Quantifying the Complexity of Superscalar Processors

Subbarao Palacharla[†] Norman P. Jouppi[‡] James E. Smith[⋆]

[†]Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706, USA
subbarao@cs.wisc.edu

[‡]Western Research Laboratory
Digital Equipment Corporation
Palo Alto, CA 94301, USA
jouppi@pa.dec.com

[⋆]Department of ECE
University of Wisconsin-Madison
Madison, WI 53706, USA
jes@ece.wisc.edu

Abstract

To characterize future performance limitations of superscalar processors, the delays of key pipeline structures in superscalar processors are 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 (delay) results and trends are expressed in terms of issue width and window size. This analysis indicates that window (wakeup and select) logic and operand bypass logic are likely to be the most critical in the future.

1 Introduction

The current trend in the microprocessor industry is towards increasingly complex out-of-order microarchitectures. The intention is to exploit larger amounts of instruction level parallelism. There is an important tradeoff, however. More complex hardware tends to limit the clock speed of a microarchitecture by lengthening critical paths. Because performance is proportional to $Clock\ Speed \times Instructions\ Per\ Cycle$ microarchitects need to study techniques that maximize the product rather than those that push the limits of each term independently. We are interested in exploring such *complexity-effective* microarchitectures. I.e those that optimize the product of complexity (as measured by the clock cycle) and effectiveness (instructions per cycle). It must be emphasized here that while complexity can be variously quantified in terms such as number of transistors, die area, clock-speed/cycle-time, and power dissipated, in this paper we measure complexity as the critical path through a piece of logic, and the longest critical path through any of the pipeline stages determines the clock speed.

It is relatively straightforward to measure the effectiveness of a microarchitecture, e.g. via trace driven simulation based on clock cycles. Such simulations count clock cycles and can provide instructions per cycle in a straightforward manner. However, the complexity of a microarchitecture is much more difficult to determine – to be very accurate, it would require a full implementation in a specific technology. What is very much needed are fairly straightforward measures, possibly only relative 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. This report represents an effort in that direction. In the next section we describe those portions of a microarchitecture that tend to have a 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 ways these delays vary with microarchitectural parameters like 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 technology trends towards smaller feature sizes.

In addition to delays, another important consideration is the pipelineability of each of these structures. Even if the delay of a structure is relatively large it might not increase the complexity of the design if the structure can be pipelined i.e. the operation of the structure can be spread over multiple pipestages. However, this is likely to affect the effectiveness by reducing the instructions per cycle by increasing latencies of functional operations or by increasing the penalty of mispredicted branches and instruction cache misses when the pipeline has to be re-filled in these cases. We study the pipelineability of critical structures and identify certain operations that have to be *atomic* i.e. performed in a single cycle for dependent instructions to execute in consecutive cycles.

Our delay analysis shows that logic associated with the issue window in a superscalar processor can be a key limiter of clock speed as we move towards wider issue widths, larger windows, and advanced technologies in which wire delays dominate overall delay. We split the issue window logic into two basic functions: *wakeup* and *selection*. At the time an instruction is ready to complete, the tag of the result is broadcast to all waiting instructions in the window so they can update their dependence information. This broadcast and the determination that an instruction has all its dependences resolved constitutes the wakeup function. The selection function is required to select a maximum of w ready instructions every cycle from the window of instructions where w is the number of functional units in the microarchitecture. In order to be able to execute dependent instructions back-to-back (in consecutive cycles) the wakeup and selection function have to be completed in a single cycle. Furthermore, the wakeup function involves broadcasting result tags on a set of wires that span the window. In advanced technologies wire delays will increasingly dominate the total delay and hence delay of the wakeup logic is likely to become a bottleneck in the future.

Another structure that can potentially limit clock speed especially in future technologies is the bypass logic. The result wires that are used to bypass operand values increase in length as the number of functional units is increased. These wire delays could ultimately dominate and force architects to choose in favor of more decentralized microarchitectures.

The rest of this report 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 structures identified in Section 2. Section 4 discusses technology trends and why wires are becoming more important than gates as feature sizes shrink. Section 5 presents a detailed analysis of each of the structures and shows how their delays vary with microarchitectural parameters and technology parameters. Section 6 discusses overall results and pipelineability of each of the structures. Finally, conclusions are in Section 7.

2 Sources of Complexity

Before delving into specific sources of complexity we describe the baseline superscalar model assumed for the study. We then list and discuss the basic structures that are the primary sources of complexity. Finally, we show how these basic structures are present in one form or another in most current implementations even though these implementations might appear to be different superficially. On the other hand, we realize that it is impossible to capture all possible microarchitectures in a single model and any results we provide 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 techniques similar to those used here can be extended for other, more advanced models as they are developed.

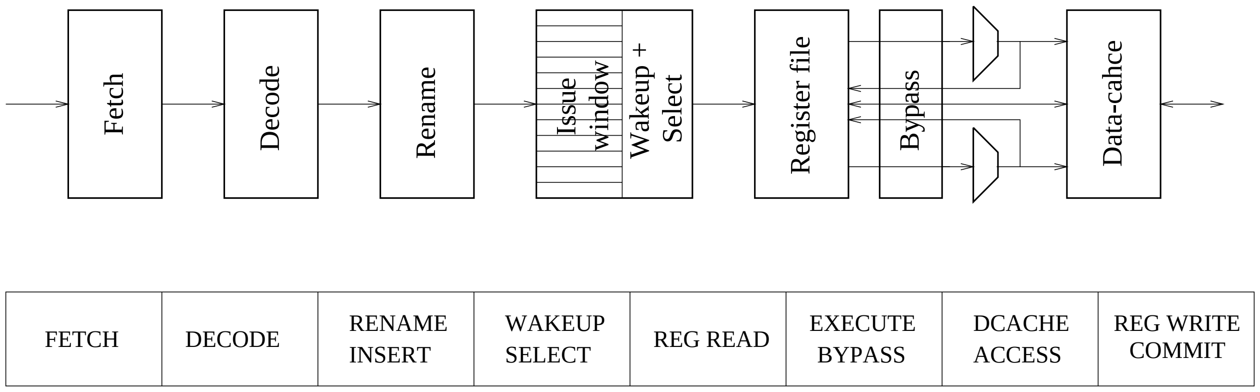


Figure 1: Baseline superscalar model

Figure 1 shows the baseline model and the associated pipeline. The fetch unit fetches multiple instructions every cycle from the instruction cache. Branches encountered by the fetch unit are predicted. Following instruction fetch, instructions are decoded and their register operands are renamed. Register renaming involves mapping the logical register operands of an instruction to the appropriate physical registers. This step eliminates write-after-read and write-after-write conflicts by converting the instructions into the single assignment form. Renamed instructions are dispatched to the issue window, where they wait for their source operands and the appropriate functional unit to become available. As soon as this condition is satisfied, the instruction is issued and executes on one of the functional units. The operand values of the 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 via loads and stores.

2.1 Basic Structures

As mentioned earlier, probably the best way to identify the primary sources of complexity in a microarchitecture is to implement the microarchitecture in a specific technology. However, this is extremely time consuming and costly. Our approach instead is to first identify those structures whose delay is a function of issue window size and issue width. Then, we select some of these for additional study and develop relatively simple delay models that can be applied in a straightforward manner without relying on detailed design.

For example, we include register renaming logic in the list of structures because its delay depends on the issue width in the following way. The number of read ports into the rename table is $(numopd \times IW)$ where $numopd$ is the number of read operands per instruction and IW is the issue width. For example, assuming 2-operand instructions, a 4-way machine would require as many as 8 read ports into the rename table whereas a 2-way machine would only require 4 read ports. On the other hand we do not include any of the functional units because their delay is independent of both the issue width and the window size.

In addition to the above criterion, our decision to study a particular structure was based on a number of other considerations. First, we are primarily 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. Second, some of these structures tend to rely on broadcast operations on long wires and hence their delays might not scale as well as logic-intensive structures in future technologies with smaller feature sizes. Third, in most cases the delay of these structures may potentially grow quadratically with issue width. Hence, we believe that these structures will become potential cycle-time determinants in future wide-issue designs in advanced technologies.

The structures we consider are:

- *Register rename logic*
  Register rename logic translates logical register designators into physical register designators. The translation is accomplished by accessing a map table with the logical register designator as the index. Each instruction is renamed as follows. The physical registers corresponding to the operand registers are read from the map table. If the instruction produces a result, the logical destination register is assigned a physical register from the pool of free registers and the map table is updated to reflect this new mapping.

  In addition to reading mappings from the map table the rename logic also has to detect true dependences between instructions being renamed in parallel. This involves comparing each logical source register to the logical destination register of earlier instructions in the current rename group. The dependence check logic is responsible for performing this task.

  From the above discussion it is obvious that the delay of rename logic is a function of the issue width because the issue width determines the number of ports into the map table and the width of the dependence check logic.

- *Wakeup logic*
  This logic is part of the issue window and is responsible for *waking up* instructions waiting in the issue window for their source operands to become available. Once an instruction is issued for execution, the tag corresponding to its result is broadcast to all the instructions in the window. Each instruction in the window compares the tag with its source operand tags. Once all the source operands of an instruction are available the instruction is flagged *ready* for execution.

  The delay of the wakeup logic is a function of the window size and the issue width. The window size determines the fanout of the broadcast; the larger the window size the greater is the length of the wires used for broadcasting. Similarly, increasing the issue width also increases the delay of the wakeup logic because the size of each window entry increases with issue width.

- *Selection logic*

  The selection logic is part of the issue window and is responsible for selecting instructions for execution from the pool of ready instructions. An instruction is said to be ready if all of its source operands are available. A typical policy used by the selection logic is *oldest ready first*.

  The delay of this logic is a function of the window size, the number of functional units, and the selection policy.

- *Data bypass logic*

  The data bypass 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. The bypass logic is implemented as a set of wires, called the result wires, that carry the result (bypassed) values from each source to all possible destinations. MUXes, called operand MUXes, are used to select the appropriate result to gate into the operand ports of functional units.

  The delay of this logic is a function of the number of functional units and the depth of the pipeline. The delay of the bypass logic depends on the length of the result wires and the load on these wires. Increasing the number of functional units increases the length of the result wires. It also increases the fan-in of the operand MUXes. Making the pipeline deeper might increase the number of sources and hence the number of result wires. Again, this also increases the fan-in of the operand MUXes.

There are other important pieces of logic that we do not consider in this report, even though their delay is a function of dispatch/issue width.

- *Register file*

  The register file provides low latency access to register operands. The access time of the register file is a function of the number of physical registers and the number of read and write ports. Farkas et. al. [11] study how the access time of the register file varies with the number of registers and the number of ports. Because it is studied elsewhere, we do not include it here.

- *Caches*

  The instruction and data caches provide low latency access to instructions and memory operands respectively. In order to provide the necessary load/store bandwidth in a superscalar processor, the cache has to be banked or duplicated. The access time of a cache is a function of the size of the cache and the associativity of the cache. Wada et. al. [31] and Wilton and Jouppi [33] have developed detailed models that estimate the access time of a cache given its size and associativity. Again, because it is studied elsewhere, we do not consider cache logic in this report.

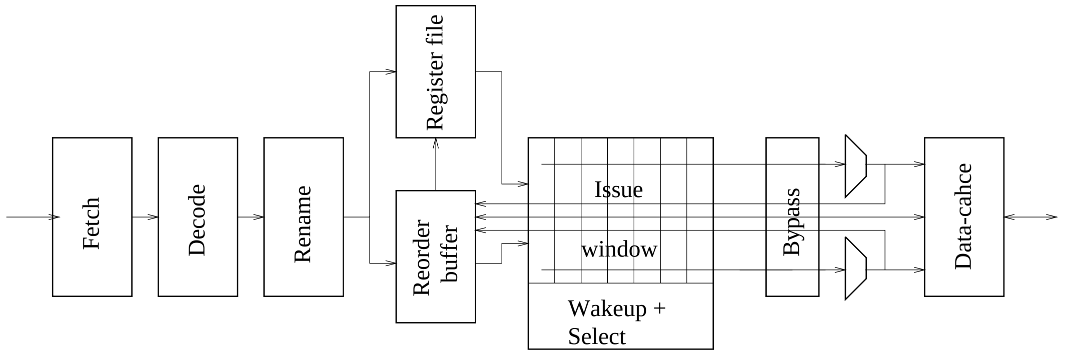
- *Instruction fetch logic*

  Instruction caches are discussed above. However, there are other important parts of fetch logic whose complexity varies with instruction dispatch/issue width. First of all, as instruction issue widths grow beyond the size of a single basic block, it will become necessary to predict multiple branches per cycle. Then, non-contiguous blocks of instructions will have to be fetched from the instruction cache and compacted into a contiguous block prior to renaming. The logic required for these operations are described in some detail in [26]. However, delay models remain to be developed. And, although they are important, we chose not to consider them here.

Finally, we must point out once again that in real designs there may be structures not listed above that may influence the overall delay of the critical path. However, our realistic aim is not to study all of them but to analyze in detail some important ones that have been reported in the literature. We believe that our basic techniques can be applied to others, however.

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 [34], the HP PA-8000 [19], and the DEC 21264 [18] are three implementations of this model. Hence, the structures identified above apply to these three processors.



FETCH	DECODE	RENAME	REG READ ROB READ INSERT	WAKEUP SELECT	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

Figure 2: Reservation station model

On the other hand, the Intel Pentium Pro [13], the PowerPC 604 [7], and the HAL SPARC64 [12] are based on the reservation station model shown in Figure 2. There are two main differences between the two models. First, in the baseline model all the register values, both speculative and non-speculative, reside in the physical register file. In the reservation station model, the reorder buffer holds speculative values and the register file holds only committed, non-speculative data. Second, operand values are not broadcast to the window entries in the baseline model – only their tags are broadcast; data values go to the physical register file. In the reservation station model completing instructions broadcast operand values to the reservation station. Issuing instructions read their operand values from the reservation station.

