IMPROVING POWER of L1 DATA CACHE and REGISTER FILE UTILIZING CRITICAL PATH INSTRUCTIONS

A Thesis

prepared by

Kuang-Lun Chen

in the

Department of Electrical Engineering

Lakehead University

Thunder Bay, Ontario, Canada

December, 2013

Abstract

As transistor's feature size shrinks, power becomes one of the limiting factors in design of modern processors. Cache and register file are the two power hungry components in processors, consuming more than one third of total processors' power budget. In this thesis, we propose new architectures for cache and register file to reduce power consumption. In the new architectures, we have SRAM cells operating at two different voltage levels and we change the structure of the cells so that they dynamically switch between nominal and reduced supply voltage.

Since power is proportional to voltage squared, an effective method to reduce power is lowering supply voltage. However, one of the side effects of using SRAM cells with reduced voltage is performance penalty. As supply voltage reduces, it takes longer to read/write from/to an SRAM cell. In this thesis, we exploit critical path instructions to overcome the performance impact of voltage scaling. Critical path instructions are chain of dependent instructions that constrain speed of processors. Those cells that are accessed frequently by critical instructions are assigned to use nominal supply voltage to preserve performance. On the other side, the cells that are seldom accessed by critical instructions are assigned to low supply voltage to reduce power consumption. To reduce overhead of voltage switching, we monitor critical instructions within long intervals and adjust the voltage of cells only when the intervals are elapsed.

We have evaluated our optimization techniques using a combination of circuit and architectural simulators. First, we used HSPICE to measure both dynamic and static power and also latency of SRAM cells for nominal and reduced supply voltages. Then, the results from HSPICE were fed into Simplescalar for architectural evaluations. Our simulation results reveal that the low power cache and register file reduce power consumption significantly with negligible impact on performance.

Acknowledgements

I thank my supervisors at Lakehead University, Dr Ehsan Atoofian and Dr Ali Manzak. It's their common interest spawned the idea and the topic of this research. Especial thanks to Dr Ehsan Atoofian for his financial support coming in time of need and also for his valuable advises on Simplescalar simulations.

I also would like to thank my parents, Pin-Ching Chen and Shiow-Min Yang who have been given me encouragements since I was a child. Thanks go to my sister Ellen Chen too who walked me through various difficult times.

Table of Contents

List of Figures

List of Tables

Section 1 Introduction

1.1 Motivation and outline of study

Semiconductor technology scaling in the recent decades has enabled significant improvement in performance of processors. However, this performance improvement is accompanied by an increase in power consumption of processors [1]. Higher power consumption requires expensive packaging and cooling systems, increasing cost and reducing reliability of computer systems. As such, power consumption became the first-order design constrain in all aspects of computer design.

In contemporary processors, a large fraction of chip area is dedicated to the caches and register files and with each technology generation, this fraction continues to grow. For example, Gowan et al. estimated that caches are the third highest power hungry component in DEC's Alpha 21264 microprocessor, consuming 15% of the total power [2]. S. Park et al. suggested that register files may consume up to 25% of total processor power when running embedded programs [21]. An effective method to reduce power consumption is to lower the power supply voltage. Since power is proportional to supply voltage squared a moderate reduction in voltage leads to significant reduction in power. However, the downside of lower voltage is performance degradation. Therefore, one must be careful on applying voltage scaling so that the impact on performance is minimal.

The motivation of this study is to find a way of balancing performance improvement and power reduction. We exploit critical path information to reduce power of L1 data cache and register file and at the same time reduce performance loss due to reduced supply voltage.

To limit performance degradation of voltage scaling, we use critical path instructions. Critical path instructions are chains of dependent instructions that determine programs' execution time [3]. To boost performance of processors, it is necessary to execute critical path instructions as quickly as possible. On the other side, instructions that can be delayed one or more cycles without affecting programs' completion time are called non-critical instructions. E. Tune *et al.* proposed a number of methods to identify critical path

instructions dynamically [3]. We use oldest instruction method to detect critical instructions as this method is more accurate than the other methods, such as Qcons in which the number of consumers determines the critical path instructions and Freed in which the number of freed-to-enqueue instructions determines the critical path instructions [3]. In this method, we examine the oldest instructions in issue window and mark them as critical if their source operands are not ready. To enhance speed of processors, critical instructions should use fast resources because they limit the execution time. Non-critical path instructions can use slow resources with negligible penalty in time in exchange for power improvement.

We exploit critical path information in two parts of a Superscalar processor: L1 data cache and integer- and floating point- register files.

Our new L1 data cache is structured into super-sets which are a number of cache rows. Each super-set is provided with nominal and reduced supply voltage and is capable of switching between the two voltages through voltage regulators. The nominal voltage preserves performance and the other supply voltage lowers power with negligible penalty in time.

We decide on supply voltage of super-sets based on critical instructions. The super-sets accessed by more critical load/store instructions and their corresponding analog blocks are assigned to use nominal voltage. On the other side, other super-sets and their corresponding analog blocks are assigned to low voltage supply for power reduction. As such, we reduce power of caches without prolonging execution time of critical instructions.

Our new integer- and floating point- register files are also structured into super-sets. Similar to the new cache, the registers that are accessed frequently by critical instructions are assigned to use nominal voltage to preserve performance. On the other side, registers that are rarely accessed by critical instructions use reduced voltage to lower power consumption. We use two analog blocks powered at different voltage levels and a lookup table containing the voltage information of each super-set. When a register is accessed by a processor, the corresponding analog block is enabled to read or write the register.

In this study, we use HSPICE to model and simulate the operations of the cache and register files. The dynamic and static powers in each cycle are measured through HSPICE and then fed into SimpleScalar for detailed architectural simulations.

1.2 Thesis Overview

The rest of the thesis is organized as follows. In chapter 2, we review related work and discuss a number of circuit-, architectural-, and software- level techniques for reducing power of functional blocks, caches, and register files. In chapter 3, we explain Superscalar processor which is the baseline architecture used in this study. In chapter 4, we elaborate circuit level structures for both L1 data caches and register files utilizing dual supply voltages in details. Critical path instructions and the techniques to identify them in a program are explained in this chapter. In chapter 5, we report simulation results for low-power cache and register file. We also discuss the effect of a number of architectural parameters on power and performance. Finally, we conclude the study and propose the future work in chapter 6.

Section 2
Literature Review

(1) A number of prior techniques exploited critical path instructions to reduce power while preserving performance. H. Yang *et al*. studied software pipelined loops, a compile-time instruction scheduling technique to reduce power [5]. The authors claim that in more than 75% of execution cycles, at least one instruction exists which has slack. A slack instruction is an instruction which its execution can be delayed without impacting performance. They used software pipelining to reschedule instructions in loops so that slack instructions are executed by slow and power efficient functional units.

Figure 2-1 shows the data dependence graph of a loop. The nodes of the graph represent functional units in a processor such as adder, multiplier, etc. The edges in the graph represent data flow in the program. For instance, the edge from S2 to S3 shows that S2 produces data that is needed by S3. In this example, a carry-over loop, s5 to s4, is marked by a gray circle and this carry-over loop also is marked in gray cells in Figures 2-2a and 2-2b. We assume that there is no dependency between iterations of the loop. The authors use this example to compare two architectures: Architecture 1 has three fast integer Add units and two fast integer Multiply units (op-code mnemonic: Mult). The latency of Add is one cycle and the latency of Mult is three cycles. Architecture 2 replaces one Add and one Mult in Architecure 1 with slow units. The latency of the slow units is twice the latency of fast units.

In Architecture 1, all resources are fast and Add instruction requires one cycle to execute. Therefore, s2 is issued one cycle ahead of s3. Figure 2-2a shows time steps of this architecture. In Architecture 2 (Figure 2-2b), s2 and s3 are scheduled in the slow Add and Mult units. To reduce impact of slow units on performance, instead of issuing s2 in time step 1, now it is scheduled in time step 0 (2 cycles ahead of s3). Both architectures have the same initiation interval of two, even the s5 to s4 carry-over loop appears later in architecture 2. Initiation interval is the number of clock cycles separating initiation of successive iterations of a loop [23]. That is to say, in both architectures, s0 of consecutive loop iterations are two cycles apart. The overall delay penalty is only two cycles for Architecture 2 regardless the

number of iterations. With more iteration, the power saved by using slow units can overcome the two cycle penalty.

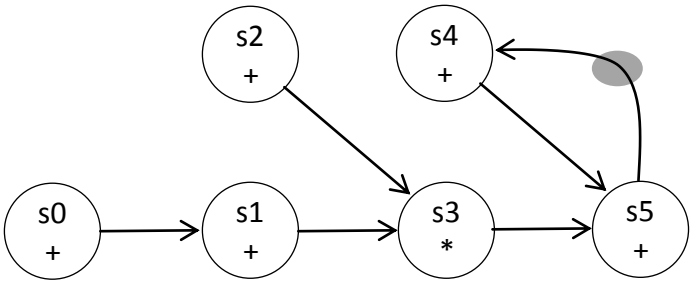


Figure 2-1: Example loop [5].

Iteration	Time Steps									
	0	1	2	3	4	5	6	7	8	9
0	s0	s1,s2	s3		s4	s5				
1			s0	s1,s2	s3		s4	s5		
2					s0	s1,s2	s3		s4	s5

(a)

Iteration	Time Steps												
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	s0,s2	s1	s3					s4	s5				
1			s0,s2	s1	s3					s4	s5		
2					s0,s2	s1	s3					s4	s5
3							s0,s2	s1	s3				
4									s0,s2	s1	s3		

(b)

Figure 2-2: (a) A schedule for architecture a1 (b) A schedule for architecture a2 [5].

H. Yang *et al*. show that this technique reduces power of functional units up to 31% with a minimal performance impact. However, the shortcoming of software scheduling is that it needs recompilation when the program runs on different machines. On the other side, we focus on hardware solutions which do not require recompilation of programs.

(2) J. Seng *et al*. used a dynamic critical path predictor to lower the power consumption in functional units and in instruction issue window [6]. This hardware solution allows the

processor to detect phase changes in programs dynamically and is more suitable for a cache design.

They explored various architectural configurations regarding their performance and power improvement. The configurations vary from six fast functional units and zero slow unit to zero fast unit and six slow units. They show that, over 50% of the performance gap between 0-fast/6-slow and 6-fast/0-slow configurations can be filled with only a single fast functional unit. This is due to the fact that the critical path instructions create a chain of dependency - one instruction waits for the prior instruction to finish. When the prior instruction is finished, the functional unit is also freed up for the next task. Therefore, with a few fast functional units, it accomplishes most of the work.

One important factor regarding performance and power is the ratio of IPC (instruction per cycle) to power. When power is lowered and/or the performance is improved, IPC to power increases. The authors also explored different number of functional units and various instruction queues regarding both sizes and organizations and benchmarked them with IPC to power ratio. It is found that the advantage of having mostly slow functional units is significant. This confirms the characteristic of a chain of dependency.

One of the shortcomings of their work is that they did not use accurate circuit simulations to extract static and dynamic power of functional units and issue window. On the other side, we model SRAM cells in transistor level and use HSPICE to calculate static and dynamic power and feed them to architectural simulations.

(3) R. Bahar *et al*. exploited the critical path instructions and classified the cache misses into critical and non-critical misses [7]. The non-critical misses are placed in a penalty buffer while critical misses remain in the main cache. The rationale is that the penalty for an L_1 cache miss and retrieval of data from other levels of caches and/or main memory is too costly in terms of time and power. It can be seen in Table 2-1 that L2 cache access in iMac G5 processor takes more than 3 times longer than that of L_1 cache access [28]. If the critical instruction data can be kept longer in the cache against the replacement policy, it improves

both performance and power. They show that this method improves performance of processors by 4% and saves power of L_1 data caches up to 13%. They later on developed a strategy for identifying the latency-tolerant data at runtime and placing them directly into a non-critical buffer, instead of moving cache misses into a penalty buffer as in their prior work [8].

Table 2-1: Memory Hierarchy, iMac G5 [28]

iMac	Reg	L_1 Inst	L_1 Data	L_2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency Cycles, Time	1, 0.6 ns	3, 1.9 ns	3, 1.9 ns	11, 6.9 ns	88, 55 ns	10^7, 12 ms

Their work focuses on both L_1 and L_2 data cache as a whole and reduces power by increasing hit rate of L_1 data cache or by reducing traffic to L2 cache. The power saved is mainly due to L_2 cache. Our approach is different since we target only L_1 cache by utilizing two different voltage supplies.

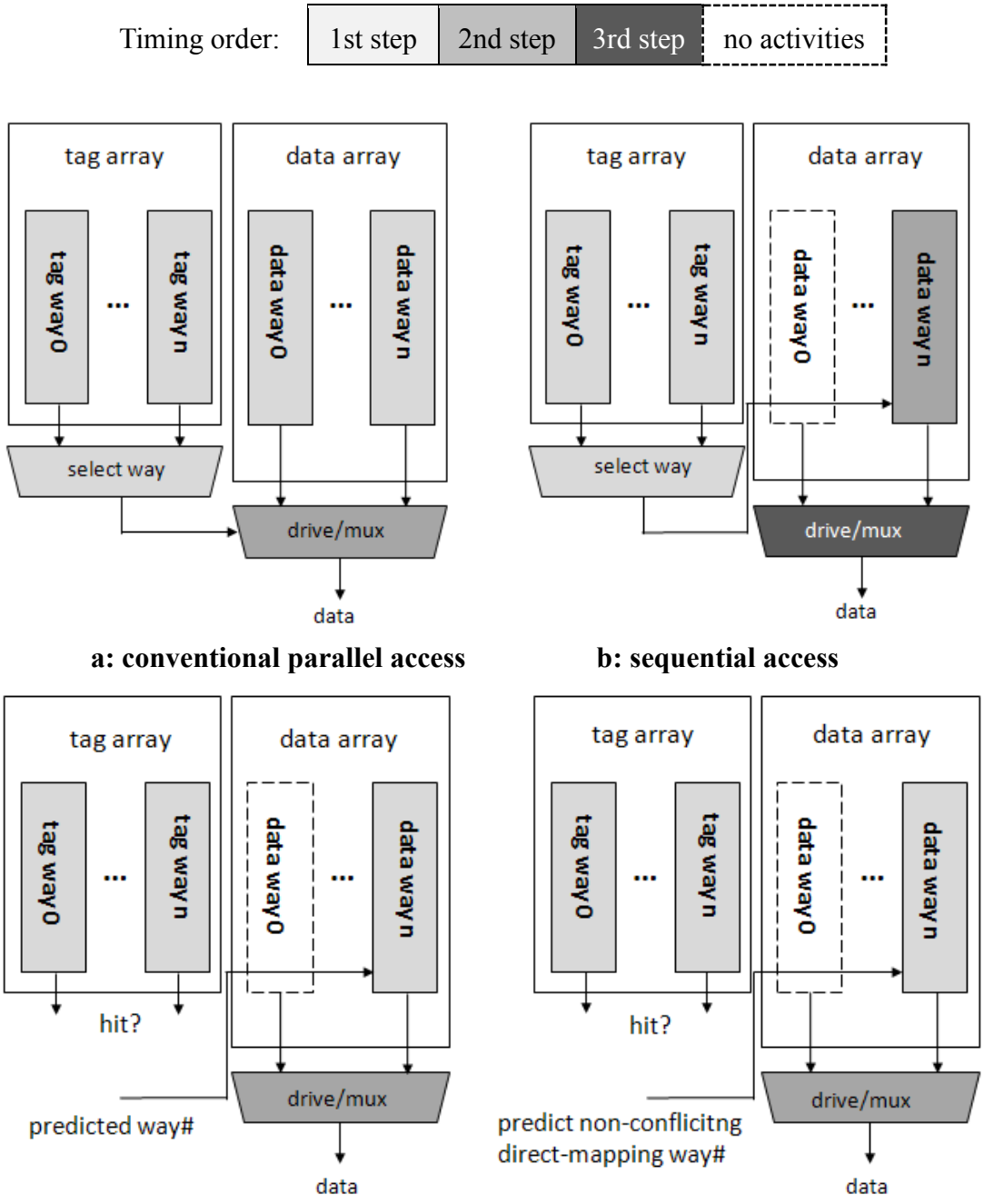
(4) R. Balasubramonian *et al*. applied the instruction criticality to L_1 data and instruction caches by partitioning the caches into 2 different banks [9]. The cold banks are slow and low power while the hot banks are fast and power hungry. They steer and place data in the data cache based on the percentage of critical instructions accessing the cache blocks. This instruction and data reorganization however increases the contention and degrades the performance in the conventional caches as it lacks the flexibility of resizing the cache banks to adapt to the variability of programs. For instance, when a section of a running program has more critical load/store instructions, the hot banks may experience frequent data replacement from memory.

Their technique is similar to having two static caches in the system. One is fast and power hungry and the other is slow and power efficient. On the other side, the approach we propose in this work is able to dynamically assign and convert a portion or all of the cache cells into high supply voltage as needed in a running program.

(5) A set-associative cache checks all tags in a set in parallel to accelerate look-up time. However, if there is a hit, only the matching way provides data. As such, the conventional caches waste energy in exchange for lower access time. M. Powell *et al*. exploited way-prediction and selective direct-mapping to access the way that is likely to have the matching tag and to avoid accessing other ways if the prediction is correct [12]. Various set-associative cache access techniques are depicted in Figure 2-3.

Figure 2-3a shows the conventional set-associative cache. All ways are accessed in parallel; however, only the data in the matching way will be multiplexed out to the processor. This wastes (n-1)/n*100 percent of energy where n is the number of ways. Figure 2-3b shows a sequential technique. In this technique, ways are accessed sequentially. In the first step, way0 is accessed. If a hit happens, data is returned to the processor; otherwise, way1 is accessed. This procedure is repeated for way2 and way3. The main drawback of this method is that it takes longer to retrieve data from cache. In way-prediction (Figure 2-3c), the matching way is predicted based on history of cache accesses. Way-prediction requires searching a history table to determine which way contains the address. The selective direct-mapping cache (Figure 2-3d) has both direct-mapped and set-associative ways. All the cache accesses are categorized as non-conflicting and conflicting. For non-conflicting addresses, we access direct-mapped cache to reduce power. For conflicting addresses, we access set-associative ways to reduce performance degradation.

Way prediction and selective direct-mapping are orthogonal to our work and can be combined with our dual-supply SRAM cells.



a: conventional parallel access **b: sequential access**

c: way prediction **b: selective direct-mapping**

Figure 2-3: Access and timing for design options [12].

(6) S. Yang *et al*. used an integrated circuit and architectural approach to reduce static leakage current [13]. They proposed to dynamically resize an instruction cache to adapt to application's required size based on the cache miss rate.

Figure 2-4 shows the anatomy of their Dynamically Resizable instruction cache (DRI i-cache). The miss counter counts the number of cache misses at each instruction cycle. At the end of each interval, depending on whether the miss counter is lower/higher than a threshold, the cache is either upsized or downsized by adjusting the number of masked index bits. The interval may take thousands of instruction cycles.

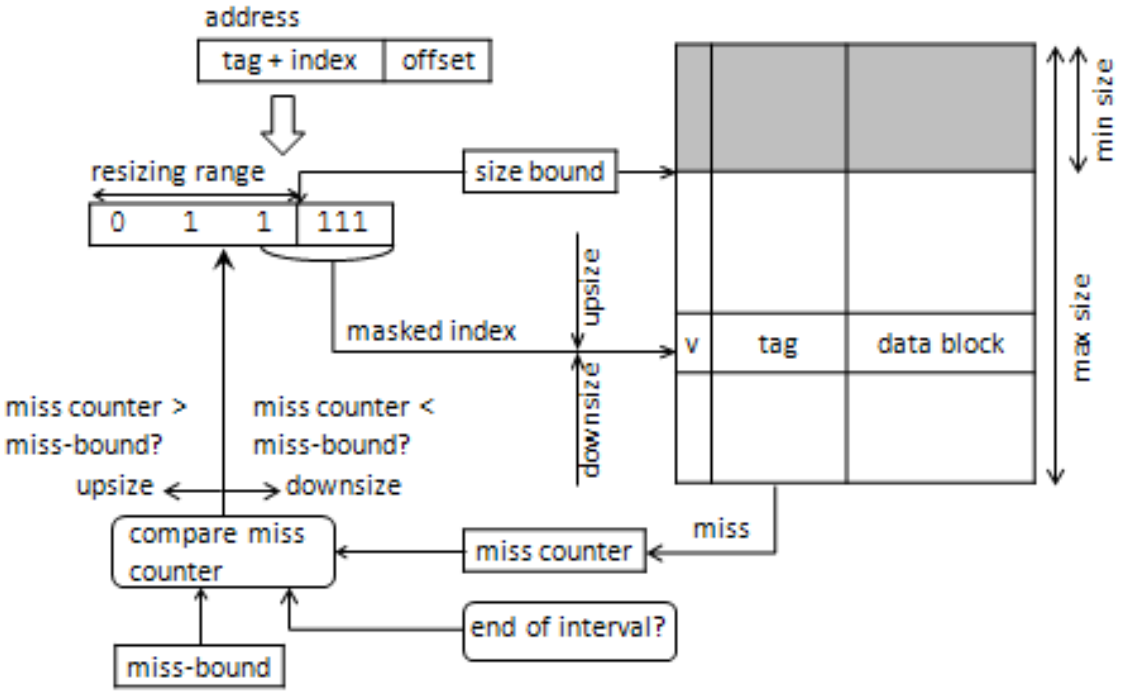


Figure 2-4: Anatomy of a Dynamically Resizable instruction cache (DRI i-cache) [13].

The unused sections of the cache are virtually turned off to eliminate leakage. By only using a smaller section of a cache and turning off all unused sections, they are able to reduce energy of caches with negligible effect on performance. This method is orthogonal to our optimization technique and can be used in conjunction with our power-aware cache cells to reduce power of caches further.

(7) J. Abella *et al.* proposed two cache architectures using different threshold voltage transistors [29]. One is of nominal threshold voltage (fast transistor) and the other is of high-threshold voltage (slow transistor) to reduce leakage current.

The first architecture is locality-based organization. In this organization, L_1 cache, called Fast Cache, is built using fast transistors. L_2 cache is built using slow transistors; therefore L_2 cache is called Slow Cache. The normal L_2 cache in the system is now called L_3 cache. They assumed the latency of Fast L_1 cache is one while the latency of Slow L_2 cache is two. This architecture factually turns a two level cache system into a three level cache system. Since the penalty of Slow L_2 is less than L_3, the power can be reduced comparing with the conventional cache organization. However, one can see that when there is an L_1 cache miss, the processor needs to access Slow L_2. This increases latency of memory hierarchy.

The second architecture is criticality-based organization. In this organization, there are a fast L_1 and a slow L_1 caches. Both of them are always accessed in parallel. If a critical load instruction hits in the slow cache and misses in the fast cache, the data is copied to the fast cache. If a critical load instruction misses in both caches, the data is retrieved from the following level of cache and only is stored in the fast cache. If a non-critical load instruction hits in either caches, data is not copied from one to the other since it is a non-critical load instruction. If a non-critical instruction misses in both caches, the data is then retrieved from the following level of cache and is only stored in the slow cache.

As one can see, comparing with locality-based architecture, total power saving in criticality-based structure is less since it accesses both caches in parallel. In addition, the data copying increases the overhead of this technique. Although criticality-based method may look inferior to the locality-based organization in term of power saving, it still has a significant power improvement comparing with conventional L_1 cache.

Our approach on the other hand, doesn't require complicated data copying or parallel cache look-up. Our cache can be accessed normally while capable of reducing power by converting sections of the cache to low-power cells.

(8) H. Hanson *et al*. utilized Multi-Threshold CMOS to reduce static energy in caches [17]. The sub-threshold leakage current depends on both temperatures (T) and transistor's threshold voltage (Vt) as indicated in the following equation.

$$I_{leak} \propto e^{(\frac{-Vt}{T})} \tag{2-1}$$

Therefore, the higher the threshold voltage is, the lower the leakage current. The Threshold voltage can be expressed as a function of V_{SB}; the voltage difference at a transistor's source and bulk/substrate which is indicated in the following equation:

$$Vt = Vt0 + \gamma\left(\sqrt{|2\Phi_F| + V_{SB}} - \sqrt{|2\Phi_F|}\right) \tag{2-2}$$

where Vt0 is the threshold for zero substrate bias (V_{SB}=0), γ denotes the body effect coefficient and $2\Phi_F$ is the surface potential [18].

