Design and Implementation of a 100 MHz Centralized Instruction Window for a Superscalar Microprocessor

Steven Wallace, Nirav Dagli, and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717
{swallace, niravd, nader}@ece.uci.edu

Abstract

The maxim of the superscalar architecture is that higher performance can be achieved by executing multiple instructions simultaneously. This can be realized in hardware by using a centralized instruction window. We present the design and implementation of a centralized instruction window capable of out-of-order issue and completion of four instructions per cycle. A compact layout (6.4mm by 2.2mm) of a 32-entry instruction window resulted from a full-custom design in 1.0 μm (drawn) 3-layer metal CMOS technology. The layout was verified by simulation and shown to operate at a clock frequency over 100 MHz.

1 Introduction

A superscalar microprocessor can resolve the dependencies of several different instructions dynamically using a reorder buffer and a centralized instruction window[2]. The design to carry out dynamic scheduling is complex. One drawback is that information about many instructions must be stored in hardware simultaneously. The task of scheduling instructions onto functional units is another difficulty. A superscalar microprocessor called the Superscalar Digital Signal Processor (SDSP) that implements these advanced superscalar techniques has been implemented at UC Irvine. It uses a reorder buffer [4] and a centralized instruction window.

A centralized instruction window keeps track of many instructions at one time, and independent instructions are issued *out-of-order* to functional units. New instructions are allocated to entries in the instruction window, while old instructions get shifted out, essentially moving the window along the length of the code. The *out-of-order completion* is handled by initially storing the results in the reorder buffer and committing them to the register file *in-order*. The reorder buffer employs *register renaming* mechanism to resolve the storage conflicts. Extensive simulations were carried out to analyze the impact of changing the various parameters associated with the instruction window - fetch bandwidth; branch prediction; number of entries; number of ALUs, multipliers and load/store units; instruction and data cache; and maximum issue/write-back ports[3]. These results were taken into consideration while designing and implementing the instruction window. Additional fine tunings have been made regarding the branch prediction, issue mechanism and write-back policy, because of the hardware tradeoffs involved.

Many new processor architectures have recently utilized limited superscalar principles. Most of them implemented the reservation station approach and disregarded the centralized instruction window approach because of a complex issue mechanism and many operand buses, the exception being the Metaflow architecture [1]. This paper, by discussing the implementation of a centralized instruction window, also highlights the simplified issue logic and maximum utilization of the operand buses by sharing them between functional units. Our work demonstrates that these advanced functions can be designed and implemented with good space and timing constraints.

2 Design

The instruction window's function is to dynamically detect instruction level parallelism and issue as many instructions as possible depending upon the available resources and issue ports. Figure 1 gives a general idea of the relative placement and interface of the two major components of a superscalar processor: the reorder buffer (RB) and instruction window (IW).

All information pertaining to an instruction is stored in the instruction window (including its opcode, source operands, destination tag, ready bit, and issued bit). Thus, unlike the deferred scheduling method employed in the DRIS[1], where another read is required to actually read the operands after an issue, here the opcode and operands of an instruction are issued directly to a functional unit. The instruction window is made up of five major fields: src1 control, src1 data, instruction scheduler, src2 data, and src2 control.

2.1 Src1 and Src2 Control

The src1 and src2 control units encompass most of the logic involved in writing results arriving from the functional units into the source operand fields. It generates the control signals to be used by the instruction scheduler. Various control bits associated with each instruction are stored in this field and updated every cycle.

Five CAM (Content Addressable Memory) cells contain the *tag* field. When results from the func-

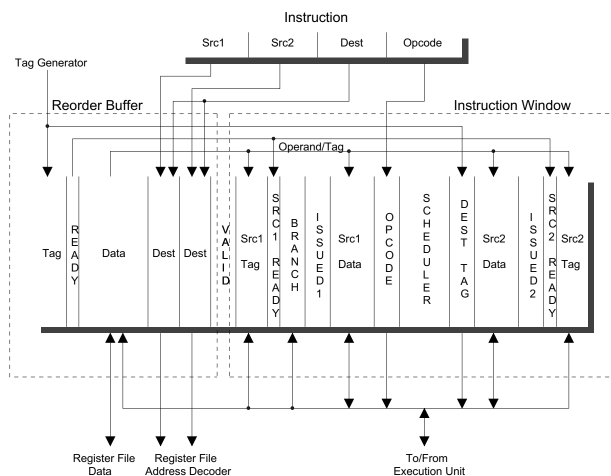

Figure 1: Block Diagram of the Reorder Buffer and Instruction Window

tional units return for writing, the tag associated with that result is used to search for all matching entries with the same tag identifier.

