Complexity-Effective Superscalar Processors

Subbarao Palacharla
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706, USA
subbarao@cs.wisc.edu

Norman P. Jouppi
Western Research Laboratory
Digital Equipment Corporation
Palo Alto, CA 94301, USA
jouppi@pa.dec.com

J. E. Smith
Dept. of Electrical and Computer Engg.
University of Wisconsin-Madison
Madison, WI 53706, USA
jes@ece.wisc.edu

Abstract

The performance tradeoff between hardware complexity and clock speed is studied. First, a generic superscalar pipeline is defined. Then the specific areas of register renaming, instruction window wakeup and selection logic, and operand bypassing are analyzed. Each is modeled and Spice simulated for feature sizes of $0.8\mu m$, $0.35\mu m$, and $0.18\mu m$. Performance results and trends are expressed in terms of issue width and window size. Our analysis indicates that window wakeup and selection logic as well as operand bypass logic are likely to be the most critical in the future.

A microarchitecture that simplifies wakeup and selection logic is proposed and discussed. This implementation puts chains of dependent instructions into queues, and issues instructions from multiple queues in parallel. Simulation shows little slowdown as compared with a completely flexible issue window when performance is measured in clock cycles. Furthermore, because only instructions at queue heads need to be awakened and selected, issue logic is simplified and the clock cycle is faster – consequently overall performance is improved. By grouping dependent instructions together, the proposed microarchitecture will help minimize performance degradation due to slow bypasses in future wide-issue machines.

1 Introduction

For many years, a major point of contention among microprocessor designers has revolved around complex implementations that attempt to maximize the number of instructions issued per clock cycle, and much simpler implementations that have a very fast clock cycle. These two camps are often referred to as "brainiacs" and "speed demons" – taken from an editorial in Microprocessor Report [7]. Of course the tradeoff is not a simple one, and through innovation and good engineering, it may be possible to achieve most, if not all, of the benefits of complex issue schemes, while still allowing a very fast clock in the implementation; that is, to develop microarchitectures we refer to as *complexity-effective*. One of two primary objectives of this paper is to propose such a complexity-effective microarchitecture. The proposed microarchitecture achieves high performance, as measured by instructions per cycle (IPC), yet it permits a design with a very high clock frequency.

Supporting the claim of high IPC with a fast clock leads to the second primary objective of this paper. It is commonplace to measure the effectiveness (i.e. IPC) of a new microarchitecture, typically by using trace driven simulation. Such simulations count clock cycles and can provide IPC in a fairly straightforward manner. However, the complexity (or simplicity) of a microarchitecture is much more difficult to determine – to be very accurate, it requires a full implementation in a specific technology. What is very much needed are fairly straightforward measures of complexity that can be used by microarchitects at a fairly early stage of the design process. Such methods would allow the determination of complexity-effectiveness. It is the second objective of this paper to take a step in the direction of characterizing complexity and complexity trends.

Before proceeding, it must be emphasized that while complexity can be variously quantified in terms such as number of transistors, die area, and power dissipated, in this paper complexity is measured as the delay of the critical path through a piece of logic, and the longest critical path through any of the pipeline stages determines the clock cycle.

The two primary objectives given above are covered in reverse order – first sources of pipeline complexity are analyzed, then a new complexity-effective microarchitecture is proposed and evaluated. In the next section we describe those portions of a microarchitecture that tend to have complexity that grows with increasing instruction-level parallelism. Of these, we focus on instruction dispatch and issue logic, and data bypass logic. We analyze potential critical paths in these structures and develop models for quantifying their delays. We study the variation of these delays with microarchitectural parameters of window size (the number of waiting instructions from which ready instructions are selected for issue) and the issue width (the number of instructions that can be issued in a cycle). We also study the impact of the technology trend towards smaller feature sizes. The complexity analysis shows that logic associated with the issue window and data bypasses are likely to be key limiters of clock speed since smaller feature sizes cause wire delays to dominate overall delay [20, 3].

Taking sources of complexity into account, we propose and evaluate a new microarchitecture. This microarchitecture is called *dependence-based* because it focuses on grouping dependent instructions rather than independent ones, as is often the case in superscalar implementations. The dependence-based microarchitecture simplifies issue window logic while exploiting similar levels of parallelism to that achieved by current superscalar microarchitectures using more complex logic.

