INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Parallelization of AES on Raspberry Pi GPU in Assembly

Paul Pauls

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Parallelization of AES on Raspberry Pi GPU in Assembly

Paul Pauls

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde
	Tobias Guggemos
Abgabetermin:	10. Juli 2017

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 10. Juli 2017

. . . .

(Unterschrift des Kandidaten)

Abstract

This thesis will conduct research upon the parallelization potential of the Raspberry Pi 1 GPU and if possible speed-up can be achieved through said parallelization. It answers this question by documenting the parallelization of the Advanced Encryption block cipher algorithm. Based on the byte manipulating complexity of AES, which GPU supported APIs like OpenGL-ES can not offer, is a GPU specific assembly language chosen as the tool of parallelization, which was created in the course of this thesis. The usability of the created AES implementation and by extension the performance potential of the GPU is determined by performing multiple benchmark tests comparing the GPU implementation as well as OpenSSL, Rijndael, tiny-AES and a FIPS 197 reference implementation. The conducted research shows the excellent speed-up potential provided by the GPU, even though the GPU implementation is in its current form significantly lacking in efficiency. Despite problems with the memory of the GPU, due to which the actually usable GPU memory is decreased from 48 KB to 192 byte does the GPU implementation by far outperform tiny-AES and the FIPS 197 reference. The GPU even performes the certainly improvable implementation of AES on par with the highly optimized Rijndael CPU implementation once the data has been transmitted to GPU memory. Though as of yet does the GPU performance of AES lag behind AES performed by OpenSSL on the CPU. On the other hand does the created GPU interface allow for an easier speed-up through parallelization on the GPU than high optimiziation on the CPU.

Contents

1. Introduction

In June 2013 the newspaper *The Guardian* published classified documents from the United States National Security Agency (NSA) leaked to them by the disputed whistleblower Edward Snowden [Gua13]. The leaked documents outlined the digital surveillance undertakings of tremendous scale conducted by the *Five Eyes* intelligence alliance consisting of the USA, the UK, Canada, Australia and New Zealand. The use of the surveillance tools crafted in the process was diverse and the Five Eyes alliance used them in ways ranging from observing the citizens of their own countries via access to smartphone messengers and social networks, to wiretapping the phone of German chancellor Angela Merkel [Reu13].

The resulting scandal caused a significant change in the mindset of companies, online-communities and security-minded individuals making them realize not just the importance but the necessity of encryption in the digital age. According to the network equipment company Sandvine, the use of encryption in the Internet has risen to 70% in 2016 with many networks exceeding 80% [San16]. Mobile network communication had at least 60% of their traffic encrypted. The need for optimized encryption is therefore obvious, even more so on mobile and Internet-of-Things (abbr. IoT) devices which lack the processing power of a traditional computer and are dependent upon optimization to keep pace.

One of the most popular of such IoT devices is the Raspberry Pi - a Single-Board Computing Platform released in February 2012. Being the first credit-card sized computing platform it grew to be to most popular choice with over 10 million units sold up to August 2016 [Ltd16]. In February 2016 the Raspberry Pi was released in its third iteration, each improving on CPU-Power, RAM and available peripherals. Nevertheless features every version the same powerful but largely unused *Graphics Processing Unit* (abbr. GPU) with a theoretical maximum processing power of 24GFLOPs distributed over 12 compute units. However even though every Raspberry Pi iteration features the same GPU, its manufacturer only released the GPU kernel driver for the System-on-Chip of the first Raspberry Pi iteration - making it an aspect of this thesis to study the accessibility of later iterations.

All in all can be said that the Raspberry Pi - foremost its first iteration - represents a class of IoT devices with a weak embedded processor but powerful parallelized GPU. On the other hand are the encryption algorithms which are required to protect from illegitimate access very resource-intensive, especially for those weak embedded processors of IoT devices, though a variety of the most popular encryption algorithms are well parallelizable. Therefore does the Raspberry Pi also represent a class of IoT devices upon which the contradiction of a resource-intensive encryption and a weak embedded processor can be solved by parallelizing the encryption on the GPU.

This thesis will address the potential of speeding up one of the most widespread, secure and parallelizable encryption algorithms - the *Advanced Encryption Standard* (abbr. AES) - on the Raspberry Pi 1 Single-Board Computing Platform. The Advanced Encryption Standard was choosen as it is trusted by the U.S. goverment and numerous private organizations as the standard for encrypting even highly confidential information of their own, to protect it even from members of their own Five Eyes alliance. The AES algorithm arised as the most secure encryption algorithm proposed in the selection process for the next encryption standard held in 2001. It competed against rivaling encryption algorithms such as Twofish and Serpent [NIS97], was examined by numerous cryptography experts and is still considered to be unbroken and deemed secure despite the efforts of the Five Eyes alliance [Sch12]. Furthermore does AES provide a high potential for speedup by parallelization. Because of AES being a block cipher can the encryption of multiple blocks be parallelized trivially by distributing the task of single block encryption to single computation units of a parallel system. The Raspberry Pi 1 offers 12 of those parallelizable compute units on its GPU, promising a significant speedup of encryption on an IoT device and an interesting study.

Approach of this Thesis

The approach undertaken in this thesis plays out as following: First of all is the necessary background knowledge established in chapter 2, which will outline the exact specifications of the Advanced Encryption Standard and how it can be parallelized on multiple compute units. Furthermore will the second chapter touch upon the Raspberry Pi GPU Architecture, as understanding its design and interaction with the CPU is crucial for a well implemented parallelization. Chapter 2 will close with a short summary of the established background while shortly introducing related research, which is limited.

Chapter 3 will provide a detailed look into how exactly general purpose computation can be realized on the Raspberry Pi GPU. The chapter accomplishes this by detailing on the created programming language that can generally be described as GPU specific assembly, how this programming language is assembled to GPU usable binaries and finally how the assembled binaries and possible additional data are transferred to the GPU for execution.

With this information about the programming of the GPU in mind will chapter 4 extensively introduce the implementation of AES on the GPU, which was created in the course of this thesis. The implementation is detailed by mirroring the specifications of the single components of the AES algorithm which were introduced in the background chapter. This chapter will close by outlining the significant problems encountered during the implementation of the AES algorithm on the GPU, which significantly affect the results of the experiments conducted in the final research chapter.

Chapter 5 will explain and showcase the conducted experiments, trying to give an answer to the initial question of this thesis about the GPU's possible parallelization potential. The last section of this chapter will provide the important discussion of those conducted experiments, drawing some optimistic conclusions. After all tests and measurements have been thoroughly examined will the final chapter 6 recapitulate on the undertaken creation and improvement of code, conducted research and achieved testresults. This thesis will end with a drawn conclusion to the question about the GPU's potential and listing of various ideas for future research based on the achieved results.

2. Background

In order to parallelize any algorithm it is first necessary to know the exact specifictions of this algorithm, which is why the first section will extensively outline the AES block cipher algorithm. In the case of AES more specifically is also a firm understanding of the possible block cipher modes of operation required to determine how AES can be parallelized best on the GPU. Following in the second chapter will be an extensive documentation of the hardware and especially the GPU of Raspberry Pi 1 Model B. A final summary of this chapter and short outlook to related work in section three, will specify how the AES algorithm is best implemented on the GPU and will therefore bridge to chapter 3, providing the information to access the hardware outlined in this chapter.

2.1. The Advanced Encryption Standard

The *Advanced Encryption Standard* is a symmetric block cipher algorithm that was first proposed in 1998 as a subset of the Rijndael cipher and standardized in 2002 by the *National Institute of Standards and Technology* (abbr. NIST). According to NIST it is the only symmetric encryption algorithm secure enough to encrypt top secret government documents [NIS01]. To this day there are no known software attacks that significantly lower the complexity of decrypting AES without any information about the used key [Ou06], however there are very successful side-channel attacks involving access to hardware [Sah09].

A block cipher algorithm like AES is a deterministic encryption algorithm operating on a fixed-length group of bits, called blocks, which in the case of AES is 128 bits, which translate to 16 bytes. The encryption algorithm furthermore only needs a key and the text one would like to encrypt, called plaintext, to create an unreadable text, called ciphertext. This ciphertext can only be decrypted and therefore read if one is in possession of the decryption key. AES is a symmetric block cipher, because it uses the same key for encryption as it does for decryption.

Calling an encryption algorithm an AES algorithm does however only partially specify the encryption, as a block cipher encryption algorithm consists of two concepts. The AES algorithm determines the first one of these two concepts, that is: AES determines how to encrypt a single block (16 bytes in AES) of data. AES does not determine how to handle the data if the input into the encryption algorithm is larger than one block, which is the second of the two concepts. This is determined by the block cipher mode of operation, e.g. the ECB mode, the CBC mode or the CTR mode to name the most popular ones. Basically, a block cipher mode of operation determines if each block of data is treated in reclusion, or if the input blocks influence each other during en-/decryption and if so, in which way.

Therefore, to correctly specify the encryption method when using AES, one has to specify the block cipher mode of operation as well. Hence is the first section dedicated to explaining the AES algorithm and how it encrypts and decrypts a single block of data. The second section addresses how AES works in combination with different block cipher modes of operation, namely AES-ECB, AES-CBC and AES-CTR.

2.1.1. The AES Block Algorithm

The AES algorithm requires a 16-byte block of plaintext and a key of either 128-bit, 192-bit or 256-bit length as input. A bigger sized key results in a better protection for the resulting ciphertext by making brute force attacks more ineffective [Axa08]. The block of plaintext is best looked upon as a 4x4 matrix, termed state throughout the algorithm, as a lot of AES functions perform matrix-like operations on the state. Initially the 16 bytes of the plaintext have the corresponding positions as shown in illustration 2.1, which is displaying a 16-byte block in memory.

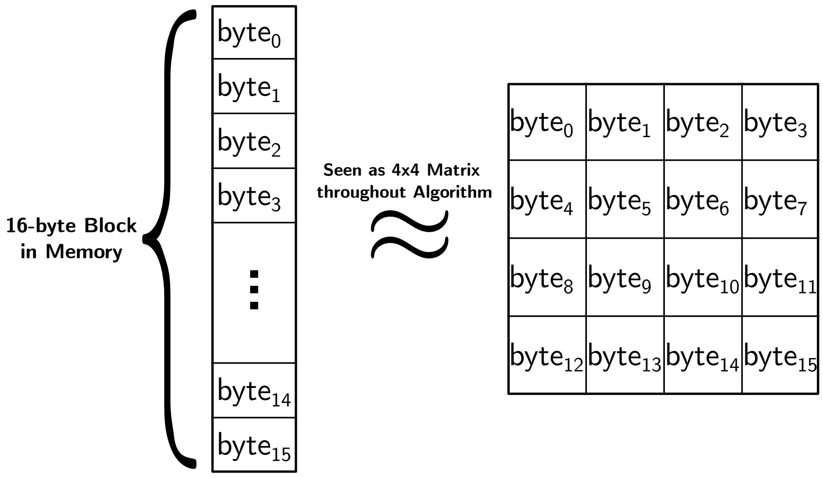


Illustration 2.1.: 16-byte Block in Memory seen as State-Matrix

After the supplied key is then applied to the plaintext according to the AES encryption algorithm, the state-matrix then represents the 16-byte block of ciphertext. This ciphertext can in turn be decrypted again by using the same key and the slightly different AES decryption algorithm. Both algorithms of AES - encryption and decryption - consist of four major parts and can be reduced to the operations depicted in illustration 2.2. The inverse functions used in decryption differ only slightly from the regular functions, nonetheless will each function be explained in detail in the paragraphs following.

First of all is the initially supplied key determinstically expanded in `key_expansion()` in order to provide a sufficient amount of different round-keys for every application of the `add_round_key()` function, which is called a total of 11 to 15 times throughout the algorithm. The key takeaway with round-keys is that they need to be different for every call of the `add_round_key()` function while also being deterministically dependent on the initially supplied key.

Following the `add_round_key()` function which is making up the whole *Initial-Encryption-Round*, is the start of the *Cycling-Encryption-Rounds*. The Cycling-Encryption-Rounds consist of `sub_bytes()`, `shift_rows()`, `mix_columns()` and `add_round_key()` in that order or the corresponding inverse functions when decrypting. The functions in the Cycling-Encryption-Rounds unit are repeated multiple times depending on the different key-sizes, meaning dependent on the intended strength of the encryption. These total cycling numbers are the same for encryption and for decryption and depend on the size of the initially supplied key (9 total cycles for 128-bit keys, 11 total cycles for 196-bit keys, 13 total cycles for 256-bit keys).

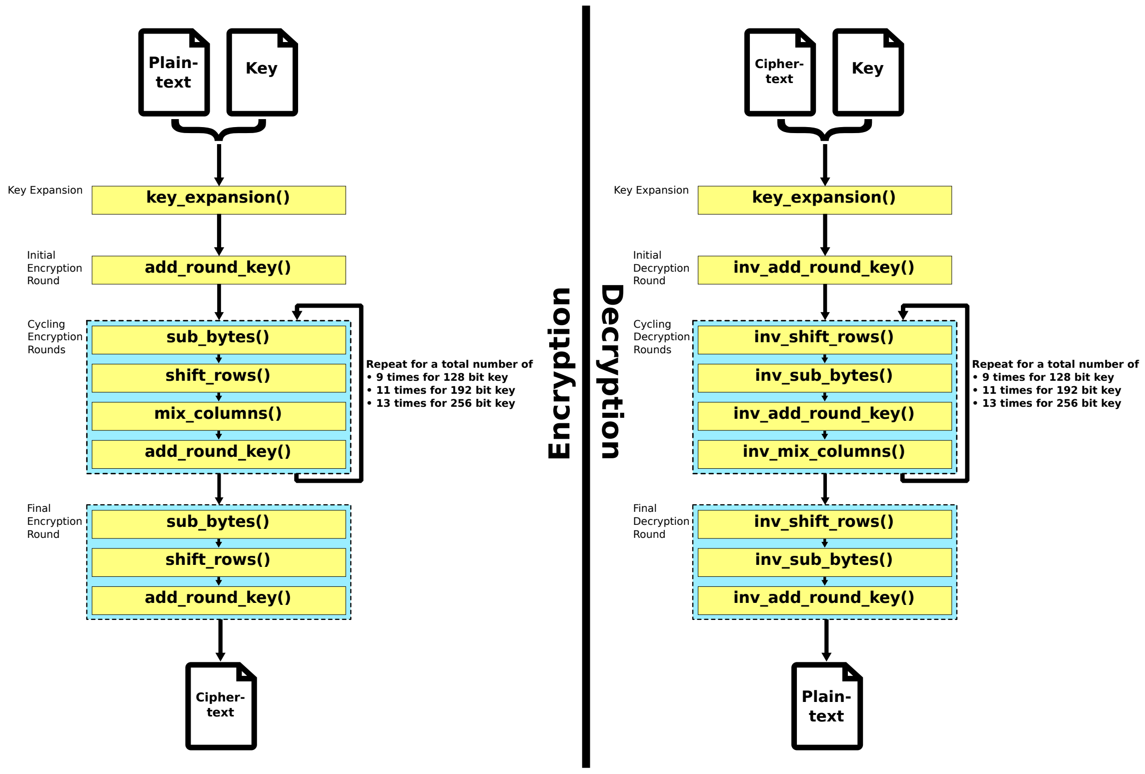


Illustration 2.2.: AES Encryption (left) and AES Decryption (right) Algorithm

The different encryption strengths of the AES algorithm differ only in the quantity of repetition cycles for the Cycling-En-/Decryption-Rounds and the therefore necessary different keysize (128-bit, 192-bit or 256-bit) and corresponding slightly different `key_expansion()`. Finally the algorithm closes with a *Final-En-/Decryption-Round*, which is basically the same as the Cycling-En-/Decryption-Round except for leaving out the `mix_columns()` function or `inv_mix_columns()` function respectively.

The 16-byte matrix-state after the Final-Encryption-Round is the resulting ciphertext. When decrypting the matrix-state represents the plaintext after the Final-Decryption-Round.

The add_round_key() and inv_add_round_key() functions

The `add_round_key()` function XORs the 16-byte matrix-state with the current round-key. A round-key is a 16-byte part of the expanded-key, which in turn is the result of the deterministically expanded initial key from the `key_expansion()`. It is called round-key because every round-key is only used once in the 11 to 15 total calls of the `add_round_key()` function, therefore making each call effectively a *round*. Being as the function only XORs two 16-byte values, the implementation is trivial and listed as A.1 in the appendix. The inverse function differs from the regular function in the aspect that the inverse function applies the round-keys in reversed order - so it applies the last 16-byte round-key of the expanded-key first. The way it applies those round-keys however is identical.

The sub_bytes() and inv_sub_bytes() functions

`Sub_bytes()` aims to bring non-linear diffusion into the encryption by substituting all bytes of the state-matrix with their multiplicative inverse value in the *Galois Field* (2^8) (abbr. GF (2^8)). This is achieved by considering the byte that is supposed to be substituted as the index of a 256-byte lookup table, which holds exactly that multiplicative inverse value in the GF (2^8) that corresponds to the index. This 256-byte lookup table, that accelerates the calculation of the multiplicative inverse of value x by simply precomputing it and saving it at the index x, is termed *Substitution Box* (abbr. S-box). The details of computation in the GF (2^8) are further outlined in the section covering `mix_columns()`, as acutal multiplications in the GF (2^8) is a regular part of the AES algorithm.

The inverse S-box for decryption is derived by finding the multiplicative inverse in GF (2^8) of each value in the regular S-box. This makes the inverse S-box therefore the regular S-box run in reverse, meaning that simply the index and the index corresponding value get switched. Listing 2.1 showcases the regular S-box and the simple byte substition that make up the `sub_bytes()` function, while the inverse S-box is listed in the appendix as listing A.2.

```c
void sub_bytes(uint8_t *state)
{
  static const uint8_t sbox[] = {
   0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
   0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
   0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
   0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
   0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
   0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
   0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
   0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
   0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
   0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
   0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
   0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
   0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
   0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
   0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
   0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16}

  for (int i = 0; i < 16; i++) {
    state[i] = sbox[state[i]];
  }
}
```

Listing 2.1: Implementation of `sub_bytes()` in C

The shift_rows() and inv_shift_rows() functions

As the state throughout the encryption should be considered a matrix as outlined in illustration 2.1, will `shift_rows()` perform a circular left shift on the bytes of each row, with each row shifting left differently. `Inv_shift_rows()` performs the shifts analogue to the right as the regular function performs them to the left. Note that `shift_rows()` and `inv_shift_rows()` do not perform a bitwise shift but rather a bytewise shift on the positions of the bytes in the matrix-state. In accordance to illustration 2.3, which displays both functions, will `shift_row()` leave the first row untouched, shifts the bytes of the second row one to the left, shifts the bytes of the third row two to the left and shifts the bytes of the fourth row three to the left. `Inv_shift_rows()` works analogue shifting the bytes to the right.

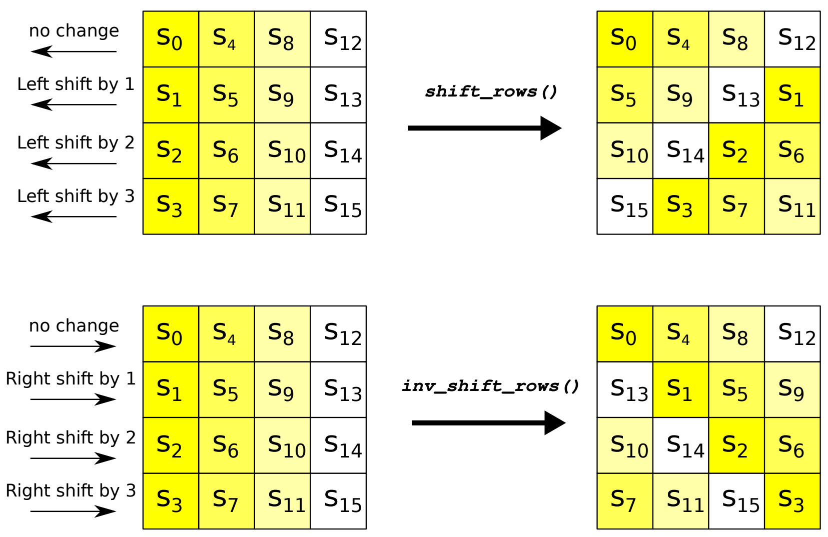

no change
Left shift by 1
Left shift by 2
Left shift by 3
shift_rows()
no change
Right shift by 1
Right shift by 2
Right shift by 3
inv_shift_rows()
S_0 S_4 S_8 S_{12}
S_1 S_5 S_9 S_{13}
S_2 S_6 S_{10} S_{14}
S_3 S_7 S_{11} S_{15}
S_0 S_4 S_8 S_{12}
S_5 S_9 S_{13} S_1
S_{10} S_{14} S_2 S_6
S_{15} S_3 S_7 S_{11}
S_0 S_4 S_8 S_{12}
S_{13} S_1 S_5 S_9
S_{10} S_{14} S_2 S_6
S_7 S_{11} S_{15} S_3

Illustration 2.3.: Functionality of `shift_rows()` and `inv_shift_rows()`

The mix_columns() and inv_mix_columns() functions

`Mix_columns()` is the primary source of diffusion in the matrix-state during the encryption, as `sub_bytes()` is primarily used as a nonlinear function to harden the encryption against parallelization in brute-force attacks [Pre98]. Both `mix_columns()` and `inv_mix_columns()` perform simply a matrix multiplication in the Galois Field (2^8) of the state-matrix and a special mixing matrix. When multiplying both matrices in the GF (2^8) does the multiplication of single elements follow a special algorithm, while the addition is performed by simply XORing both byte values. The regular `mix_columns()` and its inverse function differ only in the used special mixing matrix, both displayed in illustration 2.4, but are identical aside from that.

$$\begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \qquad \begin{pmatrix} \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} \\ \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} \\ \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} \\ \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} \end{pmatrix}$$

Illustration 2.4.: Mixing Matrices of the Encryption (left) and Decryption (right)

The special algorithm used most often for the multiplication of single elements in the GF (2^8) is the *Russian Peasant Multiplication Algorithm* shown in listing [Wik17d]. If the intricacy of performing the addition and multiplication in the GF (2^8) is considered, can the `mix_columns()` and `inv_mix_columns()` functions otherwise be regarded as regular matrix multiplications. A listing showcasing the actual reference implementation of `mix_columns()`, which also utilizes the `galois_mul()` function shown below can be found in the appendix as listing A.3.

```c
uint8_t galois_mul(uint8_t a, uint8_t b)
{
  uint8_t p = 0;     // p as the product of the multiplication

  while (b) {
    if (b & 1) {     // If b odd, then add corresponding a to p
      p = p ^ a;     // In GF(2^8), addition is XOR
    }
    if(a & 0x80) {   // If a >= 128 it will overflow when shifted left, so reduce
      a = (a << 1) ^ 0x11b; // XOR with primitive polynomial x^8 + x^4 + x^3 + x + 1
    } else {
      a <<= 1;       // Multiply a by 2
    }
    b >>= 1;         // Divide b by 2
  }
  return p;
}
```

Listing 2.2: Implementation of `galois_mul()` in C

The key_expansion() function

The AES key expansion serves to expand the initially provided key of 128-bit, 192-bit or 256-bit to a larger expanded-key in order for every call of the `add_round_key()` function to have its own round-key. Different round-keys rather than the same initially provided key are used to increases the security of AES. Though because it would mean an impractically large initial key if all round-keys would have to be supplied to the AES algorithm, does the key expansion deterministically compute all 11 to 15 required round-keys from a single 128-bit to 256-bit initial key. This single initially provided key therefore also serves as the first round-key. As the `add_round_key()` function is called 11 times for a 128-bit key, 13 times for a 192-bit key and 15 times for a 256-bit key, does the expanded-key have to store 176 bytes for the 128-bit key, 208 bytes for the 192-bit key and 240 bytes for the 256-bit key.

All three key expansion algorithms differ but are deterministic and solely based on the initially provided key. Therefore, if more than one block has to be encrypted or decrypted with the same key, the key only has to be expanded once. The same applies in regards to Encryption and Decryption, as both use the same key expansion algorithm is it only necessary to expand the key once.

Furthermore do the key expansion algorithms coincide in their basic procedure. First of all is the initially supplied key copied into the expanded key as the first round key. Iteratively, until all required round-keys are derived, is then the following round-key computated from the previous one. The three key expansion algorithms differ only in this step in the regard that each algorithm chooses a different 32-bit excerpt to be computed with a special equation. The remaining parts of a round-key is simply calculated by XORing different parts of the previous round-key. All the while is the calculation of the specially computed excerpts done by applying special functions such as `sub_word()`, `rot_word()` and `rcon_word()` as displayed in the example expansion of a 128-bit key in listing 2.3 and detailed in the paragraphs below. Listings of the remaining two key expansion algorithms can again be found in the appendix.

