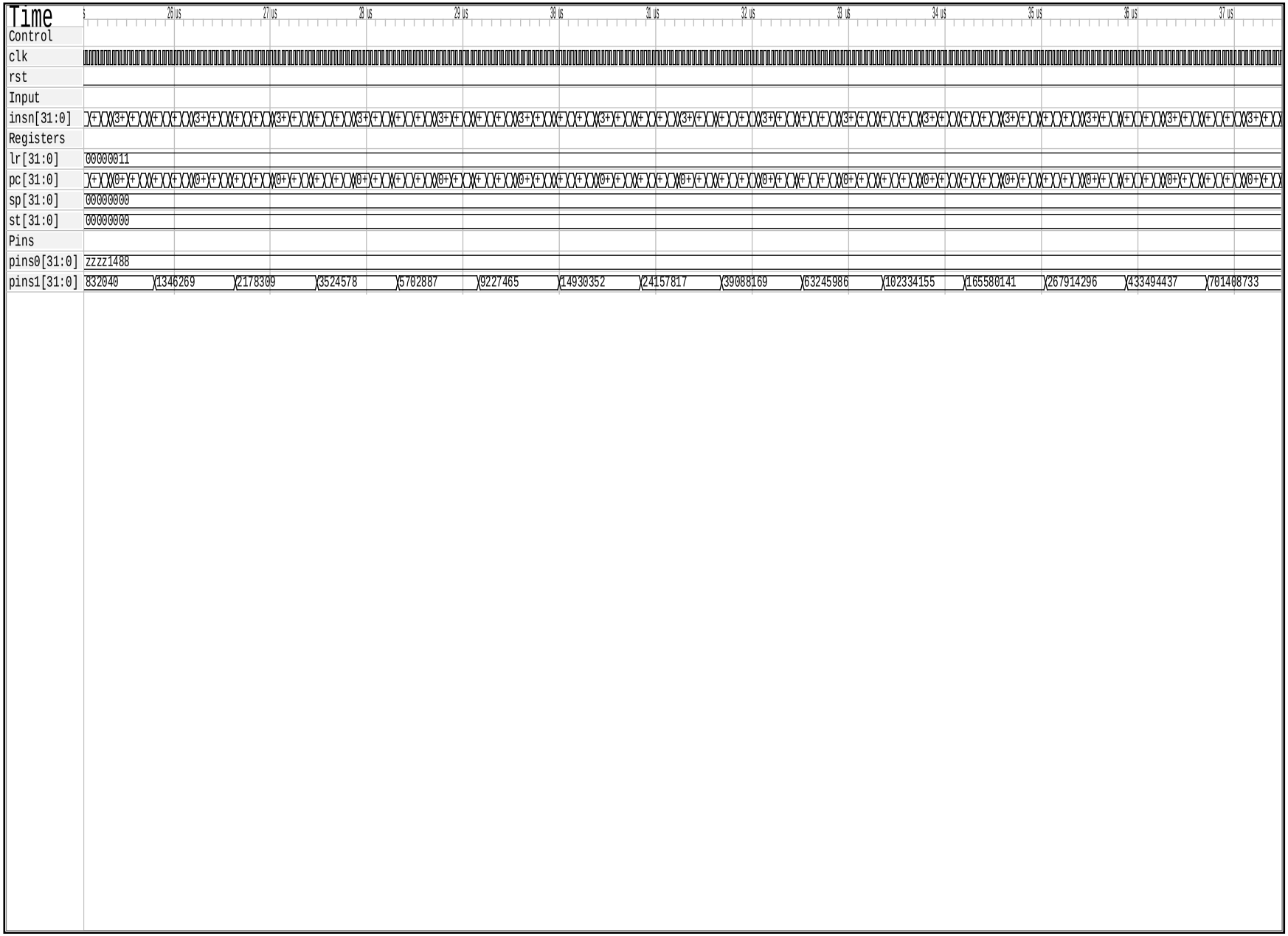

Time
1600 ns
1700 ns
1800 ns
1900 ns
2 us
2100 ns
2200 ns
2300 ns
Control
clk
rst
Input
insn[31:0]
37C60000
40652000
1DC21900
45C32086
39C00000
00000000
40652000
34670000
33C00000
Registers
lr[31:0]
00000011
pc[31:0]
00000010
00000011
00000100
00000101
00000102
00000103
00000011
00000012
00000013
sp[31:0]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

