Minimal ALU design for CRIB architecture

Alish Kanani[1]**, Lukas Pfromm**[2]**, and Jake Neau**[3]

[1]**ahkanani@wisc.edu**
[2]**pformm@wisc.edu**
[3]**jneau@wisc.edu**

INTRODUCTION

The optimization of Arithmetic Logic Units (ALUs) plays a pivotal role in advancing the performance of computer architectures. Especially in the case of CRIB [1] type of architecture where performance is directly proportional to the number of ALUs in the execution unit. This is because the execution unit in CRIB is partitioned such that each partition contains multiple entries, and each entry uses a full 64-bit integer ALU to perform all integer instructions other than complex multiplication and division instructions. With this, it is clear that increasing the Instruction Per Cycle (IPC) in CRIB requires an increase in the number of ALUs.

This project aims to do an in-depth analysis of programs and see how we can reduce the area of ALUs. By achieving this reduction the project aims to concurrently decrease power consumption and potentially alleviate latency for simple integer ALU instructions. An ALU which is larger than needed is costly because typically all the instructions do not require a full 64-bit ALU. The upper limit of an unsigned 64-bit integer is approximately $18 * 10^{18}$. It is reasonable to assume that the majority of software applications do not typically handle data values of such vast magnitudes. The size of an ALU can be determined by the width of the operand it is expected to execute. To determine the optimal configuration, we first did a basic study of the common spec2006 CPU benchmark [2]. Based on the benchmarking result, we implemented different ALU operand width predictors in the CRIB pipeline to determine the accuracy of different prediction architectures. We then tried many different ALU configurations to fit into the existing CRIB architecture and obtained the IPC impact of different configurations. The aim is to achieve more ALU resource utilization and power saving by dynamically adapting ALU configurations based on benchmark data and width prediction.

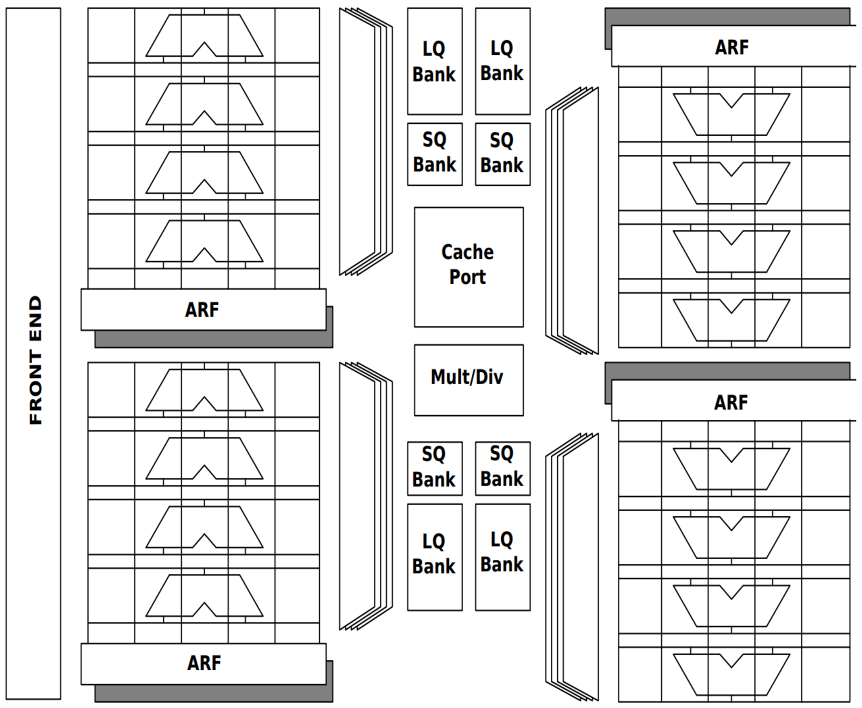


Figure 1. Partitions in CRIB [1] architecture

The rest of the report is organized as follows. Section 1 contains the required background of CRIB architecture. The detailed methodology for benchmarking, micro-architecture of width predictor, and improved ALU configurations were discussed in section 2. Section 3 contains results from all steps, section 4 concludes the report, and table 2 gives a summary of contributions from each team member.

1 BACKGROUND

The main idea behind the CRIB architecture is to have in-order execution and still meet the performance of out-of-order CPUs with significantly improved power efficiency [1]. Amongst many architectural features explained in the paper, one is to have a partitioned design where each partition has multiple entries as shown in figure 1. The idea behind having entries in the partitions is to allow the assignment of an entry for each instruction without worrying about the need to resolve dependencies. Therefore, as soon as instructions are decoded and the partition is empty, new instructions can be dispatched. This parallel dispatch can allow CRIB to have similar performance compared to out-of-order execution. Instructions can sit and wait in entries within partitions waiting for dependence, thus eliminating the need for complex register renaming, wake up & select logic, bypass network, reorder buffer, etc. which are required in typical out-of-order CPUs. These components contribute significantly to dynamic power overhead. Instead, CRIB stores architected registers in a simple rank of latches. This pipeline-free approach in the back-end of the CPU reduces power consumption by minimizing unnecessary data movement and complex logic operations.