Figure 2-5 is the schematic of their circuit to increase threshold voltage. When in sleep mode, the sleep signal is high, P3 and P4 are turned off, and P5 is turned on. This makes the body/bulk of P1 and P2 tied to Vdd+ while their source nodes are tied to two diodes. This boosts up PMOS's $|V_{SB}|$ and therefore, its threshold voltage increases. When it is not in the sleep mode, P3 and P4 are turned. Therefore, it connects the source nodes and the body/bulk of P1 and P2 to Vdd. In this mode, $|V_{SB}| = 0$; therefore Vt = Vt0.

The same analysis can be easily followed for N1 and N2. This circuit which consumes a small portion of a cache's total power reduces static power. On the other side, our approach mainly focuses on the dynamic power.

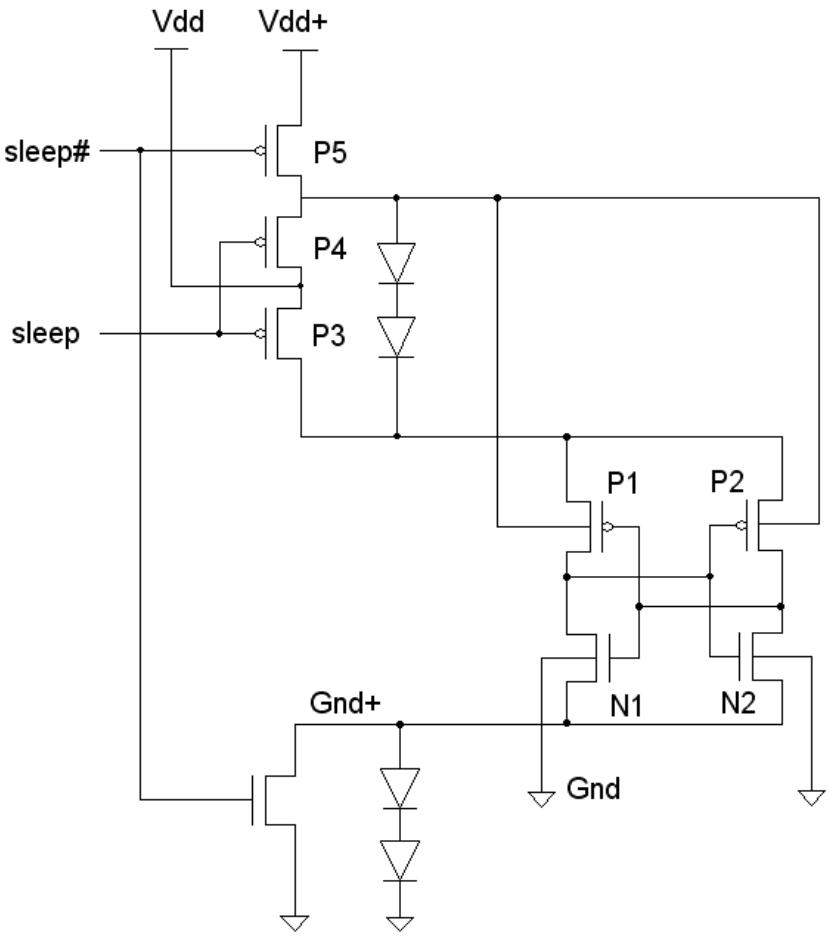


Figure 2-5: MTCMOS SRAM cell schematics [17]

(9) J. Kin *et al*. proposed a filter cache structure [19] to reduce power consumption in caches. Their filter cache is another cache which is significantly smaller than traditional L_1 cache. Due to its smaller size, the capacitive load on bit line and bit line bar of the SRAMs is much smaller and therefore provides shorter access time. Also, due to less capacitive load on bit line and bit line bar, it consumes less power than the traditional L_1 cache. Figure 2-6 shows the filter cache.

The down side for this structure is that the filter cache has higher miss rate than L_1 cache. When a miss happens, the processor needs to sequentially access L_1 cache which elongates the data hit time. The author's hypothesis is that the reduction in power consumption will

compensate the performance loss and results in a lower energy×delay product. Due to the possible high miss rate, the authors also suggest that this filter cache can be turned off or bypassed when higher performance is needed.

This filter cache design may look similar to the locality-based cache organization proposed by J. Abella *et al.* [29] which was discussed earlier in this section. However, in locality-based cache organization, the second level of L_1 cache is a slow cache. By utilizing critical path prediction, the instructions are placed in either fast L_1 or slow L_1 to achieve power reduction with minimum performance loss. Here, J. Kin *et al.*'s structure utilizes the smaller capacitive load on bit line and bit line bar to achieve the same goal. Filter cache is orthogonal to our optimization techniques and can be used in conjunction with our SRAM cells to reduce power of caches further.

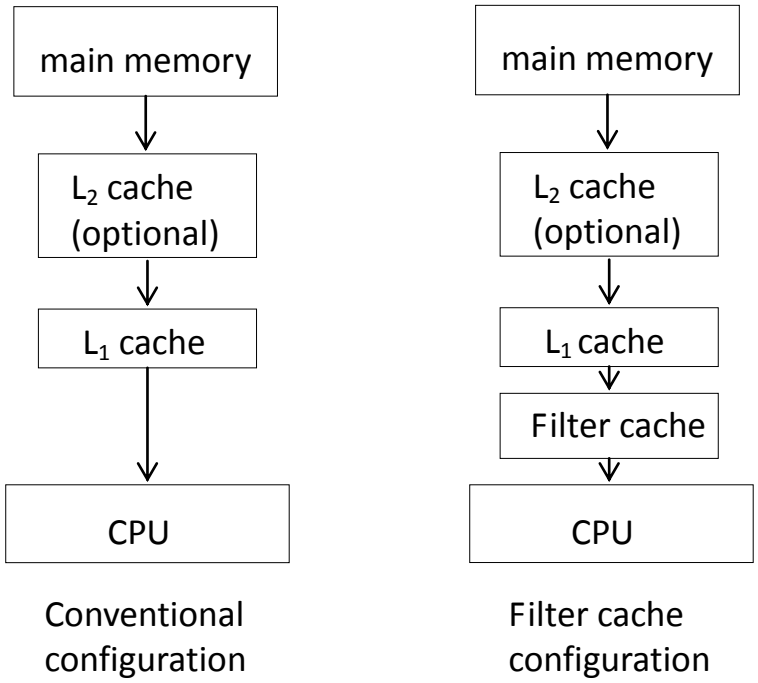


Figure 2-6: Traditional cache vs filter cache [19]

(10) C. Su *et al*. proposed block buffering to reduce power consumption in caches [20]. This technique is similar to the filter cache proposed by J. Kin *et al*. [19]. It saves power by using a buffering cache which is faster and more power efficient than L_1 cache. In C. Su *et al*.'s analysis, this structure is beneficial to the applications with high spatial locality patterns. Spatial locality is also called locality in space. It means that if an addressed in cache is referenced, the nearby addresses will be referenced soon.

They also proposed a cache sub-banking structure. Conventionally, when a cache is accessed, the entire cache line/row is accessed. For instance, for a 64kB cache with 64 Byte cache line, the whole 64-byte is read or written at the same time. If the location of the requested data in the 64-byte cache line is known, then there is no need to access unrequested data in the same cache line. Figure 2-7 depicts the sub-banking cache structure.

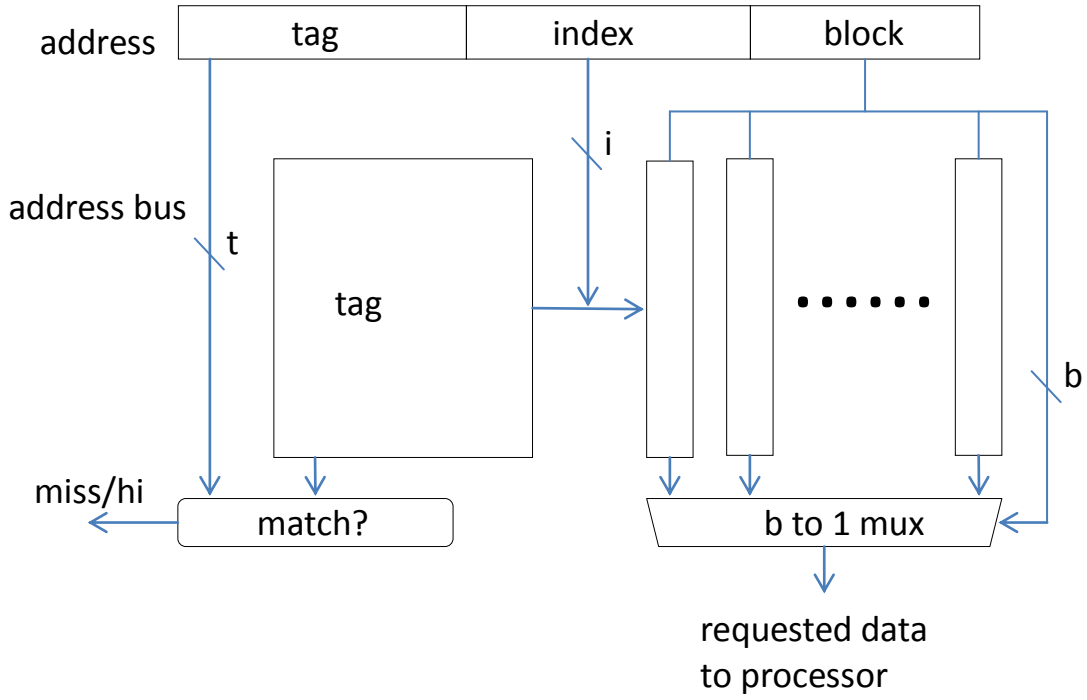


Figure 2-7: Structure of a sub-banking cache [20]

Their approach is orthogonal to our optimization technique and can be used in conjunction with our cache cells to reduce power of caches further.

(11) S. Park *et al*. reported that register files are hot spot units and may consume up to 25% of total processor power when running embedded programs [21]. J. Tseng *et al*. found that on average, 36% of instructions get their source operand values from forwarding bus [22]. This implies that 36% of register file power is wasted. Figure 2-8 shows architecture of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor. The data bypassing/forwarding paths are drawn in bold lines. It clearly points out that the register file is accessed at early instruction decode stage. At the late instruction decode stage, the multiplexers determine whether the source operand values should come from forwarding bus. If data comes from forwarding bus, then the data read from register file are discarded. If this type of unnecessary reads can be avoided (bypass-skip) or the register file is only accessed when there is no data bypassing path (on-demand reads), the power is reduced. J. Tseng *et al*. reported that the bypassing-skip technique leads to an average of 16% of power saving.

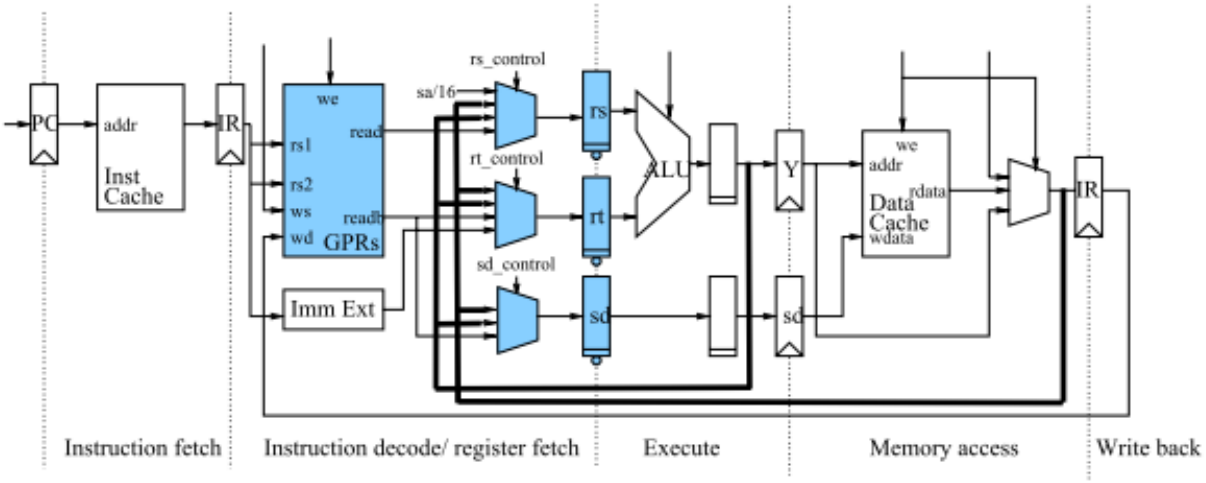


Figure 2-8: MIPS pipeline processor [21]

Traditionally, processors try to separate data dependent instructions as much as possible through inserting non-dependent instructions between them. This is necessary to avoid read-after-write hazards. S. Park *et al*. utilized the bypass-skip technique proposed by S. Tseng *et al*. along with a bypass sensitive algorithm to reduce the number of register file accesses. In their bypass sensitive algorithm, instead of separating data dependent instructions, they try to

schedule them as close as possible, so that most of the operand values are bypassed from other stages instead of coming from register files. Their heuristics shows 12% reduction in register file reads.

(12) J. Ayala *et al*. [31] proposed a technique which reduces the power consumption in register files by putting unused registers to a drowsy state with the support of compiler [32]. X. Guan *et al*. also proposed a similar technique [33]. Figure 2-9 shows their modified drowsy register file cell. This cell can be supplied by either 1v or 0.3v through selecting one of the two PMOS transistors on top of the Figure. When 0.3v is selected, the voltage is too low for the cell to operate (write or read) but is high enough to retain the cell's data. Once the cell is woken up by setting the 1v supply voltage, it can operate normally.

To support this design, they extend ISA with two additional instructions; one turns a set of registers to the drowsy state and another wakes up registers to normal mode. The compiler is responsible to use these instructions to manage register file reconfigurations. In the code generation phase, the compiler needs to detect hot spot regions such as loops and repeated functions and mark the beginning and the end of such regions. For these regions, the register file cells are set to use 1v supply voltage. For other regions, the register file cells are set to use 0.3v supply voltage. As such, power consumption of register file is reduced.

The authors also point out that turning the individual register on and off results in significant timing overhead and degrades performance. They propose turning a group of registers on and off to reduce overhead of this technique. They reported 40% reduction in power of register file with groups of 16 registers.

One of the drawbacks of this method is that it requires recompilation when the code is run on a different machine. Our approach, however, exploits the critical path information which doesn't need the compiler. Another difference is that their drowsy state only retains data while in our approach, the register cells function normally (write and read) at the low supply voltage with only one cycle penalty in time.

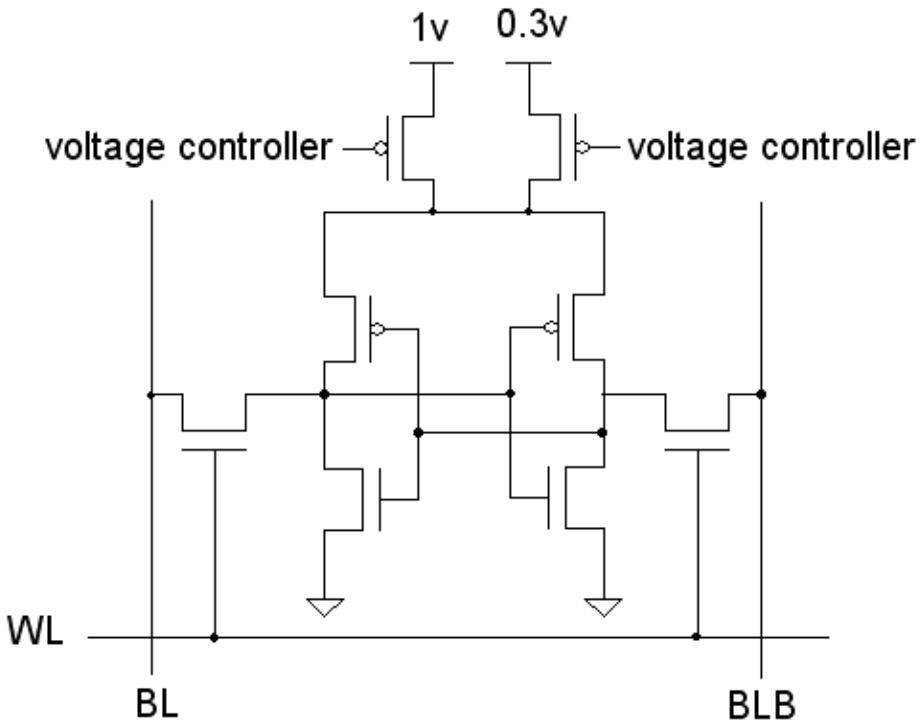


Figure 2-9: A drowsy register file cell [31]

(13) N. Gong *et al*. [35] pointed out that status of a register changes over time: empty, idle, ready and free. For example, when an instruction is renamed, its architectural register is mapped to a physical register, say R1 is mapped to P1. P1 then changes from a free state to an empty state due to this mapping. As the instruction executes, the status of P1 changes from empty state to ready state which allows write and read operations. When the last consumer instruction reads from P1 and before P1 is reclaimed back to the free list, P1 turns into an idle state. When the architectural register R1 is remapped, P1 is released and is retuned to the free list.

When the physical register is in ready state, the register must function normally. However, in empty state or idle state, the register can be put into a drowsy state to retain data and to reduce the leakage current. While the register is in free state, the register can be shut off to minimize the leakage current.

Motivated by different states of a register, they utilize gated-ground technique proposed in [36] to reduce static power in register file [35]. In their work, there are 3 states similar to [34]: work, drowsy, and dead state. Figure 2-10 shows the circuit of a low power cell.

When the cell is in work state (dead=0, drowsy=0), $N_{\mathbf{discharge}}$ is cutoff and $N_{\mathbf{data\_retention}}$ is on. So, the cell works normally. In the drowsy state (dead=0, drowsy=1), $N_{\mathbf{data\_retention}}$ is cut off and the voltage at the node (either Q or Q_bar) reaches a saturated voltage when it stores "0". Through proper transistor sizing, this saturated voltage will not be high enough to disturb the node when it stores "1" (turning on the NMOS in the branch which stores "1"). Therefore, the data is retained successfully. However, due to the stacking effect [37], the leakage current is reduced substantially. According to stacking effect, the leakage current through a 2-transistor stack is roughly an order of magnitude higher than that of a single transistor. The dead state (dead=1, drowsy=0) provides minimum leakage current by discharging the cell to "0". This is due to the fact that the leakage current of a cell storing "0" is smaller than the leakage current of a cell storing "1".

Their experiments show 12.7-14.1% power reduction in ROB-based (Re-Order Buffer based) microprocessors and 12.4-17.9% power reduction in checkpoint-based microprocessors. Our approach is different since we mostly focus on dynamic power of register file.

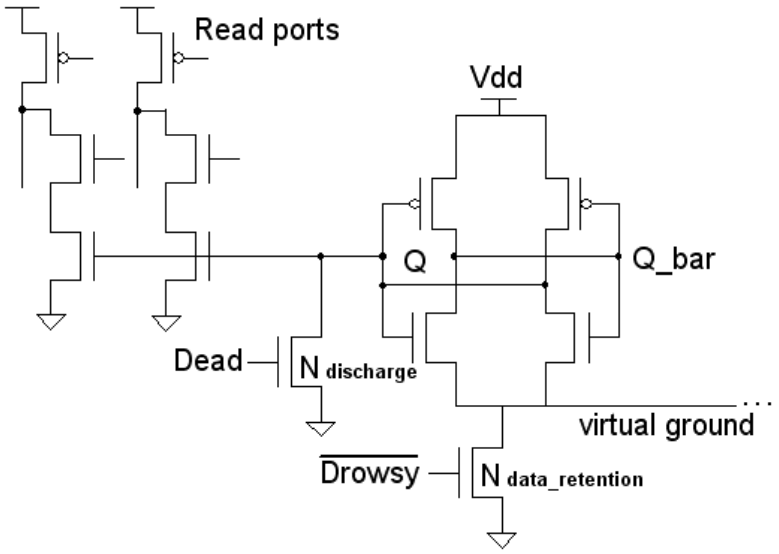


Figure 2-10: A gated ground register file cell [35].

Section 3
Super-scalar Processor Architecture

In this chapter, we discuss architecture of a superscalar processor briefly as it is the target architecture of this study. Section 3.1 explains how modern processors exploit instruction level parallelism. Sections 3.2 to 3.5 discuss the main stages of a superscalar processor: fetch, dispatch, issue, write-back and commit.

3.1 Introduction

Over the past several decades, computer technology has improved rapidly. This rapid improvement is the result of advances in semiconductor technology and innovative architectural techniques. Transistor shrinking has made it possible to build processors with multi-GHz clock frequency. The feature size of transistors reduced from 10um in 1970 to 28nm in 2011 [43], increasing clock frequency from a few Mhz to over 3 Ghz [44, 45].

One way to boost performance of processor is exploiting instruction level parallelism (ILP). Pipelining was one of the first architectural techniques which used parallelism in hardware. A pipeline processor is composed of several stages and these stages process instructions concurrently. This technique was initially developed in the late 1950's and became popular in large scale computers during the 1960's [24].

Figure 3-1a shows execution of a program in a processor that does not exploit ILP. Figure 3-1b shows the same program in a pipelined processor. We assume that each instruction goes through three stages: Instruction-Fetch (IF), Execute (EX), Write-Back (WB). In Figure 3-1a, instructions execute sequentially. This would take a total of 9 cycles to execute all three instructions. However, in Figure 3-1b, instructions execute in parallel. As soon as one instruction finishes up a stage, immediately the next instruction starts. By overlapping execution of instructions, the same instructions execute in five cycles.

```
                    C1 C2 C3 C4 C5 C6 C7 C8 C9
LD   R1, 0(R2)      IF EX WB
DSUB R4, R1, R5           IF EX WB
AND  R6, R1, R7                 IF EX WB
```

(a)

```
                    C1 C2 C3 C4 C5 C6 C7 C8 C9
LD   R1, 0(R2)      IF EX WB
DSUB R4, R1, R5        IF EX WB
AND  R6, R1, R7           IF EX WB
```

(b)

Figure 3-1: (a) An example of sequential execution. (b) An example of pipelined execution.

It's worthy to note that, in the pipelined example, the latency of an instruction is not shortened or improved. However, the throughput is significantly improved. This enhances the performance of processors as it takes less time to execute programs.

One of the limitations of a pipeline processor is that it fetches and executes at most one instruction per cycle. Therefore, the maximum performance of a pipelined processor is one instruction per cycle (IPC=1). A superscalar architecture improves performance of a pipelined processor by issuing multiple instructions per clock cycle. Likewise, a superscalar processor separates an instruction's lifetime into different phases. However, in contrast to the pipelined processor, each phase may have multiple instructions per cycle. Figure 3-2 summarizes execution flow in a superscalar processor. Instructions are fetched from a static program. The fetch stage uses a branch predictor to speculate the outcome of branch instructions. This stage generates a stream of dynamic instructions which is fed to the rest of processor. These dynamic instructions are inspected for any data dependences which may cause a processor to stall. The false data dependences (output and anti-dependences) are removed by register renaming to leave only the true data dependences. In the next step, the

stream of instructions is dispatched into issue window. In the issue window, instructions wait until their source operands become ready. Once source operands are ready, instructions are sent to free functional units for execution. From this stage, independent instructions may execute out of order. Instructions with true data dependences need to execute in order to guarantee correct outcome of programs. At the end, all the instructions are re-ordered and commit. In this last stage, if a branch misprediction is detected, all speculative instructions are squashed and instruction from the correct path are fetched and executed.

Figure 3-3 shows details of a superscalar processor. The main stages of the processor are: fetch, dispatch, issue, execution and commit. In the following sections, we will discuss each stage in details.