Another responsibility is to keep track of the *ready* bit information. Instructions are considered for scheduling only if both src1 and src2 *ready* bits are set. Since instruction scheduling must be done before results from currently executing instructions have arrived, the result tags are sent for comparison ahead of time. The result of that comparison is also used in the second phase of the next cycle when the results arrive. Thus, two sets of latches are required to store the matching results.

A unique feature of the instruction window is how conditional branches are controlled. It compares the predicted and the actual path *inside* the src1 control unit - as soon as the result of the branch condition is returned. An extra bit is used to contain a copy of the *prediction* bit that shifts in when the branch instruction is decoded. The LSB of the result of the branch condition is compared with the *prediction* bit through an XOR gate. The output from the XOR gate is used to decide whether the prediction was correct. If correctly predicted, the *src1 ready* signal is *not* allowed to pass through to the instruction scheduler (IS), i.e. the IS will assume the instruction is not ready and will not consider it for an issue. If the branch was mispredicted, then all instructions following that branch instruction are invalidated.

A simple technique is used to generate the 16 bypass control signals for full 4 by 4 bypassing. The four *match local* lines, indicating the port the instruction is issued on, are sensed to discharge any one of the 16 bypass control signals. These signals are then driven onto the bypass multiplexers located at the bottom of the *src1 data* field through the bypass drivers and the necessary bypassing takes place.

Since load bypassing is implemented, an instruction can be issued expecting that the result from a previous load instruction will get bypassed. In the event of a cache miss, the result will not be available. Some instructions could have been issued since the scheduler assumes results will be available. In this case of a *false issue*, the *issued* bit is reset, the *ready* bit is reset, and functional units are notified that the operand scheduled is invalid.

Although each row in the src1 control unit carries out identical logical function, the top four entries differ slightly from the rest of the group because a new block of four instructions shift into it. The cells use transparent latches instead of edge-triggered flip-flops and their timing lags behind the rest of the rows.

2.2 Instruction Scheduler

A block diagram of the instruction scheduler is shown in Figure 2. It forms forms the backbone of the instruction window and is the most complex part of the instruction window.

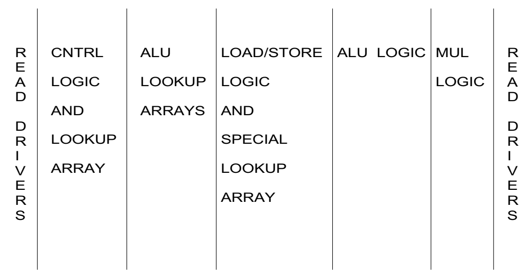

Figure 2: Instruction Scheduler

Five control bits indicating the instruction type are present in the IS, one for each instruction type (ALU, MUL, LOAD, STORE and CNTRL). These bits are then utilized along with the *src1 ready, src2 ready, valid, issued1* and *issued2* signals to control the charging/discharging of the inputs to the lookup arrays. A lookup array is special logic that is used to search for the oldest entry. At the end of the scheduling, the *read* lines are driven onto the *match local* lines of the src1 and src2 data fields through the read drivers.

2.2.1 Port Assignments

For maximum efficiency, each functional unit should have a dedicated port in the data cell of the instruction window. Unfortunately, this leads to an unreasonably large size of the cell. A 4-port data cell was chosen based on simulation results and space constraints. Due to the limited number of ports, a compromise had to be made by assigning specific FUs to specific ports:

- *port 1* : dedicated specifically to ALU1
- *port 2* : shared between ALU2 and the MULTIPLIER unit
- *port 3* : shared between ALU3 and the STORE unit
- *port 4a* : dedicated for CONTROL TRANSFER instructions

- *port 4b* : dedicated for LOAD instructions

For *port 4*, the structure of the instruction set was taken into account to achieve maximum utilization. Since the *load* instruction needs only *src2* while the *control transfer* instruction needs only *src1*, both instructions can be simultaneously issued on the same port.

