UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MATRIX SCHEDULER FOR INSTRUCTION QUEUE IN OOO PROCESSORS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

MRUNAL KRISHNA SARMA ARYASOMAYAJULA
Norman, Oklahoma
2024

MATRIX SCHEDULER FOR INSTRUCTION QUEUE IN OOO PROCESSORS

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Ronald Barnes (Chair)

Dr. Clifford Fitzmorris

Dr. Richard Veras

Dedication

This work is dedicated to the pioneers of computer architecture, who laid the foundation for modern computing and on whose shoulders I stand today.

Acknowledgments

I want to thank all my professors, teachers, and, most importantly, my mom, dad, and family for their unwavering support throughout my life.

I'd like to especially mention and thank my thesis mentor, Dr. Ronald Barnes, for teaching me computer architecture and fueling my passion to pursue in the direction of this thesis.

I want to thank Dr. Clifford Fitzmorris for teaching me how to think like hardware and understand the workings of logic and digital design.

I'd also like to thank Dr. Richard Veras for inculcating a sense of sensitivity studies, teaching me how to approach any problem through the lens of a researcher, and supporting me since the beginning of my master's.

Lastly, I'd like to thank the fantastic open-source community of RISC-V and all the tools they built.

Table of Contents

List Of Tables

List Of Figures

Abstract

Multi-core architectures and multi-threaded programs dominate today's world. In this world, delivering the highest IPC is of the utmost importance. Many mobile devices, including desktops and servers, require high performance and minimal power consumption. This has led to the RISC architecture revolution, led by ARM with the first iPhone. Since then, both Moore's law and new architectures have contributed to ever-faster and more power-efficient chips. What once were in-order processors are now being replaced with out-of-order designs to improve the throughput of instructions at the same power.

The instruction queue is one of the most power-hungry regions of an out-of-order processor. An instruction queue, in its essence, is an entity that dispatches instructions to the execution units as soon as their input data is ready. Most modern processors use a CAM (content-addressable-memory) to realize the instruction queue. In a traditional IQ using CAM, the source arguments are constantly checked every cycle to see if they are ready, and the instruction is dispatched if all of the instruction's source arguments are ready. This approach is inherently power-hungry. The matrix instruction queue is an alternative to the traditional IQ and avoids CAM. The techniques that can be used to compress the matrix IQ to run efficiently, as well as the details of the IPC of this compressed matrix IQ and its resource usage, are presented in this thesis.

Chapter 1

Introduction

Processors can be classified into two categories based on how they execute instructions: In-Order and Out-Of-Order. Both of these approaches have their advantages and disadvantages. An In-Order processor is good for power efficiency. In-order processors are so power efficient that they are the only reason modern GPUs cram thousands of cores while running at a manageable thermal budget. Out-of-order processors, on the other hand, are really good at executing several instructions at a time and for sheer throughput. They are the reason behind modern high-performance computing and many technological feats.

An In-Order processor can be visualized as a set of stages, each performing part of the computation needed by an instruction. The first breakthrough in this design is the ability to execute multiple stages with different instructions at a time. A classical model of the In-Order processor comprises five stages: Fetch, Decode, Execute, Memory, and Write-Back. Each stage is responsible for a dedicated task: fetching an instruction, decoding an instruction, executing an instruction, writing to/reading from memory, and finally broadcasting the results, respectively. An in-order processor can be run incredibly efficiently through careful orchestration of these stages.

An Out-Of-Order processor is an extended version of the In-Order processor in which some of the stages work independently to some extent. The difference is that instructions need not follow the programmer's specified order of execution. Instructions can also be speculated in some cases, meaning that the processor thinks that

an instruction will be executed next and executes it without knowing whether it will actually be executed. Execution is rolled back if it's incorrectly speculated. Several different techniques and optimizations allow this out-of-order execution. Subsequent chapters explain a few of these techniques and the execution process in greater detail.

Programmers anticipate that their code will be executed statement by statement. Out-of-order execution disrupts this expectation. To deal with this, a new stage called commit is added to an out-of-order processor. This stage allows instructions that have completed their execution out of order to reorder and create the illusion of in-order execution using a specialized component called ROB (reorder buffer).

In essence, an out-of-order processor is an in-order front-end and an in-order back-end with an out-of-order engine in between. This out-of-order engine can be visualized as a queue in which instructions wait until their operands are ready and are dispatched from the queue when the operands are ready as first described by Tomasulo [15]. This instruction queue can be engineered in several ways. This thesis describes one such engineering technique in detail.

Modern processors utilize various techniques to enhance throughput, one of which is branch prediction. This technique anticipates the outcomes of conditional statements, such as those found in if-else constructs, thereby optimizing processing efficiency by predicting whether the if or else path will be taken.

As such, branch predictors used by modern processors are complex and highly accurate. They are accurate to the point that branch prediction is not only performed until the first branch but often two branches ahead of time Seznec et al. [14]. This increased accuracy enables computer architects to explore the design space of superscalar processors (processors that can execute more than one instruction at a time) with widths upwards of 6 and 8 that were earlier deemed prohibitively expensive.

The cost of increasing the superscalar width was always reversed mainly by the inability to efficiently dispatch the complete superscalar width of instructions and looking ahead to the entire width of the instruction queue. The techniques described in this thesis enable efficient instruction queues and longer instruction queue windows.

Chapter 2

Instruction Queue Design

A traditional instruction queue (non-value capturing; a queue where the contents of the source operands are not stored in the IQ) for a RISC-V (Waterman [16]) processor can be viewed as a table of at least 4 entries: *rob index*, *rs1*, *rs2*, *rs3*. rs1 & rs2 entries are mandatory for all the base RV32/64-I instruction sets (the set of instructions that make up a significant portion of most programs), and rs3 is needed for the F extension, which enables floating point operations. **rs** fields of the queue store the physical register index of the source operands. An instruction is considered ready only when all of its source operands have finished execution.

Consider the following instruction sequence:

Table 2.1: An instruction sequence

add	p2	p1	p1
add	p3	p2	p2
add	p4	p3	p3

This sequence in Table 2.1 has true dependencies; the third instruction can only be executed after the second, and the second can only be executed after the first. When this sequence is fed into a traditional instruction queue, each instruction will store its source registers *p1, p2 & p3* respectively. As soon as the first instruction completes its execution, it will generate a broadcast, which will be picked up by the second instruction and proceed to its execution; instruction 3 will still wait in the

queue for its source registers until instruction 2 completes execution and performs its broadcast. This process repeats until all of the instructions in the queue have completed execution. In a traditional instruction queue, instructions are allocated into their corresponding slots, with space for their source register operands. Every cycle, a broadcast is performed to wake up ready instructions.

A traditional instruction queue utilizes CAM (content addressable memory) to identify dependencies and dispatch the instructions whose dependencies have been resolved. CAM is a special hardware structure that stores data, compares its contents, and performs necessary actions if the comparison succeeds. Essentially, each instruction operand is compared against all the *wake-up registers* (registers whose values have been generated by their corresponding instructions recently), and the instruction should be woken up if its dependencies are resolved. This means each source register in the IQ will have as many comparators as the number of wake-up registers. Let's say that in a cycle, m instructions are retired from the execute stage and n from the memory stage. This means that $m + n$ wake-up ports are necessary for the instruction queue. This also means that $m + n$ comparators are necessary for each source register in the IQ. As I will show in chapter 3, this complexity is not needed and can be eliminated.

Alternatively, a matrix scheduler doesn't need comparators and relies on the fact that the index, which is nothing but a sparse representation of the dependency matrix, can be densified. This densification eliminates the need for comparators but at the cost of storing the full dense representation of the dependencies. The following chapters elaborate on these details.

Let's assume there are m entries in an instruction queue and n physical registers in the register file. A matrix scheduler allocates 1 bit per physical register per IQ entry, totaling for $m * n$ bits. So, matrix IQ can be viewed as a set of registers with as many entries as the number of IQ entries in the processor (m) and each entry being as wide as

the number of physical registers of the processor (n). Let's say an instruction reserved in slot y depends on the physical register x, then it would set x^{th} bit of y^{th} register. The entry is ready to be issued to functional units when all of its dependencies are resolved. In other words, the instruction in slot y is ready when the contents of the registers of the y^{th} register is 0. One might think that it's essentially comparing to 0. However, this is not the case. Comparisons to 0 are different from comparisons to 1. By OR-ing the contents of the registers, we can easily check if the value is 0. This implies that we will need m OR gates each one n bits wide. The designs generated by this approach are shown in table 6.2 and 6.3. More details are specified in the following chapters.

Let's consider a traditional IQ with four entries. For the instruction sequence in Table 2.1, the following are the states of the IQ for each clock.

Table 2.2: Traditional IQ in clock 0

valid	**op**	**rd**	**rs1**	**rs2**
1	add	p2	p1	p1
1	add	p3	p2	p2
1	add	p4	p3	p3
0	-	-	-	-

Now, assuming *p1* is ready, the instruction that generated the value of *p1* will generate a broadcast, and the comparators in each entry of the IQ will compare the broadcasted value to *rs1* and *rs2* and will dispatch the first instruction.

Once the first instruction completes its execution, it broadcasts physical register 2 *(p2)*. Comparators will resolve instruction 2, and the IQ dispatches it to the functional units.

Table 2.3: Traditional IQ in clock 1

valid	**op**	**rd**	**rs1**	**rs2**
0	-	-	-	-
1	add	p3	p2	p2
1	add	p4	p3	p3
0	-	-	-	-

Table 2.4: Traditional IQ in clock 2

valid	**op**	**rd**	**rs1**	**rs2**
0	-	-	-	-
0	-	-	-	-
1	add	p4	p3	p3
0	-	-	-	-

However, it should be noted that in each of these clocks, all the *rs* values must be compared for all of the IQ entries.

A similar table for the matrix IQ is shown in Table 2.5.

Table 2.5: Matrix IQ in clock 0

valid	**op**	**rd**	**...**	**p1**	**p2**	**p3**	**...**
1	add	p2	...	1	0	0	...
1	add	p3	...	0	1	0	...
1	add	p4	...	0	0	1	...
0	-	-	...	-	-	-	...

Upon receiving a broadcast for physical register 1, p1 is set to 0 in all of the entries in the IQ. The first instruction is dispatched as the dependencies are wholly resolved to 0. The IQ in clock 1 is shown in Table 2.6. Note that the instruction is deemed

to be ready once all of its dependencies are resolved, meaning that the contents of the entry will be equal to 0.

Table 2.6: Matrix IQ in clock 1

valid	**op**	**rd**	**...**	**p1**	**p2**	**p3**	**...**
0	-	-	...	0	0	0	...
1	add	p3	...	0	1	0	...
1	add	p4	...	0	0	1	...
0	-	-	...	-	-	-	...

Similarly, in clock cycle 2, a broadcast is received for physical register 2, setting p2 to 0 in all entries. As a result, instruction 2 wakes up and is dispatched to the functional units during this cycle, as shown in Table 2.7.

Table 2.7: Matrix IQ in clock 2

valid	**op**	**rd**	**...**	**p1**	**p2**	**p3**	**...**
0	-	-	...	0	0	0	...
0	-	-	...	0	0	0	...
1	add	p4	...	0	0	1	...
0	-	-	...	-	-	-	...

In every clock, all the values in a row are ORed together, which results 0 in the case of an instruction whose dependent registers are all resolved. In such a case, it means that the instruction is ready to dispatch.

Chapter 3

Methodology

The first step in benchmarking the matrix IQ approach is through clock-level simulations. For this, Gem5 [4, 10, 12] was the obvious choice. Gem5 is a simulator for computer architectures that offers flexibility, where we can model any ISA with configurable parameters and swap out various modules for different ones. The IQ in Gem5 has been modeled to benchmark various configurations and gain insights into the system's behavior. The out-of-order processor of gem5 is built on the architecture of Alpha 21264 [11], which is the prototypical architecture for out-of-order processors.

One of the matrix scheduler's main requirements is not having any additional IPC penalty than the traditional scheduler. As described earlier, this is possible in the design's essence. The possibility at the architecture level will now be determined through clock-level modeling.

Gem5 is built hierarchically, with one component on top of another, and can be configured for various processors and several different architectures. So, the first thing to do is to decide on the base architecture of the processor. The sizes of various components of the processor are chosen so that they reflect real-world metrics. A superscalar issue width of 8 can easily be simulated but is uncommon in the real world due to the diminishing returns over 4-wide or 6-wide superscalars. Similarly, other metrics are chosen to reflect the real-world values seen in various processors. Some of the configuration values are shown in Table 3.1.

Table 3.1: Gem5 Out-of-Order processor configuration

Config & Value					
Superscalar width	4	L1 iCache size	16 KiB	L2 size	256 KiB
# Phy. Registers	128	L1 dCache size	64 KiB		
# IQ entries	64	L1 associativity	2	L2 associativity	8
# ROB entries	192	L1 latency	2 cycles	L2 latency	20 cycles
# LQ & SQ entries	32	L1 mshr	4	L2 mshr	20

The next step is to model the matrix IQ in Gem5 and verify the performance. The first few tests are performed using a bubble sort algorithm as shown in Listing 3.1. This allows for fast prototyping while getting the results with no downtime. The benchmarks in the following chapters are more rigorous and accurate for real-world workloads.

```
static void sort(int len, long *values) {
  for (int i = 0; i < len; i++) {
    for (int j = i; j < len; j++) {
      if (values[i] > values[j]) {
        long temp = values[i];
        values[i] = values[j];
        values[j] = temp;
      }
    }
  }
}
...
```

Listing 3.1: Bubble sort

The Gem5 code for the matrix IQ is shown in Listing 3.2. The performance of the matrix scheduler is the same as that of the traditional IQ for a similar size, as

shown in Figure 3.1. We can also see that the IPC depends on the #IQ entries, as the smaller instruction queues fill up quickly, resulting in a loss of IPC. Similarly, we see diminishing returns as the instruction queue size increases beyond 16.

```
#ifndef __CPU_O3_DEP_GRAPH_REG_HH__
#define __CPU_O3_DEP_GRAPH_REG_HH__

#include "cpu/o3/comm.hh"

namespace gem5 {
namespace o3 {
template <class DynInstPtr>
class DependencyGraphReg {
  public:
    ...
    // actual instruction queue
    std::vector<DynInstPtr> instructions;

    // The map of dependencies
    // Tracking the #operands pointing to a register
    // instead of: if an entry depends an entry or not
    std::vector<std::vector<int>> reservationStationMap;
    ...
};

template <class DynInstPtr>
void DependencyGraphReg<DynInstPtr>::insert(RegIndex reg_idx, const
   DynInstPtr &new_inst) {
    ... find the spot for the new instruction
    assert(new_inst_idx != -1);
    instructions[new_inst_idx] = new_inst;
    reservationStationMap[reg_idx][new_inst_idx]++;
```

```
}
template <class DynInstPtr>
DynInstPtr DependencyGraphReg<DynInstPtr>::pop(RegIndex reg_idx) {
    DynInstPtr inst = NULL;
    for (int inst_idx = 0; inst_idx < numReservationStations;
   inst_idx++) {
        if (0 != reservationStationMap[reg_idx][inst_idx]) {
            inst = instructions[inst_idx];
            reservationStationMap[reg_idx][inst_idx]--;
            break;
        }
    }
    return inst;
}
}
}
#endif // __CPU_O3_DEP_GRAPH_REG_HH__
```