One downside of this architecture is that each entry in a partition requires a full 64-bit integer ALU to perform common integer instructions. However, to reduce the ALU area the paper suggests having a common multiplier and division units between entries, it still requires full 64-bit ALUs for other operations. In this project, we explore an implementation of CRIB utilizing selective ALU size reduction and modified instruction routing.

2 APPROACH

In this section, we discuss our approach to implementing the minimal ALU design. Three key portions of the implementation are described, including Benchmarking to determine the optimal hardware configuration, the design of the Width Predictor, and the ALU design utilised in the modified CRIB architecture.

2.1 Benchmarking

The size of an instruction is the maximum of its input operand widths and its output value width. It should also be noted that since we will always schedule shifts to full 64-bit ALU, we don't have to worry about output operand width as for all other operations max output width can be max input width+1.

In order to determine the size of each ALU in the architecture, it is first necessary to determine the average width of instructions which will execute in the system. For example, if one quarter of all instructions required a 64-bit ALU to execute but all other instructions could fit inside a 32-bit ALU or smaller, it would be optimal to have only one quarter of all ALUs be 64 bits wide and all others can be smaller. This would minimize on the excess ALU area which would exist if 64 bit ALUs were implemented globally, as most of the bits in these other ALUs would generally not be used.

To determine the average widths of operands which could be expected to run on the system, we utilized a modified version of the GEM5 simulator [3] running the RISCV ISA [4]. The Simple Atomic CPU was utilized, as no information about memory access or out of order execution was required. At this stage, the only important information was the width of operands being passed to the ALU. Originally, gem5 was designed with a distinct separation between its ISA and CPU models. This intentional decoupling enhances configurability, allowing for the simulation of any ISA on various CPU models. However, this design choice makes it difficult to access register and immediate values within the CPU model. Specifically, it does not report the data stored at each operand register location, posing a critical limitation to understanding the state of the simulated system. We modified the simulator such that for every integer instruction being executed, the width of each operand is saved. This applies to both immediate values and operands that are stored in registers.

We chose to run a subset of the spec2006 benchmarks [2] to determine ALU widths. Spec2006 was chosen as it is one of the most common and comprehensive benchmarks available and presented a comprehensive picture of likely instruction widths. Seven benchmarks were run in total, which are discussed more comprehensively in the Results section. In order to run these benchmarks on our simulated CPU, they were first compiled for RISCV using cross platform compilation tools [5]. Each of these benchmarks was then executed individually, and the width of each integer operand was saved. These results were then compiled into a dataset which was used in the subsequent portions of the design process.

2.2 Width Predictor

As explained in section 1 CRIB is an in-order execution CPU, where instruction scheduling happens long before the operand dependency is resolved. Basically, instructions wait inside the CRIB entry for dependency to resolve instead of utilizing complex wake-up & select buffers. Because of this kind of execution model, operand width is not known to the scheduling logic. To resolve this issue, a prediction algorithm is required that can accurately predict the required ALU width of the instruction in the front end of the pipeline.

A typical program shows strong data width locality. To illustrate this, consider an instruction flow given in the below program, which adds 1-10 numbers to the x5 register. In this simple program, all instructions require an integer ALU either for data calculation or instruction address calculation. Here, every time instructions 5 and 6 would require a 16-bit ALU while instruction 7 would require the full 64 bits to calculate the branch address. Because of this repeatability, we can predict the required ALU width by storing the ALU width from the previous execution of the instruction.

```
1   addi x3,x0,0 # i = 0
2   addi x4,x0,10 # const 10
3   addi x5,x0,0
loop:
4   bge x3,x4, exit
5   addi x3,x3,1
6   add x5,x5,x3
7   j loop
exit:
```

Additionally, the predictor does not have to predict the exact size of the operand, predicting which range it belongs is enough. From the benchmarking study (section 2.1), we decided to have 16, 32, and 64-bit configurations. Therefore, the predictor's primary task is to predict the corresponding ALU configuration. We can eliminate the need to predict the width of multiplication and division instructions as the CRIB architecture inherently has separate units for these operations. In the benchmark study, it was observed that shift instructions make up approximately only 10% of all integer instructions. Therefore we propose to use full 64-bit ALUs for all shift instructions, eliminating the necessity for the predictor to accommodate shift operations.