A very important remark is the assessment that the key expansion is barely parallelizable because the next iteration of a round-key is very dependent on the previous round-key. However is it only necessary to perform the key expansion once, suggesting a key expansion on the fastest processing core before passing the trivially parallelizable blocks together with the expanded key to the parallel hardware.

```c
key_128_expansion(uint8_t key[16])
{
  uint8_t expandedKey = new expandedKey[176]; // Create expandedKey of 176 bytes

  memcpy( expandedKey, key, 16);              // Initial key is basis to expand upon

  for (int round = 4; round < 44; round++) {  // 44 rnds of 32-bit value computation
    if (round % 4 == 0) {                     // Special computation every fourth rnd
      expandedKey[round * 4] = sub_word(rot_word(expandedKey[(round - 1) * 4]))
                             ^ rcon_word((round / 4) - 1)
                             ^ expandedKey[(round - 4) * 4];
    } else {
      expandedKey[round * 4] = expandedKey[(round - 1) * 4]
                             ^ expandedKey[(round - 4) * 4];
    }
  }
}
```

Listing 2.3: Key Expansion Algorithm for 128-bit Key in C

sub_word() is very similar to the previously explained `sub_bytes()` function of the block algorithm. While `sub_bytes()` has an input of one byte, does the `sub_word()` function take four bytes as input and replace these four bytes with the corresponding bytes of the S-box. It is important to emphasize that because the encryption and decryption use the same key expansion algorithms, the `sub_word()` function only ever uses the regular S-box, never the inverse S-box.

rot_word() takes four bytes as input and performs a one byte circular left shift, very similar to the `shift_rows()` function. Illustration 2.5 provides a visualization of the simple functionality of `rot_word()`.

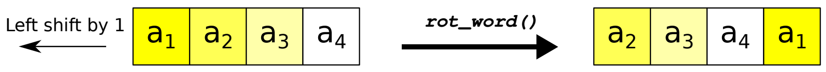


Illustration 2.5.: Visualization of the `rot_word()` Functionality

rcon_word() receives an input integer in the range of 0 to 9 while returning four bytes. The function will return the exponentiation of 2 to the given input in Rijndael's finite field [Ben12] as the first byte, while zeroing out the remaining three bytes. Listing 2.4 exhaustively illustrates this simple function.

```c
uint32_t rcon_word(uint32_t input)
{
  rcon[10] = {0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000,
              0x20000000, 0x40000000, 0x80000000, 0x1B000000, 0x36000000}

  return rcon[input];
}
```

Listing 2.4: Implementation of `rcon_word()` in C

2.1.2. Block Cipher Modes of Operation

In cryptography, a mode of operation states the algorithm according to which a symmetric block cipher (like AES) is securely applied onto multiple blocks of input. These modes of operation can differ significantly, ranging from applying the block cipher on each input block separately - as done in the ECB mode - to needing an *initialization vector* (abbr. IV) as a third input and enmeshing the encryption of every input block with the preceding input block - as dictated by the CBC mode. The CTR mode is a recent and very original mode of operation, in that it turns a block cipher encryption into a stream cipher that encrypts the plaintext one bit at a time instead of operating on blocks of fixed size.

ECB Mode - Electronic Codebook Mode

The Electronic Codebook mode of operation is the most straighforward block cipher mode of operation, in that it treats the encryption of every block in reclusion. The plaintext is divided into block-sized chunks, which in the case of AES is always 16 bytes as the only supported blocksize. Onto each block is then the encryption block algorithm applied in reclusion, without the influence of any other factor. Therefore the ECB mode is basically the vanilla version of the used block encryption algorithm - AES in this context. If the plaintext is not a multiple of 16 bytes - or more generally speaking a multiple of the block-size - then the last block needs to be padded. The simplest way to pad a block is to add null bytes until the block consists of the required 16 bytes, though more sophisticated padding schemes exist. However as padding has no significant influence on the speed of the encryption is it not relevant when determining achievable speed-up of an encryption through parallelization. Illustration 2.6 showcases the encryption and decryption scheme of the Electronic Codebook mode, summarzing the simplicity of this scheme.

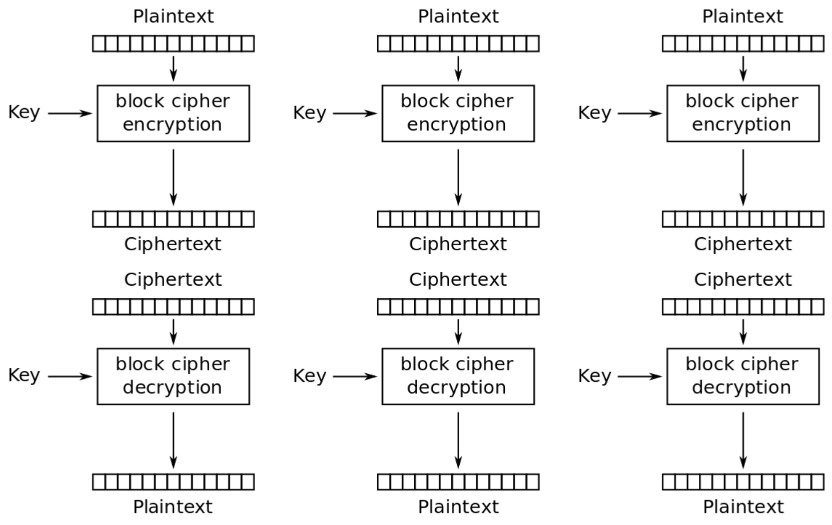


Illustration 2.6.: ECB Encryption Mode (Top) and ECB Decryption Mode (Bottom) [Wik17c]

Unfortunately though does the simplicity of ECB also create its biggest disadvantage. Identical plaintext blocks are encrypted into identical ciphertext blocks and because ECB does not further mix up the created ciphertext but rather retains the ciphertext blocks in the same order as the plaintext blocks, can single blocks easily be identified and checked for alikeness. Thus, ECB does not hide data patterns well and is generally not recommended because of it. This flaw of ECB is most visible when encrypting monotonous pictures as visualized in illustration 2.7. The monotonous white background and uniformly coloured form of *tux* makes for a very obvious data pattern and is therefore very susceptible to ECB's flaw. The CBC mode comes to the rescue in this regard as it creates sufficient pseudo randomness to effectively encrypt plaintext with data patterns.

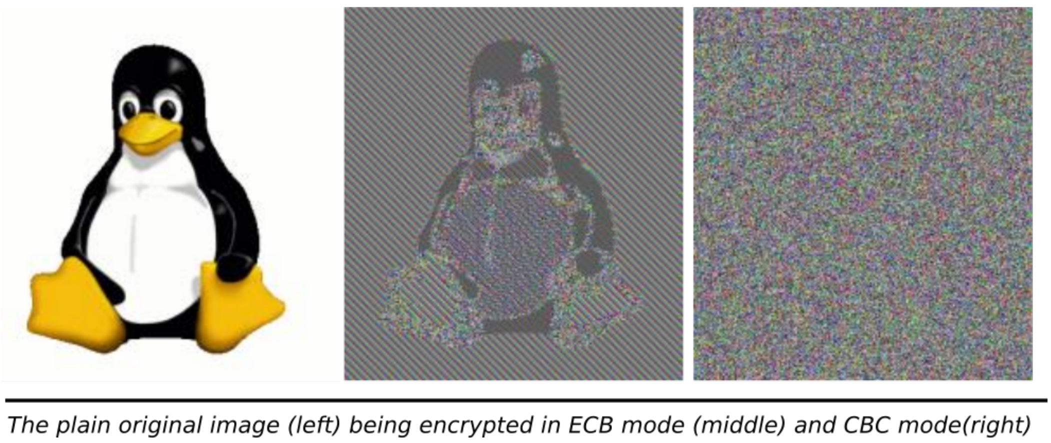

The plain original image (left) being encrypted in ECB mode (middle) and CBC mode(right)

Illustration 2.7.: Illustration of the ECB Data Pattern Flaw [Wik17e]

Nonetheless does ECB provide excellent potential for GPU parallelization, as AES-ECB is trivially parallelizable by simply passing the job of encrypting different blocks to different compute units. Doing so creates no interdependence between the compute units at all, allowing the speed-up to scale in unison with the number of available compute units.

CBC Mode - Cipher Block Chaining Mode

The Cipher Block Chaining mode of operation is together with the CTR mode one of the two block cipher modes of operation considered most secure and recommended by such renowned cryptologists as Bruce Schneier or Nils Ferguson [Lip00]. The basic principle of CBC mode is to process each block one after another by first XORing the ciphertext of the preceding block with the plaintext of the block that is about to be encrypted before regularly encrypting it with the block cipher algorithm. This way is each ciphertext block dependent on all plaintext blocks processed up to that point, which in return adds enough pseudo-randomness to effectively encrypt plaintexts with a high rate of data patterns. Despite this dependence and pseudo-randomness however is at least the decryption parallelizable, as in decryption each block only has to be XORed with the easily identifiable ciphertext of the preceding block - see illustration 2.8.

As of course the first block to encrypt does not have a preceding ciphertext with which the first block can be XORed, a so-called *initialization vector* is used instead. This initialization vector is of regular block-size and must vary for different keys as it would otherwise provide side-channel information enabling an easier reverse-engineering of the used key. However the IV does not need to be secret, but rather needs to be shared with the communication partner to enable him to successfully decrypt the first ciphertext. Not sharing the IV expresses itself in not being able to decrypt the first block of ciphertext, nonetheless can subsequent ciphertext-blocks be correctly decrypted as the necessary preceding ciphertexts are obviously present.

Due to CBC being a block cipher mode of operation, it operates on full block-sizes and can therefore require padding in the last block. CBC's main drawback is the necessary sequentiality in its encryption, making the encryption impossible to parallelize. The decryption however is parallelizable as outlined, though it is significantly more elaborate and resource

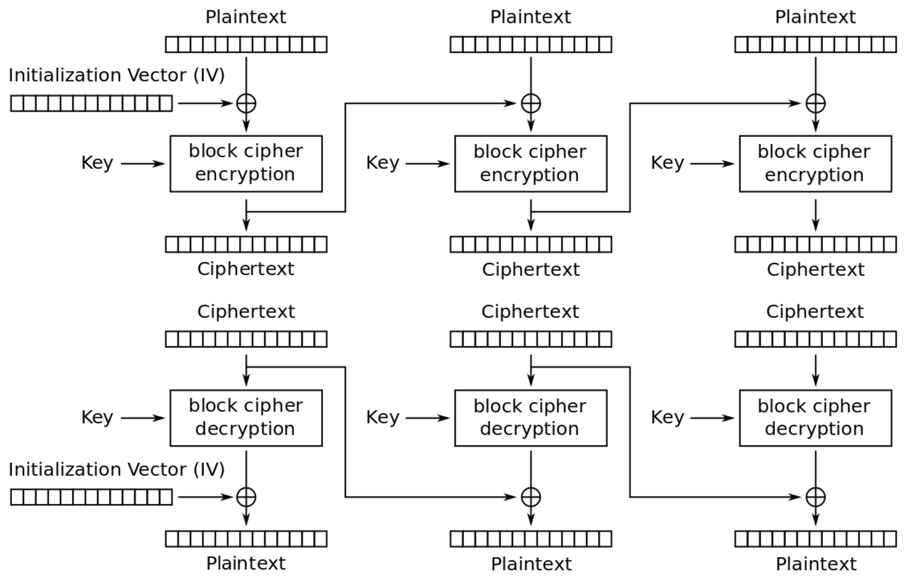


Illustration 2.8.: CBC Encryption Mode (Top) and CBC Decryption Mode (Bottom) [Wik17a]

intensive as the trivially parallelizable decryption of the ECB mode. Even though AES-CBC is therefore only partially parallelizable, due to its good security properties and the resulting popularity should its potential of parallelizability on IoT devices be further explored.

CTR Mode - Counter Mode

The Counter mode of operation is unconventional in that it turns a block cipher encryption into a stream cipher encryption. A stream cipher encryption typically combines every bit with a bit of the *keystream*. A keystream is a source of pseudorandom bits that is necessary for a stream cipher. Pseudorandomness in the keystream means that it seems to be a random sequence of bits but actually isn't, as the same pseudorandom keystream needs to be generated again in decryption. Therefore is the keystream generation necessarily deterministic.

In the CTR mode, the keystream results from block cipher encrypting a 16-byte initialization vector combined with a counter - a simple value that is incremented for each consecutive block the input is applied to. Because the counter is incremented by one (sufficient and most popular) and then added to the IV, the input for each block encryption is slightly different. As it is a property of a good block cipher encryption algorithm like AES to have high diffusion in the ciphertext on even minutely different inputs, the results of this first encryption of the IV plus counter serve as an endless pseudorandom keystream. This pseudorandom keystream is therefore deterministic and easily replicable knowing the IV and key - a necessary requirement for the keystream. Though unlike in CBC mode will the ignorance of the IV render the recipient of the ciphertext unable to decrypt it. In accordance with CBC however is it necessary to have different IVs for different keys, as it would otherwise provide side channel information.

The ciphertext in CTR mode is eventually generated by XORing the keystream with the plaintext bit for bit. As the XOR operation is reversible by repeated application, is decryption done by simply generating the same keystream but then XORing it with the ciphertext - resulting in the original plaintext. This scheme makes padding of the original plaintext unnecessary as each bit is XORed independently of any surrounding bits, enabling the stream cipher to securely stop whenever it runs out of plaintext. It is important to note that both encryption and decryption use the encryption block algorithm when generating the keystream. Illustration 2.9 will give a good overview and recap of the CTR mode.

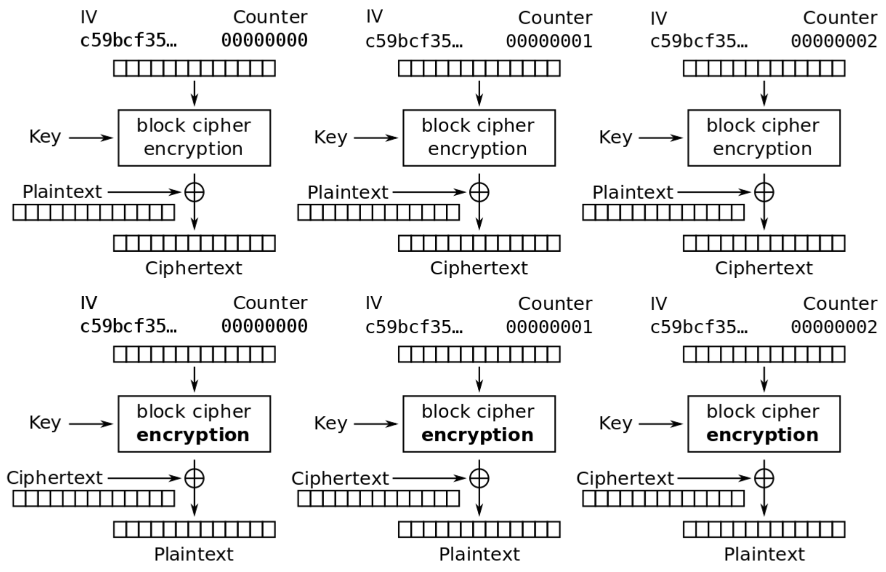


Illustration 2.9.: CTR Encryption Mode (Top) and CTR Decryption Mode (Bottom) [Wik17b]

Although unconventional, the CTR mode of operation is highly secure and more importantly parallelizable in encryption and decryption, making it suitable for acceleration tests on GPUs of mobile devices. The parallelization is possible because the deterministic property of the keystream generation enables random-reads and therefore compartmentalisation, which in turn enables the distribution of the encryption to different compute units.

2.2. The Raspberry Pi

The Raspberry Pi 1 model B used in this thesis (see illustration 2.10) is a credit card-sized single-board computer. Its GPU is far outperforming its CPU according to their specified theoretical maximum performance, making the Raspberry Pi well suited for the attempt of AES acceleration by GPU on an IoT device. Developed in the United Kingdom and released in mid 2012, it quickly grew to being the most popular single-board computer [Ltd16]. When both follow up models, the Raspberry Pi 2 and the Raspberry Pi 3, are included, the sales total to over 11 million units up to November 2016 [Mag17].

2.2.1. The Raspberry Pi System-on-Chip

The Raspberry Pi 1 has a Broadcom manufactured System-on-a-Chip - the BCM2835 - that accomodates the CPU, the GPU and 512 MB of SDRAM on a single die on the center

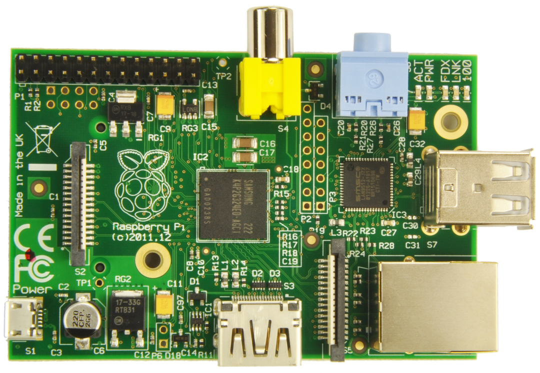


Illustration 2.10.: Raspberry Pi 1 model B [ELi17]

of the board. The featured CPU is a 32-bit RISC ARM Processor based on the ARMv6 architecture, namely the ARM1176JZF-S. The standard clock frequency is set at 700 MHz although it is possible to overclock it to a frequency of up to 1 GHz at the cost of serious instability [Mol15]. The CPU also has access to two caches - the first level one having 16 KB and the second level one having 128 KB, although the second level one is shared with the GPU. Equipping the SoC with 512 MB of SDRAM was standard for all Raspberry Pis that have been produced following the 15. Oct of 2012 as the previous 256 MB of SDRAM were insufficient [Far16]. The Raspberry Pi used in this thesis is equipped with 512 MB of SDRAM. All in all does the theoretical maximum performance of the CPU add up to 1.4 Single Precision GFLOPS - calculated by multiplying the number of cores (1), the provided clock frequency (0.7 GHz) and the number of operations per cycle (2) according to ARM's specifiction of the chip [ARM17a].

The GPU on the other hand is theoretically far more potent than the CPU. The GPU is a VideoCore IV model that was developed by Alphamosaic Ltd and is featured in all three Raspberry Pi models. It provides a clock frequency of 250 MHz on each of its 12 parallelized processing units and features a basic pipeline functionality, which brings its theoretical maximum performance to an impressive 24 Single Precision GFLOPS according to its hardware specifications [Bro12].

Given the enormous difference in theoretical maximum performance between the CPU and the GPU by a factor greater than 17, does the use of the GPU promise a great potential for speed-up. This however can only be achieved under the assumption of sufficient and well used access to the GPU as well as an effectively parallelizable algorithm - which is given for AES when used in combination with the trivially parrallelizable block cipher mode ECB. A sufficient - although in many aspects still lacking - access and documentation of the GPU has been made possible in early 2014. In March 2014 did Broadcom and the Raspberry Pi foundation release the *Architecture Reference Guide* (abbr. ARG) of the VideoCore IV GPU in conjunction with the GPU kernel driver for the BCM2835 architecture. Though each of the three Raspberry Pi iterations feature the VideoCore IV GPU for which the ARG is intended, did the GPU kernel driver only allow access to the Raspberry Pi 1 as the Raspberry Pi 2 and 3 feature slightly different SoCs (BCM 2836 and BCM 2837 respectively). The sourcecode of the VideoCore IV GPU firmware or of the supported graphics programming interfaces (OpenGL-ES, OpenVG, OpenMAX) or schematics of the bare-metal GPU architecture are however still undisclosed.

2.2.2. The Raspberry Pi GPU Architecture

The VideoCore IV is the fourth generation of the VideoCore series GPUs developed by Alphamosaic and the second generation providing a 3D system architecture. At its heart it provides 12 *Quad Processing Units* (abbr. QPUs) intended for floating-point shading of which each is based on a 32-bit big endian architecture that is clocked at 250 MHz [ARM17b].

Four QPUs are organized into one group termed a *slice* of which there are a total of 3 slices - see illustration 2.11. This organization of the QPUs is intended as a compromise between flexibility and cost, as each of the three slices has access to resources that are independent from the other slices, do however have to be shared by all four QPUs within the slice. Each slice is fitted with its own instruction cache (abbr. icache) enabling three completely independently parallelized algorithms within the GPU. Furthermore has each slice acess to two *Texture Memory Lookup Units* (abbr. TMUs), one Uniforms Cache, a Special Function Unit (abbr. SFU) and hardware specifically intended for OpenGL use such as attribute and coefficient caches. The TMU and Uniform Cache are both similar in that both of them utilize the L2 cache that is shared with the CPU and is also used to transmit the instructions to the slices. However while the Uniforms Cache is a strict FiFo queue intended only for small 32-bit messages - such as the VPM or TMU memory address assigned to each QPU - does the TMU allow random reads of large amounts of L2 memory, only limited by the L2 cache size of 128 KB. Yet it is not possible for the QPUs to write into the L2 cache. Lastly is there the Special Function Unit, intended for mathematical functions such as the square operations, logarithmic operations or exponentiations.

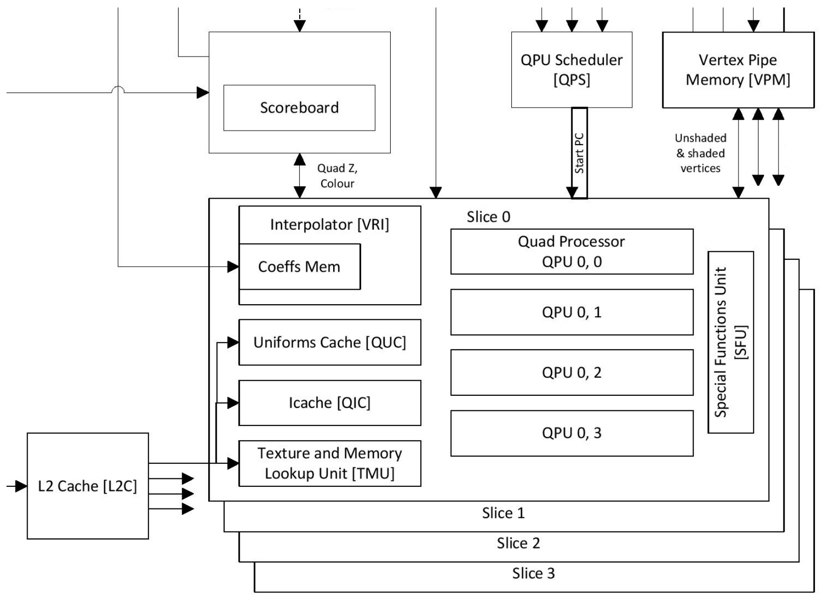


Illustration 2.11.: VideoCore IV 3D System Block Diagram [Bro13]

Shared by all QPUs is also the access to the *Vertex Pipe Memory* (abbr. VPM), the 48 KB large cache intended for saving and retrieving intermediate values or providing a cache from which data can be read back to the CPU. The VPM is controlled through the *Vertex*

Cache Manager (abbr. VCM), which assigns 4 KB of VPM to each QPU by default. Further influence on the VPM is exerted by the *QPU Scheduler* (abbr. QPS), which manages the acquisition and release of the shared global VPM-mutex - a necessity because the hardware used to access the VPM is shared in between slices. Should the mutex be acquired will it stall all access by other QPUs until the mutex is released after which the QPU scheduler will decide for the next QPU that is granted access. For further inter-process synchronisation such as signaling a completed calculation are 16 4-bit counting semaphores available.

Furthermore is the QPU Scheduler responsible for the automatic scheduling of the QPUs and for possible multithreading - two threads are possible within each QPU. The QPU scheduler itself is part of the closed source firmware blob though its behaviour can be influenced through *Control Lists* detailed beginning in section 8 of the ARG (source). Control lists and multithreading however were not needed for the implementation of the general purpose assembler or the implementation of AES in GPU specific assembly.

The QPU Core Pipeline

A single QPU is a 32-bit architecture that provides two *Arithmetic Logical Units* (abbr. ALUs) operating on 32-bit registers, one branching unit, one load unit allowing to write arbitrary 32-bit values directly to registers and multiple I/O registers to access memory, special function units or interprocess communication. The two ALUs are referred to as the *add pipeline* (or add unit) on the one hand and the *mul pipeline* (or mul unit) on the other hand, although the add unit is more of a *non-multiply* unit as it is also capable of performing shifts, rotates or the counting of leading zeros. Both ALUs can operate on integer and floating point data of 32-bit but are also capable of performing dedicated 8-bit vector operations such as vector-add, vector-sub, vector-max or vector-mul. Because each ALU still can only read from two input registers and write to one output register, it is necessary that the 8-bit values on which the vector operation should be performed on are encoded within the same 32-bit register. This functionally is in accordance to the QPU's internals, as each QPU consists of a 4-way Single-Instruction-Multiple-Data processor that is multiplexed four times - justifying the term *Quad* processing unit. The QPU therefore processes a 32-bit register by working on 8-bit quantities. This enables the QPU through the *pack & unpack* functionality to perform all pipeline operations on the 32-bit registers as if the registers contain two independent 16-bit values or four independent 8-bit values.

In addition is each QPU outfitted with 64 physical 32-bit registers that are divided into register files A and B and can be used by both ALUs as read and write registers with little restriction. Further 64 virtual registers are provided for access to I/O operations. Additionally is each QPU fitted with four general-purpose accumulators and two special-purpose accumulators. The general-purpose accumulators are intended to be heavily used for the bulk of the operations, as they do not suffer from certain signficant restrictions of the physical registers outlined in chapter 3. The special-purpose accumulators on the other hand are designed to hold the results of calls to the TMU or SFU and are only used infrequently.

A good summary of the dual ALU architecture, the complete QPU core pipeline and the presence of physical register files and accumulators is given by illustration 2.12. Evident by the relatively simple design is the absence of features found on regular CPU cores such as register forwarding or branch prediction.

A very important hardware constraint partly deductable by inspection of the upper illustration is the interdependence of the add and mul units in regard to their input registers.