Listing 3.2: Matrix IQ in Gem5

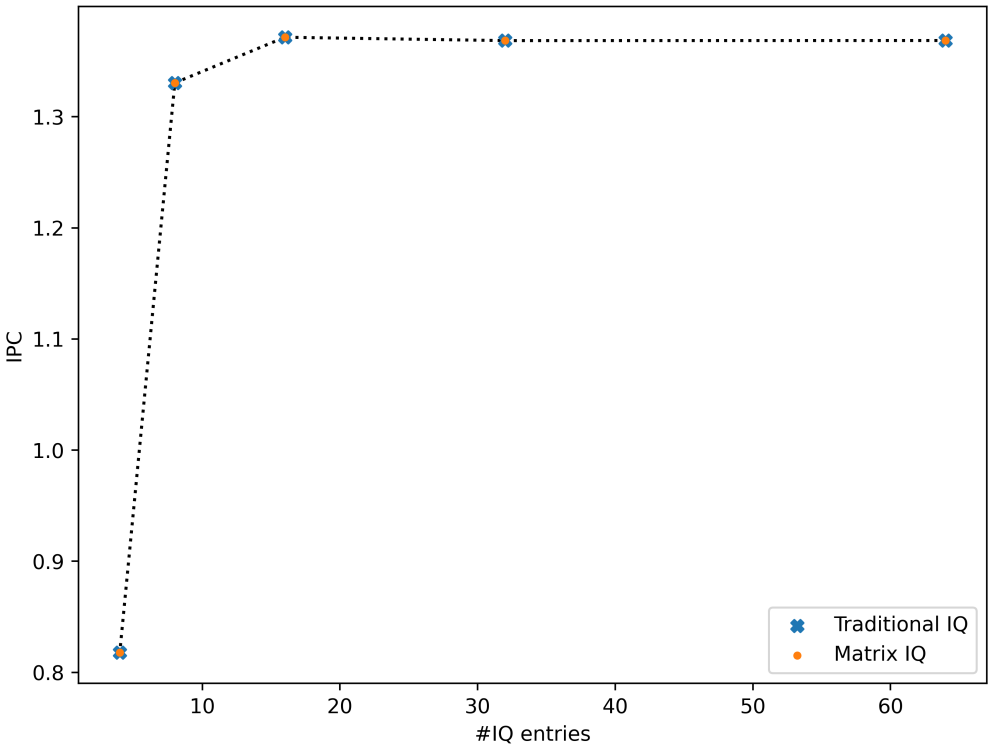


Figure 3.1: IPC of Traditional IQ & Matrix IQ

Chapter 4

ISOD

One of the significant challenges with the matrix scheduler is the space and area it might occupy on the die for a processor with a significant number of IQ entries. For an IQ with 64 entries @ 3 source registers (rs) per entry and 128 physical registers (pr). The number of bits of information stored in IQ is shown in Table 4.1.

$$\begin{aligned} \#\text{bits}_{\text{ traditional IQ}} &= \#rs * \#iq * \log_2(\#pr) \\ &= 3 * 64 * 7 = 1,344 \text{ bits} \\ \#\text{ bits}_{\text{ matrix IQ}} &= \#iq * \#pr \\ &= 64 * 128 = 8,192 \text{ bits} \end{aligned}$$

Table 4.1: Bits of matrix IQ

One of the first things to be noticed is that the number of bits in the matrix scheduler is independent of the number of source registers. Thus, the matrix scheduler scales better when the instruction is governed by more than a couple of registers. However, after substituting the values, we see a different picture. The matrix scheduler needs ˜6x more bits than the traditional approach.

ISOD (Inter-Source Operand Distance) is the numerical difference between the max source operand index and min source operand index considering the operands that are not ready by the time of issue. ISOD = -1 is an extrapolated case where all of the source operands of an instruction are ready by the time of issue.

The problem is linear scaling with respect to the number of physical registers; traditional IQs scale logarithmically, enabling them to use significantly fewer bits than matrix IQs. If we look closely, we only need to represent the registers in the range of min(rs) until max(rs). This means we can shrink the matrix IQ if we analyze the "inter source operand distance (ISOD)" and tune the IQ for an average program. We can fall back to a more traditional IQ, representing the whole space in situations where the smaller IQ cannot represent the necessary dependencies.

In order to shrink the instruction queue, we need to understand the program's dynamic instructions distribution. For a RISC-V processor, the compiler-generated instructions almost always span the complete range of opcodes. However, during run-time, not all the instructions are executed all the time. This leads to an interesting case where a few instructions are executed much more often than others.

For example, let's consider a sorting program as shown in Listing 4.1. The sets of instructions executed unconditionally are the loads and the branch. On the other hand, stores are only executed if the elements that are being sorted need to be swapped.

This means that a sorting program would need n^2 loads but only n stores. It is to be noted that loads take only one source register, whereas stores take two source registers. Similarly, by the time store instruction is executed in this case, the address must have been already calculated, as both the loads that are ahead in the program order use it, and the address update logic is simple enough to be executed in one cycle. This, in turn, means that even though the store depends on two registers, one is already ready, effectively making the store a single source operand instruction.

```
.begin:
    ld   a4,0(s2)
    ld   a3,0(a5)
    ble  a4,a3,.addrUpdate
    sd   a3,0(s2)
```

```
    sd   a4,0(a5)
.addrUpdate:
    ...
    bne  a2,a1,.begin
```

Listing 4.1: Assembly Code for bubble sort

Similarly, it is to be noted that instructions like: *jump*, and *auipc* don't depend on any registers (*pc* (program counter) being the implicit register) and can be executed right away.

With this in mind, the first experiment conducted is an analysis of the number of instructions a program has executed, the number of source operands each instruction has, and the breakdown of instructions in these programs using the gem5 simulator on benchmarks from **coremark-pro & SPEC CPU 2017** (coremark-pro benchmarks are run till completion, SPEC benchmarks are run for the first 2 billion instructions).

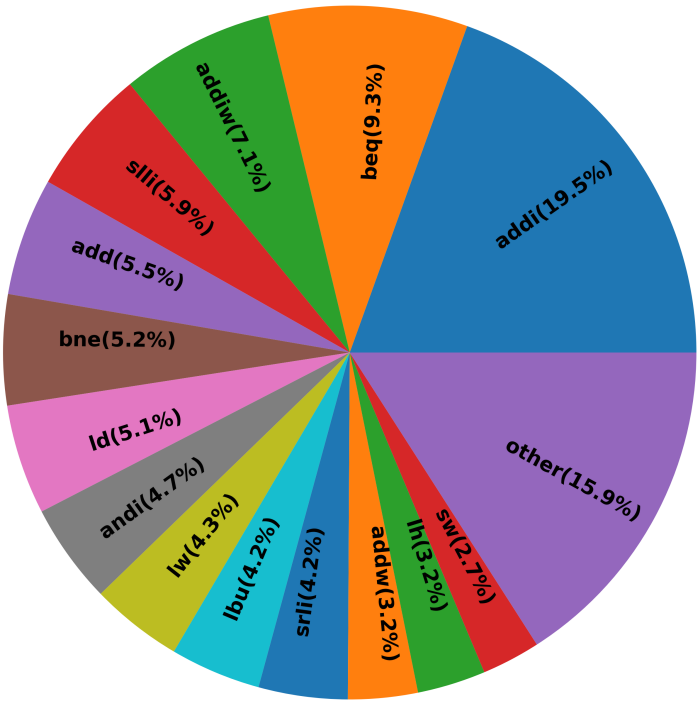


Figure 4.1: Distribution of instructions for the core benchmark of coremark-pro

From Figure 4.1, it can be noted that instructions like: addi, addiw, slli, ld, andi, lw, lbu, srli, lh, which make up the majority of the instructions, are all immediate type and only depend on one register. This implies that an IQ's second and third source operands are unused for these instructions.

Similarly, running the experiment for other benchmarks in this suite shows a similar behavior, wherein a significant portion of the dynamic instructions executed are I-type and only need one source register for their computation, as shown in Figures 4.2.

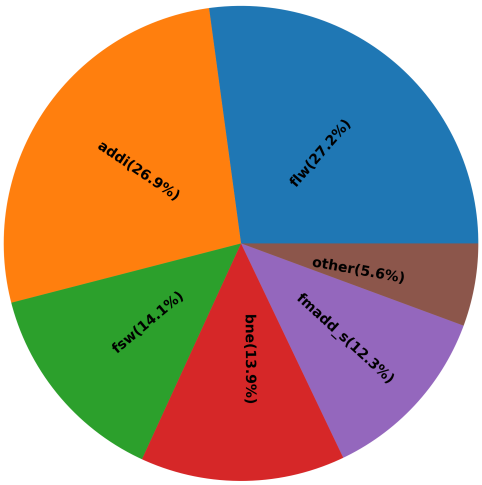
Consistently across all these benchmarks, the number of I-type instructions that need only one source register represent considerable portions (˜50%) of all the instructions executed. This leaves the room for designing heterogeneous instruction queues composed of two kinds: one made for instructions with a single source operand and one made for instructions with multiple source operands.

```
.begin:
    addi sp, sp(-32)
    sd  ra, 28(sp)
    ...
.loop:
    ld t0, 0(sp)
    # various other operations using sp
```

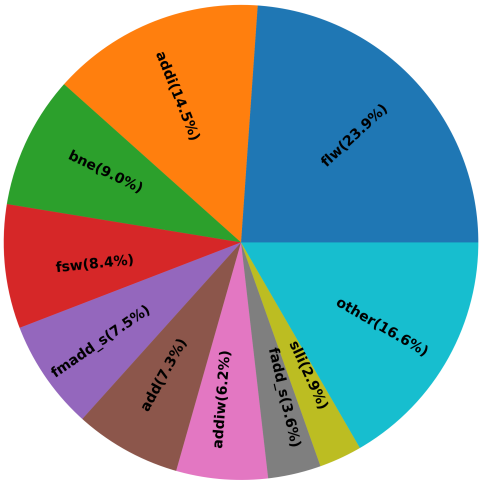
Listing 4.2: Assembly Code for a program using stack

The instruction distribution metric, while helpful, only paints half of the picture. For example, consider the following program in Listing 4.2. In this, we see that the stack address is updated once before the beginning of the procedure call, and that value is retained during the loop and throughout the execution of the procedure. This means that the register is ready by the time most of the instructions in the function

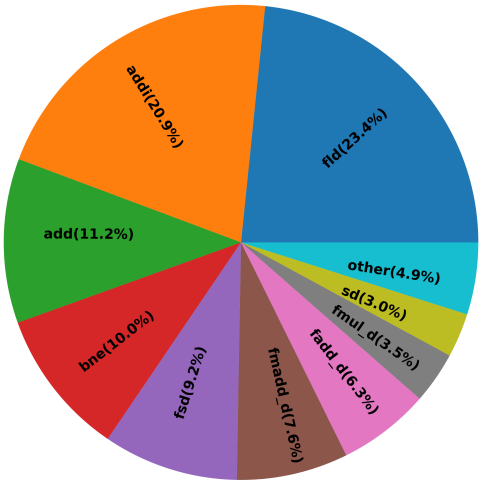
(a) Rose



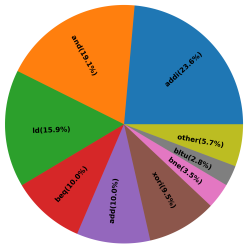
(b) Linear Algebra



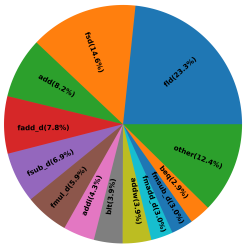
(c) Loops benchmark



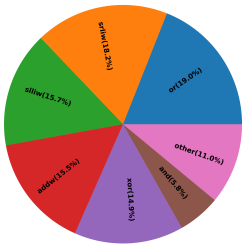
(d) Neural Network



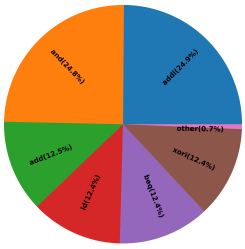
(e) Parser



(f) Radix



(g) SHA



(h) zip

Figure 4.2: Instruction distribution of various benchmarks in coremark-pro

are executed. So, the metric to be analyzed is how many instructions have their source operands ready by the time the instruction is being put into the IQ.

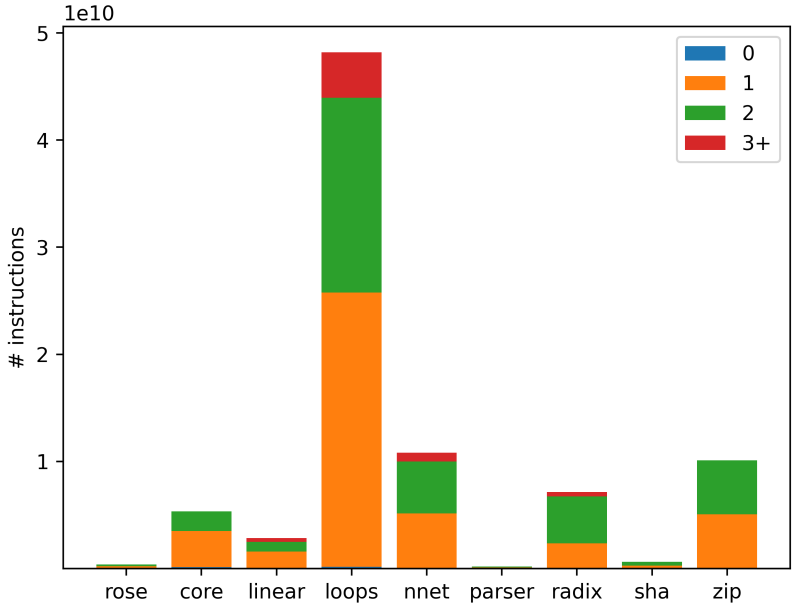


Figure 4.3: # of registers used by instructions

From Figures 4.3 and 4.4, we see that a significant portion of the instructions depend on one register only and few instructions dependent on two registers and even fewer on three or more (NOTE: three source operand instructions are most reserved for floating point operations like *fmadd* which need two registers for multiplication and one for addition for a total of three. Benchmarks in coremark-pro utilizing such instruction are: loops, neural networks, and radix. The instruction distribution of these benchmarks can also be seen in the previous Figure 4.2).

Instead of looking at the instructions from the perspective of the number of registers ready by the time of execution, we can look at the ISOD, which is the distance between source registers, we get better insights into the distribution of the instructions.

Figure 4.4: # of registers used by instructions (Normalized)

From Figure 4.5, we see that the instructions depending on a single register or no registers are dominating. This also means that the ISOD between the registers is 0. This is advantageous because a matrix IQ with ISOD = 0 should work for most instructions.

Similarly, to minimize the ISOD, a designer can prioritize selecting registers on the basis of the register index. Simply put, if the allocation of the registers is not from a free buffer but from a sorted free buffer.