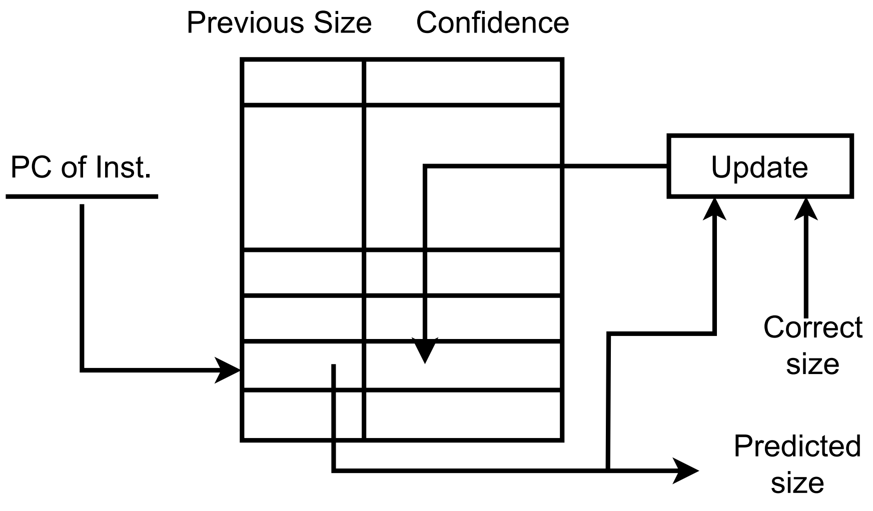
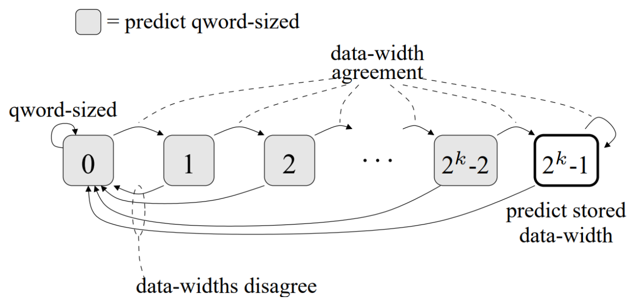


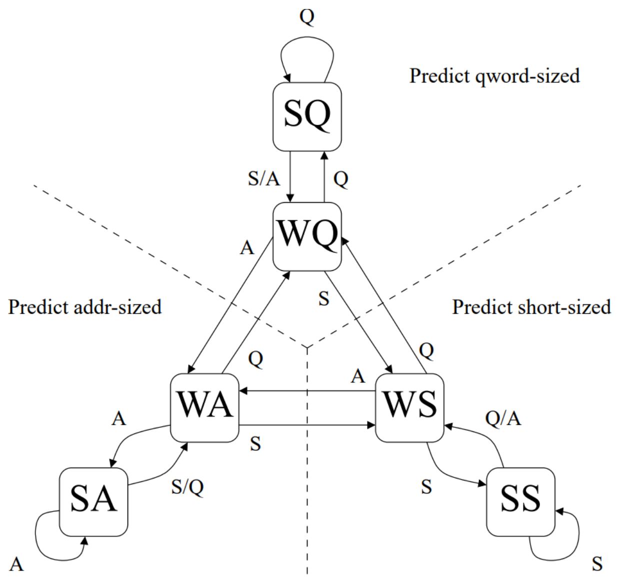
Figure 2. ALU width prediction table μ-architecture

Figure 2 shows the width prediction table. This table is indexed with the Program Counter(PC) of the instructions where each entry contains the previous ALU size and confidence value. As soon as the PC is known, it can start prediction lookup. If the instruction is integer ALU type (excluding multiplication, division, and shift), it can predict the ALU size of the instruction based on the previously executed ALU size and confidence of this value. If the instruction is not there in the table, it can make conservative predictions and after the execute stage a new entry with the correct size can be added. After the execution stage, if the prediction is correct, it would increase the confidence value. In the case of a misprediction, the correct size can be updated and the confidence value can either be decreased or reset to 0 based on which algorithm it uses to calculate confidence. As explained earlier, the width prediction table only needs to predict 16, 32, or 64-bit ALU width so the previous size can be represented with only 2 bits. This table looks similar to Smith's branch predictor [6]. There can be more complex prediction mechanisms but this simple approach gives good enough accuracy for ALU size.

To calculate the confidence value, we implemented two different approaches as suggested by [7]. Although our application of the predictor is different from this paper, it gives a good explanation of bit-length prediction and locality of instruction. The first update algorithm is a resetting counter. Here, as a confidence value, we increment a k-bit saturating counter which indicates how many times the previous width is repeated. If the counter is saturated, we are confident that it will repeat and the predicted size is assigned as the previous size. In case of misprediction, the confidence value is reset to zero and the correct size is updated in the table. This approach requires storing k-bits in the table as a confidence value as described in figure 3a. If the counter is not saturated (less than the maximum value of $2^k - 1$), the table can make a conservative decision and predict 64-bit ALU size. The second update algorithm is an extension of bimodal branch prediction. Since we need to predict between 3 configurations of ALU, [7] explains it as the trimodal predictor. As shown in figure 3b, each configuration consists of a weak and strong state. A confidence value of 0 is assigned to the weak state and 1 is assigned to the strong stage. In either case, the predicted size is assigned as the previous size. A misprediction in the strong state would move the confidence value to the weak state, while a misprediction in the weak state would change the value of the previous ALU size to another weak state. FSM given in [7] and figure 3b is self explanatory. Trimodal only needs 1-bit for confidence.