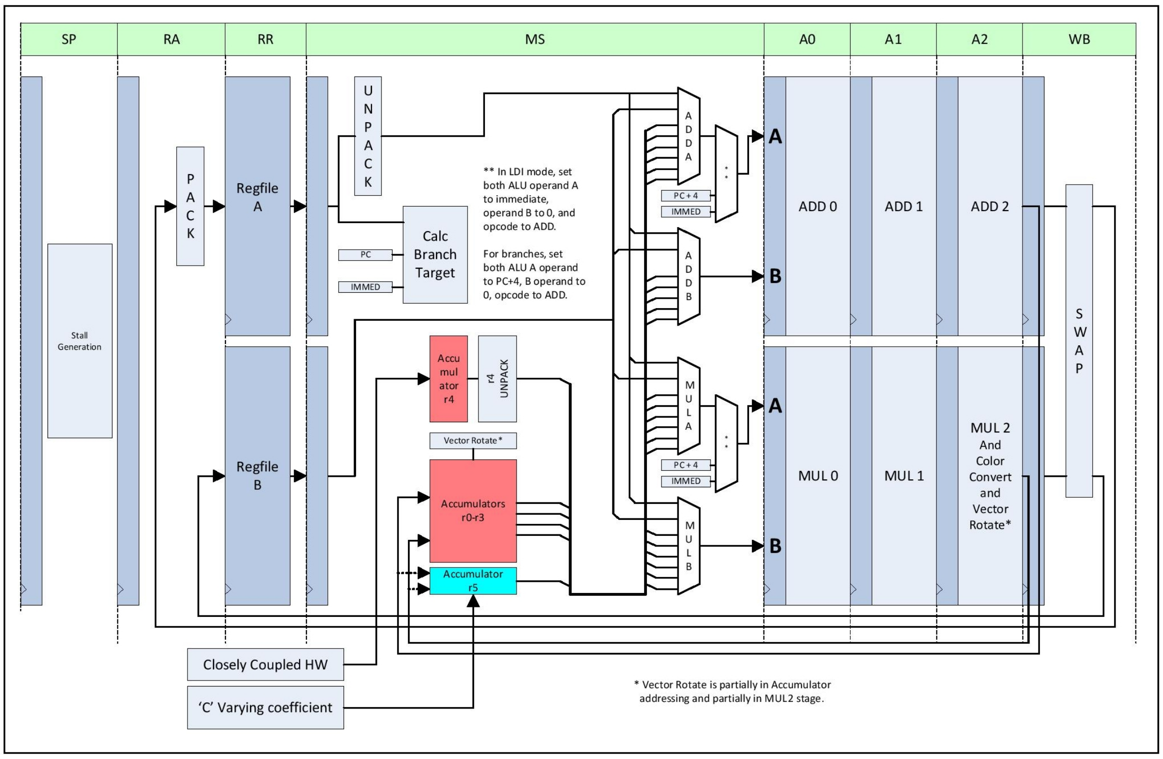

SP
RA
RR
MS
A0
A1
A2
WB
Stall Generation
PACK
Regfile A
Regfile B
UNPACK
PC
IMMED
Calc Branch Target
** In LDI mode, set both ALU operand A to immediate, operand B to 0, and opcode to ADD.
For branches, set both ALU operand to PC+4, B operand to 0, opcode to ADD.
Accumulator r4
r4 UNPACK
Vector Rotate*
Accumulators r0-r3
Accumulator r5
ADDA
ADDB
MULA
MULB
PC+4
IMMED
A
B
ADD 0
ADD 1
ADD 2
MUL 0
MUL 1
MUL 2 And Color Convert and Vector Rotate*
SWAP
Closely Coupled HW
'C' Varying coefficient
* Vector Rotate is partially in Accumulator addressing and partially in MUL2 stage.

Illustration 2.12.: VideoCore IV QPU Core Pipeline [Bro13]

The first of the two read-input registers for each ALU has to always be from register file A or has to be an accumulator. However if register file A is read in the first read-input register by one of the ALUs, the identical register has to be used as the first read-input register in the other ALU. The same holds true for the second-input register, except that the second input register has to be from register file B, an accumulator or a *small immediate*. Though this whole constraint regarding the ALU instruction grammar will be extensively explained in the upcoming chapter 3.1, is it important to retrace those constraints to the QPU hardware. Therefore does chapter 3.1 often reference illustration 2.12. However just given the fact that both ALUs can not address different registers from register file A in the same instruction is putting an enormous strain on the flexibility of the ALUs resulting in a major damper on the actually achievable maximum performance as this is not considered when calculating the theoretical maximum performance of 24 SP GFLOPS. The constraint itself though is due to the QPU only having one registerbank A, which can only be read once at a time - well visualized by the QPU core pipeline illustration.

Worth mentioning when talking about the QPU core pipeline, though more relevant when going into detail about the assembler, is the presence of the NCZ bit-flags in each ALU. NCZ bit-flags are three bits representing the events of a negative (N) result, a carry-over (C) event or a result equaling zero (Z) in the ALU instruction executed last. Depending on these flags can the following ALU operation or branch be made conditional, which most importanly can lead to genuine conditional branches that make complex algorithms possible.

2.3. Related Work and Summary of AES on the GPU

Given that block ciphers are relatively straightforward to parallelize and GPUs have long been recognized as an instrument to perform highly parallel general-purpose computing upon, there already exists a lot of research regarding the parallelization of AES on the GPU. Most AES parallelization on the GPU is done with high-level graphics programming interfaces such as CUDA, Direct3D, OpenGL and OpenCL, which can yield excellent results [Iwa10]. The GPU of the Raspberry Pi however does support non of those APIs, as outlined in the preceding chapter. The graphics programming interfaces supported by the Raspberry Pi are very high level, tailored highly to graphics programming and are therefore not very flexible, most importantly don't serve as the basis of any general-purpose parallelization research. The AES block cipher algorithm on the other hand is highly byte oriented, as all of the four different funtions that make up AES operate on single bytes.

The Architecture Reference Guide published by Broadcom alongside a basic kernel driver for the Raspberry Pi 1 model B in 2014 enables the creation of a GPU specific assembly language, which in turn seems well suited for the parallelization of the byte oriented AES algorithm. Though because of the inavailability of sourcecode for the supported graphic APIs, the GPU's firmware, an existing assembler or even for simple bare-metal schematics is the use of the published documents tremendously cumbersome. Therefore is existing research very limited and only exists in form of blogposts published by Eric Lorimer and Pete Warden [Lor14a] [War14].

Eric Lorimer started working on an inital assembler and driver access for the GPU in 2014 by utilizing only Broadcom's published hardware ARG. His intention was to speed up the computation of SHA-256 on the Raspberry Pi GPU and by his own statement was able to accelerate the computation of SHA-256 by 1430% compared to his own CPU implementation of SHA-256 in C++ [Lor14b]. Unfortunately though is the speed-up stated by him not reproducible, as the results of his implementation using all 12 compute units deviate from the correct results of SHA-256. However addressing only 1 of the 12 compute units lead the GPU implementation to work as intended and the resulting time measurements show promise.

Pete Warden based his work upon Eric Lorimer and realized the parallelization of a matrix multiplication on the GPU. The deployment of this parallelized implementation into a Tensorflow application yields a reproducible speed-up of a 660%. Likewise show his results promise as the obtainable speed-up is achieved by utilizing only 8 of the 12 compute units since according to him the bottleneck of accessing shared memory prevents an effective use of all 12 compute units.

The byte-oriented AES algorithm therefore shows promise to be effectively implemented on the GPU with the same GPU specific assembly language made possible by the ARG. Possible implementations of AES however include a statement about the utilized block cipher mode of operation. The important properties of the block cipher modes of operation introduced in section 2.1.2 are summarized in table 2.1. Although all three introduced block cipher modes differ significantly is the potential speed difference for aspects listed in the table not simply binary.

Even though the decryption is parallelizable in all three modes of operation is the ECB mode by far the fastest as the memory management - a significant aspect when using the GPU for general-purpose computing - is as simple as possible. But at the same time does the CTR mode despite its necessarily more sophisticated management of assigned memory

Aspect	**ECB**	**CBC**	**CTR**
Secure against data-patterns	No	Yes	Yes
Encryption parallelizable	Yes	No	Yes
Decryption parallelizable	Yes	Yes	Yes
Random-Read access	Yes	Yes	Yes
Initialization vector necessary	No	Yes	Yes

Table 2.1.: Comparison of Introduced Block Cipher Modes of Operation

promise a higher performance than CBC, even in decryption. This is due to the trivial parallelizability of the CTR mode but also because of the smaller required amount of memory than required by the CBC mode. All in all though is the ECB mode the best choice to study the GPU's potential and it is also the mode of operation used most common in the related work of general AES parallelization. Moreover is the ECB mode of operation, as the most simple and most easily parallelizable mode, the best choice in showcasing the parallelization potential of mobile GPUs.

3. QPU Assembly Language and Assembler

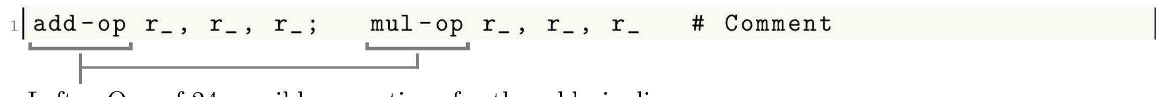
The language intended for programming single QPUs - in earlier chapters called GPU specific assembly - was named QASM, short for QPU ASM. It is realized by an assembler which signficantly improved upon the earlier draft by Eric Lorimer and is introduced in section 3.2. All of those assembled instructions and buffers are brought onto the GPU by the kernel driver. The complicated but now well documented addressing of this kernel driver and how it can possibly be improved is addressed in the last section of this chapter.

3.1. QASM - The RPi QPU Assembly Language

The Assembly language for the Raspberry Pi QPUs - termed QASM - is designed to be as closely implemented in accordance to the QPU's architecture and instruction-encoding as possible. Each seperate QPU instruction is written into a seperate line of an instruction file specified by the file ending `.qasm`. An instruction line can either specify an *Arithmetic Logical Unit* instruction, a *Load Immediate* instruction, a *Branch* instruction, a *Semaphore* instruction or simply a comment. While comments are simply invoked by a single `#`, are the remaining instructions complex and the following sections are dedicated to them. Though to give a general overview it can be said that the ALU instructions perform calculations on the QPU's 128 different registers, while branching instructions can conditionally determine the control flow. Load immediate instructions simply allow for a direct assignemnt of certain values to certain registers while semaphore instructions enable inter-process communication. The language is interpreted by its dedicated assembler and send to the GPU via the kernel driver - of which the usage of both is explained in their respective sections within this chapter.

3.1.1. ALU Instructions

The dual pipeline architecture, is enabling the execution of one add operation and one mul operation in each cycle. Therefore do instructions for this *Arithmetic Logical Unit* all follow certain patterns, consisting first of the statement of operation for the add pipeline, then the statement of operation for the mul pipeline. Both operations are seperated by a semicolon, first state their exact operation type, then state the write-register and finally state the two read-registers, which represent the arguments to the specified operation type. Illustration 3.1 and 3.2 portray the elements of such a general description of a single ALU instruction line.



```
add-op r_, r_, r_;   mul-op r_, r_, r_   # Comment
```

Left: One of 24 possible operations for the add pipeline
Right: One of   8 possible operations for the mul pipeline

Illustration 3.1.: Structure of a Single ALU Instruction - The Pipeline Operations

```
add-op r_, r_, r_;    mul-op r_, r_, r_    # Comment
```

Add pipeline instruction ends with semi-colon and an arbitrary number of blanks

Mul pipeline instruction ends after second read-register. No input read after that by Assembler

Illustration 3.2.: Structure of a Single ALU Instruction - Non-encoded Elements

ALU instructions also need to follow a set of necessary grammar rules, which stem from hardware constraints of the QPU pipeline and the special instruction encoding of ALU instructions - detailed in the upcoming chapter about the assembler. The type and order of operations executed by the pipelines does not underlie any restrictions, while the write-registers do have to follow a minor constraint regarding the grammar. Both pipelines have to write the result of their respective operation into a different register file. As outlined in chapter 2.2.2 does each QPU have two register files termed register file A and register file B, with each providing 64 registers. Register file A is addressed by using `ra#`, with `#` ranging from 0 to 63 and representing the number of the register intended to address. Analogue addressing with register file B, except the register is specified using `rb#`. To recapitulate is it therefore necessary to state a `ra#` register as the write-register for one pipeline, and a `rb#` register as the write-register for the other pipeline - visually summed up in illustration 3.3.

```
add-op r_, r_, r_;    mul-op r_, r_, r_    # Comment
```

Write-register, allowed to adress register file A or B
Not allowed to adress same register file - one has to adress register file A, the other register file B

Illustration 3.3.: Structure of a Single ALU Instruction - The Write-registers

The read-registers on the other hand follow a very strict set of rules. First of all though is it necessary to establish a basic understanding of possible read-registers. In general are accumulators, small immediates and registers from register file A or B allowed to serve as possible read-registers. Accumulators exist as a seperate small registerbank in the QPU core pipeline, are addressed by using `r#` and serve to be used in the bulk of QPU operations for reasons listed in the paragraph below. Although there exist a total of 6 accumulators, are only the accumulators ranging from `r0` to `r3` intended for general-purpose use while the accumulators `r4` and `r5` serve only as read-registers to fetch results from calls to special function units. Small immediates in turn are only allowed to be used in the second read-register and allow for a small integer value ranging from 0 to 63 to be the second argument of an operation. The small immediate itself is denoted as a decimal or hexadecimal number, with $0x3f_{16}$ being the largest small immediate. By allowing small immediates in the ALU instruction encoding is it unnecessary to go through the additional effort of invoking a seperate *load immediate* instruction just for a simple integer value as the second argument of an operation.

To return to the significant constraints of the read-registers is it first necessary to state that the constraints limit themselves to the respective first read-registers and to the respective second read-registers. So the contents of the first read-registers of the add and mul pipelines exert absolutely no influence over the contents of the second read-registers and vice versa. Instead is the first read-register of the add pipeline exerting constraints over the possible content of the first read-register of the mul pipeline. Analoguesly do the second read-registers exert constraints over each other.

Possible contents for the first read-registers are registers from register file A or accumulators. Small immediates or registers from register file B are not allowed as content for the first read-register. If both first read-registers address register file A, then the addressed register must be identical in its adressed number. If however one or even both first read-registers use an accumulator, all constraints of the first read-registers are dropped.

```
add-op r_, r_, r_;   mul-op r_, r_, r_   # Comment
```

First read-register, allowed to be register file A or accumulator
If both first read-registers adress register file A, they have to address the identical register
No restrictions if accumulators are used

Illustration 3.4.: Structure of a Single ALU Instruction - The First Read-registers

Possible contents for the second read-registers are registers from register file B, accumulators or small immediates. Registers from register file A are not allowed as content for the second read-registers. When one second read-register addresses register file B or uses a small immediate, then the other second read-register has to have the identical content. In other words, when one second read-register addresses register file B, the other second read-register has to address the identical register. When one second read-register uses a small immediate, the other second read-register has to encode the same small immediate. Though a shared characteristic with the first read-registers is that when one or even both second read-registers use an accumulator, all constraints of the second read-registers are dropped. As illustration 3.4 visually summarizes the constraints on the first read-registers, does illustration 3.5 summarize them for the second read-registers.

```
add-op r_, r_, r_;   mul-op r_, r_, r_   # Comment
```

Second read-register, allowed adress register file B, accumulator or small immediate
Not allowed for one to adress register file B while the other uses a small immediate
If both second read-registers are of register file B, they have to address the identical register
If both second read-registers are small immediates, they have to encode the same small immediate
There are no restrictions if accumulators are used.

Illustration 3.5.: Structure of a Single ALU Instruction - The Second Read-registers

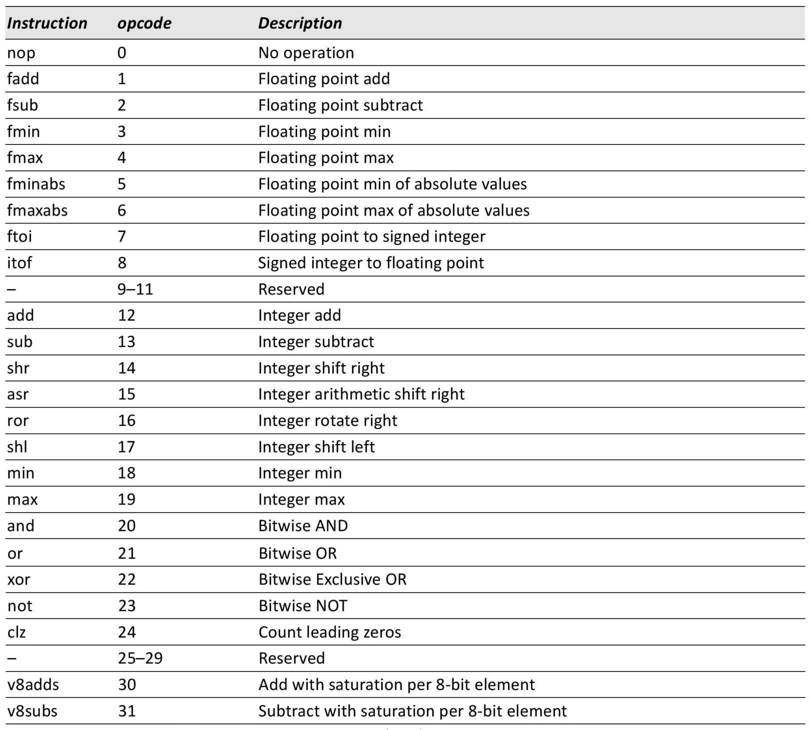
Yet another very important constraint on the QPU Assembly Language and a main contributor to the importance of accumulators is the fact that values written into regular physical registers (the first 32 registers of each register file) are not available to be read in the immediately following instruction. In other words can a register not be used as a read-register if it was used as a write-register in the previous instruction. The accumulator registers do not suffer from this restriction and can be used in the immediately following instruction - an architecture constraint that is also visually supported by illustration 2.12 of the QPU Core Pipeline. The illustration of the QPU pipeline shows that accumulator registers are updated independently and faster than physical registers.

Furthermore do the accumulators allow much more flexibility when encoding ALU instructions, as extensively explained in upper paragraphs detailing the restrictions on the read-registers. Those two powerful advantages of accumulators over regular physical registers makes them ideally suited for executing the bulk of fast short-termed operations and in return makes the physical registers well suited for saving medium-termed values.

Lastly it is important to note that although the QPU is a dual-pipeline architecture the language features a simple abbreviation of the mul pipeline instruction in case it should perform no operation. Instead of writing out a whole dummy operation for the mul pipeline to perform, a simple `nop` stated as the mul operation is sufficient for dummy write- and read-registers to be inserted by the assembler. This simple abbreviation is used extensively for the mul pipeline to have no effect in the instruction cycle, as most algorithms are intended for single-pipeline CPU usage.

Add and Mul Pipeline Operations

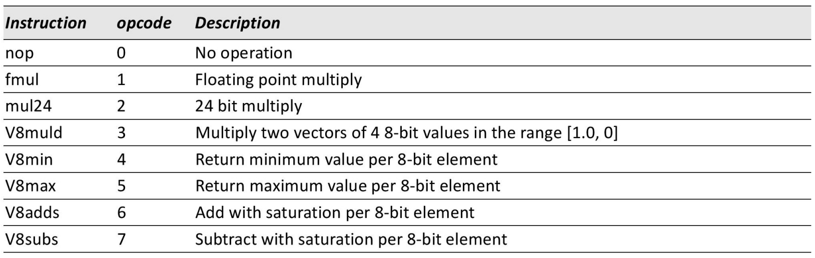
The add pipeline of the QPU is rather a referential nickname, as this pipeline is capable of many non-adding operations and can therefore also be referred to as the non-multiply-pipeline. Still, the add pipeline is capable of 24 different operations of which 7 serve floating point values and 9 serve integer values. The last 2 listed operations are suited for 8-bit vector data and the rest will perform bitwise operations. All of these operations are executable within 1 cycle and can be invoked by their exact name as the *add-op* in an ALU instruction.



Instruction	*opcode*	*Description*
nop	0	No operation
fadd	1	Floating point add
fsub	2	Floating point subtract
fmin	3	Floating point min
fmax	4	Floating point max
fminabs	5	Floating point min of absolute values
fmaxabs	6	Floating point max of absolute values
ftoi	7	Floating point to signed integer
itof	8	Signed integer to floating point
–	9–11	Reserved
add	12	Integer add
sub	13	Integer subtract
shr	14	Integer shift right
asr	15	Integer arithmetic shift right
ror	16	Integer rotate right
shl	17	Integer shift left
min	18	Integer min
max	19	Integer max
and	20	Bitwise AND
or	21	Bitwise OR
xor	22	Bitwise Exclusive OR
not	23	Bitwise NOT
clz	24	Count leading zeros
–	25–29	Reserved
v8adds	30	Add with saturation per 8-bit element
v8subs	31	Subtract with saturation per 8-bit element

Illustration 3.6.: Operations Executable by the Add Pipeline [Bro13]

The mul pipeline on the other hand is capable of 8 different operations of which 5 serve 8-bit vector data, the rest being a simple nop, a floating point multiply and an integer multiply. All of these operations are also executable within 1 cycle and can be invoked by their exact name as the *mul-op* in an ALU instruction.



Instruction	opcode	Description
nop	0	No operation
fmul	1	Floating point multiply
mul24	2	24 bit multiply
V8muld	3	Multiply two vectors of 4 8-bit values in the range [1.0, 0]
V8min	4	Return minimum value per 8-bit element
V8max	5	Return maximum value per 8-bit element
V8adds	6	Add with saturation per 8-bit element
V8subs	7	Subtract with saturation per 8-bit element

Illustration 3.7.: Operations Executable by the Mul Pipeline [Bro13]

3.1.2. Non-ALU Instructions

Load Immediates

The Load Immediate instruction is one of the most important Non-ALU instructions. It is a necessary instruction when loading 32-bit configuration values for memory setup, when loading loop counter values or simply when the parallelized algorithm requires an integer larger than the maximum small immediate value of 63_{10}. The instruction is invoked by its abbreviation *ldi*, followed by one or two write-registers and a final statement of the integer that is supposed to be written into the specified registers. The integer itself can either be stated as a hexadecimal or a decimal number, but can not exceed 32-bit in size.

As the Load Immediate instruction is not an ALU instruction it is not associated with any of the two pipelines, but takes instead the whole line of instruction. Though the constraint of the ALU instruction applies in that if the integer is saved to two write-registers, the register files of both write-registers have to be different. If the write-registers are physical registers then the constraint of unavailability of the written value in the next instruction also applies. In listing 3.1 below are three examples with short descriptions that showcase the simple use of the instruction.

```
## Load value 0x12345678 to register ra1
ldi ra1, 0x12345678;

## Load value 0x00401200 to VPMVCD_WR_SETUP register rb49 and register ra2
ldi ra2, rb49, 0x00401200;

## Load value 0x0000f0f0 to TMU0_s register rb56 and register ra3
ldi rb56, ra3, 0x0000f0f0;
```

Listing 3.1: Examples for the Load Immediate Instruction

Branches

Branches are invoked by the abbreviation *brr* followed by different arguments depending on which of the possible types of branches is intended. The unconditional branch displayed in listing 3.2 will always be executed and can be identified by a simple missing of a *conditional specifier*. A conditional specifier subjects the execution of the branch to the fullfilling of the stated condition, which in turn is dependent upon the previous ALU instruction. This again leads to the convention that the conditioning ALU instruction is placed right above the associated conditional branch. An example for the conditional branch can be seen in listing 3.3, while all possible conditions and their corresponding conditional specifiers are displayed in table 3.1.

```
## Load ra0 with 0
ldi ra0, 0x00000000

## Always branch to example_label
brr ra39, rb39, example_label
nop ra39, ra39, rb39;        nop
nop ra39, ra39, rb39;        nop
nop ra39, ra39, rb39;        nop

add ra0,  ra0,  1;           nop

example_label:

add ra0,  ra0,  1;           nop

## Eventual value of ra0: 1
```

Listing 3.2: Example of an Unconditional Branch

The conditional specifiers encode the branching condition in accordance to the current state of the NCZ bit flags. Those bit flags are updated by the last executed ALU instruction of the add pipeline and represent the NCZ events on the four 8-bit segments of the 32-bit result seperately. Chapter 2.2.2 about the QPU's core pipeline goes into detail about how the quad processing unit handles 32-bit values by processing four 8-bit segments. For each 8-bit segment exist three flags that are set in the event of a negative value (N) in the segment, in the event of a carry over (C) in the segment or in the event of a result equaling zero (Z) in the segment. A conditional specifier consists of two letters, of which the first one states the flag (N, C or Z) that should be considered while the second one states how the flags in the single segments should be set in order to fulfill the branching condition. Table 3.1 gives a detailed description on how the second letter in the conditional specifier corresponds to how the NCZ flags should be set in the single segments.