The rest of the paper is organized as follows. Section 2 describes the sources of complexity in a baseline microarchitecture. Section 3 describes the methodology we use to study the critical pipeline

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

structures identified in Section 2. Section 4 presents a detailed analysis of each of the structures and shows how their delays vary with microarchitectural parameters and technology parameters. Section 5 presents the proposed dependence-based microarchitecture and some preliminary performance results. Finally, we draw conclusions in Section 6.

2 Sources of Complexity

In this section, specific sources of pipeline complexity are considered. We realize that it is impossible to capture all possible microarchitectures in a single model, however, and any results have some obvious limitations. We can only hope to provide a fairly straightforward model that is typical of most current superscalar processors, and suggest that analyses similar to those used here can be extended to other, more advanced techniques as they are developed.

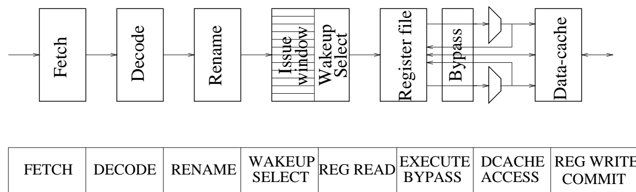

Figure 1: Baseline superscalar model.

Figure 1 shows the baseline model and the associated pipeline. The fetch unit reads multiple instructions every cycle from the instruction cache, and branches encountered by the fetch unit are predicted. Next, instructions are decoded and their register operands are renamed. Renamed instructions are dispatched to the instruction window, where they wait for their source operands and the appropriate functional unit to become available. As soon as these conditions are satisfied, instructions are issued and executed in the functional units. The operand values of an instruction are either fetched from the register file or are bypassed from earlier instructions in the pipeline. The data cache provides low latency access to memory operands.

2.1 Basic Structures

As mentioned earlier, probably the best way to identify the primary sources of complexity in a microarchitecture is to actually implement the microarchitecture in a specific technology. However, this is extremely time consuming and costly. Instead, our approach is to select certain key structures for study, and develop relatively simple delay models that can be applied in a straightforward manner without relying on detailed design.

Structures to be studied were selected using the following criteria. First, we consider structures whose delay is a function of issue window size and/or issue width; these structures are likely to become cycle-time limiters in future wide-issue superscalar designs. Second, we are interested in dispatch and issue-related structures because these structures form the core of a microarchitecture and largely determine the amount of parallelism that can be exploited. Third, some structures tend to rely on broadcast operations over long wires and hence their delays might not scale as well as logic-intensive structures in future technologies with smaller feature sizes.

The structures we consider are:

- *Register rename logic*. This logic translates logical register designators into physical register designators.
- *Wakeup logic*. This logic is part of the issue window and is responsible for waking up instructions waiting for their source operands to become available.
- *Selection logic*. This logic is another part of the issue window and is responsible for selecting instructions for execution from the pool of ready instructions.
- *Bypass logic*. This logic is responsible for bypassing operand values from instructions that have completed execution, but have not yet written their results to the register file, to subsequent instructions.

There are other important pieces of pipeline logic that are not considered in this paper, even though their delay is a function of dispatch/issue width. In most cases, their delay has been considered elsewhere. These include register files and caches. Farkas et. al. [6] study how the access time of the register file varies with the number of registers and the number of ports. The access time of a cache is a function of the size of the cache and the associativity of the cache. Wada et. al. [18] and Wilton and Jouppi [21] have developed detailed models that estimate the access time of a cache given its size and associativity.

2.2 Current Implementations

The structures identified above were presented in the context of the baseline superscalar model shown in Figure 1. The MIPS R10000 [22] and the DEC 21264 [10] are real implementations that directly fit this model. Hence, the structures identified above apply to these two processors.

On the other hand, the Intel Pentium Pro [9], the HP PA-8000 [12], the PowerPC 604 [16], and the HAL SPARC64 [8] do not completely fit the baseline model. These processors are based on a microarchitecture where the reorder buffer holds non-committed, renamed register values. In contrast, the baseline microarchitecture uses the physical register file for both committed and non-committed values. Nevertheless, the point to be noted is that the basic structures identified earlier are present in both types of microarchitectures. The only notable difference is the size of the physical register file.

Finally, while the discussion about potential sources of complexity is in the context of an out-of-order baseline superscalar model, it must be pointed out that some of the critical structures identified apply to in-order processors, too. For example, part of the register rename logic (to be discussed later) and the bypass logic are present in in-order superscalar processors.

