ABSTRACT

MAYUKH, HIRAN. FabIssue: Automatic RTL Generation of Issue Logic in Superscalar Processors for Core Customization. (Under the direction of Dr. Eric Rotenberg.)

Superscalar processors achieve high performance by fetching and executing multiple instructions each cycle. A superscalar processor exposes instruction-level parallelism (ILP) by maintaining a large pool of instructions in a structure called the issue queue. Each cycle, ready-to-execute instructions are identified in the issue queue and a certain number of them are selected to issue in parallel to the execution units. This process is handled by the issue logic. The issue logic is characterized by three parameters: (i) Issue width, which is the maximum number of instructions that can be issued in a cycle, (ii) Issue queue size, i.e., the number of issue queue entries, and (iii) Issue depth, which is the degree of pipelining of the issue logic. The configuration of the issue logic affects workload performance (defined as instructions per unit time, or IPT). This thesis consists of three facets: (a) Understand how the configuration of the issue logic, program characteristics and technology characteristics influence performance. The question here is, “What factors cause a certain issue logic configuration to be the best?” (b) Explore diversity within a benchmark suite in order to extract performance by customizing issue logic to program behavior. The question is “Do different workloads have different best issue logic configurations and if so, can performance be improved using this knowledge, and by how much?” (c) Develop a tool, FabIssue, which enables answering the above questions. This tool is a part of the FabScalar [2] toolset and can generate synthesizable register-transfer-level (RTL) designs of issue logic of desired configuration.
FabIssue: Automatic RTL Generation of Issue Logic in Superscalar Processors for Core Customization

by
Hiran Mayukh

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2010

APPROVED BY:

_______________________________  ______________________________
Dr. Eric Rotenberg  Dr. James Tuck
Committee Chair

_______________________________
Dr. Huiyang Zhou
DEDICATION

To Mom and Dad
BIOGRAPHY

Hiran Mayukh was born in Mysore, Karnataka, India, in 1986. He received Bachelor of Technology (B.Tech) degree at the National Institute of Technology Karnataka (NITK), Surathkal in 2008. He is currently a graduate student in North Carolina State University, Raleigh, under the guidance of Dr. Eric Rotenberg.
I would like to express my gratitude to Prof. Eric Rotenberg for inspiring and guiding me throughout the duration of my masters at N.C. State. He is a wonderful teacher and it has been an honor working with him. I thank Prof. James Tuck and Prof. Huiyang Zhou for their support. This thesis would not have been possible without the work of Niket Choudhary, Salil Wadhavkar, Sandeep Navada, Jayneel Gandhi, Tanmay Shah and Hashem Hashemi – you guys are awesome. Thanks to Naser Sedaghati, Rami Al-Sheikh, Julian and others at CESR, my apartment-mates Sounder Rajan Vijay Kumar, Suhas Satish and Abhinav Naik, and my family for all the support and encouragement.

This thesis was supported in part by NSF grant No. CCF-0811707, Intel and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

LIST OF TABLES ........................................................................................................ vii  
LIST OF FIGURES .................................................................................................... viii  
1. Introduction ............................................................................................................. 1  
  1.1. Issue Logic and Performance ........................................................................... 2  
  1.2. Contribution .................................................................................................... 4  
2. Motivation ................................................................................................................ 6  
  2.1. Tradeoffs in Issue Logic Design ....................................................................... 8  
  2.1.1. Issue Width ................................................................................................. 9  
  2.1.2. Issue Queue Size ....................................................................................... 9  
  2.1.3. Issue Depth ............................................................................................... 10  
  2.2. Heterogeneity and Core Customization ......................................................... 14  
  2.3. FabIssue ....................................................................................................... 17  
3. Related Work ........................................................................................................... 19  
  3.1. Issue Logic ..................................................................................................... 19  
  3.2. Design and Performance ............................................................................... 19  
  3.3. Simulators ..................................................................................................... 20  
  3.4. FabScalar ..................................................................................................... 21  
4. Design of the Issue Logic ....................................................................................... 23  
  4.1. Working of the Issue Logic ............................................................................. 24  
  4.1.1. Dispatching an Instruction .......................................................................... 25  
  4.1.2. Generation of Request Vectors .................................................................... 25  
  4.1.3. Select Logic ............................................................................................... 26  
  4.1.4. Wakeup CAM ............................................................................................. 30  
  4.1.5. Result Shift Register .................................................................................. 30  
  4.1.6. Wakeup-Select Loop .................................................................................. 31  
  4.1.7. Free List Management ............................................................................... 34  
  4.1.8. Cascaded Select trees ............................................................................... 34  
  4.2. Pipelining the Issue Logic ............................................................................... 36  
  4.2.1. The 1/1 Issue Logic Configuration ............................................................. 36  
  4.2.2. The 2/2 Issue Logic Configuration ............................................................. 37  
  4.2.3. The 3/3 Issue Logic Configuration ............................................................. 38  
  4.2.4. The 2/3 Issue Logic Configuration ............................................................. 39  
  4.2.5 Pipelining the Select Logic .......................................................................... 40  
5. FabIssue .................................................................................................................. 42  
6. Methodology .......................................................................................................... 45  
  6.1. Experiments .................................................................................................... 45  
  6.2. Assumptions and Limitations ......................................................................... 46  
  6.3. Configuration of the Simulator ....................................................................... 47  
7. Results ...................................................................................................................... 50
7.1. Issue Queue Configuration and Performance ................................................................. 50
7.2. Diversity across benchmarks ........................................................................................ 52
7.3. Diversity within a benchmark ....................................................................................... 52
7.4. Understanding Diversity Results ................................................................................... 54
  7.4.1. Lack of Diversity in Issue Depth ............................................................................ 54
  7.4.2. Lack of Diversity in Issue Width ........................................................................... 55
  7.4.3. Understanding the Best Issue Logic Configuration .............................................. 56
  7.4.4. Eliminating Issue Logic Configurations ............................................................... 59
7.5. Can the Best Configuration be Wide or Shallow? ....................................................... 61
8. Conclusion ....................................................................................................................... 63
  8.1. Summary of Results ................................................................................................... 64
REFERENCES ....................................................................................................................... 66
## LIST OF TABLES

**Table 4.1.** Instruction types in FabScalar and their execution latencies. ................................. 34
**Table 5.1.** Parameterized processor attributes in FabScalar................................................... 42
**Table 5.2.** Script call flow in FabScalar....................................................................................... 44
**Table 6.1.** Benchmarks, input arguments and run starting points................................................. 48
**Table 6.2.** Configuration of the processor......................................................................................... 49
**Table 7.1.** Performance (IPT) results in BIPS for all benchmarks run across all issue logic configurations tested. ................................................................................................................................ 50
**Table 7.2.** Cycle time of configurations A and B as a function of components of the issue logic.......................................................................................................................... 57
**Table 7.3.** Elimination of issue logic configurations by finding another configuration that always performs better............................................................................................................. 60
**Table 7.4.** Elimination of a narrow and deep configuration............................................................ 61
LIST OF FIGURES

Figure 2.1. Running a workload having serially-dependent chain of instructions on a machine with issue logic sub-pipelined into two stages................................................................. 12
Figure 2.2. Running a workload having serially-dependent chain of instructions on a machine with unpipelined issue logic.................................................................................. 13
Figure 3.1. Canonical Pipeline stages in FabScalar ................................................................................................................................. 22
Figure 4.1. Components of the Issue Logic ................................................................................................................................. 24
Figure 4.2. 4:1 select block, grantIn comes from the next level of the select tree................................. 28
Figure 4.3. Portion of a 16:1 select logic built out of 4:1 select blocks......................................................... 29
Figure 4.4. The issue logic shown with the wakeup-select loop highlighted ..................................... 32
Figure 4.5. 1/1 issue logic configuration................................................................................................................................. 37
Figure 4.6. 2/2 issue logic configuration................................................................................................................................. 38
Figure 4.7. 3/3 issue logic configuration................................................................................................................................. 39
Figure 4.8. 2/3 issue logic configuration................................................................................................................................. 40
Figure 7.1. IPT for vortex on the best core (over the course of the entire 10M run) and the best heterogeneous cores........................................................................................................... 53
1. Introduction

Superscalar processors fetch and execute multiple instructions each cycle. This is in contrast with scalar processors, which execute a maximum of one instruction per cycle - hence the qualifier 'super.' The move from scalar to superscalar processors to achieve higher instruction throughput is made possible by the independence that exists between instructions, which allows multiple instructions to execute in parallel (at the same time) without being incorrect. This parallelism present among instructions is known as instruction-level parallelism or ILP.

A modern out-of-order superscalar processor can be thought of as consisting of two parts, the front-end and the back-end. The front-end fetches multiple instructions in program order, renames the source and destination registers to eliminate false dependencies, and places instructions in the issue queue. The back-end polls the readiness of instructions in the issue queue, and selects multiple ready-to-execute instructions to be sent out of program order down the back-end pipeline stages where the instructions are executed. The final stage of the backend updates the architectural state of the machine in program order.

An instruction in the issue queue is said to be ready in a certain cycle if the instruction’s source operand values are available or expected to be available, so that issuing the instruction in that cycle (or later) results in correct execution. To achieve high instruction throughput (also called instructions per cycle or IPC), a large number
of instructions must execute each cycle, and hence, a large number of ready instructions must be available each cycle in the issue queue. Achieving this is challenging because data dependencies exist between instructions – dependent instructions will not become ready until the issue of instruction(s) producing its source operand values. To overcome the problem of data dependencies limiting available ILP and throttling instruction throughput, the front-end speculates past branches (via branch prediction). The issue queue can then build a large pool of instructions that enables looking far ahead in the program in search for independent (and hence, ready) instructions.

Instructions in the issue queue are polled for readiness each cycle and a sub-set of ready instructions are selected and granted permission to issue (the select process). These instructions are sent in parallel to the function units (the issue process). The issue logic also ensures that instructions in the issue queue become ready at the earliest possible time without compromising correctness (the wakeup process).

The part of the processor that handles the wakeup, select and issue processes is called the issue logic. This thesis deals with the issue logic and the effect of its configuration on program performance.

1.1. Issue Logic and Performance

The issue logic can be characterized by three parameters – issue width, issue queue size and issue depth:
a) Issue width is the maximum number of instructions that can be issued in a cycle.

b) Issue queue size is the maximum number of instructions that can be held by the issue queue.

c) Issue depth is the degree of pipelining of the issue logic. It is the minimum number of cycles an instruction has to spend in the issue stage.

