Design Space Exploration For VLIW Processor

Kaustubh Agarwal(4823168) and Yuanhao Xie(4624424)

I. INTRODUCTION

The aim of this assignment is to make a preliminary design for a VLIW processor for the two given benchmarks, namely Matrix and Pocsag. This report documents the Design Space Exploration process by varying parameter values in the configuration file of the ρ-VEX platform and identifying the optimal design for the execution of the benchmarks in terms of performance and area utilization metrics. We also detail the reasons which led us to choose the respective options. The design choice will eventually depend on the particular application for which we will have to optimize the benchmarks

The report is divided into sections where section II gives a description of the Benchmarks. In section III, we explain our choice of application which makes use of the provided benchmarks in its operation. Section IV illustrates our observations and justifications with the help of graphs. Section V describes the Singular optimal design for both benchmarks. Section VI gives information on the compiler optimization for both benchmarks while section VII gives our conclusion on the whole Design space exploration process.

II. BENCHMARK DESCRIPTION

We were assigned the following benchmarks namely - Matrix and Pocsag. After going through the respective C codes for both applications, we made the following observations-

A. Matrix

- This Benchmark computes the product of the two 64 bit matrices and then checks the result by comparing with a default matrix. After looking at the C code we observed that we can execute the code much faster if we can use instruction level parallelism as the operations would be independent of each other. In the first loop there would be no data dependency as no value is dependent on any other value and we can execute them in parallel.

B. Pocsag

- POCSAG is an asynchronous protocol used to transmit data to pagers. It uses one-way Frequency Shift Keying paging protocol that supports 512, 1200, and 2400 bits per second. It uses a BCH Error correction code to detect and correct one or two bit errors. BCH provides a 6 bit hamming distance between all valid code words. After looking at the C code we observed that we couldn't use Instruction Level Parallelism to the extend as we used in the matrix code. There are a lot of function calls in the code and a lot of compare loops which we think will increase the branch operations in the assembly code. A lot of instructions will be dependent on each other as in almost every communication protocol! We assume the code will follow a linear order and out of order execution and instruction level parallelism will be very hard to achieve in this benchmark.

III. APPLICATION

The application as we could understand from the benchmarks(Matrix and Pocsag) must be a sort of a embedded paging device. It should be able to encode and decode the paging signals. We assume this device will have a small size for it to be easily carried by a person and should consume very less power as in the case of Pocsag protocol. We would want to lower the area while not degrading the performance too much as the device would be doing Digital signal processing and should not have much latency. We will start our design space exploration keeping these parameters in mind.

IV. OBSERVATIONS AND JUSTIFICATION

A. Area of the system

In this section we describe how we calculated the area of our design and the assumptions which we took during the process-

- The number of Clusters is one.
- For each Issue slot there are two read ports and one write port.

$$AREA = A_{ALU}+A_{MULT}+A_{LW/SW}+A_{GR}+A_{BR}+A_{CONN}$$

where:

A_{ALU} = 3273 * no. of ALU units
A_{MULT} = 40614 * no. of MULT units
$A_{LW/SW}$ = 1500 * no. of Load/Store units
A_{GR} = 412.3125 * no. of Registers*(IssueWidth*IssueWidth/16)
A_{BR} = 32.25 * no. of Registers*(IssueWidth*IssueWidth/16)
A_{CONN} = 1000 * no. of connections to data lines

B. Matrix benchmark

The matrix code was meant to multiply two 64 bit matrices and compare the result with a default matrix. In order to find the optimal configuration of the system we decided to first run the code against the default parameters mentioned in the "Configuration.mm" file. In the default configuration the issue width was set to 4, number of ALU were 4, Number of Multipliers were 2, number of Load/Store Units was 1, number of connections for load,store and pre-fetch operations were 1, number of 32-bit general purpose registers

were 64 and the number of 1-bit branch registers was 8. After examining the code in the ”ta.log.001” file and the assembly code file ”matrix.s”, we made the following observations-

- The number of execution cycles was 653141
- A total of 16 Multiplication commands(mpylu and mpyhs) were executed in the code
- A total of 32 Addition commands were executed in the code
- A total of 24 32-bit general purpose registers were used during the process
- A total of 1 1-bit branch register was used during the process

After making these observations we concluded that the maximum number of multipliers that we can put in the system is 16 if all the multiplication commands can execute in parallel. We executed the code again with putting 16 multipliers,16 ALUs and increasing the issue width to 16. We achieved execution cycles of 282126 with a speedup of almost 2.31 times! as many commands didn't have any data dependency and they executed in blocks of 16.

