Министерство образования Республики Беларусь

Учреждение образования Белорусский государственный университет информатики и радиоэлектроники

УДК 004.272

Гридюшко
Александр Витальевич

Высокоскоростная реализация криптографических хэш-функций на базе FPGA

Автореферат

на соискание ученой степени магистра технических наук

по специальности 1-40 80 01 «Элементы и устройства вычислительной техники и систем управления»

Научный руководитель
к.т.н. доцент
Качинский М. В.

Минск 2020

КРАТКОЕ ВВЕДЕНИЕ

Одной из особенностей нашего времени является грандиозный бум информационных технологий. Использование вычислительных систем становится повсеместным, их качество постоянно растет, стоимость вычислительных ресурсов неизменно снижается, а число пользователей неуклонно увеличивается - это лишь несколько основных признаков этого информационного взрыва. Естественным следствием происходящих процессов являетя постоянное увеличение объемов информации, хранимой на носителях и передаваемой по каналам связи, что ужесточает требования к процедурам контроля целостности данных. Это должно выявлять как можно большее число искажений в информационных последовательностях, обеспечивая при этом максимальную производительность; появление нового программного обеспечения и постоянное совершенствование элементной базы и сокращение размеров элементов предъявляет жесткие требования к средствам из тестового диагностирования, которые должны обеспечивать высокую достоверность контроля, иметь высокое быстродействие, регулярную радежность и низкую стоимость.

Разные люди понимают под шифрованием разные вещи. Дети играют в игрушечные шифры и секретные языки. Это, однако, не имеет ничего общего с настоящей криптографией. Настоящая криптография (strong cryptography) должна обеспечивать такой уровень секретности, чтобы можно было надежно защитить критическую информацию от расшифровки крупными организациями — такими как мафия, транснациональные корпорации и крупные государства. Настоящая криптография в прошлом использовалась лишь в военных целях. Однако сейчас, со становлением информационного общества, она становится центральным инструментом для обеспечения конфиденциальности.

Криптографические хэш-функции используются обычно для генерации дайджеста сообщения при создании цифровой подписи. Хэш-функции отображают сообщение в имеющее фиксированный размер хэш-значение (hash value) таким образом, что все множество возможных сообщений распределяется равномерно по множеству хэш-значений. При этом криптографическая хэш-функция делает это таким образом, что практически невозможно подогнать документ к заданному хэш-значению.

Криптографические хэш-функции обычно производят значения длиной в 128 и более бит. Это число значительно больше, чем количество собщений, которые когда-либо будут существовать в мире.

Много хороших криптографических хэш-функций доступно бесплатно. Широко известные включают MD5 и SHA.

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Целью диссертации является анализ возможных архитектурных решений реализации одной из криптографических хэш-функций, позволяющих получить высокую скорость работы выбранного алгоритма, выбор архитектуры для последующей реализациина на базе FPGA и практическая реализация выбранного алгоритма. Указанная цель определяет следующие задачи:

1. Провести обзор существующих криптографических хэш-функций;
2. Выбрать разрабатываемый алгоритм и реализовать его на базе FPGA;
3. Исследовать характеристики полученного устройства и провести их сравнение с существующими аналогами.

В результате проведенного исследования было принято решение использовать алгоритм хэширования SHA-256 для реализации аппартаного ускорителя данной функции на базе FPGA. Научная новизна работы заключается в архитектурных решениях ускорителя и его отдельных узлов, а также в исследовании их характеристик. Практическая значимость работы состоит в разработке аппаратного ускорителя для алгоритма SHA256, которое может быть использовано во встраиваемых системах. Диссертационная работа состоит из введения, четырех глав, заключения и двух приложений.

В первой главе освещены краткое введение в хэш-функции, описание различных алгоритмов, принцип их функционирования. Во второй главе рассматриваются известные реализации алгоритмов, а также принципы организации архитектур, вошедшие в основу выбора криптографической хэш-функции исследовательской работы. Третья глава описывает алгоритм, схемотехническое моделирование архитектуры проекта, а так же некоторые результаты синтеза проекта в среде разработки VIVADO. В четвертой главе содержатся результаты проведенных тестов и сравнительный анализ с ранее рассмотренными реализациями.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Ядро аппаратного блока SHA256 состоит из канала данных, который манипулирует входящими словами и создает окончательный хэш, управляемый сигналами управления, и блока управления, конечного автомата, который обеспечивает сигналы управления. Основная идея данной реализации канала данных SHA256 вдохновлена некоторыми оптимизациями, описанными в статье Чавеса, Кузманова, Соузы, Василиадиса [42]. В частности, было реализовано перепланирование операций и повторное использование оборудования, разделив вычисление окончательного хэша на разные этапы конвейера. В целом, вычисление окончательного хэша должно занять больше тактов по сравнению со стандартным, но частота должна быть выше, а занимаемая область меньше.