3 Methodology

The key pipeline structures were studied in two phases. In the first phase, we selected a representative CMOS circuit for the structure. This was done by studying designs published in the literature (e.g. ISSCC [1] proceedings) and by collaborating with engineers at Digital Equipment Corporation. In cases where there was more than one possible design, we did a preliminary study of the designs to decide in favor of one that was most promising. By basing our circuits on designs published by microprocessor vendors, we believe the studied circuits are similar to circuits used in microprocessor designs. In practice, many circuit tricks could be employed to optimize critical paths. However, we believe that the relative delays between different structures should be more accurate than the absolute delays.

[1] International Solid-State and Circuits Conference.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

In the second phase we implemented the circuit and optimized the circuit for speed. We used the Hspice circuit simulator [14] from Meta-Software to simulate the circuits. Primarily, static logic was used. However, in situations where dynamic logic helped in boosting the performance significantly, we used dynamic logic. For example, in the wakeup logic, a dynamic 7-input NOR gate is used for comparisons instead of a static gate. A number of optimizations were applied to improve the speed of the circuits. First, all the transistors in the circuit were manually sized so that overall delay improved. Second, logic optimizations like two-level decomposition were applied to reduce fan-in requirements. We avoided using static gates with a fan-in greater than four. Third, in some cases transistor ordering was modified to shorten the critical path. Wire parasitics were added at appropriate nodes in the Hspice model of the circuit. These parasitics were computed by calculating the length of the wires based on the layout of the circuit and using the values of R_{metal} and C_{metal}, the resistance and parasitic capacitance of metal wires per unit length.

To study the effect of reducing the feature size on the delays of the structures, we simulated the circuits for three different feature sizes: $0.8\mu m$, $0.35\mu m$, and $0.18\mu m$ respectively. Layouts for the $0.35\mu m$ and $0.18\mu m$ process were obtained by appropriately shrinking the layouts for the $0.8\mu m$ process. The Hspice models used for the three technologies are tabulated in [15].

4 Pipeline Complexity

In this section, we analyze the critical pipeline structures. The presentation for each structure begins with a description of the logical function being implemented. Then, possible implementation schemes are discussed, and one is chosen. Next, we summarize our analysis of the overall delay in terms of the microarchitectural parameters of issue width and issue window size; a much more detailed version of the analysis appears in [15]. Finally, Hspice circuit simulation results are presented and trends are identified and compared with the earlier analysis.

4.1 Register Rename Logic

Register rename logic translates logical register designators into physical register designators by accessing a map table with the logical register designator as the index. The map table holds the current logical to physical mappings and is multi-ported because multiple instructions, each with multiple register operands, need to be renamed every cycle. The high level block diagram of the rename logic is shown in Figure 2. In addition to the map table, dependence check logic is required to detect cases where the logical register being renamed is written by an earlier instruction in the current group of instructions being renamed. The dependence check logic detects such dependences and sets up the output MUXes so that the appropriate physical register designators are selected. At the end of every rename operation, the map table is updated to reflect the new logical to physical mappings created for the result registers written by the current rename group.

4.1.1 Structure

The mapping and checkpointing functions of the rename logic can be implemented in at least two ways. These two schemes, called the RAM scheme and the CAM scheme, are described next.

- *RAM scheme*
  In the RAM scheme, implemented in the MIPS R10000 [22], the map table is a register file where the logical register designator directly accesses an entry that contains the physical register to which it is mapped. The number of entries in the map table is equal to the number of logical registers.

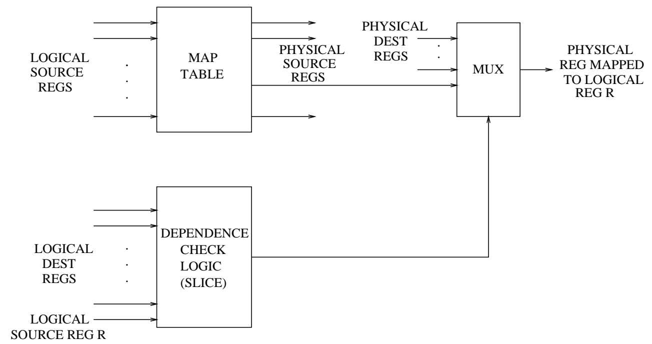

Figure 2: Register rename logic.