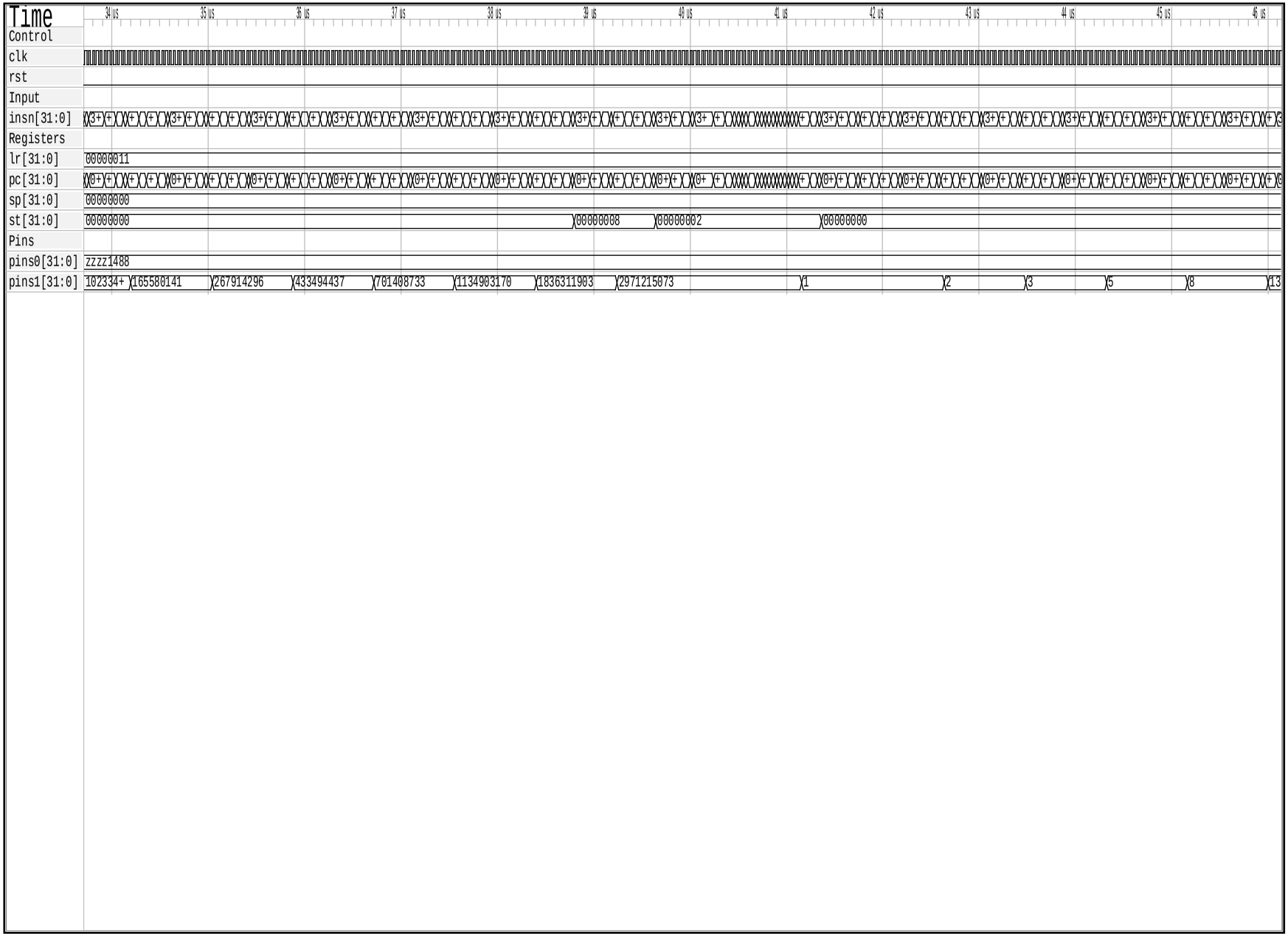

Time
14 us
14100 ns
14200 ns
14300 ns
14400 ns
14500 ns
14600 ns
14700 ns
Control
clk
rst
Input
insn[31:0]
34+
33C00000
37C60000
40652000
1DC21900
45C32086
39C00000
00000000
40652000
Registers
lr[31:0]
00000011
pc[31:0]
0+
00000013
00000010
00000011
00000100
00000101
00000102
00000103
00000011
sp[31:0]
00000000
st[31:0]
00000000
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
987
1597


Time
38900 ns
39 us
39100 ns
39200 ns
39300 ns
39400 ns
39500 ns
39600 ns
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
46500 ns
46600 ns
46700 ns
46800 ns
46900 ns
47 us
47100 ns
Control
clk
rst
Input
insn[31:0]
43C000+
00000000
43C00060
FFFFFFFF
43C00080
43C000E0
00000001
43C00160
0000000C
43C001A0
00000100
43C001E0
FFFFFFFD
41C00820
0000000D
41C51800
37C60000
40652000
Registers
lr[31:0]
00000011
pc[31:0]
000000+
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F
00000010
00000011
sp[31:0]
00000000
st[31:0]
00000002
Pins
pins0[31:0]
zzzz1488
pins1[31:0]
2971215073
1


Time
10 us
20 us
Control
clk
rst
Input
insn[31:0]
Registers
lr[31:0]
00000+
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
26 us
27 us
28 us
29 us
30 us
31 us
32 us
33 us
34 us
35 us
36 us
37 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]
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733


Time
34 us
35 us
36 us
37 us
38 us
39 us
40 us
41 us
42 us
43 us
44 us
45 us
46 us
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

2 Синтез

2.1 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 за собой резкое повышение количества использованных триггеров.

3. ASIC (заказная схема) на основе библиотеки OSU 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, а именно 10M25SAE144C8G.

2.2 Результаты синтезирования

2.2.1 YOSYS - Xilinx

Результаты синтезирования набором YOSYS на цель Xilinx 7-Series представлены в таблице 1

Таблица 1: Результаты синтеза на цель Xilinx

№	Название блока	Блоков	Шин (бит в шинах)	Ячеек
1	addsub_32	1	36 (160)	63
2	alu32_2x2	1	143 (635)	385
3	bitwise_32	1	33 (128)	61
4	bshift_32	1	78 (330)	104
5	cla_4	8	13 (30)	13
6	cla_16	2	13 (66)	13
7	cla_32	1	8 (103)	4
8	cond_calc	1	12 (15)	7
9	drev_32	2	3 (65)	32
10	emb_ram	1	**1131 (1348)**	**1247**
11	execute	1	32 (324)	137
12	execute_stage_passthrough	1	16 (174)	86
13	fa	960	5 (5)	2
14	fa_pg	32	6 (6)	3
15	fmask_32	1	2 (37)	31
16	gpio	4	74 (477)	249
17	gpio_mux	4	146 (893)	281
18	ha	32	4 (4)	2
19	insn_decoder	1	25 (281)	0
20	memory_op	1	1808 (2504)	2301
21	memory_op_stage_passthrough	1	10 (32)	15
22	mul_32	1	69 (256)	65
23	mult_32	1	3011 (3136)	2016
24	ovf_32	1	29 (121)	25
25	pipeline_interface	1	47 (509)	337
26	reg32_2x2_pc	1	**2861 (5173)**	**4920**

27	reg_hazard_checker	1	57 (97)	39
28	register_wb	1	152 (367)	286
29	right_rot_32	1	67 (133)	96
30	status_register_adaptor	1	7 (38)	0
31	tblock_32	1	6 (99)	32
32	test_periph_assembly	1	45 (1412)	8
33	test_pipeline_assembly	1	143 (2126)	11
34	test_processor_assembly	1	32 (841)	3
35	zmask_32	1	3 (65)	32
ВСЕГО			**15999 (31546)**	**15635**

2.2.2 YOSYS - iCE40

Результаты синтезирования набором YOSYS на цель iCE40 представлены в таблице 2

Таблица 2: Результаты синтеза на цель iCE40

№	Название блока	Блоков	Шин (бит в шинах)	Ячеек
1	addsub_32	1	18 (142)	45
2	alu32_2x2	1	280 (643)	393
3	bitwise_32	1	151 (246)	179
4	bshift_32	1	36 (288)	62
5	cla_4	8	12 (29)	12
6	cla_16	2	12 (65)	12
7	cla_32	1	9 (104)	5
8	cond_calc	1	18 (21)	13
9	drev_32	2	3 (65)	32
10	emb_ram	1	**30635 (62596)**	**62493**
11	execute	1	33 (325)	138
12	execute_stage_passthrough	1	16 (174)	86
13	fa	960	5 (5)	2
14	fa_pg	32	6 (6)	3
15	fmask_32	1	8 (43)	37
16	gpio	4	38 (379)	151
17	gpio_mux	4	175 (859)	247

18	ha	32	4 (4)	2
19	insn_decoder	1	25 (281)	0
20	memory_op	1	602 (1298)	1095
21	memory_op_stage_passthrough	1	10 (32)	15
22	mul_32	1	27 (214)	23
23	mult_32	1	3011 (3136)	2016
24	ovf_32	1	45 (137)	41
25	pipeline_interface	1	47 (509)	337
26	reg32_2x2_pc	1	**2653 (4965)**	**4711**
27	reg_hazard_checker	1	71 (111)	53
28	register_wb	1	41 (225)	144
29	right_rot_32	1	131 (197)	160
30	status_register_adaptor	1	7 (38)	0
31	tblock_32	1	6 (99)	32
32	test_periph_assembly	1	45 (1412)	8
33	test_pipeline_assembly	1	143 (2126)	11
34	test_processor_assembly	1	32 (841)	3
35	zmask_32	1	3 (65)	32
ВСЕГО			**44201 (90832)**	**74918**

2.2.3 YOSYS - ASIC OSU TSMC 25nm

Результаты синтезирования набором YOSYS на цель ASIC с библиотекой элементов OSU TSMC 25nm представлены в таблице 3