После правильной инициализации аппаратный блок будет обрабатывать по одному 32-битному слову за раз и генерировать 256-битный хэш каждые 16 слов (512 бит). С точки зрения AXI протокола, он будет работать в качестве ведомого устройства на шине. Входы и выходы записываются в следующие 32-битные регистры:

- $data\_reg$:по адресу 0x40000000. Он записывается ARM процессором и читается блоком SHA256, содержащий 32-битное слово, которое будет обработано аппаратным блоком.
- $status\_reg$:по адресу 0x40000004. Может быть записано и прочитано обоими (следовательно, действует как контрольный регистр)
  - $status\_reg(0) = msg\_ready$. Аппаратный блок сигнализирует о готовности обработать новое слово, установив его в 1.
  - $status\_reg(1) = hash\_valid$. Если установлено в 1, то значит, что был обработан весь 512-битный блок, и результат можно найти в регистрах от h0 до h7.
  - $status\_reg(2) = hash\_ack$. Процессор устанавливает этот бит, чтобы сигнализировать, что он прочитал хэш, и аппаратный блок может подготовиться к получению нового слова.
  - $status\_reg(3) = msg\_last$. Устанавливается процессором, когда текущее слово является последним из последних 512-битных блоков.
- $h0, h1, h2, h3, h4, h5, h6, h7$: начиная с адреса 0x40000008 до адреса 0x400000024 содержат слова, которые составляют хэш каждого 512-битного блока. Записывается блоком SHA256 и читается процессором.

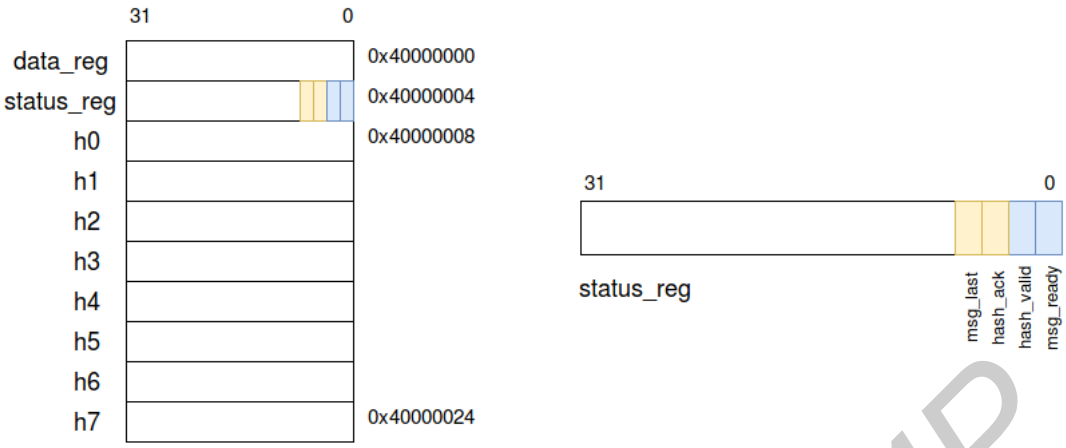

Рисунок 1. – Отображение памяти

Драйвер SHA256 должен поддерживать следующие операции: открыть, освободить, прочитать, записать. И чтение, и запись позаботятся об одном 32-битном слове за раз.

Пользовательский интерфейс предлагает пользователю предоставить сообщение для хэширования. Поскольку сообщение будет вставлено через клавиатуру, оно всегда будет кратно 8 битам. Первая часть SHA256 - это заполнение, которое можно обобщить за три операции:

1. добавить бит «1» в конце сообщения
2. добавить K '0' битов, где K - минимальное число $>= 0$, так что $len(message) + 1 + K + 64$ кратно 512
3. добавить длину сообщения в последние 64 бита

Эти операции не являются сложными в вычислительном отношении и, кроме того, они не являются регулярными (в худшем случае должен быть добавлен новый 512-битный блок). Следовательно, дополнение может быть реализовано в программном обеспечении самим приложением.