The point to be noted is that the basic structures identified earlier are also present in the reservation station model and are as critical as in the baseline model. The only notable difference is that the reservation station model has a smaller physical register file (equal to the number of architected registers) and might not demand as much bandwidth (as many ports) as the register file as the baseline model, because in this case some of the operands come from the reorder buffer and the reservation station.

While the discussion about potential sources of complexity is in the context of a baseline superscalar model that is out-of-order, it must be pointed out that some of the critical structures identified apply to in-order processors too. For example, the dependence check and bypass logic are present in in-order superscalar processors.

3 Methodology

We studied each structure in two phases. In the first phase, we selected a representative CMOS circuit. This was done by studying designs published in the literature (mainly proceedings of the ISSCC - International Solid-State and Circuits Conference) 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 select one that was most promising. In one case, register renaming, we had to study (simulate) two different schemes whose performance was similar.

In the second phase we implemented the circuit and optimized the circuit for speed. We used the HSPICE circuit simulator [22] from MetaSoftware to simulate the circuits. We mostly used static logic. However, in situations where dynamic logic helped in boosting the performance significantly, we used dynamic logic. For example, in the wakeup logic, we used a dynamic 7-input NOR gate 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, we applied logic optimizations like two-level decomposition to reduce fan-in requirements. We avoided using static gates with a fan-in greater than four. Third, in some cases we had to modify the transistor ordering to shorten the critical path. Some of the optimization sites will be pointed out when the individual circuits are described.

In order to simulate the effect of wire parasitics, we added these parasitics 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. The process parameters for the $0.8\mu m$ CMOS process were taken from [16]. These parameters were used by Wilton and Jouppi in their study of cache access times [33]. Because process parameters are proprietary information, we had to use extrapolation to come up with process parameters for the $0.35\mu m$ and $0.18\mu m$ technologies. We used the $0.8\mu m$ process parameters, $0.5\mu m$ process parameters from MOSIS, and process parameters used in the literature as inputs. The process parameters assumed for the three technologies are listed in Appendix A. Layouts for the $0.35\mu m$ and $0.18\mu m$ technologies were obtained by appropriately shrinking the layout for the $0.8\mu m$ technology.

Finally, we used basic RC circuit analysis to develop simple analytical models that captured the depen-

Symbol	Represents
IW	Issue width
$WINSIZE$	Window size
$NVREG$	Number of virtual registers
$NPREG$	Number of physical registers
$NVREG_{width}$	Number of bits in virtual register designators
$NPREG_{width}$	Number of bits in physical register designators
R_{metal}	Resistance of metal wire per unit length
C_{metal}	Parasitic capacitance of metal wire per unit length

Table 1: Terminology

dence of the delays on microarchitectural parameters like issue width and window size. We compared the relationships predicted by the Hspice simulations against those predicted by our model. In most of the cases, our models were accurate in identifying the relationships.

3.1 Caveats

The above methodology does not address the issue of how well the assumed circuits reflect real circuits for the structures. However, by basing our circuits on designs published by microprocessor vendors, we believe that the assumed circuits are close enough to real circuits. In practice, many circuit tricks could be employed to optimize the critical path for speed. However, we believe that the relative delay times between different configurations should be more accurate than the absolute delay times. Because we are mainly interested in finding trends as to how the delays of the structures vary with microarchitectural parameters like window size and issue width, and how the delays scale as the feature size is reduced, we believe that our results are valid.

3.2 Terminology

Table 1 defines some of the common terms used in the report. The remaining terms will be defined when they are introduced.

4 Technology Trends

Feature sizes of MOS devices have been steadily decreasing. This trend towards smaller devices is likely to continue at least for the next decade [3]. In this section, we briefly discuss the effect of shrinking feature sizes on circuit delays. The effect of scaling feature sizes on circuit performance is an active area of research [8, 21]. We are only interested in illustrating the trends in this section.

Circuit delays consist of logic delays and wire delays. Logic delays are delays resulting from gates that are driving other gates. The delay of a decoder that consists of NAND gates feeding NOR gates is an example of logic delay. Wire delays are the delays resulting from driving values on wires.

Logic delays
The delay of a logic gate can be written as

$$Delay_{gate} = (C_L \times V)/I$$

where C_L is the load capacitance at the output of the gate, V is the supply voltage, and I is the average charging/discharging current. I is a function of I_{dsat} - the saturation drain current of the devices forming the gate. As the feature size is reduced, the supply voltage has to be scaled down to keep the power consumption at manageable levels. Because voltages cannot be scaled arbitrarily they follow a different scaling curve from feature sizes. From [24], for submicron devices, if S is the scaling factor for feature sizes, and U is the scaling factor for supply voltages, then C_L, V, and I scale by factors of $1/S$, $1/U$, and $1/U$ respectively. Hence, the overall gate delay scales by a factor of $1/S$. Therefore, gate delays decrease uniformly as the feature size is reduced.

Wire delays
If L is the length of a wire, then the intrinsic RC delay of the wire is given by

$$Delay_{wire} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

where R_{metal}, C_{metal} are the resistance and parasitic capacitance of metal wires per unit length respectively and L is the length of the wire. The factor 0.5 is introduced because we use the first order approximation that the delay at the end of a distributed RC line is RC/2 (we assume the resistance and capacitance are distributed uniformly over the length of the wire).

In order to study the impact of shrinking feature sizes on wire delays we first have to analyze how the resistance, R_{metal}, and the parasitic capacitance, C_{metal}, of metal wires vary with feature size. We use the simple model presented by Bohr in [4] to estimate how R_{metal} and C_{metal} scale with feature size. Note that both these quantities are per unit length measures. From [4],

$$\begin{aligned} R_{metal} &= \rho/(width * thickness) \\ C_{metal} &= C_{fringe} + C_{parallel-plate} \\ &= 2 * \epsilon * \epsilon_0 * thickness/width + 2 * \epsilon * \epsilon_0 * width/thickness \end{aligned}$$

where $width$ is the width of the wire, $thickness$ is the thickness of the wire, ρ is the resistivity of metal, and ϵ and ϵ_0 are permittivity constants.

The average metal thickness has remained constant for the last few generations while the width has been decreasing in proportion to the feature size. Hence, if S is the technology scaling factor, the scaling factor for R_{metal} is S. The metal capacitance consists of two components: fringe capacitance and parallel-plate capacitance. Fringe capacitance is the result of capacitance between the side-walls of adjacent wires and capacitance between the side-walls of the wires and the substrate. Parallel-plate capacitance is the result of capacitance between the bottom-wall of the wires and the substrate. Assuming that the thickness remains constant, it can be seen from the equation for C_{metal} that the fringe component becomes the dominant component as we move towards smaller feature sizes. In [25], the authors show that as features sizes are reduced, the fringe capacitance will be responsible for an increasingly larger fraction of the total capacitance. For example, they show that for feature sizes less than $0.1\mu m$, the fringe capacitance contributes 90% of the total capacitance. In order to accentuate the effect of wire delays and to be able to identify their effects, we assume that the metal capacitance is largely determined by the fringe capacitance and therefore the scaling factor for C_{metal} is also S.

Using the above scaling factors in the equation for the wire delay we can compute the scaling factor for wire delays as

$$\begin{aligned} Scaling\ factor &= S \times S \times (1/S)^2 \\ &= 1 \end{aligned}$$

Note that the length scales as $1/S$ for local interconnects. In this study we are only interested in local interconnects. This might not be true for global interconnects like the clock because their length also depends on the die size.

Hence, as feature sizes are reduced, the wire delays remain constant. This coupled with the fact that logic delays decrease uniformly with feature size implies that wire delays will dominate logic delays in future. In reality the situation is further aggravated for two reasons. First, not all wires reduce in length perfectly (by a factor of S). Second, some of the global wires, like the clock, actually increase in length due to bigger dice that are made possible with each generation.

McFarland and Flynn [21] studied various scaling schemes for local interconnects and conclude that quasi-ideal scaling scheme as the one that closely tracks future deep submicron technologies. Quasi-ideal scaling performs ideal scaling of the horizontal dimensions but scales the thickness more slowly. The scaling factor for RC delay per unit length for their scaling model is $(0.9 \times S^{1.5} + 0.1 \times S^{2.5})$. In comparison, for our scaling model, the scaling factor for RC delay per unit length is simply S^2. Even though our model overestimates the RC delay as compared to the quasi-ideal model of McFarland and Flynn, we use it in order to emphasize wire delays and study their effects.

5 Complexity Analysis

In this section we discuss the critical pipeline structures in detail. The presentation for each structure is organized as follows. First, we describe the logical function implemented by the structure. Then, we present possible schemes for implementing the structure and describe one of the schemes in detail. Next we analyze the overall delay of the structure in terms of microarchitectural parameters like issue width and window size using simple delay models. Finally, we present Spice results, identify trends in the results and discuss how the results conform to the delay analysis performed earlier.

5.1 Register Rename Logic

The register rename logic is used to translate logical register designators into physical register designators. Logically, this is accomplished by accessing a map table with the logical register designator as the index. Because multiple instructions, each with multiple register operands, need to be renamed every cycle, the map table has to be multi-ported. For example, a 4-wide issue machine with two read operands and one write operand per instruction requires 8 read ports and 4 write ports to the mapping table. The high level block diagram of the rename logic is shown in Figure 3. The map table holds the current logical to physical mappings. 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. An example of this is shown in Figure 4. The dependence check logic detects such dependences and sets up the output MUXes so that the appropriate physical register designators are generated. The shadow table is used to checkpoint old mappings so that the processor can quickly recover to a precise state [27] from branch mispredictions [1]. 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.

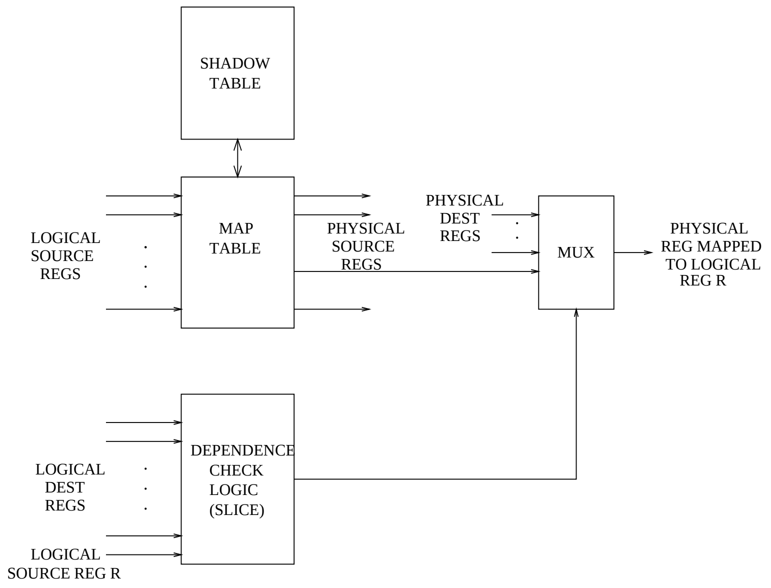


Figure 3: Register Rename Logic

5.1.1 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, as implemented in the MIPS R10000 [34], the map table is a register file where each entry contains the physical register that is mapped to the logical register whose designator is used to index the table. The number of entries in the map table is equal to the number of logical registers. A single cell of the table is shown in Figure 5. A shift register, present in every cell, is used for checkpointing old mappings.

[1]This mechanism can be used to recover from exceptions other than branch mispredicts. However, because they occur less frequently and checkpoint space is limited, we assume that checkpointing is used only for predicted branches. Other exceptions are recovered from by unwinding the reorder buffer.

The map table works like a register file. The bits of the physical register designators are stored in the cross-coupled inverters in each cell. A read operation starts with the logical register designator being applied to the decoder. The decoder decodes the logical register designator and raises one of the word lines. This triggers bit line changes which are sensed by a sense amplifier and the appropriate output is generated. Precharged bit lines are used to improve the speed of read operations. Single-ended read and write ports are used to minimize the increase in width of each cell as the number of ports is increased because the width of each cell determines the length of the wordlines and hence the time taken to drive the wordlines.

Mappings are checkpointed by copying the current contents of each cell into the shift register. Recovery is performed by writing the bit in the appropriate shift register cell back into the main cell.

CAM scheme

An alternative scheme for register renaming uses a CAM (content-addressable memory [32]) to store the current mappings. Such a scheme is implemented in the HAL SPARC [2] and the DEC 21264 [18]. The number of entries in the CAM is equal to the number of physical registers. Each entry contains two fields. The first field stores the logical register designator that is mapped to the physical register represented by the entry. The second field contains a valid bit that is set if the current mapping is valid. The valid bit is required because a single logical register might map to more than one physical register. When a mapping is changed, the logical register designator is written into the entry corresponding to a free physical register and the valid bit of the entry is set. At the same time, the valid bit of the mapping used for the previous mapping is located through an associative search and cleared.

The rename operation in this scheme proceeds as follows. The CAM is associatively searched with the logical register designator. If there is a match and the valid bit is set, a read enable word line corresponding to the CAM entry is activated. An encoder (ROM) is used to encode the read enable word lines (one per physical register) into a physical register designator. Old mappings are checkpointed by storing the valid bits from the CAM into a checkpoint RAM. To recover from an exception, the valid bits corresponding to the old mapping are loaded into the CAM from the checkpoint RAM. In the HAL design, up to 16 old mappings can be saved.

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 [2]. In order to support such a large number of physical registers, the CAM will have to be appropriately banked. On the other hand, in the RAM scheme, the number of entries in the map table is independent of the number of physical registers. However, the CAM scheme has an advantage with respect to checkpointing. In order to checkpoint in the CAM scheme, only the valid bits have to be saved. This is easily implemented by having a RAM adjacent to the column of valid bits in the CAM. In other words, the dimensions of individual CAM cells is independent of the number of checkpoints. On the other hand, in the RAM scheme, the width of individual cells is a function of the number of checkpoints because this number determines the length of the shift register in each cell.

The dependence check logic, shown in Figure 4, proceeds in parallel with the map table access. Every

[2] Farkas et. al. [11] have shown that for significant performance up to 80 physical registers are required for a 4-wide issue machine and up to 120 physical registers are required for a 8-wide issue machine.