Таблица 3: Результаты синтеза на цель ASIC osu025_stdcells

№	Название блока	Блоков	Шин (бит в шинах)	Ячеек
1	addsub_32	1	41 (165)	68
2	alu32_2x2	1	387 (879)	629
3	bitwise_32	1	458 (553)	486
4	bshift_32	1	50 (302)	76
5	cla_4	8	23 (40)	23
6	cla_16	2	23 (76)	23
7	cla_32	1	12 (107)	8

8	cond_calc	1	51 (54)	46
9	drev_32	2	67 (129)	96
10	emb_ram	1	**120377 (152307)**	**152208**
11	execute	1	204 (496)	309
12	execute_stage_passthrough	1	102 (260)	172
13	fa	960	12 (12)	9
14	fa_pg	32	12 (12)	9
15	fmask_32	1	29 (64)	58
16	gpio	4	626 (1029)	801
17	gpio_mux	4	746 (1493)	881
18	ha	32	4 (4)	2
19	insn_decoder	1	25 (281)	0
20	memory_op	1	1942 (2638)	2435
21	memory_op_stage_passthrough	1	25 (47)	30
22	mul_32	1	66 (253)	62
23	mult_32	1	3011 (3136)	2016
24	ovf_32	1	110 (202)	106
25	pipeline_interface	1	216 (678)	506
26	reg32_2x2_pc	1	**5568 (7849)**	**7597**
27	reg_hazard_checker	1	216 (256)	198
28	register_wb	1	302 (517)	436
29	right_rot_32	1	328 (394)	357
30	status_register_adaptor	1	7 (38)	0
31	tblock_32	1	40 (133)	66
32	test_periph_assembly	1	45 (1412)	8
33	test_pipeline_assembly	1	146 (2129)	14
34	test_processor_assembly	1	32 (841)	3
35	zmask_32	1	65 (127)	94
ВСЕГО			**151739 (198968)**	**183060**

2.2.4 Quartus Prime - Altera MAX10

При синтезировании проекта для временного анализа с помощью САПР Quartus Prime на цель Altera MAX10 были получены следующие результаты:

1. Всего логических элементов - **8694:**
    (a) Чисто комбинационных - **5862**
    (b) Чисто регистровых - **662**
    (c) Комбинированных - **2170**
2. Всего регистров (после стадии размещения и трассировки) - **2832**
3. Всего межсоединений (после стадии размещения и трассировки) - **35317**
4. Параметы подстановки ОЗУ:
    (a) Портов - **2**
    (b) Ширина портов, бит - **32**
    (c) Глубина памяти, слов - **1025**
    (d) Объем памяти, бит - **32800**

2.3 Результаты временного анализа

Временной анализ был проведён средствами САПР Quartus Prime, а именно внутренним анализатором TimeQuest над предварительно синтезированным и размещённым в той же САПР проектом. Оценка производилась путём определения максимального слека от входа clk схемы test_processor_assembly до выходных портов всех подключенных к нему модулей на основе двух внутренних моделей распространения сигнала в данной ПЛИС. Модели отличаются лишь температурой окружающей среды. Таким образом были получены следующие результаты:

1. Модель Slow 1200 mV 85 C:
    (a) Setup: 1.263 нс
    (b) Hold: 0.361 нс
    (c) Минимальная ширина импульса: 5.613 нс
    (d) Максимальная частота clk: 93.14 МГц
2. Модель Slow 1200 mV 0 C:
    (a) Setup: 1.825 нс

(b) Hold: 0.323 нс

(c) Минимальная ширина импульса: 5.555 нс

(d) Максимальная частота clk: 98.28 МГц

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

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

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

[3] 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.

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

Приложение 1. Instruction Set Architecture

1 Введение

1.1 Общее описание

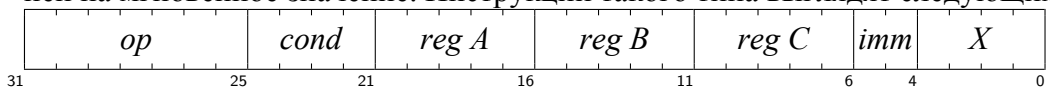
Процессор УП-1 обладает монолитной 32-битной архитектурой с типом доступа к памяти/периферии load-store, совмещённым доступом к памяти/периферии, раздельным доступом к регистрам и четырёхстадийным конвейером, что означает следующее:

- 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.

1.2 Формат инструкции

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

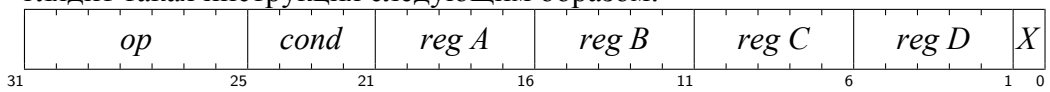
1. Инструкция с тремя и менее операндами. Такая инструкция может иметь до двух входных операндов и до одного выходного. Любой из входных операндов может быть заменён на мгновенное значение. Инструкции такого типа выглядят следующим образом:



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.

1.3 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 - $\overline{\mathrm{C}}$

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.

1.4 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.

1.5 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 4

Таблица 4: 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
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

Набор инструкций - продолжение

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

2 Описание

2.1 NOP

Пустая операция

Рис. 3: Машинное представление инструкции NOP

2.1.1 Описание

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

2.1.2 Флаги, затрагиваемые данной инструкцией:

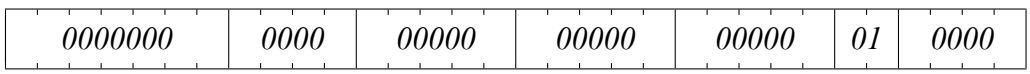
N	Z	C	V
-	-	-	-

2.1.3 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.

2.1.4 Пример использования:

```
NOP  //пропустить 1 такт
```

1. 

2.2 OR

Побитовое ИЛИ

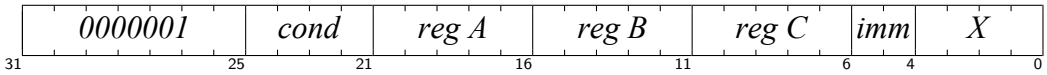


Рис. 4: Машинное представление инструкции OR

2.2.1 Описание

Производит побитовое ИЛИ двух операндов и сохраняет результат в третий

2.2.2 Флаги, затрагиваемые данной инструкцией:

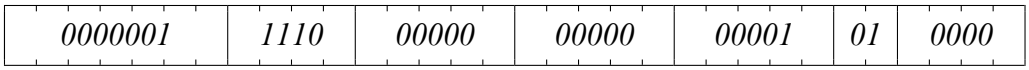
N	Z	C	V
0	+	0	0

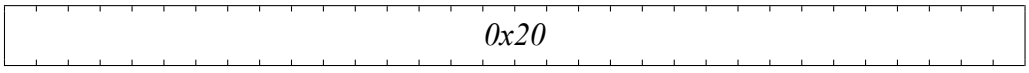
2.2.3 Свойства инструкции:

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

2.2.4 Пример использования:

```
OR r0, 0x20 -> r1  // r0 ИЛИ 32 и сохранить а r1
```

1. 

