# Contents

1 Introduction ............................................. 1
   1.1 Overview ........................................ 1
   1.2 Clock Generation and Management ................. 1

2 Sequencer ................................................ 3
   2.1 Character-based similarity ....................... 3
   2.2 Length-based similarity ......................... 5
   2.3 Memory access ..................................... 6
   2.4 VHDL-Implementation ............................... 6
      2.4.1 Memory Interface ............................. 8
      2.4.2 Hardware Controller ......................... 8
      2.4.3 AND Operation ................................ 10
      2.4.4 Window Function .............................. 10
      2.4.5 Score Computation ............................ 11

3 Sorter ..................................................... 13
   3.1 Bitonic Sort ...................................... 13
   3.2 Bubble Sort ....................................... 14
   3.3 Used combination .................................. 14

4 Implementation ........................................... 17
   4.1 Structure ......................................... 17
   4.2 Parameters/Generics ................................ 17
   4.3 Synthesis Results .................................. 18

5 Driver and Interface .................................... 23
   5.1 Memory interface .................................. 23
   5.2 Registers interface ................................ 24
   5.3 Results Interface .................................. 25

6 Experimental GUI ....................................... 27
   6.1 Search tab ........................................ 27
   6.2 Data tab .......................................... 27
   6.3 Config tab ........................................ 27
   6.4 Sample Search Query .............................. 27
1 Introduction

This is the documentation for the hardware realization of the AMASS algorithm for approximative text searching. It describes the work done at the University of Ulm. The main task was to implement the existing software golden model for searching in VHDL and to provide a defined interface from a host PC to the FPGA board.

1.1 Overview

A first overview is presented in the block diagram 1.1. It shows clearly which parts of the processing are done in hardware and which are completed on the host in software: The whole searching and sorting takes place on the FPGA, while the process control and management of different searches is handled in software. Tasks like score normalization are easier and more efficient on PC hardware as well.

1.2 Clock Generation and Management

This project uses multiple clock networks. This is necessary, since the system on the one hand has to use the PCI bus with its 33/66 MHz and on the other hand the ZBT RAM memory can operate as fast as 200 MHz. The used FPGA offers the possibility to generate different clocks and distribute the signals to the desired modules. The original clock with 200 MHz frequency is generated on the board outside the FPGA by a quartz oscillator. It is feed into the FPGA as differential signal and only then processed into a conventional clock signal. This signal is used by various PLLs to generate clocks with frequencies as needed.

The clock for the external memory is generated inside the FPGA as main clock (mclk) and then routed to output pins via flip-flops. Every RAM bank receives its own buffered clock signal. To deskew jitter and remove delays, every signal is feed back into the FPGA.

The clock for the local bus (lclk) can be varied between 400 kHz and 66 MHz. The interface registers can be accessed from both clock domains. Since the information stored in them are mostly status bits, that do not change rapidly during a search process, no additional logic is needed to provide the clock transition. The results register, that contains the output of a completed search process, is built as a FIFO, so that no data can be lost as well.
CHAPTER 1. INTRODUCTION

Figure 1.1: Overview of the system
2 Sequencer

The sequencer is the main processing part of the system. It calculates the similarity between the query and the entries in the database. In order to increase the effectivity and decrease the time to process a query, it is implemented multiple times. The modules work in parallel on different database words. These words are saved in the RAM that is located on the FPGA-board.

The interface for the RAM, in this case ZBT-RAM with a bus width of 256 bit, is provided by a module in the FPGA. Since ZBT-RAM is a static RAM that needs no refresh cycles, the interface is rather simple. The main task of the memory interface is therefore the addressing of the memory and the synchronization with the memory interface register. This register is used to access the memory from the host PC to allow changes in the database.

The similarity that is used for the project AMASS is based on two different approaches. The first one is based entirely on the used symbols or characters, while the second relies only on the word lengths. Both are combined by multiplication to create an overall similarity in the end.

2.1 Character-based similarity

The character-based similarity is calculated in several steps, depending on the query length. First the relevant part of the signature of a database entry is read via the memory interface from the ZBT-RAM, buffered with a FIFO to compensate different processing delays. Then the needed monograms and bigrams are combined with a logic AND function (see figure 2.1). This combination always uses one bigram and the two corresponding monograms. An example for the procedure is shown in figure 2.2.