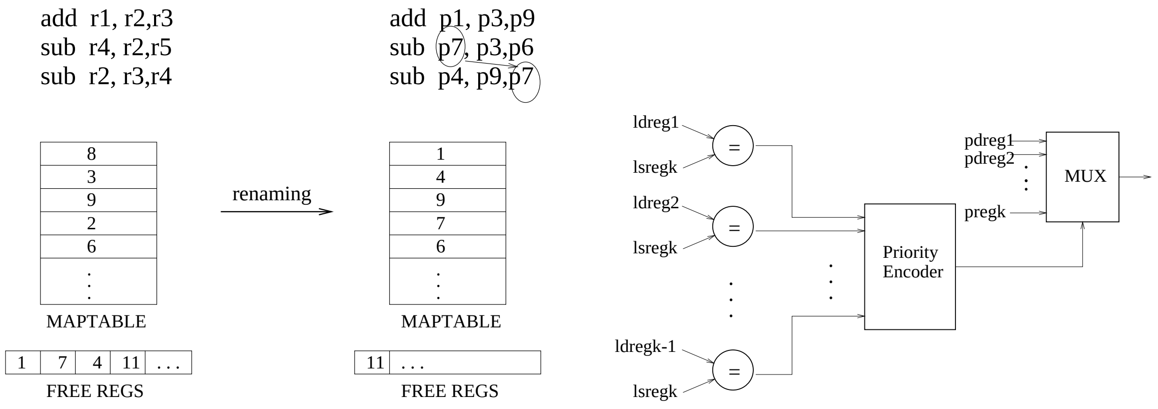


Figure 4: Renaming example and dependence check logic

logical register designator being renamed is compared against the destination register designators (logical) of earlier instructions in the current rename group. If there is a match, then the tag corresponding to the physical register assigned to the earlier instruction is used instead of the tag read from the map table. For example, in the case shown in Figure 4, the last instruction's operand register r4 is mapped to p7 and not p2. In the case of more than one match, the tag corresponding to the latest (in dynamic order) match is used. We implemented the dependence check logic for issue widths of 2, 4, and 8. 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.

5.1.2 Delay Analysis

We implemented both the RAM scheme and the CAM scheme. We found the performance of the two schemes to be comparable for the design space we explored. To keep the analysis short, we will only discuss the RAM scheme here.

A single cell of the map table is shown in Figure 5. The critical path for the rename logic is the time it takes for the bits of the physical register designator to be output after the logical register designator is applied to the address decoder. The delay of the critical path consists of four components: the time taken to decode the logical register designator, the time taken to drive the wordline, the time taken by an access stack to pull the bitline low, and the time taken by the sense amplifier to detect the change in the bitline and produce the corresponding output. The time taken for the output of the map table to pass through the output MUX is ignored because this is small compared to the rest of the rename logic and, more importantly, the control input of the MUX is available in advance because the dependence check logic is faster than the map table. Hence, the overall delay is given by,

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

Each of the components is analyzed next.

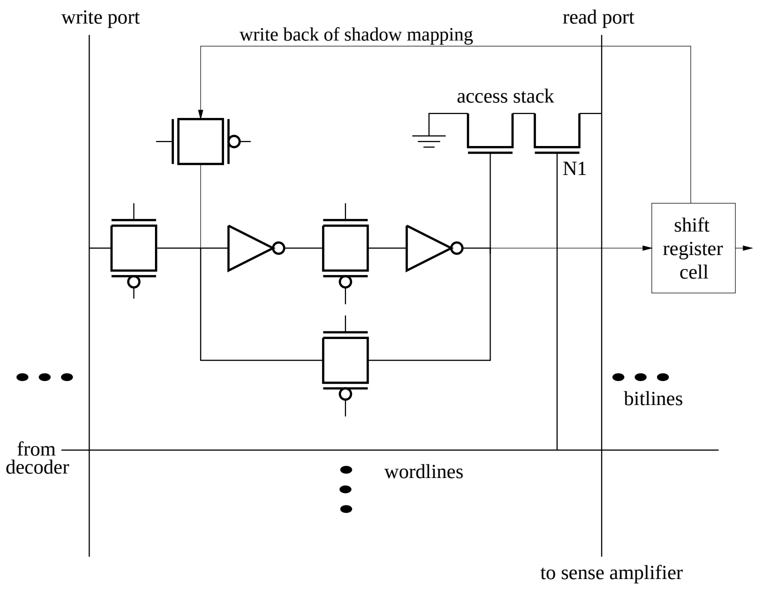


Figure 5: Map table cell

Decoder delay

The structure of the decoder is shown in Figure 6. We use predecoding [32] to improve the speed of decoding. A 3-bit predecode field generates 8 predecode lines, each of which is fed to 4 row decode gates. The predecode gates are 3-input NAND gates and the row decode gates are 3-input NOR gates. The fan-in of the NAND and NOR gates are determined by the number of bits in the logical register designator. The output of the NAND gates is connected to the input of the NOR gates by the predecode lines. The length of these lines is given by

$$PredeclineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NVREG$$

where $cellheight$ is the height of a single cell excluding the wordlines, IW is the issue width, $wordline_{spacing}$ is the spacing between wordlines, and $NVREG$ is the number of logical registers. The factor 3 in the equation results from the assumption of 3-operand instructions (2 read operands and 1 write operand), and single-ended read/write ports. With these assumptions, 3 ports (1 write port and 2 read ports) are required per cell for each instruction being renamed. Hence, for a IW-wide issue machine, a total of $3 \times IW$ wordlines are required for each cell.

The decoder delay is the time it takes to decode the logical register designator i.e. the time it takes for the output of the NOR gate to rise after the input to the NAND gate has been applied. Hence, the decoder delay can be written as

$$T_{decode} = T_{nand} + T_{nor}$$

where T_{nand} is the fall delay of the NAND gate and T_{nor} is the rise delay of the NOR gate. From the equivalent circuit of the NAND gate shown in Figure 6

$$T_{nand} = c_0 \times R_{eq} \times C_{eq}.$$

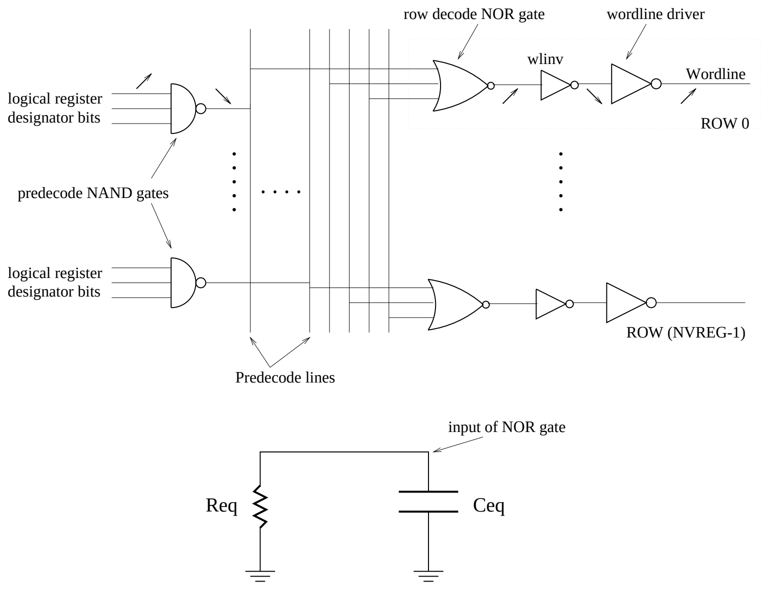


Figure 6: Decoder structure

R_{eq} consists of two components: the resistance of the NAND pull-down and the metal resistance of the predecode line connecting the NAND gate to the NOR gates. Hence,

$$R_{eq} = R_{nandpd} + 0.5 \times PredeclineLength \times R_{metal}$$

Note that we have divided the resistance of the predecode line by two; the first order approximation for the delay at the end of a distributed RC line is RC/2 (we assume the resistance and capacitance are distributed evenly over the length of the wire).

C_{eq} consists of three components: the diffusion capacitance of the NAND gate, the gate capacitance of the NOR gate, and the metal capacitance of the line connecting the line connecting the NAND gate to the NOR gate. Hence,

$$C_{eq} = C_{diffcap-nand} + C_{gatecap-nor} + PredeclineLength \times C_{metal}$$

Substituting the above equations into the overall decoder delay and simplifying, we get

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

where c_0, c_1, c_2 are constants. The quadratic component results from the intrinsic RC delay of the predecode lines connecting the NAND gates to the NOR gates. We found that, at least for the design space and technologies we explored, the quadratic component is very small relative to the other components. Hence, the delay of the decoder is linearly dependent on the issue width.

Wordline delay

The wordline delay is defined as the time taken to turn on all the access transistors (denoted by N1 in Figure 5) connected to the wordline after the logical register designator has been decoded. From the circuit shown in

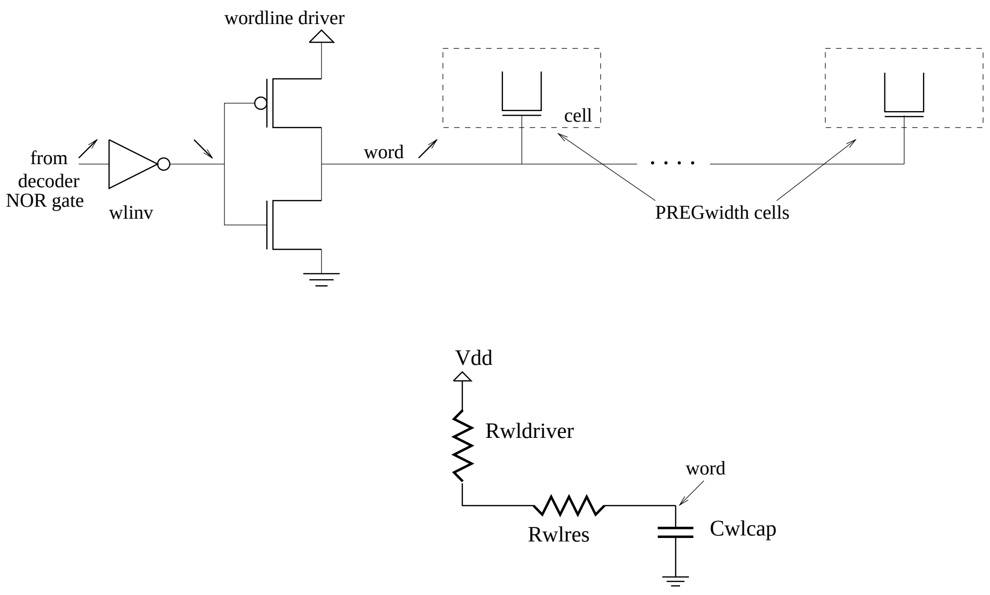


Figure 7: Wordline structure

Figure 7, the wordline delay is the sum of the fall delay of the inverter *wlinv* and the rise delay of the wordline driver. Hence,

$$T_{wordline} = T_{wlinv} + T_{wldriver}$$

From the equivalent circuit of the wordline driver shown in Figure 7, the wordline driver delay can be written as

$$T_{wldriver} = c_0 \times (R_{wldriver} + R_{wlres}) \times C_{wlcap}$$

where $R_{wldriver}$ is the effective resistance of the pull-up (p-transistor) of the driver, R_{wlres} is the resistance of the wordline, and C_{wlcap} is the amount of capacitance on the wordline. The total capacitance on the wordline consists of two components: the gate capacitance of the access transistors and the metal capacitance of the wordline wire. The resistance of the wordline is determined by the length of the wordline. Symbolically,

$$WordlineLength = (cellwidth + 3 \times IW \times bitline_{spacing} + B \times shiftreg_{width}) \times \ PREG_{width}$$

$$C_{wlcap} = PREG_{width} \times C_{gatecap-N1} + WordlineLength \times C_{metal}$$

$$R_{wlres} = 0.5 \times WordlineLength \times R_{metal}$$

where $PREG_{width}$ is the number of bits in the physical register designator, $C_{gatecap-N1}$ is the gate capacitance of the access transistor N1 in each cell, $cellwidth$ is the width of a single RAM cell excluding the bitlines, $bitline_{spacing}$ is the spacing between bitlines, B is the maximum number of shadow mappings that can be checkpointed, and $shiftreg_{width}$ is the width of a single bit of the shift register in each cell.

Factoring the above equations into the wordline delay equation and simplifying we get

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

where c_0, c_1, and c_2 are constants. Again, the quadratic component results from the intrinsic RC delay of the wordline wire and we found that the quadratic component is very small relative to the other components. Hence, the overall wordline delay is linearly dependent on the issue width.

