Which requirements favour which implementation?

Abstract—**This paper will dive in four different implementation of the insertion sort algorithm. The four implementations are FSMD (Finite State Machine with Data path), ASIP (Application-Specific Instruction-set Processor), C (software-based implementation) and Custom IP (hardware-based implementation). The implementation will be described in the first part. The second part will look further and focus on the question which requirements favour which implementation. Here different requirements will be presented and evaluated. In the end we present in which situation developing and deploying of a certain implementation makes most sense.**

I. Introduction

II. InsertionSort

We decided to implement the InsertionSort Algorithm for the project asignment. It is a well-known and simple sorting algorithm. The idea of InsertionSort is similar to 'sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group´ [1]

It is maily used for small or nearly sorted lists. For a nearly sorted list, the time complexity of insertionsort is a $O(n)$ which is the best case. However the average and worst case have both a time complexity of $O(n^2)$. Other algorithms like Mergesort or Quicksort are more prevelant. This is due to their lower average time complexity.

InsertionSort is a in-place sorting algotithm which means that its space complexity is at $O(n)$. It requires as much space as the input requires and only fixed additional inputs for some values as

III. Implemtations

This section will focus on our four different implementations of the In. We performed the implementation in the same order as they are listed in the next sections. The decision was made due to feeling most comfortable with designing and implemented an FSMD. This decision turned out to be very beneficial. In the ASIP implementation we based our work on the ASMD diagram created in the FSMD implementation. In the creation of our custom IP block, we started of by using the same code which we used in our C-program.

All implementations were done by the whole group and strictly after one another. We did not work on the next implementation before having a working version of the previous implementation.

A. FSMD

We started with a finite state machine with data path (FSMD) diagram as well as an algorithm state machine with datapath (ASMD) diagram before we started the implementation. We started with designing the FSMD. There, we choose the parts that are needed such as a RAM, receiving and transmission unit as well as comparators and registers. Based on that we defined our data path. In order to keep the implementation simple and strict to the concept, we avoided sending any data to the control path. We exchange only control signals. In the end we have nine input signals for our control path and twelve output signals that handle the dataflow in the data path.

After coming up with a FSMD we defined an ASMD. Here we devided our data flow into different states. Each state is one clock cycle long and therefore in- and output singals can only have represent value in each state. In total there are ten states. State S0 resets the FSMD to start the sorting anew. S1 receives and saves the integers from putty via a serial port to the RAM. The Enter character will trigger the transition to start states S2 to S7. Here the received numbers are sorted iteratively. Unsorted numbers will be compared to the already sorted number sequence and then inserted at the right place. This will be repeated until there are no unsorted numbers left and every number is at their right place. Again the transition to the next states is triggered by finding an Enter character. State 8 and 9 are responsible to send out the sorted numbers one by one via the UART_tx component.

After designing our diagrams we started the implementation in Vivado. We used the VHDL files of the selectionSort as a base project and adapted the fsm.vhd and insort_top.vhd file. The fsm.vhd file is defining the states and also controling the flow between the states. The insort_top.vhd file defines all individual components (like registers, RAM, uart_rx, uart_tx), their input and output signals as well as the routing of these signals to other components. In each state the signals are set acording to the state box in the ASMD diagram. To implement the decision boxes we included new functionalities for the five comparators within the insort_top.vhd file. Further the address mux and the data mux are also implemented in the insort_top.vhd file.

After the initial implementation, we tested our functionality by using the waveform. Errors, that were found while debugging, were corrected in the code as well as in the diagram.

After we had our first working version, we refactored both the code and the diagrams to ensure that both are not overcomplicated.

B. ASIP

Second implementation was the application-specific instruction set processor, which we implemented using the states described in the ASMD chart. This allowed us to stay consistent
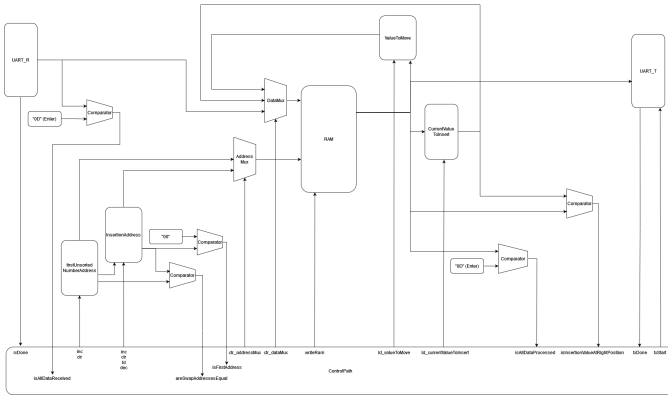
Fig. 1. FSMD diagram for the insertion sort

Fig. 2. ASMD diagram for the insertion sort

with our design choices as well as keep a structred overview of the ASIP implementation. We used the one-cycle cpu design as a template for the ASIP design and modified it by changing some core features. This included adding additional components and control signals.