(a) k-bit resetting counter



(b) Trimodal FSM

Figure 3. Confidence counter for each entry in ALU width prediction table (taken from [7])

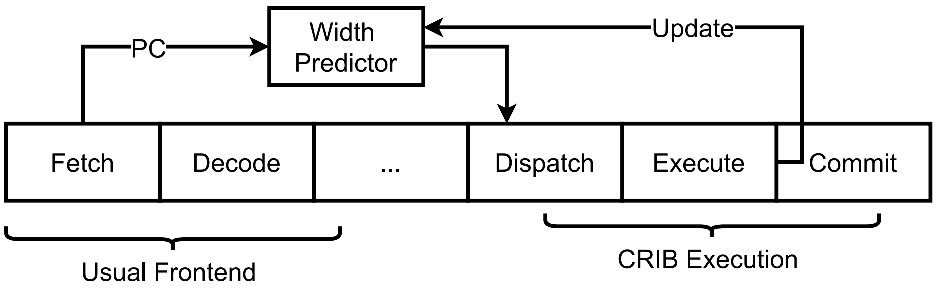
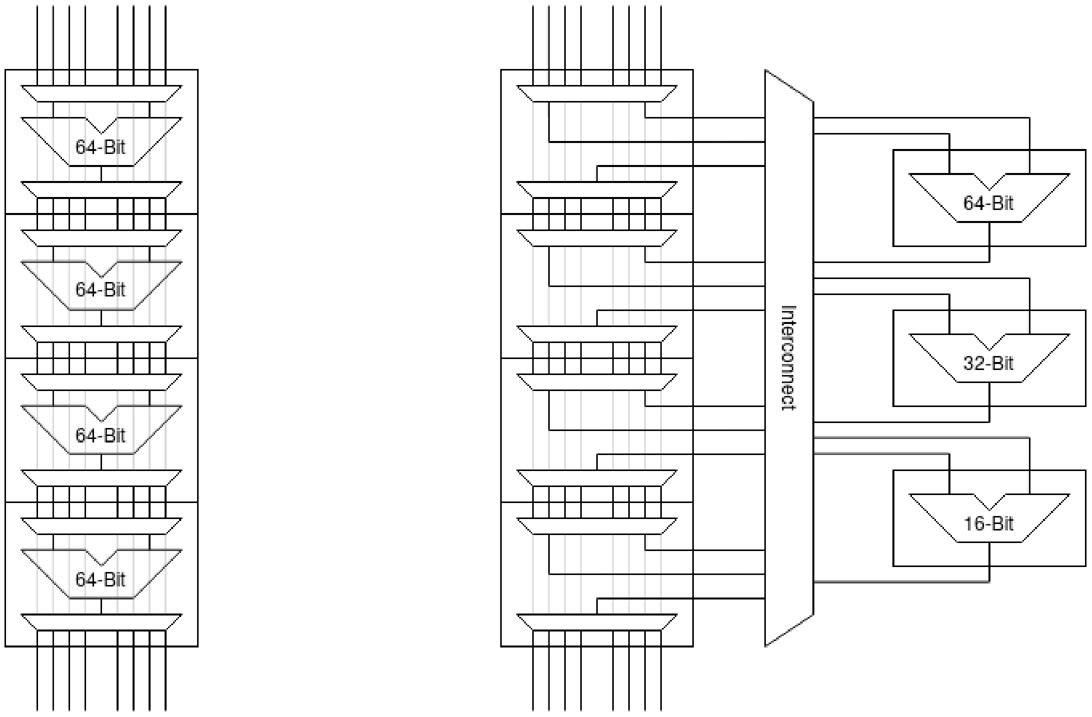


Figure 4. Width prediction in CRIB CPU pipeline

As explained above, to make a prediction, only the PC of the instruction is required. So the lookup can start in the early stage of the CRIB pipeline. If the instruction decoder decodes the instruction as integer ALU instruction (of course excluding multiplication, division, and shift), the scheduler logic in the dispatch stage will use the predicted ALU size to schedule the instruction. More details on ALU configurations and scheduling are given in the section 2.3. At the end of the execute stage, once the correct size of the operands is known, the width predictor will be updated. A high-level pipeline diagram with a width predictor is given in figure 4. As can be imagined, the width predictor is not part of the critical path. This is important as a less aggressive and power-efficient lookup can be implemented for width prediction and the table itself can be placed away from the critical pipeline path.

2.3 ALU Design

In order to reduce the space taken by the ALUs, we need to choose a scheme that can reduce the overall bit width while retaining as much performance as possible. There are three criteria that we judge possible designs on: scalability, flexibility, and overhead. Scalability means that as more CRIB partitions are added, the size and performance of the overall design remains reasonable. Flexibility means that the design retains performance across different types of program flow. Overhead is judged by how many extra resources are needed and how they negatively impact performance and area of the design.