The reason for this step is to reduce the number of collisions that occur within the bigrams: Since more than $26^2 = 676$ bigrams have to be mapped to 40 bit, naturally collisions occur and different bigrams are represented with the same signature. But as bigrams consist of monograms, it is clear that a bigram exists only then in query and database entry, if the corresponding monograms are present as well.

The so generated bit sequence is then sent to the actual sequencer. It is made up by a rectangular window function, a look-up table and a calculation unit. The processing is as follows: The bits that are masked by the window are used as address for the look-up table. Then the stored values are summed up over the query length.
CHAPTER 2. SEQUENCER

Figure 2.1: Combination of mono- and bigrams

Figure 2.2: Logic AND function to minimize collisions within the bigrams (Query: BUNDESLIGA, Database entry: BUNEDSLIGA)
Since different string manipulations lead to different bit sequences, it is possible to distinguish between them and associate them with different similarity values. These values are precalculated to match the desired similarity measure and then stored in the look-up table.

In order to cope with partial matches, when only fragments of the query and the database entry are similar, an additional bit is used that shows whether the calculation is inside such a partial match or not. If more than one part matches, the fragments are combined, but with a penalty for the mismatch in between. This additional bit is stored in the look-up table as well.

This similarity $S_1$, that depends only on the involved characters, is finally normalized to the query length. For this normalized value, $0 \leq S_1 \leq 1$ is always true. This normalization is the same for all database entries and therefore does not change the order of results. Because of that it is possible to normalize the results as a last step in software, just before presenting the results to the user or further processing.

### 2.2 Length-based similarity

The length-based similarity adds a penalty to database entries, that differ from the query in terms of length. Since all words that are considered in one path have the same length, it is sufficient to calculate this second similarity measure only once for all words and then distribute the value to all further modules.

The similarity $S_2$ is calculated as

$$S_2 = 1 - \alpha \cdot \frac{\max (l_Q, l_D) - \min (l_Q, l_D)}{\max (l_Q, l_D) + \min (l_Q, l_D)} \tag{2.1}$$

with $l_Q$ the length of the query word and $l_D$ the length of the database entries.

For the similarity $S_2 \leq 1$ is always true, and $S_2 = 1$ is only reached for $l_Q = l_D$. In all other cases the similarity is decreased.

The implementation of formula 2.1 in VHDL is not trivial, but a subsequent calculation on the host PC is not possible. This would result in a wrong sorting order in the VHDL part and therefore to incorrect search results.

Since both $l_Q$ as well as $l_D$ have ranges from 4 to 64 (7 bit), the division either requires lots of logic cells and space or up to 7 clock cycles. As the character-based similarity is potentially finished in 5 clock cycles, the sequential solution takes too long. The solution is therefore to store precalculated values for the division, so that this result has only to be multiplied with $\alpha$ and then subtracted from 1 to form the final result. These two operations can be completed with affordable amounts of logic in two clock cycles.

Most of the logic is therefore occupied by the rather large look-up-table (LUT), where the results of the division are stored. Its inputs are $\max (l_Q, l_D)$ and $\min (l_Q, l_D)$, the
output is the result of the division, with an accuracy of 8 bit. Since
\[ \max (l_Q, l_D) \geq \min (l_Q, l_D) \]  
(2.2)
is always true, the LUT does not need to cover the whole 60x60 matrix, but only half of it. This means that the amount of memory that is needed for the LUT is
\[ \frac{60 \cdot 60}{2} \cdot 8 \text{ bit} = 1800 \text{ bit} \]  
(2.3)
The big advantage of the design is the very high speed that can be achieved. This is true for the clock speed as well as for data throughput. An approximation for this similarity that is easier to implement is described in [Sch07].

2.3 Memory access

The 32 MByte ZBT-RAM has a 256 bit wide interface to the FPGA. In order to have read/write from the host PC, a module in the FPGA can be used to transfer data from the memory to the local bus and from then via the PLX bridge to the 32 bit wide PCI bus in the PC. In a similar way the host PC has access to a number of registers in the FPGA that are used for bidirectional communication between FPGA and host CPU.