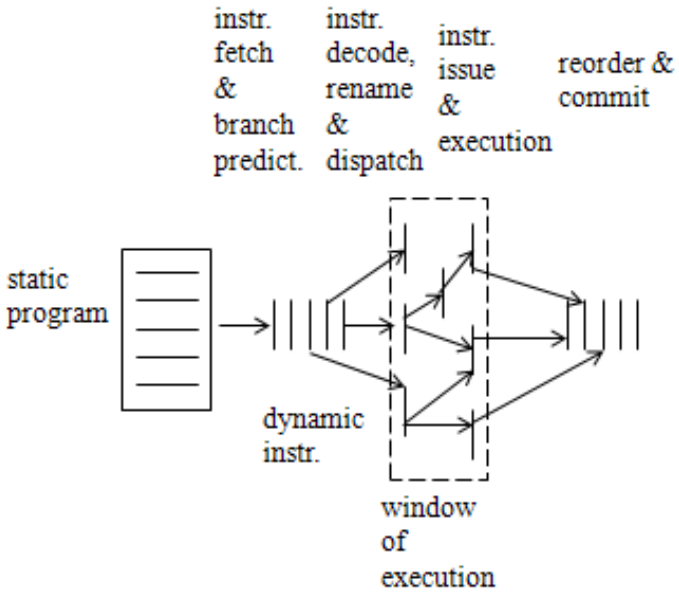


Figure 3-2: A conceptual execution flow of a superscalar processor [26]

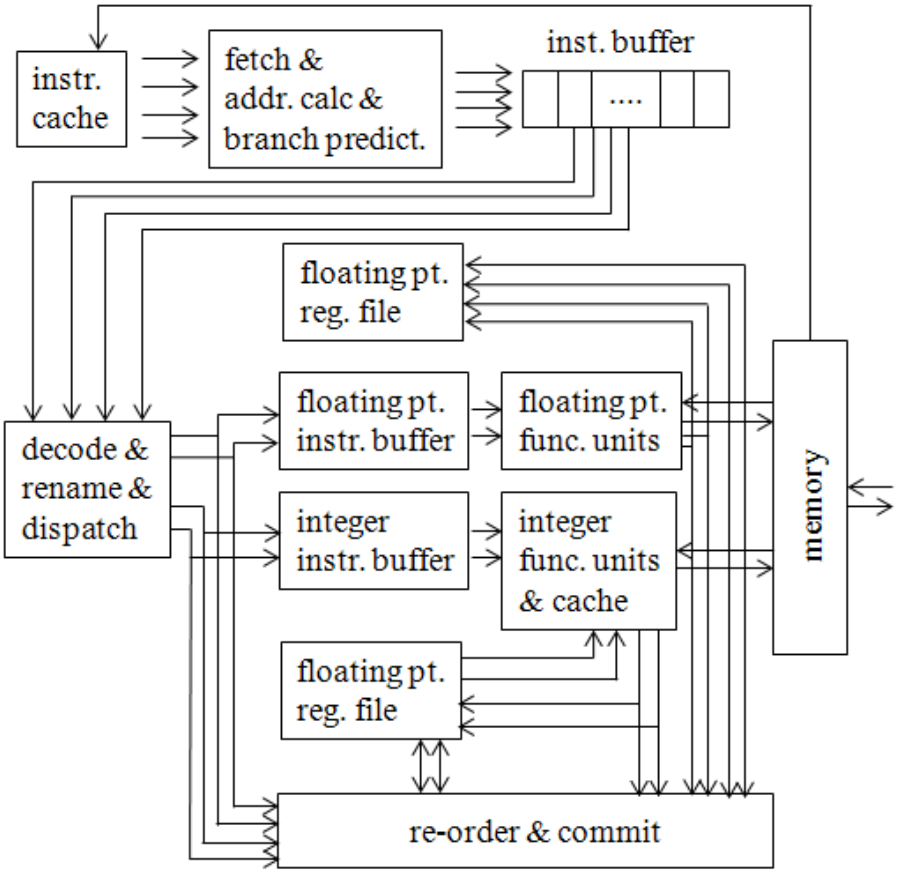


Figure 3-3: Organization of a superscalar processor [30]

3.2 Instruction fetch

The fetch unit reads instructions from memory and sends them to the rest of superscalar processor. Since main memory is slow, processors use cache, which is a small and fast memory unit, to reduce latency of memory. The address of instructions comes from Program Counter (PC). Instruction cache is searched with PC and if an instruction pointed to by PC is not found in the cache, then a cache miss occurs and memory should provide the instruction.

In fetch stage, PC is incremented to point to the next instruction. However, if a branch instruction exists in the program, then PC should be loaded with target address. If the processor stalls until the outcome of the branch is known, then it may take several cycles. Hence, modern processors incorporate branch predictors to overcome the delay of branch

instructions. If the branch is predicted correctly, then the processor continues executing instructions without interruption. If the prediction is wrong, then all instructions that have been executed speculatively should be squashed and the program will restart from the correct branch target. The simplest policy for branch prediction is Taken policy. Taken policy works very well for structures such as loops in programs. In a loop, the program jumps to the start of the loop several times before it exits the loop. Therefore, a Taken policy may be sufficient for loops in programs. However, for other control structures such as if-then-else statement, Taken policy may result in too many mispredictions. There are many other branch predictors [46, 47] which improve accuracy of Taken policy, but this is not the scope of this study.

Instructions fetched from memory are stored in an instruction buffer. The number of instructions fetched per cycle is usually higher than the instruction execution rate to ensure an uninterrupted flow of instructions. When instruction cache miss occurs the fetch unit may stall for a few cycles. Hence, to sustain continuous flow of instructions to the rest of processor, the fetch unit should fetch multiple instructions per cycle.

3.3 Instruction decode, rename and dispatch

In a single-cycle processor, an instruction reads its source operands from a register file, executes, and then writes the result back to the register before the next instruction is fetched. The data dependence does not create hazard in such a machine. However, in a pipelined processor, this is not true. Before an instruction writes the result to the register file, the next instruction may already be in execution stage with a wrong source operand. Figure 3-4 depicts an example of data hazard. The first instruction, SUB, writes r1-r3 to r2 at clock cycle 5 (CC5 in the figure). However, at clock cycle 5 the second instruction, AND, has already passed register read stage. Therefore, AND instruction gets the old data which is incorrect. One way to overcome data hazard is forwarding. As soon as SUB is executed, the result is forwarded to the AND [48].

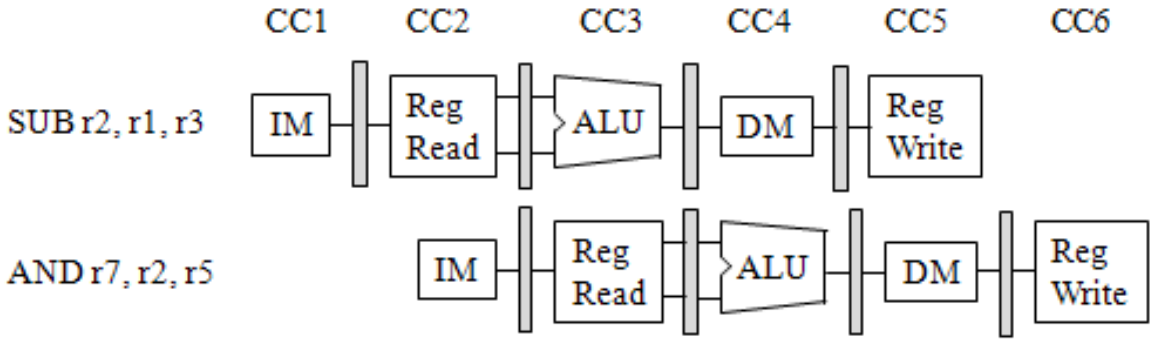


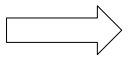
Figure 3-4: An example of data hazard [48].

In an out-of-order execution processor such as a Superscalar processor, the order of instructions may change in run-time and this may create new types of hazards. Figure 3-5a. shows a program with five instructions. MULD and ADDD write to the same register: F6. This is called output dependence. An out-of-order execution processor should not change the order of instruction with output dependence; otherwise, write after write (WAW) hazard is created. The other type of dependence is anti-dependence and it happens when an instruction reads from a register and later on, the same register is written by another instruction. For example, in Figure 3-5a. there is anti-dependence between SUBD and MULD because of F8. An out-of-order executing processor should not change the order of these two instructions; otherwise, write after read (WAR) hazard is created.

One way to eliminate output and anti-dependences (also called false dependences) is register renaming. Through register renaming, it is guaranteed that WAW and WAR hazards never happen. As such, a superscalar processor is able to change the order of instructions and reduce execution time. We elaborate register renaming through an example.

To eliminate output-dependence between MULD and ADDD, in Figure 3-5b, register F6 in MULD is renamed to P1. Any other instruction that depends on MULD, i.e. ADDD, uses P1 instead of F6. Now, ADDD does not need to wait until MULDD finishes. Since the output-dependence is removed, a processor is free to execute ADDD sooner than MULDD. This may reduce execution time of the program. Similarly, we can use register renaming to

eliminate anti-dependence. In Figure 3-5b, register F8 in SUBD is renamed to P2. Hence, SUBDD does not wait until MULD reads its source operands. It can start and even finish before MULD reads its source operands.



```
DIVD   F0, F2, F4
MULD   F6, F0, F8
SD     F6, 0(R1)
SUBD   F8, F10, F14
ADDD   F6, F10, F8
```

(a) before

```
DIVD   F0, F2, F4
MULD   P1, F0, F8
SD     P1, 0(R1)
SUBD   P2, F10, F14
ADDD   F6, F10, P2
```

(b) after

Figure 3-5: Register renaming

There are two register renaming techniques commonly implemented in microprocessors. The first uses a physical register file larger than the logical register file. The logical registers, also called architectural registers, are those registers used in a static program while the physical registers are those in the microprocessor. To track the mapping of logical registers to the physical registers, the processor uses a register renaming table. This table is indexed by logical registers and each entry of this table shows the physical register corresponding to a logical register.

When an instruction is decoded, the processor takes a physical register from the list of free physical registers and assigns it to the destination field of the instruction. Also, the register renaming table is updated. As part of renaming process, the physical register is then removed from the free list. The physical register is only reclaimed and placed back into the free list when the assigned physical register has been read and its corresponding logical register has been renamed to another physical register.

Figure 3-6 is an example of the physical register reclamation. The logic register F6 in MULD instruction in Figure 3-6a is renamed and mapped to physical register P1. F6 in ADDD instruction is renamed and mapped to P6. SD uses P1 as its source operand. Therefore, P1 can be reclaimed and placed back into free list only when ADDD instruction commits. Only at that time, we are sure that P1 is not needed by any other instruction.

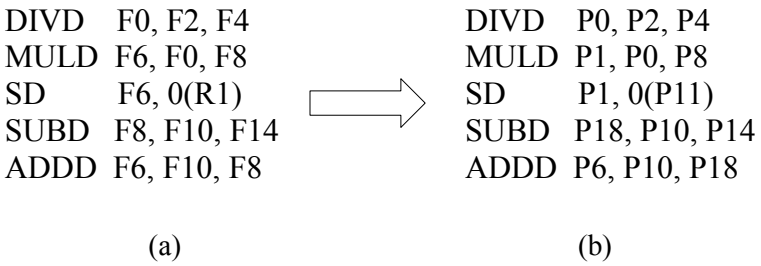
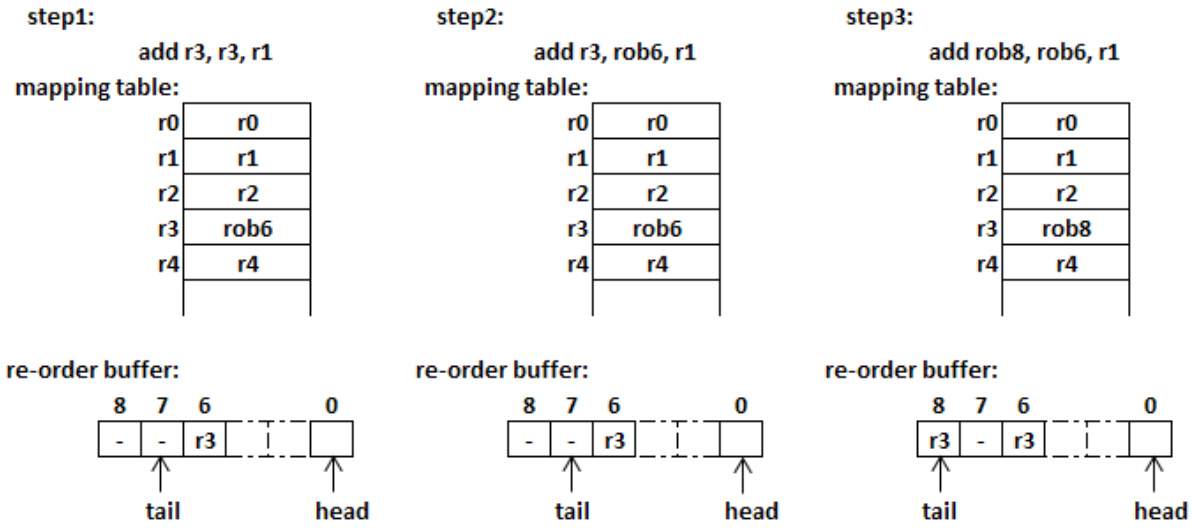


Figure 3-6: Physical register reclamation

The second method of register renaming is to use a re-order buffer. Re-order buffer is implemented in a processor as a circular buffer with head and tail pointers. When an instruction is dispatched, it is inserted to the tail entry of the re-order buffer. The result (value of destination operand) of the instruction is written into the assigned entry in the re-order buffer. In the renaming process, the logical source operands are used to access a mapping table to locate the source operands. We use the example in Figure 3-7 to illustrate this.

Before the instruction add r3,r3,r1 is dispatched, r0~r2 and r4 reside in register files and r3 resides in re-order buffer #6 (rob6) as depicted in Figure 3-7a. At this point, the head of re-order buffer points to entry #0 in the re-order buffer and the tail points to #7. When the instruction add is dispatched, the processor uses its source operands to access the mapping table and locate the locations of their values as this value can be either in register file or in another entry of the re-order buffer. In this case, the value of r1 resides in register file (r1) and the value of r3 resides in re-order buffer #6 (Figure 3-7b). The add instruction is then placed in re-order buffer #8 by the processor and the tail pointer is moved to point to entry #8 (Figure 3-7c). Since its result will be inserted into re-order buffer #8, the instruction is renamed to “add rob8, rob6, r1” and the mapping table is updated accordingly as in Figure 3-7c.



(a) **(b)** **(c)**

Figure 3-7: Renaming in re-order buffer

3.4 Instruction issue and parallel execution

After register renaming, instructions are issued to functional units. At this stage, the processor checks for availability of data and resources. Ideally, when the input operands are ready, the instruction can be issued. However, in reality with limited hardware resources, some of the instructions may be stalled until resources are freed up.

There are a number of ways of organizing instruction issuing buffers. The most common one in superscalar processor is to use reservation station (RS). Reservation station is a set of buffers and each buffer holds an instruction. Figure 3-8 shows fields of an RS buffer. When an instruction is dispatched to an RS buffer, source operands are read from register file if they are ready (e.g. r1 in Figure 3-7b). If the source operands are not ready, then the instruction is written into RS and constantly monitors completing instructions. If designator of a completing instruction matches designator of a source operand, then the value on the bus is copied into source operand's data value (Figure 3-8) and the valid bit of the source operand is set to indicate the readiness of the operand. When all the source operands are ready (valid bits are set), the instruction is ready to issue.

Notice that to increase Instruction Level Parallelism (ILP), multiple instructions can be issued at the same time in a superscalar processor, provided that there are enough hardware resources.

opcode	source designator 1	data value 1	valid bit 1	source designator 2	data value 2	valid bit 2	destination designator

Figure 3-8: A typical reservation station

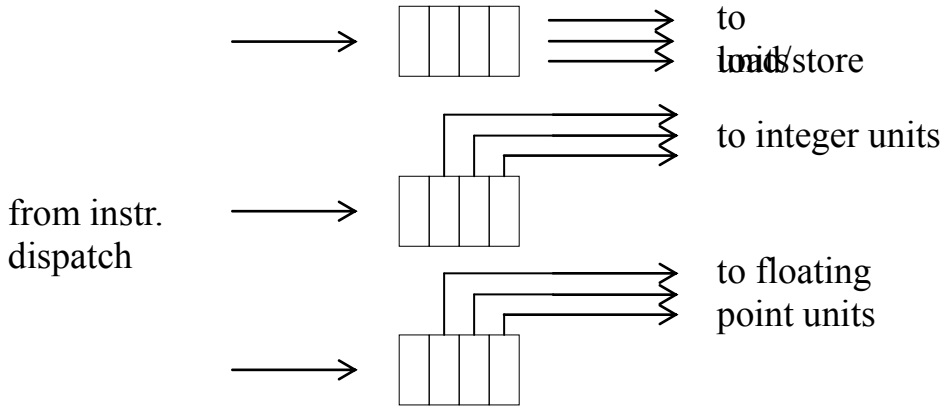


Figure 3-9: A multiple reservation station queue

3.5 Memory and cache

3.5.1 Memory gap

Over the past decades, the performance of processor and memory increased rapidly. However, the performance of processor increases much faster than memory. This resulted in increasing gap between processor and memory (Figure 3-10). One way to reduce this gap is to put a small and fast memory unit, which is called cache, between processor and main memory.

Caches are able to exploit principle of locality and hide latency of memory. According to locality, programs tend to use instructions and data that they have used recently. An

implication of locality is that we can predict which instructions or data will be used in near future based on recent instructions or data.

There are two types of localities: time and space. Locality in time means that if an address (address of instruction or data) is referenced now, it is likely to be referenced again shortly. Therefore, if we write recently accessed addresses in cache, the performance is greatly improved. In a computer program, one example of locality in time is loop. When a processor executes a loop, it generates the same instruction addresses over different iterations of the loop. Locality in space means that once an address is referenced, its adjacent addresses are likely to be referenced in the near future. To exploit locality in space, when a cache miss occurs, a chunk of consecutive addresses are transferred from main memory to the cache. An example of locality in space is an array. When a program processes an array, it usually accesses elements of the array sequentially. As such, when a cache miss occurs, in addition to the missed address, we read nearby addresses from memory and transfer them to the cache.

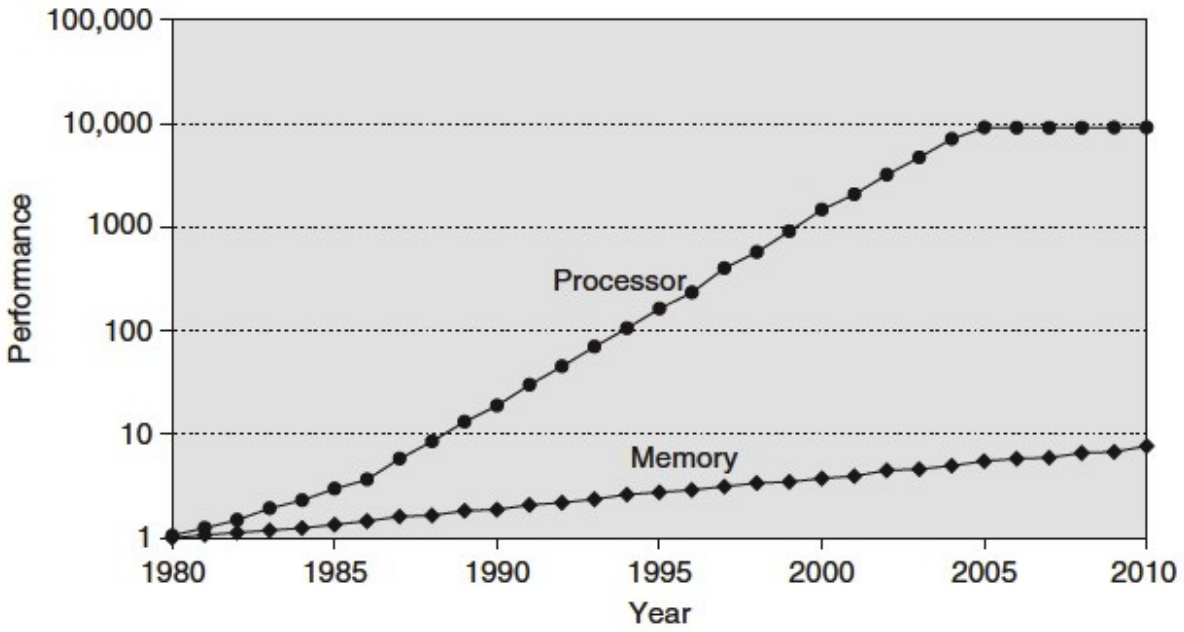


Figure 3-10: Memory gap [49]

In the following two subsections, we present two types of caches: direct-mapped cache and set-associative cache.

3.5.2 Direct-mapped cache

In direct mapped cache, each memory address is mapped to exactly one location in the cache. The address is divided into three fields: cache tag, cache index and byte select. The cache tag is used to determine if there is a cache hit or miss. The cache index is used to determine which cache row possibly contains the address. The byte select is used to select the byte within the cache block. To illustrate this, Figure 3-11 depicts a 2K-byte cache with 32-byte cache blocks.

Since cache block is 32-byte, byte select field requires 5-bit. For a total size of 2K byte direct-mapped cache with 32-byte cache block, there are a total of 64 rows (2048/32 =64). Therefore, the cache index field contains 6-bit. The rest of the address bits form the cache tag.

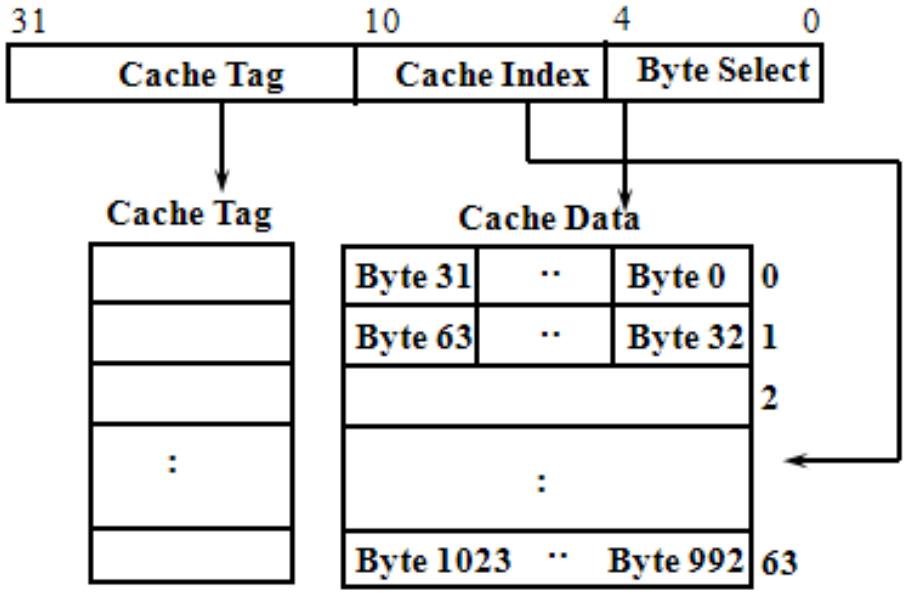


Figure 3-11: A 2k byte, 32 bit address, direct-mapped cache

Since cache is smaller than main memory, multiple addresses are mapped to the same cache row. We can determine which memory address is mapped to a cache block by comparing cache tag with tag field of memory address. For example, for the 2K-byte cache in Figure 3-11, address 6306 (14b'11 000101 00010) maps to byte number 2 (5b'00010, the

5 lowest bits in address) in row number 5 (6'b000101, the next 6 lowest bits in address). Address 4258 (14b'10 000101 00010) also maps to the same location in cache. However, the tag fields in the two addresses are different.

Notice that to exploit locality in space, the cache writes and reads the whole cache block/row at once.

3.5.3 Set-associative cache

One of the limitations of direct-mapped cache is conflict misses. Since each memory address is mapped to exactly one location in cache, the probability of aliasing is high. One way to reduce aliasing in direct-mapped cache is using multiple "ways" in the same cache row. This is the organization of a set-associative cache. In other words, a set-associative cache can be treated as multiple direct-mapped caches running in parallel. Figure 3-12 exemplifies the structure of a 2-way set-associative cache.

In Figure 3-12, way-0 and way-1 have the structure of a direct-mapped cache. During cache look-up, all the ways that cache index points to are accessed in parallel. The cache tags from all ways are compared with tag field of memory address in parallel. If one of the ways has a tag match, it is a cache hit and the corresponding data is passed through a multiplexor to the processor; otherwise, it is a miss. This structure greatly improves the cache hit rate relative to a direct-mapped cache since in the set-associative cache, there are multiple "ways" for an address, instead of just one.

The down side for a set-associative cache is power consumption. For cache look-up, "all" the ways are accessed in parallel. However, only the way that has the matching address provides data. Other ways end up wasting energy.

In the event of cache miss, the cache controller should decide which cache block should be replaced. There are several policies for replacement. The commonly used one is Least Recently Used (LRU) policy. In this policy, the least recently used way will be replaced with new address.

Set-associative cache is the target organization in this study.