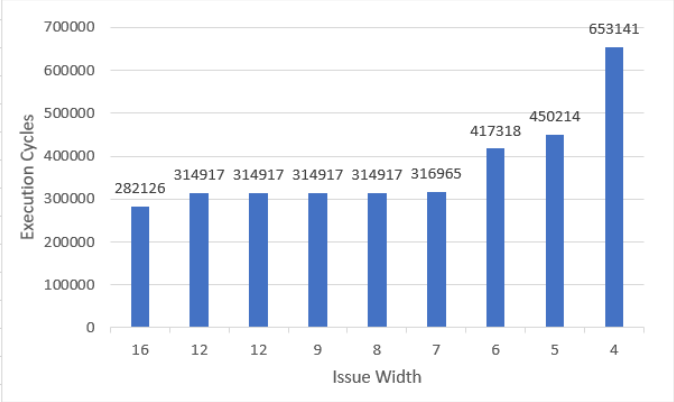

Fig. 1. Number of Execution cycles of Matrix benchmark with respect to Issue width

We then also checked the execution cycles for a issue width of 32 and to our correct prediction there was no increase in speedup as the execution cycles were same at 282126 but our area increased almost 8 times(Law of diminishing returns in effect)! In order to reduce the area we decreased issue width to 12 hence lowering the area by 32% of the previous issue width of 16. In this case we got execution cycles as 314917 - an increase of 11%! According to us this is a negligible degradation in performance in comparison to the reduction in area of the system. When considering the issue width as 8 we achieved same execution cycles of 314917 while reducing the area by 40%! After examining the ”matrix.s” file we found out that the file with issue width as 8 used only 3 branch registers instead of 4 hence reducing the branching operations which compensated for the reduction in other resources! Further reducing the issue width, ALU,Mult to 6 did not help us as the number of execution cycles went up to 417318. We decided to keep the issue width at 7 as our performance(316965 execution cycles) was still good while lowering the issue width(7) to as low as possible.We then tried to find the optimal number of units for other resources. We further reduced the area to 366189 from 419917 with virtually no increase in execution cycles by tweaking the other resources(we kept MEM STORE/LOAD =6 and MEM prefetches = 0) by the information gained from the ”matrix.s” file. Further reducing the value of ALU/Mult gave us an significant increase in execution cycles. For this code we would choose the design at issue width 7 corresponding to the last entry in table I as we still have a small area and a considerable number of execution cycles. In order we even tried to put the value of multiplier as zero to see the effect on execution cycles but we observed the same value as putting the number of multipliers as 2 which concludes that the default value of multipliers in the ρ-VEX platform. We have shown the effect of issue width on the number of execution cycles on the Pocsag benchmark in Figure 1.

TABLE I
EXECUTION CYCLES AND AREA COMPARISON IN MATRIX BENCHMARK

Execution cycles	Issue width	ADD	MUL	MEM	GR	BR	Area
282126	16	16	16	16	64	8	1200528
314917	12	12	12	12	64	4	819297
314917	8	8	8	8	64	4	493164
316965	7	7	7	7	64	4	419917
417318	6	6	6	6	64	4	349985
653141	4	4	2	1	64	8	125466
417318	7	6	7	6	30	1	362916
349862	7	7	6	6	30	1	325575
317094	7	7	6	6	30	1	366189

- As we can infer from Figure 1 and Table 1 the highest performance that we have is 282126 for issue width 16
- The most optimal point that we can have is keeping issue width as 7 which results in 349862 execution cycles while keeping a comparatively lower area.

C. Pocsag benchmark

Similarly for the Pocsag code, we ran it against the default configuration and after examining the ”ta.log.001” and ”Pocsag.s” file we observed -

- The number of execution cycles was 16753
- A total of 0 Multiplication commands(mpylu and mpyhs) were executed in the code as no multiplication was required. This gave us an indication of having zero multipliers in our design which will drastically reduce our area!
- A lot of function calls were made which were indicative of decision making.
- A lot of branching was done as the code required to check a lot of values which was indicative that we should use a lot of branching registers for our design!
- There was a lot of dependency between different values as the code required to check values at a lot of steps so as to follow the Pocsag paging protocol.