In Figures 4.5 4.6 & 4.7, we see that floating-point benchmarks like: linear algebra and neural network, we can see that the ISOD for most of the instructions is either -1 or 0. However, we also see a rise in the ISOD after 128, indicating the interactions between integer and floating point registers. In this case, the ABI of the processor is *lp64*, which means that integer values can be forwarded to floating-point values through

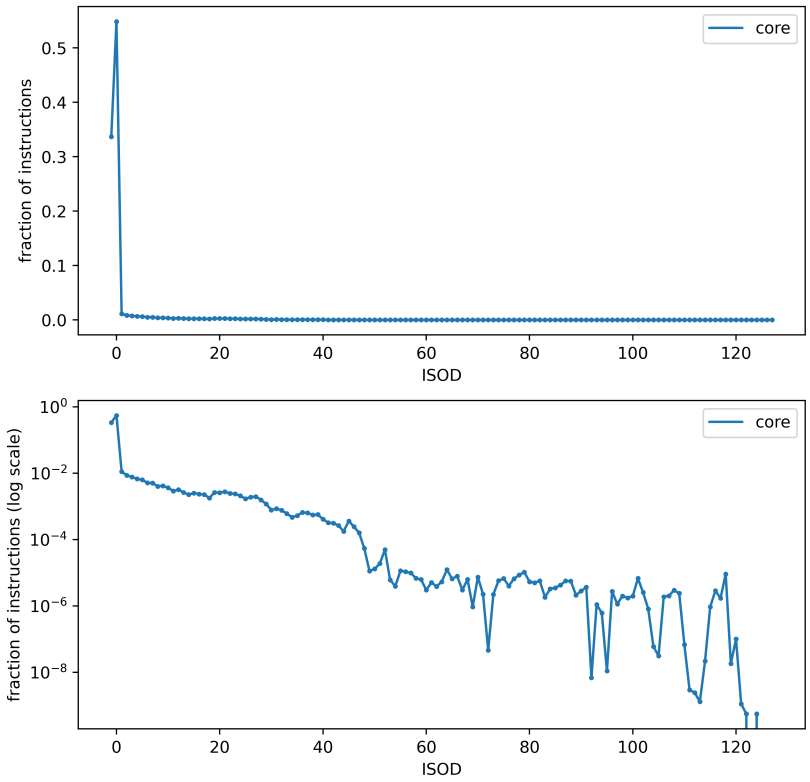
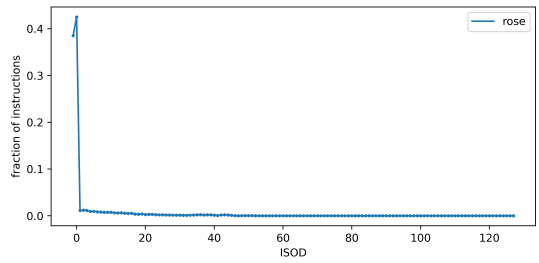
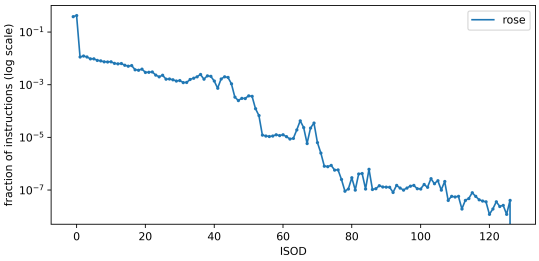
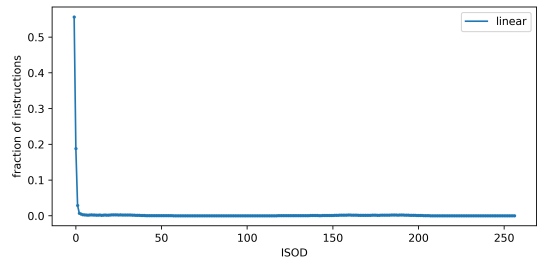


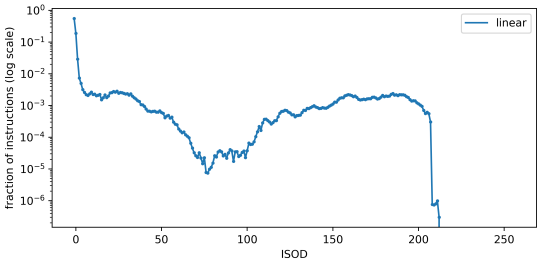
Figure 4.5: ISOD for the core benchmark of coremark-pro (with priority)



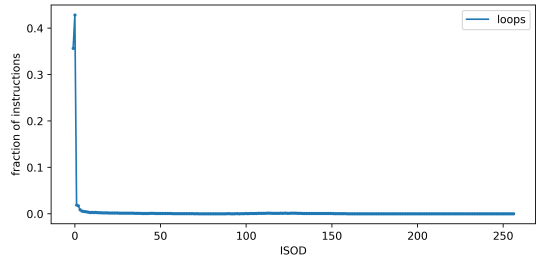


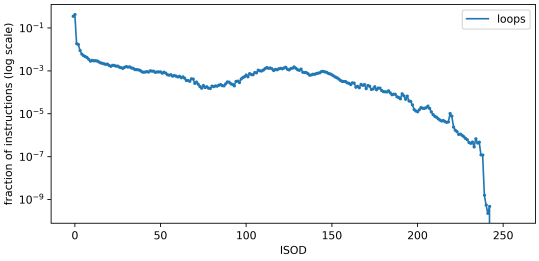
(a) Rose



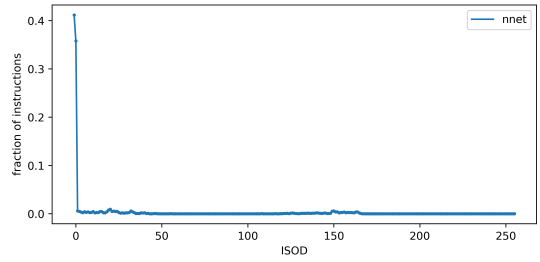


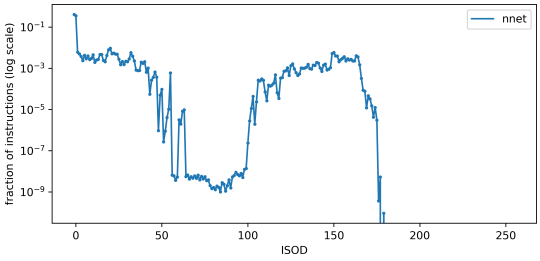
(b) Linear Algebra





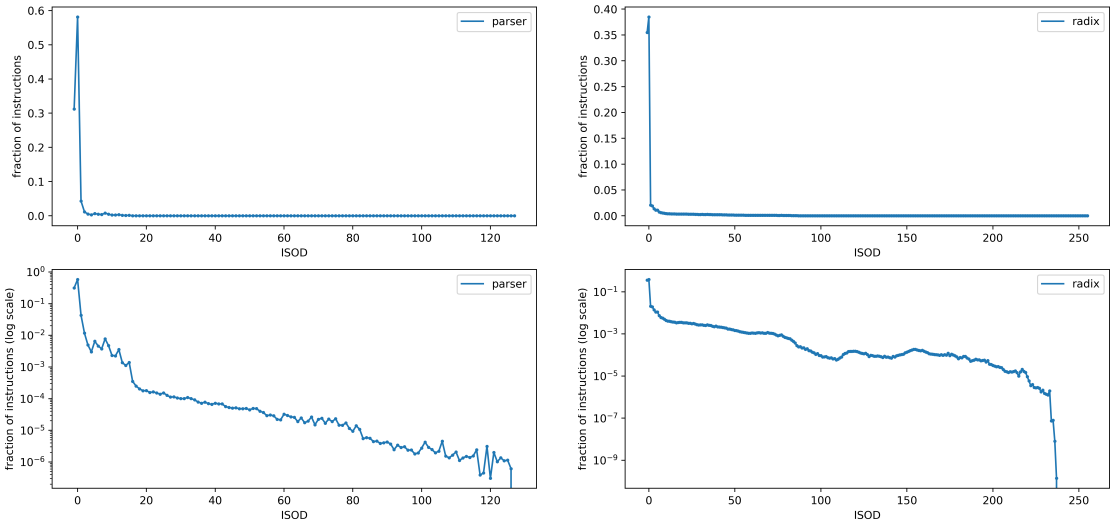
(c) Loops benchmark





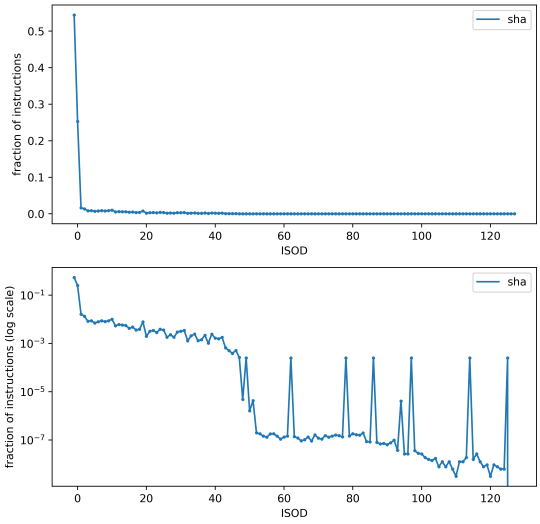
(d) Neural Network

Figure 4.6: ISOD of other benchmarks in coremark-pro

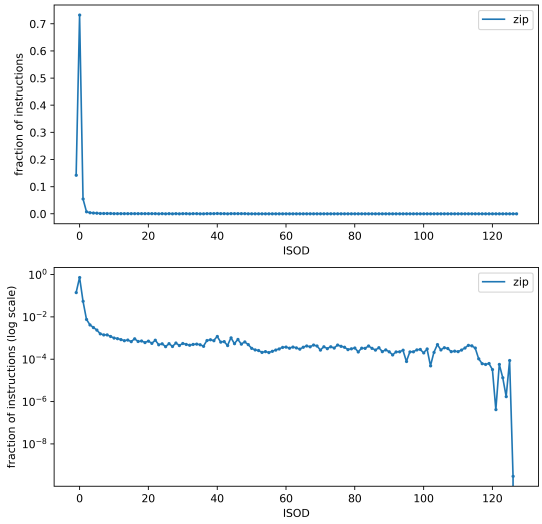


(a) Parser

(b) Radix



(c) SHA



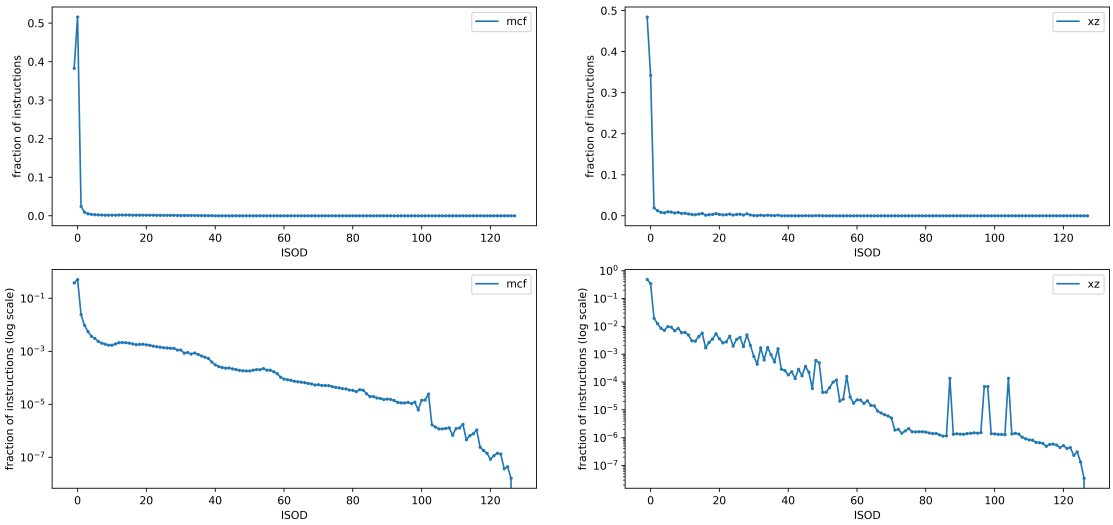
(d) zip

Figure 4.7: ISOD of the rest of the benchmarks in coremark-pro

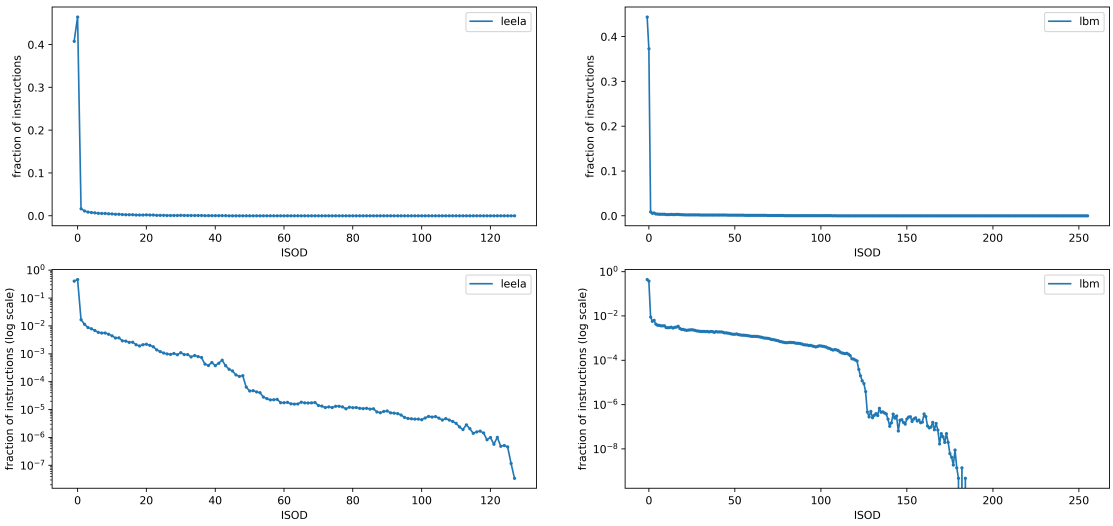
registers. The floating point registers are indexed after the integer registers, resulting in spikes in the ISOD charts beyond 128.

Similar to the ISOD metrics of coremark-pro, SPEC CPU 2017 benchmarks also show the same behavior where ISOD of -1 and 0 contribute a significant portion of the instructions, as shown in Figure 4.8. This indicates that the ISOD behavior is consistent across a variety of real-world programs.

However, it should be noted that the majority of these instructions are dependent on one register or none and are ready by the time of execution. So, having a priority in the register allocation process doesn't affect these instructions, and the designer might choose to ignore them and still work with heterogeneous instruction queues. The design of heterogeneous instruction queues and advantages are discussed in Chapter 5.



(a) MCF (b) XZ



(c) Leela (d) LBM

Figure 4.8: ISOD of select benchmarks in spec

Chapter 5

Heterogeneous IQ

Given that most instructions depend on only one register or none at all, a logical step is to create a heterogeneous IQ composed of two instruction queues, one dedicated to instructions with one or fewer registers and one for instructions with two or more registers, and determine the performance.

Modern processors often categorize instructions into separate instruction queues. For example, we see floating-point instructions allocated to a floating-point instruction queue and memory instructions allocated to their dedicated instruction queue. The emphasis is that the instructions in integer IQ need high performance and can benefit from additional circuitry for forwarding; on the other hand, forwarding and features similar to forwarding don't benefit floating-point instructions as much because the minimum number of cycles needed for operations like: *fadd* & *fmul* are a few cycles in most processors. Thus, a dedicated IQ for these instructions can simplify logic and reduce power. It is important to note that these instruction queues operate on their dedicated functional units and rarely share functional units. That is, the integer IQ is dedicated to integer functional units, and the floating-point IQ is dedicated to floating-point functional units. This kind of segregation allows the design to be simple and efficient.

On the other hand, a heterogeneous IQ would require functional units to be shared. This sharing could be performed through a ready buffer between the instruction queues and functional units. This buffer also solves another important problem of age ordering,

as described in the following sections. One could design a ready buffer to reshuffle the instructions, allowing the instructions to be reordered with respect to age, allowing higher efficiency in scheduling as shown in Figure 5.1.