In this implementation we start by initilizing registers. We then enter a loop until the condition `rx_done` is met. Afterwards data is stored into memory as we look through the data we recieved. We use intermediary conditional branch statements for the main logic. This is achieved by using store, load and clear statements to sort the data as we gradually make our way through the input data. Once the end of the input data is detected `tx_done` is set which signals that the end of data transmission.

Since the design of the ASIP was built upon the logic of the ASMD we needed to modify the opcodes to include the instructions we required for our implementation. The required opcode instructions were more than originally designed, and therefore the instruction address register had to have its size increased.

The new opcodes we required were used for branching, writing to the UART, and looping. For the conditional branching, all we required was `Branch if Equal`, `Branch if Not Equal`, and `Branch if Greater than`. In our implementation, we use these conditional operations to create loops. We also required the use of jumps, as in our ASMD diagram we needed to access earlier parts of our states frequently. We use our `LOP_UART_DONE` to loop until we meet the condition for our signal indicator `rx_done`.

The UART itself is a component in our structure that is used for recieving and transmission of data. The main functionality of our UART can be summerized in the states idle, start, transmit and stop. The signal indicator rx was key when working with the logic running the UART.

C. C coding in Vitis

Next up, we definied the C program. We setup the project - as we have done it in the lecture week - based on the roadmap of Eric Peronnin. This was done to start of a working version which can always go back to when encoutering errors. It
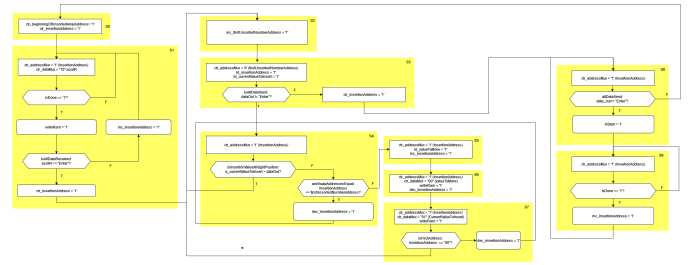
proved to be valuable since the debugger output is sometimes not easily understandble. We starte by removing the logic for the LEDs both in Vitis and afterwards in Vivado by removing the Blocks for the LEDs in the design. We then copied and basic insertion sort algorithm implementation in c from geeksforgeeks[1]. Next we changed the input and output behavior. After some debugging with "print" statements, we managed to get the code working quite fast. Most of the debugging time was spend on issues with pointers. Since no one in the group has comprehensive experience in c, some errors like that were to be expected. We verified that our final version behaved on the Zybo board exactly the same as the FSMD and ASIP implementation did on the Basys board. Once we made sure it behaved as expected we moved on to the next implementation.

D. Create custom IP block

Lastly, we implemented the IP block. We expected this to be very similar to the C porgram. As we have do it in the project before, we wanted to start based on some working version. Therefore, we implemted the FIR Filter Project again.

We encountered the most issues in this implementation. The first issue we encountered was in the udpating of the platform and application components in Vitis. They seem to have a cache which is not cleared automatically when another component is deleted. So when we made changes in our hls component, created a new xsa file and updated the platform, we encountered the same behaviour as before. Only deleting the platform and application component in Vitis, fully seemed to clear any cached information.

Further, issues were all based on the connection between our application component and our hls/IP component. As described we started of with the fir filter implementation. Since the fir filter implementation exchanges only single integer values, we wanted to modify it into sending arrays. An array cannot be set as an input for a function, so we used pointers. The values of the pointers were exchanged between these two components. Any further data was not copied. Next, we tried a different angle by exchanging a struct between these two components. The issues with this idea were in the application. We did not manage to find out how to use custom data types with the TODO API library from AMD. We also saw the option that it might be possible to share ram between these two parts. We opted to switch back to the fir

implementation and instead of using an array to exchange the data, we send the data one by one via the existing working exchange of integer values. We managed to build a working version around that idea. After confirming that we have a working version, we refactored the code and finished that part of the implementation.

Most of the time for this implementation was spend on the issues that we had. The implementation of the sorting function and the implementation in the end were quite similar to the c-code implementation. We estimate that we spend 75% of our work in this implementation with dealing with the issues. This can be atributed to the both the lack of knowledge of the tools and libraries as well as a worse debugging experience which explain later in (TODO link to subsection). Due to these issues, we tried a lot of different approaches as well as using ChatGP(TODO footnote). The solutions from ChatGPT did not prove to be valuable for us. Either it referred us to solution which worked in c but not with the interface between the two components or it created solutions which were overly complicated.