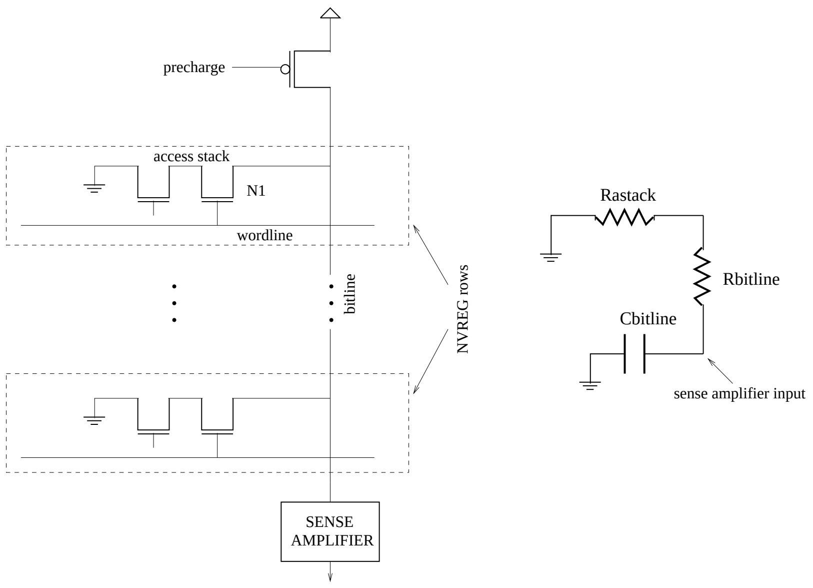


Figure 8: Bitline structure

Bitline delay

The bitline delay is defined as the time between the wordline going high (turning on the access transistor N1) and the bitline going low (reaching a voltage of $V_{bitsense}$ below its precharged value of V_{dd} where $V_{bitsense}$ is the threshold voltage of the sense amplifier.). This is the time it takes for one access stack to discharge the bitline. From the equivalent circuit shown in Figure 8 we can see that the magnitude of the delay is given by

$$T_{bitline} = c_0 \times (R_{astack} + R_{bitline}) \times C_{bitline}$$

where R_{astack} is the effective resistance of the access stack (two pass transistors in series), $R_{bitline}$ is the resistance of the bitline, and $C_{bitline}$ is the capacitance on the bitline. The bitline capacitance consists of two components: the diffusion capacitance of the access transistors connected to the bitline and the metal capacitance of the bitline. The resistance of the bitline is determined by the length of the bitline. Symbolically,

$$BitlineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NVREG$$

$$C_{bitline} = NVREG \times C_{diffcap-N1} + BitlineLength \times C_{metal}$$

$$R_{bitline} = 0.5 \times BitlineLength \times R_{metal}$$

where $NVREG$ is the number of logical registers, $C_{diffcap-N1}$ is the diffusion capacitance of the access transistor N1 that connects to the bitline, $cellheight$ is the height of a single RAM cell excluding the wordlines, and $wordline_{spacing}$ is the spacing of wordlines.

Factoring the above equations into the overall delay equation and simplifying we get

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

where c_0, c_1, and c_2 are constants. Again, we found that the quadratic component is very small relative to the other components. Hence, the overall bitline delay is linearly dependent on the issue width.

Sense amplifier delay

We used Wada's sense amplifier from [31]. Wada's sense amplifier amplifies a voltage difference of $2 \times V_{bitsense}$ to V_{dd}. Because we assumed single-ended read lines, we tied one of the inputs of the sense amplifier to a reference voltage V_{ref}. Even though the structural constitution of the sense amplifier is independent of the issue width, we found that its delay varied with issue width because its delay is a function of the slope of the input. Because the input here is the bitline voltage, the delay of the sense amplifier is a function of the bitline delay. This in turn makes the delay of the sense amplifier a function of the issue width.

Overall delay

From the above analysis, the overall delay of the register rename logic can be summarized by the following equation:

$$Delay = c_0 + c_1 \times IW + c_2 \times IW^2$$

where c_0, c_1 and c_2 are constants. However, the quadratic component is relatively small and hence, the rename delay is a linear function of the issue width for the design space we explored.

5.1.3 Spice Results

Figure 9 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 also shows the breakup of the delay into the components discussed in the previous section. Detailed results for various configurations and technologies are shown in tabular form in Appendix B.

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 the analysis in the previous section. All the components show 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 with issue width only 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 $0.18\mu m$. This occurs because logic delays in the various components are reduced in proportion to the feature size while the presence of wire delays in the

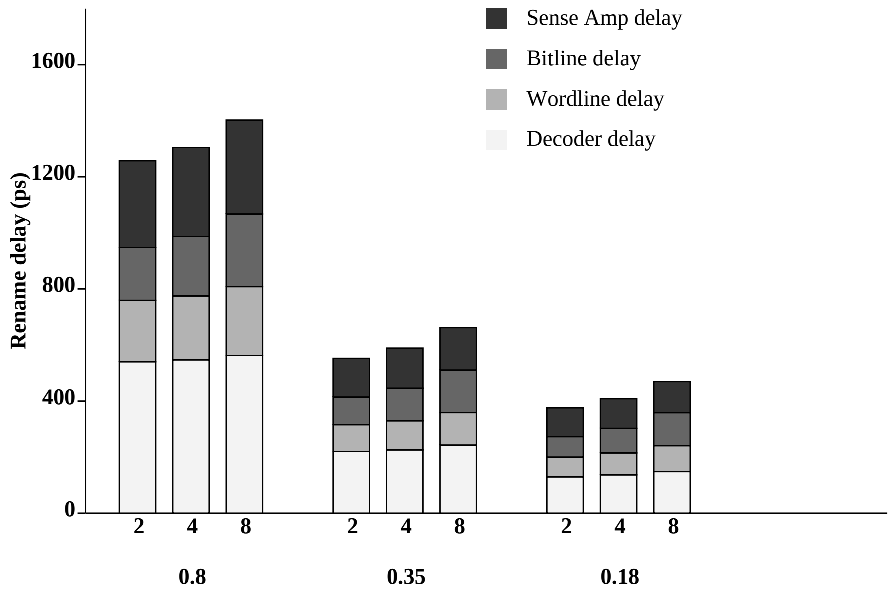


Figure 9: Rename delay versus issue width

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.

5.2 Wakeup Logic

The wakeup logic is responsible for updating source dependencies of instructions in the issue window waiting for their source operands to become available. Figure 10 illustrates the wakeup logic. Every time a result is produced, the 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 rdy flag is set to indicate this. The issue window is a CAM (content addressable memory [32]) array holding one instruction per entry. Buffers, shown at the top of the figure, are used to drive the result tags $tag1$ to $tagW$ where W is the issue width. Each entry of the CAM has $2 \times W$ 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.

5.2.1 CAM Structure

Figure 11 shows a single cell of the CAM array. The cell shown in detail compares a single bit of the operand tag with the corresponding bit of the result tag. The operand tag bit is stored in the RAM cell. The corresponding bit of the result tag is driven on the tag lines. The match line is precharged high. If there is a mismatch between the operand tag bit and the result tag bit, the match line is pulled low by one of the pull-down stacks. For example, if $tag = 0$, and $data = 1$, then the pull-down stack on the left is turned on and

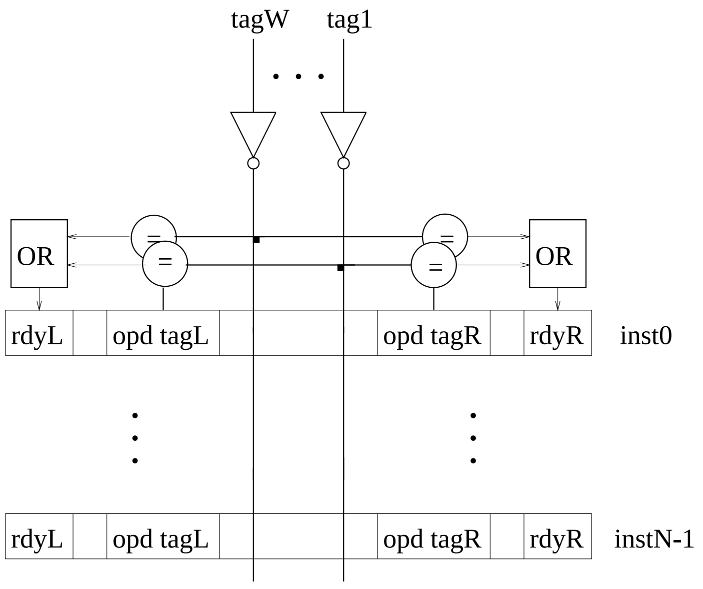


Figure 10: Wakeup Logic

it pulls the match line low. The pull-down stacks constitute the comparators shown in Figure 10. The match line extends across all the bits of the tag i.e. a mismatch in any of the bit positions will pull it low. In other words, the match line remains high only if the result tag matches the operand tag. The above operation is repeated for each of the result tags by having multiple tag and match lines as shown in the figure. Finally, all the match signals are ORed to produce the ready signal.

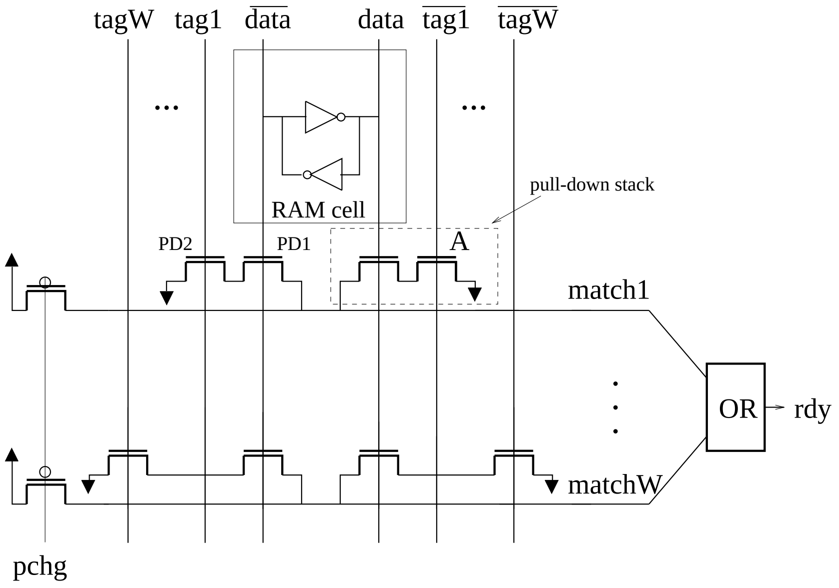


Figure 11: CAM cell

There are two of observations that can be drawn from the figure. First, there are as many match lines as the issue width. Hence, increasing issue width increases the height of each CAM row. Second, increasing issue width also increases the number of inputs to the OR block.

5.2.2 Delay Analysis

Because the match lines are precharged high, the default value of the ready signal is high. Hence, the delay of the critical path is the time it takes for a mismatch in a single bit position to pull the ready signal low. The delay consists of three components: the time taken by the buffers to drive the tag bits, the time taken for the pull-down stack corresponding to the bit position with the mismatch to pull the match line low [3], and the time taken to OR the individual match signals. Symbolically,

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

Each of the components is analyzed next.

Tag_drive_time

The tag drive circuit is shown in Figure 12. The time taken to drive the tags depends on the length of the tag lines. The length of the tag lines is given by

$$TaglineLength = (camheight + IW \times matchline\_spacing) \times WINSIZE$$

where $camheight$ is the height of a single CAM cell excluding the matchlines, and $matchline_{spacing}$ is the spacing between matchlines [4], and $WINSIZE$ is the number of window entries.

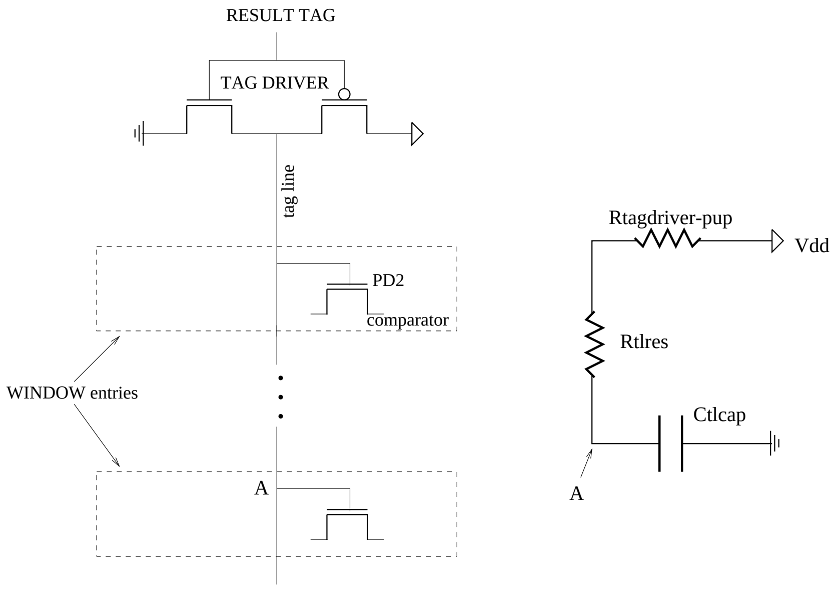


Figure 12: Tag drive structure

From the equivalent circuit shown in Figure 12, the time taken to drive the tags is given by

$$T_{tagdrive} = c_0 \times (R_{tagdriver-pup} + R_{tlres}) \times C_{tlcap}$$

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

[4]To be precise $matchline_{spacing}$ is the height of a matchline and the associated pull-down stacks.

where $R_{tagdriver-pup}$ is the resistance of the pull-up of the tag driver, R_{tlres} is the metal resistance of the tag line, and C_{tlcap} is the total capacitance on the tag line. R_{tlres} is given by

$$R_{tlres} = 0.5 \times TaglineLength \times R_{metal}$$

C_{tlcap} consists of three components: the metal capacitance determined by the length of the tag line, the gate capacitances of the comparators, and the diffusion capacitance of the tag driver.

$$C_{tlcap} = TaglineLength \times C_{metal} + C_{gatecap-comp} \times WINSIZE + C_{diffcap-tagdriver}$$

where $C_{gatecap-comp}$ is the gate capacitance of the pass transistor PD2 (shown in Figure 11) in the comparator's pull-down stack and $C_{diffcap-tagdriver}$ is the diffusion capacitance of the tag driver.

Substituting the above equations into the overall delay equation and simplifying we get

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

Note that we used a fixed size tag driver in our studies. The tag driver was sized for the largest configuration. Hence, in reality the increase in tag drive time with window size will be higher.

The above equation shows that the tag drive time increases with window size and issue width. For a given issue width, the total delay is a quadratic function of the window size. The weighting factor of the quadratic term is a function of the issue width. We found that 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. We found that 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. Hence, we believe that the tag drive time can become significant in future designs with wider issue widths, bigger windows, and smaller feature sizes.