Общий дизайн оборудования был реализован в виде иерархической структуры. Канал данных, который создает экземпляр блока расширения, и блок управления содержатся в оболочке sha256. В свою очередь, оболочка создается с помощью sha256_ctrl_axi , который определяет FSM для чтения и записи протокола AXI и определяет регистры. Некоторые полезные функции были определены в пакете, вместе с K-константами и значениями инициализации H. По причинам отладки пакет сам реализует алгоритмы SHA256, чтобы сравнивать результаты во время моделирования, но он не будет синтезирован. Блок-схема представлена на рисунке 2

Канал данных является фактическим ядром алгоритма SHA256 и был построен на конвейерной модели, как показано на рисунке 3. Блок расширения

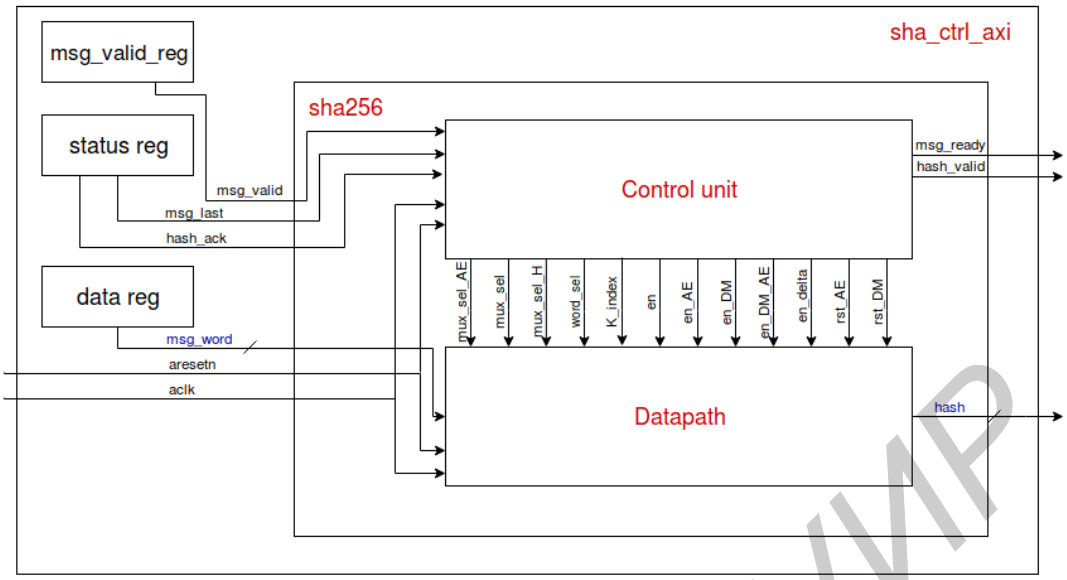

Рисунок 2. – Блок-схема

был реализован простым образом в виде цепочки из 16 триггеров. Он принимает в качестве входных данных текущее 32-битное слово и начинает сдвигать и обрабатывать его при каждом такте, реализуя следующие соотношения:

- $s0 := (w[i-15]rightrotate7)xor(w[i-15]rightrotate18)xor(w[i-15]rightshift3)$
- $s1 := (w[i-2]rightrotate17)xor(w[i-2]rightrotate19)xor(w[i-2]rightshift10)$
- $w[i] := w[i-16] + s0 + w[i-7] + s1$

Мультиплексор выбирает выход: для первых 16 тактов это сам вход $w[i]$, а для следующих 48 тактов это выход 16-го триггера цепочки.

Идея перераспределения операций заключается в том, что некоторые операции, особенно те, которые необходимы для вычисления A и E, могут быть выполнены заранее. В частности:

$$\delta[i] := H[i] + W[i] + K[i] \tag{1}$$

$$A[i+1] := \sum 0(A[i]) + \sum 1(E[i]) + Maj(A[i], B[i], C[i]) + Ch(E[i], F[i], G[i]) + \delta[i] \tag{2}$$

Поскольку $H[i] = G[i-1] = F[i-2]$, $K[i]$ является константой и $W[i]$ может быть предварительно вычислено, вычисление $\delta.,,.,\delta[i] := H[i] + W[i] + K[i]$ может быть выполнено с использованием цепочки сумматоров с сохранением переноса(ССП). Вместо того, чтобы выдавать только один результат в качестве стандартного сумматора, сумматор сохранения переноса возвращает