For finding a suitable design point we concluded to keep Memory loads as 4, Memory stores as 4 and memory

prefetches as 0. Increasing these parameters didn't improve our performance while still adding to the area and reducing them resulted in a not so negligible performance decrease. We decided to keep 62 general registers and 5 branching registers after examining the "ta.log.001" file and "Pocsag.s" file. and simulating the code. Some interesting findings are mentioned in Table II. After increasing the issue width to 8 we got the minimum execution cycles after which increasing the issue width and the resources didn't decrease the execution cycles. We also found the optimal value of memory syllables as 4 after examining the "Pocsag.s" file which showed a maximum of 4 memory executions taking place in a cycle. Considering the default configuration as starting point we observed many interesting design points. If we decreased the issue width from 4 to 3, we observed a 72% decrease in size with just 15.5% increase in execution time. If we increased the issue width to 5 from 4, we observed a 44% decrease in area and a 6.6% decrease in execution cycles! If we increase the issue width from 5 to 6 or from 5 to 7 we get a negligible decrease in execution cycles of at 3% but the area would increase a lot(around 63%). For this code we would choose the design at issue width 5 as we still have a small area and a considerable number of execution cycles. We have shown the effect of issue width on the number of execution cycles on the Pocsag benchmark in Figure 2.

TABLE II

EXECUTION CYCLES AND AREA COMPARISON IN POCSAG BENCHMARK

Execution cycles	Issue width	ADD	MUL	MEM	GR	BR	Area
15050	16	16	16	16	64	8	1200528
15050	8	8	0	4	62	5	143083
15087	7	7	0	4	62	5	115693
15529	5	4	0	4	62	5	70560
16623	4	4	2	1	64	8	125466
19206	3	3	0	3	62	5	34789
26467	2	2	0	2	62	5	19977

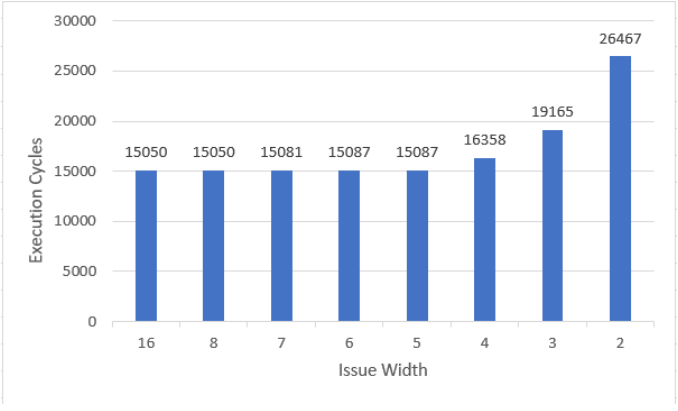

Fig. 2. Number of Execution cycles of Pocsag benchmark with respect to Issue width

- As we can infer from Figure 2 and Table II the highest performance that we have is 15050 for issue width 8
- The most optimal point that we can have is keeping issue width as 5 which results in 15529 execution cycles while keeping a comparatively lower area.

V. SINGULAR OPTIMAL DESIGN FOR BOTH BENCHMARKS

After finding out the points for highest performance and optimal performance we decided to compare both design values and tried to come to an agreement. This was a very cumbersome task as both programs were fundamentally very different and had different requirements. The matrix code can run very fast on a wide VLIW chain(Issue width =16) but the pocsag code does not require the large VLIW as it saturated very early(Issue width =7). We had to make sure that we take a large enough Issue width so as to cater the matrix benchmark but also consider it's effect on the pocsag benchmark. The pocsag benchamrk doesn't require any multipliers but the matrix code requires a substantial number of multipliers otherwise it's performance nosedives. The pocsag code also requires a lot of registers so to cater a healthy number of function calls and branch predictions while the matrix code utilizes a minimal amount. We decided to keep the number of General purpose registers as 62 and branch registers as 5 as they are rather inexpensive. We also kept the memory load/store connections as 6 each and memory prefetches as 0.We decided to keep the issue width as 7 as the optimal point by observing Figure 3 and after examining the "matrix.s file" we observed that we can't decrease the number of adders as if we do that we won't be able to take advantage of no data dependency. Finally we decided to go with the final configuration of 7 ALU units, 4 Mult units and 4 Load/Store units with a issue width of 7 which gave us an area of 278149 and an execution cycles of 382501 for matrix benchmark and 15087 for Pocsag benchmark.