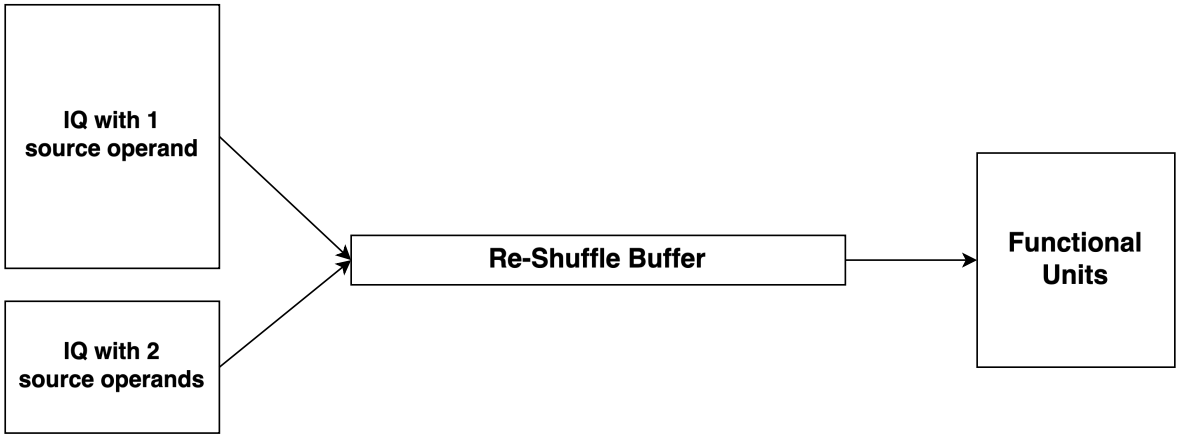
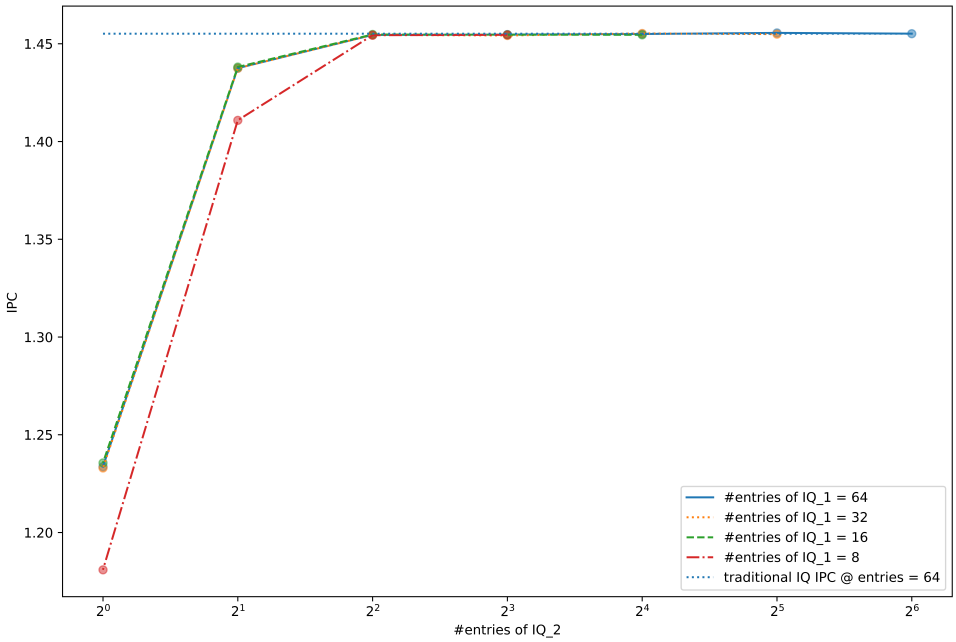


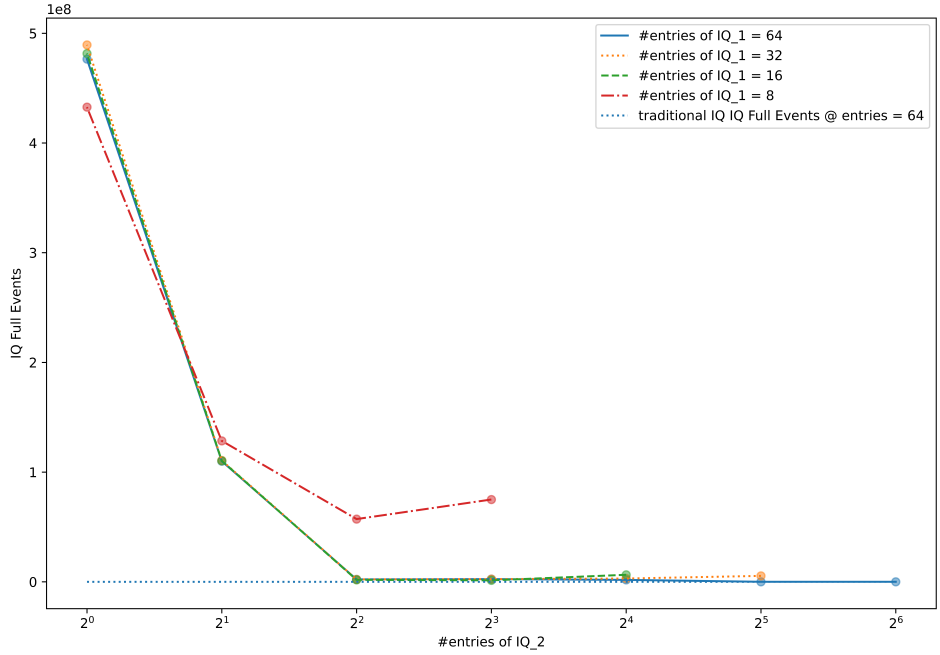
Figure 5.1: Heterogeneous IQ with reshuffling

To test the performance of the heterogeneous instruction queue, the benchmarks Core and Neural Networks were chosen. These represent a good balance between compute, memory, and contain a good distribution of integer operations and floating point operations. Figures 5.2 and 5.3 show that the heterogeneous IQ performs on par with the traditional IQ. As we reduce the number of IQ entries, there is a reduction in IPC as the IQ will not be able to look ahead into the instruction window and fails to extract instruction level parallelism after a point. Similarly, we also see a sharp spike in the number of IQ full events, as the front end fails to allocate the necessary space for instructions in the IQ as expected.

From this point, IQ_1 represents the instruction queue with only one source operand, meaning that it can only resolve instructions that need one source operand (e.g., immediate type instructions and instructions whose operands are ready by the time of

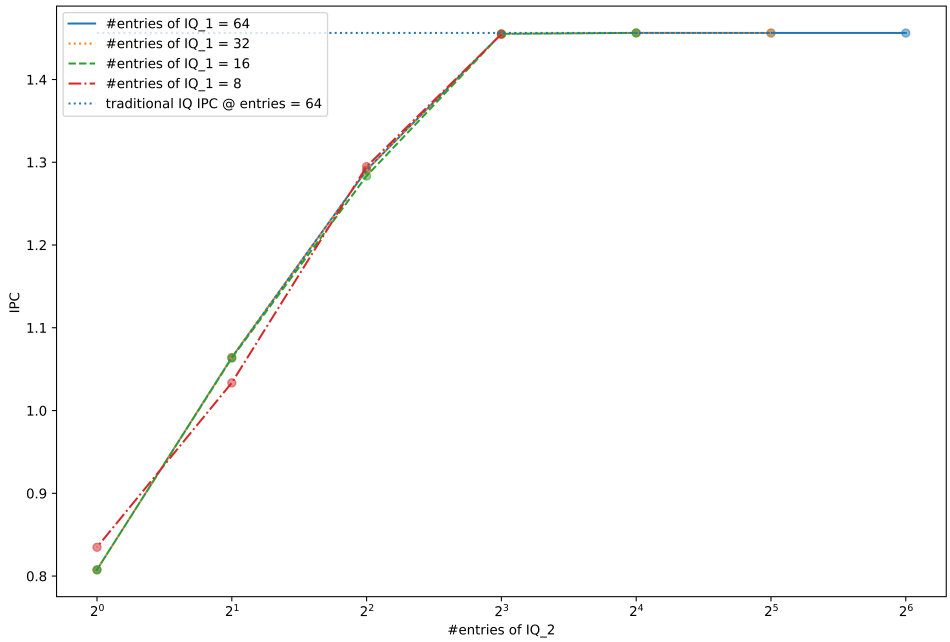


(a) IPC

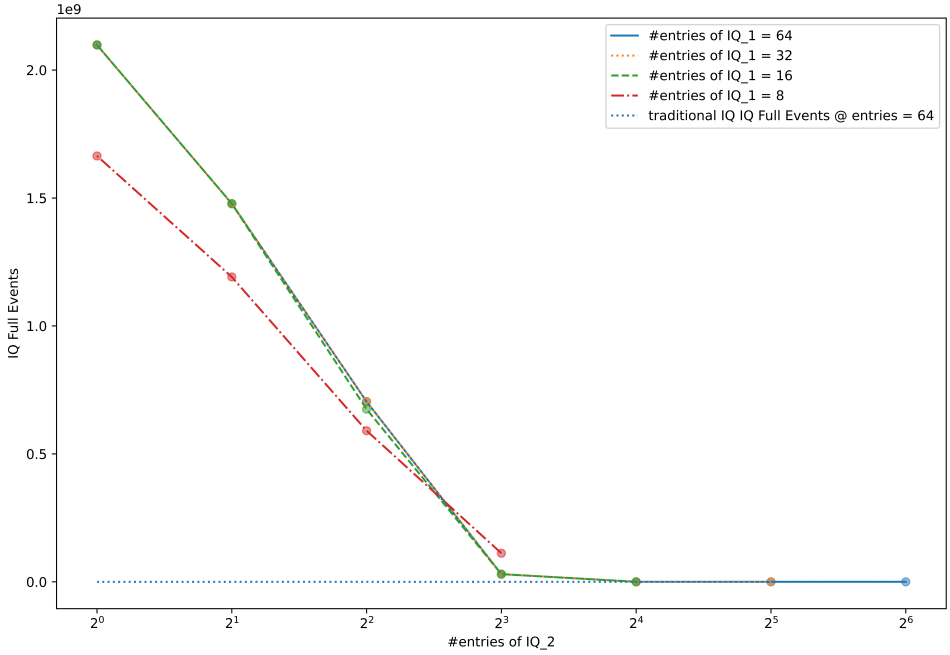


(b) IQ Full Events

Figure 5.2: IPC & # IQ full events of the Core benchmark with heterogeneous IQs of varying sizes



(a) IPC



(b) IQ Full Events

Figure 5.3: IPC & # IQ full events of the Neural Networks benchmark with heterogeneous IQs of varying sizes

dispatch) or fewer. Similarly, IQ_2 represents the instruction queue with two or more source operands.

We see that the heterogeneous IQ performs similarly to the traditional IQ but sometimes might fall short of the performance of the traditional IQ and has a slightly irregular performance. This is probably because the age ordering mechanism in the individual IQ schedules the oldest instructions first. However, the same kind of ordering is not present across the IQs. Let's consider the example in Table 5.1. Instructions of IQ_2 and IQ_1 are both issued according to age. If instructions of IQ_2 are prioritized over instructions of IQ_1 or vice-versa, the age ordering scheme will be broken subtly. This forces the instructions to be executed in a sub-optimal order, which can be fixed easily by incorporating ordering logic into the ready buffer.

Table 5.1: Missing Age Ordering across IQs

IQ_1		**IQ_2**	
instruction	*age*	*instruction*	*age*
add p2 p1 p1	1	add p3 p0 p1	2
addi p4 p1 1	3	sub p5 p0 p1	4
Upon p1 register getting ready			
Assuming IQ_1 instructions are prioritized over IQ_2			
Ready buffer contents			
add p2 p1 p1	addi p4 p1 1	add p3 p0 p1	sub p5 p0 p1
Ideal order			
add p2 p1 p1	add p3 p0 p1	addi p4 p1 1	sub p5 p0 p1

Now that we've established that a heterogeneous instruction queue can be as efficient as traditional IQ, we can focus on shrinking the matrix IQ. The first idea was to store the offset of min(rs) and then represent the $max(rs) - min(rs)$ in bits as a sequence. However, this approach is not as efficient as we'd like. It requires all the

wake-up registers to be connected to all the IQ entries at all possible offsets. This approach complicates the circuitry as shown in Figure 5.4.

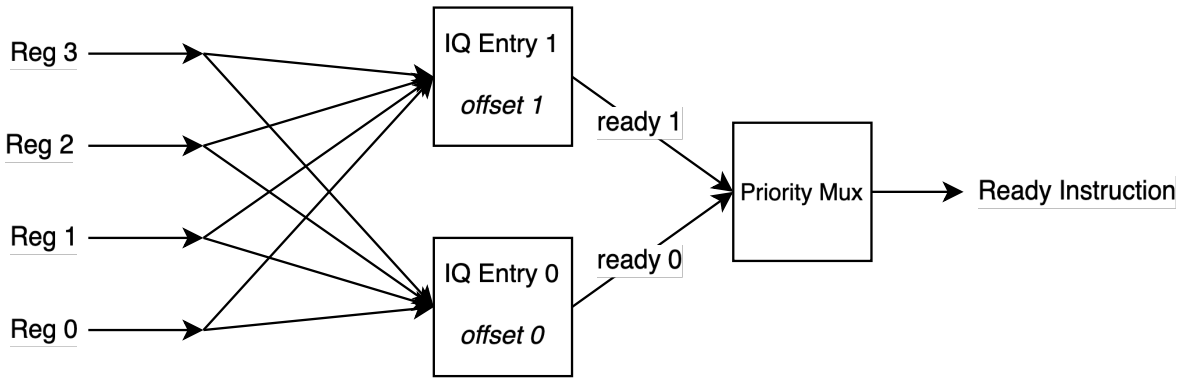


Figure 5.4: Shrunk IQ with base offset and adder

A mux-ed approach makes it much more efficient in terms of complexity. In this approach, instead of storing the min(rs) as the offset, we store the aligned index of the value and offset the necessary values from that index; then, a mux can be used to forward all the dependencies starting at that index. Similar approaches are also seen in several other places like: memory alignment, TLB alignment, etc. The circuit is shown in Figure 5.5. The first mux, which is connected to *IQ entry 0*, will forward both *reg 0 & 1* if the *offset 0* is 0. But if *offset 0* is 2, it'd forward *reg 2 & 3*. Each entry would then compare the values as necessary and mark the corresponding instruction as ready accordingly.