два вектора: частичную сумму и биты переноса. Таким образом, нет переноса переноса, и задержка значительно уменьшается. Первый ССП, самый левый на рисунке 3, получает в качестве входных данных $K[i+1], W[i+1]H[i+1]$, которые будут использоваться для вычисления значений A и E для следующего раунда алгоритма SHA256. $H[i+1]$, в свою очередь, может быть выбран из $DM7, DM6, DM5F[i]$, в соответствии с этапом алгоритма, с помощью сигнала выбора mux_sel_H. Выход этого первого ССП входит в два других ССП. Они обычно суммируют все нули, не изменяя результат, но на этапе инициализации. Фактически, когда необходимо вычислить новый хеш, A и E должны быть инициализированы постоянными значениями 0x6A09E667 и 0x510E527f соответственно. Для этого выход первого ССП поддерживается равным нулю (сигнал rst_AE) вместе со всеми другими переменными (от B до G), в то время как мультиплексоры выбирают $DM0$ и $DM4$ (сигнал mux_select_AE). Эти два значения будут неизменными через ССП и конвейерные регистры и будут загружены в A и E. Выходные данные двух ССП сохраняются в конвейерах и на следующем шаге будут суммироваться с необходимыми операндами (Maj, Ch, ...) для вычисления следующих значений A и E. Это дополнение было реализовано с помощью просто суммирования в VHDL и оптимизация остается на стадии синтеза.

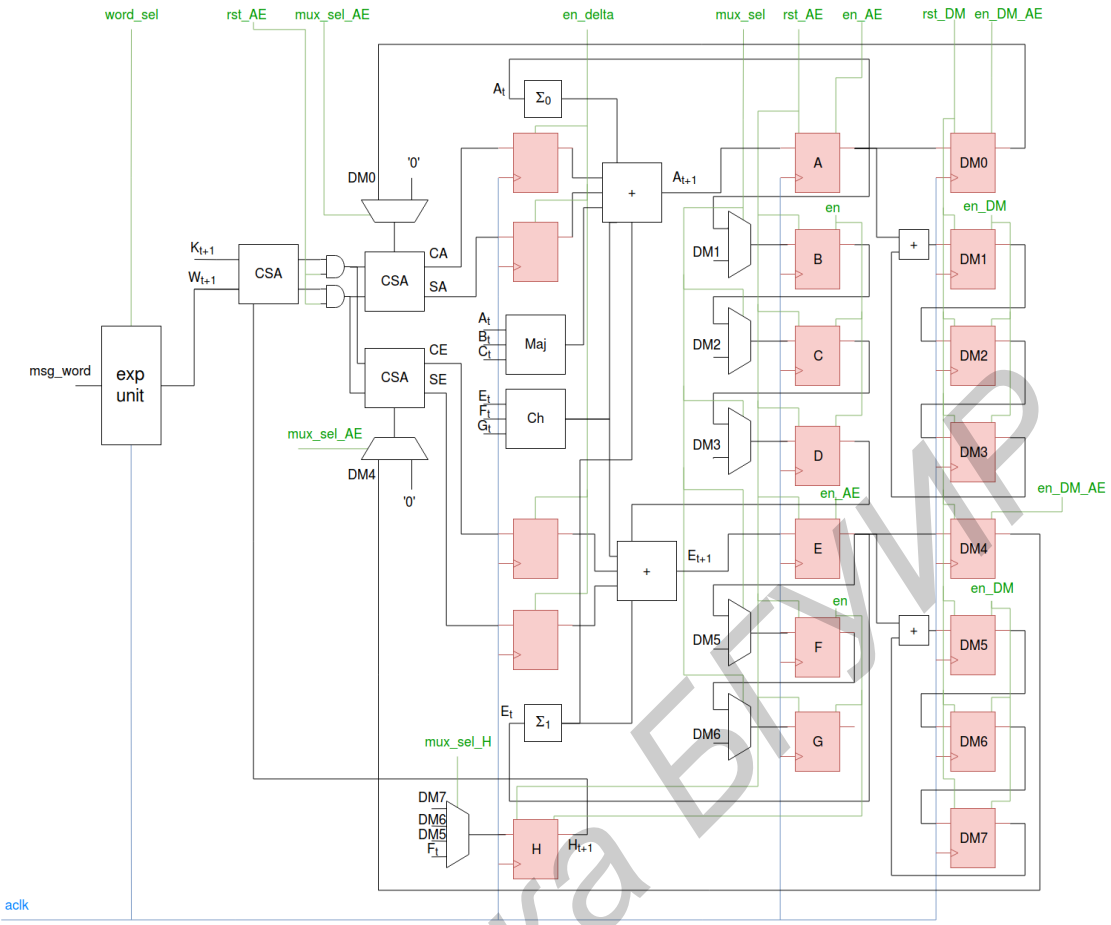

Рисунок 3. – Datapath

На последнем этапе обработки 512-битного блока полученный хэш должен быть загружен в регистры DM, суммируя значения, в настоящее время находящиеся в A в H, с значениями, уже сохраненными в DM:

- DM0 := DM0 + A
- DM1 := DM1 + B
- DM2 := DM2 + C
- DM3 := DM3 + D
- DM4 := DM4 + E
- DM5 := DM5 + F
- DM6 := DM6 + G
- DM7 := DM7 + H

Тем не менее, можно заметить, что:

- B[i] := A[i-1]
- C[i] := A[i-2]
- D[i] := A[i-3]
- F[i] := E[i-1]

- G[i] := E[i-2]
- H[i] := E[i-3]

Следовательно, можно избежать создания экземпляров 8 сумматоров, которые будут потреблять площадь и мощность, и использовать вместо них только 2 путем реализации логического цикла на основе сдвигового регистра (справа на рисунке 3). Регистры DM сбрасываются (rst_DM) на постоянные начальные значения, определенные в пакете, и включаются (en_DM) в течение последних 3 циклов вычисления, чтобы суммировать текущие значения с правильными значениями A и E и повторно загружать их в DM. $DM0$ и $DM4$ немного различаются: они загружают непосредственно значения A и E, что происходит с одним тактом задержки по отношению к другим; Вот почему им нужно специальное включение en_DM_AE.

Блок управления представляет собой автомат Мили, который состоит из 7 этапов:

1. $IDLE$: КА остается в этом состоянии до тех пор, пока не будет установлен msg_valid, сигнализирующий новое слово для обработки. Все регистры DM инициализируются постоянными значениями, определенными в пакете. От A до G сбрасываются на 0, в то время как H устанавливается на постоянное начальное значение, чтобы быть готовым к предварительному вычислению. Разрешение регистров равно 0, поэтому конвейер остается в тупике. Когда msg_valid установлен, блок управления активирует разрешения и регистры с A по G загружаются со значениями в DM.
2. $INIT$: регистры с A по H корректно инициализированы, а сигнал mux_sel установлен для старта алгоритма сжатия.
3. $COMPRESSION1$: это первая фаза алгоритма сжатия, в которой конвейер ожидает нового входящего слова (msg_valid) и выполняет вычисления. Счетчик указывает, какое значение констант K выбрать.
4. $COMPRESSION2$: когда счетчик достигает 15, запускается вторая стадия алгоритма сжатия. Конвейеру больше не нужны новые слова, потому что W предоставляется блоком расширения. Во время последних циклов сжатия регистры DM активируются, как описано выше, для запуска логического цикла.
5. $WAIT\_HASH1$: A и E нужен еще один такт, поскольку они зависят от значений других регистров. В конце этого шага конвейер готов начать новое предварительное вычисление для хеширования следующего 512-битного блока.

6. $WAIT\_HASH2$: $DM0$ и $DM4$ загружаются с правильным хэш-словом. При необходимости запускается новое предварительное вычисление δ (устанавливается msg_valid). В противном случае, если установлен msg_last, сигнализируя последний 512-битный блок, следующее состояние становится $HASH\_READY$.
7. $HASH\_READY$: конвейер выводит окончательный хэш и вызывает $hash\_valid$, ожидая $hash\_ack$.

На рисунке 4 показано поведение основных сигналов SHA256 во время хеширования одного 512-битного блока. Можно видеть, что конвейер остается в тупике во время фазы $COMPRESSION1$, потому что слова не доступны сразу.

КА для чтения и записи протокола AXI определен в оболочке sha256_ctrl_axi. Он также определяет ранее упомянутые входные и выходные регистры. Аппаратный блок SHA256 ведет себя как ведомый на шине и может записывать только последние 2 бита регистра состояния и регистры с h0 по h7. Вместо этого процессор может записывать регистр данных, $status(3)$ и $status(2)$, в то время как все остальные сопоставленные адреса будут иметь ошибку только для чтения. Процесс заботится о создании сигнала msg_valid: он устанавливается каждый раз, когда записывается текущее состояние ($s0\_axi\_awvalid$ и $s0\_axi\_wvalid$ устанавливаются одновременно), а адрес соответствует одному из регистра данных. Таким образом, аппаратные блоки знают, когда было написано новое 32-разрядное слово и доступны для вычисления.