The issue logic configuration for highest performance (defined as instructions per unit time or IPT) is a sweet-spot in the trade-off between IPC and processor frequency. Increasing the issue width and issue queue size allows the processor to exploit more ILP and increase IPC. It also entails increasing structure sizes and using longer wires, thus degrading frequency. Increasing the issue depth can provide a higher frequency but IPC may be degraded due to the extra sub-pipeline stages added. The relationship between issue logic configuration and performance is, therefore, a complex one.

Studying the exact nature of the above relationship is challenging, partly due to the definition of performance as IPT. Unlike IPC, which can be easily obtained by simulation on cycle-accurate simulators, timing information (i.e., frequency) is highly dependent on the exact physical design of the processor. Hence, estimation of performance of a single issue logic configuration for a specific workload requires a detailed design of that issue logic configuration. (Alternatives to this approach are discussed in chapter 3, Related Work.)
Furthermore, such a study would also require the ability to estimate performance for many issue logic configurations, and hence, require a library of detailed issue logic designs or a method to generate them automatically. This thesis follows the latter approach.

1.2. Contribution

a) This thesis aims to understand how workload characteristics, issue logic configuration and technology influence performance: For a given workload and technology, what factors cause one issue logic configuration to perform better than the other?

b) This thesis also explores program diversity within a benchmark suite in order to ascertain the performance benefit of customizing issue logic to program behavior: Is there enough diversity in workload characteristics so that the best issue logic configurations differ for different workloads? If so, how much do they differ in performance over the single best issue logic configuration for the entire workload set? What is the performance benefit of customizing issue logic to different program behavior in a heterogeneous multicore processor?

c) This thesis contributes FabIssue, a tool capable of generating synthesizable RTL designs for arbitrary issue logic configurations. The designs generated by the tool are used for both running benchmarks (gives IPC) and synthesis (gives frequency estimate).
The FabIssue tool has been developed as a part of the FabScalar [2] research project, which is a collaborative effort by many students over the past two years. Details of the FabScalar project are described in this thesis in order to bring out the context of this work, but parts not directly related to the work in this thesis are described only in brief.
2. Motivation

Performance in computer architecture is the average number of instructions committed per unit time (IPT). IPT is the product of average number of instructions committed per cycle (IPC) and frequency. Modern out-of-order superscalar processors achieve high IPC by exploiting instruction-level parallelism (ILP). Use of aggressive and accurate branch prediction to speculate past branches enables construction of a large instruction window that looks far ahead into the program in search of independent instructions that are issued and executed in parallel. Thus, going wider and larger translates to higher IPC. But this comes at the price of increased complexity, since each stage of the processor handles a larger number of instructions every cycle. There can be dependencies – data or control – among a group of instructions in a stage, and the hardware needed to handle this correctly brings in even more complexity. Thus, extracting higher IPC by going wider and larger entails the use of larger, more complex logic structures, which translates to larger logic depths and longer wire lengths, which in turn reduces the frequency at which the processor can be clocked. This effect of complexity on frequency can be mitigated to a large extent by sub-pipelining processor pipeline stages – in effect allowing a high frequency on a complex processor. However, pipelining also degrades IPC. Hence, there exists a trade-off between IPC and frequency in the quest for performance and the best design is one that finds the sweet-spot within this trade-off.
A program with no dependencies among its instructions would suffer no IPC reduction due to pipelining. However, dependencies – data and control – exist in real workloads. The precise reason for IPC degradation is the increase in the delay (in cycles) of critical loops due to pipelining and the effect this has on dependent instructions. A critical loop is a feedback path in the microprocessor in which flow of instructions into the loop is dictated by instructions that have entered the loop. The branch execute-fetch redirect loop (control dependencies) and the wakeup-select loop (data dependencies) are two examples. Pipelining a critical loop increases the delay (in cycles) between an instruction entering the loop and the instruction affecting the working of the loop. This increase manifests as IPC degradation.

For a processor with fixed width with constant structure sizes, the IPC vs frequency tradeoff is one of pipelining critical loops. From an IPC standpoint, critical loops are at best atomic – single-cycle loops – so that the number of instructions eligible to enter the loop is maximized. For example, a single cycle wakeup-select loop ensures that dependent instructions (consumer instructions) can be woken up and made eligible for selection the very next cycle a producer instruction is selected and issued. On the other hand, pipelining can boost performance by increasing clock frequency at the expense of atomicity. In the above example, the single-cycle wakeup-select loop can be timing critical – i.e., can dictate the maximum frequency – especially when a large issue queue is present, and hence pipelining can help. But the atomicity of the loop is now lost: Back-to-back execution of producer and
consumer instructions is not possible, which degrades IPC. This illustrates the trade-off that exists: How deep must critical loops be pipelined for the best balance between IPC and frequency?

The focus of this thesis is the issue logic – the part of the processor where the wakeup-select loop functions. The trade-offs that must be considered when deciding the best issue logic width, structure sizes and sub-pipelining depth are discussed in the following section.

2.1. Tradeoffs in Issue Logic Design

The issue logic of an out-of-order processor handles data dependencies correctly by ensuring that only ready instructions are issued for execution – instructions that are not ready are kept in the issue queue till they become ready to execute and are selected to issue. The issue logic can be defined by three parameters:

i) Issue Width: The maximum number of instructions that can issue in a cycle.

ii) Issue Queue Size: The maximum number of instructions that the issue queue can hold.

iii) Issue Depth: The degree of sub-pipelining of the issue logic.

This section examines qualitatively the effect of varying each of the above parameters on performance. The aim of the section is to understand the optimum issue logic configuration.
2.1.1. Issue Width

Issue width is number of instructions that can enter execution simultaneously in a cycle, and is considered the ‘width’ of the backend of the processor. The issue width is a direct measure of the ability of the processor to exploit ILP – a larger issue width means that more instructions can execute in parallel. Thus, higher issue width generally translates to better IPC (assuming the workload has enough ILP). It also causes lower frequencies due to increased logic complexity. The optimum issue width depends on the amount of ILP in the issue queue instruction window. A processor with too high an issue width can be said to have been designed to extract parallelism that does not exist at the cost of frequency, while one with a smaller-than-optimum issue width is incapable of extracting parallelism that would have led to higher performance, even at the resulting lower frequency.

2.1.2. Issue Queue Size

The issue queue size is the maximum number of instructions that can be held at a time in the issue logic. It is a direct measure of how far ahead into the program the processor can search for parallelism. The optimum issue queue size depends not only on branch prediction accuracy (that enables building large windows on the correct path) but also on the nature of distribution of dependent and independent instructions in the program. For a fixed issue width, a smaller-than-optimum issue queue size does not allow the processor to search far enough ahead into the program to get at the independent instructions. Increasing the issue queue size more than the optimum means that the branch prediction accuracy or the issue width
is too low to fill up the issue queue with useful instructions that can be used increase
IPC more than the accompanying degradation of frequency.

2.1.3. Issue Depth

Issue depth is the minimum number of cycles an instruction has to spend in the
issue stage. Issue depth is related to the delay (in cycles) of the wakeup-select loop
depending on where exactly the pipeline registers are placed. This sub-section
analyzes the effect of pipelining assuming a fixed issue width and issue queue size.

Understanding exactly how IPC is impacted on pipelining the wakeup-select loop –
also referred to in this thesis as ‘sub-pipelining the issue logic’ – is important in
deciding how much the issue logic be pipelined for best performance (IPT-wise) for a
given workload. The time (in seconds) taken by an instruction to traverse the issue
logic – the propagation delay – is independent of whether the wakeup-select loop is
sub-pipelined or not (neglecting pipeline latch latencies). This is because the actual
logic in the loop is unchanged by sub-pipelining: The propagation delay of the loop is
the sum of logic and wire delays of each of the sub-pipeline stages of the loop. A
hidden assumption here is that the issue logic is the timing critical path in the
microprocessor, so that sub-pipelining it enables clocking at higher frequency. Even
though the propagation delay does not change, sub-pipelining allows hiding this
delay by overlapping the latency experienced by an instruction with other
instructions. If the wakeup-select loop is filled with instructions and the workload is
such that there are always instructions available to be selected and issued, IPC will
not be impacted by sub-pipelining. In this ideal case, sub-pipelining is beneficial: The microprocessor designer sees the increase in propagation delay of the wakeup-select loop that came about from going wider fully mitigated by pipelining. The propagation delay is essentially hidden and does not affect performance of this workload.

At the other extreme, a workload consisting of a serial dependence chain would not see any benefit in sub-pipelining the issue logic. In such a workload, an instruction issues and wakes up its dependent instruction only after traversing the entire loop, and the propagation delay is not hidden by overlapping instructions (Scenario 1). In this case, the propagation delay is said to be exposed, and will be a factor that determines the performance of this workload. Now if the pipeline register sub-pipelining the issue logic is removed (Scenario 2), IPC rises but performance (IPT) will be the same as Scenario 1.

Figure 2.1 and figure 2.2 illustrate the above two scenarios. The x-axis is time, and y-axis, instructions. The workload is a serially-dependent chain of instructions – i.e., instruction B depends on A, C depends on B, D depends on C etc. Figure 2.1 illustrates the running of this workload on a machine having the issue logic sub-pipelined into two stages, Issue_1 and Issue_2. The data dependency between adjacent instructions cause an instruction to issue only after its preceding instruction has completed Issue_2.
| Cycles -> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Instr. A  | Fetch | Decode | Rename | Dispatch | Issue_1 | Issue_2 | RegRead | Execute | Writeback | Retire |
| Instr. B  | Fetch | Decode | Rename | Dispatch | Issue_1 | Issue_2 | RegRead | Execute | Writeback | Retire |
| Instr. C  | Fetch | Decode | Rename | Dispatch | Issue_1 | Issue_2 | RegRead | Execute | Writeback | Retire |
| Instr. D  | Fetch | Decode | Rename | Dispatch | Issue_1 | Issue_2 | RegRead | Execute | Writeback | Retire |

**Figure 2.1.** Running a workload having serially-dependent chain of instructions on a machine with issue logic sub-pipelined into two stages. The propagation delay is not hidden by pipelining the issue logic.

Figure 2.2 (scenario 2) has the same workload and machine, except that the pipeline register that sub-pipelined the issue stage is removed – the machine has a single-cycle wakeup – and hence has twice the cycle time as the one in figure 2.1.
Figure 2.2. Running a workload having serially-dependent chain of instructions on a machine with unpipelined issue logic. Note that the x-axis represents time; the time period is twice that of figure 2.1.