2.2.2 Issue Algorithm

The "oldest first" algorithm is used in the issue of instructions from all categories. The FIFO nature of the IW simplifies the implementation of this algorithm. The decision to implement the IW using a FIFO was a critical decision. Since the physical order of the instructions correspond to the original instruction stream order, special lookup array logic can be used to implement the scheduling algorithm with a compact layout and high speed.

The default scheduling algorithm for an instruction is to schedule instructions out-of-order using the "oldest first" algorithm. Exceptions are listed below:

- **ALU** To schedule three possible ALU instructions, the scheduler first searches through the available ALU instructions to find the oldest ALU instruction. Then it makes another sweep of the remaining instructions to the find the second oldest ALU instruction. Finally, it performs a third sweep to find the third oldest ALU instruction. Although these "sweeps" are actually implemented by using lookup arrays which finish this task in discrete time, the serial nature of the algorithm accumulates to a significant amount of time. Even to handle these three ALUs, the scheduling had to be extended over to two phases. In order to be able to issue four ALUs, the cycle time must increase, or an expensive algorithm which performs scheduling in parallel be used.

- **LOAD/STORE** Store instructions are issued in-order to the store unit to maintain the proper state[2]. A load instruction is not issued if a previous store instruction is pending. Load instructions may be issued out-of-order within store instruction boundaries.

- **MULTIPLY** The scheduling of the multiply instructions is restricted based on use of a signal that indicates the resource is available to provide support for a multiple cycle latency multiplier.

2.3 Bypassing

The instruction scheduler has been designed such that bypassing is completely hidden to it, i.e. when it issues an instruction, it does not know whether the instruction already has its operand(s) or its operand(s) will be bypassed. Bypassing is carried out by sending the destination tag of the instruction being issued to the *src1 tag* and *src2 tag* fields of the instruction window. Matching is carried out immediately and results in the assertion of the respective match lines. Appropriate instructions that were waiting on a result are added to the pool of issuable instructions.

An important feature implemented in the IW is load bypassing. On a cache hit, a load result will be available in one clock cycle and can be bypassed the same as ALU results. On the other hand, in case of cache-miss, bypassing cannot be carried out. Since load instructions are often at the root of a chain of dependencies, a cache-hit for every load operation is initially assumed. The instructions depending on its result are then setup for scheduling based upon the expected result from load bypassing. If the instruction is issued and a cache-miss occurs, the IW resets the ready and issued bits that were falsely set and informs the relevant functional units of an invalid issue.

An arguable point is that by doing this speculative forwarding, the already limited issue ports are wasted as other ready instructions could have been issued in place of these dubious instructions. But this is going to happen infrequently if there is good cache performance. Without load bypassing, a load instruction would be a 2-cycle latency instruction. This extra latency has a significant negative impact on performance, so the overall performance improvement caused by load bypassing overshadows the occasional issue port loss.

2.4 Write Back Policy

If a two cycle latency *multiply* instruction is issued, then the next clock cycle the controller checks to see if there is an issue of an *alu* instruction on the third port. If there is no issue, the third port is free, and the multiplier may use it. Otherwise, the multiply result uses the forth port. If both a multiply result and load need to use the forth port, the result of the load is blocked for a cycle.

3 Basic Cells

3.1 Src1 and Src2 Control

3.1.1 CAM Cell

A four-port CAM (Content Addressable Memory) cell is used to determine which tag entries match for writing. Each match line is initially discharged, and the register or tag used for comparison is put on its corresponding *bit* and $\overline{bit}$ lines. To evaluate, each match line uses an active p-transistor that attempts to charge it. If all bits match, then all the n-transistors in the CAM cells will be open and the match line will charge. On the other hand, if one or more bits do not match, then the n-transistors will pull the match line close to ground since the transistor gain factor of the pull-down transistors is six times that of the pull-up transistor [5]. Using this method, only entries that match will charge up the match lines. Alternatively, if a precharge method is used, then additional precharge control transistors would be required for each cell, and more importantly, an additional synchronization step would be required between the destination CAM cells and the lookup cells. Therefore, we chose the former method at the cost of power consumption.

3.1.2 Latch

The latches used in conjunction with the *ready* bit consist of a simple 10 transistor logic as shown in Figure 3. Both the *latch* and the $\overline{latch}$ signals are needed. The output of the cell reflects the changing input when the *latch* is high.