Another task of the interface module for the ZBT-RAM is the synchronization of the two involved clocks. Since the PCI bus operates at 33 MHz and the memory and most of the FPGA at up to 200 MHz, all data transfer between these two regions has to be buffered. In figure 1.1 this is the Memory Interface module.

The data organization in the memory differs from the obvious layout. To achieve a high data throughput, the signatures are not stored in serial, but in parallel as in figure 2.3.

This addressing scheme allows to read one monogram and one bigram slot for 128 different database entries in one clock cycle. This grade of parallelization is maintained throughout the whole processing system, so that 128 similarities are calculated simultaneous. The processing can start after two cycles, when four entries of the signature have been read from the memory. The 128 database entries that are dealt with at the same time are called one path. This entries of one path remain together throughout the processing and are later labeled with the same hardware key during sorting.

2.4 VHDL-Implementation

The implementation in VHDL takes care of all this considerations. Apart from the implementation of the different algorithms, the communication between the involved modules is very important. The case where data is lost because following modules are not ready yet is to be avoided at all costs.
2.4. VHDL-IMPLEMENTATION

Figure 2.3: Memory organization
2.4.1 Memory Interface

The memory interface is responsible for data transfers from and to the ZBT RAM memory. It receives the data address that should be accessed, the direction of the data flow and the memory clock and delivers the data that is stored at the given address or saves the given data at the desired address. The whole address decoding for the different blocks is done in this module.

The partitioning into 8 ICs with 4 MByte capacity each and a 32 bit wide data bus explains the 256 bit wide communication to the 32 MByte sized RAM. The memory layout according to figure 2.3 ensures the optimal usage of this data bus.

Since ZBT RAM is used, the delay between giving the command and the actual response from the memory ICs is as fast as only four clock cycles. This is a pure latency, so that during every clock cycle a different address can be accessed for read or write operations. The code for this module is based on application notes from Alpha Data, the manufacturer of the FPGA board.

To ensure a correct data flow even in the case when the processing is temporarily stopped, the data is buffered in a FIFO. It has a maximal length of 4 entries, corresponding to the latency of the memory and memory controller. This length is sufficient to enable an immediate resume of the processing, without having to wait for the memory to read new values.

2.4.2 Hardware Controller

The hardware controller is responsible for the process control and communication between all involved modules in the search query. It is designed as a state machine and is built to promote the various control signals between the modules with the appropriate delay.

The modules that are controlled in this way are the AND operation, the sequencer with window function, look-up table and calculation, the sorter, the memory interface and the registers that communicate with the host PC.

The states of the machine are

- **IDLE**: No query is running and the system waits for inputs.
- **COMPUTE**: The system is running and calculating similarities.
- **WAIT_SEQ**: The sorter has to process data and is not able to accept new results. Therefore the sequencer is stopped and waits for the sorter. The FIFO for the memory access is filled.
- **WAIT_SYNC**: The system has finished all computations and sorting and waits until the host PC has read the results from the results register.

The state machine for a protocol between the Controller, the various processing modules and the host PC is depicted in figure 2.4. The conditions that are responsible
for the different state transitions are either from within the FPGA system or from the host PC. The signals that are set by the state machine control the whole FPGA system.

Figure 2.4: State machine of the HW Controller

A second protocol is responsible for the process control between the FPGA (i.e. HW) and the host PC (i.e. SW). The protocol uses two registers in the FPGA, the status register is written by the HW and read by the SW, the control register vice versa. Figure 2.5 shows the sequence.
2.4.3 AND Operation

The AND operation combines the monograms and bigrams that are stored in the signature in order to reduce the probability of collisions within the bigrams. It is also the module that is responsible for the correct application of the additional administrative information stored in the signature. Those two bits allow the masking or demasking of database entries, so that only specific entries can be searched.

The operation that is performed is:

\[
\text{Out Data} \leq \text{not sig}_\text{Adm.Col} \text{ and sig}_\text{Mono1} \text{ and sig}_\text{Mono2} \text{ and sig}_\text{Bi12};
\]

The delay between input and output in this module is one clock cycle.