Over a period of time, the pipelined and un-pipelined scenarios of figure 2.1 and figure 2.2 have the same performance (neglecting pipeline latch delays and pipelining imbalances). The argument here is that scenario 1 did not benefit from pipelining (and higher clock frequency) due to the nature of the workload. If pipeline latch delays and pipelining imbalances are also considered, scenario 2 (unpipelined) would have a higher performance.

Instructions in the issue queue of a real workload are a complex mixture of independent, ready instructions and dependent, not-ready instructions. Hence, whether or not pipelining hides the propagation delay depends on whether
instructions enter the wakeup-select loop to overlap and hide the propagation delay. The exact nature the impact on performance due to sub-pipelining the issue logic can now be seen: If sub-pipelined too deeply, instructions will be unable to fill up the wakeup-select loop and hide the propagation delay completely. If not sub-pipelined deep enough, the processor is being underclocked for the amount of parallelism the workload possesses.

2.2. Heterogeneity and Core Customization

Program behavior varies widely. Benchmarks used to represent real-world programs differ in the amount of parallelism available and memory access patterns. This variation is seen not just among benchmarks but even between phases within a benchmark. Processor design is about finding the sweet-spot between various trade-offs under given constraints, and the variation in program behavior forces the best core design to the ‘average core’ – the core that performs best on an average across different program behaviors, or simply, across all benchmarks chosen to represent real-life programs that the processor intends to run. The core is also designed to ensure that the performance of a certain program behavior or a certain benchmark is not degraded terribly – forcing the best design to be large and beefy, capable of accommodating even program behaviors that demands lots of hardware resources for reasonable performance. This core is, thus, a compromise.
With Moore’s Law adding transistors to the chip exponentially and a growing power problem, processors have moved towards chip multicores to in search of performance. The philosophy of designing cores has remained relatively the same – the one-size-fits-all core is replicated on the chip to execute different threads or programs in parallel. Heterogeneous multiprocessors (HMPs), i.e., designing cores on a chip multicore differently and customizing each core to different program behaviors, also called asymmetric cores in literature, is a promising concept for extracting performance. The advantage of this organization is that each core is not a compromise over the entire range of program behavior, but one over a certain subset of it – with the sum total of all heterogeneous cores capable of handling all program behavior reasonably well. Now that the range of program behavior each core is expected to handle is smaller, less has to be compromised in the design of each core, and therefore can achieve a higher performance for that program behavior than the ‘average core’. Each core is still an average core – but each core is averaged over a smaller, different subset of all program behavior.

An open question in core customization is how different would the cores of a HMP be from each other. In superscalar processors, hardware complexity is used to extract IPC and pipelining is employed to mitigate the effect of the complexity on frequency. Therefore, it is conceivable that the best core for most or all program behaviors is a complex, very wide, large and deeply pipelined core – i.e., the beefy core. Kumar et al. [1] showed that when area or power budget is constrained, the
best set of cores tends to be heterogeneous. The tighter the constraint the more heterogeneous the cores become. Thus, area or power constraints act as forcing functions for heterogeneity. The argument in this thesis is that depending on program behaviors which expose the critical loops differently, propagation delay can be a forcing function for heterogeneity – even if area and power budget is unconstrained; and that the beefy core is not the best core for all program behavior.

The motivation for the argument that all workloads may not favor a single beefy core is elaborated here: The inference from the analysis in section 2.1 is that the best issue logic configuration is dependent very much on the nature of the workload – how the dependent chains and independent instructions are distributed within the workload. There are two scenarios in which the propagation delay of the wakeup-select loop is exposed:

a) Lack of ready instructions in the issue queue, as described in section 2.1.3.

b) Processor flushing events such as exceptions, load-violations or branch mispredictions (all of these are handled the same way in this thesis). The processor waits until the offending instruction reaches the head of the Active List (or Re-Order Buffer) and the entire processor is flushed. The flush clears out all instructions in the wakeup-select loop, and exposes the propagation delay, transiently till the loop is filled again.

There are two points to note:
i) Irrespective of how much the wakeup-select loop is pipelined, the propagation delay is (almost) the same and hence, the impact of exposing this critical loop is the same.

ii) The propagation delay is highest in the beefiest core, since this core is very wide. Hence, the penalty of exposing the critical loop is highest in the beefiest core.

The take-away from the above arguments is that the beefiest core (irrespective of nature of sub-pipelining of the wakeup-select loop) need not have the best issue logic for all workloads, especially if a) the workload lacks ILP or b) exceptions, load violations and branch mispredictions are common enough so the propagation delay (measured in seconds) is one of the major factors determining the performance (inverse of the total time in seconds) of a workload.

2.3. FabIssue

Determining the best issue logic for a workload (section 2.1) is challenging, because the dynamic window of instructions of workloads is a complex mixture of dependent and independent instructions. It is difficult to characterize a workload by static code information and predict its preferred issue logic configuration. Furthermore, flushes of the processor are difficult to anticipate without actually running the workload. A simpler approach is to simulate the workload on differently pipelined issue logic and pick the one giving the best performance. However, the number of issue logic designs is large. This analysis also requires not just IPC information (easily got by
simulation of workloads in cycle-accurate simulators) but also clock frequency estimate.

Studying propagation delay of the issue logic as a forcing function for heterogeneity (section 2.2) is also challenging because it requires searching through a large design space – designs that model propagation delay – to find the best designs for different program behavior. If given the design space and a suitable design space exploration technique, the designs of the cores customized to different program behavior could then be used to construct a hypothetical heterogeneous core with highest performance for the workload set, and calculate the (maximum possible) performance benefit of heterogeneity.

What is required, thus, is a tool to generate issue logic designs based on input parameters capable of providing timing information and IPC quickly. This was the motivation to develop the FabIssue tool – a component of the FabScalar [2] toolset that generates synthesizable RTL designs of issue logic of desired configuration – as a part of this thesis. There are alternatives to this approach; they are discussed in chapter 3, related work.
3. Related Work

3.1. Issue Logic

[15] studies the limits of extracting performance from ILP. An experiment in this limit study varies the issue queue size (referred to in [15] as the continuously-managed scheduler window size) and discovers that an issue queue size of 32 is sufficient to extract ILP (i.e., get a high IPC) for the workloads considered. This thesis uses IPT as a performance metric instead of IPC, and attempts to discover the best issue logic configuration (issue width, issue depth and issue queue size) for different workloads and phases within workloads.

[3] and [4] look at pipelining the issue logic without sacrificing atomicity of the wakeup-select loop. But these techniques involve speculative instruction issue, and incur a large penalty (in number of cycles) when speculation is incorrect. This thesis does not deal with speculative instruction issue.

3.2. Design and Performance

The performance gain that can be obtained by matching core design to program behavior has been explored in literature [16]. This thesis concentrates on the issue logic and matches issue logic configuration to program behavior. The configuration of the rest of the processor is kept fixed.
Kumar et al. [1] studies the benefit of heterogeneity in multicores when constraints on area and power are applied and finds that the best set of cores are more heterogeneous when area or power budget constraints are made tighter. This thesis does not consider area or power budgets – instead asking the question, “Can propagation delay of the issue logic itself cause the best set of cores to be heterogeneous?”

3.3. Simulators

Cycle accurate processor simulators are perhaps, the computer architect’s most important window into the details of what exactly happens in a microprocessor. There exists a spectrum of simulators from simple functional simulators [5] to ones that model microarchitecture in detail [6], and depending on the details modeled, run workloads to provide a wealth of runtime statistics, including IPC. These simulators can also be easily made to model any processor configuration – it may be as simple as changing a variable value in the simulator – and is valuable in studying heterogeneity. They are usually written in C/C++ for speed and efficiency.

A drawback of C/C++ simulators is their inability to model timing – i.e., generate an estimate of the clock frequency – as well as other technology-dependent parameters like area and power. However, simulators can be used in conjunction with analytical models [8], [9] that are capable of predicting technology-dependent parameters. Performance is then estimated by IPC information from the simulator and frequency
estimates from the analytical model. The accuracy of this approach is limited by the level of detail of the analytical model. Even if the model is highly detailed, it is debatable how well the model holds up once the technology changes. To be certain, an analytical model would have to be recalibrated and re-validated for every technology change.

An alternative to using cycle accurate simulators is to use RTL simulators. These are written in a hardware description language like Verilog or VHDL, and by its very nature, model microarchitectural features to a very fine detail. A processor configuration under study could be synthesized on any technology (all that is needed is the technology library file) and an estimate of the clock frequency could be obtained. The Illinois Verilog Model [11] is an attempt to create such a simulator. However, it has been shown that the IVM is at times poorly synthesizable [12]. RTL simulators also suffer from being slower in simulation time than a C/C++ simulator. Furthermore, since the simulator is specified in RTL, it may be difficult to re-configure the simulator to run another processor configuration – it may not be as simple as changing a variable in the simulator. FabScalar [2] attempts to overcome the above drawbacks and is described in detail in the following sub-section.

3.4. FabScalar

FabScalar enables automatic generation of synthesizable register-transfer-level (RTL) designs of superscalar cores of desired structure sizes, pipeline widths and
degree of sub-pipelining of pipeline stages. FabScalar views a superscalar processor as a set of canonical pipeline stages that are stitched together to create the processor. The flexibility to create arbitrary cores lies in the ability to configure each pipeline stage in any way desired as long as the pipeline stage’s interface and functionality is maintained. Figure 3.1 shows the canonical pipeline stages in FabScalar.

![Figure 3.1. Canonical Pipeline stages in FabScalar](image)

Using FabScalar, answering the question, “What performance (IPT) does a workload achieve on a core X” is answered very easily and quickly. The tool can be instructed to generate the RTL of core X, which is synthesized to obtain an estimate of the frequency. The workload can be run on a fast simulator – also part of the FabScalar toolset – which is configured to match the generated RTL at cycle-by-cycle precision to generate the IPC. Hence, workload the performance (IPT) on the core is obtained.

The RTL generated by FabScalar is intended to be synthesizable and provide a reasonable estimate of clock frequency. A superscalar processor has many memory structures, RAMs and CAMs that are not constructed out of logic gates or latches (structures that Verilog can describe). Hence, included in the FabScalar toolset is FabMem [13], a detailed, automatic memory compiler that can produce the
schematic, area, power and timing estimates of RAM and CAM structures. In order to estimate clock frequency, the entire processor RTL generated by FabScalar scripts (minus memory structures) is synthesized – the memory structures are synthesized as black-box modules whose timing characteristics are got from FabMem. Verilog descriptions of memory structures – RAMs and CAMs modeled using flip-flops and logic gates – are also available in FabScalar in order to functionally validate the RTL design by running workloads.