Tag_match_time

This is the time taken for one of the pull-down stacks to pull the match line low. From the equivalent circuit shown in Figure 13,

$$T_{tagmatch} = c_0 \times (R_{pdstack} + R_{mlres}) \times C_{mlcap}$$

where $R_{pdstack}$ is the effective resistance of the pull-down stack, R_{mlres} is the metal resistance of the match line, and C_{mlcap} is the total capacitance on the match line. R_{mlres} can be computed using

$$R_{mlres} = 0.5 \times MatchlineLength \times R_{metal}$$

where $MatchlineLength$ is the length of the match line and is given by

$$MatchlineLength = (camwidth + IW \times tagline_{spacing}) \times PREG_{width}$$

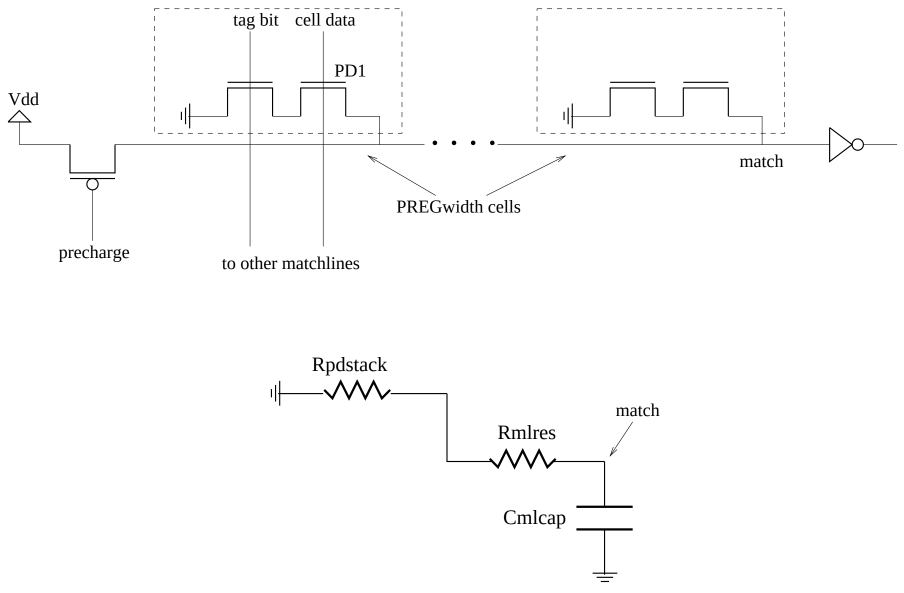


Figure 13: Tag match structure

$camwidth$ is the width of the CAM cell excluding the tag lines, $tagline\_spacing$ is the increase in the width of the CAM cell for each extra port (tag and tagbar wires) added.

C_{mlcap} consists of three components: the diffusion capacitances of all the pull-down stacks connected to the match line, the metal capacitance of the match line, and the gate capacitance of the inverter at the end of the match line. Hence,

$$C_{mlcap} = 2 \times PREG_{width} \times C_{diffcap-PD1} + MatchlineLength \times C_{metal} + C_{gatecap\_matchinv}$$

where $PREG_{width}$ is the width of the physical register designators, $C_{diffcap-PD1}$ is the diffusion capacitance of the pass transistor (marked as PD1 in Figure 11) in the pull-down stacks that is connected to the match line, and $C_{gatecap\_matchinv}$ is the gate capacitance of the inverter at the end of the match line.

Substituting the equations for R_{mlres} and C_{mlcap} into the overall delay equation and simplifying we get

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

Again, we found that the quadratic component is relatively small and hence, the tag match time is a linear function of the issue width.

A drawback of our model for the tag match time is that it does not model the dependence of the match time on the slope of the tag line signal i.e. the tag drive delay. Our results, presented in the next section, show that, as a result of this dependence, the tag match time is also a function of the window size. In other words, larger windows will result in slower fanning out of the result tags to the comparators in the window entries, thus increasing the compare time.

Match_OR_time

This is the time taken to OR the individual match lines to produce the ready signal. Because the number of

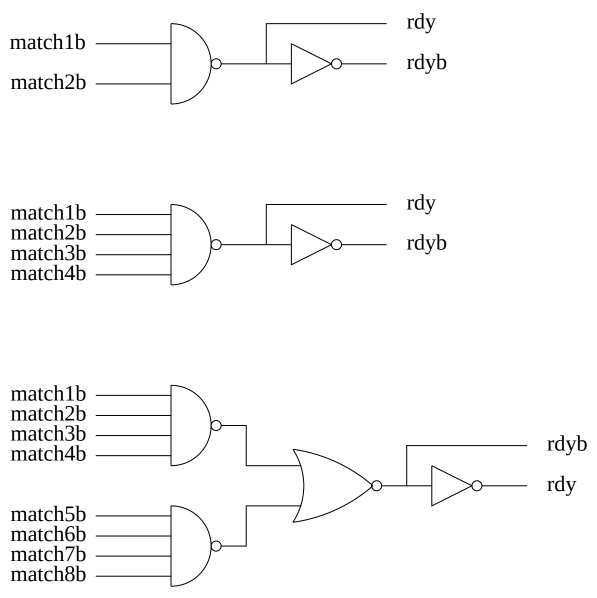


Figure 14: Logic for ORing individual match signals

match lines is the same as the issue width, the magnitude of this delay term is a direct function of issue width. Figure 14 shows the OR logic for result widths of 2, 4, and 8. For $W = 8$, we use two 4-input NAND stacks followed by a NOR gate because this is faster than using an 8-input NAND gate. Because the delay of a gate is a quadratic function of the fan-in [32, 24] we can write the delay as

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

For the design space we explored (issue widths of 2, 4, and 8), the quadratic component was relatively small.

Overall delay

The overall delay of the wakeup logic can be summarized by an equation similar to the tag drive time because it includes all the relations exhibited by the tag match time and match OR time. Therefore,

$$Delay = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c_5 \times IW^2) \times WINSIZE^2$$

5.3 Spice Results

The graph on the left in Figure 15 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 smaller issue widths. 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 the three components of the delay. On the other hand, increasing window size only lengthens

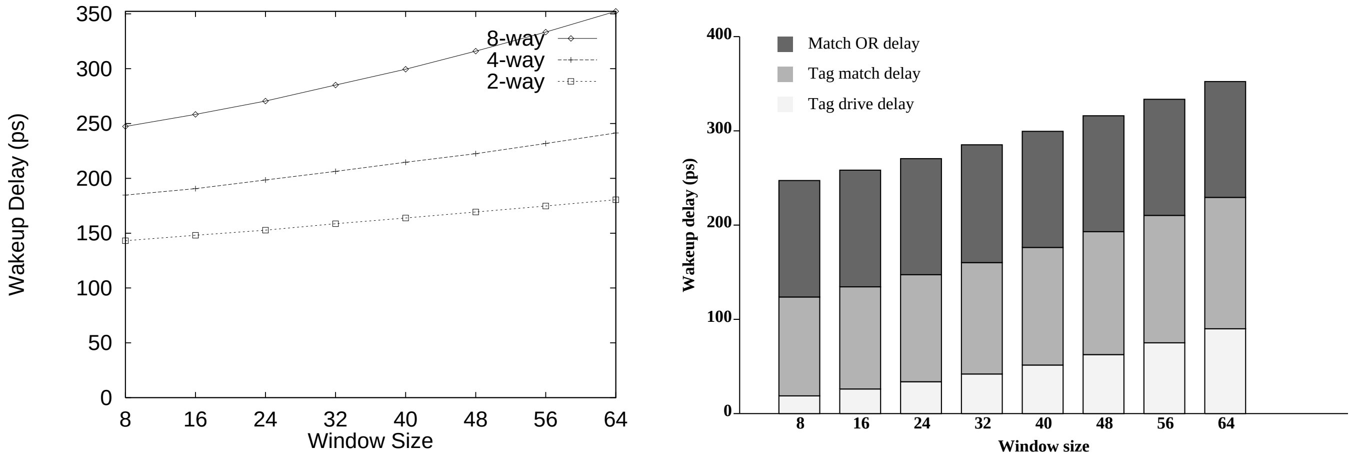


Figure 15: Wakeup logic delay versus window size

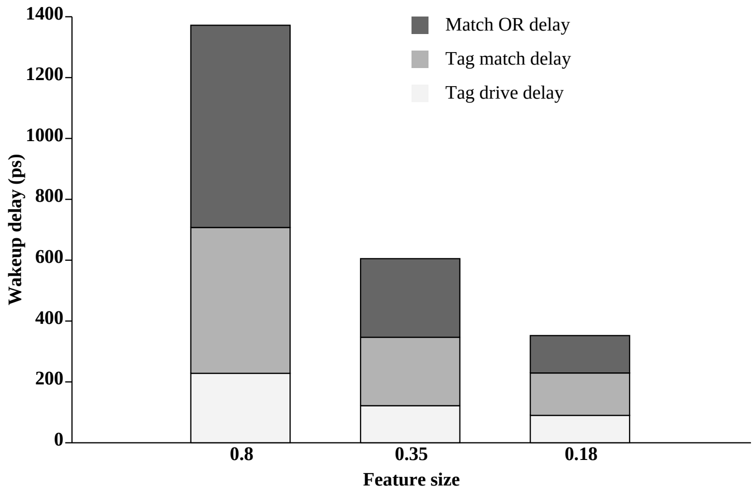


Figure 16: Wakeup delay versus feature size for an 8-way, 64-entry window processor

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. We found similar curves for $0.8\mu m$ and $0.35\mu m$ technologies. Detailed results for various configurations and technologies are shown in tabular form in Appendix B.

The bar graph on the right in Figure 15 shows the detailed breakup of the total delay for various window sizes for a 8-way processor in $0.18\mu m$ technology. The tag drive time increases rapidly with window size. For example, the tag drive time and the tag match time increase by factors of 4.78 and 1.33 respectively when the window size is increased from 8 to 64. The increase in tag drive time is higher than that of tag match time because the tag drive time is a quadratic function of the window size. The increase in tag match time with the window size is not taken into account by our simple model given above because the model does not take into consideration the slope of the input signals (determined in this case by the tag drive delay). Also, as shown by the graph, the time taken to OR the match signals only depends on the issue width and is independent of the window size.

Figure 16 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 because 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.

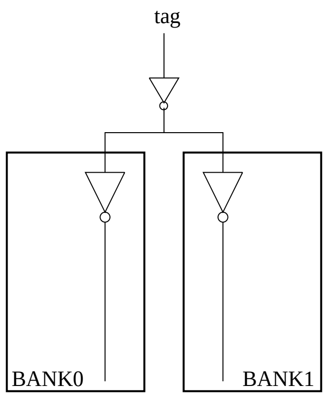


Figure 17: Banking wakeup logic

In the above simulation results the window size was limited to a maximum of 64 instructions because we found that for larger windows the intrinsic RC delay of the tag lines increases significantly. As discussed previously, the intrinsic RC delay is proportional to the square of the window size. Therefore, for implementing larger windows banking should be used as shown in Figure 17. Banking helps alleviate the intrinsic RC delay by reducing the length of the tag lines. For example, the two-way banking shown below will improve the intrinsic RC delay by a factor of four. At the same time it must be pointed out that banking will introduce some extra delay due to extra inverter stages and the parasitics introduced by the extension to the tag lines.

5.4 Selection Logic

Selection logic is responsible for selecting instructions for execution from the pool of ready instructions in the issue window. Some form of selection logic is required for two reasons. First, the number of ready instructions in the issue window can be greater than the number of functional units available. For example, for a 4-way machine with a 32-entry issue window there could be as many as 32 ready instructions. Second, some instructions can be executed only on a subset of the functional units. For example, if there is only one integer multiplier, all multiply instructions will have to be steered to that functional unit.

The inputs to the selection logic are the request (REQ) signals, one per instruction in the issue window. The request signal of an instruction is raised when all the operands of the instruction become available. As discussed in the previous section, the wakeup logic is responsible for raising the REQ signals. The outputs of the selection logic are the grant (GRANT) signals, one per request signal. On receipt of the GRANT signal, the associated instruction is issued to the functional unit and the issue window entry it occupied is freed for later use [5]. A selection policy is used to decide which of the requesting instructions is granted the functional unit. An example selection policy is *oldest ready first* - the ready instruction that occurs earliest in program order is granted the functional unit. Butler and Patt [5] studied various policies for scheduling ready instructions and found that overall performance is largely independent of the selection policy. For example, the HP PA-8000 [19] uses a selection policy that is based on the location of the instruction in the window. We assume the same selection policy in our study.

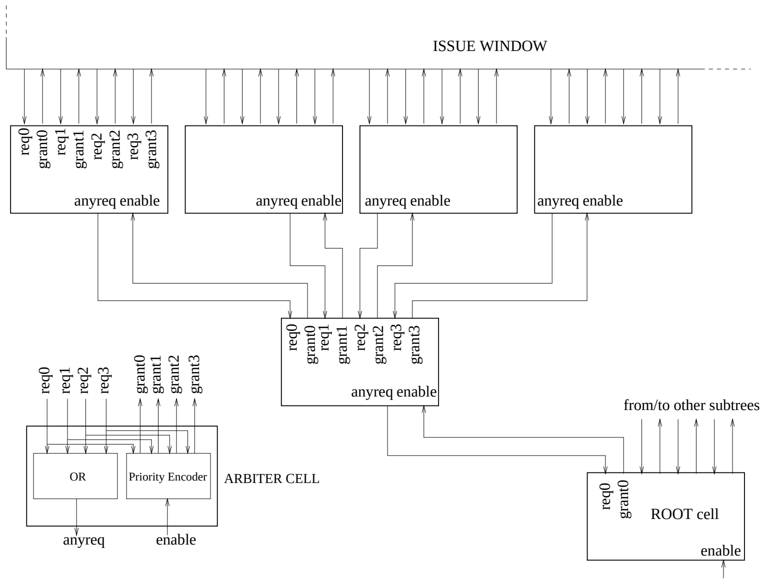


Figure 18: Selection logic

5.4.1 Structure

The assumed structure of the selection logic is shown in Figure 18. The selection logic is used to select a single instruction for execution on a functional unit. The modifications to this scheme for handling multiple functional units is discussed in the next section. The selection logic consists of a tree of arbiters. Each arbiter cell, shown in Figure 18, functions as follows. If the enable input is high, then the grant signal corresponding to the highest-priority, active input is raised. For example, if $enable = 1$, $req0 = 0$, $req1 = 1$, $req2 = 0$, and $req3 = 1$, then $grant1$ will be raised assuming priority reduces as we go from input $req0$ to input $req3$. If the enable input is low, all the grant signals are set to low. In all cases, at most one of the grant signals is high. The $anyreq$ output signal is raised if any of the input req signals is high.