IV. COMPARISON

A. Design and Implementation Complexity

In this subsection we want to evaluate how complex it is to design and implement each approach. The FSMD and the ASMD are very close from the design point of view since they rely heavily on each other. In section subsection III-A the design process was already explained for these approaches. Even though the task was not easy we still managed to do it by contibutions of the whole team. The design phase for the FSMD and ASMD took approxemately half a day of work.

The implementation of the FSMD and the ASIP varied from each other. Mainly, they differed because of the logic which we used. In FSMD we could use the states that where described in the ASMD. The statements in the ASMD are very close to the final code. While implementing we found out that we had some errors in our design, which could be fixed fast. In total, it took us one and a half day to implement the FSMD.

On the other side, the ASIP implementation needed more abstraction. We converted the states in multiple smaller opcodes, which were developped and then reused multiple times. We needed to figure out the right opcodes, define them the right way and also implement them correctly in our code. This took a little longer than the FSMD. The abstraction was higher, the readability gets more difficult and thus errors where overlooked more easily. Our main source was the lecture and the working code that we implemented while lecture week. All in all, it took us two and a half days to implement the ASIP.

For the C implementation the designing was rather easy and straight forward. We used our knowledge of the sorting algorithm and the internet to come up with a first draft. For the implementation we used the draft and made small adjustments mainly to include a serial port receiving and transmitting. The C-code design and implementation step was by far the fasted. It only took us one day of work. Obviously, we all had experiance in C coding and that can be one of the reasons why the implementation was done so quickly.

Starting the last implementation - the IP Block - we thought of an easy but yet clean design. We wanted to include the already finished C code in the example code of the FIR filter. To do so, we needed to send an whole array - with multiple cells - from the application component to the host component. We did not find a way to do this, which might due to our missing experience in this topic and the lacking documentation in the internet. We came up with other design ideas such as trying to copy the array, creating a self-definied struct or to forward a pointer to the array.

We implemented all of these ideas and they all run in our C-Simulation and our testbench file. When we were trying to run it with the CO-Simulation, they did no longer work as we expected. Our final design idea was to adapt our C code solution to the code of the aready working FIR-Filter. The design of the implementations took the longest for the IP-Block. We worked roughly four and a half days on the different designs and implementations. We expierienced a lot of different behaviour in the implementaton of the IP-Block.

To conclude, if the design and implementation complexity is a requirement for choosing an implementation we would evaluate the four implementations as follows: Worst to design and implement is the IP-Block. Here, very specific knowledge is required, and an understanding for the set-up of the application and the host component is needed. Only because an implementation is working within the C-Simulation and the testbench, it will not neccesairy work with the CO-Simulation and on the zybo board. Additionally, the waiting times of the different compiling times, the lack of a good debugging tools and the missing documentation on the internet make the implementation even more difficult.

The ASIP is slighly better from a design and implementation point of view: Once the logic of the algorithm is understood the design phase is rather straight forward. For the implementation, a further abstraction step is needed. This can cause problems. Since the opcodes with the register addresses and the immediate values are written in hexadecimal numbers, the readability decreases, so people are prone to overlook errors.

The FSMD can help with the problem of readability. Its top file includes all states that were defined earlier in the ASMD. It is easy to compare the ASMD witht the code and check if there are any mistakes. So implementation is easier. This is why from a design and implementation point of view FSMD is a decent choice.

The best implementation that can be chosen in order for a low design and implementation complexity is the C-Code. A lot of people know basic C programming and there are great documentations on the internet. In contrast to the other solutions AI can help a lot. It can come up with a code, find errors and help with the implementation.

B. Debugging and Verification Complexity

The debugging strategies and the complexity of debugging varied greatly in the different implementations. In both the FSMD and the ASIP, we used the simulation in Vivado. In both cases we had to customize the signals that we want to simulate. The difficulty with this debugging lies in finding the first unexpected or abnormal change. In both cases we used both our diagrams to go through the flow and compare our expected values to the simulated values until we found the outlier. This results in two or even three people doing the debugging. Since we worked as a team on this task, this did not present any resource issue. Nevertheless this form of debugging felt slower and more complicated compared to bugging python code in a modern IDE in which you can mark the line of code, to which you want to jump. Although being a slow process, it still proved to be an effective tool to find any mistakes that we made. We managed to find all our mistakes with this form of debugging. One benefit of this debugging method is that you do not have to know where you want to debug. Since you get many singals at once, the unnormal behaving signal is very likely included. This reduces the amount of restarting the debugging procedure itselfs.