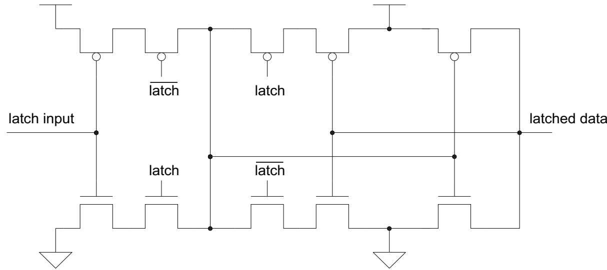

Figure 3: Transparent Latch Schematic

3.1.3 Driver Cells

Figure 4 shows the read and write drivers used in the design for driving the *match_local* signal. The *match_local* signal activates the corresponding ports on all the data cells in that specific row for read/write operation.

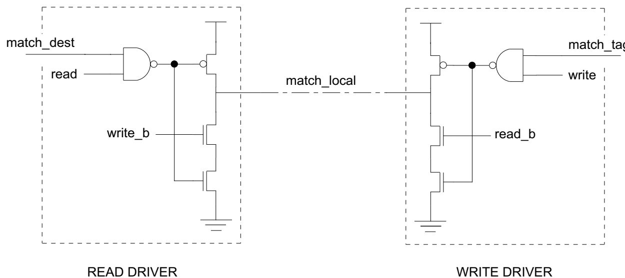

Figure 4: Driver Cells

The read and write drivers function as a single distributed multiplexer, driving either the *match_dest* signal (during read phase) or the *match_tag* signal (during write phase) onto the *match_local* signal. They keep the *match_local* discharged during other phases. The major advantage of using the drivers in this configuration is that only the n-transistors in the final stage are in series. In essence, the function of tristating the p-transistor in the final stage is transferred to the nand gate, avoiding the unnecessarily large size required for the two series p-transistors in order to maintain the same driving capability. This also ensures the relatively identical size of the n-diffusion region and the p-diffusion region, resulting in a very compact design for the large drivers.

3.2 Data Cell

Figure 5 is the schematic of the shift/storage cell used in the instruction window. Instead of the transmission gates used in the shift/storage cell in the RB, series transistors are used to provide better switching.

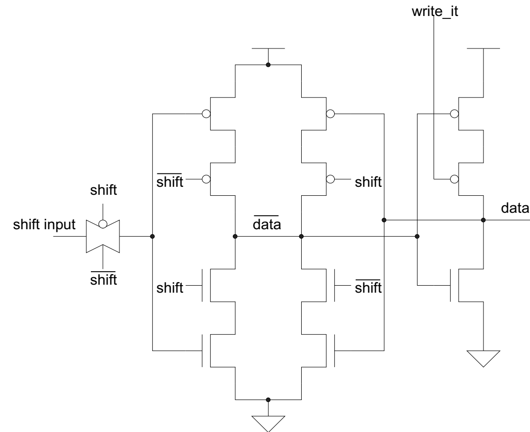

Figure 5: IW Shift/Storage Cell

Normally a data cell is written by placing the *data* and $\overline{data}$ on the *bit* and $\overline{bit}$ lines of the cell and enabling the n pass transistors. A "0" is written by discharging the stored value through the *bit* line while a "1" is written by discharging the stored value through the $\overline{bit}$ line.

The *src1* and *src2* fields of the IW are written to only once in their lifetime - from the time they are created for a new instruction, to the time they get shifted out and discarded from the IW. The design of this cell takes advantage of this fact. Each time a new entry is created, a "1" is shifted in to the data bits of the *src* field. Since the cell will have a value of "1" by default, only a "0" needs to be *written*. This means that the $\overline{bit}$ ports are no longer required.

To allow the worst case write of a "0" to four entries by the same result, the series *write_it* p-transistor is used to turn off the path from V_{dd} during the write cycle. The width of the IW is the critical dimension in the design, so two layers of n and p transistors are used to create a thin cell.

3.3 Instruction Scheduler

The design is comprised of 32 rows, each the same height as in the control and data portions of the instruction window. Because lookup array logic allows limited amount of routing space horizontally, all lookup arrays are grouped together at the center of the IS. They are similar to the ones used in the reorder buffer [4], but physically inverted to find the oldest instruction. The lookup arrays for load and store dependencies carry carry out the inverse function.