2.4.4 Window Function

The sliding window function masks the bit sequence except for the values that are processed at the moment. It works like a FIFO with length 7. Additionally it provides logic to allow quick transitions between successive database entries. Since the window is initialized with 001 and at the end of the sequence finalized with 100. These two special sequences are generated within the module and inserted at the appropriate positions.

By buffering the values from the AND operation between different paths, the throughput is maximized and no clock cycles are lost.
2.4. VHDL-IMPLEMENTATION

2.4.5 Score Computation

The actual computation of the similarities is done in this module. It uses dual-port RAM cells on the FPGA as a look-up table. The two ports allow 2 calculation modules the parallel access to the saved data. The bit sequence that is currently in the window function is used to address this table. It returns a similarity score for the current bit sequence. The computation then adds up these retrieved values over the query length. Partial matches are handled by additionally stored information on the position in a matching cluster.

When the query length is reached, the score has to be finalized, so that one additional clock cycle is required. The output of the module is the character-based similarity, coded as 11 bit integer according to section 2.1. This value has to be multiplied with the length based score and is then sent to the sorter. Every score computation module is responsible for the parallel computation of two similarities.
After the amount of similarity between the query and the current entries in the database are calculated in the sequencer, the entries have to be sorted according to the similarity. Finally a list of the overall $l$ most similar entries has to be compiled. For this purpose, a sorter module is needed.

Since the task is twofold, it is useful to divide the work into different modules. Most of the sorting is done by a bitonic sort algorithm, while the merging into the final list is accomplished by a bubble sort.

### 3.1 Bitonic Sort

The bitonic sort is an algorithm that allows a quite simple hardware implementation, since it can reuse large logic parts in different stages of the sorting. It was first described by Batcher in [Bat68]. It is based on the idea of sorting bitonic sequences. A sequence is called bitonic, if it is a juxtaposition of two monotonic sequences, one ascending and the other descending. The sequence can be splitted and the parts interchanged, it remains bitonic. This bitonic sequence with $2n$ elements is ordered by a network of two sorters into one monotonic sequence. One sorter recieves the $n$ lower elements, the other the remaining $n$ bigger elements.

The sorter consists of simple comparison elements as shown in figure 3.1. This is already the complete sorter for 2 elements.

This structure allows the iterative construction of a sorter. In the first step ordered lists of length 2 are produced, the next step merges two of those lists to form a sorted list.

Figure 3.1: Basic element for sorting network
list of 4 elements, and so on. To sort $2^p$ arbitrary elements, $\frac{1}{2}p(p + 1)$ levels or stages each with $2^{p-1}$ comparison elements for a total of $(p^2 + p)2^{p-2}$ elements are needed. This number is slightly higher than for an odd-even sorter. But the big advantage for a bitonic sorter is that the structure is very regular and therefore a smaller amount of hardware is sufficient if it is reused.

Stone proved in [Sto71] that a perfect shuffle can be used together with the bitonic sort. So in order to construct a complete sorter, one stage with variable connections is sufficient. Depending on the step, the basic elements are connected and the output is feed back into the structure. Additional information on how to implement a bitonic sort on an FPGA processor can be found in [Lay07].

### 3.2 Bubble Sort

The second sorting algorithm used in this project is the bubble sort. It compares neighboring elements and switches position if necessary. This sorting needs $l$ cycles in the worst case to sort $l$ elements. The required logic is relatively simple.

### 3.3 Used combination

In the application 128 elements have to be sorted according to a value represented with 11 bit. Since the original position of the elements has to be saved, additional 7 bit have to be added during the sorting. In order to reduce the latency of the system, without using a fully parallel system which needs huge amounts of logic, the sorting is partitioned into 3 stages and additional merging.

In the first stage, the 11 bit input are concatenated with a 5 bit counter that repeats 4 times the numbers 0 to 31. This 5 bit cyclic ID allows the backtracking of the various entries and later the association of the sorted elements with entries in the database. The unsorted elements are processed by the bitonic sort algorithm and the output generated are 16 sorted lists with 8 elements each. This is achieved in 7 clock cycles.