However, we would still incur an IPC penalty due to the missing age ordering between the instructions. To solve this, one would need to re-order the instructions in flight using the aforementioned reshuffle buffer. A reshuffle buffer is nothing but a sorting network implemented in hardware; let's assume that both IQ_1 and IQ_2 hold 4 instructions, each ready to dispatch at the end of the cycle. It means that there are 8 instructions that need to be ordered. However, ordering is maintained within the first

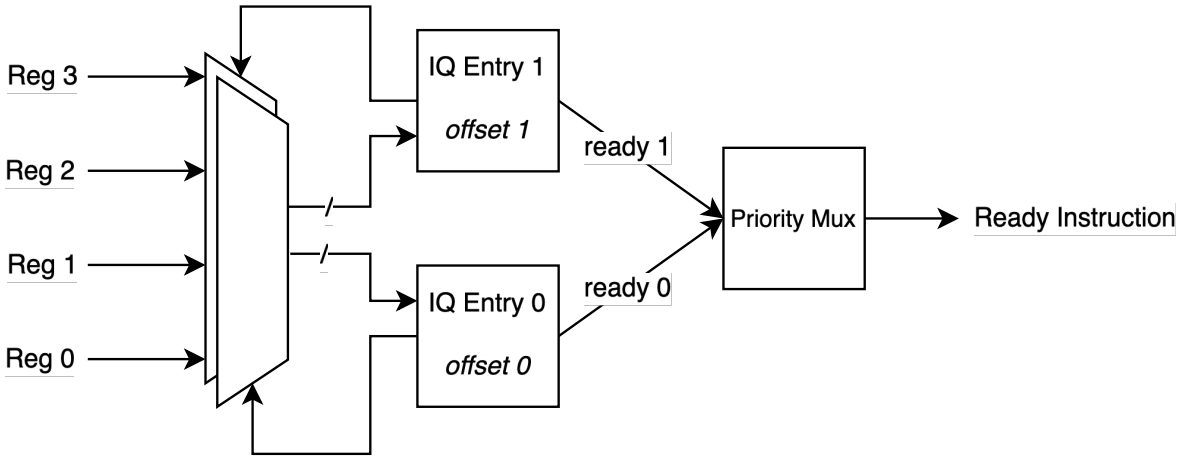


Figure 5.5: Shrunk IQ with base offset Mux for selection

4 instructions and the next 4. Thus, the final sorting network would resemble Figure 5.6. Note that the sorting buffer would only require **9** $log_2(age_{max})bit$ **comparators**, which pales in comparison to the total number of comparators we would use otherwise. Also, notice that this network is much simpler than a sorting network that sorts 8 unsorted elements.

Table 5.2 illustrates the shrunken IQ for max ISOD = 2 and its entries for the instruction sequence in Table 2.1. From this point onwards, the offset will be called the sparse index, and bits representing the dependencies will be called the dense index. In contrast to Table 2.5, we need not represent all the entries of all the physical registers here, we only represent two entries in this case.

Table 5.2: Shrunken IQ

valid	**op**	**rd**	**off**	**+0**	**+1**
1	add	p2	0	1	0
1	add	p3	2	1	0
1	add	p4	2	0	1
0	-	-	-	-	-

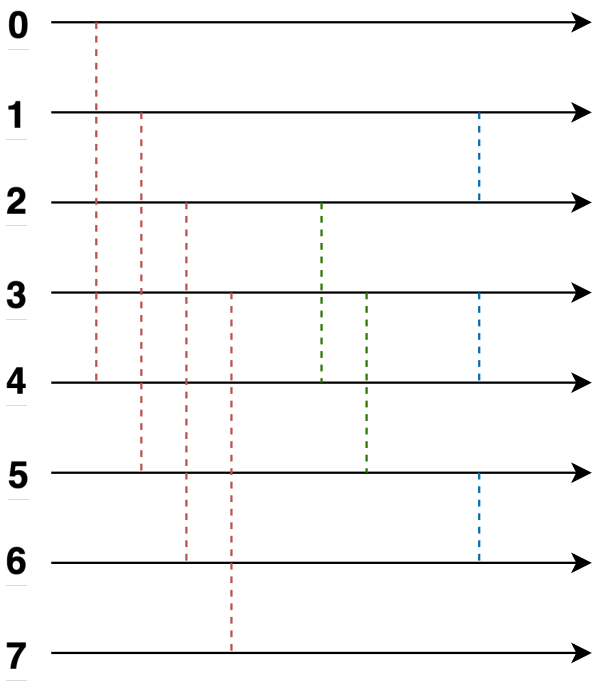


Figure 5.6: Sorting Network for Reshuffle buffer

In other words, to represent n physical registers, which need $\log_2(n)$ bits to index, an index can also be generated using $\log_2(n) - \log_2(d)$ bits for the mux (sparse) indexing and d bits for the representation of the dense indices.

Checking back on Figures 4.5, 4.6 & 4.7, we can see that about 80% of instructions can easily be handled using IQ of size 1. So, going back to the design boards and modeling the IQ in GEM5 with heterogeneous IQs composed of IQ_1 made of matrix IQ with *max(ISOD) = [1, 2, 4, 8, 16]* and IQ_2 made of a traditional instruction queue, we see the following IPC charts shown in Figures 5.7 & 5.8.

We see that the heterogeneous IQ composed of the matrix IQ and traditional IQ still performs similarly to the full traditional IQ. The small changes in IPC essentially boil down to the scheduling differences and differences in issue and dispatch.

Heterogeneous instruction queues can also be applied to traditional IQ. One can model a heterogeneous traditional RISCV IQ with 3 different variations:

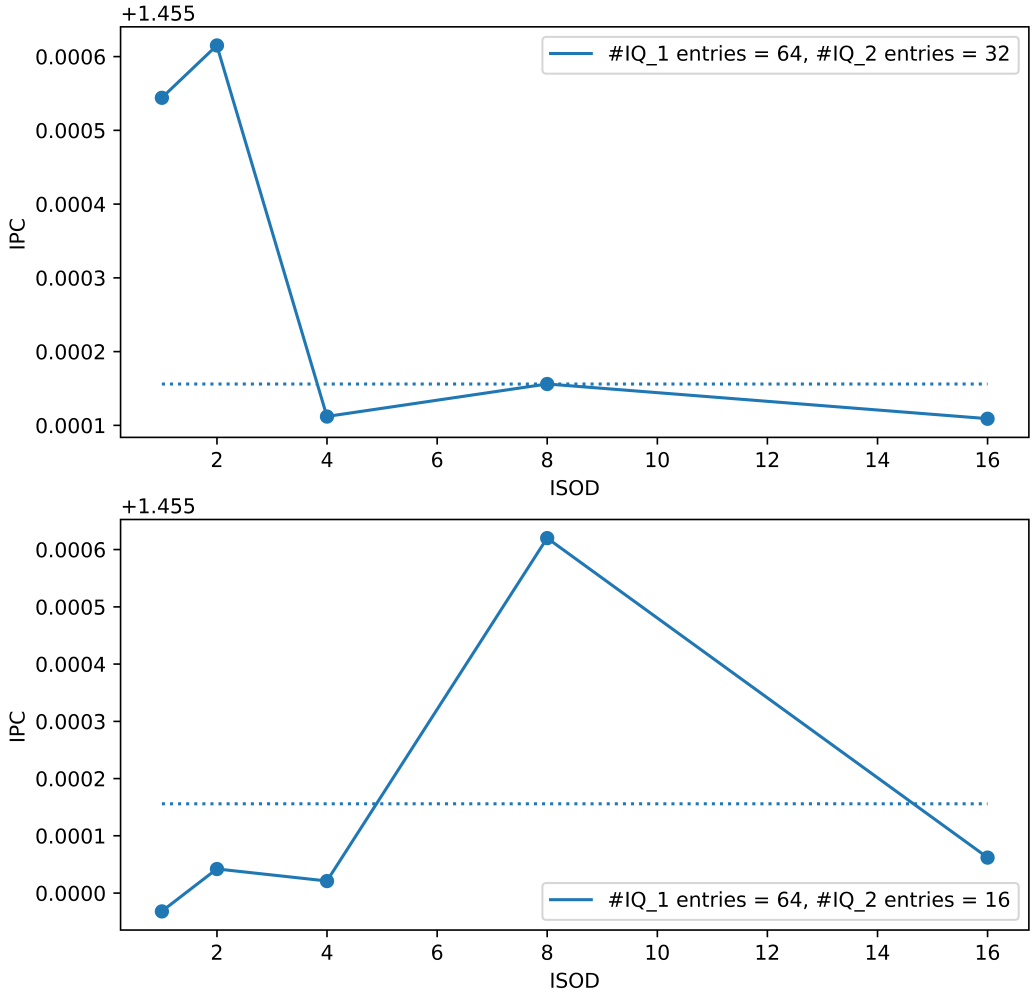


Figure 5.7: Heterogeneous IQ (matrix and traditional) IPC for Core benchmark

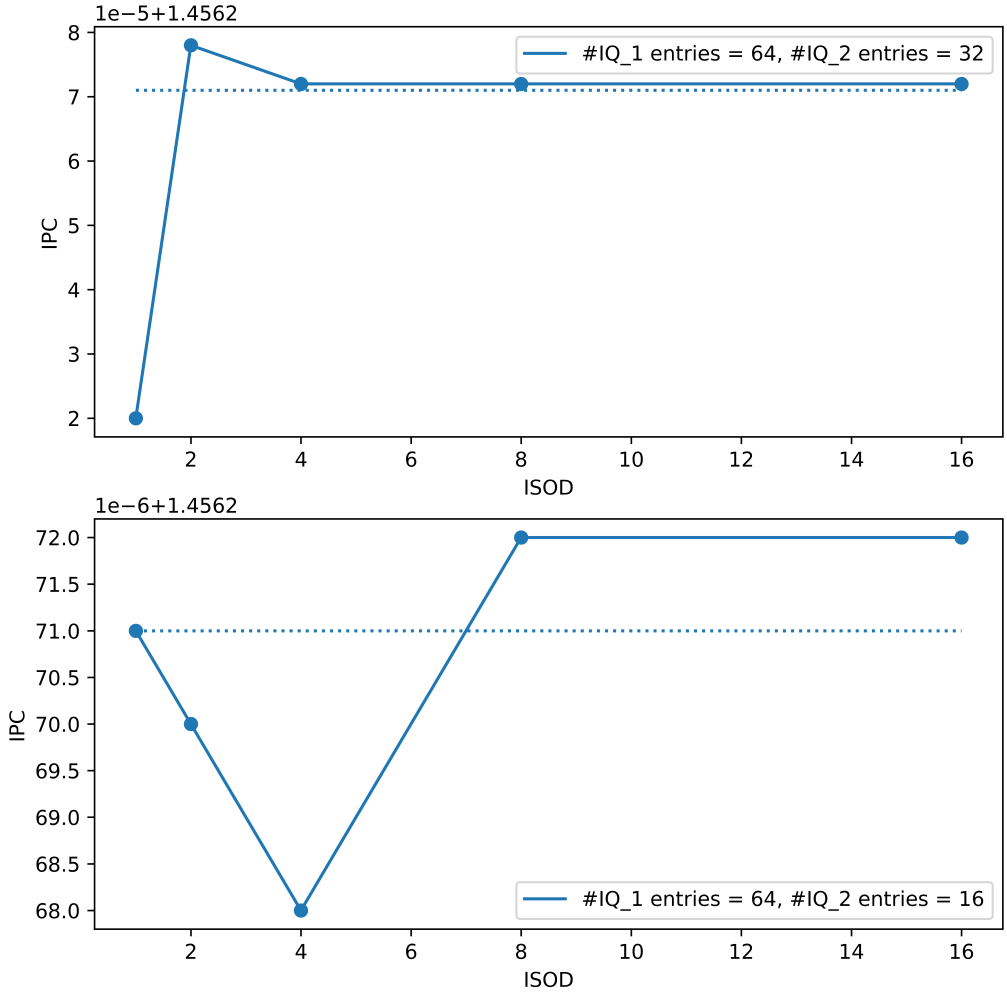


Figure 5.8: Heterogeneous IQ (matrix and traditional) IPC for Neural Networks benchmark

- IQ with only one source operand
- IQ with two source operands
- IQ with three source operands

An IQ with only one source operand can be dedicated to instructions of the immediate category, which constitutes instructions like load, addi, slli, etc. It can also be utilized by instructions where one of the source operands has already completed its execution before the issues into the IQ.

Instruction queues with two or more operands should be used to allocate instructions that need to wait on both of their source operands. These can also operate as fallbacks in the case where IQ with only one source operand is full.

We would ideally evaluate the power usage and power metrics for the heterogeneous instruction queues through simulation. However, power models in architectural simulations are mostly rudimentary and do not offer much higher accuracy than power modeling by hand. So, instead of modeling power in the simulators, which is expected to yield similar results as hand-calculated models, the following equations demonstrate the resource estimation of heterogeneous instruction queues.

Assume a traditional instruction queue with 64 entries with two source operands. For a 4-issue-wide superscalar processor, this would naturally require 4 wake-up ports for operation; along with those ports, it would additionally need 4 more wake-up ports for forwarding for a total of 8 wake-up ports. Assuming that there are 128 physical registers, in every clock, we would be required to compare all of these values against all the values in the IQ, totaling for **1024 7-bit comparators** as shown in Table 5.3.

Now, let's consider a heterogeneous instruction queue that yields a similar IPC as of the earlier traditional version with a generous #IQ_1 entries = 64 #IQ_2 entries

Table 5.3: Comparators in Traditional IQ

#Physical Registers	= 128
#Bits / Phy. Register	= $log_2(128) = 7$
#Comparators	= #Entries In IQ * #Wake Up Ports * #Operands
#Wake Up Ports	= 8
#Entries In IQ	= 64
#Operands	= 2
#Comparators	= 64 * 8 * 2 = 1024
#Registers to store indices	= #Entries In IQ * #Bits/Phy. Reg * #Operands
#Registers to store indices	= 64 * 7 * 2 = 896

= 16. For a similar wake-up configuration, the number of comparators is shown in Table 5.4. We see that we only need **768 7-bit comparators**, which is only 75% of comparators needed for the full IQ and has a similar IPC as that of the full IQ.

Table 5.4: Comparators in Heterogeneous IQ

#Wake Up Ports	= 8
#Entries In IQ_1	= 64
#Operands	= 1
#Comparators IQ_1	= 64 * 8 * 1 = 512
#Entries in IQ_2	= 16
#Comparators IQ_2	= 16 * 8 * 2 = 256
#Total Comparators	= 512 + 256 = 768
#Registers to store indices	= 64 * 7 * 1 + 16 * 7 * 2 = 672

Similarly, for a heterogeneous IQ with a generous #IQ_1 entries = 64 made of matrix IQ *max(ISOD) = 1* and IQ_2 made of traditional IQ with 16 entries, the number of

Table 5.5: Resources of in Heterogeneous IQ (Matrix IQ & Traditional IQ)

#Entries In IQ_1	= 64
max(ISOD)	= 1
#Muxes for IQ_1	= #Entries in IQ_1 = 64
Mux size	= # Physical Registers $\rightarrow$ 1 = 128 $\rightarrow$ 1
#Registers to store indices	= $\#entries * (sparseindex + denseindex)$
#Registers to store indices	= 64 * (7 + 1) = 512
#Entries in IQ_2	= 16
#Comparators IQ_2	= 16 * 8 * 2 = 256
#Registers to store indices(IQ_2)	= 16 * 7 * 2 = 224

comparators needed would be **256 7-bit comparators** and would also use **64 128$\rightarrow$1** mux gates. So, the final design requirements are shown in Table 5.5.

Finally, the differences boil down to the fact that a matrix scheduler uses muxes as compared to a traditional IQ, which uses comparators. In this case, each entry of the matrix scheduler in IQ_1 uses 1 128$\rightarrow$1 mux, whereas an entry in the traditional IQ uses $7 * 2 * 16 = 224$ comparators. This reduction in number of gates should yield reduced area and power consumption of the design.

From all of this, it's highly likely that the heterogenous IQ composed of the matrix IQ and traditional IQ should have the equivalent performance of traditional IQ of similar size at a lower power consumption. Of course, nothing comes for free. In this case, the reason behind the advantages of a matrix scheduler is also the reason behind its disadvantages, the muxes. Muxes allows us to shrink the IQ to work at higher efficiencies. At the same time, they could impact the clock periods of the processor. Traversing the levels of mux would require additional clock time, which is not present in the case of the comparators, which largely run in parallel. However, this slight addition of the clock period hopefully shouldn't skew the results. As long as the critical path

in the processor is not the IQ, even with this additional clock time, it still shouldn't be in the critical path.