It is important to note that the frequency estimated by this approach – just like the one obtained by an analytical model – may not absolutely match a fabricated processor of the same configuration. This is because the layout generated by synthesis and place-and-route tools does not involve the manually-done pre-fabrication fine tuning that processors undergo, where designers tweak critical paths by layout- and transistor-level alterations. However, automatic generation of processor designs requires no design effort once the tool is in place. In the design effort vs. accuracy trade-off seen here, FabScalar sacrifices some accuracy for zero design effort. Even so, this approach using synthesis is still useful in understanding how changing microarchitecture design affects IPC and frequency.
4. Design of the Issue Logic

FabIssue, the set of scripts in FabScalar that deals with issue logic generation, has an in-built issue logic design template. Depending on the input arguments to FabIssue, it outputs the RTL of the desired issue logic design based on the template. This chapter describes the template issue logic design.

4.1. Working of the Issue Logic

Figure 4.1 shows the components of the issue logic. The working of the issue logic is broken down and described in the sub-sections below with respect to figure 4.1.

Figure 4.1. Components of the Issue Logic
4.1.1. Dispatching an Instruction

When an instruction packet is dispatched from the front-end, the source physical registers (and some control information) are stripped off from the packet and written into the wakeup CAM. The rest of the packet is written into the payload RAM. The reason for this division is that the entire instruction packet is not required for the wakeup-select process – the source registers are sufficient. Hence the rest of the packet placed in the payload RAM is read only when the instruction issues. The number of entries in the wakeup CAM and the payload RAM are the same (called issue queue size). An instruction is dispatched into the same entry location in the CAM and RAM. This location, i.e., the issue queue entry at which the instruction is dispatched into, is decided during the dispatch process by popping out a free (unused) issue queue entry from the issue queue free list, a structure that maintains a list of free entries in the issue queue.

4.1.2. Generation of Request Vectors

In addition to source physical register tags, an instruction also keeps certain control bits (not in the CAM or RAM, but in flip-flops). These are:

a) **Valid bit**: A bit specifying if this issue queue entry is valid (being used).

b) **Source valid bits**: Two bits, one per source operand, specifying if the instruction has a source operand. Instructions can have 0, 1 or 2 valid sources.
c) *Source ready bits:* Two bits, one per source operand, specifying if the source operand is available (or will be available when the instruction reaches the function units for execution if issued now). These bits are initialized during instruction dispatch by polling the physical register file valid bits. The source ready bits are set in the wakeup process.

d) *Scheduled bit:* A bit specifying if this instruction has already been issued or selected to be issued.

An issue queue entry that is valid, has all its valid source operands in the ready state and has not been scheduled is said to be ready to issue. Ready instructions raise the request line corresponding to its issue queue entry to let the select logic know that it can issue now.

### 4.1.3. Select Logic

The select logic is priority logic that takes in all request vectors and generates a one-hot grant vector. The grant vector lines feed the payload RAM read bit-lines, causing the granted instruction’s payload to be read out and sent down the pipeline. The grant vector also sets the entry’s scheduled bit.

Instead of monolithic priority logic block, the select tree is composed of smaller select blocks, which when stitched together, functions like the monolithic priority logic. With this approach, the select blocks can be manually designed and fine-tuned at the layout level to be fast, and a fast select logic of any required size can be
composed using these select blocks. In this thesis, no fine tuning is done on the select blocks, but the select logic is indeed constructed using select blocks (this gives the flexibility of replacing select blocks generated by synthesis with manually designed select blocks).

The select logic (also called select tree) is constructed using select blocks by connecting them in a tree-like arrangement, with each successive level of the tree having lesser select blocks than the previous, and the final level having one select block. Figure 4.2 shows the logic in the 4:1 select block; this block has a 4-bit vector input and a 4-bit, one hot vector output grantOut. grantOut signals function as the grantIn signals to the select blocks that constitute the previous level of the select tree. The select blocks also output requestOut, a signal that feeds a requestIn of the select block in the next level of the select tree. The grant vector is the grantOut signals of the first level of select blocks.
All select blocks (except the ones in the first level) choose between select blocks of the previous level by sending grant signals to exactly one of them (if this select block received grantIn from the next level). The first level select blocks uses the request vector as input. It is thus, a hierarchical selection process.

In Figure 4.3, part of a 16:1 select logic built out of 4:1 select blocks in a two level fashion is shown. Request signals flow down the select tree (towards higher levels) till the final stage of the select tree. grantOut signals flow up the select tree (towards lower levels) to form the grant vector. The selection policy is static – a lower issue queue entry gets higher priority during the selection process.
Figure 4.3. Portion of a 16:1 select logic built out of 4:1 select blocks. req0 to req3 form the first 4 bits of the request vector and grant0 to grant3 are the first 4 bits of the final grant vector.

The select block at the final level of the select tree is given a grantIn signal that is always held high. This is to mean that the function units are always ready to accept
an instruction once every cycle, i.e., either the function units either have a latency of one cycle or are fully pipelined.

4.1.4. Wakeup CAM

An issuing instruction broadcasts its destination physical register tag to all instructions in the issue queue. Each instruction in the issue queue compares the broadcasted tag to its source physical register tags, and on a match, sets the corresponding source ready bit. This is the wakeup process. The comparison and matching is handled by the wakeup CAM, which compares all its entries with the tag placed at its read port (i.e., the wakeup port), and raises a set of match lines (which feed the source ready bits). Since there are a maximum of two sources per instruction, there are two wakeup CAMs, one per source register.

4.1.5. Result Shift Register

Instructions must be woken up as early as possible to maximize IPC. However, an instruction woken up too early would reach the execute stage and finds that a source register value is unavailable (not yet produced) and hence, cannot execute and generate the correct result. Ideally, the instruction should be woken up so that if it issues immediately after waking up, it finds that its source register value has just been produced as it enters a function unit for execution. The value is then got via a bypass from the outputs of the function units to the inputs of the function units. Hence, the time interval (in cycles) between the value producer instruction issuing
and the value consumer instruction issuing must be, at least, the number of cycles the producer instruction takes to execute.

The result shift register (RSR) enforces this minimum required delay between issuing of the producer instruction and broadcast of its destination tag to wakeup dependent instructions. When an instruction issues, it pushes its destination tag into the tail of the RSR. The RSR shifts tags, once every cycle, towards its head, and broadcasts the tag at its head to the CAM wakeup ports. Thus, the required delay between issue and tag broadcast is got by fixing the number of registers in the RSR, which is determined by the execution latency of the issuing instruction. If there are instruction types differing in execution latencies, different RSRs are used for to each instruction types.

4.1.6. Wakeup-Select Loop

Figure 4.4 highlights the wakeup-select loop in the issue logic. The wakeup-select loop starts at the ready bits of the instructions in the issue queue. The ready bits contribute to the generation of the request vector, which feeds the select logic. The grant vector generated by the select logic is used to read out the instruction’s payload from the payload RAM. One piece of information read from the payload RAM, the instruction’s destination physical register tag, is pushed into the RSR. When the tag reaches the head of the RSR, it is fed into one of the read ports of the wakeup CAM (a wakeup port). The CAM compares this tag with the source physical register tags in each of its entries and on a match, raises the match line. The match
lines set the ready bits of the corresponding source. This completes the loop from ready bits to ready bits.

**Figure 4.4.** The issue logic shown with the wakeup-select loop highlighted

For best IPC, the wakeup process (setting of the ready bits) must be done so that value consumer instructions are woken up ‘n’ cycles after the value producer instruction issues, where ‘n’ is the execution latency of the producer instruction. This way, the consumer instruction (if selected) will follow the producer instruction down the pipeline at a distance ‘n’ (measured in number of pipeline stages in between the instructions, which is equivalent to number of cycles separating them). When the consumer instruction reaches the function unit, its source value will have just been produced by the producer instruction.
For instructions with a single-cycle execution latency, consumer instructions have to be woken up the very next cycle the instruction issues. This is possible only if the wakeup-select loop is unpipelined, i.e., there are no pipeline registers in the feedback path from ready bits to ready bits and the RSR latency is zero (that is, RSR broadcasts tags the very cycle it receives the tags). For instructions with a 2-cycle execution latency, the RSR consists of a single register, so that tag broadcast starts at the next cycle after issue, and wakeup happens two cycles after issue of the instruction. An instruction with an n-cycle execution latency will use an RSR consisting of (n-1) shift registers.

An exception to the above wakeup-select process is the load instruction. Speculatively waking up load-dependent instructions (with no mechanism for instruction replay in the issue logic) would be problematic if the load misses in the L1 cache: Dependent instructions reaching function units would find that its source operand value is not available. Hence, loads have to be handled differently. “Execution latency” of load instructions is unpredictable, and hence, when a load issues, the exact number of cycles to wait before waking up its dependents is not fixed. Therefore, loads do not broadcast destination register tags using the RSR. Once a load executes and gets its value from the cache hierarchy, it broadcasts its tag from the writeback stage and wakes up its dependents.
4.1.7. Free List Management

The free list keeps track of the used and free issue queue entries. When an instruction is to be dispatched to the issue queue, a free issue queue entry number is popped out from the free list and the instruction is written into that entry. When an instruction is selected, the instruction’s issue queue entry is written into the free list. The actual freeing process (setting the issue queue entry’s valid bit zero) may take multiple cycles, and hence, in the interval between selecting an entry and resetting its valid bit, the scheduled bit acts as the guard to prevent the entry from requesting for selection again.

4.1.8. Cascaded Select trees

The backend of FabScalar processors have dedicated pipeline ways for different instruction types. An instruction of a certain type can execute only in a pipeline way of its instruction type. The instruction types are given in Table 4.1.

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Execution Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Instructions</td>
<td>1</td>
</tr>
<tr>
<td>Complex Instructions</td>
<td>3</td>
</tr>
<tr>
<td>Control Transfer Instructions</td>
<td>1</td>
</tr>
<tr>
<td>Loads/Stores</td>
<td>2 (hit)</td>
</tr>
</tbody>
</table>

Four request vectors (one corresponding to each instruction type) are generated and fed to four select trees. If the machine has an issue width of four (i.e., four pipeline ways, one per instruction type), each of the four select trees generates a single grant
vector, which grants each selected instruction, access to one pipeline way (of its type) to issue. But in a 5-wide issue machine (2 simple instruction pipeline ways, and one pipeline way each for the other three types), the select logic processing requests of simple instructions takes in a single request vector but outputs two distinct one-hot grant vectors (one for each simple instruction pipeline way). Such a select logic behaves like two cascaded select trees: the first works as a normal select tree, while the second uses a masked-out request vector as its input, with the request line corresponding to the entry selected by the first tree masked out. This ensures that the two trees generate two distinct one-hot grant vectors. The idea is extended to an n-cascaded select tree if there are n pipeline ways of an instruction type.