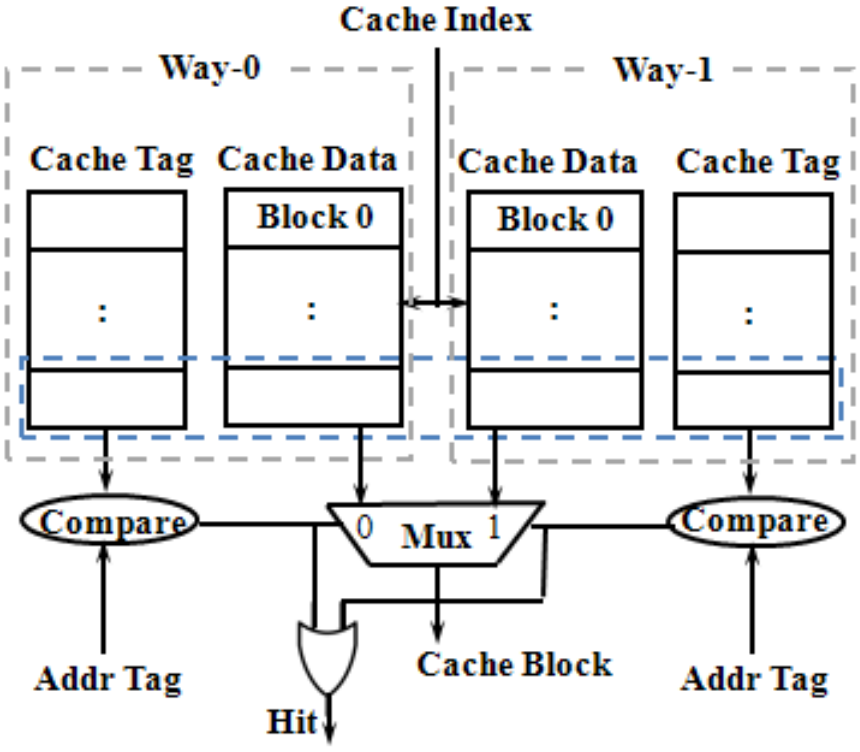


Figure 3-12: A 2-way set-associative cache

3.6 Commit

The final stage of a superscalar processor is commit. In this stage, all instructions are processed in order to implement the appearance of a sequential model. Also in this stage, a processor can determine whether a branch instruction is mispredicted. If misprediction has occurred, then speculative instructions should be removed from processor and instructions from the correct path should be fetched and executed. To commit an instruction, its result is written back to the register file or to the main memory and its entry is removed from the re-order buffer.

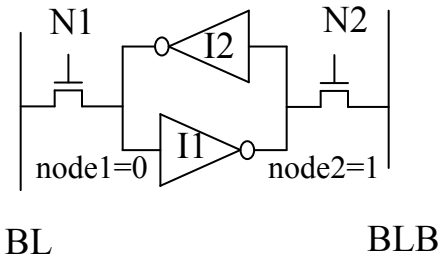
Section 4
Low Power Cache, Low Power Register File, and Critical Path Instructions

In this chapter, we cover the details of low power cache and register file. In section 4.1, we discuss basics of an SRAM cell, its operation and structure of a cache in circuit level. In section 4.2, we present the structure of the low power cache proposed in this study. In section 4.3, we explain the architecture of the low power register file. In section 4.4, we discuss critical path instructions.

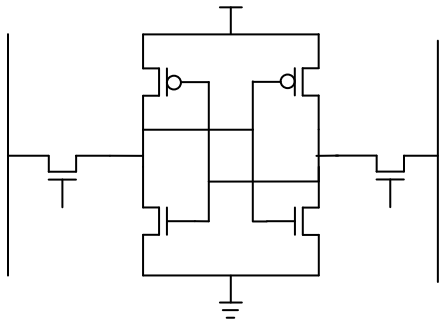
4.1 SRAM basics

4.1.1 Standard 6T SRAM cells

Figure 4-1a depicts the core of a 6T (6 Transistor) SRAM cell. The cell is composed of two inverters connected head to tail. The two inverters form a positive feedback loop to latch data. If node 1 stores "0", then the output of I1 is "1". Input "1" of I2 re-enforced "0" in node1. To read from the SRAM cell, there are 2 NMOS pass transistors on each side. When these two transistors are turned on, node 1 and node 2 can be written or read through BL and BLB, respectively. Figure 4-1b is the 6T SRAM cell in transistor level.



(a)



(b)

Figure 4-1: An SRAM cell

4.1.2 4T2R topology SRAM cells

There is another type of SRAM cell: 4T2R. In this topology, the two top PMOS transistors are replaced by two resistors as depicted in Figure 4-2.

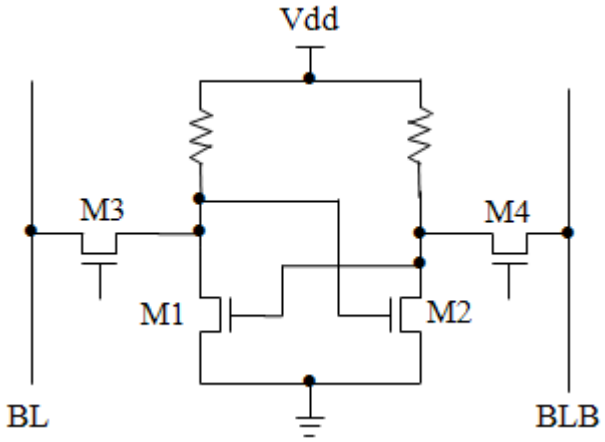


Figure 4-2: A 4T2R SRAM cell.

The advantage of this type of SRAM cell is that it uses only a smaller area which makes the SRAM core more compact. When the area of a chip is the concern, this type of topology can be exploited. Why does it use a smaller area than standard 6T type? This is because in the semiconductor processing, the PMOS and NMOS must be isolated from each other. If not, there would be P-N junction forward biased and this causes power to ground short

circuit. For a single well process with p type substrate, PMOS in Figure 4-1 must be built in n-well as depicted in Figure 4-3.

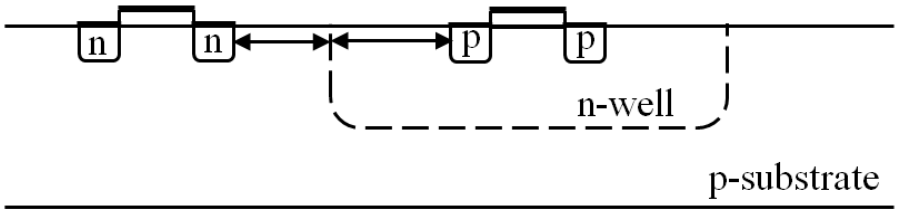


Figure 4-3: A profile of an NMOS and PMOS in silicon.

As indicated in Figure 4-3, in order to overcome manufacturing variation, there must be some separation between PMOS and the edge of n-well and between NMOS and the edge of n-well. This would cause the total area of 6T cell larger than 4T2R.

The down side of 4T2R topology is higher leakage current. Depending on value stored in SRAM, either M1 or M2 is on. This creates a leaky path between Vdd and ground. So, for this type of SRAM cell to work with smaller area, a high resistive material, usually the lighted doped poly-silicon, must be available in the semiconductor processes [30]. In this study, we choose 6T since its power consumption is less than 4T2R. In the next two sections, we explain how read and write operations work in 6T SRAM cells.

4.1.3 Read operation

The sizes of the transistors in an SRAM cell can not be arbitrarily chosen. They must follow basic transistor's drain current and gate-source voltage equations to guarantee proper operation of the SRAM cell. For the purpose of deriving the equations for the transistor sizes, the voltage levels in an SRAM cell at the beginning of Read operation are shown in Figure 4-4.

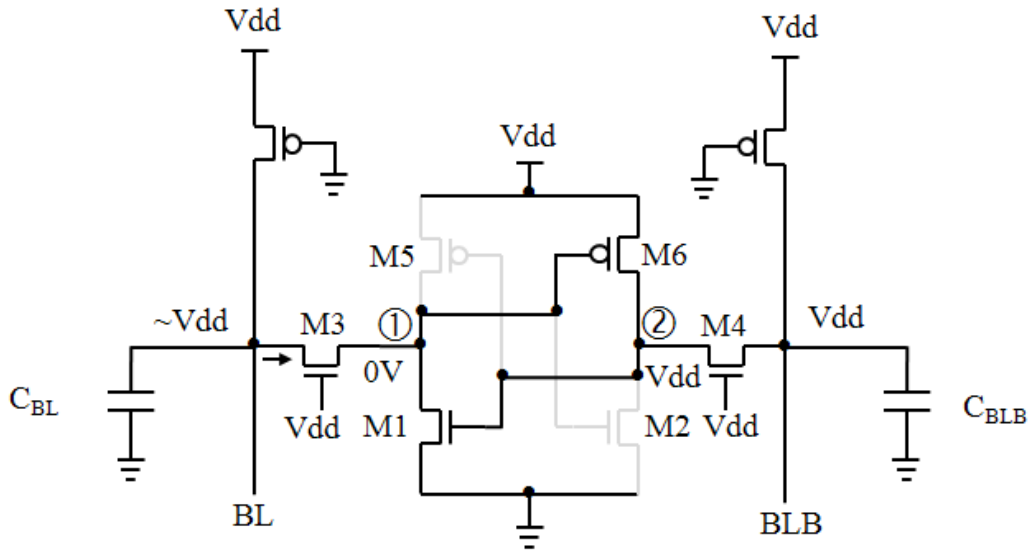


Figure 4-4: An SRAM cell in read operation [39].

We assume the SRAM cell in Figure 4-4 stores "0" (0V) in node 1 and stores "1" (Vdd) in node 2. Therefore, M2 and M5 are turned off (grayed out in the figure). Also the BL and BLB are pre-charged to Vdd by two PMOS transistors.

The pass transistors M3 and M4 are turned on by applying Vdd to their gates at the beginning of a read operation. Therefore, during the read operation, the voltage of node 1 would rise and the voltage of BL would be lowered. However, the voltage of node 1 can not exceed the threshold voltage of M2. If so, M2 would be turned on and the read operation is a destructive read (upsetting the stored data). To ensure this doesn't happen, M1 must be in triode as its Vds (drain to source voltage) is less than Vth (threshold voltage). For M3, its source (node 1) is ~0V while its drain (BL) is ~Vdd; therefore, it must be biased in active region.

The equations of an NMOS transistor biased in triode and active region are as follows:

$$I_d = \frac{k}{2}\frac{W}{L}\left(2(v_{gs} - v_{th})v_{ds} - {v_{ds}}^2\right) \qquad \text{Triode Region} \qquad (4\text{-}1)$$

$$I_d = \frac{k}{2}\frac{W}{L}(v_{gs} - v_{th})^2 \qquad \text{Active Region} \qquad (4\text{-}2)$$

The current of M1 and M3 must be equal because they are the only transistors in the discharging path. Therefore, with the above argument, we can write the following equation.

$$\frac{k_{n3}}{2}\left(\frac{W}{L}\right)_3 (v_{dd} - v_1 - v_{thn})^2 = \frac{K_{n1}}{2}\left(\frac{W}{L}\right)_1 (2(v_{dd} - v_{thn})v_1 - {v_1}^2)) \qquad (4\text{-}3)$$

At the boundary condition, $v_1 = v_{thn}$, we can further derive:

$$\frac{k_{n3}}{2}\left(\frac{W}{L}\right)_3 (v_{dd} - 2v_{thn})^2 = \frac{K_{n1}}{2}\left(\frac{W}{L}\right)_1 (2(v_{dd} - 1.5v_{thn})v_{thn}) \qquad (4\text{-}4)$$

$$\frac{k_{n,3}}{k_{n,1}} = \frac{\left({}^{W}/_{L}\right)_3}{\left({}^{W}/_{L}\right)_1} < \frac{2(V_{DD} - 1.5V_{thn})V_{thn}}{(V_{DD} - 2V_{thn})^2} \qquad (4\text{-}5)$$

The transistor sizing relationship between M3 and M1 is established in equation 4-5. In equation 4-5, W is the width and L is the channel length of the transistor. The subscript indicates a transistor in Figure 4-4. For examples, $(W/L)_3$ is the width and length of transistor M3 in Figure 4-4. V_{thn} is the threshold voltage of NMOS. In this example, we assume that node 1 stores "0" and node 2 stores "1". Since the circuit is symmetric, the same equation is valid for the case with opposite values.

4.1.4 Write operation

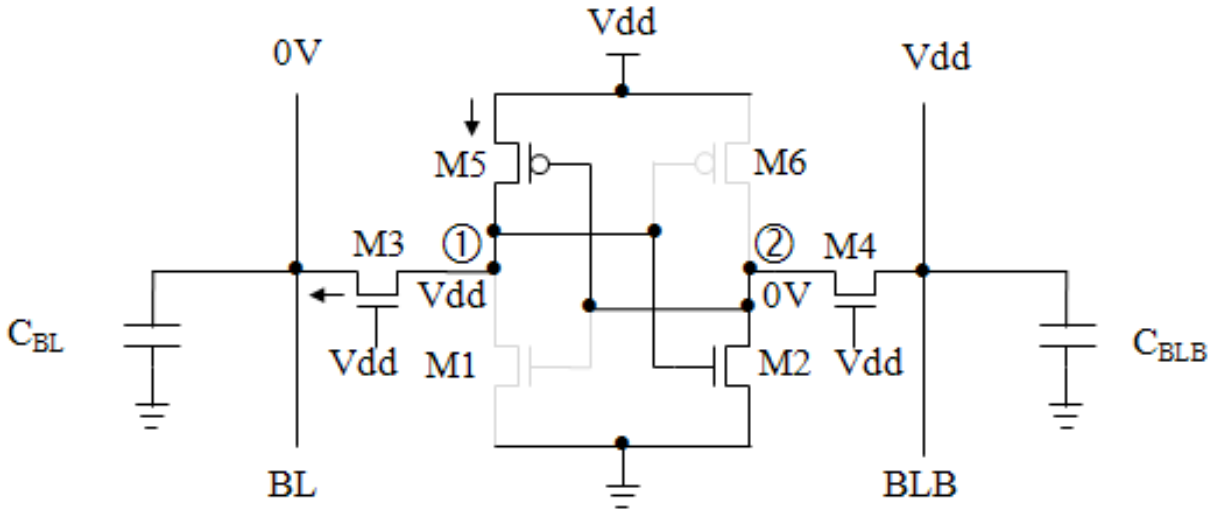


Figure 4-5: An SRAM cell for write operation [39].

Figure 4-5 shows an SRAM cell that stores "1" at node 1 and "0" at node 2. Therefore, M1 and M6 are turned off (grayed out in the figure). To write opposite data to this SRAM cell, BL should be 0V and BLB should be at Vdd. At the beginning of a Write operation, both M3 and M4 are turned on by applying Vdd to their gates.

At node 2, the situation is the same as read operation. BLB is connected to Vdd and node 2 is 0v. The equation 4-5 established for the read operation ensures that the voltage of node 2 does not exceed threshold voltage (Vth). So the new data is not written through this node.

At node 1, the voltage (Vdd) must be discharged due to the 0V on BL. When it reaches the threshold voltage of M2, M2 is cut off and node 2 is released from ground. Therefore, its voltage raises by the current from BLB. When voltage of node 2 exceeds the threshold voltage of M1, M1 turns on which also accelerates the discharge of node 1. Note that, at this moment, M5 is on. Therefore, it forms a short circuit from Vdd to ground which draws a higher current. However, this stops immediately when the voltage of node 2 exceeds (Vdd-

Vth). At this time, M5 is cut off. This stops the current draw in M5 and M1 and also helps node 1 to discharge.

The tipping point of a write operation is when the voltage of node 1 reaches the threshold voltage of M2. At this point, since the drain to source voltage of M3 is Vth, M3 must be in triode region. The drain to source voltage of M5 is Vdd-Vth. Therefore, it must be in active region. By combining equations 4-1 and 4-2 for M3 and M5 with the condition that their currents are equal, we can derive the following equation:

$$\frac{k_{ps}}{2}\left(\frac{W}{L}\right)_5 (0 - v_{dd} - v_{thp})^2 = \frac{K_{ns}}{2}\left(\frac{W}{L}\right)_3 (2(v_{dd} - v_{thn})v_1 - {v_1}^2)) \quad (4\text{-}6)$$

At the boundary condition, we can further derive,

$$\frac{k_{ps}}{2}\left(\frac{W}{L}\right)_5 (v_{dd} + v_{thp})^2 = \frac{K_{ns}}{2}\left(\frac{W}{L}\right)_3 (2(v_{dd} - 1.5v_{thn})v_{thn}) \quad (4\text{-}7)$$

$$\frac{(W/L)_5}{(W/L)_3} < \frac{\mu_n}{\mu_p}\frac{2(V_{DD} - 1.5V_{thn})V_{thn}}{(V_{DD} + V_{thp})^2} \quad (4\text{-}8)$$

where, $k_{ps} = \mu_p \times k$ and $k_{ns} = \mu_n \times k$.

In this equation, μ_n is the electron mobility in NMOS and μ_p is the hole mobility in PMOS. Equation 4-8 shows the transistor sizing relationship between M3 and M5.

4.1.5 Transistor sizing constraints

Some literatures suggested that the SRAM sizing constraints should follow $K_{n,pd} > K_{access} > K_{p,pu}$ [40] (Figure 4-6). In our simulations, we first follow this constraint and adjust transistor sizes until we can successfully write and read at our target low supply voltage: 0.7v.

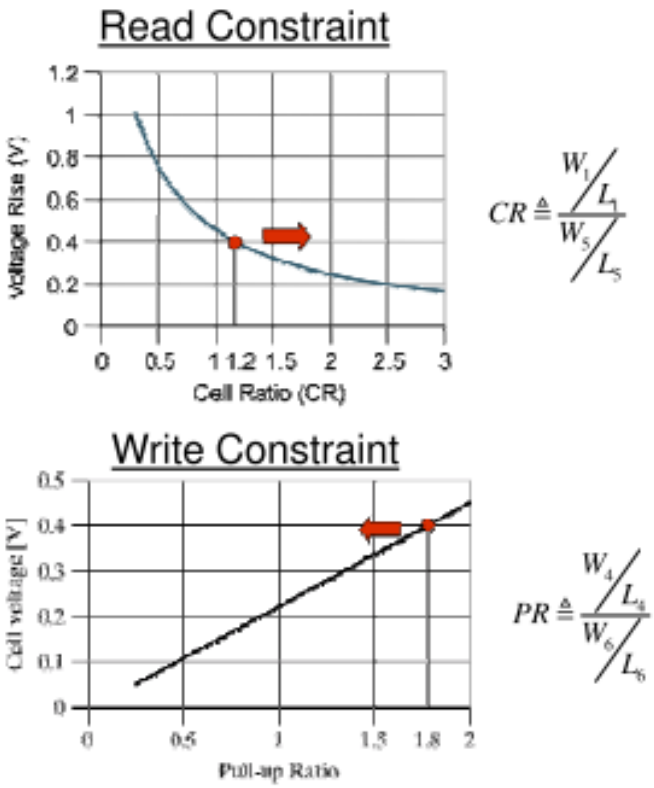


Figure 4-6: SRAM sizing constraints [40].

Figure 4-7 shows the schematic of our final SRAM cell design. Note that we use 32nm transistor model from PTM (Predictive Technology Model) provided by Arizona State University [4]. It only provides the transistor models without documentation on whether it is a single well, twin well or a triple well technology. It is common to have more than one well in an advanced technology such as 32nm, however. In such an advanced technology, feature sizes are smaller, transistors are closer to each other, and switching is faster. So, it is essential to use transistor isolation to prevent substrate noise. We assume that the technology we have in this study is a twin well technology which allows body of a transistor tie to its source.

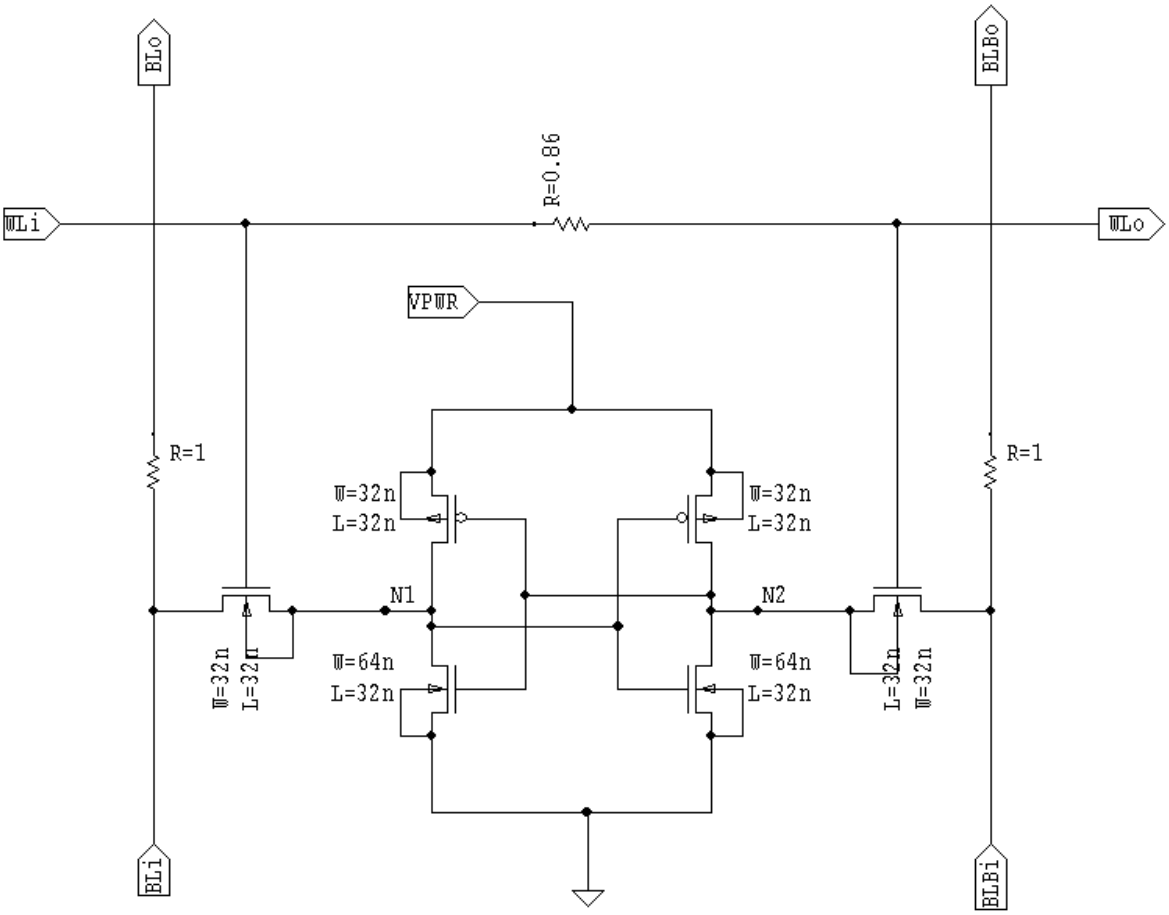


Figure 4-7: Final SRAM cell in our design.

4.1.6 SRAM columns

Figure 4-8 depicts a typical SRAM column which is comprised of various functional blocks in addition to 6T SRAM cells, such as pre-charges, decoders, sense amps, latches and write drivers. In the following subsections, we explain these units one-by-one and then present our final design after fine tuning.

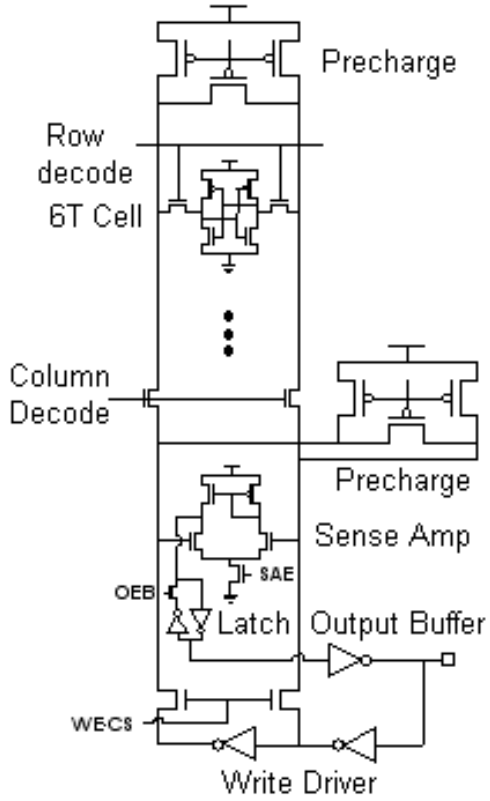


Figure 4-8: A typical SRAM column structure.