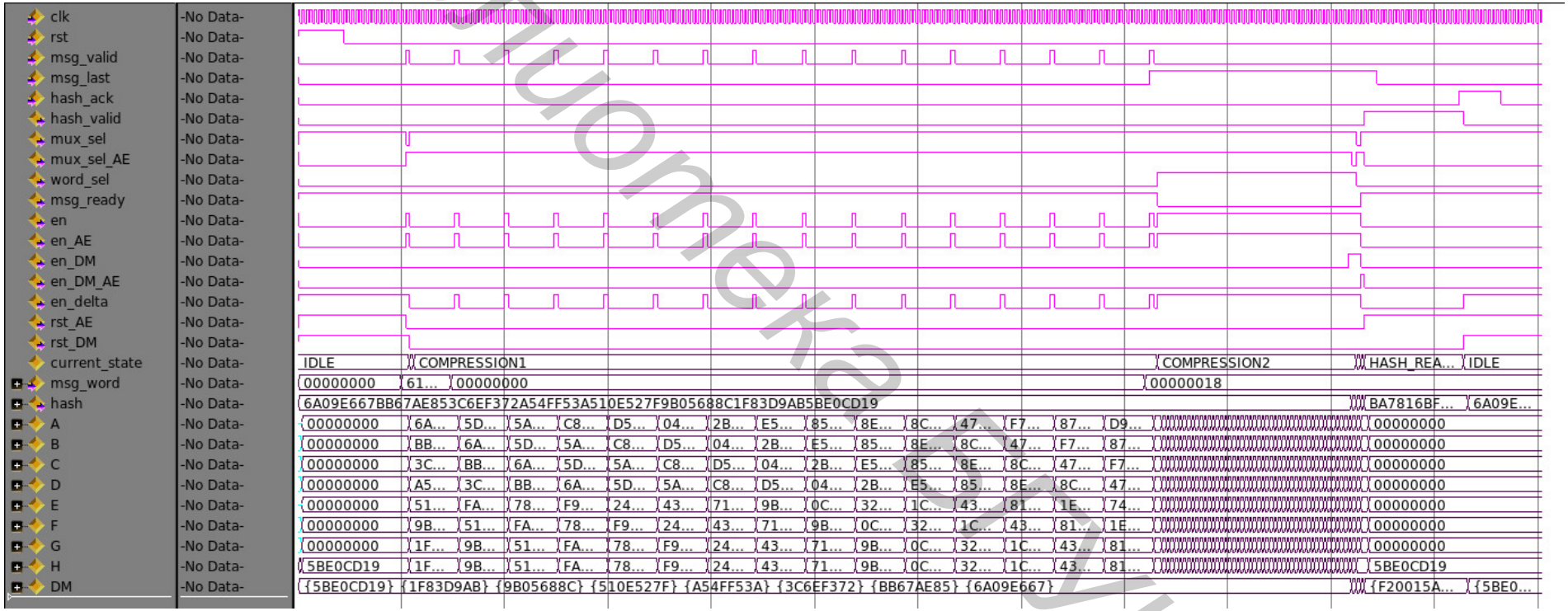

Рисунок 4. – Временная диаграмма

ЗАКЛЮЧЕНИЕ

В диссертационной работе был проведен анализ возможных архитектурных решений криптографических хэш-алгоритмов, позволяющих получить высокую скорость. Архитектура описана с помощью языков VHDL, C и синтезирована средствами САПР Vivado 2019. Для подтверждения работоспособности разработанного устройства написана программа для получения значений в требуемых точках алгоритма вычислений, а также проведено тестирование процессора на известных сообщениях. Тестирование подтвердило правильность функционирования устройства. Исследованы характеристики (аппаратные затраты и производительность) разработанного процессора при его реализации на FPGA семейства Zynq. Предлагаемая реализация по сравнению с известными имеет примерно сопоставимую производительность, однако требует на 20-30% меньше аппаратных ресурсов кристалла ПЛИС. Результаты разработки свидетельствуют о выполнении цели и всех задач исследования. Разработанный процессор алгоритма SHA256 может быть использован как аппаратный ускоритель в системах, требующих высокой скорости работы алгоритма.

Список публикаций соискателя

1–А. Гридюшко, А. В. Высокоскоростная реализация криптографических хэш-функций на базе FPGA / А. В. Гридюшко // Компьютерные системы и сети: 55-я юбилейная научная конференция аспирантов, магистрантов и студентов, Минск, 22-26 апреля 2019 г. / Белорусский государственный университет информатики и радиоэлектроники. – Минск, 2019. – С. 255 – 256.