Select trees can be placed in series to generate a cascaded select tree (the output of a tree is used to mask the request vector it received, and then sent to the next select tree’s request vector input). This approach lets synthesis take care of the complexity of designing the cascaded trees. Alternatively, select trees are overlapped at the level of select blocks to create the cascaded select tree. Since synthesis is done after flattening the RTL design, the two approaches are seen to produce similar final designs for the cascaded select tree.
4.2. Pipelining the Issue Logic

Instructions with multiple-cycle execution latencies suffer no IPC degradation due to pipelining the wakeup-select loop. This is because such instructions take multiple cycles to wake up consumers (reflected in the number of registers in the RSR used by these instructions) and hence any extra delay (in cycles) in waking up consumers due to sub-pipelining registers in the wakeup-select loop can be offset by reducing the registers in the RSR correspondingly. For instance, a complex instruction with an execution latency of 10 cycles uses an RSR consisting of 9 shift registers if the wakeup-select loop is unpipelined, and an RSR consisting of 7 shift registers if the wakeup-select loop is pipelined into 3 parts (two pipeline registers).

Hence, IPC degradation due to loss in atomicity of the wakeup-select loop affects only instructions having an execution latency smaller than the loop depth (in cycles) of the wakeup-select loop. But simple single-cycle instructions form a large portion of integer workloads; hence IPC is markedly degraded for such workloads when the loop is sub-pipelined.

The issue logic configurations described in this section here onwards are for single-cycle execution latency instructions.

4.2.1. The 1/1 Issue Logic Configuration

The issue logic is unpipelined in the 1/1 issue logic configuration (figure 4.5). In this notation of “1/1”, the first “1” refers to the number of pipeline registers encountered
when traversing the wakeup-select loop once; and the second “1” refers to the number of pipeline registers encountered by an instruction during the entire issue process. In terms of number of cycles, 1/1 means that consumer instructions are woken up one cycle after the producer instruction issues, and that instructions spend a minimum of one cycle in the issue stage. At least one pipeline register is necessary in the wakeup-select loop, without which a combination feedback loop is set up. Here, the ready bits act as a pipeline register between the CAM and the select logic.

**Figure 4.5.** 1/1 issue logic configuration. Note that there is no RSR, since select and payload read and wakeup have to happen in the same cycle.

### 4.2.2. The 2/2 Issue Logic Configuration

The logical places to add pipeline registers to sub-pipeline the issue logic are between the select logic and payload RAM, and between the payload RAM and the
wakeup CAM. The 2/2 configuration (figure 4.6) adds a pipeline register between the payload RAM and the wakeup CAM by adding a shift register to the RSR.

Figure 4.6. 2/2 issue logic configuration. Note that the RSR is a single register. The pipelining splits the issue logic into wakeup and select-payload read.

4.2.3. The 3/3 Issue Logic Configuration

This configuration adds a pipeline register in between the select logic and the payload RAM. The grant vector produced by the select tree is used without passing through the pipeline register to set the scheduled bits, and is used after passing through the pipeline register to read out of the payload RAM.
Figure 4.7. 3/3 issue logic configuration. Wakeup, select and payload read now happen in separate cycles for an instruction.

4.2.4. The 2/3 Issue Logic Configuration

The reason that the three configurations discussed above have the same wakeup-select loop depth and issue stage depth is that the dependent instruction’s wakeup process needs the issuing instruction’s destination physical register tag which is stored in the payload RAM. Hence, an instruction’s issue and the waking up of its dependent instructions can both happen only after payload RAM read. However, if the instruction’s destination tag is available before the payload read, the wakeup process can begin immediately after the selection process is completed. This can be achieved by modifying the select logic to accept destination tags of instructions along with the request vector, so that the tags can be multiplexed down the select tree (in parallel with the selection process) and that the select tree produces the selected instruction’s destination tag along with the grant vector. A structure to hold
destination physical register tags which feeds the select tree – written to when an instruction is dispatched – is also required.

**Figure 4.8.** 2/3 issue logic configuration. The destination physical register tags are now got from the select tree instead of the payload RAM. The additional structure that holds the destination tags (and feeds the select tree) is written to during dispatch.

### 4.2.5 Pipelining the Select Logic

The select logic cannot be pipelined, because it would violate atomicity of the loop that sets the scheduled bits (which prevent an already granted instruction from requesting). That is, the loop starting at the scheduled bits, which contributes to the generation of request vectors, feeding the select logic to produce the grant vector used to set the scheduled bits – has to happen in one cycle. If not, an instruction would raise its request line for multiple cycles even if it was granted the first cycle it
requested – leading to the possibility that an instruction is granted multiple times. Hence, the select logic is not pipelined – this is seen to have important repercussions on frequency, and is discussed in the chapter 7, results.
5. FabIssue

FabIssue is the set of scripts in the FabScalar toolset that generates desired issue logic designs. This section describes all scripts in FabScalar, and points out the ones that are part of FabIssue.

Given the enormous complexity of generating arbitrary, functionally correct and synthesizable cores, and the large number of parameters that can be varied in superscalar cores, FabScalar divides processor attributes into two classes, which are dealt with differently:

i) Sizes of structures within the processor

ii) Pipeline stage widths and sub-pipelining depths within pipeline stages

The first class, structure sizes, is handled within the RTL description by parameterizing all structure sizes. Changing the value of the parameter that describes the size of a structure (in the top level file where it is defined) is all that is needed to create a different processor. Table 5.1 lists the attributes in the processor which are parameterized.

Table 5.1. Parameterized processor attributes in FabScalar. The parameters that affect the issue logic are highlighted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Parameterized Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Target Buffer Size</td>
<td>Load Store Queue Size</td>
</tr>
<tr>
<td>Branch Predictor Table Size</td>
<td>Instruction Buffer Size</td>
</tr>
<tr>
<td>Return Address Stack Size</td>
<td>Execution latency of Instruction types</td>
</tr>
<tr>
<td>Control Transfer Instruction Queue Size</td>
<td>Number of Issue Queue Entries</td>
</tr>
<tr>
<td>Physical Register File Size</td>
<td>Rename Map Table Size</td>
</tr>
<tr>
<td>Active List Size</td>
<td></td>
</tr>
</tbody>
</table>

42
The second class of parameters is handled by scripts in FabScalar: Scripts (written in Perl) that take in the widths and sub-pipelining depths of individual pipeline stages as input and generates the RTL description (Verilog) of the processor. These scripts function in a hierarchical manner: A single top-level script handles generation of the entire processor and calls second-level scripts that handle generation of one pipeline stage each; each second-level script calls third-level scripts to generate the Verilog files in its pipeline stage. There is a 1:1 correspondence between third-level scripts and Verilog files describing the processor, i.e., each third-level script generates one Verilog file. Finally, third-level scripts may invoke module scripts, which generate Verilog description of modules that have been instantiated within the Verilog file generated by the third-level script. For example, IssueQSelect.pl, which generates the select logic, calls SelectBlock.pl, which defines the select block module (the select logic is composed of select blocks instantiations stitched together). The call flow of the scripts is shown in Table 5.2. The highlighted scripts constitute FabIssue.
Table 5.2. Script call flow in FabScalar. The scripts that are part of FabIssue are highlighted.

<table>
<thead>
<tr>
<th>Top-Level Scripts</th>
<th>Second-Level Scripts</th>
<th>Third-Level Scripts</th>
<th>Module Scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch.pl</td>
<td>FetchStage1.pl</td>
<td></td>
<td>SRAM_v.pl</td>
</tr>
<tr>
<td></td>
<td>Fetch1Fetch2.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FetchStage2.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BranchPrediction.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BTB.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CTI.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1i.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAS.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SelectInst.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch2Decode.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decode.pl</td>
<td>Decode.pl</td>
<td></td>
<td>RAM.pl</td>
</tr>
<tr>
<td></td>
<td>PreDecode_PISA.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decode_PISA.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>InstBufRename.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>InstructionBuffer.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rename.pl</td>
<td>Rename.pl</td>
<td></td>
<td>FREELIST_RAM.pl</td>
</tr>
<tr>
<td></td>
<td>RenameMapTable.pl</td>
<td></td>
<td>RMT_RAM.pl</td>
</tr>
<tr>
<td></td>
<td>SpecFreeList.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RenameDispatch.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dispatch.pl</td>
<td>Dispatch.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>issue.pl</td>
<td>IssueQueue.pl</td>
<td></td>
<td>CAM.pl</td>
</tr>
<tr>
<td></td>
<td>freeList.pl</td>
<td></td>
<td>DecodedMux.pl</td>
</tr>
<tr>
<td></td>
<td>cascadedIssueQSelect</td>
<td></td>
<td>Encoder.pl</td>
</tr>
<tr>
<td></td>
<td>IssueQSelect.pl</td>
<td></td>
<td>PriorityEncoder.pl</td>
</tr>
<tr>
<td></td>
<td>issueqRegRead.pl</td>
<td></td>
<td>PAYLOAD_RAM.pl</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RSR.pl</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>selectBlock.pl</td>
</tr>
<tr>
<td>regread.pl</td>
<td>RegRead.pl</td>
<td></td>
<td>SRAM_PRF.pl</td>
</tr>
<tr>
<td></td>
<td>RegReadExecute.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>execute.pl</td>
<td>AgenLsu.pl</td>
<td></td>
<td>AGEN.pl</td>
</tr>
<tr>
<td></td>
<td>Execute.pl</td>
<td></td>
<td>Complex_ALU.pl</td>
</tr>
<tr>
<td></td>
<td>ForwardCheck.pl</td>
<td></td>
<td>Ctrl_ALU.pl</td>
</tr>
<tr>
<td></td>
<td>fu0.pl</td>
<td></td>
<td>Simple_ALU.pl</td>
</tr>
<tr>
<td></td>
<td>fu1.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fu2.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fu3.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>writeback.pl</td>
<td>WriteBack.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retire.pl</td>
<td>ActiveList.pl</td>
<td></td>
<td>AMT_RAM.pl</td>
</tr>
<tr>
<td></td>
<td>ArchMapTable.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory.pl</td>
<td>LSU.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1DataCache.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CommitLoad.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CommitStore.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DispatchedLoad.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DispatchedStore.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fabscalar.pl</td>
<td>FABSCALAR.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interface.pl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ISA.pl</td>
</tr>
</tbody>
</table>
6. Methodology

6.1. Experiments

The purpose of the experiments conducted in this thesis is to ascertain the performance improvement from customizing issue logic to workloads. Hence, the FabIssue tool is used to generate a range of cores which differ only in issue logic configuration. All workloads are run on every generated core and performance (in IPT) is tabulated. A hypothetical best multicore is constructed, consisting of the best cores for each workload. The performance of the workload set on this hypothetical heterogeneous multicore is compared with a homogeneous multicore consisting of replicated copies of the single best core across all workloads (i.e., the average core). The performance difference is the performance benefit of issue logic heterogeneity quantified.