Conditional Specifier	**Branch Condition**
brr — brr.* — bra	Always
brr.zf	If all 8-bit segments equal 0
brr.ze	If no 8-bit segment equals 0
brr.zs	If any 8-bit segment equals 0
brr.zc	If any 8-bit segment does not equal 0
brr.nf	If all 8-bit segments represent a negative
brr.ne	If no 8-bit segment represents a negative
brr.ns	If any 8-bit segment represents a negative
brr.nc	If any 8-bit segment does not represent a negative
brr.cf	If a carry-over took place on all 8-bit segments
brr.ce	If no carry-over took place on all 8-bit segments
brr.cs	If a carry-over took place on any 8-bit segment
brr.cc	If no carry-over took place on any 8-bit segment

Table 3.1.: Overview of Conditional Specifiers for Branches

Both types of branches have in common that they are usually followed by three nop instructions as the QPU hardware needs three instruction cycles to figure out the branching target and to actually branch there. All the while though does the hardware still fetch instructions and sequentially execute them. Although calls to special I/O registers invoking the VPM, TMU or SFU are not permitted within those three instructions following a branch, operations involving physical registers that are not used in the branching instruction are allowed.

```
## Load ra0 with 2
ldi ra0, 0x00000002

## Branch if ra0 != 0 to example_label
sub ra0,  ra0,  1;          nop
brr.ze ra39, rb39, example_label
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop

add ra0,  ra0,  10;         nop

example_label:
add ra0,  ra0,  10;         nop

## Eventual value of ra0: 11
```

Listing 3.3: Example of a Conditional Branch

Part of every branch instruction is, despite the specification of the branching target label, the naming of two arbitrary registers. Those registers serve as write-registers and get passed a 32-bit value that represents the current *program counter* value (abbr. pc) at the time of the branch. The program counter value holds the information about how far along the chain of instructions the QPU has already gotten, meaning at which instruction QPU is at the moment. The program counter value saved in those write-registers represents the address of the instruction that would follow right after the branch is actually executed, so the saved value equals pc+4, with pc representing the address at which the branch is invoked.

This information about the pc is crucial for a *Branchback* functionality that enables regular function-calling within the QPU assembler code. It works by first regularly invoking a branching instruction but substituting one of the nop write-registers with a physical register into which the branchback address is stored for a later return. After the execution of the code at the target label (during which it is important not to overwrite the branchback address) an unconditional branching instruction needs to be made with the target label stated as `BRANCHBACK` and a fourth argument being the register from register file A into which the branchback address was previously saved. It is an important language constraint that although the branchback address can be written in either register file (or both for the matter), when supplementing the fourth register with the branchback register it needs to be from register file A - not an accumulator or register from register file B. After this special branch will the execution of programcode continue from the point of the original branch that saved the branchback address.

```
## Load ra0 with 2
ldi ra0, 0x00000002

## Branch if ra0 != 0 to example_function
sub ra0,  ra0,  1;          nop
brr.ze ra1, rb39, example_function
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop

add ra0,  ra0,  10;         nop

## Eventual value of ra0: 14

## Function that multiplies register ra0 by 4 and branches back according to ra1
example_function:
nop rb39, ra0,  4;          mul24 ra0, ra0, 4

brr ra39, rb39, BRANCHBACK, ra1
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop
```

Listing 3.4: Example of the Branchback Functionality

Semaphores

Semaphores are intended for interprocess communication between QPUs in case of interdependencies of results or similar reliance. All semaphores - the GPU makes a total of 16 semaphores available to all QPUs - are initiated with the value 0 and will stall the QPU if the value of the addressed semaphore is unlike 0. A stalled QPU will continue once the addressed semaphore is brought back to 0 again by another QPU. As all semaphores are internally 4-bit counting registers, is it possible to stall all (or preferably all but one) QPUs at once. Calls to the semaphores are invoked by stating *sema*, the intention to increase (*up*) or decrease (*down*) it and the specification of the addressed semaphore (0 to 15).

```
## Increase semaphore 0
sema up, 0

## Decrease semaphore 4
sema down, 4
```

Listing 3.5: Examples for the Use of Semaphores

3.1.3. QPU Register Map

Each QPU has a total of 128 registers at its disposal. Those 128 registers are primarily divided into two register files - 64 registers of register file A and 64 registers of register file B. Both of those 64 registers are then again divided by the lower numbered 32 registers being physical registers and the higher numbererd 32 registers being virtual I/O registers.

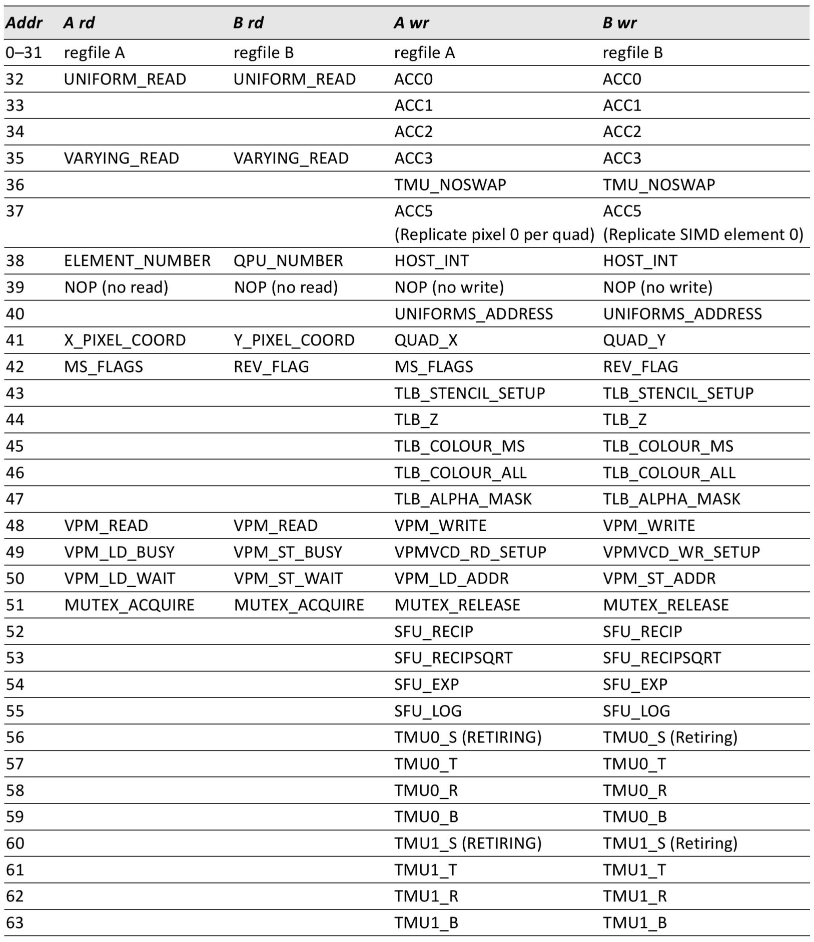
Therefore does each QPU have 32 physical registers of register file A, 32 physical registers of register file B, 32 virtual registers of register file A and 32 virtual registers of register file B. An exhaustive listing of all registers is stated in illustration 3.8, showing the register address map. While the 64 physical registers are simply intended for writing and reading medium-termed values, do the 64 virtual registers differ in their functionality depending upon if they are read from or written into.

The accumulator registers `ra32-ra35` and `rb32-rb35` are special in the sense that in order to write into the according accumulator one has to write to the named virtual registers. However in order to read from accumulators one has to read from the registers `r0-r5`. As there is no virtual register to read from the accumulators the functionality to read from them was implemented in the Assembler. Noteworthy is also that one can only write into the first 4 general-purpose accumulators `r0-r3` by writing into `ra32-ra35` or `rb32-rb35`. The accumulators `r4` and `r5` only serve to hold the results of special purpose I/O functions such as a call to the SFU or TMU, as mentioned in the previous section.

3.1.4. The Vertex Pipe Memory

The *Vertex Pipe Memory* (abbr. VPM) is a 48 KB large cache accessible by all QPUs that is intended for storing results of calculations and to serve as the memory gateway to the CPU, as the CPU can only fetch results from the GPU that are stored in the VPM. Though access to the VPM cache is bidirectional, meaning the VPM serves as a read and write cache for the QPUs and can also be written into by the CPU. The VPM itself is by default divided equally into 12 VPM segments of 4 KB each, whereby each segment is assigned to one of the 12 QPUs.

Each of those 12 VPM segments can be considered as a two-dimensional byte array of 32-bit words that is 16 words wide with a maximum height of 64 words. The VPM segments can



Addr	A rd	B rd	A wr	B wr
0–31	regfile A	regfile B	regfile A	regfile B
32	UNIFORM_READ	UNIFORM_READ	ACC0	ACC0
33			ACC1	ACC1
34			ACC2	ACC2
35	VARYING_READ	VARYING_READ	ACC3	ACC3
36			TMU_NOSWAP	TMU_NOSWAP
37			ACC5 (Replicate pixel 0 per quad)	ACC5 (Replicate SIMD element 0)
38	ELEMENT_NUMBER	QPU_NUMBER	HOST_INT	HOST_INT
39	NOP (no read)	NOP (no read)	NOP (no write)	NOP (no write)
40			UNIFORMS_ADDRESS	UNIFORMS_ADDRESS
41	X_PIXEL_COORD	Y_PIXEL_COORD	QUAD_X	QUAD_Y
42	MS_FLAGS	REV_FLAG	MS_FLAGS	REV_FLAG
43			TLB_STENCIL_SETUP	TLB_STENCIL_SETUP
44			TLB_Z	TLB_Z
45			TLB_COLOUR_MS	TLB_COLOUR_MS
46			TLB_COLOUR_ALL	TLB_COLOUR_ALL
47			TLB_ALPHA_MASK	TLB_ALPHA_MASK
48	VPM_READ	VPM_READ	VPM_WRITE	VPM_WRITE
49	VPM_LD_BUSY	VPM_ST_BUSY	VPMVCD_RD_SETUP	VPMVCD_WR_SETUP
50	VPM_LD_WAIT	VPM_ST_WAIT	VPM_LD_ADDR	VPM_ST_ADDR
51	MUTEX_ACQUIRE	MUTEX_ACQUIRE	MUTEX_RELEASE	MUTEX_RELEASE
52			SFU_RECIP	SFU_RECIP
53			SFU_RECIPSQRT	SFU_RECIPSQRT
54			SFU_EXP	SFU_EXP
55			SFU_LOG	SFU_LOG
56			TMU0_S (RETIRING)	TMU0_S (Retiring)
57			TMU0_T	TMU0_T
58			TMU0_R	TMU0_R
59			TMU0_B	TMU0_B
60			TMU1_S (RETIRING)	TMU1_S (Retiring)
61			TMU1_T	TMU1_T
62			TMU1_R	TMU1_R
63			TMU1_B	TMU1_B

Illustration 3.8.: The QPU Register Address Map Including Virtual Registers [Bro13]

be setup and accessed in different layout modes, termed VPM *orientations*, that determine the way of how the addressing corresponds to single bytes within the array. There are two basic orientations possible - either a horizontal or a vertical orientation. Illustration 3.9 demonstrates a 4 KB VPM segment set up in the horizontal orientation, as it is the more intuitive one. The illustration also displays several buffers of data saved with different VPM load configurations.

Setup and Access to the Vertex Pipe Memory

Access of individual QPUs to the VPM is automated by the *Vertex Cache Manager* (abbr. VCM), which among other things arranges the default division of the VPM and assignment of the VPM segments. Due to this default segmentation of the VPM does each QPU execute the same set of instructions in order to access its assigned segment of the VPM. Though QPU access to other segments than the assigned one is possible with an extended VPM setup. However does this require distinct instruction sets to execute for each QPU in order for them to not write on the same VPM segment.

When accessing the VPM regularly though, is it first necessary for a QPU to acquire the shared VPM hardware mutex. The acquisition of the VPM mutex will stall the attempts of

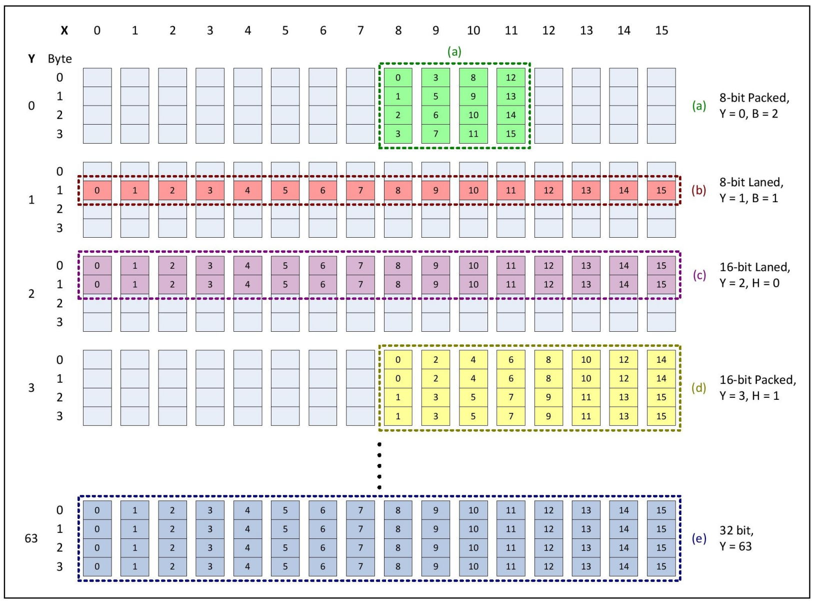

X
Y
Byte
(a) 8-bit Packed, Y = 0, B = 2
(b) 8-bit Laned, Y = 1, B = 1
(c) 16-bit Laned, Y = 2, H = 0
(d) 16-bit Packed, Y = 3, H = 1
(e) 32 bit, Y = 63

Illustration 3.9.: VideoCore IV VPM Horizontal Access Mode Examples [Bro13]

other QPUs to acquire the mutex until the mutex is released and the QPU scheduler decides for the next QPU that is granted mutex acquisition. The mutex is necessary because the hardware to access the VPM is shared by all 4 QPUs within a single slice. Overwriting the VPM access configuration while another QPU is still reading from or writing to the VPM therefore results in random reads and writes.

The VPM mutex is acquired by reading from register `ra51` or `rb51` with an arbitrary operation. The execution of the instructions is stalled until the mutex is free to acquire by the QPU. By writing into register `ra51` or `rb51` is the VPM mutex released again.

Listing 3.6 shows an exemplary load of 16 bytes from the CPU to the VPM and a subsequent read of those values from the VPM to the QPU. The VPM load is done by writing the immediate representing the VPM load configuration to register `ra49`, passing the address of the intended VPM buffer to register `ra50` and then waiting for the VPM load to complete by reading from register `ra50`, which stalls the QPU's instruction until the VPM load finished. In similar fashion is the VPM read initiated as first of all the 32-bit configuration is written to register `ra49` - though with a different ID field this time representing a VPM read instead of a VPM load. The VPM values are then read according to configuration simply by reading from register `ra48`.

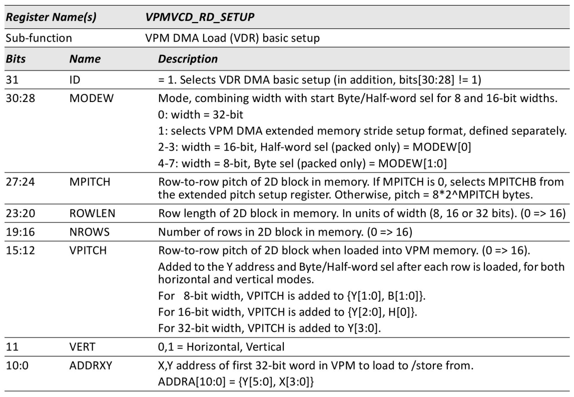
The 32-bit VPM configuration hexadecimals are conceived according to illustration 3.10, which showcase how different bit segments within a 32-bit configuration correspond to a single VPM configuration field. Those and further tables presenting the configuration fields for further VPM access modes can be found in the ARG on page 57 and following.

While most configuration fields are sufficiently - though certainly not exhaustively - explained is it important to comment on configuration fields whose explanation is lacking. The configuration fields MPITCH and ROWLEN of the VPM load setup basically both state the length of a single row in the VPM array, once in bytes and once in units of chosen width. The configuration field VPITCH is best set to 0 for an intuitive addressing, allowing for the lowest 4 bits in ADDR to correspond to the X coordinate of the two-dimensional byte array

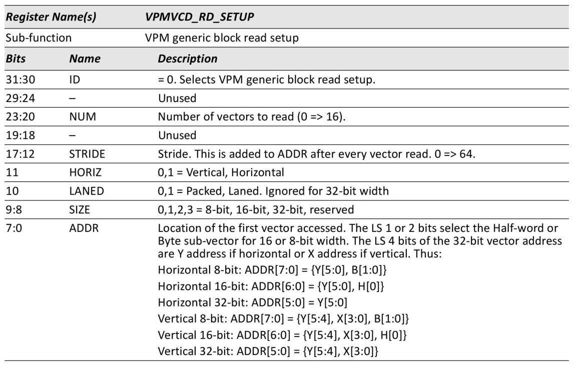
```
## Receive VPM buffer address via uniform
or  ra31, ra32, 0;          nop

## Acquire VPM mutex
or  ra39, ra51, rb39;       nop

## VPM DMA load setup
## ID=1, MODEW=0, MPITCH=1, ROWLEN=4, NROWS=1, VPITCH=0, VERT=0, ADDRXY=0
ldi ra49, 0x81410000
or  ra50, ra31, 0;          nop
or  ra39, ra50, rb39;       nop

## VPM read setup
## ID=0, NUM=4, STRIDE=1, HORIZ=0, LANED=0, SIZE=2, ADDR=0
ldi ra49, 0x00401200
or  ra0,  ra48, 0;          nop   # ra0 = data[0]
or  ra1,  ra48, 0;          nop   # ra1 = data[1]
or  ra2,  ra48, 0;          nop   # ra2 = data[2]
or  ra3,  ra48, 0;          nop   # ra3 = data[3]

## Release VPM mutex
or  ra51, ra39, rb39;       nop
```

Listing 3.6: Implementation of Transmitting 16 byte from CPU to QPU via VPM

while the remaining bits correspond to the Y coordinate (see illustration 3.9). Lastly is it important to state opposing VPM setup orientations in the VERT and HORIZ configuration fields. While the orientation stated in the VPM load setup determines the actual VPM orientation, is an opposing orientation required in the VPM read setup in order for the QPU to read as intended.



Register Name(s)	***VPMVCD_RD_SETUP***	
Sub-function	VPM DMA Load (VDR) basic setup	
Bits	***Name***	***Description***
31	ID	= 1. Selects VDR DMA basic setup (in addition, bits[30:28] != 1)
30:28	MODEW	Mode, combining width with start Byte/Half-word sel for 8 and 16-bit widths. 0: width = 32-bit 1: selects VPM DMA extended memory stride setup format, defined separately. 2-3: width = 16-bit, Half-word sel (packed only) = MODEW[0] 4-7: width = 8-bit, Byte sel (packed only) = MODEW[1:0]
27:24	MPITCH	Row-to-row pitch of 2D block in memory. If MPITCH is 0, selects MPITCHB from the extended pitch setup register. Otherwise, pitch = 8*2^MPITCH bytes.
23:20	ROWLEN	Row length of 2D block in memory. In units of width (8, 16 or 32 bits). (0 => 16)
19:16	NROWS	Number of rows in 2D block in memory. (0 => 16)
15:12	VPITCH	Row-to-row pitch of 2D block when loaded into VPM memory. (0 => 16). Added to the Y address and Byte/Half-word sel after each row is loaded, for both horizontal and vertical modes. For 8-bit width, VPITCH is added to {Y[1:0], B[1:0]}. For 16-bit width, VPITCH is added to {Y[2:0], H[0]}. For 32-bit width, VPITCH is added to Y[3:0].
11	VERT	0,1 = Horizontal, Vertical
10:0	ADDRXY	X,Y address of first 32-bit word in VPM to load to /store from. ADDRA[10:0] = {Y[5:0], X[3:0]}



Register Name(s)	***VPMVCD_RD_SETUP***	
Sub-function	VPM generic block read setup	
Bits	***Name***	***Description***
31:30	ID	= 0. Selects VPM generic block read setup.
29:24	–	Unused
23:20	NUM	Number of vectors to read (0 => 16).
19:18	–	Unused
17:12	STRIDE	Stride. This is added to ADDR after every vector read. 0 => 64.
11	HORIZ	0,1 = Vertical, Horizontal
10	LANED	0,1 = Packed, Laned. Ignored for 32-bit width
9:8	SIZE	0,1,2,3 = 8-bit, 16-bit, 32-bit, reserved

7:0	ADDR	Location of the first vector accessed. The LS 1 or 2 bits select the Half-word or Byte sub-vector for 16 or 8-bit width. The LS 4 bits of the 32-bit vector address are Y address if horizontal or X address if vertical. Thus:
		Horizontal 8-bit: ADDR[7:0] = {Y[5:0], B[1:0]}
		Horizontal 16-bit: ADDR[6:0] = {Y[5:0], H[0]}
		Horizontal 32-bit: ADDR[5:0] = Y[5:0]
		Vertical 8-bit: ADDR[7:0] = {Y[5:4], X[3:0], B[1:0]}
		Vertical 16-bit: ADDR[6:0] = {Y[5:4], X[3:0], H[0]}
		Vertical 32-bit: ADDR[5:0] = {Y[5:4], X[3:0]}

Illustration 3.10.: Configuration Fields for VPM Load & Read [Bro13]

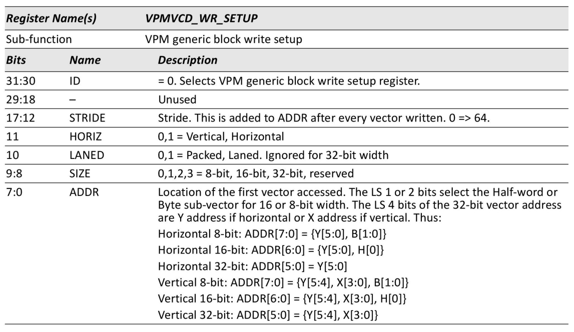
For the reason of thoroughness does listing 3.7 show an exemplary store of 16 bytes from the QPU to the CPU via the VPM. Write & store configuration hexadecimals are written to register `rb49` instead of register `ra49` and the adress where the written values are intended to be stored is passed to register `rb50`, not `ra50`. Aside from those details can the write & store process be seen as the mirror of the load & read process in regards to the sequence of instructions.

Though despite the write & store and the load & read processes being mirror operations to each other, do the VPM configurations and configuration fields differ significantly. Details to the write & store configuration fields are listed in illustration 3.11. Though while the necessity of opposing VPM orientation still applies in the VPM write & store process, does now the configuration field UNITS state the number of allocated rows in the VPM array and the configuration field DEPTH alone states the length of those rows in units of chosen

```
## Receive VPM buffer address via uniform
or  ra31, ra32, 0;          nop

## Acquire VPM mutex
or  ra39, ra51, rb39;       nop

## VPM write setup
## ID=0, STRIDE=1, HORIZ=0, LANED=0, SIZE=2, ADDR=0
ldi rb49, 0x00001200

or  ra48, ra0,  0;          nop     # write data[0]
or  ra48, ra1,  0;          nop     # write data[1]
or  ra48, ra2,  0;          nop     # write data[2]
or  ra48, ra3,  0;          nop     # write data[3]

## VPM DMA store setup
## ID=2, UNITS=1, DEPTH=4, LANED=0, HORIZ=1, VPMBASE=0, MODEW=0
ldi rb49, 0x80844000
or  rb50, ra31, 0;          nop
or  ra39, ra39, rb50;       nop

## Release VPM mutex
or  ra51, ra39, rb39;       nop
```

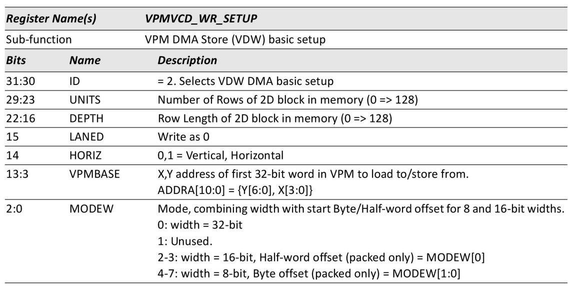
Listing 3.7: Implementation of Transmitting 16 byte from QPU to CPU via VPM

width. As with the VPM load & read process does the VPM store setup determine the actual VPM orientation. Though the configuration field VPITCH does not exist in the corresponding VPM store configuration, does the addressing resulting from the chosen value in ADDR correspond to the intuitive addressing in the load & read process when the VPITCH configuration field is set to 0.



Register Name(s)	***VPMVCD_WR_SETUP***	
Sub-function	VPM generic block write setup	
Bits	***Name***	***Description***
31:30	ID	= 0. Selects VPM generic block write setup register.
29:18	–	Unused
17:12	STRIDE	Stride. This is added to ADDR after every vector written. 0 => 64.
11	HORIZ	0,1 = Vertical, Horizontal
10	LANED	0,1 = Packed, Laned. Ignored for 32-bit width
9:8	SIZE	0,1,2,3 = 8-bit, 16-bit, 32-bit, reserved

7:0	ADDR	Location of the first vector accessed. The LS 1 or 2 bits select the Half-word or Byte sub-vector for 16 or 8-bit width. The LS 4 bits of the 32-bit vector address are Y address if horizontal or X address if vertical. Thus:
		Horizontal 8-bit: ADDR[7:0] = {Y[5:0], B[1:0]}
		Horizontal 16-bit: ADDR[6:0] = {Y[5:0], H[0]}
		Horizontal 32-bit: ADDR[5:0] = Y[5:0]
		Vertical 8-bit: ADDR[7:0] = {Y[5:4], X[3:0], B[1:0]}
		Vertical 16-bit: ADDR[6:0] = {Y[5:4], X[3:0], H[0]}
		Vertical 32-bit: ADDR[5:0] = {Y[5:4], X[3:0]}



Register Name(s)	***VPMVCD_WR_SETUP***	
Sub-function	VPM DMA Store (VDW) basic setup	
Bits	***Name***	***Description***
31:30	ID	= 2. Selects VDW DMA basic setup
29:23	UNITS	Number of Rows of 2D block in memory (0 => 128)
22:16	DEPTH	Row Length of 2D block in memory (0 => 128)
15	LANED	Write as 0
14	HORIZ	0,1 = Vertical, Horizontal
13:3	VPMBASE	X,Y address of first 32-bit word in VPM to load to/store from. ADDRA[10:0] = {Y[6:0], X[3:0]}
2:0	MODEW	Mode, combining width with start Byte/Half-word offset for 8 and 16-bit widths. 0: width = 32-bit 1: Unused. 2-3: width = 16-bit, Half-word offset (packed only) = MODEW[0] 4-7: width = 8-bit, Byte offset (packed only) = MODEW[1:0]

Illustration 3.11.: Configuration Fields for VPM Write & Store [Bro13]

Problems with the Vertex Pipe Memory

Summarizing the Vertex Pipe Memory, one can hardly overestimate the importance of this memory cache. For a start does it provide the only possibility to save medium-termed data larger than the memory provided by the QPU registers as it poses the only writeable cache of GPU. On the other hand does the VPM serve as the memory gateway to the CPU and proper control of the VPM is therefore necessary to exchange data between the CPU and GPU. Problems with the setup and control of the VPM therefore weigh gravely on the possible achievable performance of the GPU.