Chapter 6

Implementation

The second-fold aim of this thesis is to implement an out-of-order processor called O3 with matrix scheduler and demonstrate its feasibility through a hardware description language. The reason behind creating a new processor and not using extending existing processors is that the matrix IQ necessitates a new design such that updating existing processors would only make it more complicated.

To do this, the language Chisel was chosen. Chisel is a relatively new language that focuses on ease of use, ability to optimize boolean logic, and safety through features like: combinational loop detection and type safety.

The requirements of the implemented processor are:

- Must be synthesizable and have a small footprint
- Must be divided into stages, and stages can communicate through latches
- Each stage should be simple and allow for higher clocks

The requirements of the instruction queue are:

- It has to be as efficient (IPC) as the traditional instruction queue
- It has to be able to resolve dependencies within one cycle
- It must have a way to incorporate age while selecting the instructions

An out-of-order processor is composed of 3 notable sections: In-Order front end, Out-of-Order execution & In-Order back end. The front end of the processor is composed of: Fetch, Decode, and RAT. The out-of-order core components are: Instruction Queue, Register file, Execute, and Memory. The back end is consists of: Re-Order buffer, Retirement RAT & Commit.

An out-of-order processor is different from an in-order one in that issuing instructions requires allocations in components like IQ, load-store queues, ROB entries, and physical registers. These can be allocated during the execution of the specific stage, but it is extremely inefficient as these could lead to stalls at various locations of the pipeline, and each of these stalls should be handled differently. A much better design is to allocate the required entries as soon as the instructions are decoded. This means that the decode stage can now stall due to the failure of allocation of all of these stages. A stall in the in-order front end is much easier to manage than a stall in the out-of-order core, as shown in Listing 6.1.

```
class OutOfOrderCPU()(implicit val p: Parameters) extends Module {
  ...

  val dstReg = RegInit(...)
  val robIdx = RegInit(...)
  val iqIdx = RegInit(...)
  val memIdx = RegInit(...)

  when(flush) {
    dstReg.valid, robIdx.valid, iqIdx.valid, memIdx.valid := false.B
   ...
  }

  when(flush || !ifIdSignals.valid || !ifIdSignals.bits.stage.fetch.
   instruction.valid) {
    rename.io.allocate := false.B
    rob.io.allocate := false.B
    iq.io.allocate := false.B
    memory.io.allocate.valid := false.B
    ...
    decode.io.inst := NOP.U
    ...
    dStall := false.B
  }.otherwise {
    decode.io.inst := ifIdSignals.bits.stage.fetch.instruction.bits
    val decodeSignalsOut = decode.io.signals

    val dstRegW = Wire(Valid(UInt(log2Ceil(nPhyRegs).W)))
    val robIdxW = Wire(Valid(UInt(log2Ceil(nROBEntries).W)))
    val iqIdxW = Wire(Valid(UInt(log2Ceil(nIQEntries).W)))
    val memIdxW = Wire(Valid(new LoadStoreIndex()))
```

```
 val regShouldAllocate = decodeSignalsOut.regWrite &&
decodeSignalsOut.rd =/= 0.U

 when(regShouldAllocate && !dstReg.valid) {
   rename.io.allocate := true.B
   dstRegW := rename.io.allocatedIdx
 }.otherwise {
   rename.io.allocate := false.B
   dstRegW := dstReg
 }
 when(!robIdx.valid) {
   rob.io.allocate := true.B
   robIdxW := rob.io.allocatedIdx
 }.otherwise {
   rob.io.allocate := false.B
   robIdxW := robIdx
 }
 when(!iqIdx.valid) {
   iq.io.allocate := true.B
   iqIdxW := iq.io.allocatedIdx
 }.otherwise {
   iq.io.allocate := false.B
   iqIdxW := iqIdx
 }

 val memShouldAllocate = isValid(decodeSignalsOut.memRead) ||
isValid(decodeSignalsOut.memWrite)

 when(memShouldAllocate && !memIdx.valid) {
   memory.io.allocate.valid := memShouldAllocate
```

```
      memory.io.allocate.bits := Mux(isValid(decodeSignalsOut.memRead
   ), MemRWDirection.read, MemRWDirection.write)
      memIdxW := memory.io.allocatedIdx
    }.otherwise {
      memory.io.allocate.valid := false.B
      memory.io.allocate.bits := DontCare
      memIdxW := memIdx
    }
    val regAllocSuccess = (regShouldAllocate && dstRegW.valid) || (!
   regShouldAllocate)
    val robAllocSuccess = robIdxW.valid
    val iqAllocSuccess = iqIdxW.valid
    val memAllocSuccess = (memShouldAllocate && memIdxW.valid) || (!
   memShouldAllocate)
    when(regAllocSuccess && robAllocSuccess && iqAllocSuccess &&
   memAllocSuccess) {
      idRenameSignals.valid := true.B
      idRenameSignals.bits := ...
      idRenameSignals.bits.stage.alloc... := dstRegW...
      dstReg, robIdx, iqIdx, memIdx.valid := false.B...
      dStall := false.B
    }.otherwise {
      idRenameSignals.valid := false.B...
      dstReg, robIdx, iqIdx, memIdx := dstRegW...
      dStall := true.B
    }
  }
  ...
}
```

Listing 6.1: Fetch and Decode stages routing in the O3 processor

During a flush, when either a branch is mispredicted, or a load-store dependency is violated, the processor marks the instruction to be flushed in the ROB. When the ROB tries to retire the instruction, it checks if it should be flushed and begins the flushing process if necessary.

Similarly, if the instruction is retired, the memory stage needs to check if it's a retired store. If it's a store that has retired and an existing load has loaded from the same address that the store is storing, then the load needs to be flushed.

The flushing process is:

- Update the program counter to point to the appropriate address
- Reset all the IQ entries list and mark all the entries to be free
- Overwrite the contents of RAT from the Retirement RAT (RRAT)
- Mark all the registers as free except for the ones pointed by RRAT
- Invalidate all the Load Queue entries
- Mark all the stores that haven't retired as invalid
- Mark all the instructions of all the stages as invalid

The stages of the processor are connected via latches, which allow operating at higher frequencies through pipelining, as shown in Figure 6.1.

During a branch misprediction, which is detected in the execute stage, the execute stage checks the ROB for the next instruction's program counter. If the next ROB entry is allocated and the PC is assigned, then the actual PC and the predicted PC are compared; if these values are different, then the next ROB entry needs to be flushed, resetting the values. However, a flush in the middle of the ROB would require the RAT to be reconstructed from the instruction entries leading up to the flushing entry.

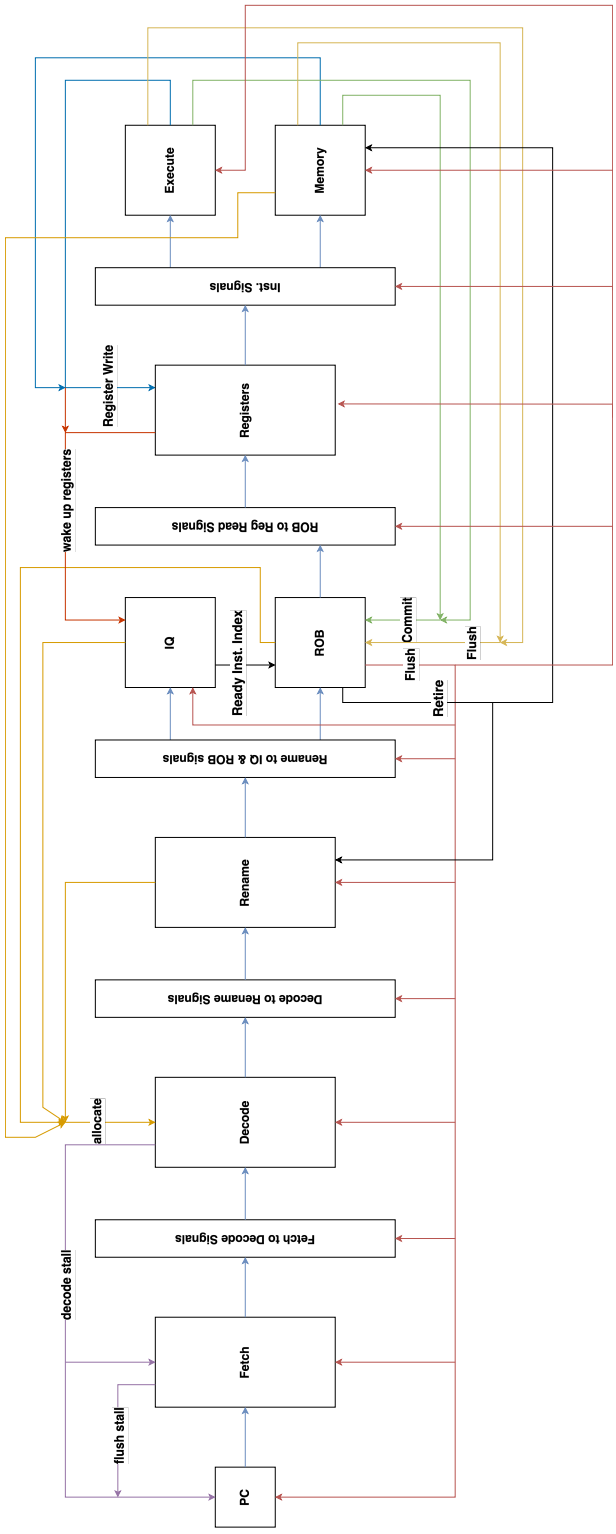


Figure 6.1: Datapath of the Out of Order Processor

To combat this, the actual flushing is only performed when the flushed instruction reaches the ROB head (i.e., when it is about to retire). This design attains a good balance of complexity and efficiency, as today's branch predictors are extremely accurate. Listing 6.2 shows the logic of flushing in ROB. Note that all of the entries of ROB (*entryValidList, entryCommittedList, entryFlushedList, entryExceptedList, entryPCList, entryDataList*) are broken up into individual register arrays. This is done because having a common register array would necessitate read/write ports for reading and writing to each of the entries, which are otherwise unused leading hardware, leading to higher resource usage.

```
class ROB()(implicit val params: Parameters) extends Module {
  ...
  val robHead = RegInit(0.U(log2Ceil(nROBEntries).W))
  val robTail = RegInit(0.U(log2Ceil(nROBEntries).W))
  ...
  val entryValidList = Mem(nROBEntries, Bool())
  val entryCommittedList = Mem(nROBEntries, Bool())
  val entryFlushedList = Mem(nROBEntries, Bool())
  val entryExceptedList = Mem(nROBEntries, Bool())
  val entryPCList = Mem(nROBEntries, UInt(addrWidth.W))
  val entryDataList = Mem(nROBEntries, new ROBSignals())
  ...
  when(io.flush.valid) {
    when(!entryValidList(io.flush.bits.robIdx)) {
      when(robTail === io.flush.bits.robIdx) {
        robTailIn1 := robTail + 1.U
      }
      entryValidList(io.flush.bits.robIdx) := true.B
      entryCommittedList(io.flush.bits.robIdx) := true.B
    }
```

```
    entryFlushedList(io.flush.bits.robIdx) := true.B

    when(io.flush.bits.updatePc.valid) {
      entryPCList(io.flush.bits.robIdx) := io.flush.bits.updatePc.
   bits
    }
  }
  ...
  when(entryValidList(robHead) && entryCommittedList(robHead) &&
   robHead =/= robTail) {
    val nextRobHead = robHead + 1.U
    entryValidList(robHead) := false.B

    io.retireInst.valid := true.B
    io.retireInst.bits.pc := entryPCList(robHead)
    io.retireInst.bits.flush := entryFlushedList(robHead)
    io.retireInst.bits.signals := entryDataList(robHead)

    when(entryFlushedList(robHead)) {
      robTail := nextRobHead
    }
    robHead := nextRobHead
  }.otherwise {
    io.retireInst.valid := false.B
    io.retireInst.bits := DontCare
  }
}
```

Listing 6.2: ROB stage and flushing logic of O3 processor

Similarly, the memory stage is built upon the store ordering philosophy includes a load queue and store queue to disambiguate loads and stores and reorder them to the correct order. Load queue and store queue are allocated in the decode stage, which runs in the in-order front end, making them naturally follow the program order irrespective of the execution path. However, during speculation, it is possible for a load to be issued before a store, whereas the program order stores before the load. These kinds of situations need dedicated logic to handle dependencies and flushing.

As González et al. [8] describes, the architecture of AMD K6 includes load-store ordering of the processor. O3 follows a similar philosophy in memory disambiguation to that of K6. A store instruction's contents are stored in the store buffer until the instruction retires, while the memory stage writes back to the memory asynchronously. When a store retires, the load queue is checked for any loads that have been executed with the same address as the store. This is a situation where the store has not completed, but the load has speculatively executed, and the contents are incorrectly fetched. The only way to recover from this is to flush the load. This memory disambiguation logic is shown in the Listing 6.3.

```
class MemoryO3()(implicit params: Parameters) extends Module {
  ...

  when(io.retireIdx.valid) {
    when(io.retireIdx.bits.rwDirection === MemRWDirection.read) {
      loadQueue(io.retireIdx.bits.asLoadIndex()).reserved := false.B
      loadQueueHead := loadQueueHead + 1.U
    }
    when(io.retireIdx.bits.rwDirection === MemRWDirection.write) {
      val stIdx = io.retireIdx.bits.asStoreIndex()
      val stAddr = storeQueue(stIdx).address.bits
      storeQueue(stIdx).retired := true.B
```

```
      storeRetireTail := storeRetireTail + 1.U

      val flIdx = PriorityMux(Seq.tabulate(nLdQEntries + 1){ idx =>
        val flushIdxW = Wire(Valid(UInt(log2Ceil(nROBEntries).W)))
        if (idx < nLdQEntries) {
          val ldIdx = loadQueueHead + idx.U
          val loadQueueLen = loadQueueTail - loadQueueHead
          val valid = idx.U < loadQueueLen && loadQueue(ldIdx).
   address.valid && align(loadQueue(ldIdx).address.bits, dataBytes) =
   == align(stAddr, dataBytes)
          val robIdx = loadQueue(ldIdx).robIdx.bits
          flushIdxW.valid := valid
          flushIdxW.bits := robIdx
          (valid, flushIdxW)
        } else {
          flushIdxW.valid := false.B
          flushIdxW.bits := DontCare
          (true.B, flushIdxW)
        }
      })

      io.flushIdx := flIdx
    }
  }
}
```

Listing 6.3: Memory disambiguation in O3 processor

The next component of the processor is the Instruction Queue. This is heart of the Out of Order engine. As mentioned earlier, the instruction queue is built upon the matrix scheduling ideology. The operation of the instruction queue can be described in 4 steps:

- Allocate the entry in IQ
- Mark the correct dependencies in the IQ during the issue
- Mark the dependency registers as 0 if the register is ready
- If all the dependency registers are 0, then the instruction is ready for dispatch

Although this design works correctly, it is not as efficient as we would like it to be. Specifically, this design, shown in Figure 6.2, causes instructions to be woken up one cycle after the source register is ready. This creates a "loose loop" (Borch et al. [5]) and makes the processor inefficient in handling back-to-back dependent instructions. Note that the dependency write to these registers is simplified (A more complete design will have dependencies flow in to set the register values through an OR gate).