4.1.6.1 Pre-charge

The pre-charge transistors are used to pre-charge BL and BLB to a known voltage, normally Vdd. This should be done while SRAM is not active. During write or read operation, the pre-charge must be disabled to allow normal operation of the SRAM cell. Figure 4-9 shows our design. We selected large transistors (W/L=512nm/32nm) to charge BL and BLB, which are highly capacitive nodes, in a short period of time.

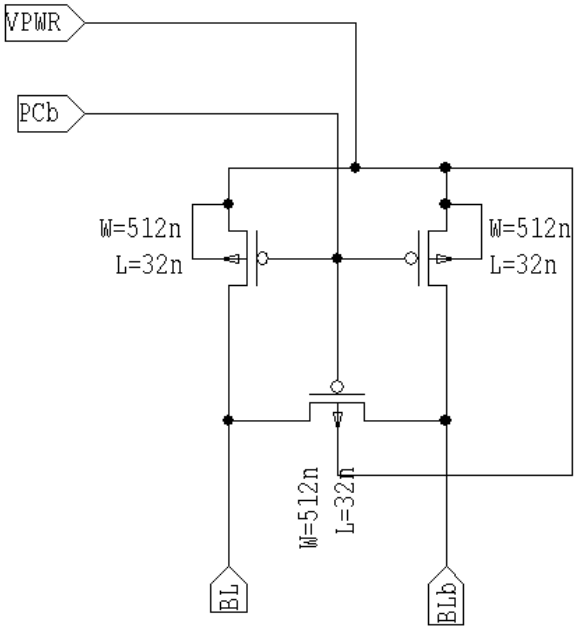


Figure 4-9: Circuit of pre-charge.

There is a PMOS in pre-charge circuit connected to BL and BLB. This PMOS is called equalizer. It is used to equalize the voltages on BL and BLB. This is to overcome the process variation between transistors. Without the equalizer, there must be a voltage offset between BL and BLB. This offset voltage may affect signal sensing if its polarity is opposite to the signal being sensed. If this happens, the net signal amplitude is weakened (Vnet=Vsignal-Voffset) and it would be difficult to sense it correctly. Note that due to the highly capacitive nature of BL and BLB, normally an SRAM column has a pre-charge on both top and bottom of the column. All the three transistors are controlled by Pre-Charge Bar (PCB) signal.

We tried transistors with different sizes. We selected minimum sizes so that when combining blocks in Figure 4-9, 4-11, 4-14 and 4-17, the SRAM cells are able to read and write properly.

4.1.6.2 Column decoder

The purpose of a column decoder is to allow different SRAM columns to share the same analog block such as sensing amplifiers, and write drivers (Figure 4-10).

In our cache design, we use two analog blocks supplied by different voltages. These column decoders on the other hand select which analog block can write and read the SRAM column.

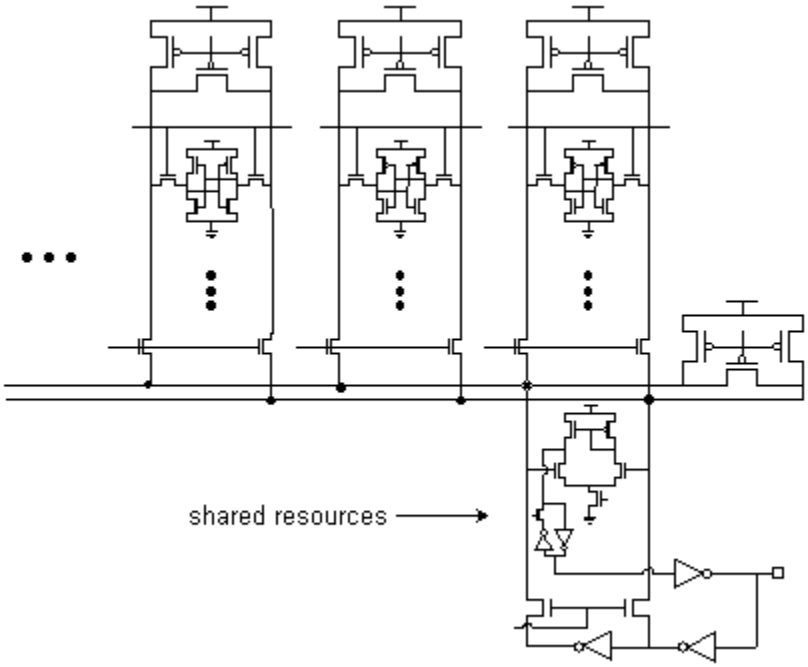


Figure 4-10: SRAM columns sharing same analog resources.

Figure 4-11 shows the transistor sizes of column decoder in our design. They are made of large transistors to reduce voltage drop in BL and BLB.

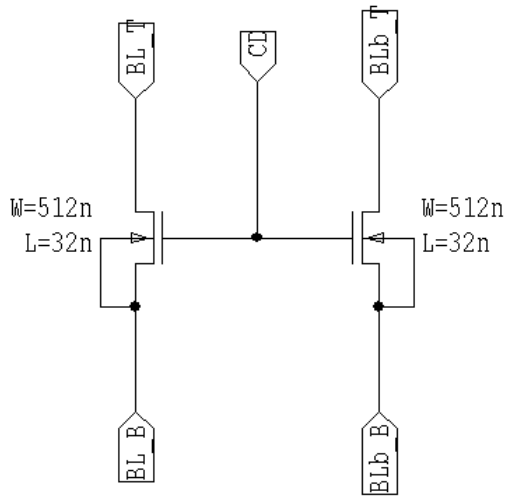


Figure 4-11: Column decoder (CD).

4.1.6.3 Write driver

The write driver is implemented using two inverters. These two inverters create "1" and "0" signals on BL and BLB. They are big in size because the inverter whose output is zero needs to completely discharge BL or BLB in order to flip the data stored in an SRAM cell. Those two nodes (BL and BLB) are pulled to Vdd by pre-charge (Figure 4-9) during the inactive period of time (no write or read). In addition to these two inverters, there must be two NMOS transistors gating the write driver so that the driver's outputs do not affect the read operation. These two gating transistors are controlled by write enable signal (WE). They also are big in size to limit voltage drop across drain and source. Figure 4-12 is the schematic of the design.

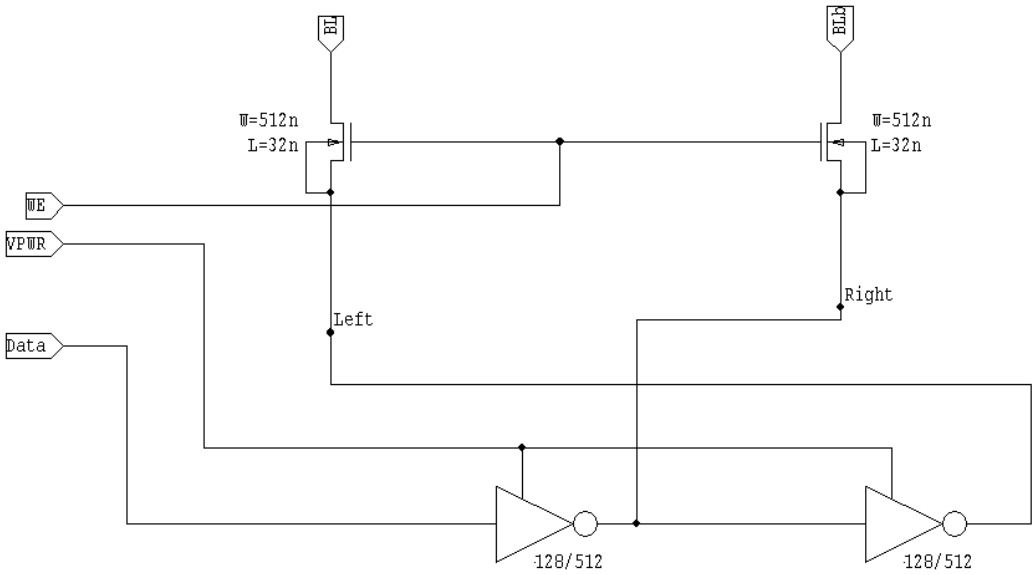


Figure 4-12: Write driver.

4.1.6.4 Sense amplifier

Sense amplifier is the heart of an SRAM. At the beginning of a read operation, both BL and BLB are pre-charged to Vdd. When the SRAM cell is selected, either BL or BLB is discharged by the node storing "0". Sense amplifier must be able to sense this voltage level split between BL and BLB. There are many types of sense amplifiers that can be used and they are discussed in the literature [10, 11, 38, 39, 40]. We investigated 32nm PTM model and found this technology has prominent short-channel effect and the gain of a single stage differential amplifier may not be enough to amplify the signal for latch. Therefore, we decided to use two-stage dual amplifier topology as depicted in Figure 4-13.

In Figure 4-13, A1 senses voltage difference between BLB (+) and BL (-) while A2 senses difference between BL (+) and BLB (-). A1 and A2 double the voltage difference of their inputs. A3 then further amplifies the output of A1 and A2 and sends the signal to the next stage: latch.

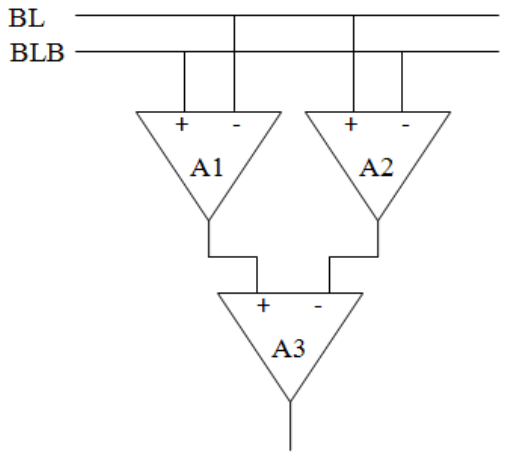
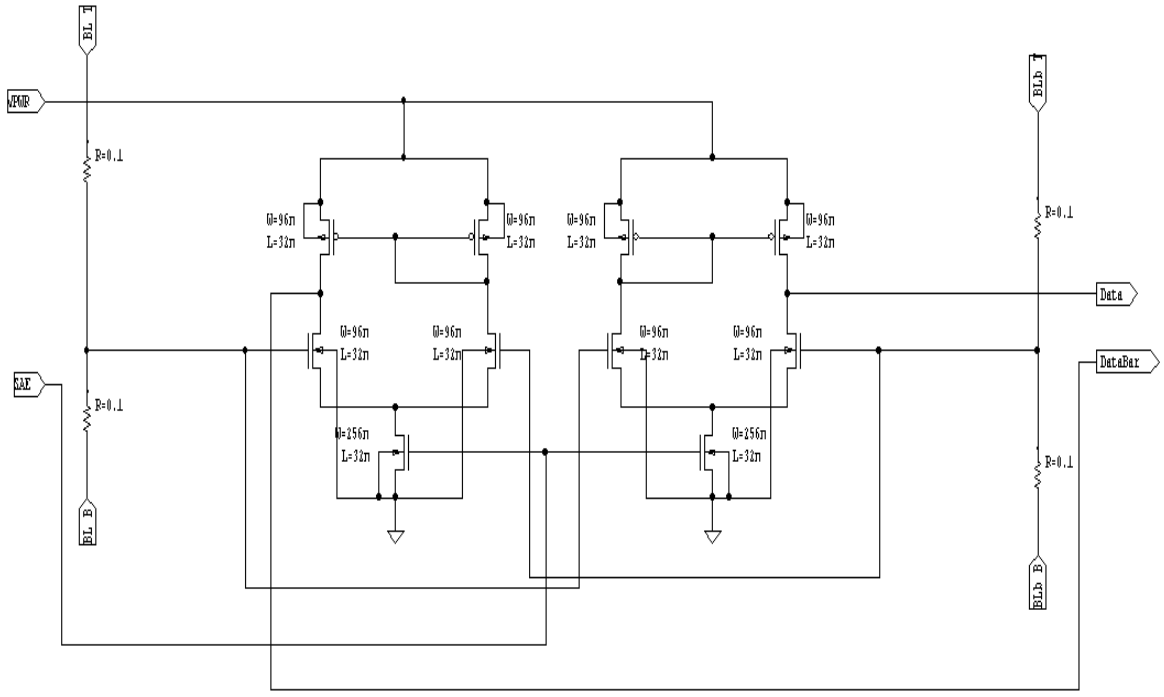
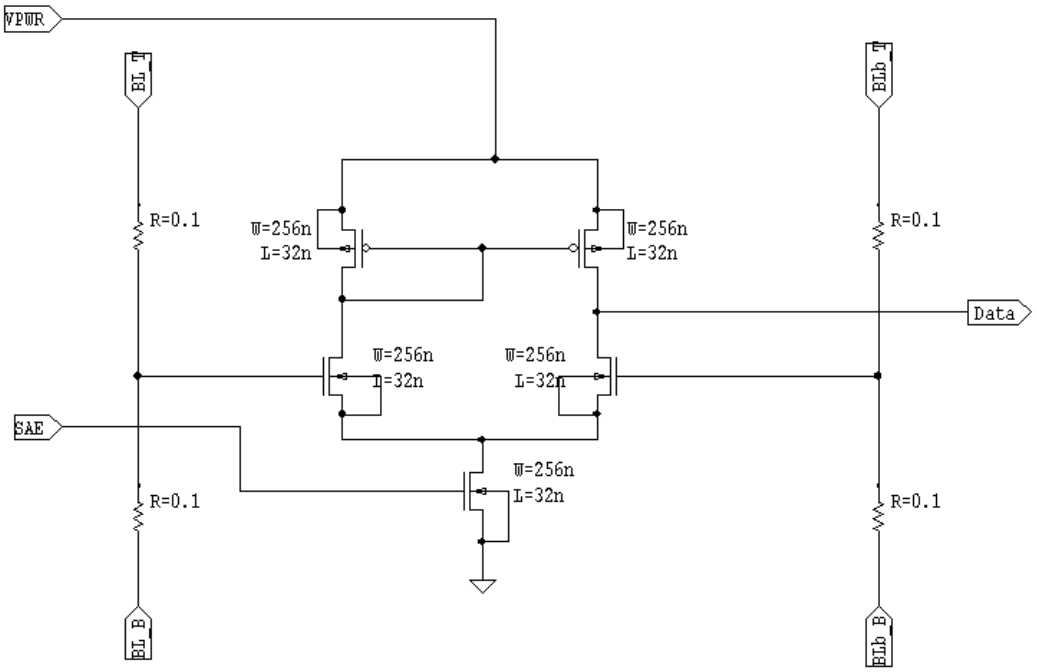


Figure 4-13: 2-level sense amplifier.

Figure 4-14a shows the transistor level schematic of the first level sense amplifier comprised of A1 and A2 [18]. Figure 4-14b shows the schematic of the final stage [18]. The transistors' sizes are determined by fine tuning through HSPICE simulation.



(a)



(b)

Figure 4-14: (a) First stage (b) Final stage of sense amplifier.

The tail NMOS transistors at the bottom of all the stages of sense amplifier are controlled by SAE (Sense Amplifier Enable) signal. Sense amplifiers are needed only during the read cycles. Therefore, during the write cycles or when the SRAM is inactive, they should be turned off by disabling SAE signal.

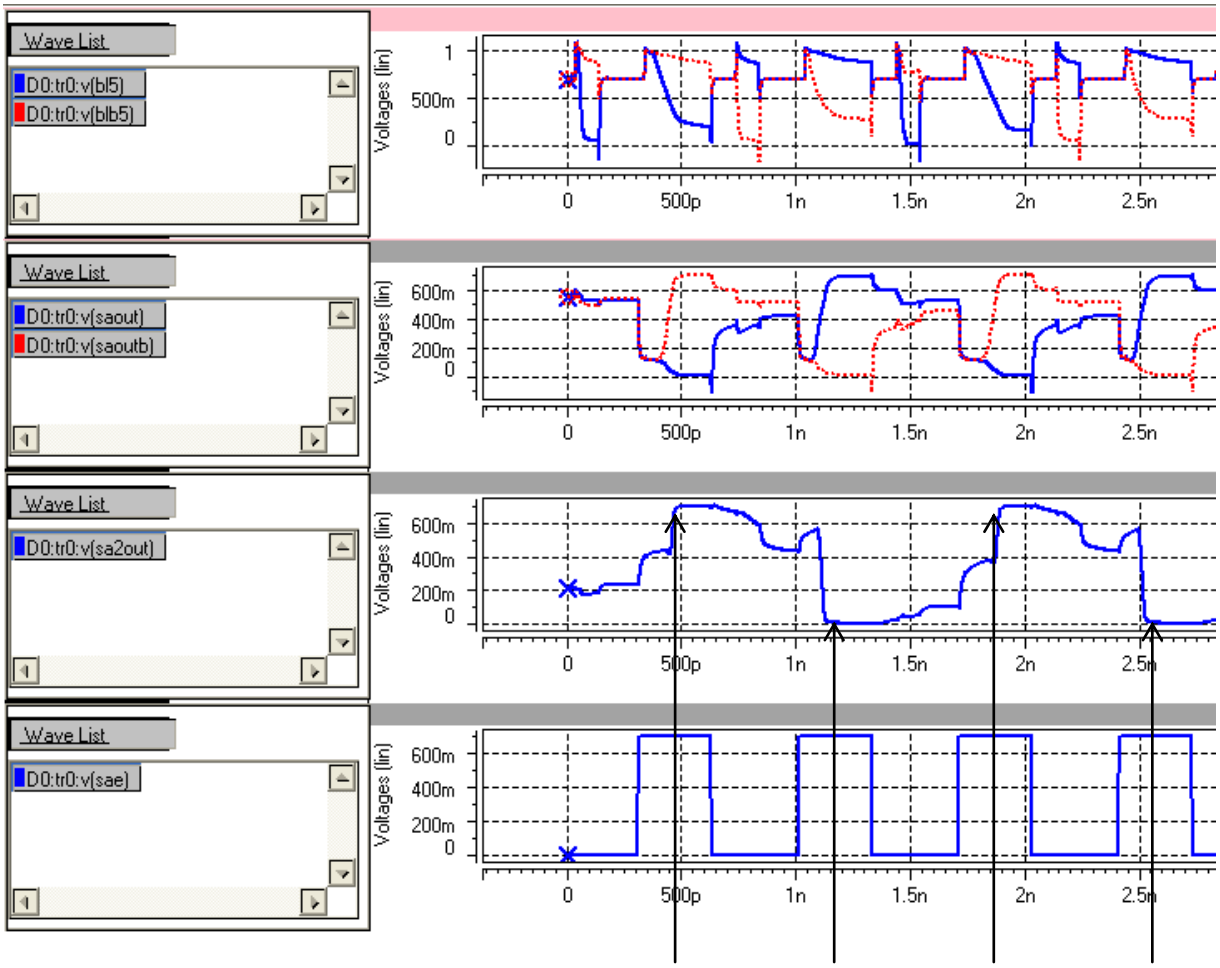


Figure 4-15: Waveforms of sense amplifier block.

Figure 4-15 shows the waveforms of input and output of sense amplifier. The top panel is the input signals on BL (solid line) and BLB (dashed line). The second panel is the waveforms of output of the first 2 sense amplifiers in the first stage; saout (solid line) and saoutb (dashed line). The third panel is the output of the final stage sense amplifier. The bottom panel is the SAE signal. To ensure proper write and read operations, we must alternatively write “1”, read “1”, write “0” and read “0”. The four arrows in Figure 4-15 point to read “1”, read “0”, read “1” and read “0”. Since we are able to read them properly, the write operations are implemented correctly.

4.1.6.5 Latch

The purpose of a latch is to retain the data at the output port while the SRAM can be in write cycles or inactive cycles. Figure 4-16 shows the topology of a gated-latch. When SAE is asserted, I2 is disabled (Figure 4-16b). This allows the data from the final stage of sense amplifier to drive I1 without interference with I2. When the read cycle ends, SAE signal is de-asserted and I2 is enabled. As such, the data is latched (Figure 4-16a). Figure 4-17 shows the schematic of this gated latch in transistor level after fine-tuning with HSPICE.

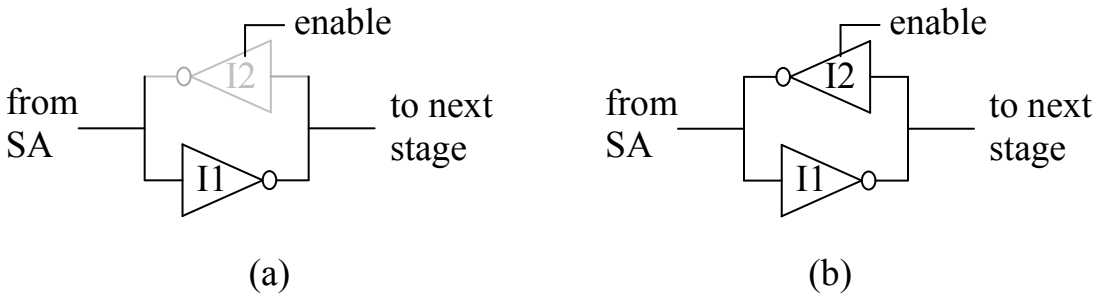


Figure 4-16: Latch states when (a) SAE is de-asserted (b) SAE is asserted.

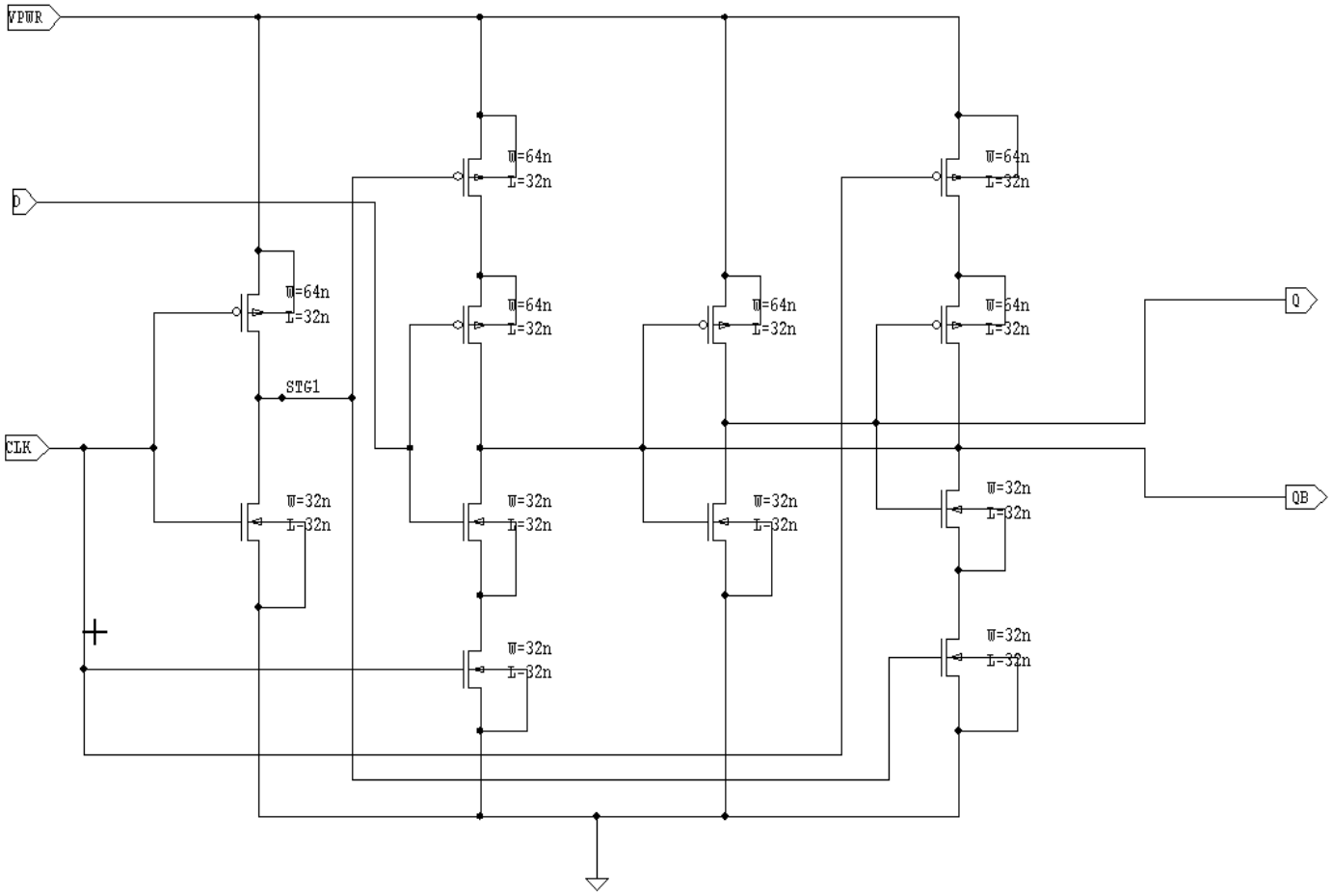


Figure 4-17: Schematic of gated data latch.