2. 

2.3 NOR

Побитовое ИЛИ-НЕ

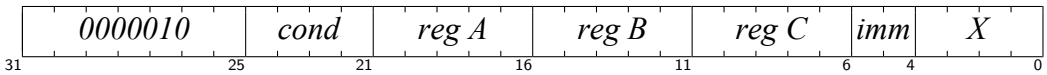


Рис. 5: Машинное представление инструкции NOR

2.3.1 Описание

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

2.3.2 Флаги, затрагиваемые данной инструкцией:

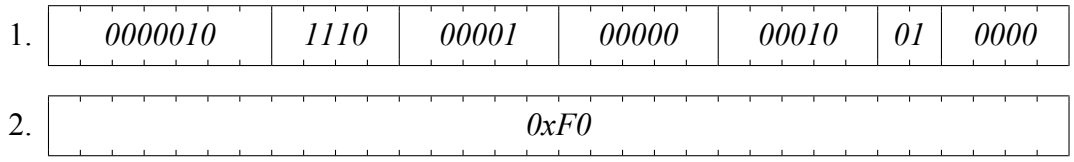
N	Z	C	V
0	+	0	0

2.3.3 Свойства инструкции:

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

2.3.4 Пример использования:

```
NOR r1, 0xF0 -> r2  // r1 ИЛИ-НЕ 240 и сохранить в r2
```



2.4 AND

Побитовое И

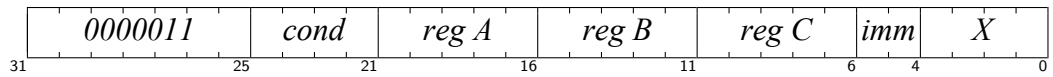


Рис. 6: Машинное представление инструкции AND

2.4.1 Описание

Производит побитовое И двух операндов и сохраняет результат в третий

2.4.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	0	0

2.4.3 Свойства инструкции:

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

2.4.4 Пример использования:

```
AND r0, 0xFF -> r1  // r0 И 255 и сохранить а r1
```

1. *0000011* | *1110* | *00000* | *00000* | *00001* | *01* | *0000*
2. *0xFF*

2.5 NAND

Побитовое И-НЕ

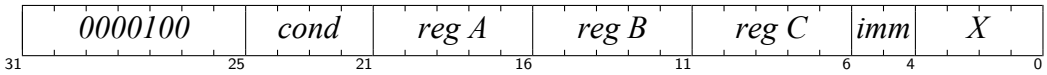


Рис. 7: Машинное представление инструкции NAND

2.5.1 Описание

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

2.5.2 Флаги, затрагиваемые данной инструкцией:

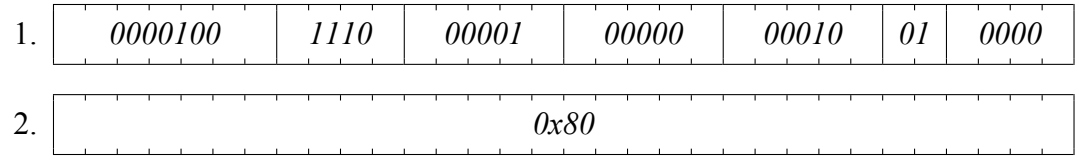
N	Z	C	V
0	+	0	0

2.5.3 Свойства инструкции:

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

2.5.4 Пример использования:

```
NAND г1, 0x80 -> г2  // г1 И-НЕ 128 и сохранить в г2
```



2.6 INV

Побитовая инверсия

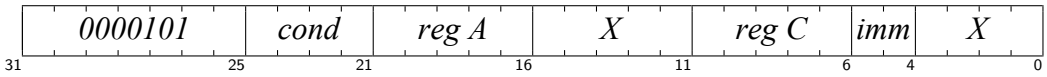


Рис. 8: Машинное представление инструкции INV

2.6.1 Описание

Инвертирует содержимое операнда и сохраняет результат во второй.

2.6.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	0	0

2.6.3 Свойства инструкции:

- Исполнение занимает 1 такт.
- Может исполняться условно.

- Может использовать одно мгновенное значение
- Производит запись в регистр.
- Не меняет потока исполнения.

2.6.4 Пример использования:

```
INV r1 -> r2  //  инвертировать r1 и сохранить в r2
```

1. | *0000101* | *1110* | *00001* | *00000* | *00010* | *00* | *0000* |

2.7 XOR

Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ

Рис. 9: Машинное представление инструкции XOR

2.7.1 Описание

Производит побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ двух операндов и сохраняет результат в третий

2.7.2 Флаги, затрагиваемые данной инструкцией:

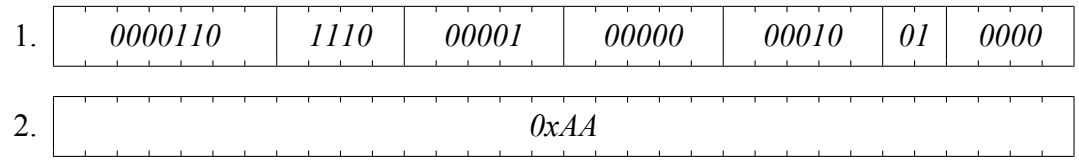
N	Z	C	V
0	+	0	0

2.7.3 Свойства инструкции:

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

2.7.4 Пример использования:

```
XOR r1, 0xAA -> r2  // r1 ИСКЛ. ИЛИ 170 и сохранить в r2
```



2.8 XNOR

Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ

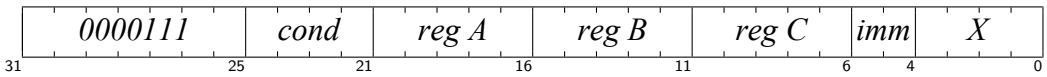


Рис. 10: Машинное представление инструкции XNOR

2.8.1 Описание

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

2.8.2 Флаги, затрагиваемые данной инструкцией:

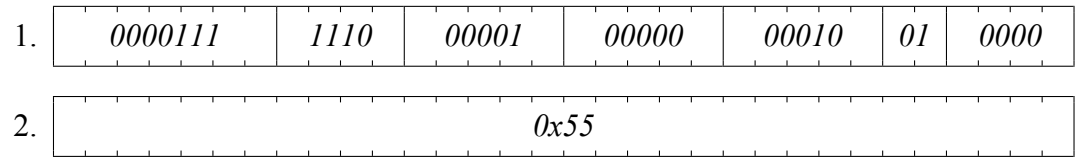
N	Z	C	V
0	+	0	0

2.8.3 Свойства инструкции:

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

2.8.4 Пример использования:

```
XNOR r1, 0x55 -> r2  // r1 ИСКЛ. ИЛИ-НЕ 85 и сохранить в r2
```



2.9 LSL

Логический сдвиг влево

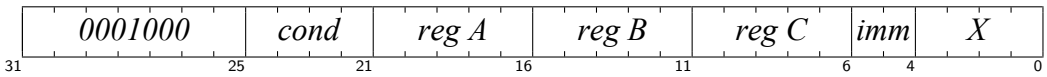


Рис. 11: Машинное представление инструкции LSL

2.9.1 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.