The above experiment is repeated at the granularity of phases within a benchmark. The idea of core heterogeneity is to customize cores to program behavior, and not to customize cores to benchmarks. Benchmarks typically have varying behavior in the course of its run. This experiment generates IPT results for short segments within a benchmark (short enough to be approximated to be a “phase” – i.e., a section of code with unchanging program behavior). A full analysis of the relationship between program behavior and core configuration requires understanding exactly why certain phases choose certain core configurations, and is beyond the scope of this thesis.
However, this experiment is intended to quantify the maximum performance gain from customization to phase behavior, even if only issue logic is varied.

6.2. Assumptions and Limitations

This study tries to isolate the effect of varying issue queue configurations from the rest of the processor. The metric used to evaluate a configuration is IPT. Hence, the processor is configured to make IPC dependent on the issue logic as much as possible (i.e., the issue logic is made the IPC bottleneck); and frequency is assumed to be dictated by the issue logic.

To ensure that issue logic configuration is the IPC bottleneck, all other structures in the processor are made large. The front-end width (fetch, decode, rename and dispatch widths) is made very wide. Retire width is retained at four, but retire width is confirmed to not be the bottleneck because (a) average IPC of any benchmark or phase is never greater than four and (b) the active list is made large enough to ensure that the active list never causes stalls due to a limited retire width.

There are certain restrictions and limitations in the current FabIssue tool with respect to the mixture of function units that can be automatically generated:

a) The number of instruction types is fixed at four. Since each pipeline way in the backend is dedicated to executing an instruction type, the minimum number of pipeline ways in the backend, and hence, issue width, is four.
b) The script infrastructure currently allows adding extra pipeline ways of the simple instruction type only. A machine with a six-wide issue logic can only have a backend configuration of (3,1,1,1), which stands for three simple instruction pipeline ways, one complex instruction pipeline way, one control instruction pipeline way and one memory instruction pipeline way. A five-wide issue machine has the backend configuration (2,1,1,1). However, this is not a very serious restriction – since the workloads run are integer benchmarks where the majority of instructions are of the simple instruction type.

Caches are not modeled in the FabScalar toolset and hence, all load instructions are hits. However, when a load issues, the processor does not assume a cache hit and speculatively wake up load-dependent instructions – loads wakeup dependent instructions only when it reaches the writeback stage.

6.3. Configuration of the Simulator

FabScalar processors follow the SimpleScalar [5] PISA and currently do not support floating-point instructions. Hence, six SPECint2000 [10] benchmarks are used, and the regions run are checked to not have any retiring floating-point instructions. All simulations are 10 million instruction runs starting at SimPoints [7]. The input arguments and starting points of the benchmarks are shown in Table 6.1.
The simulator framework is set up to compare the PC, destination register tags and
destination register value of instructions retiring in Verilog with the SimpleScalar
functional simulator. Hence the functional correctness of the RTL design is validated
for each configuration for every run.

Issue width (and hence the processor backend configuration) is varied from 4 to 6
and 8. The issue logic sub-pipelining configurations used are 1/1, 2/2, 3/3 and 2/3.
The issue queue size is varied from 8 to 16 and 32. In total, 36 configurations are
created by all possible combinations of these parameters. The names given of the
configurations are of the format:

\[ \text{IS<issue\_width>_<issue\_depth\_configuration>_<issue\_queue\_size>} \]

For example, the largest and deepest pipelined configuration is IS8_3/3_32. The
configuration of the rest of the processor is given in table 6.2.

### Table 6.1. Benchmarks, input arguments and run starting points

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input arguments</th>
<th>Start of run (instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>input.program 58</td>
<td>40.6 billion</td>
</tr>
<tr>
<td>Gap</td>
<td>–q –m 64M</td>
<td>161.9 billion</td>
</tr>
<tr>
<td>Gzip</td>
<td>input.graphic 60</td>
<td>77.4 billion</td>
</tr>
<tr>
<td>Mcf</td>
<td>inp.in</td>
<td>44.1 billion</td>
</tr>
<tr>
<td>Parser</td>
<td>2.1.dict –batch</td>
<td>280.3 billion</td>
</tr>
<tr>
<td>Vortex</td>
<td>lendian2.raw</td>
<td>40.8 billion</td>
</tr>
</tbody>
</table>
Table 6.2. Configuration of the processor

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Target Buffer Size</td>
<td>4096</td>
<td>Fetch Width</td>
<td>8</td>
</tr>
<tr>
<td>Branch Predictor Table Size</td>
<td>65536</td>
<td>Fetch Depth</td>
<td>1</td>
</tr>
<tr>
<td>Return Address Stack Size</td>
<td>64</td>
<td>Decode Width</td>
<td>8</td>
</tr>
<tr>
<td>CTI Queue Size</td>
<td>32</td>
<td>Decode Depth</td>
<td>1</td>
</tr>
<tr>
<td>Instruction Buffer Size</td>
<td>64</td>
<td>Dispatch Width</td>
<td>8</td>
</tr>
<tr>
<td>Physical Register File Size</td>
<td>512</td>
<td>Dispatch Depth</td>
<td>1</td>
</tr>
<tr>
<td>Load Store Queue Size</td>
<td>32+32</td>
<td>RegRead Depth</td>
<td>1</td>
</tr>
<tr>
<td>Number of MSHR</td>
<td>8</td>
<td>Retire Width</td>
<td>4</td>
</tr>
<tr>
<td>Rename Map Table Size</td>
<td>34</td>
<td>Active List Size</td>
<td>512</td>
</tr>
</tbody>
</table>

Synthesis is done using the FreePDK 45nm standard cell library [14].
7. Results

7.1. Issue Queue Configuration and Performance

All six benchmarks are runs on all issue queue configurations generated, and performance (IPT) is tabulated in Table 7.1. Performance is measured in billion instructions per second (BIPS). The format of the configuration column is:

\[ \text{IS<issue_width>_<issue_depth_configuration>_<issue_queue_size>} \]

The highest and lowest performing configurations are highlighted for each benchmark.

Table 7.1. Performance (IPT) results in BIPS for all benchmarks run across all issue logic configurations tested. The highest and lowest performing configurations are highlighted.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>bzip</th>
<th>gap</th>
<th>gzip</th>
<th>mcf</th>
<th>parser</th>
<th>Vortex</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS4_1.1_8</td>
<td>0.98</td>
<td>0.68</td>
<td>0.77</td>
<td>0.67</td>
<td>0.81</td>
<td>0.89</td>
</tr>
<tr>
<td>IS4_2.2_8</td>
<td>1.30</td>
<td>0.93</td>
<td>1.01</td>
<td>0.89</td>
<td>1.08</td>
<td>1.24</td>
</tr>
<tr>
<td>IS4_3.3_8</td>
<td>1.97</td>
<td>1.41</td>
<td>1.52</td>
<td>1.35</td>
<td>1.64</td>
<td>1.89</td>
</tr>
<tr>
<td>IS4_2.3_8</td>
<td>1.85</td>
<td>1.31</td>
<td>1.42</td>
<td>1.20</td>
<td>1.52</td>
<td>1.73</td>
</tr>
<tr>
<td>IS6_1.1_8</td>
<td>0.80</td>
<td>0.53</td>
<td>0.60</td>
<td>0.50</td>
<td>0.63</td>
<td>0.67</td>
</tr>
<tr>
<td>IS6_2.2_8</td>
<td>0.96</td>
<td>0.67</td>
<td>0.73</td>
<td>0.62</td>
<td>0.78</td>
<td>0.88</td>
</tr>
<tr>
<td>IS6_3.3_8</td>
<td>1.84</td>
<td>1.28</td>
<td>1.39</td>
<td>1.19</td>
<td>1.49</td>
<td>1.69</td>
</tr>
<tr>
<td>IS6_2.3_8</td>
<td>1.72</td>
<td>1.18</td>
<td>1.29</td>
<td>1.06</td>
<td>1.37</td>
<td>1.54</td>
</tr>
<tr>
<td>IS8_1.1_8</td>
<td>0.64</td>
<td>0.42</td>
<td>0.48</td>
<td>0.39</td>
<td>0.50</td>
<td>0.54</td>
</tr>
<tr>
<td>IS8_2.2_8</td>
<td>0.73</td>
<td>0.51</td>
<td>0.55</td>
<td>0.47</td>
<td>0.59</td>
<td>0.66</td>
</tr>
<tr>
<td>IS8_3.3_8</td>
<td>1.35</td>
<td>0.94</td>
<td>1.02</td>
<td>0.87</td>
<td>1.09</td>
<td>1.23</td>
</tr>
<tr>
<td>IS8_2.3_8</td>
<td>1.26</td>
<td>0.87</td>
<td>0.95</td>
<td>0.78</td>
<td>1.00</td>
<td>1.12</td>
</tr>
</tbody>
</table>
Table 7.1. Continued