- *CAM scheme*
  An alternate scheme for register renaming uses a CAM (content-addressable memory) [19] to store the current mappings. Such a scheme is implemented in the HAL SPARC [2] and the DEC 21264 [10]. The number of entries in the CAM is equal to the number of physical registers. Each entry contains two fields: the logical register designator that is mapped to the physical register represented by the entry and a valid bit that is set if the current mapping is valid. Renaming is accomplished by matching on the logical register designator field.

In general, the CAM scheme is less scalable than the RAM scheme because the number of CAM entries, which is equal to the number of physical registers, tends to increase with issue width. Also, for the design space we are interested in, the performance was found to be comparable. Consequently, we focus on the RAM method below. A more detailed discussion of the trade-offs involved can be found in [15].

The dependence check logic proceeds in parallel with the map table access. Every logical register designator being renamed is compared against the logical destination register designators of earlier instructions in the current rename group. If there is a match, then the physical register assigned to the result of the earlier instruction is used instead of the one read from the map table. In the case of multiple matches, the register corresponding to the latest (in dynamic order) match is used. Dependence check logic for issue widths of 2, 4, and 8 was implemented. We found that for these issue widths, the delay of the dependence check logic is less than the delay of the map table, and hence the check can be hidden behind the map table access.

4.1.2 Delay Analysis

As the name suggests, the RAM scheme operates like a standard RAM. Address decoders drive word lines; an access stack at the addressed cell pulls a bitline low. The bitline changes are sensed by a sense amplifier which in turn produces the output. Symbolically the rename delay can be written as,

$$T_{rename} = T_{decode} + T_{wordline} + T_{bitline} + T_{senseamp}$$

The analysis presented here and in following subsections focuses on those parts of the delay that are a function of the issue width and window size. All sources of delay are considered in detail in [15]. In the rename logic, the window size is not a factor, and the issue width affects delay through its impact on wire lengths. Increasing

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

the issue width increases the number of bitlines and wordlines in each cell thus making each cell bigger. This in turn increases the length of the predecode, wordline, and bitline wires and the associated wire delays. The net effect is the following relationships for the delay components:

$$T_{decode}, T_{wordline}, T_{bitline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

where IW is the issue width and c_0, c_1, and c_2 are constants that are fixed for a given technology and instruction set architecture; derivation of the constants for each component is given in [15]. In each case, the quadratic component, resulting from the intrinsic RC delay of wires, is relatively small for the design space and technologies we explored. Hence, the decode, wordline, and bitline delays are effectively linear functions of the issue width.

For the sense amplifier, we found that even though its structural constitution is independent of the issue width, its delay is a function of the slope of the input – the bitline delay – and therefore varies linearly with issue width.

4.1.3 Spice Results

For our Hspice simulations, Figure 3 shows how the delay of the rename logic varies with the issue width i.e. the number of instructions being renamed every cycle for the three technologies. The graph includes the breakdown of delay into components discussed in the previous section.

A number of observations can be made from the graph. The total delay increases linearly with issue width for all the technologies. This is in conformance with our analysis, summarized in the previous section. Furthermore, each of the components shows a linear increase with issue width. The increase in the bitline delay is larger than the increase in the wordline delay as issue width is increased because the bitlines are longer than the wordlines in our design. The bitline length is proportional to the number of logical registers (32 in most cases) whereas the wordline length is proportional to the width of the physical register designator (less than 8 for the design space we explored).

Another important observation that can be made from the graph is that the relative increase in wordline delay, bitline delay, and hence, total delay as a function of issue width worsens as the feature size is reduced. For example, as the issue width is increased from 2 to 8, the percentage increase in bitline delay shoots up from 37% to 53% as the feature size is reduced from $0.8\mu m$ to $0.18\mu m$. Logic delays in the various components are reduced in proportion to the feature size, while the presence of wire delays in the wordline and bitline components cause the wordline and bitline components to fall at a slower rate. In other words, wire delays in the wordline and bitline structures will become increasingly important as feature sizes are reduced.

4.2 Wakeup Logic

Wakeup logic is responsible for updating source dependences for instructions in the issue window waiting for their source operands to become available.