2.9.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	0

2.9.3 Свойства инструкции:

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

2.9.4 Пример использования:

```
LSL r2, 0x04 -
> r1  // сдвинуть r2 влево на 4 бита и сохранить в r1
```

1. | *0001000* | *1110* | *00010* | *00000* | *00001* | *01* | *0000* |
2. | *0x04* |

2.10 LSR

Логический сдвиг вправо

Рис. 12: Машинное представление инструкции LSR

2.10.1 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.

2.10.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	0

2.10.3 Свойства инструкции:

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

2.10.4 Пример использования:

```
LSR r2, r0 -
> r1  // сдвинуть r2 влево на (r0) бит и сохранить в r1
```

1.

0001001	*1110*	*00010*	*00000*	*00001*	*00*	*0000*

2.11 ASR

Арифметический сдвиг вправо

Рис. 13: Машинное представление инструкции ASR

2.11.1 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 .

2.11.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	0

2.11.3 Свойства инструкции:

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

2.11.4 Пример использования:

```
ASR r2, r0 -> r1  // сдвинуть с сохранением знака r2 на (r0)
                  // бит и сохранить в r1
```

1.

0001010	1110	00010	00000	00001	00	0000

2.12 ASL

Арифметический сдвиг влево

Рис. 14: Машинное представление инструкции ASL

2.12.1 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.

2.12.2 Флаги, затрагиваемые данной инструкцией:

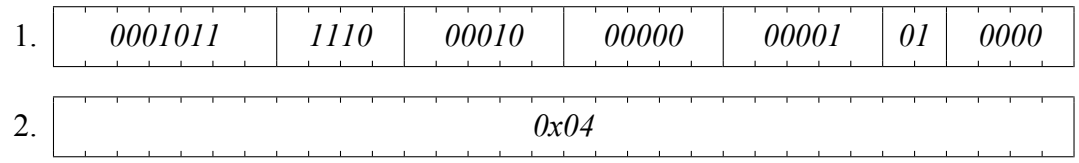
N	Z	C	V
+	+	+	0

2.12.3 Свойства инструкции:

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

2.12.4 Пример использования:

```
ASL r2, 0x04 -
> r1  // сдвинуть r2 влево с сохранением знака на
                    // 4 бита и сохранить в r1
```



2.13 CSR

Циклический сдвиг вправо

Рис. 15: Машинное представление инструкции CSR

2.13.1 Описание

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

2.13.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	0

2.13.3 Свойства инструкции:

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

2.13.4 Пример использования:

```
CSR r2, r0 -> r1  // Циклический сдвиг r2 на (r0)
                  // бит и сохранение в r1
```

1.

0001010	1110	00010	00000	00001	00	0000

2.14 CSL

Арифметический сдвиг влево

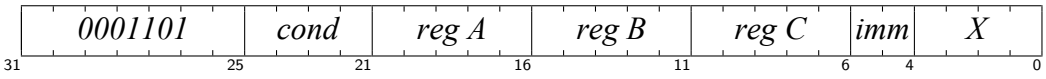


Рис. 16: Машинное представление инструкции CSL

2.14.1 Описание

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

2.14.2 Флаги, затрагиваемые данной инструкцией:

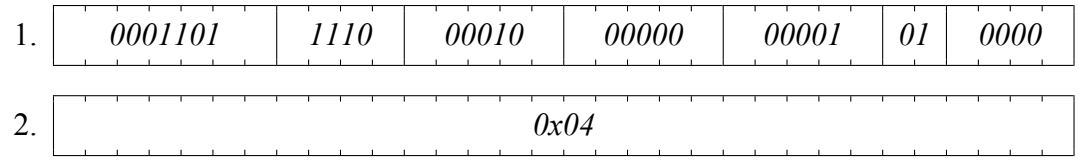
N	Z	C	V
+	+	+	0

2.14.3 Свойства инструкции:

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

2.14.4 Пример использования:

```
CSL r2, 0x04 -> r1  // Циклический сдвиг r2 влево на
                    // 4 бита и сохраниение в r1
```



2.15 ADD

Сложение

Рис. 17: Машинное представление инструкции ADD

2.15.1 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.

2.15.2 Флаги, затрагиваемые данной инструкцией:

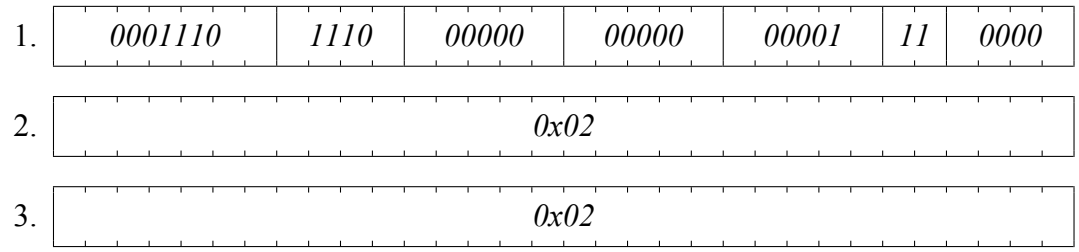
N	Z	C	V
+	+	+	+

2.15.3 Свойства инструкции:

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

2.15.4 Пример использования:

```
ADD 0x02, 0x02 -> r1  // сложить 2 и 2 и сохранить в r1
```



2.16 SUB

Вычитание

Рис. 18: Машинное представление инструкции SUB

2.16.1 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.

2.16.2 Флаги, затрагиваемые данной инструкцией:

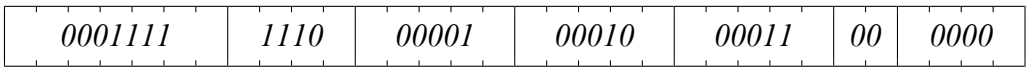
N	Z	C	V
+	+	+	+

2.16.3 Свойства инструкции:

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

2.16.4 Пример использования:

```
SUB r1, r2 -> r3  // вычесть r2 из r1 и сохранить в r3
```

1. 

2.17 MULL

Умножение (32-битная версия)

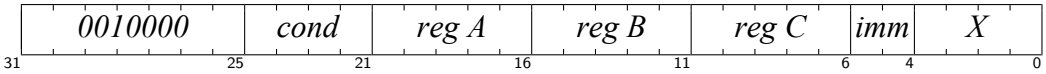


Рис. 19: Машинное представление инструкции MULL

2.17.1 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 ???

2.17.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	+	0

2.17.3 Свойства инструкции:

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

2.17.4 Пример использования:

```
MULL 0x42, r2 -> r3  // Умножить 0x42 на r2 и сохранить в r3
```

1. | *0010000* | *1110* | *00000* | *00010* | *00011* | *10* | *0000* |

2. | *0x42* |

2.18 MULH

Умножение с сохранением старшего слова.