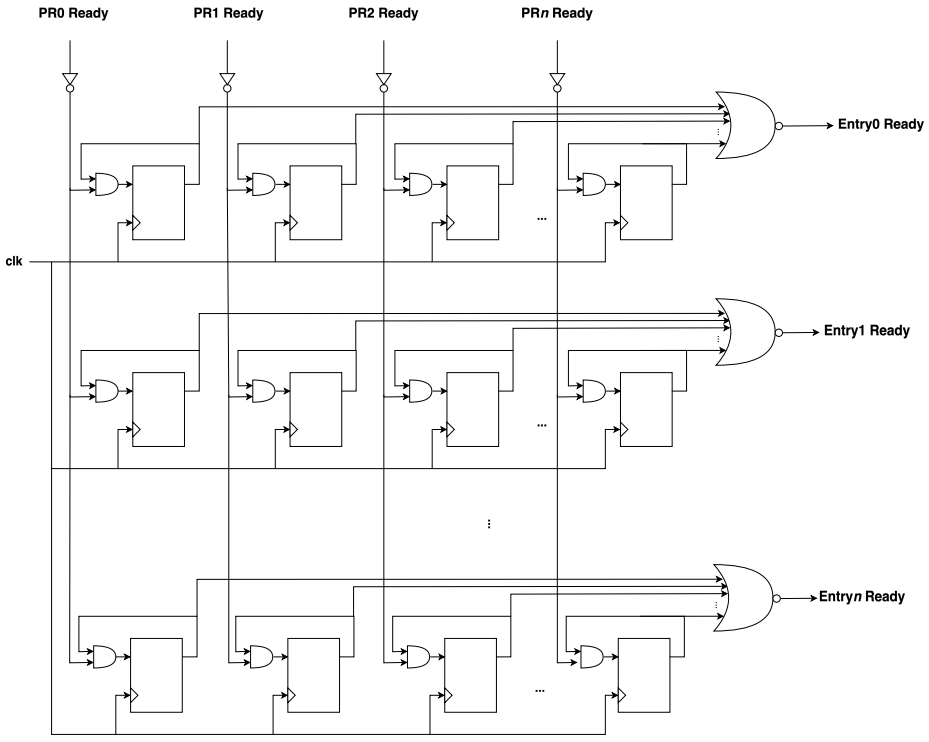


Figure 6.2: IQ with dedicated write back to dependency registers (simplified)

A traditional IQ can only have a limited number of wake-up ports since the complexity scales linearly with respect to the number of wake-up ports. However, a bit vector indicating the readiness of the registers is better suited for the matrix scheduler in which the individual registers and dependencies are modeled. It is the chosen approach for this processor because it helps us design an IQ which wakes up ready instructions within one cycle. The idea behind the single cycle implementation of the IQ is that all the registers that are needed for an instruction will all be ready during its execution. Figure 6.3 shows the updated design, which can wake up the instruction in a single cycle.

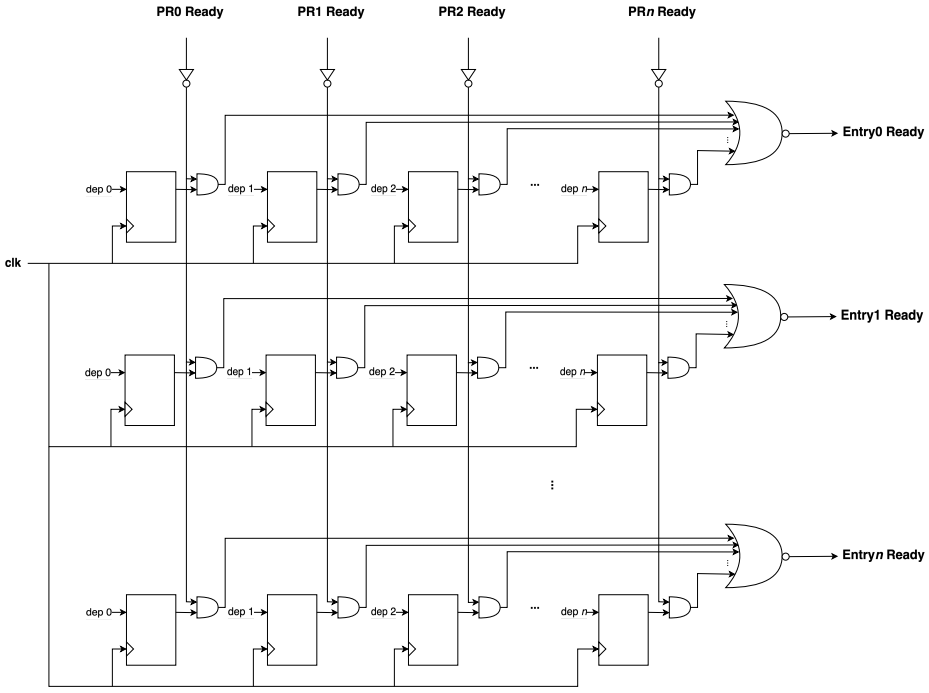


Figure 6.3: Matrix IQ with single cycle wakeup

In a traditional IQ, wake-up ports are limited due to the complexity of the design it creates. Specifically, in the design, each wake-up port fans out to all the entries of IQ, and comparators run, resolving the dependencies. In contrast, the design created by the matrix IQ is much simpler. The register-ready bit fans out to all the entries,

the dependencies are resolved, and the OR across all the dependencies indicates if the instruction is ready or not.

A priority mux is used to select one instruction that is ready. It is to be noted that in the current version of the IQ, the mux logic is simple and prioritizes the instruction at the top of the queue. However, this can be updated to age ordering through techniques very similar to the techniques used for traditional IQ. Listing 6.4 shows the design of the instruction queue.

```
class InstructionQueue()(implicit val p: Parameters) extends Module {
  ...

  for (e <- 0 until nEntries) {
    val deps = VecInit(Seq.tabulate(nPhyRegs) { reg => !io.wakeUpRegs
   (reg) & map(e)(reg) })
    val depPending = deps.reduceTree(_ | _)

    readyInsts(e) := !depPending & occupancies(e) & allocations(e)
  }

  ...
}
```

Listing 6.4: Dependency resolving logic in the O3 processor

Traditionally, wake-up registers are passed using the register index. For example, If an integer ALU generates the value of physical register 3 (PR3), the bits representing 3 will be broadcast to the IQ, indicating that dependents of register 3 can be woken up now. This means that for a processor with n physical registers and m wake-up ports (usually equal to the width of the super-scalar), the number of bits required to signal are: $m * log_2(n)$. This linearly grows with the width of the superscalar processor.

Considering the option to pass all the wakeup registers as a bit vector doesn't make the circuit any more complex than it already is.

However, in the approach implemented in the O3 processor, all the registers and their ready values are passed in their individual wires. This simplifies the design of the instruction queue and retains the constant width of the wakeup port equal to the number of physical registers n. As mentioned earlier, this approach is also necessary for resolving dependencies in a single cycle.

The final and complete hardware of the instruction queue is shown in Listing 6.5.

```
class InstructionQueue()(implicit val p: Parameters) extends Module {
  ...

  val allocations = RegInit(0.U(nEntries.W))
  val occupancies = RegInit(0.U(nEntries.W))
  val stations = Mem(nEntries, UInt(log2Ceil(nROBEntries).W))
  val map = RegInit(VecInit(Seq.fill(nEntries) { 0.U(nPhyRegs.W) }))
  val readyInsts = Wire(Vec(nEntries, Bool()))

  val freeStationIdx = PriorityMux(Seq.tabulate(nEntries + 1){ idx =>
    if (idx < nEntries) {
      val idxW = Wire(Valid(UInt(log2Ceil(nEntries).W)))
      idxW.valid := true.B
      idxW.bits := idx.U
      (!allocations(idx), idxW)
    } else {
      val idxW = Wire(Valid(UInt(log2Ceil(nEntries).W)))
      idxW.valid := false.B
      idxW.bits := DontCare
      (true.B, idxW)
    }
  })
```

```
val readyInstIdx = PriorityMux(Seq.tabulate(nEntries + 1){ idx =>
  if (idx < nEntries) {
    val instSignals = Wire(Valid(UInt(log2Ceil(nEntries).W)))
    instSignals.valid := true.B
    instSignals.bits := idx.U
    (readyInsts(idx), instSignals)
  } else {
    val instSignals = Wire(Valid(UInt(log2Ceil(nEntries).W)))
    instSignals.valid := false.B
    instSignals.bits := DontCare
    (true.B, instSignals)
  }
})

for (e <- 0 until nEntries) {
  val deps = VecInit(Seq.tabulate(nPhyRegs) { reg => !io.wakeUpRegs
 (reg) & map(e)(reg) })
  val depPending = deps.reduceTree(_ | _)
  readyInsts(e) := !depPending & occupancies(e) & allocations(e)
}

val nextAllocationsIntermediate = Wire(UInt(nEntries.W))
val nextAllocations = Wire(UInt(nEntries.W))
val nextOccupanciesIntermediate = Wire(UInt(nEntries.W))
val nextOccupancies = Wire(UInt(nEntries.W))
when(readyInstIdx.valid) {
  nextAllocationsIntermediate := allocations & (~(1.U <<
 readyInstIdx.bits)).asUInt
  nextOccupanciesIntermediate := occupancies & (~(1.U <<
 readyInstIdx.bits)).asUInt
```

```
}.otherwise {
  nextAllocationsIntermediate := allocations
  nextOccupanciesIntermediate := occupancies
}
when(io.allocate && freeStationIdx.valid) {
  val stationIdx = freeStationIdx.bits
  io.allocatedIdx.valid := true.B
  io.allocatedIdx.bits := stationIdx
  nextAllocations := nextAllocationsIntermediate | (1.U <<
 stationIdx).asUInt
}.otherwise {
  io.allocatedIdx.valid := false.B
  io.allocatedIdx.bits := DontCare
  nextAllocations := nextAllocationsIntermediate
}

when(io.instSignals.valid) {
  val stationIdx = io.instSignals.bits.iqIdx
  stations(stationIdx) := io.instSignals.bits.robIdx
  val mask = Wire(Vec(nPhyRegs, Bool()))
  for (r <- 0 until nPhyRegs) {
    mask(r) := false.B
  }
  when(io.instSignals.bits.rs1PhyReg.valid) {
    mask(io.instSignals.bits.rs1PhyReg.bits) := true.B
  }
  when(io.instSignals.bits.rs2PhyReg.valid) {
    mask(io.instSignals.bits.rs2PhyReg.bits) := true.B
  }
  map(stationIdx) := mask.asUInt
```

```
    nextOccupancies := nextOccupanciesIntermediate | (1.U <<
   stationIdx).asUInt
  }.otherwise {
    nextOccupancies := nextOccupanciesIntermediate
  }

  allocations := nextAllocations
  occupancies := nextOccupancies

  when(io.flush) {
    allocations := 0.U
    occupancies := 0.U
  }

  io.readyInstSignals.valid := readyInstIdx.valid
  io.readyInstSignals.bits := Mux(readyInstIdx.valid, stations(
   readyInstIdx.bits), DontCare)
}
```

Listing 6.5: Instruction Queue logic of the O3 processor

Finally, this design is synthesized using Vivado; the final output of the generated schematic can be seen in Figure 6.4, and a zoomed-in version indicating the latches between the stages can be seen in Figure 6.5. The timing of the processor is shown in Figure 6.6, and a clearer floor plan is shown in Figure 6.7.