4.2.1 Structure

Wakeup logic is illustrated in Figure 4. Every time a result is produced, a tag associated with the result is broadcast to all the instructions in the issue window. Each instruction then compares the tag with the tags of its source operands. If there is a match, the operand is marked as available by setting the rdyL or rdyR flag. Once all the operands of an instruction become available (both rdyL and rdyR are set), the instruction is ready to execute, and the ready flag is set

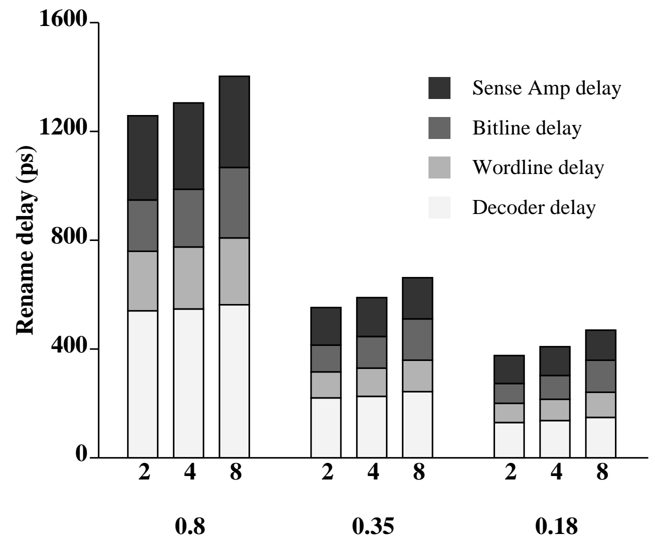

Figure 3: Rename delay versus issue width.

to indicate this. The issue window is a CAM array holding one instruction per entry. Buffers, shown at the top of the figure, are used to drive the result tags $tag1$ to $tagIW$, where IW is the issue width. Each entry of the CAM has $2 \times IW$ comparators to compare each of the results tags against the two operand tags of the entry. The OR logic ORs the comparator outputs and sets the rdyL/rdyR flags.

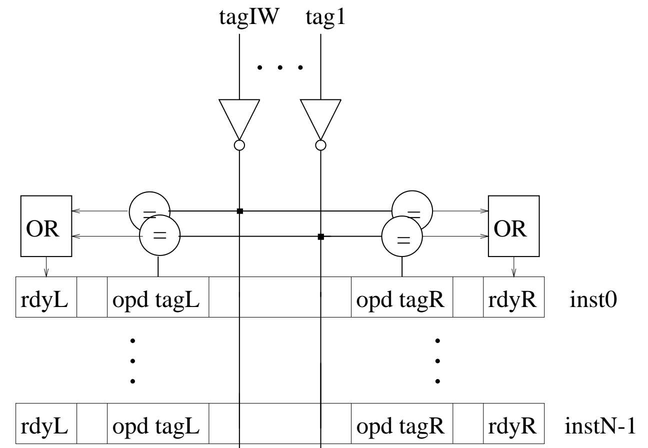

Figure 4: Wakeup logic.

4.2.2 Delay Analysis

The delay consists of three components: the time taken by the buffers to drive the tag bits, the time taken by the comparators in a pull-down stack corresponding to a mismatching bit position to pull the matchline low [2], and the time taken to OR the individual match signals (matchlines). Symbolically,

$$Delay = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

The time taken to drive the tags depends on the length of the tag lines and the number of comparators on the tag lines. Increasing the window size increases both these terms. For a given window size,

[2] We assume that only one pull-down stack is turned on since we are interested in the worst-case delay.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

increasing issue width also increases both the terms in the following way. Increasing issue width increases the number of matchlines in each cell and hence increases the height of each cell. Also, increasing issue width increases the number of comparators in each cell. Note that we assume the maximum number of tags produced per cycle is equal to the maximum issue width.

In simplified form (see [15] for a more detailed analysis), the time taken to drive the tags is:

$$T_{tagdrive} = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c_5 \times IW^2) \times WINSIZE^2$$

The above equation shows that the tag drive time is a quadratic function of the window size. The weighting factor of the quadratic term is a function of the issue width. The weighting factor becomes significant for issue widths beyond 2. For a given window size, the tag drive time is also a quadratic function of the issue width. For current technologies ($0.35\mu m$ and longer) the quadratic component is relatively small and the tag drive time is largely a linear function of issue width. However, as the feature size is reduced to $0.18\mu m$, the quadratic component also increases in significance. The quadratic component results from the intrinsic RC delay of the tag lines.

In reality, both issue width and window size will be simultaneously increased because a larger window is required for finding more independent instructions to take advantage of wider issue. Hence, the tag drive time will become significant in future designs with wider issue widths, bigger windows, and smaller feature sizes.