However one severe problem with the proper control of the VPM exist and expresses itself in partially random reads and writes from and to the VPM. The problem of random reads from the VPM occurs when between multiple VPM reads the hardware mutex is released and reacquired without also performing a VPM load within the same timespan in which the

VPM mutex is held. Is this the case are the reads of the QPU partially random in that the VPM segments of neighboring QPUs within the same slice are read instead of the VPM segment that is actually assigned to the currently reading QPU. Listing 3.8 provides a very simple example showing how to cause such a partially random read.

```
## Receive VPM buffer address via uniform
or  ra31, ra32, 0;          nop

## Acquire VPM mutex
or  ra39, ra51, rb39;       nop

## VPM DMA load setup
## ID=1, MODEW=0, MPITCH=1, ROWLEN=4, NROWS=1, VPITCH=0, VERT=0, ADDRXY=0
ldi ra49, 0x81410000
or  ra50, ra31, 0;          nop
or  ra39, ra50, rb39;       nop

## VPM read setup
## ID=0, NUM=2, STRIDE=1, HORIZ=0, LANED=0, SIZE=2, ADDR=0
ldi ra49, 0x00201200
or  ra0,  ra48, 0;          nop    # ra0 = data[0]
or  ra1,  ra48, 0;          nop    # ra1 = data[1]

## Release VPM mutex
or  ra51, ra39, rb39;       nop

...

## Reacquire VPM mutex
or  ra39, ra51, rb39;       nop

## VPM read setup
## ID=0, NUM=2, STRIDE=1, HORIZ=0, LANED=0, SIZE=2, ADDR=0
ldi ra49, 0x00201200
or  ra2,  ra48, 0;          nop    # ra2 = random data
or  ra3,  ra48, 0;          nop    # ra3 = random data

## Release VPM mutex
or  ra51, ra39, rb39;       nop
```

Listing 3.8: Example on How to Cause Partially Random VPM Read

The impact of this problem is enormous. Not only does it effectively limit the instructions to merely one VPM load and one VPM read per transmitted set of QPU instructions; because multiple VPM reads lead to partially random reads and multiple VPM loads unnecessarily stall the QPU and overwrite possibly already processed data. It therefore also limits the maximum of usable VPM memory to the maximum amount of memory the QPUs can provide with their physical registers, since there is only one instance during which the QPU registers can be provided with the data to be processed. In the best case scenario and under the assumption that a parallelized algorithm utilizes all 64 physical registers for data that needs to be processed while requiring no further physical register for algorithm control, is each QPU able to hold 256 byte of data. Therefore would all QPUs only be able to utilize 3KB out of a total 48KB VPM memory and the persisting VPM random read problem would diminish the VPM's utilization to only 6.25% of its potential.

Additionally though is there also the problem of random writes to the VPM memory, which occur analogously when multiple VPM writes are performed in between the release and reacquisition of the VPM mutex.

The problem itself is likely caused by the common use of the hardware with which to access the VPM memory. Four QPUs within a slice share the access to the VPM and as demonstrated in illustration 3.12 does the randomly chosen last QPU within a slice determine which VPM segment is addressed when only a VPM read operation is executed without a preceding VPM load operation. This most likely procedure of operation is concluded by the significant behaviour of random reads limiting themself to neighboring QPUs within the same slice. However the problem also arises because of missing bare metal hardware schematics and closed source firmware, as the problem seems solvable and most likely has

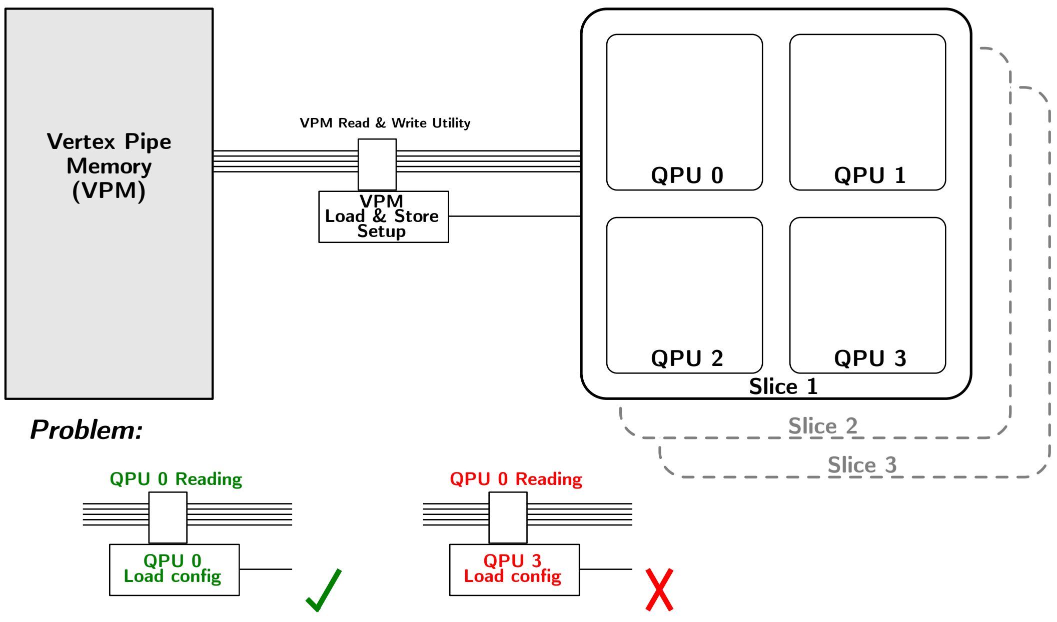


Illustration 3.12.: Visualization of VPM Mutex Problem

an existing solution which however is withheld due to undisclosed detailed documentation.

3.1.5. The Texture Memory Units

The *Texture Memory Unit* (abbr. TMU) is originally intended to provide streams of textures to the GPU as necessary for the OpenGL ES graphic programming interface. Internally though is the TMU a segment of the 128 KB level 2 cache of the CPU that is shared with the GPU and serves the GPU as a read-only memory. The TMU is available to each QPU as two units or a single unit, depending upon the need of two or one stream of textures. The textures provided by the TMU are simply a stream of 32-bit values accessed via a simple register I/O instruction on the QPU and passed to QPU via a special configuration of the data sent in the kernel driver. The TMU can preload 8 textures in single unit mode (4 textures each in two unit mode) via 8 single I/O instructions without the necessity of a mutex and therefore works best to serve read-only buffers with irregular access to the QPU.

The read-only buffers provided by the TMU allow for random reads, which are achieved by multiplying the intended index of the 32-bit buffer by 4 and adding it to the TMU buffer base address. The result is then written to register `ra56` or `rb56`. The TMU buffer base address has to be received first by simply passing it via a uniform value from the driver. This way up to 8 TMU values can be preloaded and later retrieved by simply outfitting a regular ALU instruction with the conditional specifier *.tmu*. The TMU value can be read from accumulator `r4` in the following instruction.

Should the TMU be used in two unit mode is it necessary to first load a value unequal 0 to register ra36 or rb36 to enable the manual control of both TMU units. The first TMU is then accessed by preloading to register `ra56` or `rb56` and the conditional specifier *.tmu0*. The second TMU is accessed by preloading to register `ra60` or `rb60` and the conditional specifier *.tmu1*. Listing 3.9 will show the TMU used in both modes.

```
## Receive TMU buffer base addresses via uniform
or  ra30, ra32, 0;          nop
or  ra31, ra32, 0;          nop

## Preload the first 64 bit to TMU
add ra56, ra30, 0;          nop
add ra56, ra30, 4;          nop

## Save preloaded 64 bit from TMU to ra1 and ra2
nop.tmu   ra39, ra39, rb39; nop
or        ra1,  r4,   0;    nop
nop.tmu   ra39, ra39, rb39; nop
or        ra2,  r4,   0;    nop

## Set TMU to two unit mode
ldi ra36, 1

## Preload 32 bit from buffer 1 and 32 bit from buffer 2
add ra56, ra30, 8;          nop
add ra56, ra31, 0;          nop

## Save both preloaded 62 bit from TMU0 and TMU1 to ra3 and ra4
nop.tmu0  ra39, ra39, rb39; nop
or        ra3,  r4,   0;    nop
nop.tmu1  ra39, ra39, rb39; nop
or        ra4,  r4,   0;    nop
```

Listing 3.9: Examples for Both Usage Modes of TMU Access

3.1.6. QPU Termination

A QPU is correctly terminated by first sending an interrupt signal to the host and secondly initiating the QPU termination. The interrupt signal to the host serves to let the driver know about how many QPUs are still left computing so the driver can initate the memory transfer if all QPUs are done. The QPU termination following is initiated through the conditional specifier *.tend* and requires 2 following nop instructions so the QPU may correctly clean up and not get stuck in a state of expecting further instructions. This erroneous state will lead to the unavailability of the QPU for further calls and makes a reboot of the Raspberry Pi necessary. Listing 3.10 shows the correct termination instructions.

```
## Trigger interrupt to finish the program and signal host
or  ra38, ra39, rb39;       nop
nop.tend  ra39, ra39, rb39; nop
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop
```

Listing 3.10: Termination of QPU

3.2. Implementation of Assembler

The assembler for the QASM programming language is in its basic form taken from Eric Lorimer, however it was greatly enhanced in its capabilities and severe bugs have been fixed for this thesis. The compiled assembler can be used with the first argument being the input file containing the QASM instruction code and preferably ending with the postfix `.qasm`. The input file ending of however is not enforced. The optional second argument being the `-o` option followed by the intended output file, though if no second argument is stated will the assembler write the binary output to the file `out.bin`.

When the assembler binary is executed it will then go through each line of the input file, determine if the line represents a single and valid instruction and then call the appropriate function that will convert the line into a 64-bit binary encoding a single instruction. The encoding of each of the different instructions that were introduced in chapter 3.1 will be examined in the upcoming sections. Furthermore is it important to establish that as the general-purpose assembler is still being considered to be in an early stage of development, it will only catch the most common errors that are in violation with the QASM programming language.

3.2.1. ALU Instruction Encoding

ALU instructions are the most common instructions in the QASM language and come in two different variants. The regular variant called *alu* instruction and the variant in which one or both of the second read-registers are replaced with a 6-bit small immediate, called *alu small imm* instruction. The instruction encoding of both is displayed in illustration 3.13 while the single instruction fields are outlined in illustration 3.14. The respective variant is indicated by the replacement of the `sig` instruction field with a distinctive instruction header.

Although some explanatory chapters are referenced along this walkthrough of the ALU instruction encoding, can further references to the ARG for certain instruction fields also be found in their respective descriptions in illustration 3.14. The references and descriptions for those instruction fields are also applicable for the encoding of the additional Non-ALU instructions, as the instruction encoding of the QPU architecture is highly uniform and shares many instruction-fields throughout different instructions.

	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
alu	sig				unpack			pm	pack				cond_add			cond_mul			sf	ws	waddr_add						waddr_mul					
	op_mul			op_add					raddr_a						raddr_b						add_a			add_b			mul_a			mul_b		
alu small imm	1	1	0	1	unpack			pm	pack				cond_add			cond_mul			sf	ws	waddr_add						waddr_mul					
	op_mul			op_add					raddr_a						small_immed						add_a			add_b			mul_a			mul_b		

Illustration 3.13.: Instruction Encoding of ALU and ALU Small Imm [Bro13]

The assembler handles both ALU variants within the same function *assembleALUInstr()* and changes the `sig` instruction field to the *alu small imm* header if a small immediate is read in the currently processed line. Otherwise will the value for the `sig` instruction field - which encodes special conditional specifiers like *.tend* or *.tmu* - be read only via conditional specifiers from the add pipeline, not the mul pipeline. The following unpack, pm and pack instruction fields encode the *pack* vector operations outlined in chapter 2.2.2 about the QPU Core Pipeline and can be invoked by a conditional specifier either stating a pack or unpack operation combined with the desired vector, as outlined in the ARG on p.31 (e.g. *.pack16a*). The `cond_add` and `cond_mul` fields encode conditions in case the add operation or mul operation should only by executed if the respective condition is fulfilled - much like the conditionals of branches. However this functionality is not yet implemented in the current assembler and both instruction fields are set to always perform the operation. As the last of the instruction configurations are the `sf` and `ws` bits set automatically by the assembler. Those instruction fields do only represent functionality that is a regular component of the QASM language, like the statement if the add pipeline should write to a register of register file A or register file B.

While the `op_mul` and `op_add` fields are encoding the respective operation according to

Field	*Bits*	*Description*
sig	4	Signaling bits (see “Signaling Bits” on page 29)
unpack	3	Unpack mode (see “Pack/Unpack Bits” on page 30)
pm	1	Pack/Unpack select (see “Pack/Unpack Bits” on page 30)
pack	4	Pack mode (see “Pack/Unpack Bits” on page 30)
cond_add	3	Add ALU condition code (see “ALU Input Muxes” on page 28)
cond_mul	3	Mul ALU condition code (see “ALU Input Muxes” on page 28)
sf	1	Set flags Flags are updated from the add ALU unless the add ALU performed a NOP (or its condition code was NEVER) in which case flags are updated from the mul ALU
ws	1	Write swap for add and multiply unit outputs If ws = 0, add alu writes to regfile a, mult to regfile b If ws = 1, add alu writes to regfile b, mult to regfile a
waddr_add	6	Write address for add output If ws = 0, regfile writes go to regfile a, else regfile b
waddr_mul	6	Write address for multiply output If ws = 0, regfile writes go to regfile b, else regfile a
op_mul	3	Multiply opcode (see “Op Mul” on page 36)
op_add	5	Add opcode (see “Op Add” on page 35)
raddr_a	6	Read address for register file a
raddr_b	6	Read address for register file b
small_immed	6	Small immediate or specified vector rotation for mul ALU output (see “ALU Input Muxes” on page 28)
add_a	3	Input mux control for A port of add ALU (A add operand) (see “ALU Input Muxes” on page 28)
add_b	3	Input mux control for B port of add ALU (B add operand) (see “ALU Input Muxes” on page 28)
mul_a	3	Input mux control for A port of mul ALU (A mul operand) (see “ALU Input Muxes” on page 28)
mul_b	3	Input mux control for B port of mul ALU (B mul operand) (see “ALU Input Muxes” on page 28)

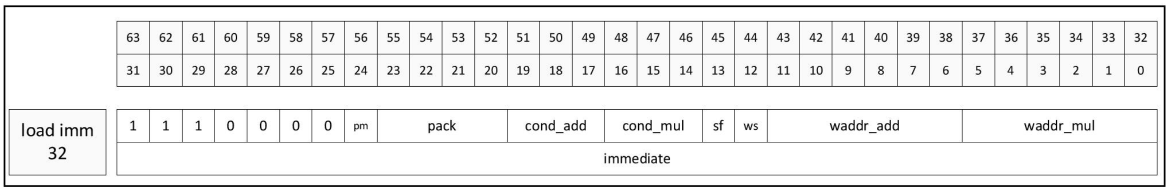
Illustration 3.14.: ALU Instruction Fields [Bro13]

the opcodes found in illustrations 3.6 and 3.7, are the rest of the instruction fields intended to encode the addressed registers of the current line. The fields `waddr_add` and `waddr_mul` encode the respective specific register number of the addressed write-registers of the add and mul pipeline. Though because those instruction fields only encode the register number - not the register file - is the information about the respective register file of each write-register saved in the previously set `ws` bit. Further along are the `raddr`, `add` and `mul` register encoded, which interplay in an intricated way that results in the complicated ALU grammar regarding the read-registers. The `add_a` and `add_b` instruction fields encode if accumulators are used in the first read-register and second read-register of the add pipeline and if so, which accumulators exactly. The `mul_a` and `mul_b` instruction fields do the same for the mul pipeline. Whereas the `raddr_a` and `raadr_b` instruction fields encode the addressed register of register file A and addressed register of register file B respectively in case of no accumulators being used. In the case of one of the second read-registers containing a small immediate is the 6-bit segment for the `raadr_b` replaced with 6-bit segment for the `small_immed`, which encodes the used small immediate instead of a register.

The early draft of the assembler by Eric Lorimer was especially lacking in the regard of encoding this intricated interplay, which often resulted in incomprehensible instruction behaviour of technically correct QASM code. The causing bugs however have been fixed in the current version of the assembler and an extensive check for errors vioalting the grammar of the ALU instructions has instead been implemented.

At last is it important to note that the feature of the QASM language regarding a simple skipping of the mul pipeline instruction by solely stating `nop`, is realized within the assembler by merely encoding the NOP registers `ra39` or `rb39` as the write-register. While doing so are the read-registers copied from the add pipeline to allow for greatest simplicity of the encoding. The formed instruction has the effect of performing a nop operation on the chosen read-registers and then disregarding the result as it is written to a NOP register.

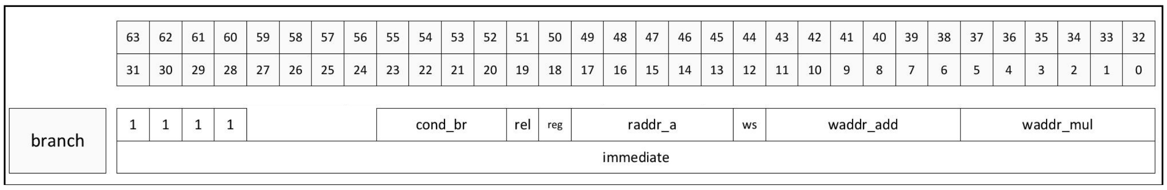
3.2.2. Load Immediate Instruction Encoding



	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
load imm 32	1	1	1	0	0	0	0	pm	pack				cond_add			cond_mul			sf	ws	waddr_add						waddr_mul					
	immediate																															

The Load immediate is in its functionality as well as in its encoding in many aspects very similar to the regular ALU instruction. Therefore does the intended uniform encoding of instructions in turn ensure a maximum reuse of existing instruction fields. The encoding entails a significantly bigger instruction header for clear identification and simply replaces the instruction fields stating the operation and read registers - as both make no sense in this instruction - with the intended 32-bit immediate. This immediate is then written to the registers encoded through `waddr_add`, `waddr_mul` and `ws`, with all the conditionals and restrictions of the regular ALU instruction still applying. Although the QPU supports two further load immediate intructions which would load 16 individual 2-bit values on each element of the SIMD array, is the binary encoding of those instructions not implemented by the assembler.

3.2.3. Branch Instruction Encoding

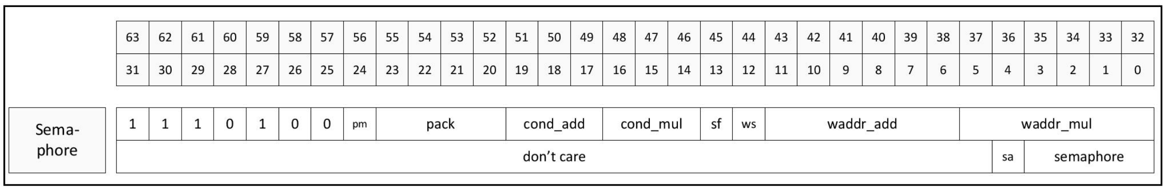


	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
branch	1	1	1	1					cond_br				rel	reg	raddr_a					ws	waddr_add						waddr_mul					
	immediate																															

The encoding of the branch instruction, although similar to that of the load immediate, has some new instruction fields and also uses the immediate represented by the less significant 32 bit in a different way. First of all is the new instruction field `cond br` encoding one of the conditional specifiers listed in table 3.1, that will make the branch dependent on the current values of the NCZ flags set by the assembler. The `reg` bit will be set if the branch instruction is outfitted with an additional register from the register file A, which indicates a branchback branch. The register of register file A holding the branchback address is encoded in the instruction field `raddr_a`. Herein lies the reason for the language constraint that although the branchback address can be saved in either regfile, if it is supplied as a fourth argument to the branch then it needs to be from register file A.

The `immediate` instruction field represting the less significant 32 bit of the instruction encoding is not directly supplied but rather calculated by the assembler out of the third branch instruction argument - the supplied target label. The assembler saves all target labels with their corresponding program counter value within an array of structs and sets the `immediate` instruction field to this corresponding program counter value if the target label is referenced. Therefore does the branching instruction enable the branch by stating the new absolute program counter value at which point processing is supposed to continue. However, the `immediate` instruction field can also encode a relative change to the current program counter value if the `rel` bit is set and the branch is therefore made relative.

3.2.4. Semaphore Instruction Encoding



	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sema-phore	1	1	1	0	1	0	0	pm	pack				cond_add			cond_mul			sf	ws	waddr_add						waddr_mul					
	don't care																											sa	semaphore			

The encoding of Semaphores is nearly identical to that of load immediates due to them using the same instruction decoding hardware. However does the semaphore not only replace the immediate with a large field of don't cares, but rather are all instruction fields of the most significant 32 bit basically dummies without any effect and are accordingly set by the assembler. The presence of these instructions fields is only due to the mentioned sharing of decoding hardware. The remaining instruction fields will encode if the semaphore should be increased (`sa` set) or decreased (`sa` not set) and which semaphore should be addressed by encoding it in the 4-bit `semaphore` instruction field.

3.3. The GPU Kernel Driver

The kernel driver for the Rapsberry Pi GPU was released along with the Architecture Reference Guide in 2014 under a 3-clause BSD license [Bro14]. It provides an access point - although an undocumented one - to the GPU of the Raspberry Pi and was used by the graphics programming interfaces that the Raspberry Pi officially supports, whose source code nevertheless was never disclosed. Unfortunately though is the Kernel driver optimized for the use by those graphic programming interfaces [Wik17f] and rewriting it to enable an optimized general use is intended for the future. Illustration 3.15 describes well the current state of accesibility to documentation and source code regarding the RPi GPU.

The kernel driver provides 12 functions for the preparation and transmission of the data to the GPU. All functions including their arguments are displayed in listing 3.11, which recites the contents of the mailbox.h - the header of the kernel driver. Following this listing is a detailed description of usage of each function and the order in which those functions are supposed to be called. Though not all functions listed in the header of the kernel driver are also described in detail, as some functions are not necessary for a successful full access to the GPU and furthermore lack all documentation about their functionality.