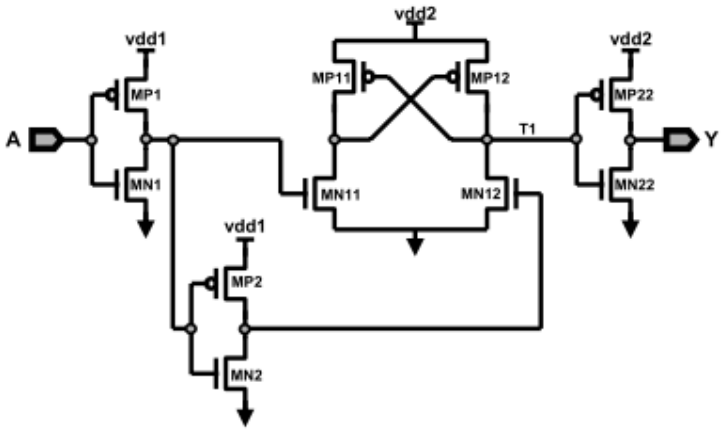
4.1.6.6 Level shifter

Since we need to have two supply voltages in our low power cache, there must be a level shifter to shift the signal from low level to high level or vice versa.

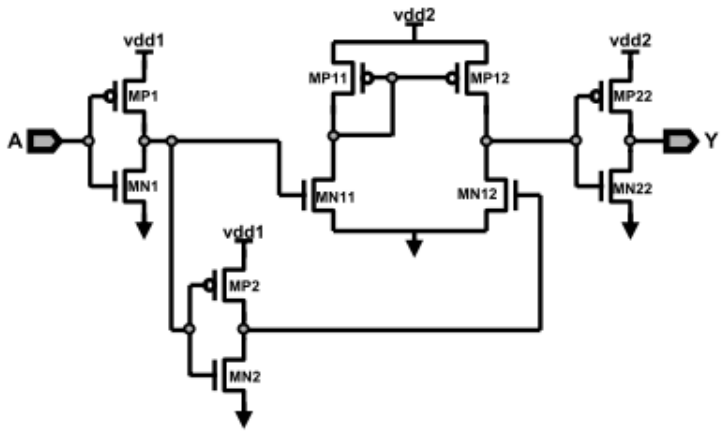
There are many types of level shifters. Figure 4-18 shows two popular level shifters [41]. In both level shifters, low voltage devices drive the NMOS of high voltage devices and let PMOS of the high voltage devices pull themselves to higher Vdd. The circuit in Figure 4-18a has interlocked (latch) PMOS while Figure 4-18b has current mirror type of PMOS transistors. One can imagine that the circuit in Figure 4-18b has a slight advantage over the other circuit in term of speed while the circuit in Figure 4-18a provides a differential output.

In our design, we choose the topology in Figure 4-18a to minimize current consumption. Figure 4-19 shows the schematic of our design. Note that in Figure 4-19, the two NMOS transistors at the bottom are bigger. This is because in this type of topology, NMOS is the main driving force to pull one of the interlocked PMOS gate voltages to 0v which turns on the PMOS. Once turned on, the PMOS pulls another node to high and flips the state. This requires a large NMOS to initiate the state change.

The transistor sizes are chosen to be minimum size to keep power consumption low.



(a)



(b)

Figure 4-18: Conventional level shifters (a) latch-type (b) current-mirror type [41].

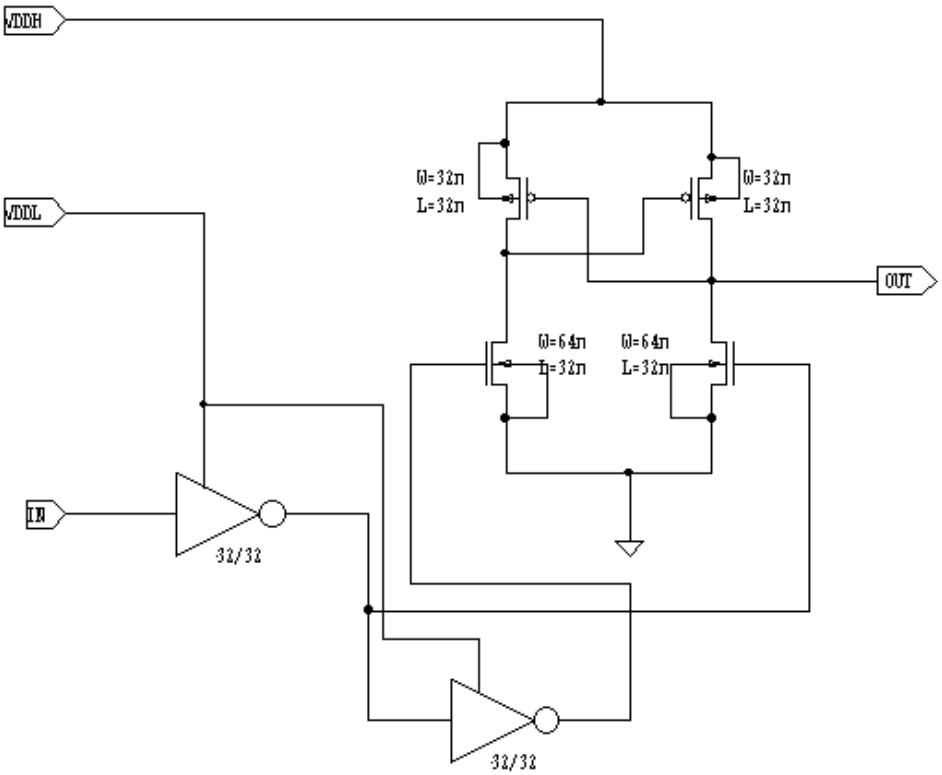


Figure 4-19: Schematic of our design.

4.1.6.7 Row decoder

The row decoder is used to select a specific row in SRAM cell array by turning on the two pass transistors M3 and M4 in Figures 4-2 and 4-4. There are many ways to implement the row decoder. In this study, we choose the conventional static type of decoders. The static decoder uses logical AND gates to select cache rows. For instance, if a row should be selected when address = 23 which is 10111 in binary representation, then the decoder calculates bit-wise AND of $A4, \overline{A3}, A2, A1, A0$ signals.

The AND gates are a combination of NAND and NOT gates. The NAND gates normally have only two or three inputs, even though conceptually a logic gate can have more than just three inputs. This is because each input controls one NMOS in the pull down network in NAND gates. Having more NMOS transistors in the network slows down the gate.

Figure 4-20 shows the pull down network of a two input NAND gate with marked voltage levels. MN1 gets all the desired voltage levels such as Vsb1=0v and Vgs1=Vdd. However, the voltage levels at MN2 are weaker. Vgs2 = Vdd-Vds1 is less than desired Vdd and Vsb2 is not 0v which increases MN2's threshold voltage. One can see that with a three inputs NAND gate, its MN3 will limit its speed. More than three inputs are not practical in a design. If there is a complicated combinatorial unit, it must be broken down to two or three input logic gates.

We selected minimum size transistors to keep power consumption low.

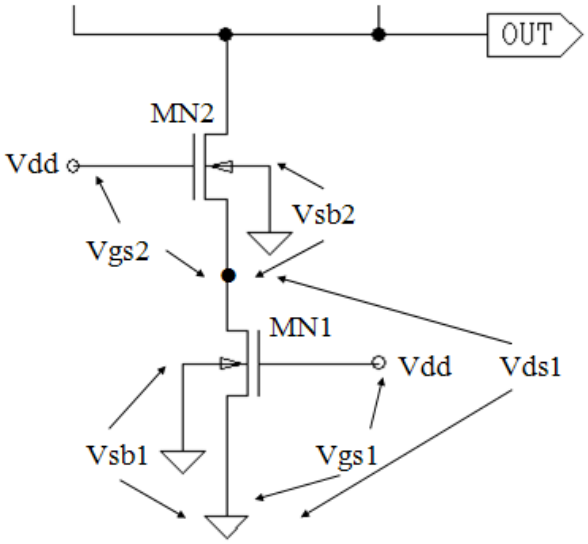


Figure 4-20: Pull down network of a two input NAND gate.

4.1.6.8 Tag comparison

In a cache, the tag filed of input address is compared with tag field of a cache row to determine cache hit/miss. Comparison can be implemented using bit-wise XOR (eXclusive OR). Figure 4-21 is the truth table of XOR gate and its symbol. Figure 4-22 shows the implementation of an XOR using NAND gates.

Out	In1	In2
0	0	0
1	0	1
1	1	0
0	1	1

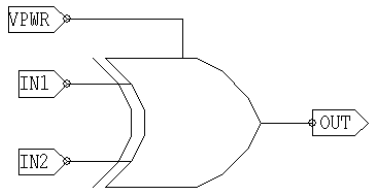


Figure 4-21: Truth table of XOR gate and its symbol.

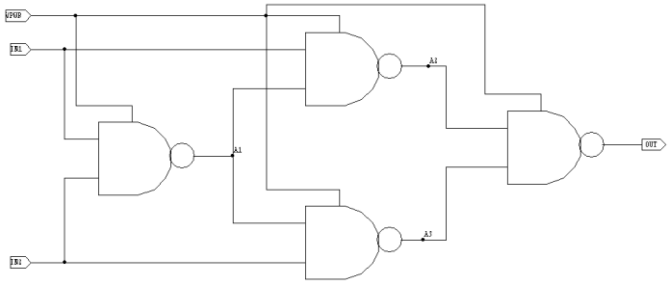


Figure 4-22: Schematic of XOR implementation.

4.2 Low power cache

A conventional cache is powered by a single supply voltage. In this study, we use SRAM cells with two supply voltages to reduce power consumption (Figure 4-23). One supply voltage is 0.9v which is the nominal voltage of 32nm Predictive Technology Model (PTM) provided by Arizona State University [4]. The other one is 0.7v. This voltage is the lowest possible voltage in which still cache reads/writes data properly with only one cycle of penalty. Figure 4-23 shows the overall structure of the low-power cache.

The main reason that we selected 0.7v for low power voltage is that our simulations show we only need to sacrifice one cycle in performance. In other words, 0.7v is the lowest voltage with only one cycle penalty. Further reduction of voltage leads to performance penalty of at least two cycles.

The new cache is organized as high- and low-voltage super-sets. Each super-set is composed of a number of cache rows and each row is composed of a number of cache blocks. All SRAM-cells in a super-set are either connected to 0.9v or 0.7v. The low power cache depicted in Figure 4-23 is a 64K Byte 4-way set-associative cache with 64-byte cache blocks. Therefore, there are 256 rows in this cache (64K÷64÷4=256). Assuming that size of super-set is 32 rows, there are 8 super-sets in this cache.

In order to write and read the cells at the same voltage level, there are two sets of analog blocks for each cache column. One is powered by high Vdd (0.9v) and the other is powered by low Vdd (0.7v). The analog blocks include pre-charge, sense amplifiers, write drivers, latches, etc. When the processor accesses a row in a super-set which is powered by high Vdd, the analog block turns on and writes or reads the row. While the high Vdd analog block is being accessed, the low Vdd analog blocks have no activity and so only consume static power which is significantly lower than the dynamic power.

In Figure 4-23, all the super-sets and analog blocks powered by high Vdd are in red color while all the super-sets and analog blocks powered by low Vdd are in blue color.

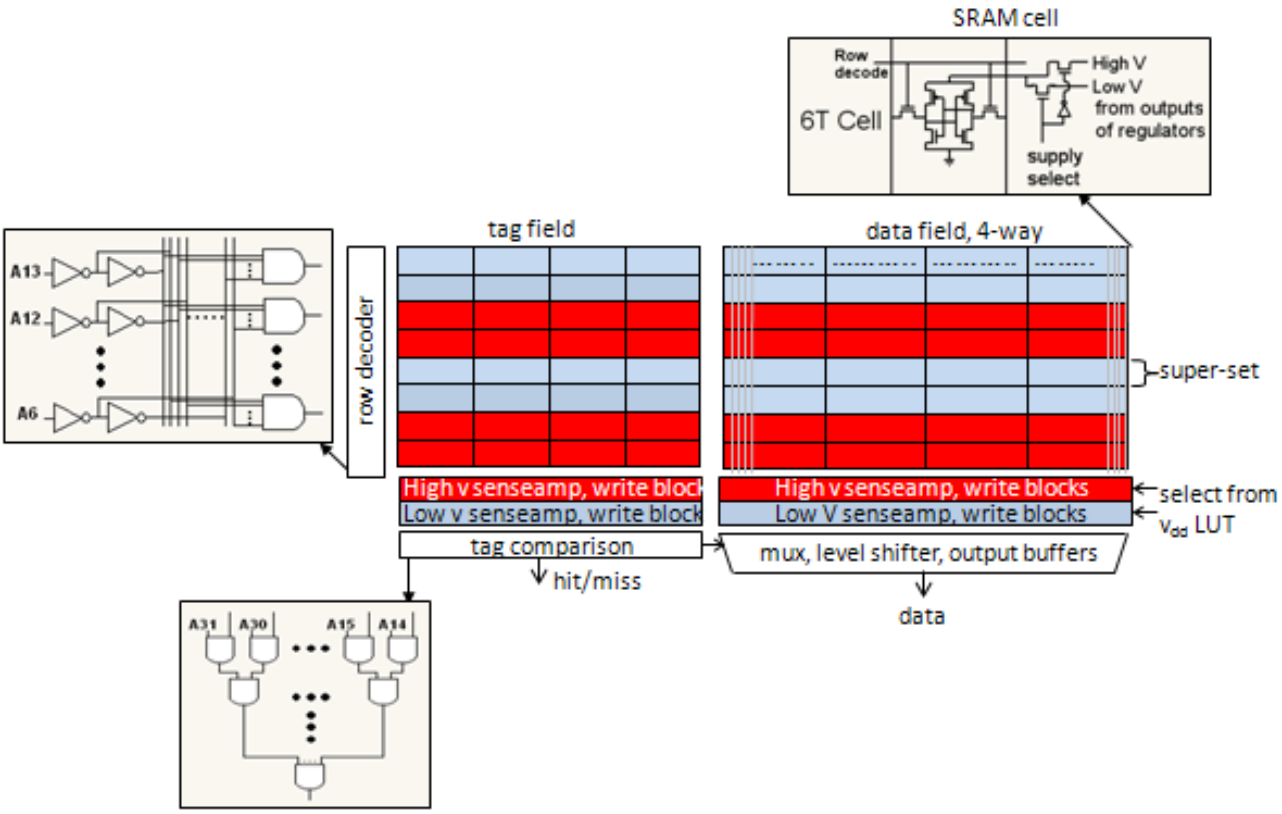


Figure 4-23: Low power cache design.

4.3 Low power register file

The organization of a register file is simpler than a cache. A register file does not have tag field and also does not need tag comparison. Figure 4-24 depicts our low power register file which has 32 registers. The size of each register is 64-bit.

Similar to the low-power cache, the registers are also grouped by super-sets. When writing or reading the low Vdd super-set registers, the analog blocks powered by low voltage is turned on to perform the task. When writing or reading the high Vdd super-set registers, the analog blocks powered by high voltage is turned on to perform the task.

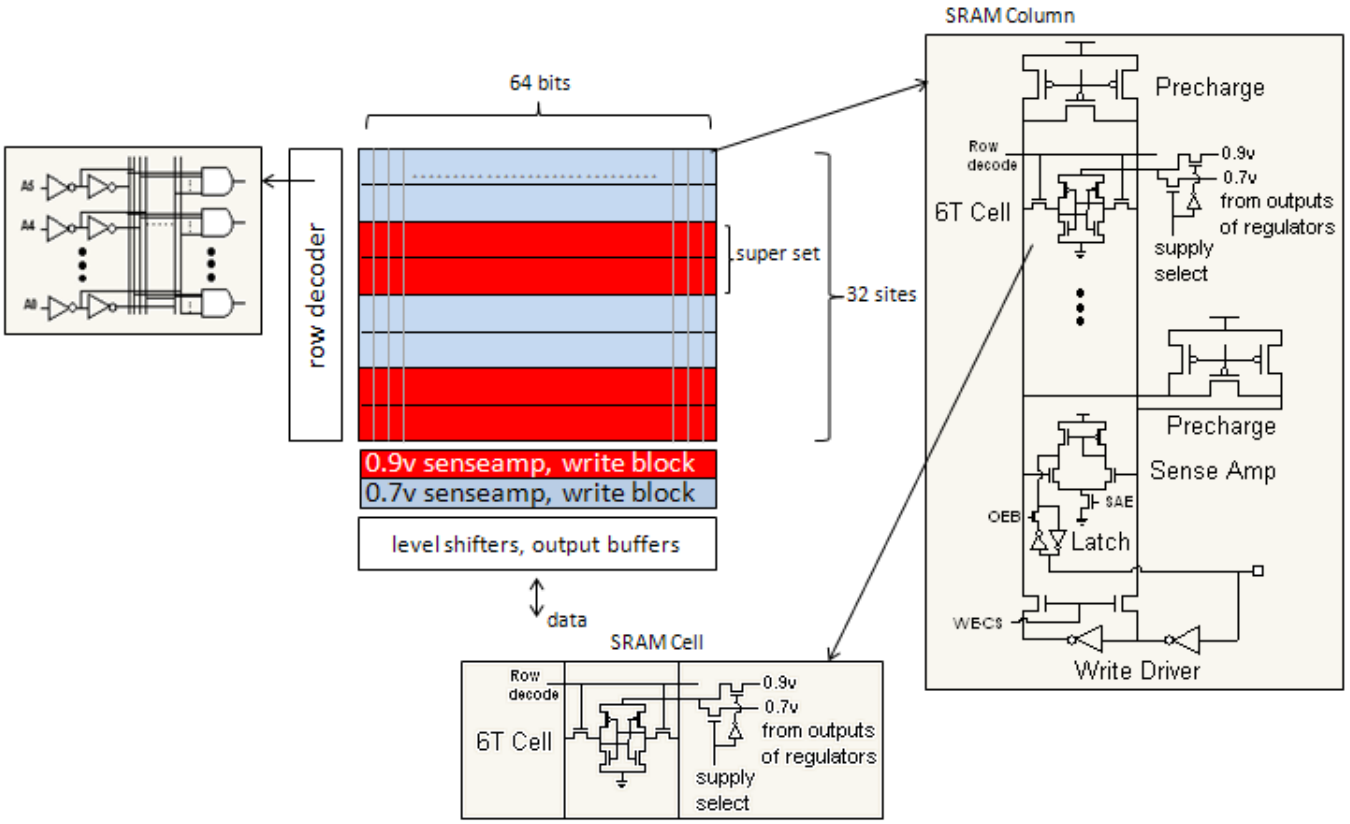


Figure 4-24: Low power register file.

4.4 Critical path instruction prediction

As described in section 3.2, modern out-of-order executing processors remove false data dependences such as Write after Write (WAW) and Write after Read (WAR) by register renaming. However, true dependences still remain in programs which may result in RAW (Read after Write) hazards. Chain of dependent instructions form critical paths and limit the performance of processors. Due to dependency, critical instruction should execute sequentially. So, to run programs fast, it is essential to execute critical instructions as quickly as possible. On the other side, execution of instructions that are not critical can be postponed without affecting performance. However, in contemporary processors, non-critical instructions spend as much power as critical instructions.

We can reduce power consumption of processors by dividing our resources into low/high power units. Low power units use low Vdd and are slow. On the other side, high power units use high Vdd and are fast. We send critical instructions to the fast units to preserve performance. On the other side, we send non-critical instructions to slow units to save power. However, in order to bring about all of these benefits, a mechanism is required to predict critical path instructions.

Figure 4-25 is a piece of code fragment showing data dependences. Instructions 1, 4, 6, 7, and 8 are dependent as one instruction waits for the result of another instruction. When this piece of code is dispatched to a re-order buffer as described in section 3.2, instructions 2, 3, and 5 would be issued earlier than other instructions because their input operands become ready sooner. Therefore, one heuristic for identifying critical path instructions is to look at the entries in the head of re-order buffer. If an instruction reaches to the head of re-order buffer and still is not ready, then it is an indication that the instruction is on the critical path.

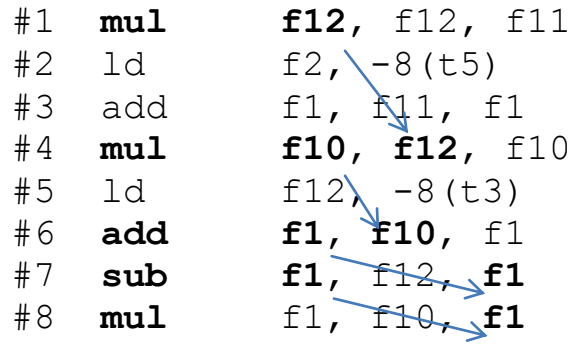


Figure 4-25: A code fragment of showing data dependence.

E. Tune *et al*. [3] proposed a number of methods to identify critical path instructions dynamically such as QOLD (Oldest Instruction in Queue), ALOLD (Oldest in Active List), QOLDDEP (DEPendence with OLDest instruction in Queue), QCONS (Most CONSumers in Queue), and FREED3 (FREED up at least 3 instructions in queue). Among those methods, QOLD and ALOLD perform better than the other types of policies.

In QOLD, we examine the instructions in the re-order buffer. If instructions reach to the head of re-order buffer but still can not be issued for execution, we consider them as critical

instructions. This is to utilize the pattern of a Superscalar processor described in section 3. In a Superscalar processor, to exploit instruction level parallelism, the instructions with their source operands ready are issued to execute regardless other instructions (out of order issue). This is because in dispatch stage, false data dependences such as WAW or WAR are removed by register renaming. When instructions queued in re-order buffer, there is only true data dependence Read after Write (RAW) left. Therefore, when an instruction's source operands are ready in the re-order buffer, RAW hazard is no longer existed and the instruction can be issued regardless other instructions, provided there is enough hardware resource, i.e. no structure hazard. Upon completion, an instruction is removed from re-order buffer by sliding down the head pointer of the buffer. Therefore, when an instruction gets to the head of re-order buffer and still is unable to issue, it is waiting for older instructions to provide its source operands. We deem this instruction is in the critical path.

In QCONS, we inspect each instruction in the write back stage and walk through its output dependent list and count its number of dependent instructions (consumers). If the number of dependent instructions exceeds a pre-defined threshold, we deem this instruction as a critical instruction. An instruction with many dependent instructions is critical since its outcome is needed for exertion of other instructions.

Note that QCONS policy has nothing to do with re-order buffer; therefore, it does not take into account the waiting time of an instruction and can easily result in misleading short term dependence paths as described in [3]. For instance, an ADD instruction may have multiple dependent instructions which exceed the pre-defined threshold and we deem it as critical. However, the producers of this ADD instruction's source operands come from non-critical path and when ADD is en-queued in re-order buffer, the source operands are always ready. This ADD instruction is not in the actual critical path because it won't cause any waiting or slow-down. Therefore, one can image why this technique is less efficient than QOLD policy.

FREED is a subset of QCONS. When an instruction is executed, it feeds its results to its dependent instructions. Some of those dependent instructions then have all their source operands ready but some of them may still need to wait for other source operands. For those

dependent instructions ready to issue, we say they are “freed” to en-queue. In this policy, we inspect each instruction in write back stage (when an instruction is execution), and walk through its output dependent list and count the number of instructions that are ready to en-queue after input dependence is satisfied. If the number of freed instructions exceeds a pre-defined threshold, we treat this instruction critical. FREED policy also would include short term dependence paths as QCONS.

Section 5 Methodology and Results

In this chapter, we report the impact of new cache and register file on power and performance. In Section 5.1, we explain how we use critical path instructions in the context of low-power cache. We also discuss the experimental framework used to evaluate the new cache. In Section 5.2, we cover critical path instruction and simulation results for low power register file.