The overall selection logic works in two phases. In the first phase, the request signals are propagated up the tree. Each cell raises the $anyreq$ signal if any of its input request signals is high. This in turn raises the

[5] In some designs, for example the HP PA-8000 [19], the entry is freed only after the instruction has been committed i.e. its result value is made part of the architectural state.

input request signal of its parent arbiter cell. Hence, at the root cell one or more of the input request signals will be high if there are one or more instructions that are ready. The root cell then grants the functional unit to one of its children by raising one of its grant outputs. This initiates the second phase. In this phase, the grant signal is propagated down the tree to the instruction that is selected. At each level, the grant signal is propagated down the subtree that contains the selected instruction.

The enable signal to the root cell is high whenever the functional unit is ready to execute an instruction. For example, for single-cycle ALUs, the enable signal will be permanently tied to high.

The selection policy implemented by our assumed structure is static and is strictly based on location of the instruction in the issue window. The leftmost entries in the window have the highest priority. The *oldest ready first* policy can be implemented using our scheme by compacting the issue window to the left every time instructions are issued and by inserting new instructions at the right end. This ensures that instructions that occur earlier in program order occupy the leftmost entries in the window and hence have higher priority than later instructions. However, it is possible that the complexity resulting from compaction could degrade performance. We did not analyze the complexity of compacting in this study.

5.5 Handling Multiple Functional Units

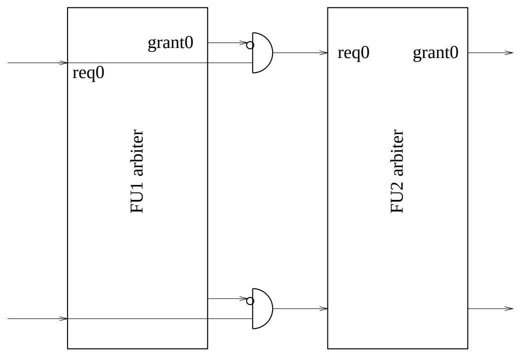


Figure 19: Handling multiple functional units

If there are multiple functional units of the same type, then the selection logic (shown in Figure 19) comprises a number of blocks of the type studied in the previous section, stacked in series. The request signals to each block are derived from the requests to the previous block by masking the request that was granted the previous resource.

An alternative to the above scheme is to extend the arbiter cells so that the request and grant signals encode the number of resources being requested and granted respectively. However, we believe that this could considerably slow down the arbiter cells and hence would perform worse than the stacked design.

The stacked design might not be a feasible alternative beyond two functional units because the resulting delay can be significant. An alternative option is to statically partition the window entries among the functional units. For example, in the MIPS R10000 [34], the window is partitioned into three sets called the integer queue, floating-point queue, and the address queue. Only instructions in the integer queue are monitored for execution on the two integer functional units. Similarly, in the HP PA-8000 [19], the window is partitioned into the ALU queue and the MEM queue. The ALU queue buffers integer and floating-point instructions. Only instructions in the ALU queue are monitored for execution on the two integer functional units and two floating-point functional units. The MEM queue buffers load/store instructions. The instructions in the MEM queue are monitored for execution on the load/store units.

5.5.1 Delay Analysis

The delay of the selection logic is the time it takes to generate the grant signal after the request signal has been raised. This is equal to the sum of two terms: the time taken for the request signal to propagate to the root of the tree and the time taken for the grant signal to propagate from the root to the selected instruction. Symbolically,

$$Delay = (L-1) \times T_{reqpropd} + T_{root} + (L-1) \times T_{grantpropd}$$

where $L = \log_4(WINSIZE)$ is the height of the selection tree, $T_{reqpropd}$ is the time taken for the request signal to propagate through an arbiter cell, T_{root} is the delay of the $grant$ output at the root cell, and $T_{grantpropd}$ is the time taken for the grant signal to propagate through an arbiter cell. Hence, the overall selection delay can be written as

$$Delay = c_0 + c_1 \times log_4(WINSIZE)$$

where c_0 and c_1 are constants.

From the above equations we can see that the delay of the selection logic is proportional to the height of the tree and the delay of the arbiter cells. The delay has a logarithmic relationship with the window size. Increasing issue width can also increase the selection delay if a stacked scheme, as described in the previous section, is used to handle multiple functional units. For the rest of the discussion, we will assume that a single functional unit is being scheduled and hence no stacking is used. The delay for a stacked design can be easily computed by multiplying our delay results by the stacking depth. One way to improve the delay of the selection logic is to increase the radix of the selection tree. However, as we will see shortly, this increases the delay of a single arbiter cell and could make the overall delay worse.

Arbiter logic

The circuit for generating the *anyreq* signal is shown in Figure 20. The *anyreq* signal is raised if one or more of the input request signals is active. The circuit, implementing the OR function, consists of a dynamic NOR gate followed by an inverter. The dynamic gate was chosen instead of a static OR gate for speed reasons. The circuit operates as follows. The *anyreqb* node is precharged high. When one or more of the input request signals go high, the corresponding pull-downs pull the *anyreqb* node low. The inverter in turn raises the *anyreq* signal high. The value of $T_{reqpropd}$ in the delay equation is the delay of the OR circuit.

The priority encoder in the arbiter cell is responsible for generating the grant signals. The logic equations for the grant signals are:

$$grant0 = req0 \cap enable$$

$$\begin{aligned} grant1 &= \overline{req0} \cap req1 \cap enable \\ grant2 &= \overline{req0} \cap \overline{req1} \cap req2 \cap enable \\ grant3 &= \overline{req0} \cap \overline{req1} \cap \overline{req2} \cap req3 \cap enable \end{aligned}$$

For example, $grant2$ is high only if the cell is enabled, the input requests $req0$ and $req1$ are low, and $req2$ is high. Because the request signals at each cell, except for the root cell, are available well in advance of the enable signal we use a two-level implementation for evaluating the grant signals. As an example, the circuit for evaluating $grant1$ is shown in Figure 20. The first stage evaluates the $grant1$ signal (node $grant1p$) assuming the $enable$ signal is high. In the second stage, the $grant1p$ signal is ANDed with the $enable$ to produce the $grant1$ signal. This two-level decomposition was chosen because it removes the logic for $grant1p$ from the critical path. This optimization does not apply at the root cell because at the root cell the request signals arrive after the enable signal.

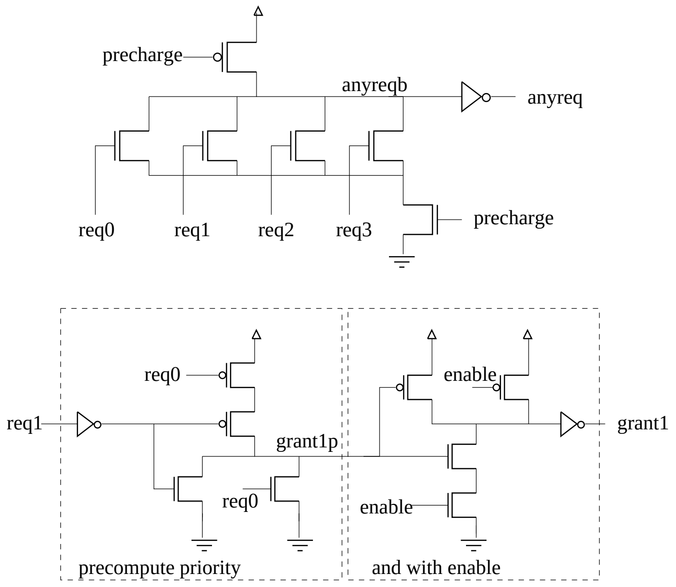


Figure 20: Arbiter logic

The policy used by the selection logic is embedded in the above equations for the grant outputs of the arbiter cell. For example, the design presented assumes static priority with $req0$ having the highest priority. Implementing an alternative policy would require appropriate modifications to these equations. Again, the designer has to be careful while selecting a policy because using a complex policy can increase the delay of the selection logic by slowing down individual arbiter cells.

Increasing the number of inputs to the arbiter cell slows down both the OR logic and the priority encoder logic. The OR logic slows down because the load capacitance contributed by the diffusion capacitance of the pull-downs increases linearly with the number of inputs. The priority logic slows down because the delay of the logic used to compute priority increases due to the higher fan-in. We found the optimal number of inputs to be four in our case. The selection logic in the MIPS R10000, described in [30], is also based on four-input arbiter cells.

5.5.2 Spice Results

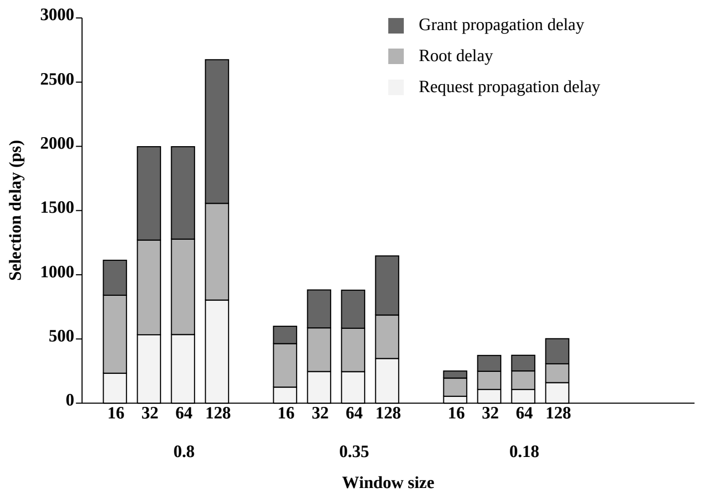


Figure 21: Selection delay versus window size

Figure 21 shows the delay of the selection logic for various window sizes in the three technologies assuming a single functional unit is being scheduled. The delay is broken down into the three components discussed earlier. From the graph we can see that for all the three technologies, the delay increases logarithmically with window size. Also, the increase in delay is less than 100% when the window size is increased from 16 instructions to 32 instructions (or from 64 instructions to 128 instructions) because the middle term in the delay equation, the delay at the root cell, is independent of window size. Detailed results are presented in tabular form in Appendix B.

The various components of the total delay scale well as the feature size is reduced. This is not surprising because all the delays are logic delays. It must be pointed out that the selection delays presented here are optimistic because we do not consider the wires in the circuit, especially if it is the case that the request signals [6] originate from the CAM entries in which the instructions reside. On the other hand, it might be possible to minimize the effect of these wire delays if the ready signals are stored in a smaller, more compact array.

5.6 Data Bypass Logic

The data bypass logic is responsible for bypassing result values to subsequent instructions from instructions that have completed execution but have not yet written their results to the register file. The hardware datapaths and control added for this purpose form the bypass logic. The number of bypasses required is determined by the depth of the pipeline and the issue width of the microarchitecture. As pointed out in [1], if IW is the issue width, and if there are S pipestages after the first result-producing stage, then a fully bypassed design would require $(2 \times IW^2 \times S)$ bypass paths assuming 2-input functional units. In other words, the number

[6]The ready flags discussed in the wakeup logic presented in Section 5.2.

of bypass paths grows quadratically with issue width. The current trend towards deeper pipelines and wider degree of issue only multiplies the number of bypass paths and makes the bypass logic even more critical.

The bypass logic consists of two components: the datapath and the control. The datapath comprises busses, called the result busses, that are used to broadcast bypass values from each source to all possible destinations. The sources of bypass values are the functional units and the cache ports. Buffers are used to drive the bypass values on the result busses. In addition to the result busses, the datapath comprises operand MUXes. Operand MUXes are required to gate in the appropriate result on to the operand busses. The fan-in of the operand MUXes is one greater than the number of result busses. The extra input to the MUX is for the case of reading the operand from the register file.

The control logic is responsible for controlling the operand MUXes. The control logic compares the tag of the result value to the tag of the source value that is required at each functional unit. If there is a match, the MUX control is set so that the result value is driven on the appropriate operand bus.

The key factor that determines the speed of the bypass logic is the delay of the result wires that are used to transmit bypassed values. The control adds to this delay; however, for our analysis, we will ignore the control because its delay is a small fraction of the total delay. Also, as we move towards smaller feature sizes, wire delays resulting from the result wires will be responsible for a significant fraction of the total delay.

5.6.1 Structure

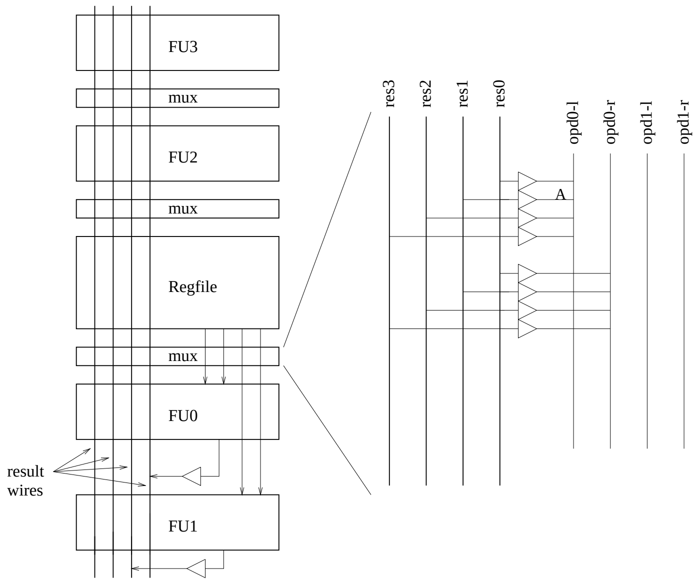


Figure 22: Bypass Logic

A commonly used structure for the bypass logic is shown in Figure 22. The figure shows a bit-slice of the datapath. There are four functional units marked FU0 to FU3. Consider the bit slice of FU0. It gets its two operand bits from the *opd0-l* and *opd0-r* operand wires. The result bit is driven on the *res0* result wire by the result driver. Tristate buffers are used to drive the result bits on to the operand wires from the result wires. These buffers implement the MUXes shown in the figure. For example, in order to bypass the result of functional unit FU1 to the left input of functional unit FU0, the tristate driver marked A is switched on. The driver A connects the *res1* wire and the *opd0-l* wire. In the case where bypasses are not activated, the operand bits are placed on the operand wires by the register file read ports [7]. The result bits are written to the register file in addition to being bypassed.