For the c-program we used print-statements for debugging. We viewed the output directly via the serial connection in putty. This form of debugging was quite familiar to all team members. It was intiutive and faster compared to the method above, since we did not have to search for the data. However the print statements have to be included in various places and also have to be deleted afterwards. The code is therefore changed for debugging. Despide this downside we managed to debug quite a bit faster than the first method.

Even though the IP block is also code which is written in c, the debugging experience is quite different and more complex than in the method before. Since there are seperated components, the complexity increases and there are more things which can work or not work and need therefore debugging. The building of the hls component and afterwards the c-simulation could be debugged as normal c-code could be debugged. The difficulties start at the RTL simulation. The RTL simulation could fail without giving good indications of the reasons for it. This could be due to issues in transfering data or because of not sending data or for running into a loop which was not cought in the c-simulation. As far as we could see, there is not even a debugging tool for it. Our solution was to uncomment sections of the code until we find the section which causes the issue. This is complex and prone to errors since the code logic is heavily effected by the debugging. When you RTL simulation worked, it produced a waveform file. However this waveform file was more complex and difficult to understand since. There are many signals in the waveform file to represent the logic of the c code. However they are auto generated and therefore have unintuitive names. We tried debugging with this but could not make any use of it. The next point of debugging is in the application component. However at this point there are many possible areas in which the bug itself could be located. This means that we can print the values which we send and receive of the hls component but depending on the results, we can not located the bug clearly. Due to the lack of knowledge of this interface between the components, we made many mistakes with the interface itself which were neither located in one or the other component and were therefore difficult to debug.

The different implementations result in various different debugging experiences and required skills for debugging. We spend the most time for debugging on the last implementation which was unexpected. The complexity of the interaction between the various different parts resulted in a complex debugging process. Further the lack of tools for debugging this complex situation, resulted in a slow process. So the result is that the third implementation was by far the fasted to bebug. Both the first and the second implementation took some time to debug but it was still manageable. The last implementation was very difficult to debug and it was unpredictable how fast an error could be found. This should be taken into the cost consideration for the implementation.

C. Scalability, Flexibility, Reusability and Maintainability

The flexibility/adaptability and reusability of the solutions varies a lot. Our first implementation is complementy written in VHDL. The FSMD was just specificly designed for this sorting algortihm. There is nothing flexible in this implementation. If we want to change the sorting algorithm, we would have at least to redesign the full control flow. Depending on the sorting algorithm, we might need more additional components. Everthing has likely to be renamed and restructured. At that point a new algorithm would effect anything in the code but the most basic components. Any other algorithm besides sorting algorithms would likely change all code. Anything which is already implemented in hardware has to be changed fully or is not reusable.

The ASIP implementation is a little bit more flexible. A change in sorting algorithms changes the control flow. However it depends on the sorting algorithm if anything else needs to be changed. Since the instruction set can be reused for anything, similar algorithms can also be run on the ASIP. Just the instruction ram needs to be exchangeable. This enables small changes even after the hardware is created. However the complexibity of implementing a change is rather high. It is not easy predictable what consequences a single new instruction can have on all other instructions.

The fourth implementation is more flexible than the first two implementation. The huge flexibility benefit comes the fact that is it not specified how the IP blocks are used. The IP blocks act as predefined algorithms which are likely to be used and optimised. What data and how the data flows into these algorithms, is not specified at the time of creating the hardware. Since there are many algorithms for different standards, this concept can be seen in many modern computers and chips. Standard algorithms for video encoding and deconding can be implemented directly into the hardware. Other algorithms for hashing or encryption/decryption of messages can also be implemented in hardware to increase the speed of opertaion.

Any mistakes in the implementation of the IPs, can however still result in high costs and broken products if this is not detected before creating the hardware.

The third implementation is the most flexible implementation. In this implementation there is nothing specific about the hardware. The code can be deployed to any hardware which can run the c code. Any mistakes in the code can be fixed by redeploying the code to the hardware. Any new algorithms can also be deployed, additional functionality can be added afterwards and maintaining it is possible as long as the hardware does not break. This makes it undoubtfully the most flexible implementation. However at this point we lose the benefits of implementing algorithms in hardware.

The solutions one, two and four offer different kinds of flexibility. Depending on the clarity of requirements and the maintainability it might not be possible to chose the first or second implementation. Therefore implementing a few crucial algorithms as IP blocks, might be the middle ground which is chosen to ensure that there is still enough that can be changed after deployment. If even that is not possible, the only solution that remains is writing non hardware specific code.

Scalability? What would you write about that??

D. Other comparing methods

E. Cost (Development and Production) and efficiency

REFERENCES

REFERENCES

[1] "Insertion sort algorithm," *GeeksforGeeks*, 7.03.2013. [Online]. Available: https://www.geeksforgeeks.org/insertion-sort-algorithm/.