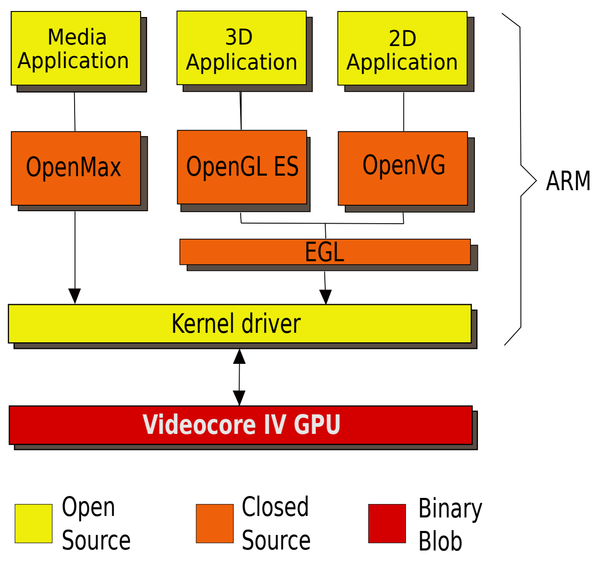


Illustration 3.15.: RaspberryPi GPU Open Source Status [Wik17f]

```
#include <linux/ioctl.h>

#define MAJOR_NUM 100
#define IOCTL_MBOX_PROPERTY _IOWR(MAJOR_NUM, 0, char *)
#define DEVICE_FILE_NAME /dev/vcio

int mbox_open();
void mbox_close(int file_desc);

unsigned get_version(int file_desc);
unsigned mem_alloc(int file_desc, unsigned size, unsigned align, unsigned flags);
unsigned mem_free(int file_desc, unsigned handle);
unsigned mem_lock(int file_desc, unsigned handle);
unsigned mem_unlock(int file_desc, unsigned handle);
void *mapmem(unsigned base, unsigned size);
void unmapmem(void *addr, unsigned size);

unsigned execute_code(int file_desc, unsigned code, unsigned r0, unsigned r1, unsigned r2,
    unsigned r3, unsigned r4, unsigned r5);
unsigned execute_qpu(int file_desc, unsigned num_qpus, unsigned control, unsigned noflush,
    unsigned timeout);
unsigned qpu_enable(int file_desc, unsigned enable);
```

Listing 3.11: mailbox.h - Kernel Driver Header

mbox_open

The `mbox_open()` function is the first to be called and it gets no arguments at all. Its purpose is to return a file descriptor that will serve as the *mailbox* through which data will be send and received in future functions and it will therefore be referred to as such. Internally will the function open up a pipe to `/dev/vcio` to communicate with the GPU.

qpu_enable

`qpu_enable()` will be called at least twice during every use of the kernel driver. Its first argument is the mailbox descriptor to enable communication with the GPU. The second argument is either the value 1 to start up the GPU - being the mandatory first use of this function - or the value 0 in case the GPU is supposed to shut down - being the second use of this function. The function then internally encodes the corresponding instruction for the

GPU's control and sends it.

mem_alloc

The function `mem_alloc()` takes four arguments and returns the very important handle to the GPU memory, termed `gpu_handle`. The first argument is again the mailbox. The second argument is the total amount of data in bytes that is intended to be transmitted to the GPU either by VPM or TMU in the next upcoming transmission. The third argument represents the amount of VPM memory allocated for every QPU in bytes and maxes out at 4096 as the VPM has only enough memory to allocate 4 KB for each QPU. The fourth argument is a configuration flag for the GPU in case of a direct or cached memory access. As direct memory access (GPU flag `0x0`) has not yet been sufficiently put to the test it is recommended to use cached memory access (GPU flag `0xc`).

mem_lock

`mem_lock()` will actually allocate the memory according to the configuration set forth in the `mem_alloc()` function. The function will return the GPU memory base address, referred to as *gpu_mem_base*. The first argument being passed is the mailbox, while the second argument is the handle returned by the `mem_alloc()` function, namely the gpu_handle.

mapmem

`mapmem()` fullfills the function of returning a base address of CPU memory that is intended to mirror the allocated memory on the GPU. This CPU memory is then filled with the data intended for transmission using the memory base address `mapmem()` will return. The first argument of `mapmem()` is the base address of the GPU memory the function is supposed to mirror, which was referred to as gpu_mem_base. The second argument is the same total bytecount that was used in `mem_alloc()` to allocate the appropriate amount of memory.

execute_qpu

The `execute_qpu()` function will start the transmission of the data and kick of the processing on the QPUs. It has five arguments, the first one being the mailbox and the second one being the amount of QPUs that are supposed to be run, possibly ranging from 1 to 12. The fourth argument states if during instruction execution the GPU memory should occassionally be flushed back to CPU memory, which is useful when debugging but vastly decreases performance. The fifth argument will simply give a time limit in miliseconds after which the GPU call should return so the CPU execution won't stall in case of an inadvertantly erroneous GPU calculation.

The third argument is very important in that it will be the base address of a buffer, which will hold important addresses for each QPU. This buffer holds the base address of the code each QPU is supposed to execute and will hold the base address of the uniforms each QPU can obtain. The base address of the uniforms is very important as the QPUs will receive the base addresses of VPM and TMU buffers through the retrieval of uniform values.

unmapmem

First function of the four closing functions that will free the allocated memory and close pipes. `unmapmem()` will free the memory allocated on the CPU side that is supposed to mirror the memory allocated on the GPU side. The first argument is the base address of the CPU memory and the second argument the total bytecount of transmitted data, which was also used in `mapmem()`.

mem_unlock

This function will unlock the memory allocated on the GPU, which is necessary before the memory can be freed by `mem_free()`. The first argument is the mailbox, while the second argument is supposed to be the handle (not the base address) of the GPU memory, referred to as gpu_handle.

mem_free

`mem_free()` will actually free the just unlocked memory on the GPU. The arguments are identical to those of the `mem_unlock()` function.

mbox_close

The `mbox_close()` function merely closes the pipe to the GPU through which information was sent, which was referred to as mailbox. Therefore is the mailbox filedescriptor the only argument.

3.3.1. Example Use of Kernel Driver

As the use of the kernel driver is obviously complex, as demonstrated in the descriptions of the used functions, will the following listing 3.12 demonstrate a generic and easy example of how to use the kernel driver in the most simplest of cases. The actual implementation used to access the GPU for the implementation of AES in the QASM language, showcasing a vastly more complex and complete example, can be found in the attached source code as the file `aes128_ecb_gpu.c` in the directory `aes128_ecb_gpu`.

```c
unsigned mailbox = mbox_open();                       // Open pipe for GPU communication

qpu_enable(mailbox, 1);                               // Enable QPUs

unsigned totalBytecount = qpuCodeBytecount +          // Bytecount of QPU code
                          dataBytecount +             // Bytecount of data
                          uniformsBytecount +         // Bytecount of uniforms
                          messagesBytecount;          // Bytecount of messages

// Allocate and lock memory on GPU, then create a mirror memory on CPU
unsigned gpu_handle = mem_alloc(mailbox, totalBytecount, 4096, 0xc);
unsigned gpu_mem_base = mem_lock(mailbox, gpu_handle);
void*    cpu_mem_base = mapmem(gpu_mem_base, totalBytecount);

struct GPUMemoryMap *cpu_map = (struct GPUMemoryMap*) cpu_mem_base;

// Map buffers to be transmitted to allocated GPU memory
memset(cpu_map, 0x0, sizeof(struct GPUMemoryMap));
unsigned vc_code = gpu_mem_base + offsetof(struct GPUMemoryMap, qpuCode);
unsigned vc_data = gpu_mem_base + offsetof(struct GPUMemoryMap, data);
unsigned vc_uni  = gpu_mem_base + offsetof(struct GPUMemoryMap, uniforms);
unsigned vc_msg  = gpu_mem_base + offsetof(struct GPUMemoryMap, messages);

memcpy(cpu_map->qpuCode, qpuCode, qpuCodeBytecount);
memcpy(cpu_map->data,    data,    totalBytecount);

for (int i=0; i < 12; i++) {                          // For each QPU (12)
  cpu_map->uniforms[i] = vc_data;                     // Address of data

  cpu_map->messages[i*2 + 0] = vc_uni + (i * sizeof(unsigned)); // Address uniforms
  cpu_map->messages[i*2 + 1] = vc_code;               // Address qpuCode
}

execute_qpu(mailbox, 12, vc_msg, 1, 10000);           // Start execution

memcpy(data, cpu_map->data, totalBytecount);          // Fetch the results

unmapmem(cpu_mem_base, totalBytecount);               // Free CPU memory
mem_unlock(mailbox, gpu_handle);                      // Unlock memory
mem_free(mailbox, gpu_handle);                        // Free GPU memory
qpu_enable(mailbox, 0);                               // Disable QPUs
mbox_close(mailbox);                                  // Close mailbox pipe
```

Listing 3.12: Example Use of Kernel Driver

3.3.2. Transmitting and Receiving Data through Kernel Driver

Although not shown in the upper simplified example of accessing the GPU, is the segmentation of different buffers of data which are to be sent to the GPU a considerable issue. Listing 3.12 however only shows an example in which there is merely one buffer intended that is to be accessed via the VPM and which is supposed to be fully fetched back.

Buffers which are to be accessed via the VPM need to be sectionend and seperately assigned for each QPU, while for buffers that are accessed via the TMU a single declaration and transmission of the buffer is sufficient in order for every QPU to receive access. The separation and assignment of differing VPM segments is however not performed by the declaration of entirely different buffers - though this would be possible - but is rather handled more straightforward by passing a different buffer offset value to each QPU. The transmission of different buffer segments to different VPM memory segments is then handled by the kernel driver and GPU firmware. in the exemplary listing 3.13 below are the buffers intended for VPM usage sectioned into 12 segments (for each QPU one segment) and are therefore assigned in offset steps of 40 bytes for `vpmBuffer_1` and 10 bytes for `vpmBuffer_2`. The shown `tmuBuffer` is not segmented as TMU buffers are accessible by all QPUs as a whole.

```
 1 ...
 2
 3 memcpy(cpu_map->qpuCode, qpuCode,     qpuCodeBytecount);
 4 memcpy(cpu_map->data,    tmuBuffer,   60);         // 60 byte of TMU data for all QPU
 5 memcpy(cpu_map->data,    vpmBuffer_1, 480);        // 40 byte of VPM data for each QPU
 6 memcpy(cpu_map->data,    vpmBuffer_2, 120);        // 10 byte of VPM data for each QPU
 7
 8 for (int i=0; i < 12; i++) {                       // For each QPU (12)
 9   cpu_map->uniforms[i*3 + 0] = vc_data;            // Address tmuBuffer
10   cpu_map->uniforms[i*3 + 1] = vc_data + 60     + (480/12)*i;    // Assigned vpmBuffer_1
11   cpu_map->uniforms[i*3 + 2] = vc_data + 60+480 + (120/12)*i;    // Assigned vpmBuffer_2
12
13   cpu_map->messages[i*2 + 0] = vc_uni + (i*3*sizeof(unsigned)); // Address uniforms
14   cpu_map->messages[i*2 + 1] = vc_code;            // Address qpuCode
15 }
16
17 execute_qpu(mailbox, 12, vc_msg, 1, 10000);        // Start execution
18
19 memcpy(vpmBuffer_2, cpu_map->data + 60+480, 120); // Fetch vpmBuffer_2
20
21 ...
```

Listing 3.13: Data Setup for 1 TMU and 2 VPM Buffers

The base address of each seperate buffer - VPM and TMU buffer alike - has to be transmitted to the QPU via a uniform, which is done in a straightforward sequential fashion referencing `vc_data` which again represents the base address of the CPU memory map - termed `cpu_map` - mirroring the GPU memory. The base addresses of each separate buffer is stated in relation to `vc_data` as it points to the only buffer of the CPU memory map into which all data intended to transmit has to be copied into. At last is an eventual memory fetch shown in line 19, fetching 120 GPU processed bytes into `vpmBuffer_2`.

4. Implementation of AES

The first section of this chapter will outline the exact manner of how the data required by AES is made available to the GPU and therefore how excatly the kernel driver was accessed. Section 4.2 describes the single components of the the AES GPU implementation while the last section will outline occured problems, which mostly stem from problems with the VPM.

The block cipher mode of operation for the AES algorithm was chosen to be the ECB mode, since its simplicity and property of trivial parallelizability allow for the most promising showcase of the GPU's potential. Because the ECB mode treats every block in reclusion is parallelization achievable by simply assigning each block that is supposed to be encrypted or decrypted to a different QPU, which then applies the chosen algorithm. Regarding the AES block cipher algorithm are the AES variants utilizing a 128-bit key and a 256-bit key realized in this thesis, as they make up both sides in regards to the security vs speed tradeoff.

4.1. Implementation of AES on the CPU side

To access the GPU it first of all has to be set up correctly as outlined in chapter 3.3.1. Data that is passed to the GPU is made up of the assembled instructions, a 1024-byte S-box, 176-240 bytes for the expandedKey and 192 bytes of plain- or ciphertext that is to be processed. The assembled instructions total to 5304 bytes when transmitting the encryption algorithm and to 12680 bytes when transmitting the decryption algorithm. The S-box and expandedKey are supplied to the QPUs via the TMU, as only read access to those buffers is required. Because the plain- or ciphertext will be processed and therefore modified on the QPUs, are those texts transmitted via the VPM as it is the only read and write memory available allowing for a later fetch of the processed results. The plain- or ciphertext is therefore divided into equal parts that will correspond to the VPM segments. Because the GPU implementation of AES utilizes all 12 QPUs is the data transmitted via VPM therefore divided into 12 segments each making up 16 bytes, which is exactly the size of one block for the AES block cipher algorithm. The whole process of data transmission from the CPU to the GPU is visualized in illustration 4.1.

The S-box is transmitted as a 1024-byte buffer even though the S-box actually holds no more than 256 bytes of unique data and is therefore implemented as only a 256-byte buffer in regular CPU implementations. Though for the GPU implementation of AES was the S-box expanded by seperating each byte of the 256-byte S-box with 3 nulled bytes. This way does the S-box correspond naturally to the returned 32-bit of the TMU, which in turn saves 752 bytes of additional instruction code while adding 768 byte of S-box buffer. The resulting 1024-byte S-box implementation - as again outlined in the QASM implemention of `sub_bytes` and `inv_sub_bytes` - comes significantly ahead of the 256-byte S-box implementation.

The necessary key expansion for the 176-240 byte buffer of the expandedKey is performed first on the CPU and not parallelized on the 12 QPUs because the key expansion is barely parallelizable, but certainly not on the Raspberry Pi GPU. As outlined in chapter 2.1.1 is the calculation of each round-key highly dependent on the previous round-key, making the

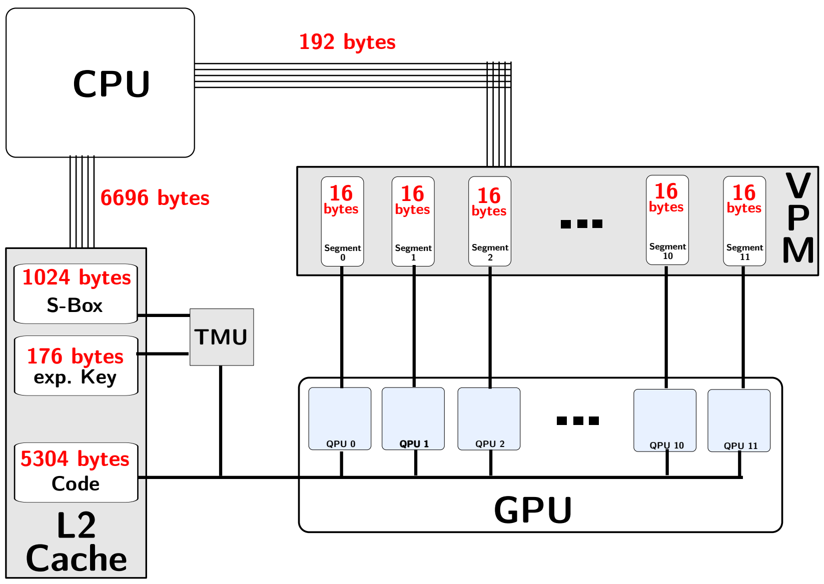


Illustration 4.1.: Visualization of the Transmissision of every Buffer to the GPU

calculation of different round-keys necessarily sequential. Parallelizing the calculation of a single round-key however barely makes sense, as a single-round key consists of only 16 byte. Meaningful parallelization on 4 processing units would require immensely faster inter process communication than ALU processing capabilities on all processing units, which simply is not given on the Raspberry Pi. Therefore is the key expansion for the AES algorithm performed on the CPU.

Noteworthy, especially in regards to the occured problems with the AES implementation, is the total amount of data transmitted from the CPU to the GPU every time the GPU receives 192 byte to process. For each encryption does the CPU transmit 6696-6760 bytes, while for the each decryption 14072-14136 bytes are transmitted.

4.2. Implementation of AES on the GPU side

Besides the key expansion does AES consist of four further functions according to the algorithm's illustration 2.2 in chapter 2.1. As each QPU receives one block via the VPM it has to perform the listed four functions in stated order. The upcoming sections will detail how each QPU receives its block and stores it back, how the control flow is realised and how each of the four functions was translated to QASM. All upcoming instructions are executed identically on each QPU and differ only in the assigned VPM memory, which is automatically differently assigned by the VCM Hardware though the encoded instructions are identical.

AES Preparations in QASM

The QPUs start by loading the address of the S-box buffer, expandedKey buffer and block buffer via uniform read into the registers `ra29-ra31`. Following are multiple load immediate instructions, which load constants into the registers rb0-rb8 as well as a counter value to ra28. The constants are necessary values throughout the AES algorithm and contain immediates larger than 6 bits or immediates for instruction with special conditional specifier as those instructions can't load small immediates. Register `ra28` specifies the required number of Cycling-Rounds according to illustration 2.2 and contains the value 9 in the AES128 implementation and the value 13 in the AES256 implementation. Both AES variants differ only the set value of this register when it comes to the QASM implementation.

Following up is the VPM load and read of the assigned block that is surrounded by VPM mutex acquire and release. Which block is assigned to which QPU is determined by the GPU driver. The QPU is set up to load 16 bytes to the VPM and then read those 16 bytes into registers `ra20-ra23`. Because AES is an algorithm that operates on single bytes and not 32-bit words which are stored in registers `ra20-ra23`, is a split of those 4-byte words necessary. Listing 4.1 showcase this necessary splitting at least for register `ra20`, which then has to be repeated for registers `ra21-ra23`. Eventually will registers `ra0-ra15` correspond to the 16 bytes that make up the *state* (see illustration 2.1) of the AES algorithm and on which the upcoming four QASM functions are performed.

```
## Receive S-box, expandedKey, block buffer address via uniform
or   ra31, ra32, 0;          nop    # ra31 = TMU address of S-box
or   ra30, ra32, 0;          nop    # ra30 = TMU address of expandedKey
or   ra29, ra32, 0;          nop    # ra29 = VPM address of block


## Load Cycling-Rounds counter and constants
ldi ra28, 9                         # Cycling-Rounds counter
ldi rb0,  0x000000ff                # Word splitting immediates
ldi rb1,  0x0000ff00
ldi rb2,  0x00ff0000
ldi rb3,  0xff000000
ldi rb4,  0x00000100                # Galois mask immediates
ldi rb5,  0x0000011b
ldi rb6,  16                        # TMU round-key load immediates
ldi rb7,  0
ldi rb8,  4                         # TMU S-box load immediate


## Acquire VPM mutex
or   ra39, ra51, rb39;       nop

## VPM DMA load setup, ID=1, MODEW=0, MPITCH=1, ROWLEN=4, NROWS=1, VPITCH=0, VERT=0,
ldi ra49, 0x81410000
or   ra50, ra29, 0;          nop
or   ra39, ra50, rb39;       nop

## VPM read setup, ID=0, NUM=4, STRIDE=1, HORIZ=0, LANED=0, SIZE=2, ADDR=0
ldi ra49, 0x00401200
or   ra20, ra48, 0;          nop    # ra20 = block[0-3]
or   ra21, ra48, 0;          nop    # ra21 = block[4-7]
or   ra22, ra48, 0;          nop    # ra22 = block[8-11]
or   ra23, ra48, 0;          nop    # ra23 = block[12-15]

## Release VPM mutex
or   ra51, ra39, rb39;       nop


## Word splitting of 32-bit words in ra20-ra23 to bytes in ra0-ra15
and  ra0,  ra20, rb0;        nop    # ra0 = block[0]
and  ra32, ra20, rb1;        nop
shr  ra1,  r0,   0x8;        nop    # ra1 = block[1]
and  ra32, ra20, rb2;        nop
shr  ra2,  r0,   0x10;       nop    # ra2 = block[2]
and  ra32, ra20, rb3;        nop
shr  ra3,  r0,   0x18;       nop    # ra3 = block[3]

...
```

Listing 4.1: QASM Preparations for AES

AES Control Flow in QASM

Control flow within the QASM implementation of AES is very straightforward as most functions are implemented inline for performance reasons. Merely a branch to repeat the Cycling-Rounds is undertaken after the second `add_round_key()` function. Using placeholders for functions does listing 4.2 outline the sequence of functions each QPU is undergoing.

```
<<AES Preparations>>

<<add_round_key()>>

cycle:

<<sub_bytes>>
<<shift_rows>>
<<mix_columns>>
<<add_round_key>>

## Branch to repeat Cycling-Rounds
sub    ra28, ra28, 1;       nop
brr.ze ra39, rb39, cycle          # branch if ra28 != 0 to cycle
nop    ra39, ra39, rb39;    nop
nop    ra39, ra39, rb39;    nop
nop    ra39, ra39, rb39;    nop

<<sub_bytes>>
<<shift_rows>>
<<add_round_key>>

<<AES Postprocessing>>
```

Listing 4.2: Control Flow in QASM AES

add_round_key() and inv_add_round_key() in QASM

For the `add_round_key()` and `inv_add_round_key()` function do the QPUs first of all fetch the current round-key from the TMU and load it to the registers `ra16-ra19`. The 4 round-key bytes in each register are then simultaneously split and added to the corresponding byte of state, as shown in listing 4.3. After the current round-key is fetched from the TMU is the expandedKey pointer increased to the the next round-key or decreased to the preceding round-key in case of decryption, during which the round-keys are applied backwards. Performing a complete `add_round_key()` function requires 53 instruction cycles.

```
## Preload round-key to be fetched
add ra56, ra30, 0;          nop
add ra56, ra30, 4;          nop
add ra56, ra30, 8;          nop
add ra56, ra30, 12;         nop

## Advance TMU expandedKey address to next round-key
add.tmu ra30, ra30, rb6;    nop

## Fetch round-key from TMU
or.tmu  ra16, r4,   rb7;    nop
or.tmu  ra17, r4,   rb7;    nop
or.tmu  ra18, r4,   rb7;    nop
or      ra19, r4,   rb7;    nop

## Split round-key word and xor with corresponding state bytes
and ra32, ra16, rb3;        nop   # r0 = round-key[0-3] & 0xff000000
shr ra32, r0,   0x18;       nop   # r0 = round-key[0]
xor ra0,  ra0,  r0;         nop   # block[0] = block[0] ^ round-key[0]

and ra32, ra16, rb2;        nop   # r0 = round-key[0-3] & 0x00ff0000
shr ra32, r0,   0x10;       nop   # r0 = round-key[1]
xor ra1,  ra1,  r0;         nop   # block[1] = block[1] ^ round-key[1]

and ra32, ra16, rb1;        nop   # r0 = round-key[0-3] & 0x0000ff00
shr ra32, r0,   0x8;        nop   # r0 = key[2]
xor ra2,  ra2,  r0;         nop   # block[2] = block[2] ^ round-key[2]

and ra32, ra16, rb0;        nop   # r0 = round-key[0-3] & 0x000000ff
xor ra3,  ra3,  r0;         nop   # block[3] = block[3] ^ round-key[3]

...
```

Listing 4.3: Implementation of `add_round_key()` in QASM

sub_bytes() and inv_sub_bytes() in QASM

The `sub_bytes()` function is simple in that it replaces the state-byte with the corresponding value the S-box holds at the index being the value of the state-byte. However because the GPU being a 32-bit architecture is the S-box not implemented as a 256-byte buffer but rather as a 1024-byte buffers for reasons outlined in the preceding chapter 4.1. This enables the `sub_bytes()` function and its inverse counterpart to execute a single byte substition within 3 instead of 6 instruction cycles and saves the mentioned 752 bytes of instruction code. The increase in total size of the instruction code by 16 bytes is by far redeemed by halving the required time for a single `sub_bytes()` execution, which in its optimized form takes a total of 49 instruction cycles to complete.

```
nop      ra39, ra0,  rb8;     mul24 rb33, ra0, rb8;

add      ra56, ra31, r1;      nop                         # TMU_load(&sbox + block[0]*4)
nop.tmu  ra39, ra1,  rb8;     mul24 rb33, ra1, rb8;
or       ra0,  r4,   0;       nop                         # block[0] = sbox[block[0]]

add      ra56, ra31, r1;      nop                         # TMU_load(&sbox + block[1]*4)
nop.tmu  ra39, ra2,  rb8;     mul24 rb33, ra2, rb8;
or       ra1,  r4,   0;       nop                         # block[1] = sbox[block[1]]

...
```

Listing 4.4: Implementation of `sub_bytes()` in QASM

shift_row() and inv_shift_row() in QASM

`shift_row()` and `inv_shift_row()` are the simplest functions in the reference C implementation and are equally straightforward in their QASM translation. Both functions assign each state-byte a different state-byte according to illustration 2.3 in chapter 2.1.1 and are performed by saving the first state-byte in a temporary accumulator value and then sequentially filling the correct state-byte into the state-byte that is currently backed up. A complete `shift_row()` function requires only 16 instruction cycles.

mix_columns() and inv_mix_columns() in QASM

There are many examplary implementations of a matrix multiplication in regular Assembly, providing good guidance for a QASM implementation of the matrix multiplications of `mix_columns()` and `inv_mix_columns()` that multiply the current state of the block with the matrices shown in illustration 2.4. The challenge of the `mix_columns()` functions however consists in the fact that the matrix multiplication - which is straightforwad in QASM - is performed in the GF (2^8). Hence is the realization of the galois multiplication in QASM the actual difficult aspect of the `mix_columns()` functions.

The most comprehensible way of realizing the galois multiplication is by translating the *Russian Peasant Multiplication* algorithm [Wik17d] to QASM and calling it by function. Listing A.6 in the appendix shows the most direct translation of the algorithm to QASM requiring many conditional branches and a total of 20 instructions for only a single while loop - of which there are several necessary for the finished galois multiplication. The function takes inputs from accumulator `r0` and `r1` and returns the result in acummulator `r2`.

However because there are only 7 different values for the second argument of the galois multiplication - as there are only seven different values in the `mix_columns()` matrices - can the galois multiplication be fitted to those arguments and conditional branches can be evaluated beforehand since the second argument is known. Are furthermore the remaining conditional branches replaced by intricate logic shifts can the galois multiplication in QASM become highly performant and can be executed in 7 to 18 instruction cycles, depending on the second argument. Listing 4.5 shows such a highly optimized implementation, which can perform a complete galois multiplication of arbitrary input with the value `0x03`. Listing A.7 in the appendix shows another optimized galois multiplication with the second argument being a larger number (`0x0e`) and therefore requiring 18 instruction cycles.

```
## p = gal_mul(a, 0x03), r2 = p, r0 = a
xor ra34, r0, 0x00000000;   nop       # p = p ^ a

shl ra32, r0, 1;            nop       # a = a << 1

and ra33, r0, 0x100;        nop       # if (a & 0x100) then a = a xor 0x11b
shr ra33, r1, 8;            nop
nop ra39, r1, 0x11b;        mul24 rb33, r1, 0x11b
xor ra32, r0, r1;           nop

xor ra34, r0, r2;           nop       # p = p ^ a

## Result of gal_mul(r0, 0x03) now in r2
```

Listing 4.5: Implementation of p = gal_mul(a, 0x03) in QASM

The `mix_columns()` and `inv_mix_columns()` functions can be declared the by far most expensive function in the QASM implementation of the AES algorithm. A single round of the `mix_columns` function requires 272 instruction cycles, while the inverse function even requires 1200 instruction cycles due to the inverse function utilizing significantly higher multiplicants in its matrix and therefore requiring by far more elaborate galois multiplications.