The next stage merges these lists in 3 clock cycles and provides 8 sorted lists with 16 elements each.

The third stage finally generates 4 sorted lists with 32 elements each, every list containing elements numbered 0 to 31. The latency is 3 cycles again. Every element receives an additional 2 bit to distinguish between the lists. That way every element can be tracked back to the original position in the list and with that can be associated with a database entry.

All three stages use the bitonic sort and merge idea.

Since a text search is usually done with more than 128 database entries, an overall list has to be generated that represents the most similar words overall. For this purpose a fourth stage is used. It takes the best words from the 4 sorted lists and inserts them into a bubble sort procedure with $l$ elements. This size is the desired size of
3.3. USED COMBINATION

the final result list. The new elements are inserted at the lowest position if they are greater than the current smallest element. In this way the smallest element is always at the lowest position and new insertions are correct. The whole list is sorted from the bottom up, by having all elements bubble upwards until they reach their correct position.

When no more elements have to be inserted, the next similarities can be calculated by the sequencer. When all database entries have been considered, the bubble sort needs an extra \( l \) cycles to ensure the correct position of all elements. In this worst case, the greatest element is inserted in the last processing path. The time it takes for this element to reach the top is exactly the size of the list. After this final sorting, the output can be presented to the user.

Figure 3.2: Overview of sorting process
4 Implementation

This section deals with various aspects of the actual implementation.

4.1 Structure

The structure of the whole project is already described in figure 1.1. In particular, the sequencer, that is responsible for the computation of the character-based similarity, is instantiated multiple times to allow parallel processing. One block consists of one dual-port block RAM, two window functions and accordingly two computational modules. To process 128 database entries in parallel, 64 of these sequencer blocks are required.

In addition to that, one module for the length-based similarity is needed. This single module is sufficient, since the length-based similarity is the same for all database entries within one path. The result of this calculation is distributed to the parallel sequencer, so that the two independent scores can be multiplied to form the final similarity.

This multiplication is performed in the DSP slices of the FPGA. Since only 96 of them are present, but 128 multiplications are required, 64 of the slices are multiplexed and used sequentially.

4.2 Parameters/Generics

The design can be adjusted to various situations by the use of generic parameters. These parameters are as follows, together with their default value in brackets.

- **LB_ADDR_SIZE** The width of the local bus for addresses in bits (23)
- **LB_DATA_SIZE** The width of the local bus for data in bits (32)
- **NUM_BANK** The number of clocks used in the design (8)
- **NUM_CLOCK** The number of clocks used for the SSRAM banks (8) This corresponds to one clock per bank
- **SIZE_SSRAM_ADDR** Size of the SSRAM address bus in bits (21) Only 20 bits are used to address the 4MByte per bank.
- **DATA_WIDTH** Size of the SSRAM data bus in bits (32)
CHAPTER 4. IMPLEMENTATION

**CTL_WIDTH** The number of bits composing the SSRAM control bus

**BANK_GROUP** Logical division/regrouping of the SSRAM memory (1)

**HW_BUS_SIZE** Width of the hardware bus at the output of the memory (256)

**QUERY_SIZE** Maximum character count of the query, must be a power of 2 (64)

**SAMS_NBR** Maximum number of SAMs, must be a power of 2 (64)

**SIG_SIZE** Number of bits used for the signature of one database entry (128)

**COST_SIZE** Number of bits used to code the costs for manipulating the character string in the sequencer (6)

**CLUSTER_SIZE** Number of bits used to code the position relative to the current cluster (2)

**WINDOW_SIZE** Size of the sliding window used for the char-based similarity (7)

**SCORE_SIZE** Number of bits used to code the similarity at the output of the sequencer

**SIZE_HW_KEY_PART** Number of bits used to describe the HW key, composed of SAM number and path number (14)

**SIZE_FIFO_SORTER** Size of the final results list, corresponds to the number of possible entries in the bubble sort (200)

**SIZE_ALPHA** Number of bits used to represent the parameter alpha

**SIZE_SCORE1** Bit width of the length-based similarity

The system is tested with these default values. Probably not all possible combinations of parameters will work similarly well.