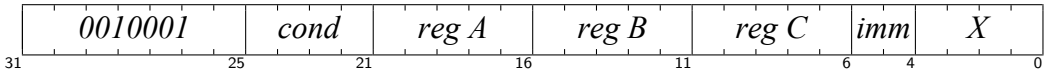


Рис. 20: Машинное представление инструкции MULH

2.18.1 Описание

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

2.18.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
0	+	+	0

2.18.3 Свойства инструкции:

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

2.18.4 Пример использования:

```
MULH 0x42, r2 -> r3  // Умножить 0x42 на r2 и сохранить
                     // старшее слово в r3
```

1. | *0010001* | *1110* | *00000* | *00010* | *00011* | *10* | *0000* |

2. | *0x42* |

2.19 MUL

Умножение (полная версия)

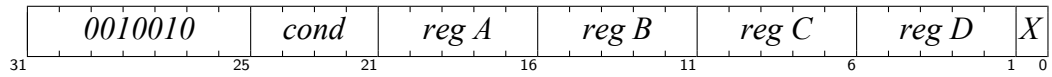


Рис. 21: Машинное представление инструкции MUL

2.19.1 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.

2.19.2 Флаги, затрагиваемые данной инструкцией:

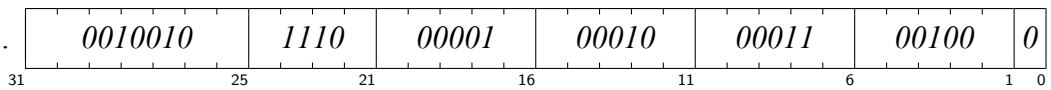
N	Z	C	V
0	+	+	0

2.19.3 Свойства инструкции:

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

2.19.4 Пример использования:

```
MUL г1, г2 -> г3, г4 // Умножить г1 на г2 и сохранить
                     // младший бит в г3, старший в г4
```

1. 

2.20 CSG

Изменить знак

Рис. 22: Машинное представление инструкции CSG

2.20.1 Описание

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

2.20.2 Флаги, затрагиваемые данной инструкцией:

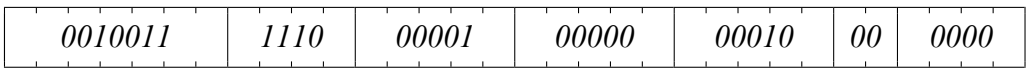
N	Z	C	V
+	+	+	+

2.20.3 Свойства инструкции:

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

2.20.4 Пример использования:

```
CSG r1 -> r2  //  инвертировать знак r1 и сохранить в r2
```

1. 

2.21 INC

Инкремент

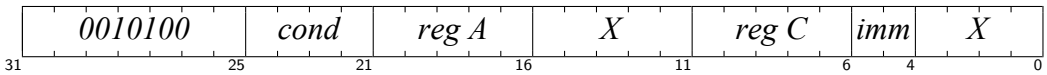


Рис. 23: Машинное представление инструкции INC

2.21.1 Описание

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

2.21.2 Флаги, затрагиваемые данной инструкцией:

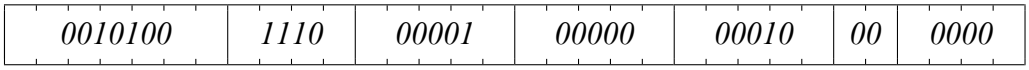
N	Z	C	V
+	+	+	+

2.21.3 Свойства инструкции:

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

2.21.4 Пример использования:

```
INC r1 -> r2  //  инкремент r1 с сохранением в r2
```

1. 

2.22 DEC

Декремент

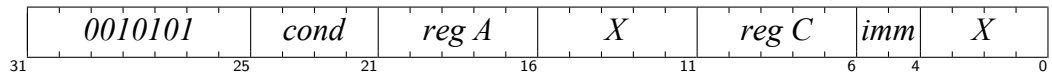


Рис. 24: Машинное представление инструкции DEC

2.22.1 Описание

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

2.22.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

2.22.3 Свойства инструкции:

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

2.22.4 Пример использования:

```
DEC r1 -> r2  //  декремент r1 с сохранением в r2
```

1.

0010101	*1110*	*00001*	*00000*	*00010*	*00*	*0000*

2.23 CMP

Сравнение

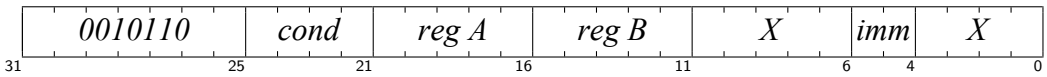


Рис. 25: Машинное представление инструкции CMP

2.23.1 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.

2.23.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
+	+	+	+

2.23.3 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.

2.23.4 Пример использования:

```
CMP r1, 0x01   // Сравнить r1 с 1
DEC_EQ r1 -> r1 // Если равен - декрементировать
```

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

2.24 CMN

Сравнение с обратным знаком

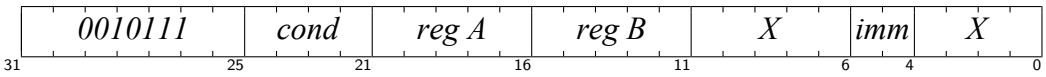


Рис. 26: Машинное представление инструкции CMN

2.24.1 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.

2.24.2 Флаги, затрагиваемые данной инструкцией:

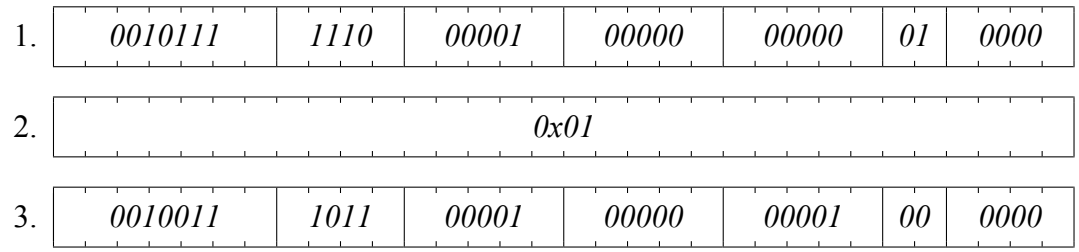
N	Z	C	V
+	+	+	+

2.24.3 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.

2.24.4 Пример использования:

```
CMN r1, 0x00   // Сравнить r1 с 0
CSG_LT r1 -> r1 // Если строго меньше -
изменить знак на противоположный
```



2.25 TST

Проверка («И»)

Рис. 27: Машинное представление инструкции TST

2.25.1 Описание

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

2.25.2 Флаги, затрагиваемые данной инструкцией:

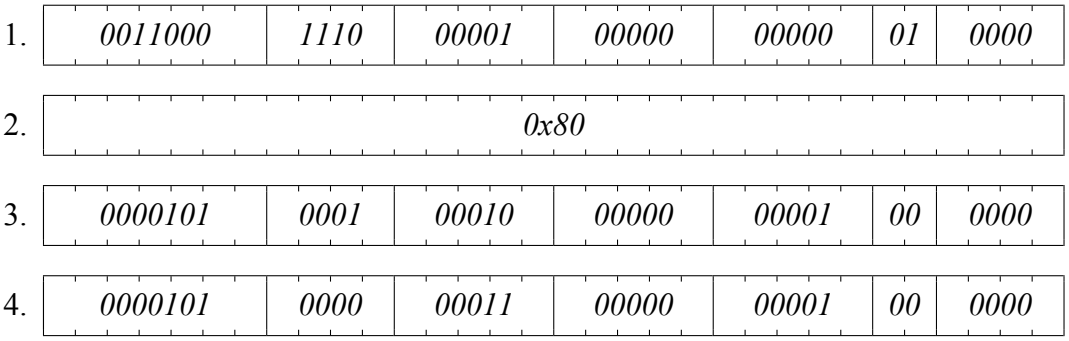
N	Z	C	V
0	+	0	0

2.25.3 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.