| IS4_1.1_16   | 1.30 | 0.77 | 0.81 | 1.13 | 0.99 | 0.91 |
| IS4_2.2_16   | 1.69 | 1.00 | 1.03 | 1.52 | 1.25 | 1.31 |
| IS4_3.3_16   | 2.80 | 1.65 | 1.70 | 2.51 | 2.07 | 2.17 |
| IS4_2.3_16   | 2.55 | 1.53 | 1.59 | 2.18 | 1.91 | 1.99 |
| IS6_1.1_16   | 1.07 | 0.57 | 0.64 | 0.83 | 0.75 | 0.61 |
| IS6_2.2_16   | 1.32 | 0.69 | 0.73 | 1.05 | 0.89 | 0.80 |
| IS6_3.3_16   | 2.43 | 1.28 | 1.34 | 1.94 | 1.64 | 1.47 |
| IS6_2.3_16   | 2.28 | 1.17 | 1.28 | 1.61 | 1.45 | 1.30 |
| IS8_1.1_16   | 0.86 | 0.44 | 0.49 | 0.63 | 0.58 | 0.45 |
| IS8_2.2_16   | 1.00 | 0.51 | 0.53 | 0.75 | 0.65 | 0.60 |
| IS8_3.3_16   | 1.64 | 0.84 | 0.87 | 1.24 | 1.08 | 1.00 |
| IS8_2.3_16   | 1.54 | 0.77 | 0.83 | 1.06 | 0.96 | 0.95 |
| IS4_1.1_32   | 0.50 | 0.55 | 0.51 | 0.94 | 0.66 | 0.62 |
| IS4_2.2_32   | 0.51 | 0.73 | 0.65 | 1.24 | 0.85 | 0.81 |
| IS4_3.3_32   | 0.90 | 1.27 | 1.15 | 2.17 | 1.49 | 1.42 |
| IS4_2.3_32   | 1.07 | 1.22 | 1.13 | 1.89 | 1.42 | 1.37 |
| IS6_1.1_32   | 0.76 | 0.42 | 0.43 | 0.66 | 0.50 | 0.41 |
| IS6_2.2_32   | 1.23 | 0.52 | 0.51 | 0.83 | 0.59 | 0.54 |
| IS6_3.3_32   | 2.13 | 0.90 | 0.89 | 1.43 | 1.02 | 0.93 |
| IS6_2.3_32   | 1.93 | 0.84 | 0.84 | 1.22 | 0.98 | 0.85 |
| IS8_1.1_32   | 0.42 | 0.33 | 0.34 | 0.50 | 0.38 | 0.30 |
| IS8_2.2_32   | 0.55 | 0.39 | 0.38 | 0.62 | 0.44 | 0.38 |
| IS8_3.3_32   | 0.86 | 0.60 | 0.60 | 0.96 | 0.68 | 0.59 |
| IS8_2.3_32   | 0.87 | 0.57 | 0.56 | 0.83 | 0.67 | 0.57 |

Table 7.1 demonstrates the wide range of performance values of benchmarks across all tested issue logic configurations. For instance, bzip achieves the lowest IPT of 0.42 BIPS for an issue queue size of 32, pipelining configuration 1/1 and issue width 8; and the highest IPC of 2.8 BIPS for issue queue size of 16, pipelining configuration of 3/3 and issue width of 4. Other benchmarks show similar performance results.
7.2. Diversity across benchmarks

For each benchmark run, the best configuration comes out to be the same, the deepest pipelined, narrowest (w.r.t. issue width) issue logic with a mid-sized issue queue. The configuration is IS4_3/3_16. The lack of diversity in the best issue logic configuration could be explained by the idea that benchmarks vary in program behavior considerably in 10M instruction runs; the best core gets tailored to run many program behaviors reasonably well and hence, the best core is the same average core. The second experiment (whose results are discussed in section 7.3) does away with the problem of large granularities of viewing code when measuring performance. However, measuring performance at large granularities is found to be not the only cause of lack of diversity, and is the focus of this chapter. Furthermore, the preference for the deepest pipelined, yet narrowest issue logic is unintuitive, and is discussed in section 7.4.

7.3. Diversity within a benchmark

This experiment views a 10M benchmark as composed of small sections of code (10k and 100k instructions) which have similar program behavior within the section. Cores are customized to individual sections of code and the performance on the best (hypothetical) heterogeneous multicore – composed of the best performing cores of each section of code – is calculated.
Figure 7.1 shows the IPT (in BIPS) of the best individual core and the best composed heterogeneous multicore for the benchmark vortex. The best core on an average for the entire 10M run is IS4_3/3_16, i.e., a 4-wide, deeply pipelined 16 issue queue entry configuration. The heterogeneous core is a combination of IS4_3/3_16 (63% 100k sections favored this), IS4_3/3_8 (36%) and IS4_3/3_32 (1%).

![IPT for vortex on the best core (over the course of the entire 10M run) and the best heterogeneous cores. The benchmark is viewed at the granularity of 100k instructions. x-axis is IPT (BIPS), y-axis corresponds to the instructions in the run.](image)

**Figure 7.1.** IPT for vortex on the best core (over the course of the entire 10M run) and the best heterogeneous cores. The benchmark is viewed at the granularity of 100k instructions. x-axis is IPT (BIPS), y-axis corresponds to the instructions in the run.

A very similar result is obtained when the experiment is run at a granularity of 10k instructions.
To ascertain the benefit the (small) diversity seen above, the harmonic mean of the IPTs of the sub-section are calculated for the average core (already known to be the final IPC on the core) and the ideal heterogeneous multicore. The average core achieves an IPT of 2.17 BIPS while the heterogeneous multicore reaches 2.21 BIPS.

This benefit is very small, largely due to lack of diversity in issue width and issue depths. An analysis of the causes for this dearth of diversity is given in the next section.

7.4. Understanding Diversity Results

The lack of diversity in the configurations chosen by benchmarks and sub-sections of benchmarks are explained in detail here below:

7.4.1. Lack of Diversity in Issue Depth

As the issue logic is pipelined, there comes a sweet-spot at which the gain in frequency due to pipelining is offset by IPC degradation due to the increased delay (in cycles) of the wakeup-select loop. The argument here is that the location of this sweet-spot depends on the nature of the workload, and hence, different workloads prefer differently pipelined issue logic. The FabIssue tool has the ability to pipeline the issue logic to a maximum of three sub-stages: the wakeup CAM, the select tree and the payload RAM. The select tree delay dictates the frequency; its delay is large as compared to the CAM and the RAM.
The select tree cannot be pipelined any further because the atomicity of the feedback loop that sets the scheduled bits cannot be broken. This loop starts at the scheduled bits, which contribute to generating the request vector, which feeds into the select logic to generate the grant vector, which set the scheduled bits in the next cycle. In short, the issue logic cannot be pipelined any further beneficially. The results show that a sub-pipelining degree of 3 is still too small to reach the sweet-spot for all benchmarks. Hence, all benchmarks prefer the deepest pipelined configuration available, and no diversity is seen in issue widths across benchmarks.

7.4.2. Lack of Diversity in Issue Width

Increasing issue width improves performance if the gain in IPC that is achieved by going wider (and exploiting ILP) is more than the degradation of cycle time due to increase in logic complexity. There are two reasons for ILP in the FabScalar system to be low:

a) High branch misprediction rate (due to use of bimodal branch predictor) that prevents building large windows to be able to see farther ahead in the program for independent work.

b) High branch misprediction penalty due to the recovery model; mispredicted branches are handled like exceptions.

Going wider also involves cascading the simple function unit select tree. The delay of the cascaded select tree scales roughly linearly as the degree of cascading. Since
the select tree is already in the timing critical path, the frequency hit taken cascading the select tree when going wider is too high to be offset by the increase in IPC. For instance, going from 4-wide to 5-wide in a 3-sub-pipelined machine halves frequency, but IPC certainly does not double!

The lack of ILP means that going wider does not provide enough IPC boost to offset the associated steep decrease in frequency - and all benchmarks pick the configuration with lowest issue width possible (i.e., 4).

7.4.3. Understanding the Best Issue Logic Configuration

It can be seen that the best cores for all benchmarks and sub-sections of benchmarks have the narrowest (lowest issue width), deepest (largest issue depth) issue logic. This brings about the question: What are the disadvantages of a wide or shallow issue logic?

To answer this question, two issue logic configurations are compared:

A) Narrow (issue width = 4) and deep (issue depth = 3) configuration

B) Wide (issue width = 6) and shallow (issue depth = 2) configuration

The issue queue size is the same, 32 entries, for both configurations, A and B. The numbers above are chosen so that the two configurations have:

i) Equal scope, i.e., equal ability to look into the future of the program in search of parallelism. This is achieved by ensuring that the issue queue size is the same.
ii) Equal ability to exploit equal parallelism. This is done by keeping the product of issue width and issue depth the same for the two configurations. Hence, the maximum parallelism that the two configurations can exploit is the same. For example, if the “parallelism available” in the issue queue per cycle is 12 instructions, both configurations would achieve their peak issue rate.

A simple way to view the above configurations is to picture the above configurations to be identical except that configuration A has its ability to exploit parallelism split as 4 times 3, while configuration B has it 6 times 2. Given these ‘equal-scope, equal-parallelism’ configurations, the question in the beginning of this sub-section becomes one of understanding why configuration A is preferred over B. The breakdowns of the delays of the pieces that make up the issue logic are given in Table 7.2. The cycle time of configuration A is the maximum of the three components (CAM, select tree, RAM), while the cycle time of configuration B is the sum of the delays of the select tree and the RAM (this sum happens to be greater than the delay of the wakeup CAM).

| Table 7.2. Cycle time of configurations A and B as a function of components of the issue logic. Times are in ns. |
|---|---|---|---|---|
| Configuration | Wakeup CAM | Select Tree | Payload RAM | Cycle Time |
| A (IS4_3.3_32) | 0.3093 | 0.39 | 0.5195 | 0.5195 |
| B (IS6_2.2_32) | 0.3621 | 0.80 | 0.5789 | 1.3789 |
If the workload has virtually unlimited parallelism (or at least, a parallelism of 12 overlapped instructions in the issue pipeline), both configurations attain their maximum issue rate. Under ideal conditions of perfect branch prediction, zero load violations or exceptions and assuming that all instructions are single-cycle latency instructions, configuration A achieves performance of:

\[(4 \text{ instructions} / 0.5195 \text{ ns}) = 7.70 \text{ BIPS}\]

Configuration B achieves a performance of:

\[(6 \text{ instructions} / 1.3789 \text{ ns}) = 4.35 \text{ BIPS}\]

On the other extreme, if the workload is a serially-dependent chain of single-cycle latency instructions, configuration A issues one instruction every 3 cycles, and configuration B issues one instruction every 2 cycles. Performance achieved by the two configurations are:

A) \[1 \text{ instruction} / (3 * 0.5195 \text{ ns}) = 0.642 \text{ BIPS}\]

B) \[1 \text{ instruction} / (2 * 1.3789 \text{ ns}) = 0.363 \text{ BIPS}\]

This illustrates the disadvantages that a wider (hence shallower – width times depth is kept constant) issue logic suffer. The width forces use of cascaded select trees which degrade frequency to the extent that configuration B achieves a lower overall performance even with higher IPC. Also, configuration B has 6 wakeup ports and 6 payload RAM read ports as compared to configuration A’s 4 each, hence in a situation where the propagation delay of the wakeup-select loop is exposed, being
wider means having a larger propagation delay, which translates to achieving lower performance.