### 4.3 Synthesis Results

The whole system can be mapped without problems onto the used FPGA, a Xilinx Virtex-4 LX 160. The slice usage is around 20,000 slices, out of 135,000 available. This usage is divided into the various sections of the design as follows.

**SSRAM Controller** The controller for the ZBT RAM uses about 2,000 slices together with 32 RAM blocks and can be clocked at a maximum of 240 MHz.

**Control Registers** These block RAMs together with the accompanying logic consume slightly more than 500 slices plus 12 RAM blocks, the maximum frequency is almost 250 MHz.
4.3. SYNTHESIS RESULTS

**HW Controller** The Controller uses only about 120 slices, since the whole controller is built around delays between the different signals and their logic combinations. Accordingly it can be clocked as high as 260 MHz.

**Sequencer** One instance of the sequencer is comprised of one RAM block, two sliding windows and 2 calculation units. They need about 90 slices in addition to the RAM and can be clocked at a maximum of 230 MHz. 64 of these instances are present in the system.

**Full Sequencer** The full sequencer is made up from the 64 sequencers and additional 64 DSP slices for the multiplication of character- and length-based scores.

**Length Score** The length-based similarity is calculated once for all sequencers in this module. It uses about 60 slices and one DSP slice for the multiplication. The highest possible frequency is only 185 MHz.

**Sorter (1. Stage)** The first stage of the sorter takes the completely unsorted input and produces 16 sorted lists with size 8. For this process, about 1.200 slices are used, the latency is 7 clock cycles at a maximum frequency of 265 MHz.

**Merger (2. Stage)** The second stage takes this output and keeps on sorting, so that the output are 8 sorted lists with size 16. This stage occupies 1.200 slices and requires 3 clock cycles at a maximum frequency of 265 MHz as well.

**Merger (3. Stage)** Finally the third stage takes this input and produces 4 merged lists with 32 entries each. Another 1.200 slices with a maximum frequency of 265 MHz do the merging in 3 clock cycles. These three stages all use the bitonic sort algorithm.

**Bubble Sort** The final module that collects the valid results and stores the top database entries is the bubble sorter. It occupies 4.800 slices that are clocked with at most 200 MHz. The latency during the sorting is not deterministic, but a wait time at the end of the input in the size of the sorter in clock cycles has to be ensured to guarantee the correct order.

A detailed synthesis report for the whole system is depicted in table 4.1. These results are taken directly from the synthesis tool. The high value for the maximum LCLK frequency can not be used completely, since this clock is used for the transfer to the PCI bridge, which has a maximum clock of 66 MHz specified.
### Device Utilization Summary

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Util.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of slice Flip Flops</td>
<td>20,256</td>
<td>135,168</td>
<td>14%</td>
</tr>
<tr>
<td>Number of 4 input LUTs</td>
<td>33,068</td>
<td>135,168</td>
<td>24%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logic Distribution</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of occupied slices</td>
<td>20,477</td>
<td>67,584</td>
<td>30%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total Number of 4 input LUTs</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number used as logic</td>
<td>33,068</td>
<td>135,168</td>
<td>26%</td>
</tr>
<tr>
<td>Number used as route-thru</td>
<td>1,855</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used for Dual Port RAMs</td>
<td>512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as Shift registers</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>567</td>
<td>768</td>
<td>73%</td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs</td>
<td>5</td>
<td>32</td>
<td>15%</td>
</tr>
<tr>
<td>Number used as BUFGs</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as BUFGCTRLs</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of FIFO16/RAMB16s</td>
<td>76</td>
<td>288</td>
<td>26%</td>
</tr>
<tr>
<td>Number used as FIFO16s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as RAMB16s</td>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of DSP48s</td>
<td>65</td>
<td>96</td>
<td>67%</td>
</tr>
<tr>
<td>Number of DCM_ADVs</td>
<td>3</td>
<td>12</td>
<td>25%</td>
</tr>
</tbody>
</table>

| Total equivalent gate count for design | 5,423,916 |
| Additional JTAG gate count for IOBs  | 27,216    |

| Maximum frequency for MCLK         | 110.767 MHz | after PAR |
| Maximum frequency for LCLK         | 191.490 MHz | after Synth |

Table 4.1: Synthesis results for the whole system
4.3. SYNTHESIS RESULTS

The interface for the devised FPGA system consists of several C functions. They are used to control the board via the host PC and to retrieve the produced data for further processing. The communication uses the dual-port RAM blocks on the Virtex-4 to allow an easy synchronisation between hardware and software.
5 Driver and Interface

The whole system needs only to access three groups of interface registers, the accesses consisting of 24 C functions. They allow the complete control over the processes running on the FPGA and enable the tuning of various parameters.