(a) Base CRIB ALU design **(b)** A possible configuration of the ALU redesign (fourth design)

Figure 5. Comparison of CRIB ALU designs

The first design considered was to replace certain ALUs in a CRIB partition with their smaller width counterparts. The dispatcher was then used to assign instructions to the ALU in a partition that could support their width. This design has very good scalability with low overhead because partitions would dispatch instructions separately and the only needed modification is a slightly more powerful dispatcher. Where this design has issues is in flexibility. CRIB partitions have an inherent ordering where dependent instructions need to be located further down the partition from their parents in order to receive the correct operands. This design limits us to dispatching large width instructions either strictly before or after their smaller width counterparts. This means the design can only account for one type of ordering between large and small dependent instructions and would suffer a performance penalty with the opposite ordering.

The second design considered was to replace all 64-bit ALUs with 16-bit ALUs and break larger instructions into a series of 16-bit instructions. This design has good scalability as we can assign instructions to ALUs as normal. Where this design suffers is overhead. We need a fairly powerful unit to do instruction conversion, and shift instructions are hard to split up. This means that although most of the operations in the ALU could be done in 16-bits, we would need a full-size shifter in each unit, and the shifter is one of the largest parts of the ALU.

The third design considered was to have replace all ALUs with their 16-bit counterparts and have a shared resource for larger width instructions. This design has good flexibility, and low overhead due to only needing one unit with a large ALU for the whole chip. The design suffers from scalability as when more partitions are added, there will be more contention for the larger width shared ALU, especially in programs like hmmer which were found to have a large amount of 64-bit and 32-bit instructions.

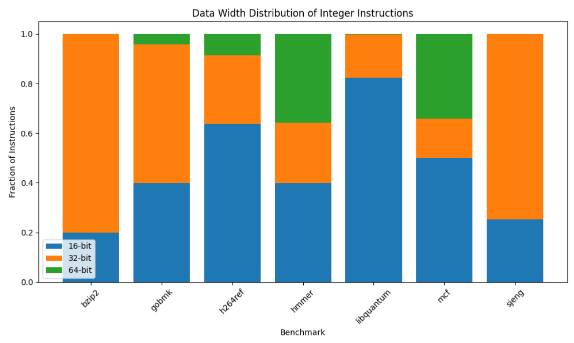
We implemented our fourth design where ALUs are removed from the CRIB units are accessed across an interconnect. Each CRIB partition has its own interconnect to a set of ALUs that can be assigned for execution. One CRIB unit is able to reserve one ALU for the duration of execution in the CRIB partition. In the event that an instruction requires an ALU that is already reserved, we end dispatch and assign the instruction to an ALU that time around. With this design, we can even test configurations with less than four ALUs per partition as some instructions like memory operations do not need them. This practice can be replicated for many partitions so it is scalable. We do not have the instruction dependence problem as in the first design, so it is quite flexible. The overhead is also small as the interconnect only needs two four-to-one multiplexers per ALU (assuming four CRIB units per CRIB partition). Another benefit of this design is that four ALUs are not needed for one CRIB partition. Some instructions like memory operations will not use an ALU, so it may make sense not to include four ALUs in a partition if all four will not be used during execution.

To implement the fourth design, minor modifications are needed in the dispatch unit. The number of bits an instruction will require can be inferred at dispatch using the width predictor. An instruction will be dispatched to a CRIB partition if the right-sized ALU that the instruction requires is not claimed. If the right-sized ALU is claimed, the dispatcher can also use a larger ALU if one is available. If an ALU is found, the dispatch unit marks that ALU as claimed. If no ALU is found, dispatch ends and the current instruction will be assigned next time around. The mapping of CRIB unit to ALUs is stored in the interconnect to facilitate communication between them. This is demonstrated in Figure 5. During the execution stage, the true width of the instruction is checked. If the true width is too large for the currently assigned ALU, the current instruction and all subsequent instructions are squashed and replayed. This width detection is not much overhead as a simple OR tree is required for the remaining bits. For example, assume that the width predictor predicted that instruction needs 32-bit ALU for execution. When the dependency of instruction is resolved and ALUs start executing, an OR tree of 32-bit would perform OR operation for the remaining 32 bits. If the output of this logic is 1, it indicates that the predictor has mispredicted the required ALU width, and the instruction will subsequently be re-executed using a 64-bit ALU.

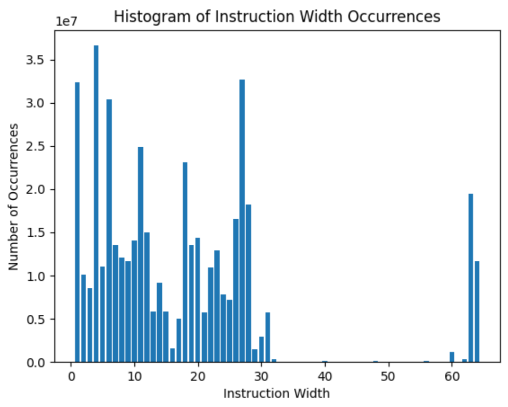
For the ALU modifications themselves, they are scaled based on their bit width. For example, all instructions in the 16-bit ALU require only 16 bits. This allows the interconnect to use smaller bit width multiplexers for the smaller ALUs to reduces space further. All shift logic is moved to the 64-bit ALU as it was found during testing that shifts only account for around 10% of all integer instructions and the shift unit takes up a large amount of space. With these modifications, we estimate that a 16-bit ALU will be at least 75% smaller (including the interconnects) than a full-width ALU in the base design. Similarly, the 32-bit ALU is estimated to be 50% smaller than the full-width ALU. Interestingly, a smaller width also means that operations like add and subtract will be faster in the smaller ALUs compared to the larger ones.