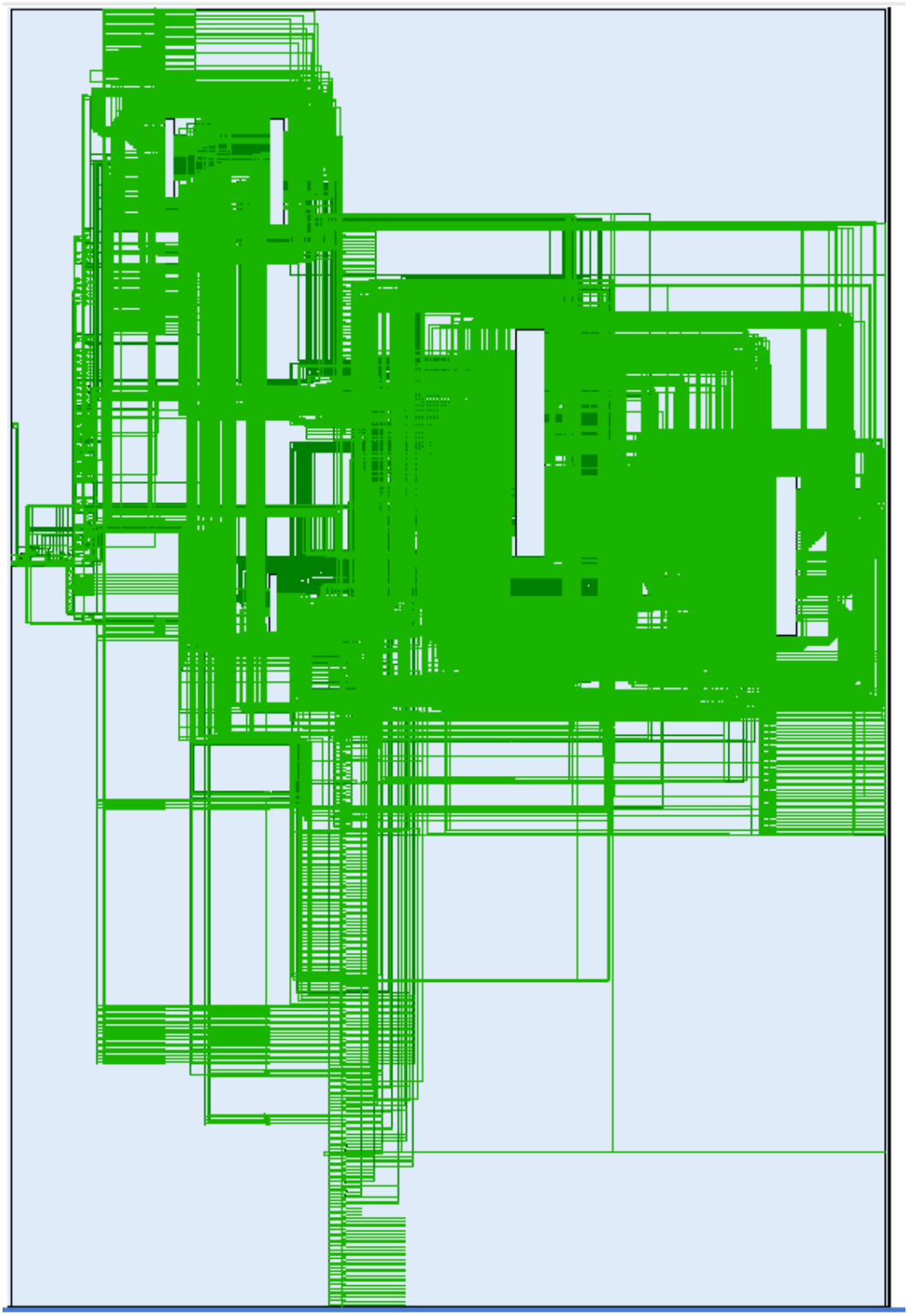


Figure 6.4: Schematic of the generated processor

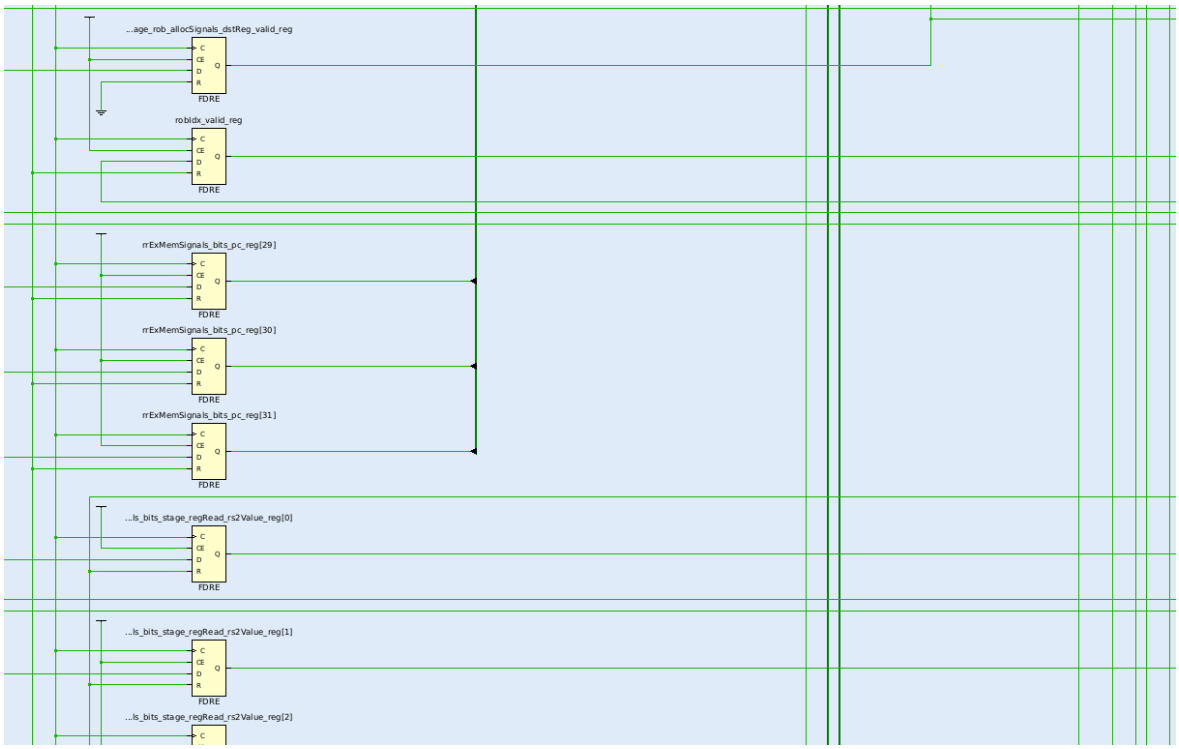


Figure 6.5: Schematic of the generated processor in latches between stages

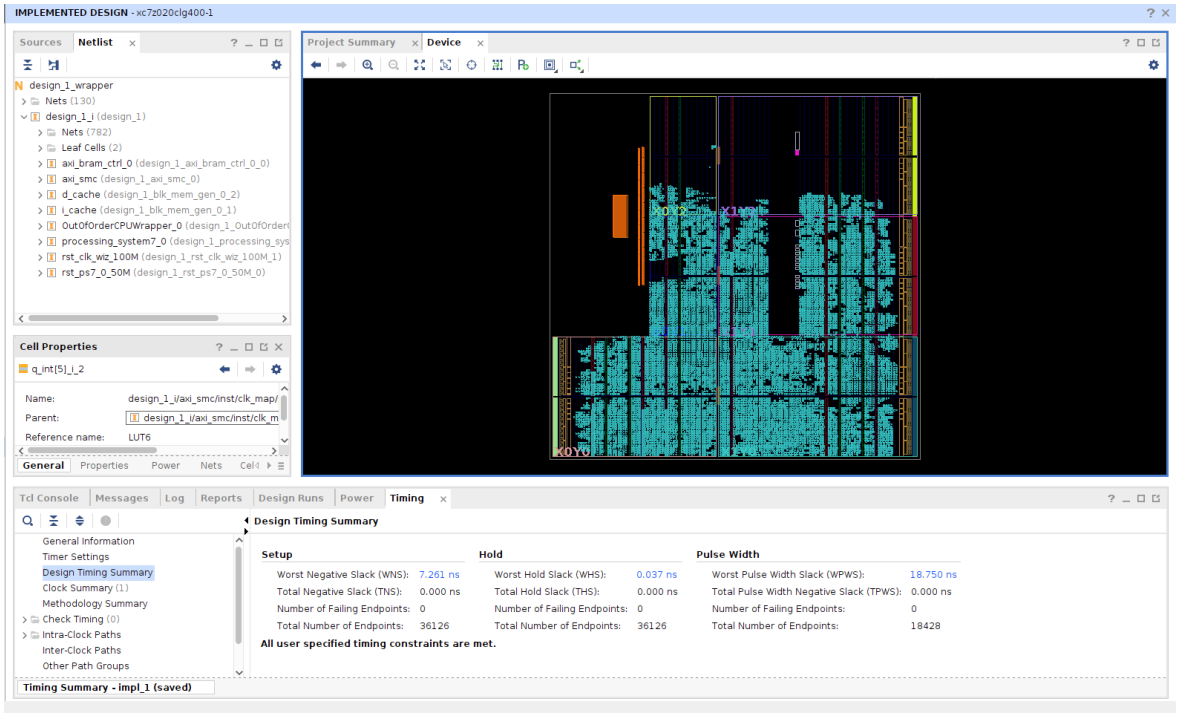


Figure 6.6: Timing and floor plan of the processor

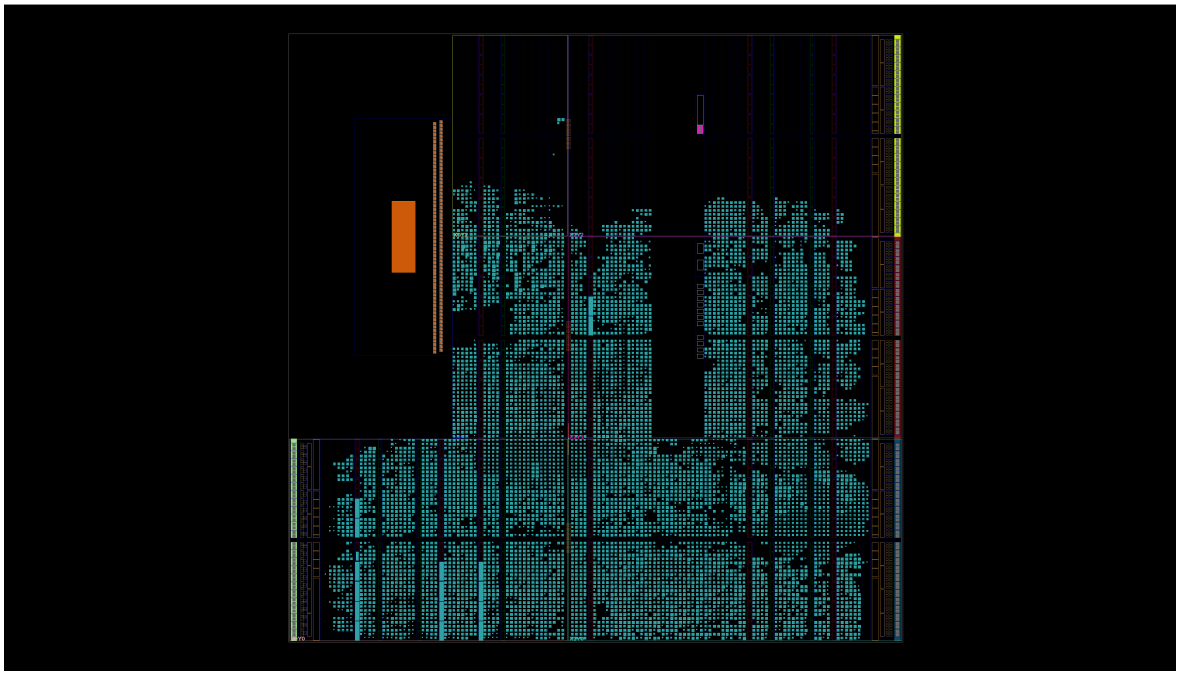


Figure 6.7: Floor plan of the processor

Chapter 7

Related Work

Since the inception of Out-of-Order processing by Tomasulo [15], computer architects have devised various scheduling tactics like: age ordering, priority-based scheduling, and resource-aware scheduling to schedule instructions efficiently. These approaches have been primarily constrained to the power-hungry CAM approaches; the ideas of matrix schedulers have not been explored thoroughly.

Matrix schedulers aren't new; the ideas of CAM-less matrix schedulers have been in the literature since at least the early 2000s. Goshima et al. [9] has proposed that the matrix schedulers can be implemented by tracking dependencies between instructions, techniques similar to register renaming can be implemented to resolve dependencies, and dependency matrices can be narrowed by following the idea that the producer of a register is always just a few instructions away from the consumer.

Similarly, Sassone et al. [13] presented that most of the instructions don't broadcast, and even if they broadcast, all the instructions in the IQ usually neglect that broadcast and has proposed subscription-based wake-up can result in efficient hardware designs with lower power consumption.

Brekelbaum et al. [6] presents hierarchical scheduling windows that model the instruction queues, which are longer and shorter in the look-ahead distances and could extract ILP. Studies by Austin and Sohi [3] and Fatehi and Gratz [7] present that even today, much ILP is still being left on the table, which could be extracted using longer IQ windows.

An article by Abella et al. [1] describes a matrix scheduler that can be implemented using bit matrices of register dependencies, as discussed in this thesis. However, the article does not mention compression techniques that could be implemented or analyze any metrics of the matrix schedulers.

Alipour et al. [2] suggests a radical and efficient approach using FIFO queues. One could develop power-efficient instruction queues similar to the techniques proposed in this thesis by classifying instructions as ready and almost ready. However, the studies conducted in the paper focus on x86 processors and approach the problem from a different perspective than that of this thesis.

Modern compilers are smart, smart to the point that they schedule instructions based on the heuristics of the target architecture and aim to extract the utmost performance at all levels of parallelism: ILP, TLP, etc. Even then, processors today are not able to take full advantage of the machine code due to the inherent limitation of the size of the instruction queue. One way to take advantage is to increase the instruction queue size, i.e., the processor's look-ahead window.

Increasing the IQ width helps the processor extract additional instruction-level parallelism (ILP). In other words, increasing the width of IQ has a direct impact on the processor's single-threaded performance, and architects have constantly pursued increasing IQ widths. The limiting factor in this case is that longer IQ widths directly impact the power consumption & timing of the stage, and also affect the critical path of the processor.

However, all of these approaches work on a variant of the matrix scheduler that barely corresponds to the proposed idea or fails to address the fact that the matrix can be compressed using the inter-source operand distance (ISOD) metric.

Chapter 8

Summary and Conclusions

This work demonstrated the utility of structuring the instruction queue into multiple queues: one designated for instructions with one or fewer source operands and the other for instructions with two or more source operands. In such a system, an efficient allocation technique would involve allocating space on both queues during decode, deallocating the unused IQ space during issue, and deallocating the used IQ during dispatch. A stall might be necessary if either of the queues reaches capacity.

One potential heterogeneous instruction queue implementation approach involves using the matrix instruction queue to support the queue for instructions with one or fewer source operands. Selecting an optimal Inter Source Operand Distance (ISOD) for the matrix instruction queue will require careful consideration, ideally informed by benchmarking results. Based on the findings in this study, an ISOD of at least 1 or higher performs reliably within the context of the RISC-V ISA.

A key metric that compilers aim to optimize is register pressure. Compilers consistently work to minimize the number of registers a program uses. This approach typically involves moving older variables to memory while keeping more recent variables in registers. This strategy has the beneficial effect of lowering the ISOD and is visible in the results above.

In conclusion, this thesis introduces the concept of ISOD, explores the ideas of designing heterogeneous instruction queues, and describes the ways of compressing matrix instruction queues for use in heterogeneous instruction queues. However, due

to the complexities of isolating the power metrics of these very large scale circuits from other components, specific power measurements have not been detailed in this work and are left as an area for future research and development.

Bibliography

[1] Abella, J., R. Canal, and A. Gonzalez, 2003: Power- and complexity-aware issue queue designs. *IEEE Micro*, **23 (5)**, 50–58, https://doi.org/10.1109/MM.2003.1240212.

[2] Alipour, M., R. Kumar, S. Kaxiras, and D. Black-Schaffer, 2019: Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 716–721, https://doi.org/10.23919/DATE.2019.8715034.

[3] Austin, T. M., and G. S. Sohi, 1992: Dynamic dependency analysis of ordinary programs. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, USA, 342–351, ISCA '92, https://doi.org/10.1145/139669.140395, URL https://doi.org/10.1145/139669.140395.

[4] Binkert, N. L., and Coauthors, 2011: The gem5 simulator. *SIGARCH Comput. Archit. News*, **39 (2)**, 1–7, https://doi.org/10.1145/2024716.2024718, URL https://doi.org/10.1145/2024716.2024718.

[5] Borch, E., E. Tune, S. Manne, and J. Emer, 2002: Loose loops sink chips. *Proceedings Eighth International Symposium on High Performance Computer Architecture*, 299–310, https://doi.org/10.1109/HPCA.2002.995719.

[6] Brekelbaum, E., J. Rupley, C. Wilkerson, and B. Black, 2002: Hierarchical scheduling windows. *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 27–36, https://doi.org/10.1109/MICRO.2002.1176236.

[7] Fatehi, E., and P. Gratz, 2014: Ilp and tlp in shared memory applications: a limit study. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, Association for Computing Machinery, New York, NY, USA, 113–126, PACT '14, https://doi.org/10.1145/2628071.2628093, URL https://doi.org/10.1145/2628071.2628093.

[8] González, A., F. Latorre, and G. Magklis, 2010: *Processor Microarchitecture: An Implementation Perspective*, Vol. 5. https://doi.org/10.2200/S00309ED1V01Y201011CAC012.

[9] Goshima, M., K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita, 2001: A high-speed dynamic instruction scheduling scheme for supersealar processors. *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 225–236, https://doi.org/10.1109/MICRO.2001.991121.

[10] Hansson, A., N. Agarwal, A. Kolli, T. F. Wenisch, and A. N. Udipi, 2014: Simulating DRAM controllers for future system architecture exploration. *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, IEEE Computer Society, 201–210, https://doi.org/10.1109/ISPASS.2014.6844484, URL https://doi.org/10.1109/ISPASS.2014.6844484.

[11] Kessler, R., 1999: The alpha 21264 microprocessor. *IEEE Micro*, **19 (2)**, 24–36, https://doi.org/10.1109/40.755465.

[12] Lowe-Power, J., and Coauthors, 2020: The gem5 simulator: Version 20.0+. *CoRR*, **abs/2007.03152**, URL https://arxiv.org/abs/2007.03152, 2007.03152.

[13] Sassone, P. G., J. Rupley, E. Brekelbaum, G. H. Loh, and B. Black, 2007: Matrix scheduler reloaded. *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, USA, 335–346, ISCA '07, https://doi.org/10.1145/1250662.1250704, URL https://doi.org/10.1145/1250662.1250704.

[14] Seznec, A., S. Jourdan, P. Sainrat, and P. Michaud, 1996: Multiple-block ahead branch predictors. *SIGPLAN Not.*, **31 (9)**, 116–127, https://doi.org/10.1145/248209.237169, URL https://doi.org/10.1145/248209.237169.

[15] Tomasulo, R. M., 1967: An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, **11 (1)**, 25–33, https://doi.org/10.1147/rd.111.0025.

[16] Waterman, A., 2016: Design of the risc-v instruction set architecture. URL https://api.semanticscholar.org/CorpusID:63861396.

1 Appendix A

All the code is open-sourced and can be found at:

- https://github.com/enqueueDequeue/gem5
- https://github.com/enqueueDequeue/riscy