The delay of the bypass logic is largely determined by the time it takes for the driver at the output of each functional unit to drive the result value on the corresponding result wires. This in turn depends on the length of the result wires. From the figure it can be seen that the length of the wires is a function of the layout. For the layout presented in the figure, the length of the result wires is determined by the height of the functional units and the register file. Alternative layouts are discussed in the results section.

5.6.2 Delay Analysis

As discussed before, the delay of the bypass logic can be approximated by the wire delay of the result wires. Considering the wires as distributed RC lines, the delay is given by

$$\begin{aligned} Delay &= 0.5 \times R_{metal} \times L \times C_{metal} \times L \\ &= 0.5 \times L^2 \times R_{metal} \times C_{metal} \end{aligned}$$

where L is the length of the result wires.

From the above equation we can see that as the issue width is increased, the length of the result wires increases and this causes the bypass delay to grow quadratically with issue width. Increasing the depth of the pipeline also increases the delay of the bypass logic as follows. Increasing the depth increases the fan-in of the operand MUXes connected to a given result wire. This in turn increases the amount of capacitance to be charged or discharged on each result wire because the diffusion capacitance of the tristate buffers of the operand MUXes adds to the capacitance on the wires. However, this component of the delay is not captured by our simple model. We expect this component of the delay to become relatively less significant as the feature size is reduced.

5.6.3 Results

Table 2 shows typical heights of functional units. The lengths were estimated based on data presented in papers [14, 28, 15] describing specific implementations of functional units. For example, the height of a complete ALU is 3200 λ, where λ is half the feature size. The complete ALU, referred to as ALU_{gen}, comprises an adder, a shifter, and a logic unit. Similarly, the height of a simple ALU is 1700 λ. The simple ALU, referred to as ALU_{simple}, contains an adder, a logic unit but no shifter. A load/store unit, referred to as a LDST unit, consists of an adder for computing effective addresses.

From the table the total length of the result wires for a 4-way machine with 1 ALU_{gen}, 1 ALU_{simple}, and two LDST units can be calculated as follows.

$$Length = regfile\ height + ALU_{gen}\ height + ALU_{simple}\ height + 2 \times LDST\ height$$

[7]In a reservation-station based microarchitecture like the Intel Pentium Pro [13], the operand bits come from the data field of the reservation station entry.

Functional unit	Height (λ)	Description
Adder	1400	64-bit adder
Shifter	1500	64-bit barrel shifter
Logic Unit	300	Performs logical operations
Complete ALU (ALU_{gen})	3200	Comprises adder, shifter, and logic unit
Simple ALU (ALU_{simple})	1700	Comprises adder and logic unit
Load/Store (LDST) Unit	1400	Comprises adder for effective address calculation

Table 2: Functional unit heights

Issue width	Functional unit mix	$\sum_{i=1}^{W} FUi\ height$ (λ)	Register file height (λ)	Wire length (λ)	Delay (ps)
4	1 ALU_{gen}, 1 ALU_{simple}, 2 LDST	7700	12800	20500	184.9
8	2 ALU_{gen}, 2 ALU_{simple}, 4 LDST	15400	33600	49000	1056.4

Table 3: Result wire delays for a 4-way and a 8-way processor

$$\begin{aligned} &= 12800\ \lambda + 3200\ \lambda + 1700\ \lambda + 2 \times 1400\ \lambda \\ &= 20500\ \lambda \end{aligned}$$

The above calculations do not take into consideration the height of an integer multiplier or divider because these units are usually placed at either end and hence the result wires do not have to extend over them. Also, we have ignored the height of the rows of MUXes (drivers) between the functional units. The register file height is computed using the formula

$$Height = NPREG \times (cellheight + wordline_{spacing} \times (3 \times IW))$$

where $(3 \times IW)$ is the total number of ports for a IW-wide machine assuming 2 single-ended read ports and 1 single-ended write port are required for each instruction, $NPREG$ is the number of physical registers, $cellheight$ is the height of an individual RAM cell excluding the wordlines, and $wordline_{spacing}$ is the extension in height of each cell for each wordline added [8].

Using similar calculations, we computed the wire delays for hypothetical 4-way and 8-way machines. Note that the wire delays remain constant across the three technologies according to the scaling model assumed in this study. Table 3 shows the results.

There are two observations that can be made from the table. First, while the wire delay for the 4-way machine seems to be within a clock period for current technologies (feature sizes) this might not be true for future technologies. Second, the wire delay for the 8-way machine is significant and can potentially degrade

[8]We use $cellheight = 40\lambda$, $NPREG = 80$ for 4-way and $NPREG = 120$ for 8-way, and $wordline_{spacing} = 10\lambda$ in our calculations.

the clock speed even for current technologies.

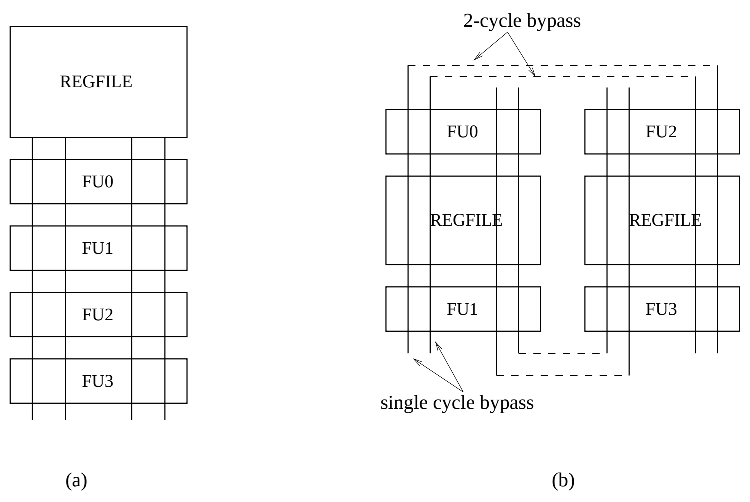


Figure 23: Alternative layouts for bypassing

Alternative Layouts

The results presented in the previous section assume a particular layout; the functional units are placed on either side of the register file. However, as mentioned before, the length of the result wires is a function of the layout. Hence, microarchitects will have to study alternative layouts in order to reduce bypass delays. Figure 23 shows some alternative layouts.

In alternative (a), all the functional units are placed on one side of the register file. In this case the result wires do not have to extend over the register file. However, the length of the operand wires originating from the register file increase relative to Figure 22 thus stretching the register file access time. Also, this organization has the disadvantage that the sense amplifiers of the register file cannot be distributed on both sides. This might lead to an increase in the width of the register file and hence, can also increase the register file access time.

In the long term, microarchitects will have to consider *clustered* organizations like the one shown in alternative (b). Each cluster has its own copy of the register file. Bypasses within a cluster complete in a single cycle while inter-cluster bypasses take 2 or more cycles. Such a scheme is implemented in the DEC 21264 [18]. The hardware or the compiler or both will have to ensure that inter-cluster bypasses occur infrequently. In addition to mitigating the delay of the bypass logic, this organization also has the advantage of faster register files because there are fewer ports on each register file. Another technique [1] that can be used to improve bypass performance is to use an incomplete bypass network. In an incomplete bypass network only the frequently used bypass paths are provided while interlocks are used in the remaining situations. For an 8-way machine with deep pipelines, this would exclude a large number of bypass paths.

Issue width	Window size	Rename delay (ps)	Wakeup+Selection delay (ps)	Bypass delay (ps)
4	32	351.0	578.0	184.9
8	64	427.9	724.0	1056.4

Table 4: Overall delay results for $0.18\mu m$ technology

6 Overall Delay Results and Pipelining Issues

The overall results for a 4-way and a 8-way microarchitecture in $0.18\mu m$ technology are shown in Table 4. The results for the $0.8\mu m$ and $0.35\mu m$ technology are shown in Appendix B. For the 4-way machine, the window logic (wakeup + select) has the greatest delay among all the structures considered and hence determines the critical path delay. The register rename delay comes next; it is about 39% faster than the delay of the window logic. The bypass delay is relatively small in this case. The results are similar for the 8-way machine except for bypass logic, which has a much higher delay. One way of dealing with this problem is to use clustering of register files and functional units. For example, the 8-way machine could be implemented as two 4-way clusters with single-cycle bypasses within each cluster and multi-cycle (2 or 3 cycle) bypasses across the two clusters. Therefore, in both cases the delay of the window logic is the largest and hence, the window logic is the most crucial structure among the list of structures we studied.

Until now, the delay of each of the critical structures was analyzed in detail. However, in addition to the delay, another important consideration is the pipelineability of the structures. Even if the delay of a structure is relatively large it can be eliminated from the critical path if it can be pipelined i.e. its operation is spread over multiple pipestages. For example, almost all designs spread floating-point operations over multiple cycles.

However, while pipelining can improve performance by facilitating a faster clock, it can result in a number of side-effects that can degrade performance too. First, the extra stages introduced by deeper pipelining in the front end increase the penalty of mispredicted branches. Also, the penalty of instruction cache misses will increase as a result of extra pipestages that have to be re-filled. At the same time accurate branch prediction can alleviate these problems to a certain extent. Hence, if the performance improvement achieved as a result of deeper pipelining (faster clock) surpasses the performance degradation caused by the extra stages, then pipelining might be an attractive option. The current trend in the microprocessor industry is towards deeper pipelining. For example, the pipeline in the Intel Pentium Pro [13] has as many as 14 pipestages.

The general subject of the effect of pipelining depth on overall performance has been the focus of a number of studies [20, 17, 10]. We took a different approach in our study. We studied the feasibility of pipelining each of the critical structures and identified *atomic* operations, if any, implemented by the structures. We define an operation to be *atomic* if the operation has to be completed within a single cycle in order to execute dependent instructions in consecutive cycles. An obvious example of an atomic operation is a simple ALU operation (integer add, logical and, etc.). If the ALU operation is spread over multiple pipestages then dependent instructions cannot execute in consecutive cycles and the resulting pipeline bubbles can result in serious performance degradation especially in programs with limited parallelism. Because atomic operations will not be pipelined for performance reasons, we believe that the latency of these atomic operations will ultimately limit the degree of pipelining. Consequently, the delays of these atomic operations are crucial and will determine the complexity of a microarchitecture.

- *Register renaming*

  The register rename logic can be pipelined by spreading the dependence checking and the map table access over multiple pipestages. While it is easy to see how dependence checking can be pipelined, it is not so obvious how the map table access can be pipelined. However, there are schemes [6, 23] for pipelining RAMs that can be employed to pipeline the map table access. In addition, in order to ensure that each rename group sees the map table updates performed by previous rename groups, the updates have to be bypassed around the map table i.e. the updates should be visible before the writes to the table actually complete. Hence, we think that even though the design will be complicated, register renaming can be pipelined.

  It must be pointed out that before attempting to pipeline renaming, there are a number of tricks that can be used to reduce its latency. First, the map table can be duplicated to reduce the number of ports on each copy of the table. Second, because not all instructions have two operands and because it is likely that instructions in a rename group have common operands, the port requirements on the map table can be reduced with little effect on performance.

- *Wakeup and selection*

  Wakeup and select together constitute an atomic operation. If they are spread across multiple pipestages, dependent instructions cannot execute in consecutive cycles as shown in Figure 24. The `add` and the `sub` instructions cannot be executed back-to-back because the result of select stage has to feed the wakeup stage. The resulting pipeline bubbles can seriously degrade performance especially in programs with limited parallelism. Hence, wakeup and select together constitute an atomic operation and must be accomplished in a single cycle.

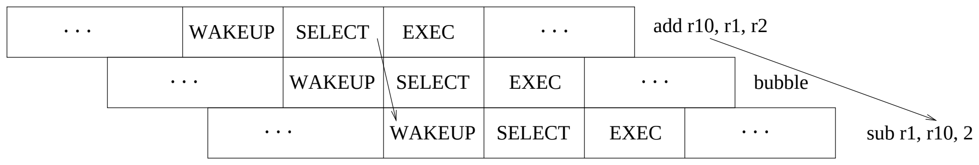


Figure 24: Pipelining wakeup and select

- *Data bypassing*

  Data bypassing is another example of an atomic operation. In order for dependent operations to execute in consecutive cycles, the bypass value must be made available to the dependent instruction within a cycle. Results presented earlier show this is feasible for a 4-way machine because the delay of the bypass logic in this case is relatively small. For wide issue machines (width greater than 4) some form of clustering will be required to make this feasible.

- *Register file access*

  Again, the techniques used to pipeline RAM can be employed to pipeline the register file. Tullsen et. al. [29] studied the effect of spreading register read over two pipestages. They found that single thread performance degraded by only 2% for their design.

  Once again, it must be mentioned that instead of pipelining the register file, architects can reduce its latency by duplicating the register file. Each copy of the register file will have half the number of read ports as the original register file. This technique has been used in the DEC 21264 [18]. In this case two copies of the integer register file are used.

- *Cache access*

  Cache access can be pipelined in a number of ways. One scheme, implemented in the DEC 21064 [9], reads the tags and the data in the first cycle and performs the hit/miss detection operation in the second cycle. A second, more aggressive scheme could pipeline both the tag RAM and the data RAM themselves. While pipelining the cache can facilitate a faster clock, it can degrade performance by increasing the load-use latency. However, we believe that current out-of-order microarchitectures can tolerate the extra cycle of latency and hence minimize the increase in cycle count.

  A related trade-off is to size the L1 data and instruction caches so that they can be accessed in a single cycle and use a bigger L2 cache to service the L1 misses.

The above discussion shows that as microarchitects employ deeper pipelines to enable ultra-fast clocks, it is likely that the window logic (wakeup and selection) is going to become the most critical structure.