5.1 Memory interface

The memory interface, called LB_if_MemRegs is responsible for the dataflow to and from the ZBT RAM. It provides functions to read and write the memory, either in small chunks (one signature) or in a kind of burst mode blocks of 128 signatures.

**read_block_memory** This boolean function reads 128 signatures (i.e. one block) from the RAM at the given address and stores them in the provided table. The parameters for this function are

- **SAM_Nbr** The number of SAM to be addressed (coded as 6 bit integer)
- **Path_Nbr** The path number inside the SAM (coded as 14 bit integer)
- **data_table** A pointer to the memory where the retrieved data is stored

The return value is 0 if the operation was successful, false otherwise.

**write_block_memory** This function is used to fill the ZBT RAM with the precomputed signatures in burst mode. It writes 128 signatures at a time. The parameters are exactly the same as for the **read_block_memory** function.

**read_chunk_memory** This function reads a single column from 128 signatures from the ZBT RAM and stores it. It takes as parameters

- **SAM_Nbr**
- **Path_Nbr** as before and additionally
- **Slot_Nbr** the column inside the signature (coded as 7 bit integer).
- **data_chunk** stores the retrieved data

As before, the function return 0 if successful, false otherwise.

**write_chunk_memory** Again this is the equivalent for the write process, one column for 128 signatures is written to the ZBT RAM at the given address.
### 5.2 Registers interface

There are several registers that control the processes inside the FPGA. These registers can be read and written via C functions.

**Read Control Reg** Reads the bit stored in the control register (see fig 2.5).

**Write Control Reg** Write 2 bit into the control register. If a "‘1’" is stored, the FPGA starts processing a query. A "‘2’" causes the system to pause.

**Read Status Reg** The FPGA stores its status in this 2 bit register, so that the host PC knows what is happening. A "‘0’" stands for IDLE, "‘1’" for WORKING and "‘2’" signals that the FPGA has finished processing and is ready to transmit the results.

**Read Nbr SAM Used Reg** Reads the register containing the number of SAMs used and stores the data.

**Write Nbr SAM Used Reg** Write the same register.

**Read Tacts Counter Reg** Read the number of clock cycles needed to process the query on the FPGA. This number does not take into account the transfer of SAM data as well as the transfer of the final results, since these depend on the PCI bridge transfer. The number of cycles is coded as 22 bit integer and gives an accurate impression of the processing speed.

**Read Nbr Res Reg** This function reads the register containing the number of results to use. It defines the number of entries that are available in the results register at the end. It is coded as a 12 bit integer.

**Read SAMs PA Regs** This function reads the physical address (PA) of the given SAM. The SAM_Nbr is a 6 bit integer.

**Write SAMs PA Regs** This equivalent writes the physical address of the given SAM. During initialization it is necessary to define the PA of all SAMs involved in the query.

**Read SAMs PA Regs** In burst mode this function reads all PAs of SAMs, from 0 to nbr_2_read-1. The retrieved data is stored in the provided table.

**Write SAMs PA Regs** The same is possible for writing. All addresses from SAM 0 to SAM nbr_2_read-1 are written in the respective registers.

**Read SAMs NP Regs** Read the number of paths (NP) for the given SAM. This is a measure of the size of the SAM and defines how many paths of the SAM have to be searched.

**Write SAMs NP Regs** Again this has to be written during initialization.
5.3. RESULTS INTERFACE

**Read_SAMs_NP_Regs** Just like the physical address, the number of paths can be read in burst mode, from SAM 0 to SAM nbr_2_read-1.