3.3.1 Lookup Arrays

If more than one instruction of the same type is ready to be issued, the oldest instruction should be scheduled. All the matching entries above the oldest instruction should be discharged. Therefore, a circuit has been designed that performs this task in constant

time with respect to the number of entries, n, and requires $\lg n$ space.

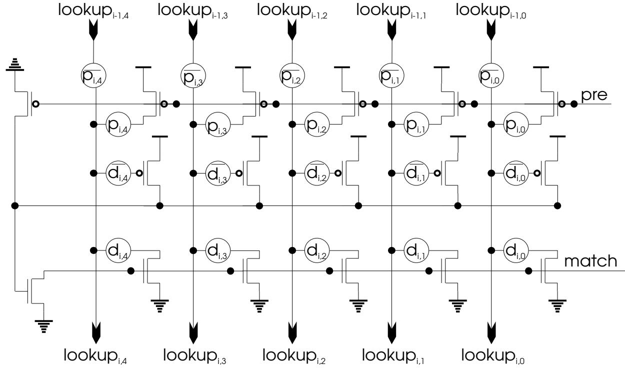

Figure 6: Lookup Cell in the ith Row

Let $m = \lg n$, $i = 0...n-1$, and $j = 0...m-1$. If $n = 32$, Figure 6 shows the schematic for the ith entry (starting from the top). In the schematic, there are circles with a variable inside them. If the variable evaluates to a value of 0, then the circle represents a short circuit. Otherwise, it is an open circuit, and the connecting transistor need not be present. These variables, p (precharge lookup) and d (discharge lookup), conform to the following equations:

$$p_{i,j} = i \bmod 2^{j+1}$$

and

$$d_{i,j} = \left\lfloor \frac{i \bmod 2^{j+1}}{j+1} \right\rfloor .$$

3.3.2 Storage Cells

Modified version of the data cell is used in the instruction scheduler to store the various instruction-type bits. The data cell for the ALU instruction-type cell has three read ports, while the other data cells have just one read port. The ports open only during the fourth phase, as the internal match lines may change their state during the scheduling phases.

3.3.3 Precharge Control Cells

Along with each data cell is a corresponding precharge control cell for charging and discharging of the inputs to lookup arrays. All are simple, except for the ALU precharge control cell which needs complex logic to control the three sweeps during scheduling.

4 Timing

The instruction window uses a 4-phase clock running at 100 MHz. All the control signals are generated off this clock.

The SDSP simulator developed at UCI aided in design verification [3]. Various benchmarks were run on the simulator, and the inputs to the instruction window were converted to IRSIM stimulus. For the first 1,000 cycles of each program, the layout was simulated, and the resulting outputs were successfully compared against the expected outputs from the SDSP simulator.

4.1 Src1 and Src2 Control

The control bits of the decoded instructions are shifted into the control fields during the first phase. The *src*1 and *src2 tags* for the operands and the *ready* bits are read from the reorder buffer and latched in during the third phase. The match against these tags is carried out in the same phase (the CAM logic has been designed to allow the tag bits to change during the match and still produce the correct result). The resulting match information is stored in transparent latches for future use. In the fourth phase, the match information and the *ready* bits are used to negate the issue of the instruction in case of a false issue. Table 1 lists the various events occurring in the instruction window during the four phases.

Phase	Top 4 rows	Others
1	Shift in control bits.	Source tag match. Resulting *mi* latched. Shift in control bits.
2		Write *ready* bit. Write data.
3	Source tag match. Resulting *mi* latched. Write *ready* bit. Shift in data.	Latch *mi* into *mf*.
4	Read data. Set/reset *issued* bit.	Read data Set/reset *issued* bit.

Table 1: Instruction Window Phase Table

After the instructions are inside the instruction window, the *src1 tags* and *src2 tags* shift at the rising edge of phase 1, and the tags are compared. The evaluated match lines are latched at the end of the first phase into transparent latches. They are then transferred to the second set of transparent latches during the third phase. Their state is used in the second phase of the next cycle to assert the *match_local* lines for the write operation. The *ready* bit is set in the second phase. In case of an invalid issue, the *ready* and *issued* bits are reset in the fourth phase. Bypassing of results is carried out in the fourth phase by comparing the match lines of the issued instruction against the write lines.

4.2 Src1 and Src2 Data

Figure 7 is a timing diagram of the read and write operations for operands. The delay information with each transition represents the worst case scenario. The worst case is from the src1 data field because it also contains the *opcode* field and therefore has more capacitance.