3 RESULTS

This section includes results for benchmarking, width predictors and different ALU configurations discussed earlier.



(a) Data width distribution by minimum ALU size



(b) Distribution of instruction widths across all benchmarks

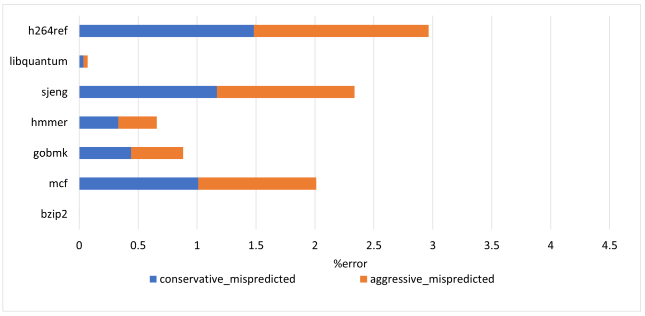
Figure 6. Results of the Benchmarking experiments

3.1 Benchmarking

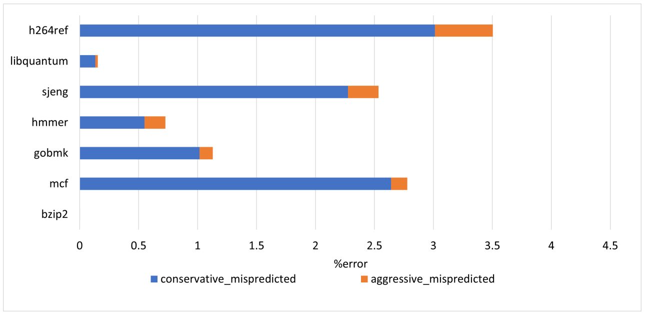
Figure 6 shows the results of the benchmarking experiments described in the approach. Each benchmark was executed for 200M instructions on the aforementioned Atomic Simple CPU using the RISCV ISA. The results show a wide distribution of instruction width requirements, which is a positive for our work as it confirms that our implementation was successful and gives a good indication of what real-world instruction widths would look like. It can be clearly seen that a vast amount of instructions can be executed using a 32-bit ALU, and about half of those would fit within a 16-bit ALU. We do not focus on any of these benchmarks in particular, but rather on the results as a whole since we are not designing the system to execute any particular workload. These instruction widths are then used to influence the design of the subsequent work.

3.2 Width Predictor

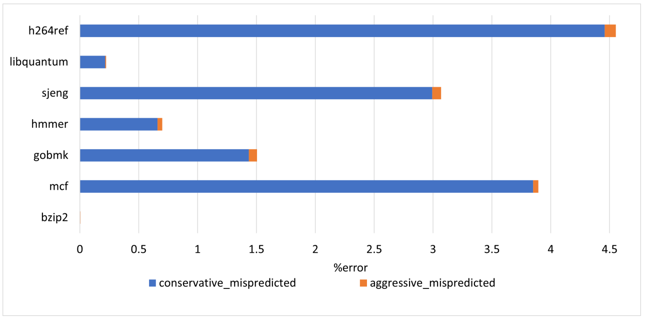
We implemented both resetting counter and trimodal FSM based width prediction table for the CRIB CPU model in the gem5 [3] simulator. First, we ported the CRIB(i.e. REVCORE CPU) to the latest version of gem5 as there were some modifications in the RISC-V ISA [4] because of which we were not able to simulate some instructions compiled with RISC-V cross-compiler [5] in the existing simulator.



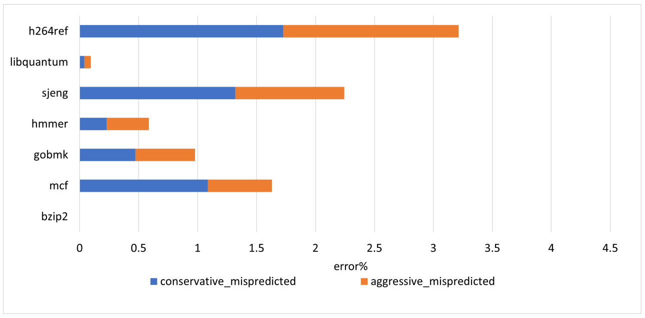
(a) Resetting counter with the saturating value of 1



(b) Resetting counter with the saturating value of 3



(c) Resetting counter with the saturating value of 7



(d) Trimodal FSM

Figure 7. Average conservative and aggressive misprediction percentage for width prediction with resetting counter and trimodal FSM

There are two types of mispredictions in width predictors: conservative and aggressive. An aggressive misprediction occurs when the instruction requires a larger ALU, but the predicted size is smaller. On the other hand, a conservative misprediction happens when the predictor predicts a larger ALU size than what is actually required. Distinguishing between these two types of mispredictions is crucial because, in the case of conservative misprediction, the only loss is the under-utilization of the ALUs, but instructions can still be executed without any issue. On the other hand, for aggressive misprediction, the instruction would need to be squashed and replayed with the correct-sized ALU, directly diminishing the performance of the CPU. Therefore, reducing the rate of aggressive mispredictions should be the primary target to achieve the same performance as the base CRIB design.