**Write_SAMs_NP_Regs** The same is true for writing the registers.

**Read_Query_Regs** Read the monograms and bigrams that make up the query. The positions are coded as 8 bit values. 2*query_length slots are read, the mapping is done in a sequential way: Monogram Bigram Monogram Bigram ... Figure 5.1 shows the details.

**Write_Query_Regs** Write the existing monogram and bigram positions of the query in the appropriate register during initialization. 2*query_length positions are needed.

**Read_Command_Regs** Read the requests stored up to nbr_2_read-1 in the command register. A command consists in the number of the SAM to process for the current query.

**Write_Command_Regs** Write the commands for the HW-Controller during initialization.

### 5.3 Results Interface

This interface is used to retrieve the final results after processing a query.

**read_fifo_results** This function reads the stored results from the results register. One entry consists of:

- The computed score, coded as 11 bit integer
- The HW key, composed of
  - SAM number (6 bit)
  - Path number (14 bit) and
  - Position inside the path (7 bit)

Entries are read from 0 up to nbr_2_read, as defined in the Nbr_Res_Reg.
Figure 5.1: Register representation of the query word

Note: $N$ represents the number of slots (Monograms + Bigrams) in the Query, with $8 \leq N \leq 127$. 
6 Experimental GUI

In order to ease the debugging of the system, an experimental graphical user interface (GUI) was implemented. It allows to set all needed values for a query and gives a multitude of information that can be used for debugging. It was implemented using Qt as a QWidget. The GUI is separated into 3 tabs.

6.1 Search tab

This tab allows to enter a query, start the searching and then displays the results. Figure 6.1 shows a screenshot.

6.2 Data tab

This tab can be used to send the SAMs or other config data to the board. It also contains a debug area, where various registers can be read and displayed in a list view. A screenshot is shown in figure 6.2.

6.3 Config tab

The third tab shows several information about the FPGA board, such as revision and used FPGA. Additionally the clocks for the board can be set and the board programmed with a bitfile (compiled VHDL code). Again figure 6.3 shows a screenshot.

6.4 Sample Search Query

The usage of this GUI is quite intuitive. To explain a sample search query, the whole process is described in the following. These steps do not cover all possibilities provided by the various tabs of the GUI.

1. Ensure that both the application AMASS_GUI as well as the board drivers as provided by AlphaData are installed in the PC, where the FPGA board is installed.

2. Once the application is running, set the MCLK clock in the Clock Management section of the Config tab to a reasonable value. For the current code, a frequency between 70 and 110 MHz works without problems.
Figure 6.1: GUI Search tab
Figure 6.2: GUI Data tab
Figure 6.3: GUI Config tab
3. The next step is to program the FPGA. For this, browse for the `amass.bit` bitfile in the `Load Bitfile` section of the Config tab and select `Download`.

4. After that, the board has to be initialized. For this, the word list (database) and other information has to be provided. They are defined in the file `init.txt` and are loaded when the button `Init Board` is pressed. Always make sure, that the relative path to these files is correct and they are present on the system. This ends the initialization of the board. It can now process queries.

5. A query can be given in the Search tab of the GUI. To transfer it to the board, press the `Download` button.

6. To allow adjustment to different applications, the SAMs that are processed can be selected. This is done in the two boxes just below the `Start Search` button. Both values can range from 0 to 63, while the second always has to be greater than or equal to the first selection.

7. The search is started by pressing the `Start Search` button. After the process has finished, the results together with a simple processing time analysis are presented.

8. To start a search with the same query word, but different parameters, it is possible to adjust these values and press `Start Search` again. This reduces overhead. To use another query word, it has to be downloaded.

9. At all times, the application can be ended by pressing the `Quit` button. Using this button ensures that all temporary data is deleted and the handle for the FPGA board is released.

When using the system in this way, some special conditions can not be handled by the board, so that the user has to ensure to meet these requirements:

- The query has to be at least 4 characters long, otherwise the board will crash.
- The initialization order is not arbitrary, skipping one step might crash the board.
- In case of a crashed or hanging board, it is usually sufficient to close the application and reopen it. If this is not the case, the PC has to be rebooted to restore the PCI bus.
Bibliography