The tag match time is primarily a function of the length of the matchline, which varies linearly with the issue width. The match OR time is the time taken to OR the match lines, and the number of matchlines is a linear function of issue width. Both of these (refer to [15]) have a delay:

$$T_{tagmatch}, T_{matchOR} = c_0 + c_1 \times IW + c_2 \times IW^2$$

However, in both cases the quadratic term is very small for the design space we consider, so these delays are linear functions of issue width.

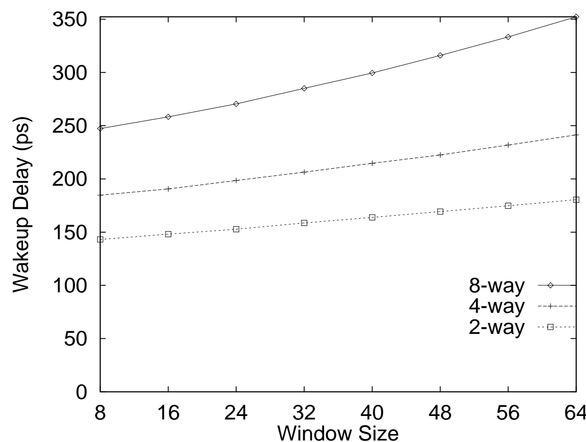

Figure 5: Wakeup logic delay versus window size.

4.2.3 Spice Results

The graph in Figure 5 shows how the delay of the wakeup logic varies with window size and issue width for $0.18\mu m$ technology. As expected, the delay increases as window size and issue width are increased. The quadratic dependence of the total delay on the window size results from the quadratic increase in tag drive time as discussed in the previous section. This effect is clearly visible for issue width of 8 and is less significant for issue width of 4. We found similar curves for $0.8\mu m$ and $0.35\mu m$ technologies. The quadratic dependence of delay on window size was more prominent in the curves for $0.18\mu m$ technology than in the case of the other two technologies.

Also, issue width has a greater impact on the delay than window size because increasing issue width increases all three components of the delay. On the other hand, increasing window size only lengthens the tag drive time and to a small extent the tag match time. Overall, the results show that the delay increases by almost 34% going from 2-way to 4-way and by 46% going from 4-way to 8-way for a window size of 64 instructions. In reality, the increase in delay is going to be even worse because in order to sustain a wider issue width, a larger window is required to find independent instructions.

Figure 6 shows the effect of reducing feature sizes on the various components of the wakeup delay for an 8-way, 64-entry window processor. The tag drive and tag match delays do not scale as well as the match OR delay. This is expected since tag drive and tag match delays include wire delays whereas the match OR delay only consists of logic delays. Quantitatively, the fraction of the total delay contributed by tag drive and tag match delay increases from 52% to 65% as the feature size is reduced from $0.8\mu m$ to $0.18\mu m$. This shows that the performance of the broadcast operation will become more crucial in future technologies.

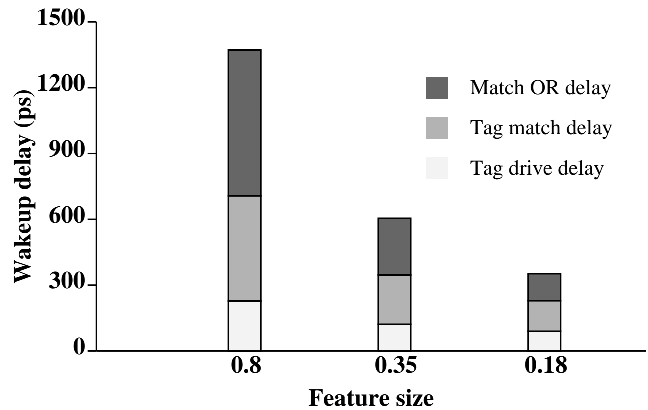

Figure 6: Wakeup delay versus feature size.

4.3 Selection Logic

Selection logic is responsible for choosing instructions for execution from the pool of ready instructions in the issue window. Some form of selection logic is required because the number and types of ready instructions may exceed the number and types of functional units available to execute them.

Inputs to the selection logic are request (REQ) signals, one per instruction in the issue window. The request signal of an instruction is raised when the wakeup logic determines that all its operands are available. The outputs of the selection logic are grant (GRANT) signals, one per request signal. On receipt of the GRANT signal, the associated instruction is issued to the functional unit.

A *selection policy* is used to decide which of the requesting instructions is granted. An example selection policy is *oldest first* - the ready instruction that occurs earliest in program order is granted

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.