For resetting the counter, we implemented counters which can saturate with values 1, 3, or 7 corresponding to 1, 2, or 3 bits. For this study, we simulated 7 integer benchmarks from the spec-2006 CPU benchmark suit. For each benchmark, we used 100M instructions for warm-up and the next 100M instructions for accuracy calculation. The results for all implemented width predictors are given in the figure 7. For experimental purposes, we did not set a limit on width predictor table size. Still, we found that even for 100M instructions only a few thousand new entries were added to the table because of this, we expect that in a realistic case, a table with a few hundred entries would give similar accuracy. As can be seen in the plots, the width predictors perform well giving less than 2% average error combining both conservative and aggressive misprediction. While the resetting counter with a saturating value of 7 gives the highest overall error, as explained above, our primary goal is to minimize aggressive mispredictions. This makes it the best-suited option for our application, as the average aggressive misprediction rate is less than 0.05%.

3.3 ALU Design

A gem5 model of CRIB was used to implement the new ALU design. Modifications were made to the dispatch unit to allow dispatching to different-width ALUs. The squashing of mispredicted width instructions was attempted, but the required methods for squashing instructions are too closely intertwined with the branch predictor unit and were not able to be meaningfully extended to width prediction. As such, it is assumed that the width predictor has perfect accuracy for the sake of testing.

There are two other simplifications made for the sake of testing. First, the delay from the interconnect is negligible compared to the delay of the ALUs, so it will not be modeled. Second, we do not know much faster some instructions will be in the smaller ALUs without logic-level testing, so it is assumed all ALUs will take the same amount as the full-width ALU in the base design of time regardless of their width. For size comparison we assume size is proportional to ALU width.

ALU Combination	bzip2	mcf	gobmk	hmmer	sjeng	libquantum	h264ref
Base	0.63041	0.29965	0.47827	0.85684	0.26063	1.10009	0.87586
1 64-Bit, 1 32-Bit, 2 16-Bit	0.63012	0.29409	0.45894	0.73705	0.26001	1.09698	0.82270
1 64-Bit, 1 32-Bit, 1 16-Bit	0.63016	0.29361	0.45645	0.73718	0.26001	1.09692	0.81996
1 64-Bit, 1 32-Bit	0.63024	0.29203	0.44597	0.73104	0.25721	1.09047	0.80286
2 64-Bit	0.62987	0.29587	0.45787	0.79965	0.25907	1.09215	0.840023

Table 1. Recorded IPC for different ALU combinations on spec-2006 benchmarks

Five different configurations of ALUs were tested:

- The base design (one full-width ALU in each CRIB unit): 100% of base size
- One 64-bit, one 32-bit, and two 16-bit ALUs: 50% of base size
- One 64-bit, one 32-bit, and one 16-bit ALU: 43.75% of base size
- One 32-bit and one 16-bit ALU: 37.5% of base size
- Two 64-bit ALUs: 50% of base size