To put this high weight of this function in perspective is it important to state the second most expensive function - the `add_round_key()` function - merely requires a total of 53 instruction cycles. Adding all instructions cycles of 11 calls each to the `inv_add_round_key()`, `inv_sub_bytes()` and `inv_shift_rows()` functions required for a single execution of the AES algorithm, does the total come to merely 1298 instruction cycles. All the while does a single call to the `inv_mix_columns()` require 1200 instruction cycles, which adds up to 10800 instruction cycles over the course of a complete single exeuction of AES. Consequently can be concluded that each QPU will spend 88% of it's time processing solely the `inv_mix_columns()` function when decrypting. During encryption, the `mix_columns()` function still demands a share of 47% of the total processing time.

AES Postprocessing in QASM

The postprocessing mirrors the preparations each QPU undergoes before the execution of the block algorithm. The postprocessing consists of three parts. First of all are the statebytes which are saved in the registers `ra0-ra15` merged back to 32-bit words and saved into registers `ra20-ra23`. Following this is the writing of those values into the VPM and then initiating a VPM store, which results in the processed block being sent back to the CPU. All of those VPM operations are necessarily surrounded by VPM mutex acquires and releases. As in the preparation phase of the AES algorithm, executes each QPU the same VPM configuration instructions but gets automatically assigned different segments of the VPM by the VCM. Finally is each QPU triggering a host interrupt and the `qpu_execute()` call in the driver returns when all QPUs triggered that interrupt.

```
...

shl ra33, ra13, 0x8;        nop
or  ra32, ra12, r1;         nop
shl ra33, ra14, 0x10;       nop
or  ra32, r0,   r1;         nop
shl ra33, ra15, 0x18;       nop
or  ra23, r0,   r1;         nop     # ra23 = block[12-15]


## Acquire VPM mutex
or  ra39, ra51, rb39;       nop

## VPM write setup, ID=0, STRIDE=1, HORIZ=0, LANED=0, SIZE=2, ADDR=0
ldi rb49, 0x00001200
or  ra48, ra20, 0;          nop     # block[0-3]
or  ra48, ra21, 0;          nop     # block[4-7]
or  ra48, ra22, 0;          nop     # block[8-11]
or  ra48, ra23, 0;          nop     # block[12-15]

## VPM DMA store setup, ID=2, UNITS=1, DEPTH=4, LANED=0, HORIZ=1, VPMBASE=0, MODEW=0
ldi rb49, 0x80844000
or  rb50, ra29, 0;          nop
or  ra39, ra39, rb50;       nop

## Release VPM mutex
or  ra51, ra39, rb39;       nop

## Trigger interrupt to finish the program and signal host
or  ra38, ra39, rb39;       nop
nop.tend  ra39, ra39, rb39; nop
nop ra39, ra39, rb39;       nop
nop ra39, ra39, rb39;       nop
```

Listing 4.6: QASM Postprocessing for AES

4.3. Summary and Problems of AES Implementation

Problems regarding the implementation of AES can be attributed to the Problems regarding VPM access, outlined in chapter 3.1.4. The most suitable implementation of the VPM management would be to transmit 48 KB at once from the CPU to the VPM by the VPM load command, filling up each 4 KB segment with blocks that are intended to be processed with the AES algorithm. The QASM implementation of the AES algorithm would be identical to the current one - each QPU reading 16 bytes, processing it, then writing it back - except for branching back to the start of the algorithm for a total of 256 times, each time increasing the VPM addressing to the next block that is supposed to be processed. After each 16-byte block in the 4 KB VPM segment is processed would the data be transmitted back to the CPU via a VPM store command.

However, as detailed in chapter 3.1.4 does the VPM currently not support multiple VPM reads that are seperated by a VPM mutex release and reacquire, which in turn is an absolute necessity for an actual parallelization. Therefore is the QASM implementation of AES forced to load and read the data that is supposed to be processed within a single VPM mutex acquisition, limiting the implementation to assign each QPU only 16 bytes of ciphertext. Reading a second 16-byte block to the QPU within a single VPM mutex acquisition would be possible, though this would cause a serious increase in instruction size making it barely worthwhile.

Instead was the problem solved in the clearer but unsatisfactory fashion of repeatedly transmitting 192 byte to and from the GPU, while still keeping the acquired GPU communication pipe and the GPU configuration active. This way is the signifcant overhead of starting and setting up the GPU unnecessary if more than 192 byte of plain or ciphertext is intended to be processed. Nevertheless causes this current workaround implementation massive drawbacks, since it is necessary to transmit the 6504 bytes of overhead (see chapter

4.1) each time 192 bytes of data is to be processed on the GPU. When decrypting does the overhead even amount to a staggering 13880 bytes, which is 72 times more than the transmitted data one is actually interested in. The overhead of encryption still outweighs the actually interesting transmitted data by a factor of 33.

Besides this obvious performance drain of being limited to process only 192 byte at once, can AES be efficiently implemented in the QASM programming language. Even though the `mix_columns()` functions require an enormous amount of processing time, are they realized efficiently if one intents to process the galois multiplication by computing it. On the other hand can the `mix_columns()` functions be made massively more efficient by realizing the galois multiplication via multiple lookup tables, as done in the CPU implementations against which the result of the GPU implementation have been thoroughly tested. Those lookup tables would require 6 additional buffers of 256 byte each being transmitted to the GPU via the TMU, though it would bring down the `mix_columns()` and `inv_mix_columns()` function down to the 49 instruction cycles needed for the `sub_bytes()` function.

However the GPU implementation of AES was done according to NIST's FIPS 197 official AES description, wherein the galois multiplication is not realized by lookup tables but rather by computation. Aside from this performance issue does the resulting AES implementation work out well, scaling to arbitrary input sizes and consistently returning correctly processed data. During all benchmarks introduced in the subsequent chapter, totaling the encryption and decryption of multiple megabytes of data, has not one bit that was returned by the GPU implementation deviated from the industry standard encryption performed by OpenSSL.

5. Evaluation

The Raspberry Pi GPU has a theoretical maximum performance of 24 SP GFLOPS, allowing for a potential speed-up of 1714% compared to the 1.4 SP GFLOPS of the CPU. This chapter is going to evaluate if this speed-up is possible with the use case being data encryption using the AES block cipher algorithm developed in the course of this thesis and introduced in the preceding chapters. In order to evaluate the GPU implementation of AES is it compared to various quasi-standard implementations of AES utilizing the CPU. Section 5.2 will introduce the conducted experiments, followed by the discussion of them in the last section of this chapter.

5.1. Testsetup

All tests were executed on a Raspberry Pi Model 1 B that was produced past the 15. October of 2012 and therefore features 512 MB of SDRAM. The used operating system was the official Raspbian OS in the jessie lite version. It was used in the release version 2017-04-10, which features a Linux Kernel in version 4.4.50 and a gcc compiler in version 4.9.2. The program testing the GPU and CPU libraries of AES is compiled with the enclosed makefile, automatically assembles the existing QPU code and is executed with `sudo a.out`. The library testing program will then measure the required time of each library's function using `clock_gettime()` and write out the results to a `.csv` file for logging purposes.

The GPU implementation of AES has been tested in 2 slightly different versions in order to provide more expressivenes in the conducted experiments, while four different CPU libraries realizing AES have also been tested for comparison.

Both GPU implementations differ in the aspect of setting up the GPU before starting the benchmark or not. In the version called 'GPU, preloaded Pipe' is the pipe acquisition to the GPU as well as most of the GPU memory setups done seperately beforehand, leaving only the key expansion, memory copy of plaintext and transmission kickoff to the timing function. This benchmark is intended to represent the actual usage of the GPU implementation in a real scenario, as the whole GPU setup can simply be done beforehand when the GPU implementation is loaded as an encryption kernel module or similar. On the other hand do the benchmarks of the standard GPU implementation, simply called 'GPU', include the GPU setup in its timing function.

The four different CPU libraries realizing AES come down as follows: OpenSSL in version 1.01 optimized for the ARMv6 architectur with architecture specific assembly [Deb17]. Rijndael in its original form written in assembly-oriented C as proposed by its creators at the competition of finding the next Advanced Encryption Standard in 2001 [Rij02]. Tiny-AES 128, one of the most popular AES implementations in C for embedded devices though only available for AES-128 (github version of the 23. June 2017). A custom reference implementation of AES, strictly implemented in C according to NIST's AES explaining document FIPS 197 [NIS01].

The testvector examined in this thesis is restricted to the 128-bit key and 256-bit key variants of AES, utilizing the ECB block cipher mode of encryption and comparing five reference implementations. Possible input sizes were chosen to be 192 byte, 1536 byte and 10KB. This testvector is visualized in illustration 5.1.

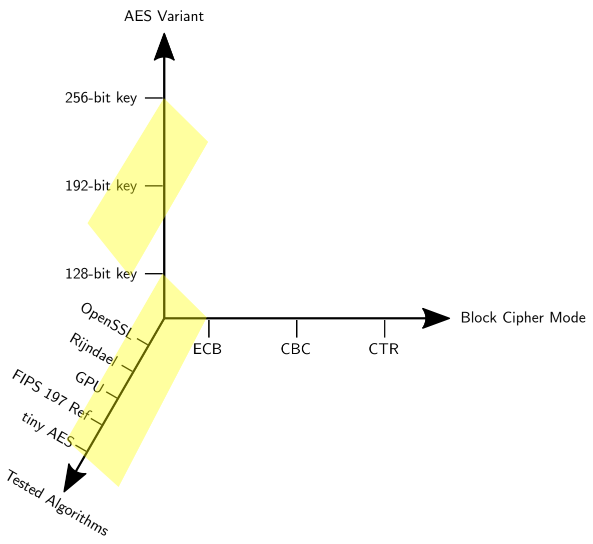
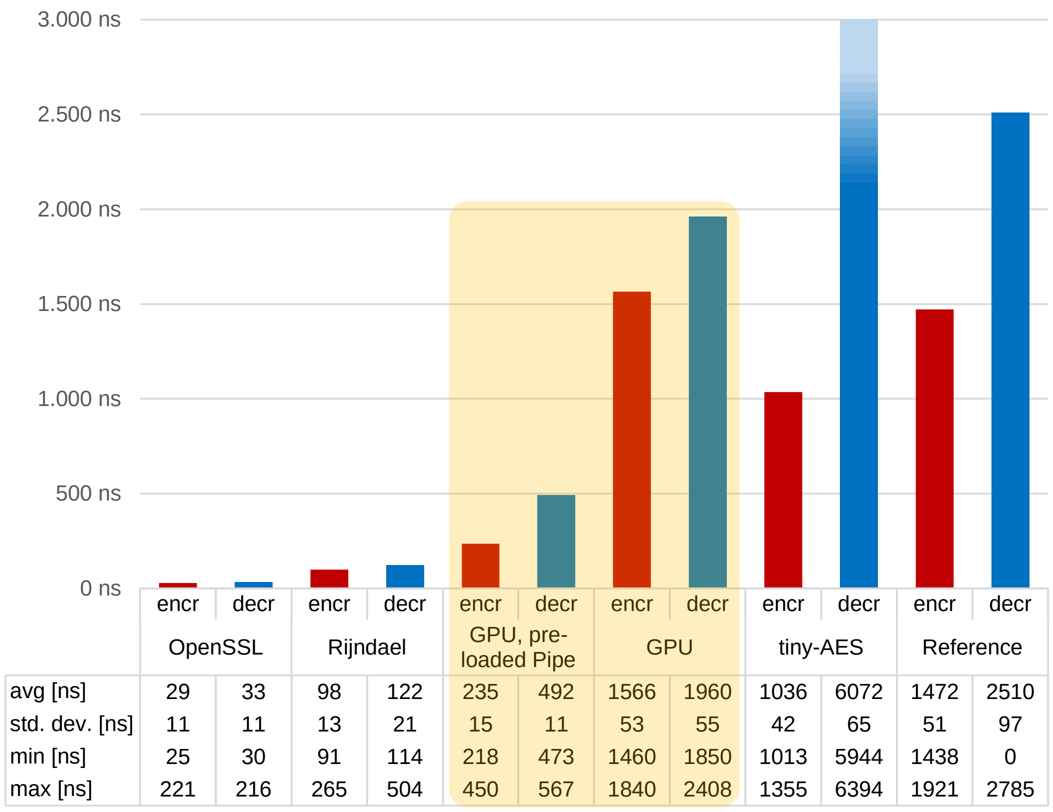


Illustration 5.1.: Testvector of Conducted Experiments

5.2. Experimental Results

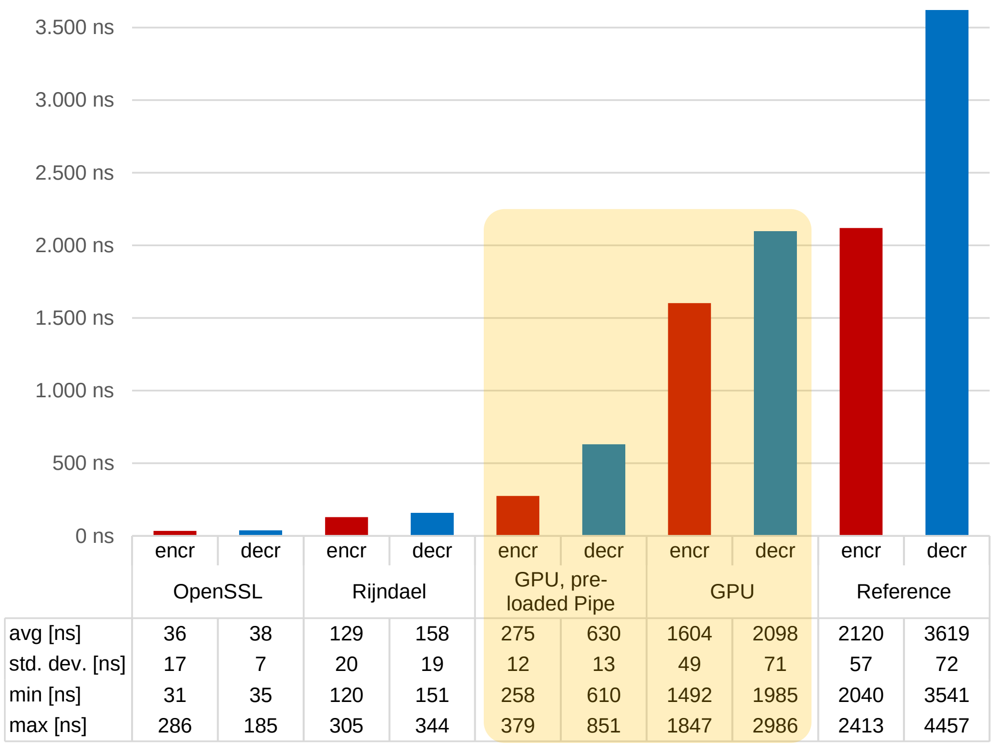
The benchmark results of the first experiment is shown in illustration 5.2. It depicts the encryption (red) and decryption (blue) of 192 bytes in a bar chart visualizing the required time in nanoseconds on the y-axis and listing 5 different implementations of AES in the 128-bit key variant. The showcased timings are made up of the resulting averages of 1000 independent runs of those implementations. The two leftmost AES implementations are OpenSSL and Rijndael. The two centered AES implementations both depict the GPU AES implementation created for this thesis, differing in the aspect of preloading the GPU. The two rightmost AES implementations represent tiny-AES the custom reference implementation of AES according to FIPS 197.

The table below the bar graph depicting the achieved benchmark results, details further benchmark statistics in nanoseconds. Topmost is the exact average timing of the 1000 runs listed, based on which the bars in the graph are visualized. Below is the statement of encountered standard deviation in nanoseconds. For the GPU implementations - including all benchmarks listed below - did the standard deviation never exceed a relative value of 3%, indicating an obviously needed reliable communication with the GPU. The runs requiring the least and most time are also listed at the bottom of the table to provide further interesting statistical information, especially when combined with the standard deviation.



	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr
	OpenSSL		Rijndael		GPU, pre-loaded Pipe		GPU		tiny-AES		Reference	
avg [ns]	29	33	98	122	235	492	1566	1960	1036	6072	1472	2510
std. dev. [ns]	11	11	13	21	15	11	53	55	42	65	51	97
min [ns]	25	30	91	114	218	473	1460	1850	1013	5944	1438	0
max [ns]	221	216	265	504	450	567	1840	2408	1355	6394	1921	2785

Illustration 5.2.: Benchmark of AES128, 192 Bytes Input



	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr
	OpenSSL		Rijndael		GPU, pre-loaded Pipe		GPU		Reference	
avg [ns]	36	38	129	158	275	630	1604	2098	2120	3619
std. dev. [ns]	17	7	20	19	12	13	49	71	57	72
min [ns]	31	35	120	151	258	610	1492	1985	2040	3541
max [ns]	286	185	305	344	379	851	1847	2986	2413	4457

Illustration 5.3.: Benchmark of AES256, 192 Bytes Input

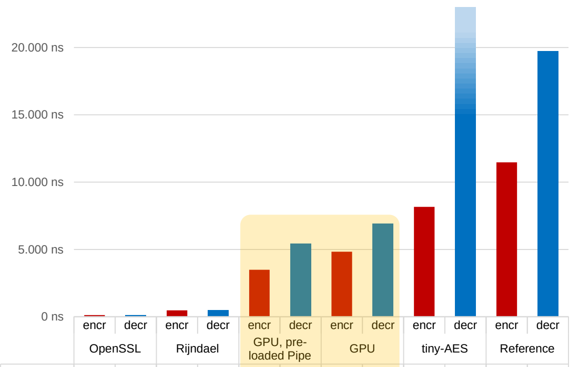
An important property of the visualized graphs is the fading out graph bar for the decryption of tiny-AES. Because the required time for decryption with the tiny-AES algorithm far outweighs the required time of other algorithms, does this completely ruin the scale and usability of the resulting graph. As this overshadowing is constant throughout all benchmarks of the AES algorithm in its 128-bit key variant, is the readjustment of the scale while simultaneously fading out the bar graph the more comprehensible practice. The exact required time of the tiny-AES decryption can still be seen in the statistics below the bar graph.

Furthermore does Illustration 5.3 depict the encryption and decryption of the same input size of 192 bytes - visualized in the same manner. Though because the popular implementation tiny-AES does not feature a 256-bit key variant, does the 5.3 only illustrate 4 instead of 5 different implementations.

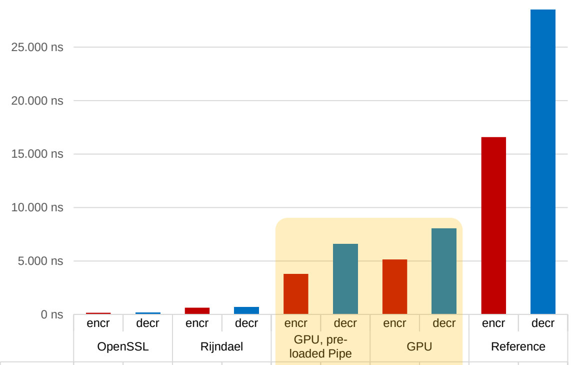
Significant is the assessment that the GPU implementation of AES-256 does better in relation to its competing implementations than AES-128. For one does the GPU implementation of AES-256 beat out the FIPS 197 reference implementation even when the GPU setup is taken into the timing account - which is not the case for the AES-128 GPU implementation. Additionally does the AES-256 implementation better in relation to the highly optimized standard implementations. The 'GPU, preloaded pipe' implementation of the AES-256 version takes 2.1 and 7.6 times longer than the Rijndael and OpenSSL implementations respectively, while the same implementation of the AES-128 version takes 2.4 and 8.1 times longer.

Considering the differences in absolute required time between the AES-128 and AES-256 implementaiton does the difference between both presetup implementations come down to 40ns.

Moreover is the input size of 192 bytes not the only input size tested. Illustration 5.4 showcases the benchmarks of AES in the 128-bit key and 256-bit key variant for an input size of 1536 byte, while illustration 5.5 showcases the benchmarks for an input size of 10KB. The input size of 1536 byte was chosen as this incorporates the maximum payload size of an ethernet frame, which is 1500 bytes, and is therefore interesting in regards to the real applicability of the GPU implementation. The input size of 10KB was chosen as this input size roughly marks the point after which there is basically no discernible difference in the required time of the regular GPU implementation and the presetup one.



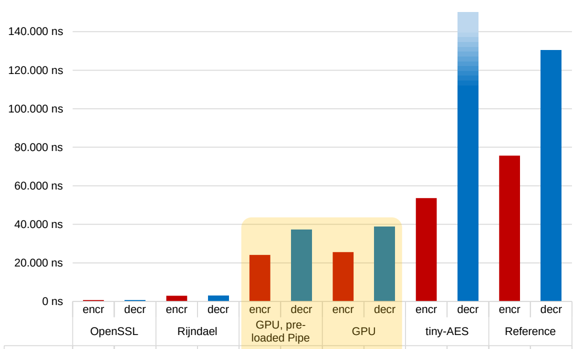
	OpenSSL		Rijndael		GPU, pre-loaded Pipe		GPU		tiny-AES		Reference	
	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr
avg [ns]	124	128	476	502	3489	5444	4830	6921	8159	48083	11472	19728
std. dev. [ns]	12	15	26	28	132	93	88	98	70	161	108	136
min [ns]	118	123	464	488	0	5138	4492	6557	8040	47902	11400	19605
max [ns]	274	346	712	791	3780	6781	5528	8238	9241	50611	13450	22262



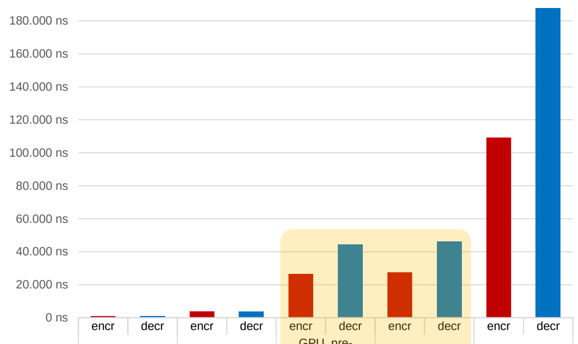
	OpenSSL		Rijndael		GPU, pre-loaded Pipe		GPU		Reference	
	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr
avg [ns]	156	162	629	659	3798	6562	5142	8023	16593	28490
std. dev. [ns]	15	14	28	32	74	98	87	95	90	165
min [ns]	150	157	614	643	3508	6265	4775	7667	16416	28337
max [ns]	378	384	803	1083	4220	7455	6060	8824	18010	31970

Illustration 5.4.: Benchmark of AES128 (Left) and AES256 (Right), 1536 Bytes Input

When examining the average timing of those benchmarks does the fact draw attention that the required average time does not scale in unison with the input sizes. Taking AES-128 for example and comparing the benchmarks of the 192 byte input and the 1536 byte input can be assessed that although the input size is increased by a factor of 8, does the encryption of the presetup GPU version take 14.8 times as long. Though the exact opposite is the case when examining the regular GPU version without a preceding setup, as the encryption for 1536 bytes takes only 3.1 times as long as the encryption for 192 bytes.



	OpenSSL		Rijndael		GPU, pre-loaded Pipe		GPU		tiny-AES		Reference	
	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr
avg [ns]	747	754	2914	2937	24122	37119	25599	38700	53649	318141	75632	130274
std. dev. [ns]	43	30	59	61	247	188	250	221	101	574	166	233
min [ns]	727	737	2847	2872	22858	36664	24783	37872	53456	317597	75427	130111
max [ns]	992	894	3043	3075	24475	37510	25952	39086	53964	322800	76879	132143



	OpenSSL		Rijndael		GPU, pre-loaded Pipe		GPU		Reference	
	encr	decr	encr	decr	encr	decr	encr	decr	encr	decr
avg [ns]	950	968	3871	3904	26605	44615	27566	46179	109243	187895
std. dev. [ns]	34	52	75	103	260	252	330	268	133	358
min [ns]	928	939	3785	3813	25559	43829	26194	45205	109083	187094
max [ns]	1074	1330	4095	4682	26984	45658	28225	46908	109737	189508

Illustration 5.5.: Benchmark of AES128 (Left) and AES256 (Right), 10KB Input

5.3. Discussion

As the first and most obvious assessment can be said that the GPU implementation of AES is in all variants significantly lacking behind the industry standard implementation of OpenSSL and the original Rijndael implementation. Though it is important to note that both of those implementations are highly optimized in their utilized programming language as well as in their AES algorithm procedure, with OpenSSL even featuring ARMv6 specific assembly instructions.

By far more significant though is the fact that both implementations feature a highly optimized `mix_columns()` function, which are the weak spot of straightforward AES implementations as outlined in chapter 4.2. Both implementations realize the required galois multiplication by preprocessing the possible outcomes and providing those via lookup tables, totaling 1536 bytes alone. In return do both optimized implementations replace the very processing intensive galois multiplication by a simple and massively faster table lookup. Attesting to this optimization is the fact that both encryption and decryption of the OpenSSL and Rijndael implementation take up very similar time, as aside from the differing effort for the galois multiplication both encryption and decryption have very similar instructions.