In the read phase, the match lines are asserted by the read drivers from the instruction scheduler and are used by the bypassing logic to generate the bypassing control signals. As Figure 8 illustrates, this becomes the critical path of the instruction window. The entire

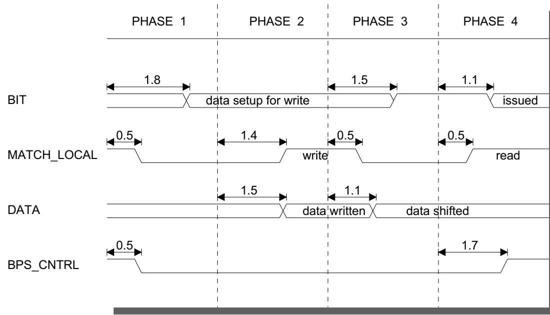

Figure 7: Instruction Window Timing Diagram (nsec)

delay for the operands to reach the functional units will depend upon routing and placement.

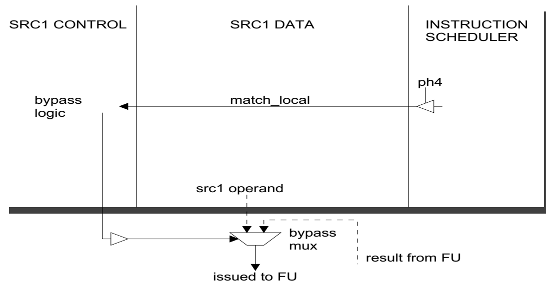

Figure 8: Critical Path for the Instruction Window

4.3 Instruction Scheduler

Phase	Instruction Scheduler
1	Shift the control bits.
2	Start scheduling of ALU and MUL instructions.
3	Start scheduling of the rest of the instructions.
4	Issue instructions.

Table 2: Instruction Scheduler Timing

Table 2 lists the major events occurring in the instruction scheduler during the four phases. A detailed description of events follows:

- *phase 1:* The shifting of all the control bits local in the instruction scheduler is carried out during phase 1. Their new values need to be stable before the start of phase 2, when the scheduling begins.

- *phase 2:* Due to the serial nature of the oldest first algorithm, the scheduling for the ALU starts one phase earlier. Since the multiplier shares a port with the ALU, it also starts its scheduling in this phase. A lookup array searches for the oldest pending store regardless if it has its operands ready to prevent the scheduling of any subsequent load instructions. Also, a lookup array searches for the oldest pending load instruction and prevents scheduling of any following store instruction.

- *phase 3:* The scheduling of a control transfer instruction is carried out in this phase. Also, the pending load and store information generated in the previous phase is now used to schedule load and store instructions. The store and third ALU instructions use a common lookup array since they share the same port. Because of the dynamic nature of a lookup array, a store instruction could override the last ALU instruction to be scheduled. This does not create a problem since the oldest first algorithm is still maintained.

- *phase 4:* The match lines from scheduling are driven onto the src1 and src2 fields through the read drivers. The operand values and control bits are delivered to the appropriate functional units.

5 Conclusion

In summary, we presented our design and implementation of a centralized instruction window which is an integral part of a superscalar architecture. It is capable of decoding and issuing four instructions per cycle. It was verified by simulation on several different benchmarks and can tolerate a 100 MHz clock rate with considerable safety margins.

An important decision was the use of a FIFO which allowed the use of special logic arrays for fast searching and scheduling of instructions. Full bypassing of instructions was implemented, and the critical path came from generating the bypassing control and delivering the operands to the functional units.

References

[1] Val Popescu et al., “Metaflow Architecture,” *IEEE Micro*, pages 10–13,63–73, June 1991.

[2] Mike Johnson, *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, 1991.

[3] Steven Wallace and Nader Bagherzadeh, “Performance Issues of a Superscalar Microprocessor,” *Proceedings of the 1994 International Conference on Parallel Processing*, volume 1, pages 293–297, August 1994.

[4] Steven Wallace, Nirav Dagli, and Nader Bagherzadeh, “Design and Implementation of a 100 MHz Reorder Buffer,” *37th Midwest Symposium on Circuits and Systems*, August 1994.

[5] Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design*, Addison Wesley, Reading, MA, 1993.