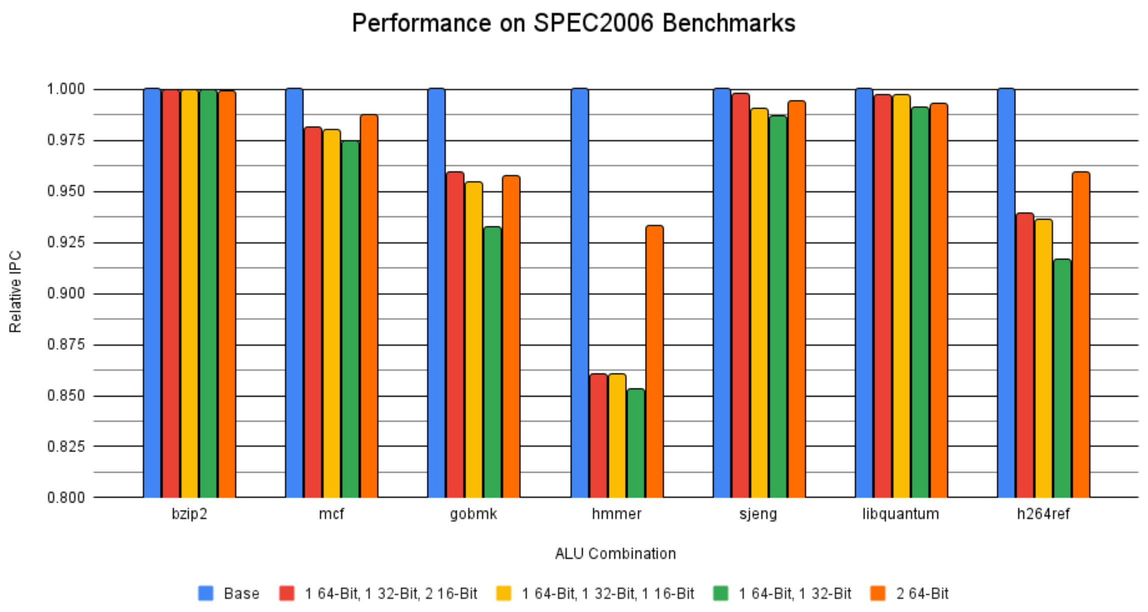


Figure 8. Performance of different ALU combinations

Each configuration was tested for 200M instructions against seven spec2006 benchmarks. IPCs were recorded for each combination of ALUs on each benchmark in Table 1 and plotted relative to the IPC of the base configuration for each test in Figure 8. Take note that the scale starts at 0.8 Relative IPC to more clearly display the difference between ALU configurations.

4 CONCLUSION

4.1 Summary

Benchmarking results show that most instructions are 16-bit or 32-bit with some workflows having more 64-bit instructions. What we see in the performance evaluation is when we downsize ALU area and number, performance tends to decrease the most for those 64-bit workflows. The hmmer benchmark saw the greatest performance decrease and this is in line with the width benchmarking which reveals it has the most 64-bit instructions out of all the benchmarks. Decreasing the number of 64-bit ALUs to 2 rather than the width of the existing CPUs retains the most performance on hmmer likely due to the larger number of 64-bit instructions.

The performance evaluation shows, unsurprisingly, that the base model has best performance in all cases. Performance falls off between about 2.5% and 5% for most benchmarks as we downsize the number and size of ALUs, but the difference in IPC between having one 64-bit, one 32-bit, and two 16-bit ALUs and having one 64-bit and one 32-bit ALU usually around 1% to 2%. With the latter case being sized at 37.5% compared to the size of the base case, this makes it have the best performance potential for decreased size in every benchmark. The case with two 64-bit ALUs has the same ALU size decrease as the case with 1 64-bit, 1 32-bit, and 2 16-bit ALUs at 50% while meeting the same performance in most cases and well exceeding performance in workflows with many 64-bit instructions.

The other factor to consider is the instruction width predictor. First, in any case besides only using 64-bit ALUs, we need an instruction width predictor. This gives the 2 64-bit ALU case a large advantage because it can save the space that would otherwise be needed for the width predictor. Secondly, performance will be further decreased on an instruction width mispredict which is not accounted in different ALU configuration simulations. This means that the 2 64-bit ALU case will have an even larger performance advantage compared to other ALU combinations.

We see either the one 64-bit and one 32-bit ALU case and the 2 64-bit ALU case as the most likely contenders for future research. If the need for reduced ALU size is large and the size of the instruction width predictor is relatively small compared to the ALUs, the one 64-bit and one 32-bit ALU case has the largest reduction in size for the smallest reduction in performance. Alternatively, if performance on 64-bit workflows is desired or the size of the instruction width predictor is relatively large compared to the ALUs, the two 64-bit ALU case is faster than the other combinations while still providing a hefty size reduction without the need for a width predictor.

4.2 Contributions

Name	Contribution
Lukas	- Compiled of spec-2006 binaries for RISC-V ISA using riscv-gnu-toolchain - Modified of Atomic CPU model in gem5 for ALU width benchmarking
Alish	- Ported CRIB CPU (provided REVCORE CPU) from an older version of gem5 to the latest gem5 - Implemented both width predictors in gem5 for CRIB CPU
Jake	- Modified dispatch logic in gem5 model of CRIB CPU to test all different ALU configurations - Interfaced width predictor with updated dispatch logic in gem5

Table 2. Contribution table in the order of project flow

4.3 Future Work

For all the experiments, we used only starting 200M instructions, but for the complete analysis, simpoint methodology could be used. In the case of the width predictor, we didn't do any experiments with the number of entries in the table. Although previous results suggest that there won't be much effect on accuracy with a few hundred entries, we would still do a more realistic study to confirm.

The next step would be to implement one of the ALU designs in RTL to gauge the true performance and area savings. We could then properly model the interconnect delays and possible speed savings with the smaller ALUs. An ideal candidate for this would be the design with two 64-bit ALUs as it does not require width prediction.

Another possible angle to consider is building units with different ALU functionality across the interconnect. For example, if we were to divide the fully featured ALU into one-half ALU that does shifting and logical operations and another half ALU that does adding and comparing, we could effectively double the number of CRIB units that can use the ALU for about the same area cost.

REFERENCES

[1] Erika Gunadi and Mikko H Lipasti. Crib: Consolidated rename, issue, and bypass. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 23–32, 2011.

[2] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[3] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[4] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[5] Risc-v gnu compiler toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain. Accessed: Nov 20, 2023.

[6] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[7] Gabriel H Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 395–405. IEEE, 2002.