5.1 Low power cache

5.1.1 Critical instruction policy (QOLD)

As described in section 4.3, critical instructions tend to accumulate at the head of re-order buffer. To detect critical instructions, we inspect the first N instructions in the head of Load/Store Queue (LSQ) each cycle. N is a predefined number which is set once and is used for the lifetime of processor. If any of these N instructions is "not ready" to issue, it is marked as critical load/store instructions.

We track number of accesses to a cache super-set by critical load/store instructions to determine the voltage of the super-set. Each time that a super-set (which is composed of a number of cache rows) is accessed by a critical load/store instruction, we increment a counter. We evaluate the counters after M instructions are committed, where M is called criticality interval and is a pre-determined parameter. If the counter exceeds a pre-determined threshold, all SRAM cells in the super-set are assigned to high supply voltage to preserve performance; otherwise, we use low supply voltage to save power. It is important to note that all cache cells within a set have the same supply voltage even if some of them are accessed by critical and some by non-critical instructions. After adjusting the supply voltage of the super-set, all the criticality counters are reset to zero for the next cache criticality interval (M). With this organization, the cache can dynamically increase voltage of super-sets to preserve performance or reduce the voltage of super-sets to save power.

In this study, in addition to QOLD policy, we also ran other policies such as QCONS and FREED. The methodology and benchmark results of QCONS are listed and discussed in section 7.1 and that of FREED are in section 7.2.

5.1.2 HSPICE simulations and results

In this section, we explain how we model cache in HSPICE. A cache access is composed of three major phases: 1) decoder which selects a cache row. 2) Access to a cache block to write or read data. 3) Tag comparison if it is a read cycle. In the following subsections, we discuss how delay and power consumption of the three phases are measured in HSPICE. To estimate energy, we use the following equation:

$$E = \int IVdt = \int V_{RMS} I_{RMS} dt = V_{RMS} I_{RMS} \int dt = V_{RMS} I_{RMS} t_{cycle} \qquad (5\text{-}1)$$

To estimate either dynamic or static power, we calculate average values over several cycles. We ignore the first few cycles in simulations to avoid the possible wrong initial conditions set by HSPICE. For instance, in decoder, we take the average of cycle #6, #7, #8 and #9 in our simulations to estimate I_{RMS}.

5.1.2.1 Decoder

Figure 5-1 shows the simulation setup for a row decoder. It is composed of two parts: pre-decoder and final decoder. Each part is supplied by a single power supply in order to monitor power supply current separately. For each address line, pre-decoder generates both signals and complemented signals such as Ao0 and Ao0B for input A0 in Figure 5-2. Final decoder calculates bit-wise AND of all inputs to generate word-line signal for a cache row.

The whole cache shares a pre-decoder. However, each cache row has its own final decoder. Therefore, for a cache with 256 rows, the current drawn from power supply should be multiplied by 256 to estimate the worse case scenario.

To estimate power in decoder, we use a pessimistic approach and assume that half of address bits toggle. We use pulse signals in HSPICE to model toggling of signals in A0, A2,

A4, and A8 in Figure 5-3. This results in worst case scenario since most of the gates in Figure 5-3 consume dynamic power.

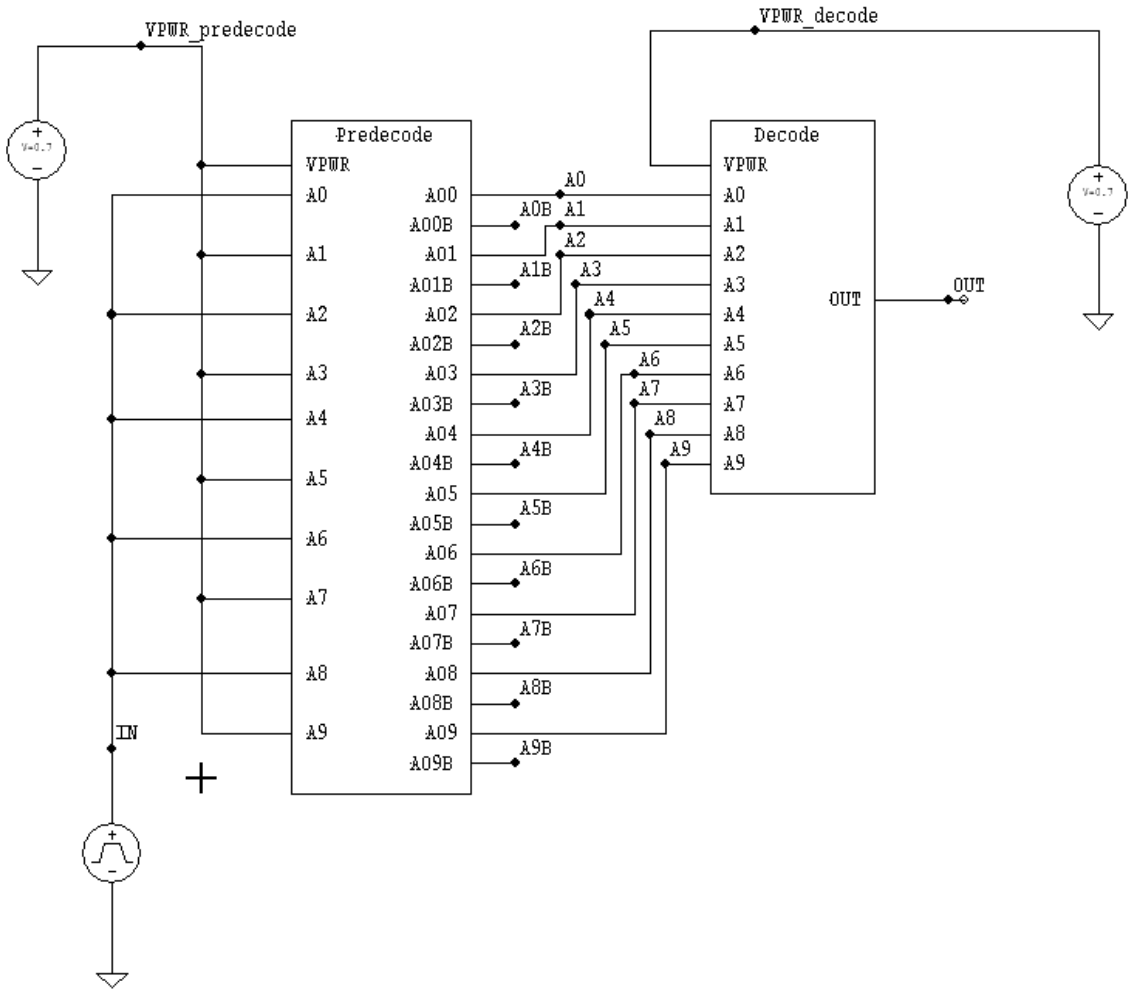


Figure 5-1: Decoder simulation setup.

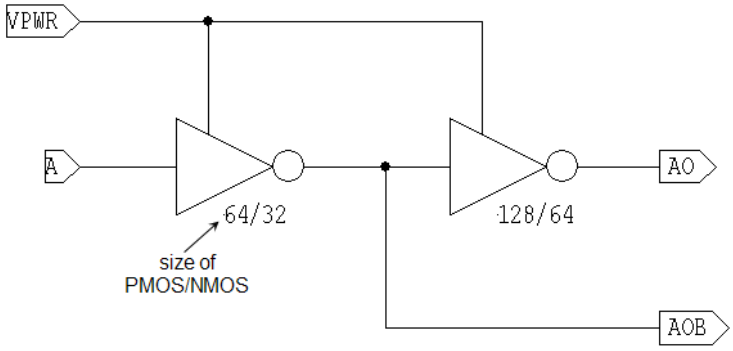


Figure 5-2: Predecoder circuit for each input.

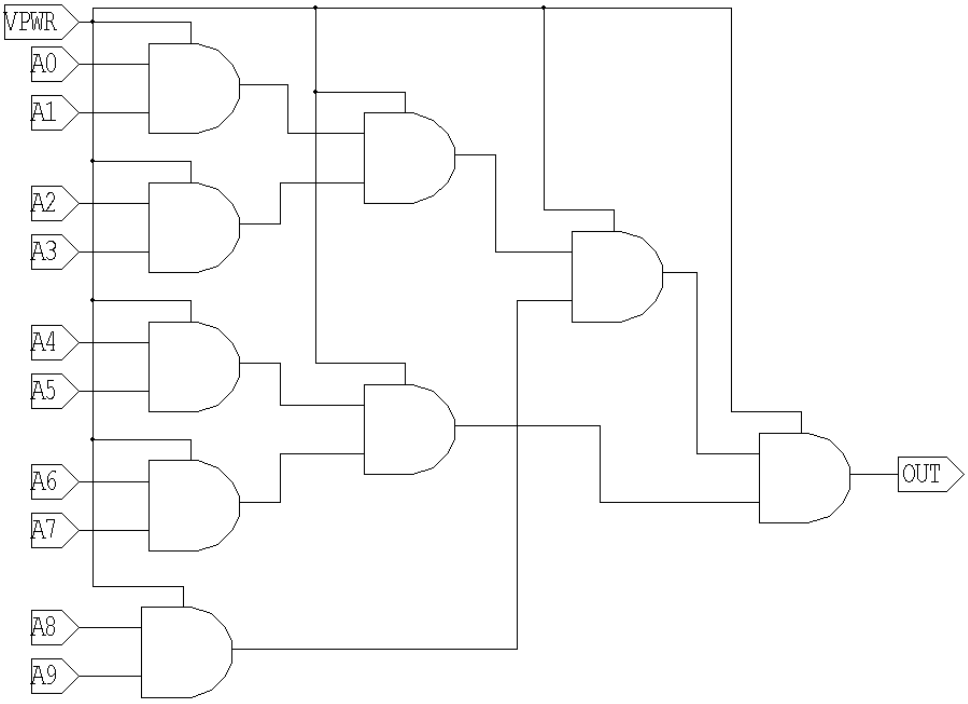
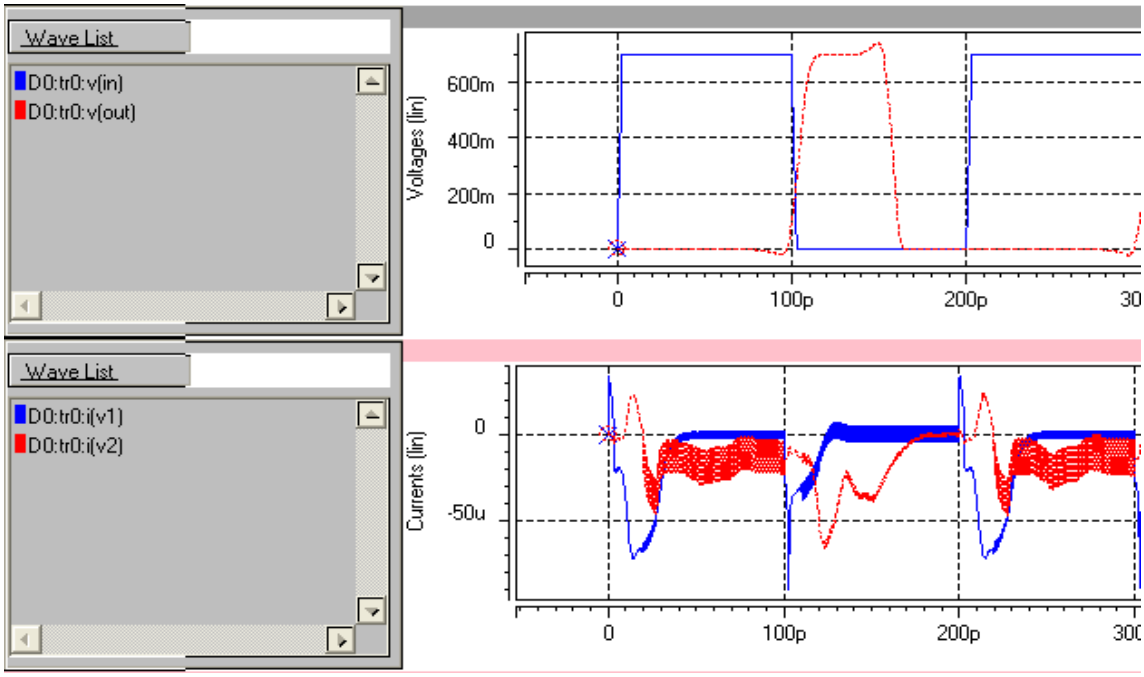


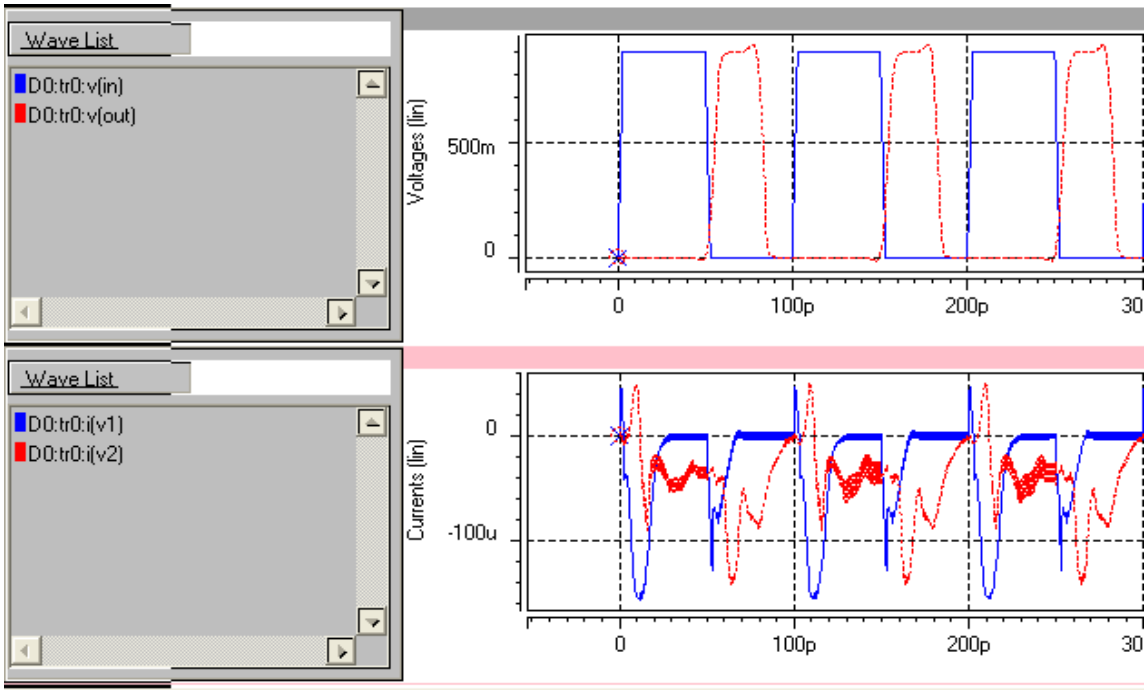
Figure 5-3: Row decoder circuit for each row.

The waveforms of simulation results are plotted in Figure 5-4. Figure 5-4a shows simulation results when Vdd= 0.7v. The top panel has 2 signals: IN (address, solid line) and OUT (row select, dashed line). The bottom panel plots the current of pre-decoder (v1, solid line) and final decoder (v2, dashed line). Note that IN signal has two states; 0 and 1. We use

.MEASURE command in HSPICE to get the RMS values of currents then multiply it by Vdd to get the total energy. The results are tabulated in Table 5-1.



(a)



(b)

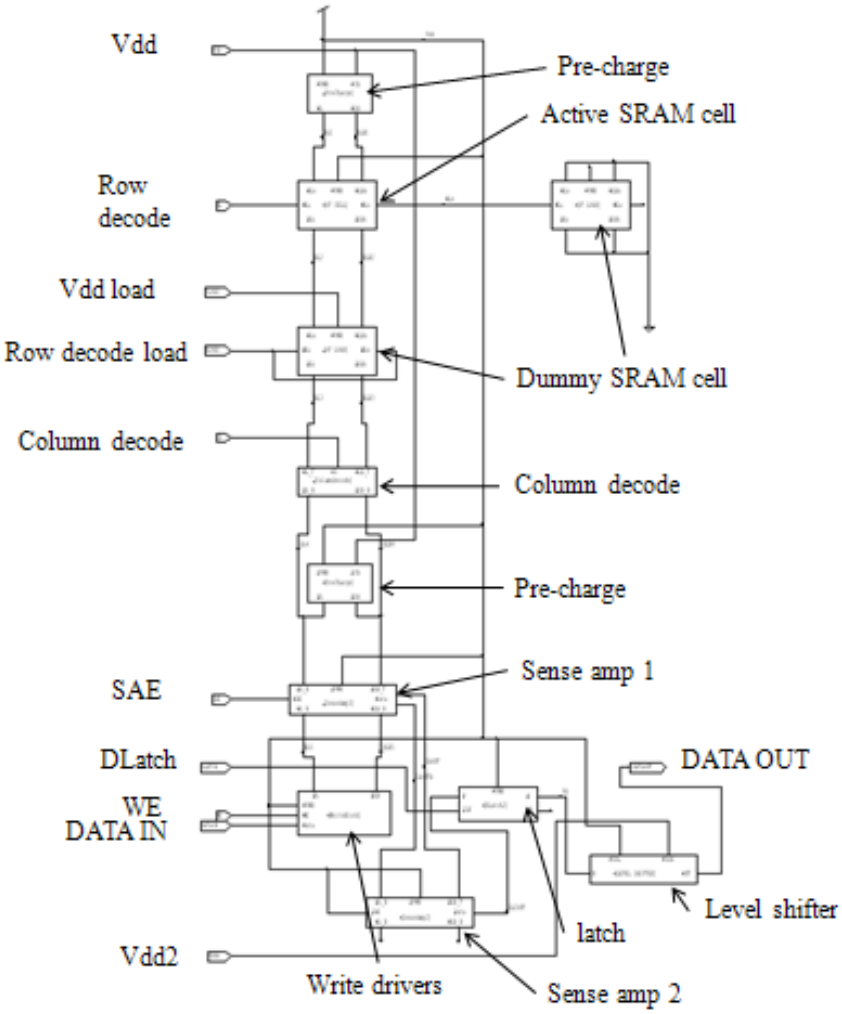
Figure 5-4: Waveforms for decoder (a) 0.7v (b) 0.9v.

Table 5-1: Total energy consumption in decoder

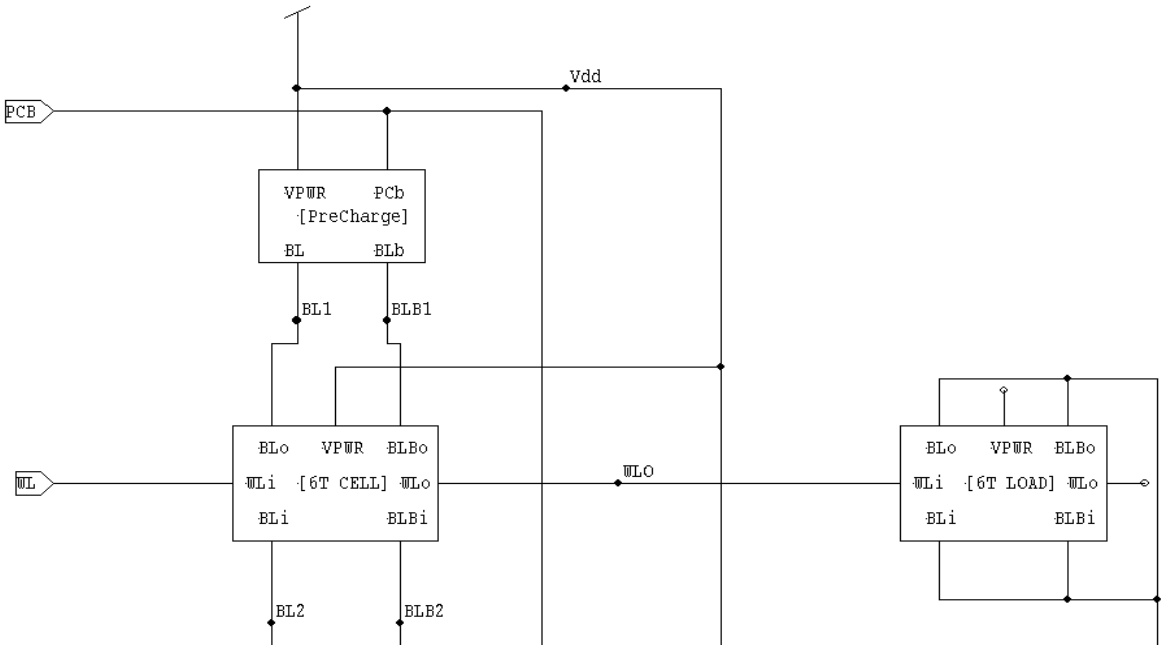
Total Energy	0.7v	0.9v	0.7v/0.9v (%)
dynamic	1.12367E-12	1.84544E-12	60.89%
static	1.77239E-14	1.08329E-13	16.36%
time	1.5E-10	6E-11	250.00%

5.1.2.2 SRAM column

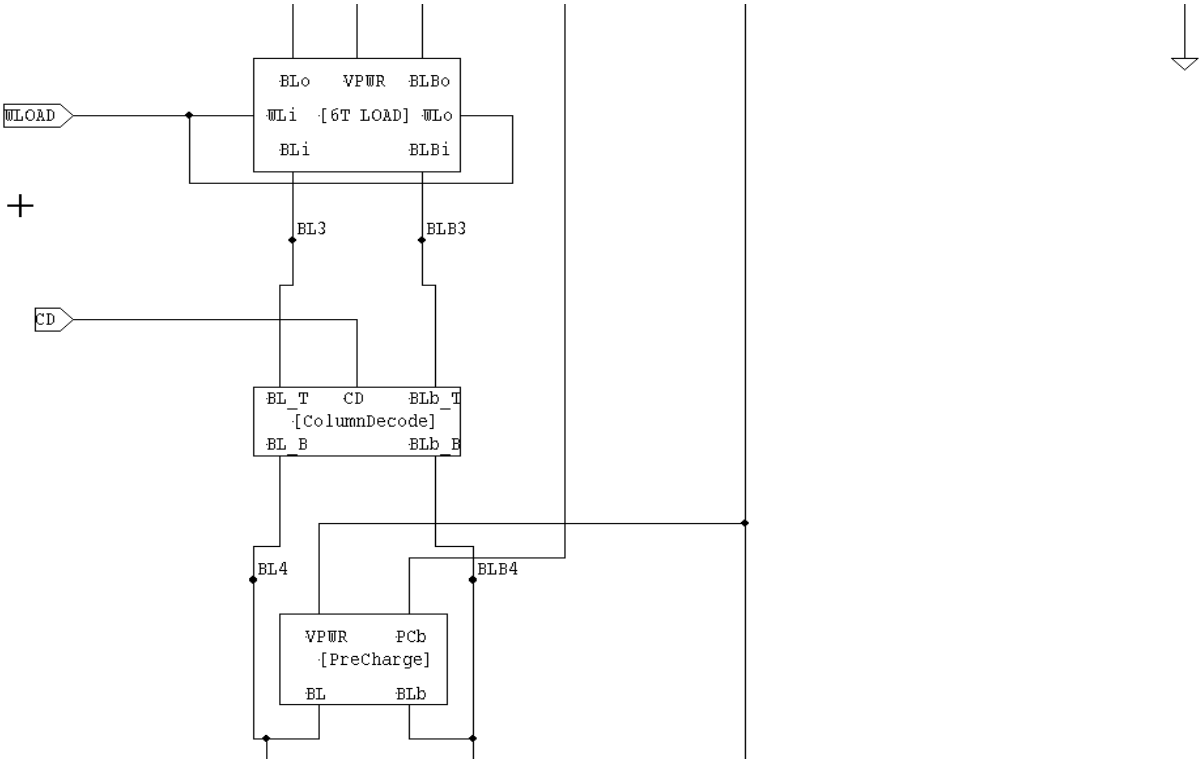
Figure 5-5 shows the simulation setup for SRAM column. It includes all analog blocks including sense amplifiers, write drivers, latches, level shifters, etc. It also includes an active 6T SRAM cell which we perform write “1”, read “1”, write “0” and read “0” alternatively to ensure both write and read operations are correct. Also, two dummy SRAM cells are included to simulate the capacitive load on WL, BL and BLB. Details of the sub-circuits are discussed in section 4.1.6.