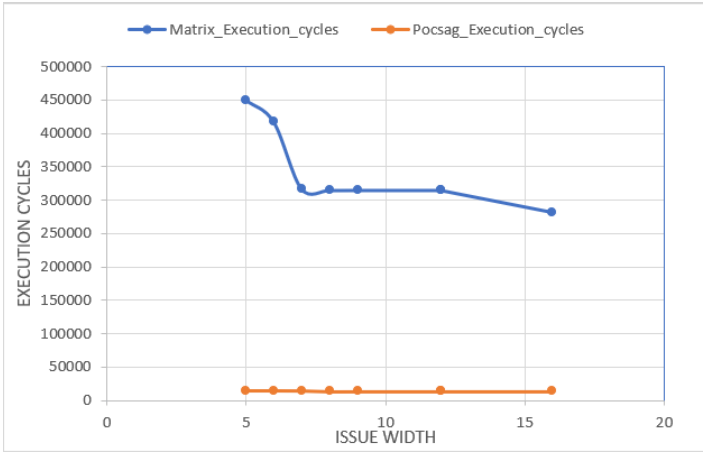

Fig. 3. Number of Execution cycles of Pocsag and Matrix benchmark with respect to Issue width

VI. COMPILER OPTIMIZATION

We decided to further reduce the execution cycles of both benchmarks for the Singular Optimal Design. We used pragmas for the matrix benchmark while we used autoinline

compiler flag optimization for Pocsag benchamrk. The statements in the individual iterations of the loop in the Matrix function are independent of each other and are completely parallelizable. Hence, we confidently inform the compiler that there are no memory aliases by specifying the pragma ivdep directive. We used a loop unrolling pragma with a first unrolling phase value of 32, and used the precondition pragma with a value of 32. The execution cycles reduced from 382501 to 263719 - a decrease of 31% for the same area as shown in figure 4.

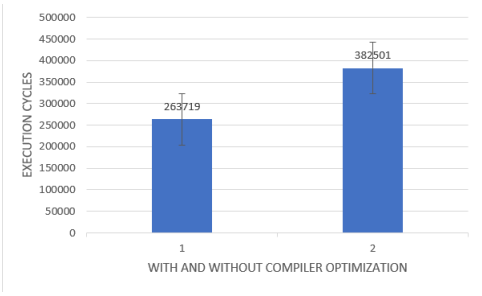

Fig. 4. Number of Execution cycles of Matrix benchmark with and without compiler Optimization

For the pogsac code we used the autoinling optimization to enable automatic function inlining to inline functions in the same module and reduced the execution cycles from 15087 to 11809 - a decrease of 22% for the same area as shown in figure 5.

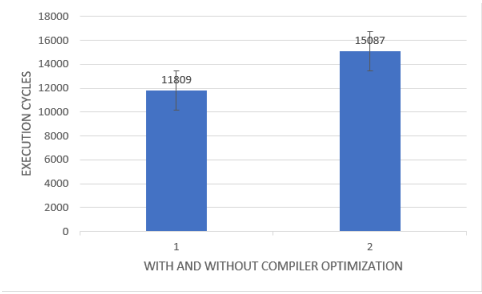

Fig. 5. Number of Execution cycles of Pocsag benchmark with and without compiler Optimization

VII. CONCLUSIONS

After providing a particular design space for both benchmarks we concluded that the VLIW processor can be optimally reconfigured for a particular application. In our case choosing a common design point for both applications proved to be a very cumbersome task due to the fundamental differences between the two benchmark codes(Pocsag and matrix). We realized how hard it is to choose a particular design space considering the various trade-off between performance and area of the system. We learned how data dependency and structural hazards affect the performance of a system. We analyzed the performance of the processor for various different possibilities.

For pocsag we saw that the code was very linearly dependent and increasing the parameters(like multipliers) didn't improve the performance drastically while the number of multiplier units were essential to the matrix benchmark. We analyzed the assembly file to see what kind of operations are being performed in the code and altered the number of functional units and issue width accordingly. Matrix code had very less dependency on Registers and a high dependency on the number of multipliers while the pocsag had zero dependency on the number of multiplier units and a high dependency on registers due to many branching statements. Based on our target application, we did a area-performance trade-off can be performed and we chose the best configuration. We also observed how a single resource block can affect the whole design space process because of the efficiency it provides and the excessive area which it adds to the system. Further, we improved the performance by optimizing autoinline flag in the compiler for pocsag benchmark and pragma for Matrix benchmark.

REFERENCES

[1] ρ-VEX project. The Dynamically Reconfigurable VLIW Processor. http://rvex.ewi.tudelft.nl/
[2] Adam Hickerson. THE POCSAG PAGING Protocol. https://www.raveon.com/pdfiles/AN142(POCSAG).pdf.
[3] ρ-VEX user manual. Jeroen van Straten, TU Delft, 2017.