Both GPU implementations as well as the unoptimized CPU implementations on the other hand realize the galois multiplication by computing it. A detailed instruction analysis of the `mix_columns()` functions in section 4.2 however came to the conclusion that this computing of the galois multiplication makes up 47% of the QPUs total processing time when encrypting and staggering 88% when decrypting. The required time for the `mix_columns()` function would be divided by a factor of 5.5 when encrypting and even by a factor of 24.5 when decrypting if the function would be implemented using a lookup table only requiring 49 instruction cycles as stated in section 4.3.

This in turn decreases the meaningfulness of comparing the lookup table optimized OpenSSL and Rijndael implementation with the processing intensive GPU, tiny-AES and FIPS 197 reference implementation.

An interesting and important aspect of the GPU implementation is the difference in required time caused by a preceding setup of the GPU, which can be averaged to about 1300ns. As this preceding setup of the GPU is very well realizable by loading an encryption kernel module at system boot, are the benchmark results of the presetup GPU implementation signficantly more relevant and important than those of the regular GPU implementation. Though more importantly does this setup time for the GPU seem to be the reason for why not only the required time doesn't scale with input size, but rather the regular GPU and presetup GPU implementation scaling in opposing fashions (one correlating significantly negative, the other significantly positive).

One assessment of the previous section was that between the 192 byte and 1536 byte benchmarks the input size was increased by a factor of 8, though the presetup implementation took 14.8 times longer while the regular implementation correlated negative and required only 3.1 times more time. This assessment can be found in a similar manner with the 256-bit key variant, however barely with the input size scaling comparison between the 1536 byte and 10KB benchmarks. This behaviour is owed to the fact that with 192 byte as an input size, the data only needs to be transmitted to and fetched from the GPU once, not requiring a reset of the GPU for it to be ready to receive another batch of data to be processed. This GPU reset is necessary for a repeated transmission of data while still keeping the GPU set up the same way.

Since this GPU reset takes 200ns and is required before every repeated transmission, does in return the input size now scale very closely with the required time. Furthermore does this also explain the negative correlation of the required time of the regular GPU implementation with a higher input size. Because only the first transmission of data requires 1300ns for the setup of the GPU, do additional data transmissions only require 200ns of reset time as well as the average required time determined for 192 byte if the GPU would already be presetup. In the case of encryption with AES-128 does this average required time come down to 235 as shown in illustration 5.2 Calculating the resutling expected time for a 1536 byte encryption with AES-128 is therefore done by adding the setup time (1300ns) with the time for encrypting 192 byte while considering the GPU reset time (235ns + 200ns) and then multiplying it by 8, as 192 byte is one eighth of 1536 byte. The calculated expected time therefore adds up to 4780ns, which comes very close to the benchmarked 4830ns in illustration 5.4, further validating that input size and required time do actually scale under consideration of certain properties.

At last is an important conclusion to be drawn by comparing the benchmark results of AES-128 and AES-256. Not only does the GPU implementation of AES-256 perform better than AES-128 in relation to their compared CPU implementations. Going further can the comparison of both variants even give some indiciation about the required time for a single Cycling-Encryption-Round.

The time difference between the GPU implementation of AES-128 and AES-256 comes down to 40ns when encrypting. However both implementations barely differ in the amount of data transmitted to the GPU in order to execute the GPU algorithm. When encrypting does the AES-128 variant transmit 6696 bytes while the AES-256 variant transmits 6760 bytes, as outlined in section 4.1. Therefore can the time difference of 40ns be equalised to the execution of the additional 4 Cycling-Encryption-Rounds undertaken by the AES-256 algorithm. Hence does the execution of a single Cycling-Encryption-Round take 10ns on the GPU with the current implementation.

Assuming that the time required for the execution of the Initial-Encryption-Round and Final-Encryption-Round combined equals the required time of a single Cycling-Encryption-Round (although they fall about 200 instruction cycles short). The AES-256 algorithm can then be stated as performing 14 Cycling-Encryption-Rounds and therefore requiring 140ns on the GPU. The remaining 135ns that were required for a complete execution are split into performing the key expansion, transmitting 6760 bytes from the CPU to the GPU and loading the state bytes from VPM memory to the GPU register. The GPU implementation of the AES algorithm therefore requires about the same time as the Rijndael implementation does on the CPU, once the data has been transmitted to the GPU.

This chapter shall be concluded by performing the thought experiment of replacing the processed galois multiplication by a galois multiplication realized with lookup tables. The `mix_columns()` function would be implemented nearly identical to the `sub_bytes()` function, requiring 53 instruction cycles. Whereas the current implementation of `mix_columns()` requires 272 instruction cycles. If the required instruction cycles of the `mix_columns()` function come down to 53 instructions would this in turn decrease the required total instructions for a Cycling-Encryption-Round from 390 instructions (53 + 16 + 272 + 49, see section 4.2) to 171 instructions (53 + 16 + 53 + 49). This reduces the required instruction cycles by 56 % and would in turn bring down the algorithm execution on the GPU down to 61ns.

To summarize can therefore be said that the OpenSSL implementation on the CPU significantly outperforms the GPU implementation of AES. This would still be the case if the galois multiplication would be implemented more efficiently by lookup tables and the VPM problem would be solved. The solution of the VPM problem would make the 135ns for data transfer insignifcant as it would only be required twice for arbitrary amounts (up to 48KB) of data, however would the application of the AES-256 algorithm - in this scenario requiring 61ns - still require 25ns more than OpenSSL for every 192 bytes. The Rijndael implementation on the other hand would be greatly outperformed and the performance advantage to other reference implementations vastly increased.

6. Summary and Future Work

This thesis set out to investigate the potential of the Raspberry Pi GPU by parallelizing the AES block cipher algorithm in GPU specific assembly. Greatly improving the existing way of accessing the GPU as well as greatly improving the assembler provided an important foundation to realize this goal. Not only can the assembler for the QASM programming language now for the first time encode QASM instructions correctly, but those can finally be executed reliable on all 12 QPUs. The assembler was furthermore provided with a significant expansion of important functionalities as well as an extensive documentation detailing the QASM programming language, how exactly the instructions are encoded and how to properly configure and address the GPU memory. This enables an excellent starting point for future research conducted on the GPU.

All in all can be said that in the course of this thesis an extensively documented interface to the GPU was created, that allows for reliably access to a general purpose multicore system on a simple IoT device.

This enabled the assessment of the GPU's capabillity, which in fact does show great performance potential and therefore resolves the original question of this thesis. This great potential can be concluded from several facts. One of those facts being that although the GPU implementation of AES realizes the galois multiplication by processing it instead of utilizing a lookup table and therefore spends 47% of its total encryption processing time with the `mix_columns()` function, the GPU is still able to perform a complete AES block cipher algorithm on par with the highly optimized Rijndael CPU implementation once the required data is located in GPU memory. Though because the transmission of data from the CPU to the GPU is obviously a necessary part of the GPU implementaiton, does the CPU outperform the GPU implementation at the moment.

Once the galois multiplication is realized with lookup tables does the GPU performance even come close to that of OpenSSL - the industry's most trusted and optimized implementation. All of those results were achieved with only a first draft of the GPU's AES implementation, which certainly offers more possibility of improvement than just the galois multiplication. Therefore owing the impressive results mostly to the fact that it could be implemented in parallel.

On the other hand - and this makes up the second fact of why the GPU shows great performance potential - is the slow transmission of data to the GPU, causing the defeat against the CPU implementations, due to the massive data overhead required for the processing of a single 192 byte batch. The overhead required to encrypt 192 byte of plaintext amounts to 33 times as much as the actually important plaintext data - even increasing to a factor of 72 when transmitting 192 byte for decryption. This problem however is caused by an only temporarily unsolved configuration problem of the VPM, for which most certainly there is a yet undisclosed solution.

Once the configuration problem of the VPM is solved, does this also resolve any issues with the crippling amount of overhead, truly unleashing the GPU's potential. The moment this is the case will the possible amount of data that can be processed within a single transmission to the GPU explode from 192 byte to 48KB. Combined with additional methods of speed-up - aside the galois multiplication - that are certainly possible, does the GPU have a realistic chance of computing AES faster than the OpenSSL implementation on the CPU. Nonetheless does the GPU offer in any way an excellent speed-up potential for algorithms, as the parallelization of the algorithm on the GPU most likely is more easily accomplished than highly optimizing the algorithm on the CPU.

Furthermore is it important to record that parallelizing the AES block cipher algorithm is not the best way to show off the GPU's maximum potential, despite the GPU showing exactly that. First of all does the AES algorithm require many accesses to lookup tables - even in the current unoptimized version is a frequent read from the S-box and expandedKey lookup tables necessary. Though the usual bottleneck of multicore systems can be identified as the memory bus, from which can be concluded that an algorithm that is more processing intensive will certainly perform better when parallelized on the multicore GPU, providing a more realistic picture of the GPU's true maximum potential. Second of all does the AES algorithm barely, if at all, utilize the mul pipeline, which is mostly put aside by letting it perform `nop` operations. This in fact alone halves the GPU's achievable parallelization performance, as one half of the GPU's ALU hardware is basically idling. Therefore can be reasoned that an algorithm utilizing the mul pipelines operations more effective, will also reflect the GPU's true maximum potential better.

However to conclude - the last paragraph and essentially the whole thesis - can be said that all of those aspects only add further to the assessment that the GPU really does show excellent potential for speed-up through algorithm paralleliztion.

Future Work

This thesis is only a first step for researching the potential of the Raspberry Pi GPU, with possible ideas ranging from improving the AES implementation created in the course of this thesis to making the enhanced assembler universally usable. Most specific to this thesis and directly tying on to the undertaken research is the implementation of the galois multiplication as lookup tables. Though general advancements in the understanding and usability of the assembler and GPU hardware will benefit this research as well.

An important improvement to the usability of the assembler is the solution of the grave VPM problem resulting in random reads and writes. This would enable the access to all 48KB of the VPM memory. Furthermore would the performance of the GPU be seriously increased if access to the fourth slice containing QPU 12 to 15 could be established. Although those QPUs are listed as present in the hardware architecture reference guide, was it by now not possible to access them with the official kernel driver, suggesting a deactivation of this slice by hardware. A further documentation and exploration of the kernel driver would also be necessary in order to facilitate access to the VideoCore IV GPU for System-on-Chips other than the BCM2835 featured in the Raspberry Pi 1, with the natural expansion seeming to be the SoCs of the Raspberry Pi 2 and Raspberry Pi 3. This would in turn make the assembler, AES implementation and undertaken reasearch more widely usable and applicable.

Additional interesting research is also embodied by further exploring the real usability of the AES implementation created in this thesis and testing the parallelization potential for further block cipher modes of operation. Especially the CTR mode, which is parallelizable in encryption and decryption, seems complex in realization on the CPU but seems well fitted for a parallelization attempt on the GPU. A closing impulse for possible future work regarding the real usability, is the realization of the created AES implementation as a kernel module - including the functionality of a preceding setup.

A. Appendix

```c
/* XORs the state with the current roundKey
 */
void add_round_key(uint8_t *state, const uint8_t *expandedKey, int round)
{
  for (int i = 0; i < 16; i++) {
    state[i] ^= expandedKey[(16 * round) + i];
  }
}
```

Listing A.1: Implementation of `add_round_key()` in C

```c
/* Replaces each byte of the state with the corresponding invSbox byte
 */
void inv_sub_bytes(uint8_t *state)
{
  static const uint8_t invSbox[] = {
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
    };

  for (int i = 0; i < 16; i++) {
    state[i] = invSbox[state[i]];
  }
}
```

Listing A.2: Implementation of `inv_sub_bytes()` in C

```c
void mix_columns(uint8_t *state)
{
  static const uint8_t mixMultMatrix[] = {
    0x02, 0x03, 0x01, 0x01,
    0x01, 0x02, 0x03, 0x01,
    0x01, 0x01, 0x02, 0x03,
    0x03, 0x01, 0x01, 0x02};

  // Because state[1|2|3] need state[0] for calculation, state[0] can not
  // be overwritten before all calculations are done. Therefore are the
  // calculations saved in a temporary array.
  uint8_t tmp[4];

  for (int i = 0; i < 4; i++) {
    tmp[0] = galois_mul(state[(4 * i) + 0], mixMultMatrix[0])
           ^ galois_mul(state[(4 * i) + 1], mixMultMatrix[1])
           ^ galois_mul(state[(4 * i) + 2], mixMultMatrix[2])
           ^ galois_mul(state[(4 * i) + 3], mixMultMatrix[3]);
    tmp[1] = galois_mul(state[(4 * i) + 0], mixMultMatrix[4])
           ^ galois_mul(state[(4 * i) + 1], mixMultMatrix[5])
           ^ galois_mul(state[(4 * i) + 2], mixMultMatrix[6])
           ^ galois_mul(state[(4 * i) + 3], mixMultMatrix[7]);
    tmp[2] = galois_mul(state[(4 * i) + 0], mixMultMatrix[8])
           ^ galois_mul(state[(4 * i) + 1], mixMultMatrix[9])
           ^ galois_mul(state[(4 * i) + 2], mixMultMatrix[10])
           ^ galois_mul(state[(4 * i) + 3], mixMultMatrix[11]);
    tmp[3] = galois_mul(state[(4 * i) + 0], mixMultMatrix[12])
           ^ galois_mul(state[(4 * i) + 1], mixMultMatrix[13])
           ^ galois_mul(state[(4 * i) + 2], mixMultMatrix[14])
           ^ galois_mul(state[(4 * i) + 3], mixMultMatrix[15]);

    state[(4 * i) + 0] = tmp[0];
    state[(4 * i) + 1] = tmp[1];
    state[(4 * i) + 2] = tmp[2];
    state[(4 * i) + 3] = tmp[3];
  }
}

uint8_t galois_mul(uint8_t a, uint8_t b)
{
  uint8_t p = 0;     // p as the product of the multiplication

  while (b) {
    if (b & 1) {     // If b odd, then add corresponding a to p
      p = p ^ a;     // In GF(2^8), addition is XOR
    }
    if(a & 0x80) {   // If a >= 128 it will overflow when shifted left, so reduce
      a = (a << 1) ^ 0x11b; // XOR with primitive polynomial x^8 + x^4 + x^3 + x + 1
    } else {
      a <<= 1;       // Multiply a by 2
    }
    b >>= 1;         // Divide b by 2
  }
  return p;
}
```

Listing A.3: Implementation of `mix_columns()` in C

```
key192_expansion(uint32 key[6])
{
  // Create an expandedKey with the size of 208 bytes or 52 uint32
  uint32 expandedKey = new expandedKey[52];

  // First copy initial key to expandedKey as a basis to expand upon
  for (int round = 0; round < 6; round++) {
    expandedKey[round] = key[round];
  }

  for (int round = 6; round < 52; round++) {
    // Perform special expansion on every 6th round
    if (round % 6 == 0) {
      expandedKey[round * 4] = sub_word(rot_word(expandedKey[(round - 1) * 4]))
                               XOR rcon_word((round / 6) - 1)
                               XOR expandedKey[(round - 6) * 4]
    } else {
      expandedKey[round * 4] = expandedKey[(round - 1) * 4]
                               XOR expandedKey[(round - 6) * 4]
    }
  }
}
```

Listing A.4: Key Expansion Algorithm for 192-bit Key in Pseudocode

```
key256_expansion(uint32 key[8])
{
  // Create an expandedKey with the size of 240 bytes or 60 uint32
  uint32 expandedKey = new expandedKey[60];

  // First copy initial key to expandedKey as a basis to expand upon
  for (int round = 0; round < 8; round++) {
    expandedKey[round] = key[round];
  }

  for (int round = 8; round < 60; round++) {
    // Perform special expansion on every 4th and 8th round
    if (round % 4 == 0) {
      if(round % 8 == 0) {
        expandedKey[round * 4] = sub_word(rot_word(expandedKey[(round - 1) * 4]))
                                 XOR rcon_word((round / 8) - 1)
                                 XOR expandedKey[(round - 8) * 4]
      } else {
        expandedKey[round * 4] = sub_word(expandedKey[(round - 1) * 4])
                                 XOR expandedKey[(round - 8) * 4]
      }
    } else {
      expandedKey[round * 4] = expandedKey[(round - 1) * 4]
                               XOR expandedKey[(round - 8) * 4]
    }
  }
}
```

Listing A.5: Key Expansion Algorithm for 256-bit Key in Pseudocode

```
## This implementation of galois_mul(uint8_t a, uint8_t) takes input in r0 (a)
## and ra20 (b) and overwrites accumulators r1 and r2. The output is stored in
## accumulator r1. The Implementation takes b in a regular register so that
## register r3 will not be overwritten and several galois_mul can easily
## combined in r3 and r1.
galoisMul:

ldi ra33, 0x00000000                      # r1 = p = 0

galWhile:
    # if (b & 1)
    and ra34, ra20, 0x00000001;     nop
    brr.zf ra39, rb39, galFor1End       # branch if (b & 1) == 0 to 'galFor1End'
    nop ra39, ra39, rb39;           nop
    nop ra39, ra39, rb39;           nop
    nop ra39, ra39, rb39;           nop

        xor ra33, r1, r0;           nop    # p = p ^ a

    galFor1End:

    # if (a & 0x80)
    and ra34, r0, rb4;              nop
    brr.zf ra39, rb39, galFor2Else  # branch if (a & 0x80) == 0 to 'galFor2Else'
    nop ra39, ra39, rb39;           nop
    nop ra39, ra39, rb39;           nop
    nop ra39, ra39, rb39;           nop

        shl ra32, r0, 1;            nop    # a = a << 1
        ldi ra34, 0x0000011b
        xor ra32, r0, r2;           nop    # a = a ^ 0x11b

        brr ra39, rb39, galFor2End         # branch to 'galFor2End'
        nop ra39, ra39, rb39;       nop
        nop ra39, ra39, rb39;       nop
        nop ra39, ra39, rb39;       nop

    galFor2Else:

        shl ra32, r0, 1;            nop    # a = a << 1

    galFor2End:

    shr ra20, ra20, 1;              nop    # b = b >> 1

brr.ze ra39, rb39, galWhile                # branch if b /= 0 to 'galWhile'
nop ra39, ra39, rb39;               nop
nop ra39, ra39, rb39;               nop
nop ra39, ra39, rb39;               nop
```

Listing A.6: Implementation of Generic Galois Multiplication in QASM

```
## p = gal_mul(a, 0x0e), r2 = p, r0 = a
shl ra32, r0, 1;            nop          # a = a << 1
and ra33, r0, 0x100;        nop          # if (a & 0x100) then a = a xor 0x11b
shr ra33, r1, 8;            nop
nop ra39, r1, 0x11b;        mul24 rb33, r1, 0x11b
xor ra32, r0, r1;           nop

xor ra34, r0, 0x00000000;   nop          # p = p ^ a

shl ra32, r0, 1;            nop          # a = a << 1
and ra33, r0, 0x100;        nop          # if (a & 0x100) then a = a xor 0x11b
shr ra33, r1, 8;            nop
nop ra39, r1, 0x11b;        mul24 rb33, r1, 0x11b
xor ra32, r0, r1;           nop

xor ra34, r0, r2;           nop          # p = p ^ a

shl ra32, r0, 1;            nop          # a = a << 1
and ra33, r0, 0x100;        nop          # if (a & 0x100) then a = a xor 0x11b
shr ra33, r1, 8;            nop
nop ra39, r1, 0x11b;        mul24 rb33, r1, 0x11b
xor ra32, r0, r1;           nop

xor ra34, r0, r2;           nop          # p = p ^ a
## Result of gal_mul(r0, 0x03) now in r2
```

Listing A.7: Implementation of p = gal_mul(a, 0x0e) in QASM

List of Figures

List of Tables

Listings

Bibliography

[ARM17a] ARM: *ARM1176JZF-S Technical Reference Manual.* `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/Cegdejjh.html`. Version: 2017

[ARM17b] ARM: *ARM1176JZF-S Technical Reference Manual.* `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/Cdfbbchb.html`. Version: 2017

[Axa08] AXANTUM: *128-Bit Versus 256-Bit AES Encryption.* `http://www.axantum.com/AxCrypt/etc/seagate128vs256.pdf`. Version: 2008

[Ben12] BENVENUTO: *Galois Field in Cryptography.* `https://sites.math.washington.edu/~morrow/336_12/papers/juan.pdf`. Version: 2012

[Bro12] BROADCOM: *BCM2835.* `https://web.archive.org/web/20120513032855/http://www.broadcom.com/products/BCM2835`. Version: 2012

[Bro13] BROADCOM: *VideCore IV Architecture Reference Guide.* `https://docs.broadcom.com/docs/12358545`. Version: 2013

[Bro14] BRODKIN: *Raspberry Pi marks 2nd birthday with plan for open source graphics driver.* `https://arstechnica.com/information-technology/2014/02/raspberry-pi-marks-2nd-birthday-with-plan-for-open-source-graphics-driver/`. Version: 2014

[Deb17] DEBIAN: *Libssl-dev 1.01t.* `https://packages.debian.org/jessie/libssl-dev`. Version: 2017

[ELi17] ELINUX: *RpiFront.jpg.* `http://elinux.org/images/9/96/RpiFront.jpg`. Version: 2017

[Far16] FARNELL: *RASPBRRY-MODA+-512M - Evaluation Board.* `http://uk.farnell.com/raspberry-pi/raspbrry-moda-512m/sbc-raspberry-pi-model-a-512mb/dp/2536236`. Version: 2016

[Gua13] GUARDIAN, The: *Edward Snowden: the whistleblower behind the NSA surveillance revelations.* `https://www.theguardian.com/world/2013/jun/09/edward-snowden-nsa-whistleblower-surveillance`. Version: 2013

[Iwa10] IWAI, Nisikawa Kurokawa: *AES encryption implementation on CUDA GPU and its analysis.* `https://www.researchgate.net/publication/224213275_AES_encryption_implementation_on_CUDA_GPU_and_its_analysis`. Version: 2010

[Lip00] LIPMAA, Wagner Rogaway: *Comments to NIST concerning AES modes of operation: CTR-mode encryption.* 2000

[Lor14a] Lorimer: *Hacking the GPU for Fun and Profit.* `https://rpiplayground.wordpress.com/`. Version: 2014

[Lor14b] Lorimer: *SHA-256 Implementation on QPUS.* `https://www.raspberrypi.org/forums/viewtopic.php?f=33&t=77231`. Version: 2014

[Ltd16] Ltd, Raspberry: *Ten millionth raspberry Pi and a new kit.* `https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/`. Version: 2016

[Mag17] Magazine, MagPi: *Issue 53.* `https://www.raspberrypi.org/magpi/issues/53/`. Version: 2017

[Mol15] Molnar, Peter: *Overclocking and Stability Testing the Raspberry Pi 2.* `https://retroresolution.com/2015/11/21/overclocking-and-stability-testing-the-raspberry-pi-2-part-1/`. Version: 2015

[NIS97] NIST: *Preselection of Round 1.* `http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt`. Version: 1997

[NIS01] NIS: *FIPS 197.* `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`. Version: 2001

[Ou06] Ou, George: *Is encryption really crackable.* `http://www.webcitation.org/5rocpRxhN`. Version: 2006

[Pre98] Preneel, Bosselaers Rijmen: *Principles and Performance of Cryptographic Algorithms.* `http://www.drdobbs.com/algorithm-alley/184410756`. Version: 1998

[Reu13] Reuters: *Merkel frosty on the U.S. over 'unacceptable' spying allegations.* `https://www.reuters.com/article/us-eu-summit-idUSBRE99N0BJ20131024`. Version: 2013

[Rij02] Rijmen, Daemen: *Rijndael Encryption Algorithm.* `http://www.efgh.com/software/rijndael.htm`. Version: 2002

[Sah09] Saha, RoyChowdhury Mukhopadhyay: *A Diagonal Fault Attack on the Advanced Encryption Standard.* `http://eprint.iacr.org/2009/581.pdf`. Version: 2009

[San16] Sandvine: *70 Percent of global internet traffic will be encrypted in 2016.* `https://www.sandvine.com/pr/2016/2/11/sandvine-70-of-global-internet-traffic-will-be-encrypted-in-2016.html`. Version: 2016

[Sch12] Schneier, Bruce: *Can the NSA breach AES.* `https://www.schneier.com/blog/archives/2012/03/can_the_nsa_bre.html`. Version: 2012

[War14] Warden: *How to optimize Raspberry Pi Code using its GPU.* `https://petewarden.com/2014/08/07/how-to-optimize-raspberry-pi-code-using-its-gpu/`. Version: 2014

[Wik17a] WIKIPEDIA: *CBC Mode Illustration*. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CBC_encryption.svg`. Version: 2017

[Wik17b] WIKIPEDIA: *CTR Mode Illustration*. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:CTR_encryption.svg`. Version: 2017

[Wik17c] WIKIPEDIA: *ECB Mode Illustration*. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:ECB_encryption.svg`. Version: 2017

[Wik17d] WIKIPEDIA: *Finite Field Arithmetic*. `https://en.wikipedia.org/wiki/Finite_field_arithmetic`. Version: 2017

[Wik17e] WIKIPEDIA: *Illustration Data Pattern Flaw*. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Electronic_Codebook_.28ECB.29`. Version: 2017

[Wik17f] WIKIPEDIA: *Raspberry Pi Driver API*. `https://en.wikipedia.org/wiki/Raspberry_Pi#Driver_APIs`. Version: 2017