2.25.4 Пример использования:

```
TST r1, 0x80   // Проверить наличие в r1 восьмого бита
INV_NEQ r2 -> r1 // Если присутствует - инвертировать r2 в r1
INV_EQ r3 -> r1  // Иначе инвертировать r3 в r1
```



2.26 BR

Прямой переход

Рис. 28: Машинное представление инструкции BR

2.26.1 Описание

Производит прямой переход по адресу в операнде.

2.26.2 Флаги, затрагиваемые данной инструкцией:

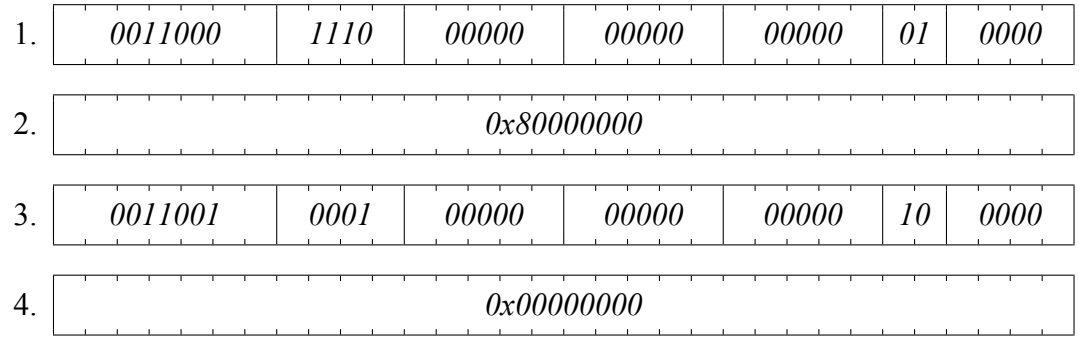
N	Z	C	V
-	-	-	-

2.26.3 Свойства инструкции:

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

2.26.4 Пример использования:

```
TST r0, 0x80000000 // Проверить присутствие в r0 32-го бита
BR_NEQ 0x00000000   // Если присутствует - перейти по адресу 0
```



2.27 RBR

Относительный переход

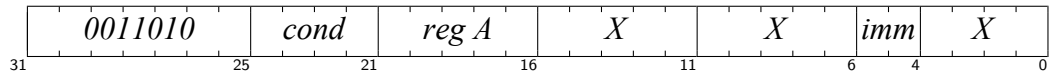


Рис. 29: Машинное представление инструкции RBR

2.27.1 Описание

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

2.27.2 Флаги, затрагиваемые данной инструкцией:

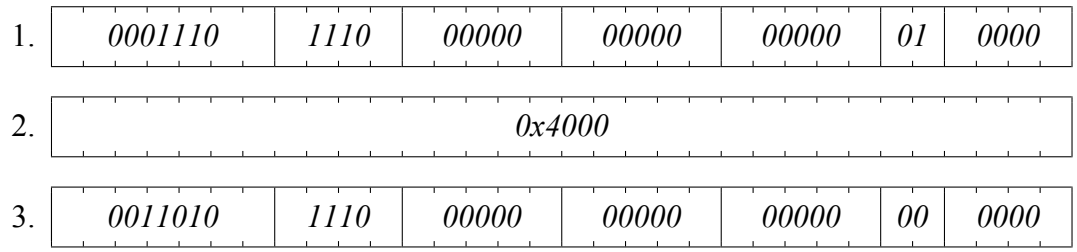
N	Z	C	V
-	-	-	-

2.27.3 Свойства инструкции:

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

2.27.4 Пример использования:

```
ADD r0, 0x4000 -> r0    // Прибавить к r0 смещение 4000h
RBR r0                  // Перейти по смещению r0
```



2.28 BRL

Переход с сохранением адреса возврата

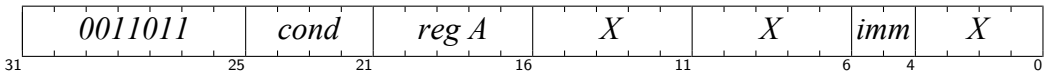


Рис. 30: Машинное представление инструкции BRL

2.28.1 Описание

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

2.28.2 Флаги, затрагиваемые данной инструкцией:

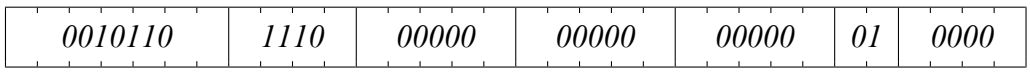
N	Z	C	V
-	-	-	-

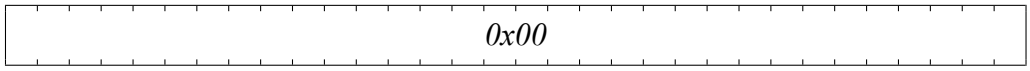
2.28.3 Свойства инструкции:

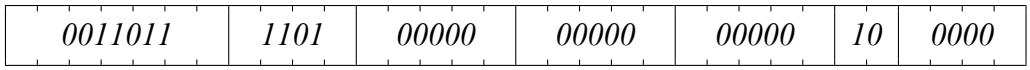
- Исполнение занимает 4 такта.
- Может исполняться условно.
- Может использовать одно мгновенное значение
- Производит запись в регистры pc и lr.
- Изменяет поток исполнения.

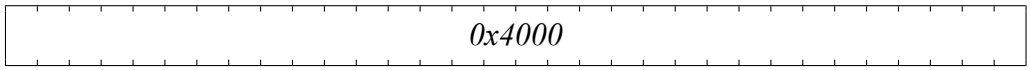
2.28.4 Пример использования:

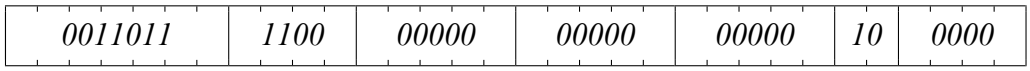
```
CMP r0, 0x00              // Сравение r0 с 0
BRL_LE 0x4000             // Если меньше либо равен -
перейти в процедуру 0x4000
BRL_GT 0x4200             // Иначе перейти в процедуру 0x4200
```

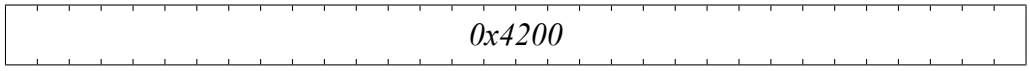
1. 

2. 

3. 

4. 

5. 

6. 