7 Conclusions

In this report we analyzed the delay of some critical structures in superscalar processors. These structures are critical in the sense that their delay is a function of issue width, issue window size and wire delays and hence, it is likely that the delay of these structures will determine the cycle time in future designs in advanced technologies. We studied how the delays varied with issue width and window size. We also studied how the delays scale as feature sizes shrink and wire delays become more prominent.

Our results show that the logic associated with managing the issue window of a superscalar processor is likely to become the most critical structure as we move towards wider-issue, larger windows, and advanced technologies in which wire delays dominate. One of the functions implemented by the window logic is the broadcast of result tags to all the waiting instructions in the window. The delay of this operation is determined by the delay of wires that span the issue window. We found that the delay of this operation increases quadratically with window size and issue width. Hence, this operation does not scale well as we move towards larger windows, wider issue widths, and advanced technologies in which wire delays dominate. Furthermore, in order to be able to execute dependent instructions in consecutive cycles (back-to-back) the delay of the window logic (wakeup delay + selection delay) should be less than a cycle.

In addition to the window logic, a second structure that needs careful consideration especially in future technologies is the data bypass logic. The length of the result wires used to broadcast bypass values increases linearly with issue width and hence, the delay of the data bypass logic increases quadratically with issue width. We believe that these wire delays will force architects to consider *clustered* microarchitectures like the one employed by the DEC 21264 [18].

Acknowledgements

This work was supported in part by an internship at DEC Western Research Laboratory, and by grants from the NSF Grant MIP-9505853, and the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

We thank the staff at DEC-WRL for providing us with the CAD tools used in this study. Thanks also to Andy Glew for answering specific questions regarding the implementation of the Intel Pentium Pro and for related discussions.

References

[1] P. S. Ahuja, D. W. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 36–45, November 1995.

[2] C. Asato, R. Montoye, J. Gmuender, E. W. Simmons, A. Ike, and J. Zasio. A 14-port 3.8ns 116-word 64b Read-Renaming Register File. In *1995 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 104–105, February 1995.

[3] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors, 1994.

[4] Mark T. Bohr. Interconnect Scaling - The Real Limiter to High Performance ULSI. In *1995 International Electron Devices Meeting Technical Digest*, pages 241–244, 1995.

[5] M. Butler and Y. N. Patt. An Investigation of the Performance of Various Dynamic Scheduling Techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 1–9, December 1992.

[6] T. Chappell. A 2ns cycle, 4 ns access 512kb CMOS ECL SRAM. In *1991 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 50–51, February 1991.

[7] Marvin Denman. Design of the PowerPC 604e RISC Microprocessor, December 1995. Tutorial Talk at 28th Annual International Symposium on Microarchitecture.

[8] R. Dennard et al. Design of Ion-implanted MOSFETS with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, SC-9:256–268, 1974.

[9] D. Dobberpuhl et al. A 200-MHz 64-b dual-issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits*, 27:1555–1557, 1992.

[10] P. K. Dubey and M. J. Flynn. Optimal Pipelining. *Journal of Parallel and Distributed Computing*, 8:10–19, 1990.

[11] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, pages 40–51, February 1996.

[12] Linley Gwennap. HAL Reveals Multichip SPARC Processor. *Microprocessor Report*, 9(3), March 1995.

[13] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2), February 1995.

[14] I. S. Hwang and A. L. Fisher. A 3.1ns 32b CMOS Adder in Multiple Output Domino Logic. In *1988 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 140–141, February 1988.

[15] Inoue et al. A 0.4um 1.4ns 32b Dynamic Adder using Non-Precharge Multiplexers and Reduced Precharge Voltage Techniques. In *1995 Symposium on VLSI Circuits Digest of Technical Papers*, pages 9–10, June 1995.

[16] Mark G. Johnson and Norman P. Jouppi. Transistor Model for a Synthetic 0.8um CMOS Process, May 1990. Class notes for Stanford University EE371.

[17] Norman P. Jouppi and David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[18] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, October 1996. Presentation at the 9th Annual Microprocessor Forum, San Jose, California.

[19] Ashok Kumar. The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor. In *Proceedings of the Hot Chips VIII*, pages 9–20, August 1996.

[20] S. R. Kunkel and J. E. Smith. Optimal Pipelining in Supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986.

[21] Grant McFarland and Michael Flynn. Limits of Scaling MOSFETS. Technical Report CSL-TR-95-662 (Revised), Stanford University, November 1995.

[22] Meta-Software Inc. *HSpice User's Manual*, June 1987.

[23] Robert J. Proebsting. Speed Enhancement Technique for CMOS Circuits, January 1991. United States Patent No. 4,985,643.

[24] Jan M. Rabaey. *Digital Integrated Circuits - A Design Perspective*. Prentice Hall Electronics and VLSI Series, 1996.

[25] K. Rahmat, O. S. Nakagawa, S-Y. Oh, and J. Moll. A Scaling Scheme for Interconnect in Deep-Submicron Processes. Technical Report HPL-95-77, Hewlett-Packard Laboratories, July 1995.

[26] Eric Rotenberg, Steve Bennet, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proccedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[27] J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37:562–573, May 1988.

[28] M. Suzuki et al. A 1.4ns 32b CMOS ALU in Double Pass-Transistor Logic. *IEEE Journal of Solid-State Circuits*, 28(11), November 1993.

[29] Dean M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[30] N. Vasseghi et al. 200 MHz Superscalar RISC Processor Circuit Design Issues. In *1996 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 356–357, February 1996.

[31] Tomohisa Wada, Suresh Rajan, and Steven A. Przybylski. An Analytical Access Time Model for On-Chip Cache Memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.

[32] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, Second Edition, 1993.

[33] Steven J. E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, DEC Western Research Laboratory, July 1994.

[34] K. C. Yeager. MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, April 1996.

A Technology Parameters

The Hspice Level 3 models used to simulate the synthetic $0.8\mu m$, $0.35\mu m$, and $0.18\mu m$ CMOS technologies are given in Table 5. Table 6 gives the metal resistance and capacitance values assumed for the three technologies.

Parameter	$0.8\mu m$	$0.35\mu m$	$0.18\mu m$
tox	165	70	35
vto	0.77(-0.87)	0.67(-0.77)	0.55(-0.55)
uo	570(145)	535(122)	450(80)
gamma	0.8(0.73)	0.53(0.42)	0.40(0.32)
vmax	2.7e5(0.0)	1.8e5(0.0)	1.05e5(0.0)
theta	0.404(0.233)	0.404(0.233)	0.404(0.233)
eta	0.04(0.028)	0.024(0.018)	0.008(0.008)
kappa	1.2(0.04)	1.2(0.04)	1.2(0.04)
phi	0.90	0.90	0.90
nsub	8.8e16(9.0e16)	1.38e17(1.38e17)	4.07e17(4.07e17)
nfs	4e11	4e11	4e11
xj	0.2μ	0.2μ	0.2μ
cj	2e-4(5e-4)	5.4e-4(9.3e-4)	10.6e-4(21.3e-4)
mj	0.389(0.420)	0.389(0.420)	0.389(0.420)
cjsw	4e-10	1.5e-10	3.0e-11
mjsw	0.26(0.31)	0.26(0.31)	0.26(0.31)
pb	0.80	0.80	0.80
cgso	2.1e-10(2.7e-10)	1.8e-10(2.4e-10)	1.8e-10(2.4e-10)
cgdo	2.1e-10(2.7e-10)	1.8e-10(2.4e-10)	1.8e-10(2.4e-10)
delta	0.0	0.0	0.0
ld	0.0001μ	0.0001μ	0.0001μ
rsh	0.5	0.5	0.5
Vdd	5.0	2.5	2.0

Table 5: Spice parameters (Hspice Level 3 model)

Technology	R_{metal} (Ω/μm)	C_{metal} (fF/μm)
$0.8\mu m$	0.02	0.275
$0.35\mu m$	0.046	0.628
$0.18\mu m$	0.09	1.22

Table 6: Metal resistance and capacitance

B Delay Results

Issue width	Decoder delay (ps)	Wordline drive delay (ps)	Bitline delay (ps)	Sense Amp delay(ps)	Total delay (ps)
$0.8\mu m$ technology					
2	540.3	218.9	188.5	309.7	1502.2
4	547.1	227.9	212.1	317.5	1566.9
8	562.5	245.8	259.1	335.1	1700.9
$0.35\mu m$ technology					
2	220.2	95.6	98.6	137.9	649.4
4	225.8	103.9	116.2	143.0	698.5
8	243.1	115.8	151.7	151.4	800.8
$0.18\mu m$ technology					
2	129.6	70.6	72.9	102.8	435.4
4	136.8	78.2	87.6	105.8	478.9
8	148.4	92.5	117.8	110.7	561.7

Table 7: Breakup of rename delay

Window Size	Tag Drive Delay (ps)	Tag Match Delay(ps)	Match OR Delay(ps)	Total Delay(ps)
Issue Width = 2				
8	73.0	331.3	248.1	652.4
16	82.6	333.1	248.5	664.2
24	92.6	337.3	248.8	678.7
32	103.7	344.0	249.1	696.9
40	114.9	347.7	248.9	711.5
48	126.3	352.4	248.7	727.5
56	137.4	358.7	249.2	745.4
64	149.1	364.6	248.7	762.4
Issue Width = 4				
8	74.5	368.2	407.0	849.7
16	86.4	372.4	406.8	865.6
24	98.8	377.6	403.9	880.3
32	112.3	384.8	409.2	906.2
40	126.2	392.3	408.7	927.2
48	140.6	400.1	404.2	944.9
56	156.3	409.0	404.1	969.4
64	172.4	416.9	403.3	992.7
Issue Width = 8				
8	77.5	400.2	665.3	1143.0
16	93.3	406.6	665.7	1165.5
24	111.4	415.2	664.8	1191.4
32	130.7	425.2	658.5	1214.4
40	151.5	437.7	660.2	1249.5
48	174.4	451.0	658.3	1283.8
56	199.3	465.0	664.6	1328.9
64	228.2	479.2	664.6	1372.0

Table 8: Breakup of wakeup delay for $0.8\mu m$ technology

Window Size	Tag Drive Delay(ps)	Tag Match Delay(ps)	Match OR Delay(ps)	Total Delay(ps)
Issue Width = 2				
8	28.5	126.1	101.3	255.8
16	33.4	128.7	101.5	263.7
24	38.3	129.1	101.2	268.6
32	43.7	133.2	97.3	274.1
40	49.7	136.3	101.2	287.3
48	53.1	138.8	97.4	289.3
56	58.9	142.7	101.1	302.8
64	64.4	145.0	98.9	308.3
Issue Width = 4				
8	29.7	147.1	155.8	332.6
16	36.0	151.2	158.3	345.4
24	42.7	155.0	159.1	356.8
32	50.5	157.7	158.4	366.7
40	56.3	163.2	159.0	378.5
48	63.2	168.1	159.6	390.9
56	72.0	171.9	157.0	400.9
64	80.9	179.0	159.1	419.0
Issue Width = 8				
8	32.2	173.4	257.6	463.2
16	41.6	177.5	257.8	476.9
24	51.1	183.7	257.8	492.5
32	61.9	190.6	257.7	510.1
40	74.7	199.1	257.7	531.5
48	88.8	208.9	257.6	555.3
56	102.9	216.4	258.4	577.7
64	121.8	224.8	258.4	605.0

Table 9: Breakup of wakeup delay for $0.35\mu m$ technology

Window Size	Tag Drive Delay(ps)	Tag Match Delay(ps)	Match OR Delay(ps)	Total Delay(ps)
Issue Width = 2				
8	14.6	67.9	60.7	143.1
16	18.8	68.7	60.6	148.1
24	22.4	69.8	60.6	152.7
32	26.1	71.8	60.6	158.6
40	29.9	73.6	60.3	163.8
48	33.7	75.7	59.9	169.3
56	36.6	77.3	61.0	174.8
64	41.4	79.4	59.7	180.5
Issue Width = 4				
8	15.8	84.1	84.7	184.7
16	21.1	85.1	84.4	190.6
24	26.1	87.6	84.8	198.5
32	31.2	90.8	84.3	206.3
40	36.6	93.3	84.8	214.7
48	41.7	96.5	84.4	222.5
56	47.5	99.4	84.8	231.8
64	54.1	102.8	84.4	241.3
Issue Width = 8				
8	18.8	104.9	123.6	247.3
16	26.1	108.4	123.8	258.3
24	33.8	113.6	123.1	270.5
32	42.0	118.2	125.0	285.1
40	51.5	124.8	123.2	299.5
48	62.6	130.4	123.0	316.0
56	75.1	135.2	123.2	333.4
64	90.0	139.4	122.9	352.3

Table 10: Breakup of wakeup delay for $0.18\mu m$ technology

Window size	$T_{reqpropd}$(ps)	T_{root}(ps)	$T_{grantpropd}$(ps)	Total delay(ps)
$0.8\mu m$ technology				
16	233.2	607.2	272.5	1113.0
32	532.5	737.6	727.4	1997.5
64	534.6	742.9	719.8	1997.4
128	802.8	753.4	1118.5	2674.6
$0.35\mu m$ technology				
16	125.0	338.5	135.4	598.9
32	246.6	339.7	295.4	881.7
64	245.5	338.0	296.3	879.8
128	347.9	338.5	460.3	1146.7
$0.18\mu m$ technology				
16	53.6	141.7	55.1	250.4
32	107.0	141.2	123.5	371.7
64	106.9	144.2	121.9	373.0
128	159.9	146.7	195.5	502.1

Table 11: Breakup of selection delay

Issue width	Window size	Rename delay (ps)	Wakeup+Selection delay (ps)	Bypass delay (ps)
4	32	1577.9	2903.7	184.9
8	64	1710.5	3369.4	1056.4

Table 12: Overall delay results for $0.8\mu m$ technology

Issue width	Window size	Rename delay (ps)	Wakeup+Selection delay (ps)	Bypass delay (ps)
4	32	627.2	1248.4	184.9
8	64	726.6	1484.8	1056.4

Table 13: Overall delay results for $0.35\mu m$ technology