Hence, as compared to configuration A, configuration B suffers twice over in terms of frequency in going wider – a double whammy. The above analysis has been done with hypothetical workloads having unlimited and serial-chain parallelism. A real workload generally lies in between these extremes of available parallelism, but performance of the two configurations at these extreme points is sufficient to state that configuration A will always perform better than configuration B. In short, by simply analyzing the distribution of propagation delay among these equal-scope configurations, configuration B can be eliminated (even without running a benchmark). It is interesting to note that the above elimination can be made even if configurations A and B do not have equal-parallelism (i.e., product of issue width and depth) – comparing peak IPT and serial-chain IPT is sufficient to eliminate configuration B.

7.4.4. Eliminating Issue Logic Configurations

The analysis in the above sub-section, along with explaining a lack of diversity in issue width and depth, also brings up an interesting question: How many issue logic configurations can be eliminated (found to never be the best performing for any program behavior) by analyzing the distribution of logic delays of the issue logic? The condition for eliminating a configuration is that there exists another configuration of equal issue queue size with:
a) Higher peak IPT i.e., the performance achieved when there are enough independent instructions to keep the issue pipeline full at all times.

b) Lower “effective” propagation delay, i.e., product of issue depth and cycle time (cycle time is decided by the chunk of issue logic with the greatest logic delay). This is equivalent to having a greater serial-chain IPT, the performance achieved on a serially-dependent chain of instructions. The serial-chain IPT is the lower bound on performance.

Table 7.3 lists all configurations with an issue queue size of 16 and whether a configuration can be eliminated or not.

Table 7.3. Elimination of issue logic configurations by finding another configuration that always performs better. Times are in ns, IPT in BIPS.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Wakeup CAM</th>
<th>Select Tree</th>
<th>Payload RAM</th>
<th>Cycle Time</th>
<th>Peak IPT</th>
<th>Serial-chain IPT</th>
<th>Eliminated by</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS4_1.1_16</td>
<td>0.2855</td>
<td>0.3</td>
<td>0.45893</td>
<td>1.04443</td>
<td>3.8298</td>
<td>0.9575</td>
<td>None</td>
</tr>
<tr>
<td>IS4_2.2_16</td>
<td>0.2855</td>
<td>0.3</td>
<td>0.45893</td>
<td>0.75893</td>
<td>5.2706</td>
<td>0.6588</td>
<td>IS4_3.3_16</td>
</tr>
<tr>
<td>IS4_3.3_16</td>
<td>0.2855</td>
<td>0.3</td>
<td>0.45893</td>
<td>0.45893</td>
<td>8.7159</td>
<td>0.7263</td>
<td>IS4_3.3_16</td>
</tr>
<tr>
<td>IS6_1.1_16</td>
<td>0.3336</td>
<td>0.62</td>
<td>0.52069</td>
<td>1.47429</td>
<td>4.0698</td>
<td>0.6783</td>
<td>IS4_3.3_16</td>
</tr>
<tr>
<td>IS6_2.2_16</td>
<td>0.3336</td>
<td>0.62</td>
<td>0.52069</td>
<td>1.14069</td>
<td>5.2600</td>
<td>0.4383</td>
<td>IS4_3.3_16</td>
</tr>
<tr>
<td>IS6_3.3_16</td>
<td>0.3336</td>
<td>0.62</td>
<td>0.52069</td>
<td>0.62</td>
<td>9.6774</td>
<td>0.5376</td>
<td>None</td>
</tr>
<tr>
<td>IS8_1.1_16</td>
<td>0.3674</td>
<td>0.94</td>
<td>0.6094</td>
<td>1.9168</td>
<td>4.1736</td>
<td>0.5217</td>
<td>IS4_2.2_16</td>
</tr>
<tr>
<td>IS8_2.2_16</td>
<td>0.3674</td>
<td>0.94</td>
<td>0.6094</td>
<td>1.5494</td>
<td>5.1633</td>
<td>0.3227</td>
<td>IS8_3.3_16</td>
</tr>
<tr>
<td>IS8_3.3_16</td>
<td>0.3674</td>
<td>0.94</td>
<td>0.6094</td>
<td>0.94</td>
<td>8.5106</td>
<td>0.3546</td>
<td>IS8_3.3_16</td>
</tr>
</tbody>
</table>

Thus, out of the 9 configurations listed, 6 can be eliminated by the above approach. For issue queue sizes of 8 and 32, very similar trends are seen. The reason for lack of diversity can now be appreciated.
It is important to note that the results here are not universal – they are strongly dependent on the timing results obtained by the methodology in this thesis. Section 7.5 looks at what could bring about different results.

### 7.5. Can the Best Configuration be Wide or Shallow?

The two disadvantages suffered by wide and shallow issue queue configurations – the double whammy – forces the best configuration to be the narrow and deep for all benchmarks and phases within a benchmark. The reason for this result is a combination of limitations of the FabIssue tool (inability to produce configuration which may have given better IPC), use of synthesized select trees, whose already high logic delays scale linearly as degree of cascading, and simply the nature of pipelining that brings about ability to exploit parallelism and at the same time, improving cycle time. Hence the question: Can the best issue logic configuration be wide or shallow?

Table 7.4. shows two configurations, A and B (similar to Table 7.2) but with hypothetical values for the logic delays of issue logic components.

**Table 7.4.** Elimination of a narrow and deep configuration (A). The values are hypothetical. This example is used to construct the case when a narrow and deep configuration is always worse than a wide and shallow configuration (B). Times are in ns.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Wakeup CAM</th>
<th>Select Tree</th>
<th>Payload RAM</th>
<th>Cycle Time</th>
<th>Peak IPT</th>
<th>Serial-chain IPT</th>
<th>Eliminated by</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (IS4_3.3_32)</td>
<td>0.9</td>
<td>0.2</td>
<td>0.3</td>
<td>0.9</td>
<td>4.4444</td>
<td>0.3704</td>
<td>IS6_2.2_32</td>
</tr>
<tr>
<td>B (IS6_2.2_32)</td>
<td>1.0</td>
<td>0.4</td>
<td>0.7</td>
<td>1.1</td>
<td>5.4545</td>
<td>0.4545</td>
<td>None</td>
</tr>
</tbody>
</table>

61
Configuration B achieves higher performance in both high and low parallelism workloads. Configuration A can thus be eliminated.

The reason for the wide and shallow configuration being better is the imbalance in pipelining that is present in configuration A in the above (hypothetical) example. Depending on the nature of scaling of delays of components that make up the issue logic, a deeply pipelined issue logic could be imbalanced to the extent that the frequency benefit it extracts (if any) is not sufficient to offset the extra “effective” propagation delay (i.e., issue depth times cycle time, this is exposed when workload parallelism is low) due to pipelining. Effective propagation delay is the inverse of the serial-chain IPT in the above table.
8. Conclusion

Out-of-order superscalar processors fetch, issue and execute multiple instructions per cycle – the issue logic is the part of the processor that deals with data dependencies and ensures that instructions are issued in a timely manner. This thesis studies the construction and working of the issue logic and attempts to understand how workload performance is affected by issue logic configuration. The parameters defining issue logic configuration are issue width, issue depth and issue queue size. This thesis presents FabIssue, a tool in the FabScalar toolset that can generate synthesizable RTL issue logic designs of desired configuration, and is used to generate various issue logic configurations that are used to run benchmarks to obtain IPC and synthesized to estimate frequency. The reasons for certain issue logic configurations to perform better (performance is defined as IPT) than other configurations are analyzed. This thesis also uses FabIssue to experiment with different issue logic configurations in the context of multicore heterogeneity. The hypothetical heterogeneous multicore with highest performance for the workload set – and for a single workload viewed as a series of program phases – is constructed and the benefit of heterogeneity is estimated. The diversity in best cores (which differ only in issue logic configurations) is studied. The insights and results of the thesis are enumerated below.
8.1. Summary of Results

1) The loop that sets the scheduled bits of selected instructions in the issue queue must be atomic. The select tree is part of this loop, and hence, the select tree cannot be pipelined. Timing numbers generated in this thesis show that the select tree has a large logic delay (relative to the other parts of the issue logic – the wakeup CAM and the payload RAM). Under the assumption that the issue logic is the decider of frequency, the select tree delay accounts for a large portion of the cycle time. Since the select tree is unpipelinable, pipelining cannot be used to boost frequency beyond a certain point. Hence, the design of the select tree is critical, especially if the processor is designed to be wide and accommodate a large window (i.e., large issue queue size).

2) The select logic becomes even more timing critical when the issue width is increased. In a processor with pipeline ways dedicated to execute instructions of a certain type, a large issue width means multiple backend pipeline ways capable of executing instructions of a certain type. Multiple unique instructions of the same type have to be selected, leading to cascaded select trees. The logic delay of cascaded trees scale roughly linearly with the degree of cascading, worsening the already high select logic delay. The inference from this observation is that in wide processors with a unified issue queue but dedicated execution units, the mixture of function units (which decides the degree of select tree cascading) would have to be such that cascading of select trees is kept to a minimum.
3) A wide issue logic configuration suffers twice over – with respect to frequency – as compared to a narrow and deep issue logic design. Firstly, going wider means cascading select trees and degrading frequency. Secondly, going wider means increasing the number of wakeup ports and payload RAM read ports – this increases the read latencies of these memory structures – again degrading frequency. This understanding gained in this thesis regarding pipelining the issue logic can be stated from another point of view: Pipelining the issue logic has two beneficial effects.

a) It may provide a higher frequency, depending on whether there exists a location in the issue logic where a pipeline register can be placed to achieve reasonably balanced pipelining.

b) It increases the ability of the issue logic to exploit parallelism: An n-wide 2-deep configuration is – in a different way – able to exploit instruction-level parallelism as a 2n-wide 1-deep configuration. Whether this ability to exploit parallelism is actually used depends on the nature of the workload (whether there is sufficient parallelism).

4) The lack of diversity in the best issue logic configurations is a pointer to the direction in which the FabIssue tool, and the FabScalar toolset, must be further developed, which is, removing the limitations on the tool that currently lead to lower-than-expected IPC, especially for the complex (wide, large and deep) designs.
REFERENCES