2.29 RET

Возврат

Рис. 31: Машинное представление инструкции RET

2.29.1 Описание

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

2.29.2 Флаги, затрагиваемые данной инструкцией:

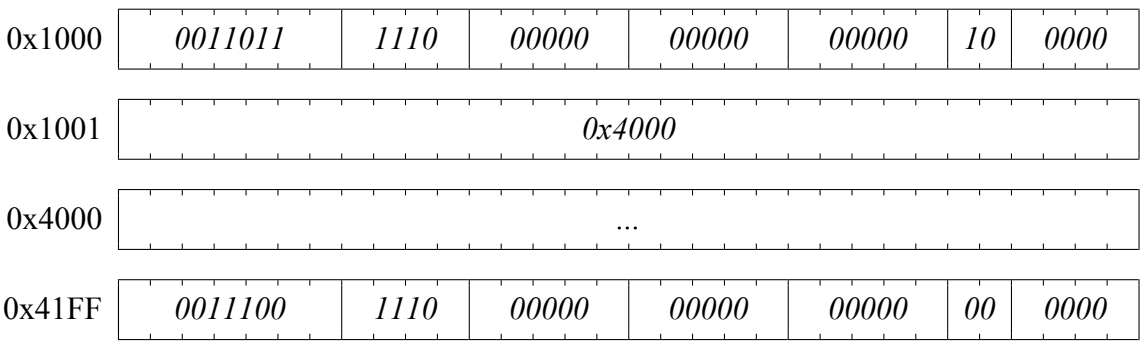
N	Z	C	V
-	-	-	-

2.29.3 Свойства инструкции:

- Исполнение занимает 4 такта.
- Может исполняться условно.
- Не может использовать мгновенные значения.
- Производит запись в регистр pc.
- Изменяет поток исполнения.

2.29.4 Пример использования:

```
0x1000: BRL 0x4000      // Перейти в процедуру 0x4000
0x4000: .....           // Тело процедуры
0x41FF: RET             // Конец процедуры - возврат (в 0x1002)
```



2.30 LDR

Чтение из ОЗУ

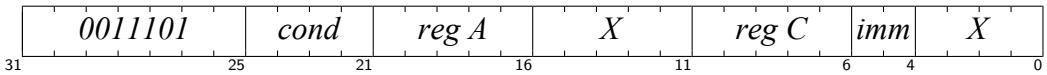


Рис. 32: Машинное представление инструкции LDR

2.30.1 Описание

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

2.30.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

2.30.3 Свойства инструкции:

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

2.30.4 Пример использования:

```
LDR r15 -
> r1  //  Чтение ОЗУ по адресу в r15 с сохранением в r1
```

1. | 0011101 | 1110 | 01111 | 00000 | 00001 | 00 | 0000 |

2.31 STR

Запись в ОЗУ

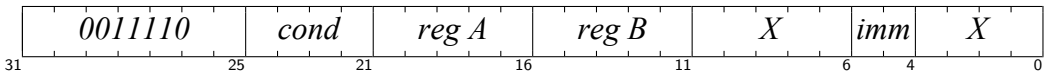


Рис. 33: Машинное представление инструкции STR

2.31.1 Описание

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

2.31.2 Флаги, затрагиваемые данной инструкцией:

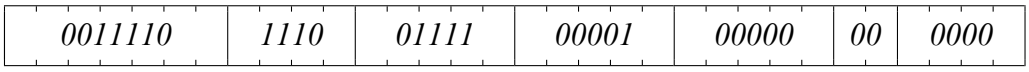
N	Z	C	V
-	-	-	-

2.31.3 Свойства инструкции:

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

2.31.4 Пример использования:

```
STR r15, r1  // Запись r1 по адресу в r15
```

1. 

2.32 IN

Чтение из периферийного регистра

Рис. 34: Машинное представление инструкции IN

2.32.1 Описание

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

2.32.2 Флаги, затрагиваемые данной инструкцией:

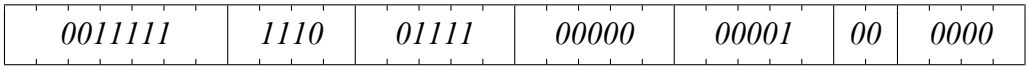
N	Z	C	V
-	-	-	-

2.32.3 Свойства инструкции:

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

2.32.4 Пример использования:

```
IN r15 -
> r1  //  Чтение периферии по адресу в r15 с сохранением в r1
```

1. 

2.33 OUT

Запись в периферийный регистр

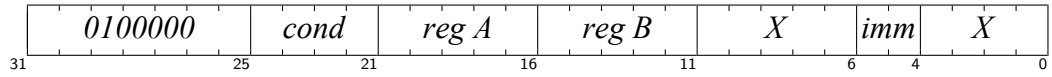


Рис. 35: Машинное представление инструкции STR

2.33.1 Описание

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

2.33.2 Флаги, затрагиваемые данной инструкцией:

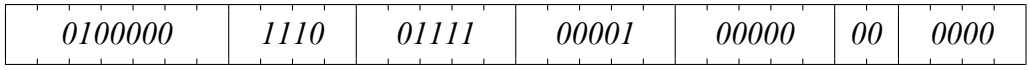
N	Z	C	V
-	-	-	-

2.33.3 Свойства инструкции:

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

2.33.4 Пример использования:

```
OUT r15, r1  // Запись r1 в регистр по адресу в r15
```

1. 

2.34 MOVS

Копирование регистра

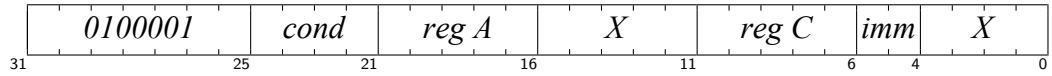


Рис. 36: Машинное представление инструкции MOVS

2.34.1 Описание

Копирует содержимое первого операнда во второй.

2.34.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

2.34.3 Свойства инструкции:

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

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

2.34.4 Пример использования:

```
MOVS r0 -> r15  //  Скопировать r0 в r15
```

1.

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

2.35 MOV

Копирование двух регистров

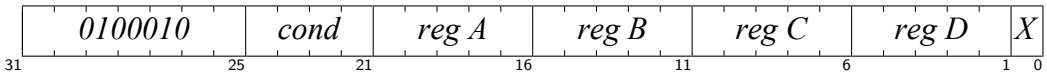


Рис. 37: Машинное представление инструкции MOV

2.35.1 Описание

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

2.35.2 Флаги, затрагиваемые данной инструкцией:

N	Z	C	V
-	-	-	-

2.35.3 Свойства инструкции:

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

2.35.4 Пример использования:

```
MOV г0, г1 -> г15, г16  //  Скопировать г0 в г15, г1 в г16
```

1.

0100010	*1110*	*00000*	*00001*	*01111*	*10000*	*0*

Приложение 2. Исходный код

3 Структура

Данный проект содержит в себе две части:

1. Процессор УП-1.
2. MultiplierGenerator.

Исходные коды выложены в свободный доступ в сети Интернет по адресу:
https://github.com/m0r0zzz/CPU32

3.1 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

3.2 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.

4 Метрики кода

4.1 Процессор УП-1

В таблице 5 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 5: Метрики кода проекта CPU32

4.2 MultiplierGenerator

В таблице 6 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 6: Метрики кода проекта MultiplierGenerator