Vdd
Pre-charge
Active SRAM cell
Row decode
Vdd load
Row decode load
Dummy SRAM cell
Column decode
Column decode
Pre-charge
Sense amp 1
SAE
DLatch
DATA OUT
WE
DATA IN
Level shifter
latch
Vdd2
Write drivers
Sense amp 2

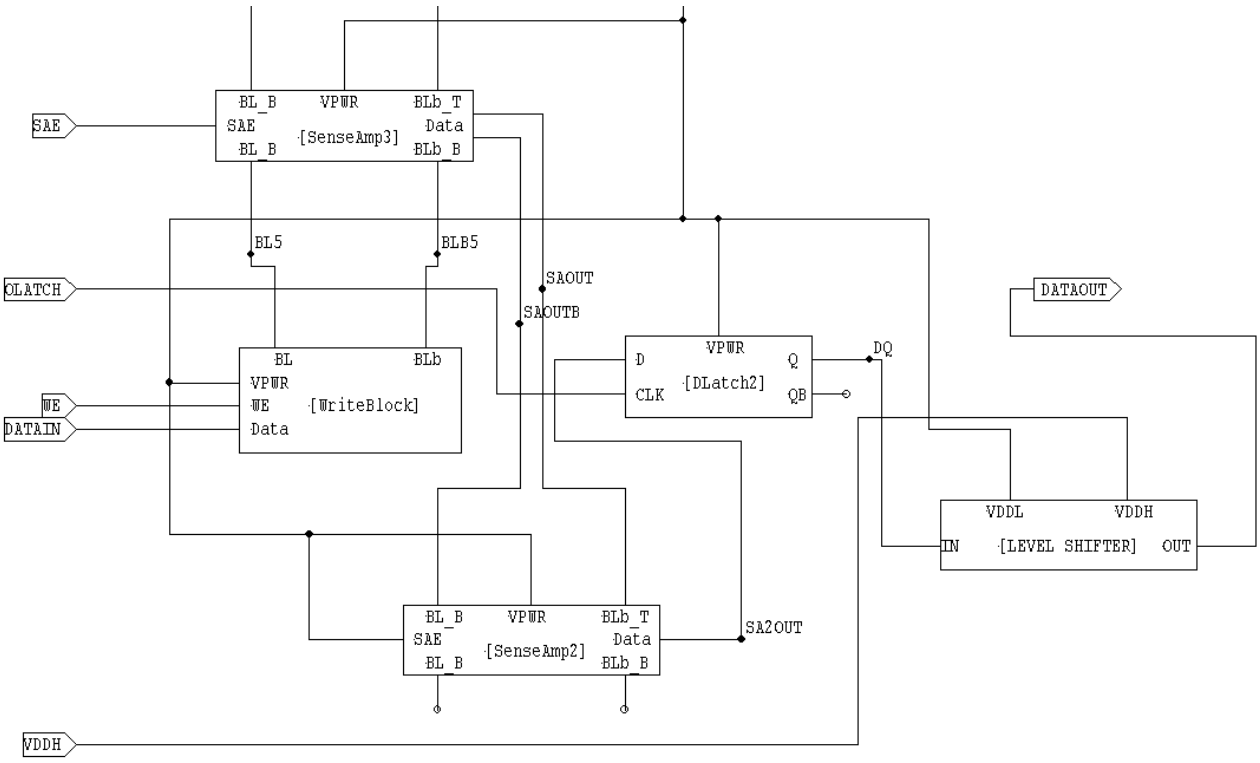
(a)


Vdd
PCB
VPWR PCb
[PreCharge]
BL BLb
BL1
BLB1
BLo VPWR BLBo
WLi [6T CELL] WLo
BLi BLBi
WL
WLO
BLo VPWR BLBo
WLi [6T LOAD] WLo
BLi BLBi
BL2
BLB2

(b)


BLo VPWR BLBo
WLi [6T LOAD] WLo
BLi BLBi
WLOAD
BL3
BLB3
CD
BL_T CD BLb_T
[ColumnDecode]
BL_B BLb_B
BL4
BLB4
VPWR PCb
[PreCharge]
BL BLb

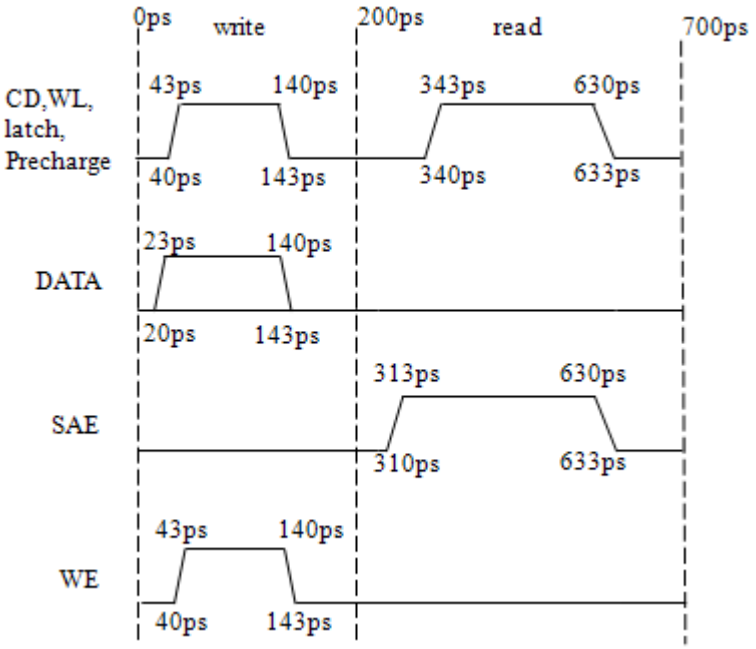
(c)



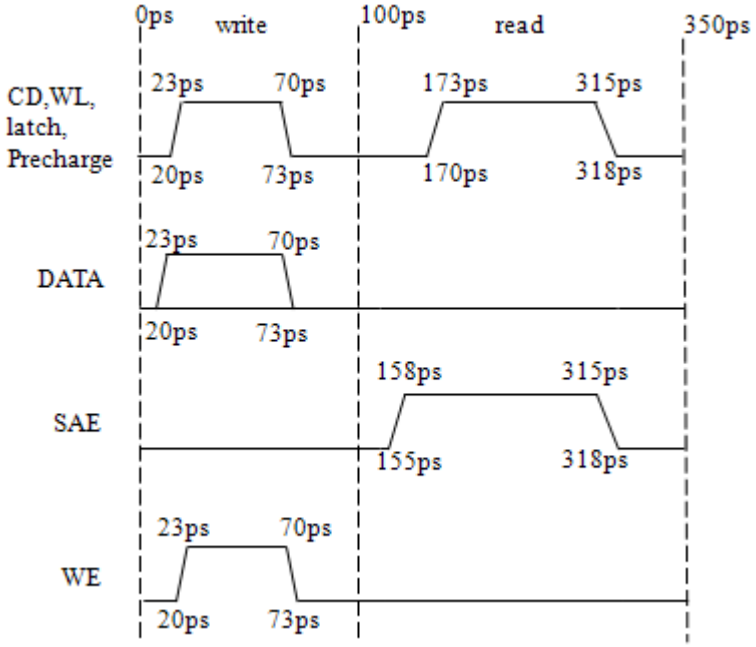
(d)

Figure 5-5: (a) Structure of an SRAM column (b) details of top section (c) details of middle section (d) details of bottom section.

Figure 5-6a shows the time set used for Vdd=0.7v and Figure 5-6b shows the time set for Vdd=0.9v. These time sets ensure that SRAM cells read and write properly.



(a)



(b)

Figure 5-6: (a) 0.7v time set (b) 0.9v time set

As described in section 5.1.2, we write and read alternative data for many cycles. The current for read and write cycles is the average current over several cycles. The reason we calculate average current of several cycles is that the current is different when a cell stores "0" or "1".

The simulation waveforms for Vdd=0.7v are plotted in Figure 5-7. The top panel is Data-In signal which is the data being written into an SRAM cell. The second panel from top is WE (write enable) signal. Whenever WE is high, it indicates a write cycle. Therefore, one can see we alternatively write "1", then "0" indicated by the red color in the first panel. Third panel from top is SAE (sense amp enable) signal. Whenever SAE is high, it is a read cycle. The fourth panel is the Data-Out after level shifter. One can see that we alternatively read "1" and "0" indicated by the red color arrows in the fourth panel. The bottom panel is the current drawn from Vdd. Note that, the initial condition of level shifter set by HSPICE is "1" in this simulation.

The simulation waveforms for Vdd=0.9v are plotted in Figure 5-8. Our circuit can write and read data correctly with only half a cycle. Note that, in this simulation, the initial condition of level shifter set by HSPICE is "0".

The results for SRAM column are tabulated in Table 5-2.

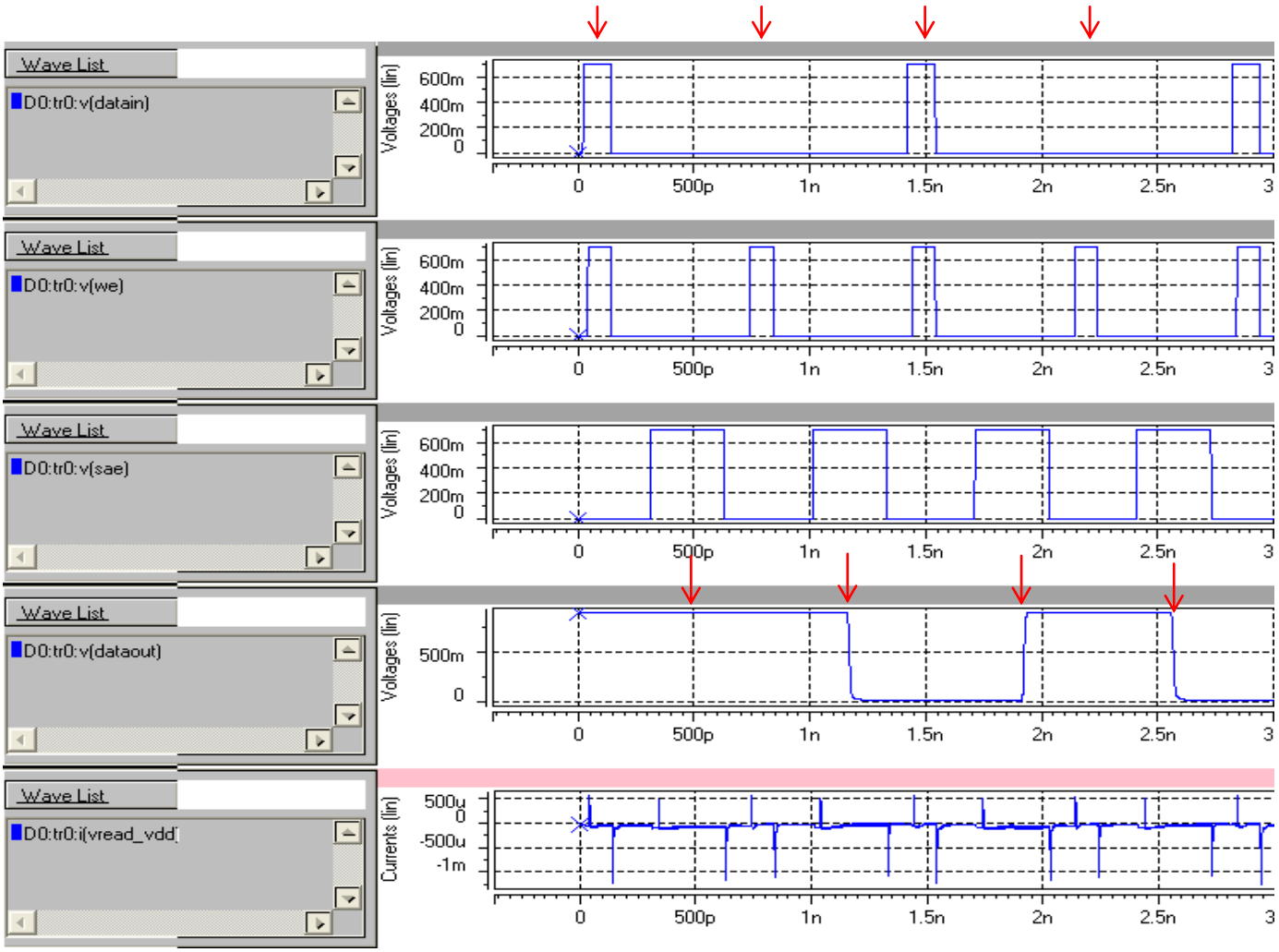


Figure 5-7: 0.7v SRAM column simulation waveforms.

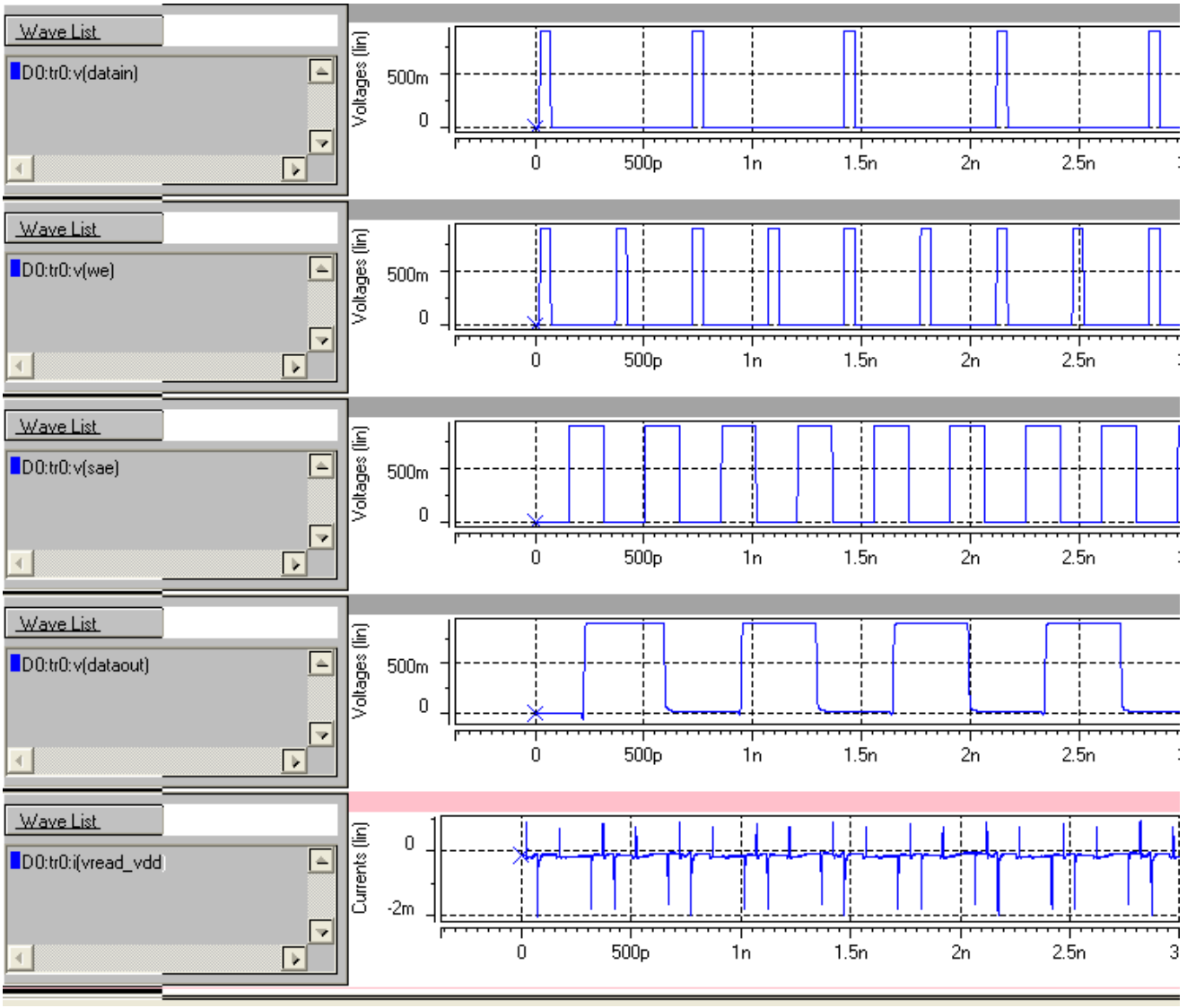


Figure 5-8: 0.9v SRAM column simulation waveforms.

Table 5-2: Total energy consumptions in SRAM columns

Total Energy	0.7v	0.9v	0.7v/0.9v (%)
dynamic write	4.80414E-11	1.37471E-10	34.95%
dynamic read	9.09439E-11	1.53455E-10	59.26%
static	2.05137E-11	3.87018E-11	53.00%
time write	2E-10	1E-10	200.00%
time read	5E-10	2.5E-10	200.00%

5.1.2.3 Tag Comparison

Figure 5-9 shows simulation set up for tag comparison. With the same methodology, we assume half of the input signals toggle. By doing this, the states of half of XOR gates and the states of all OR gates change (Figure 5-10). This results in worst case scenario for estimating dynamic currents.

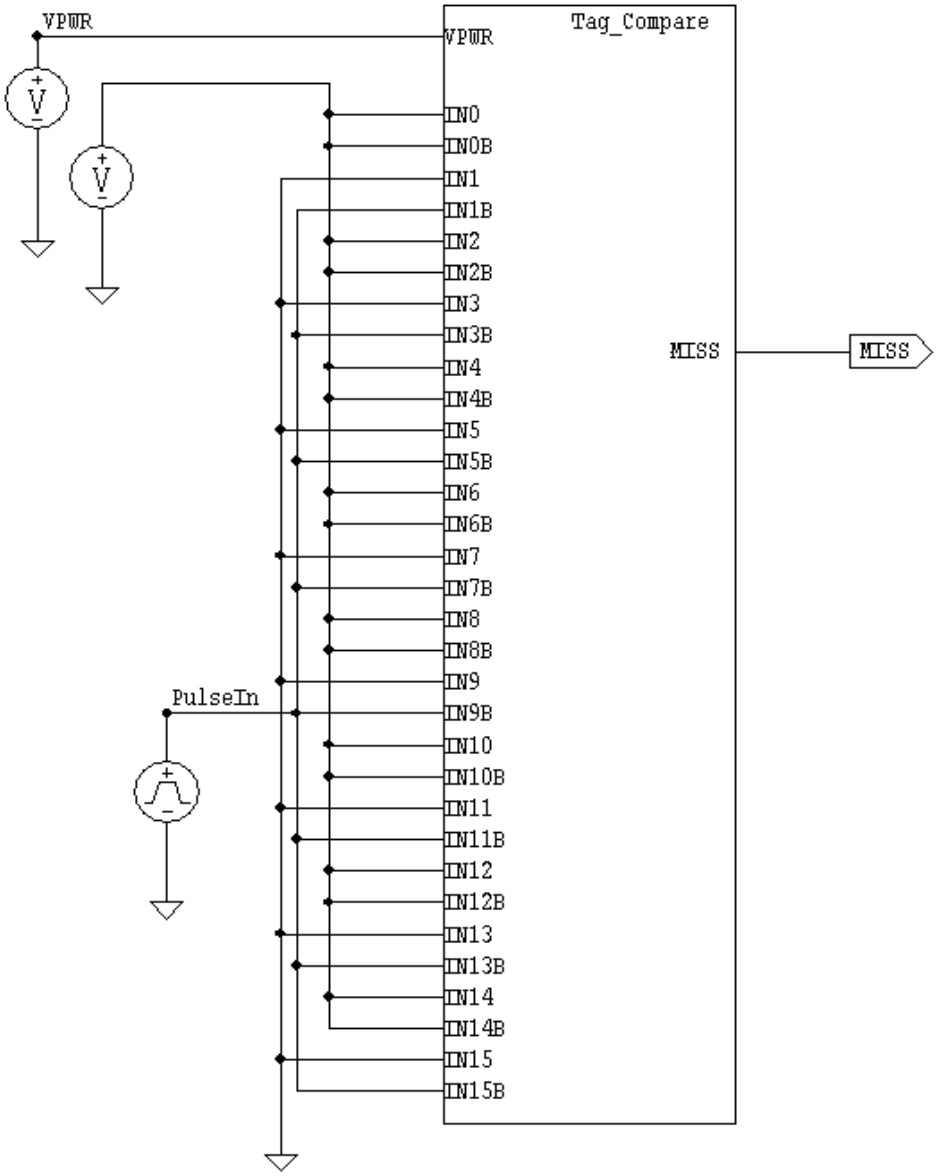


Figure 5-9: Simulation set up for tag comparison.

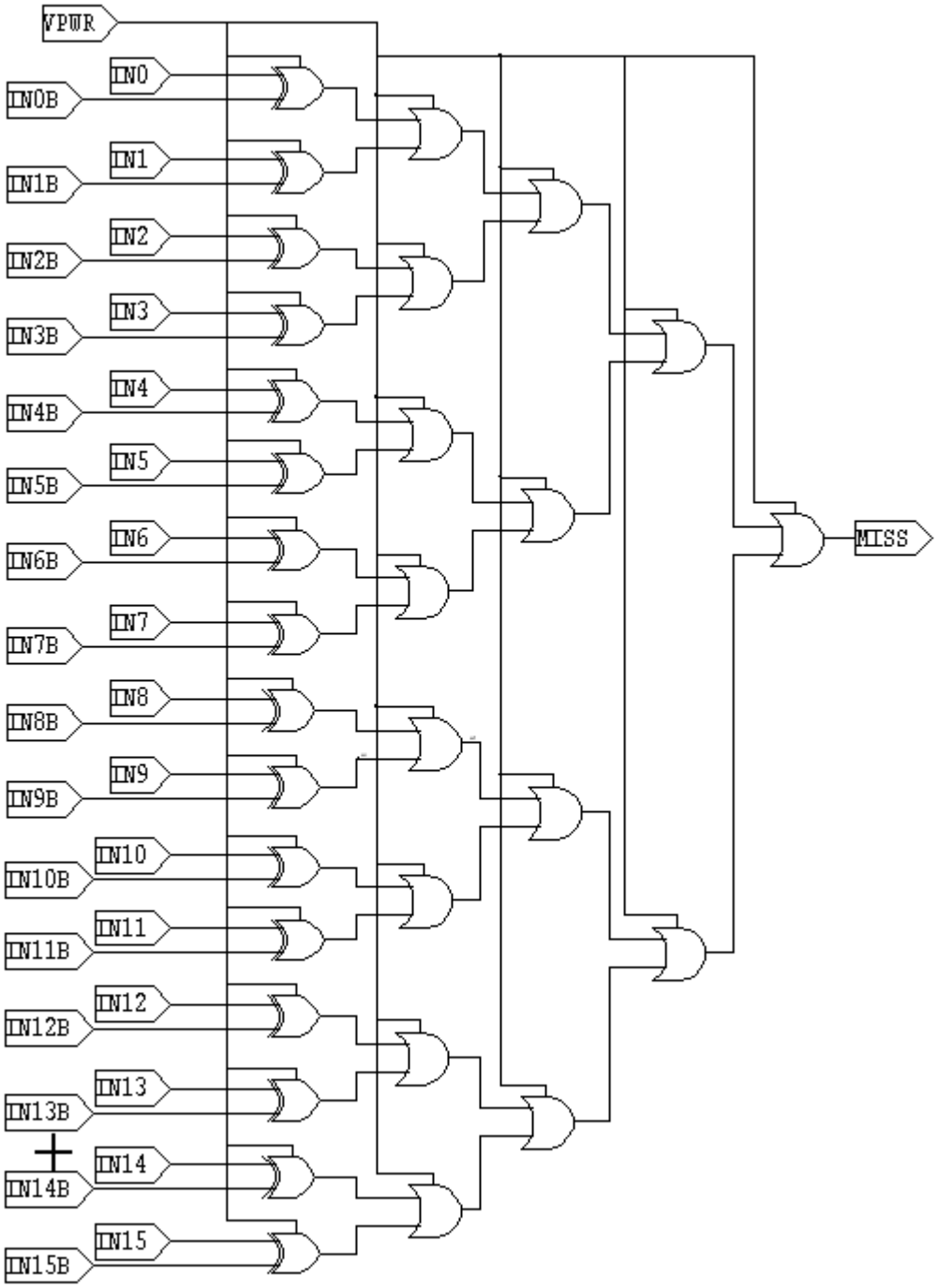


Figure 5-10: Schematic for tag comparison.

The simulation waveforms for 0.7v are plotted in Figure 5-11. In the top panel, the solid line is miss signal indicating a cache miss. The dashed line is the input control signal. The

bottom panel is the current drawn from Vdd. The simulation waveforms for 0.9v are plotted in Figure 5-12. The total energy consumptions are tabulated in Table 5-3.

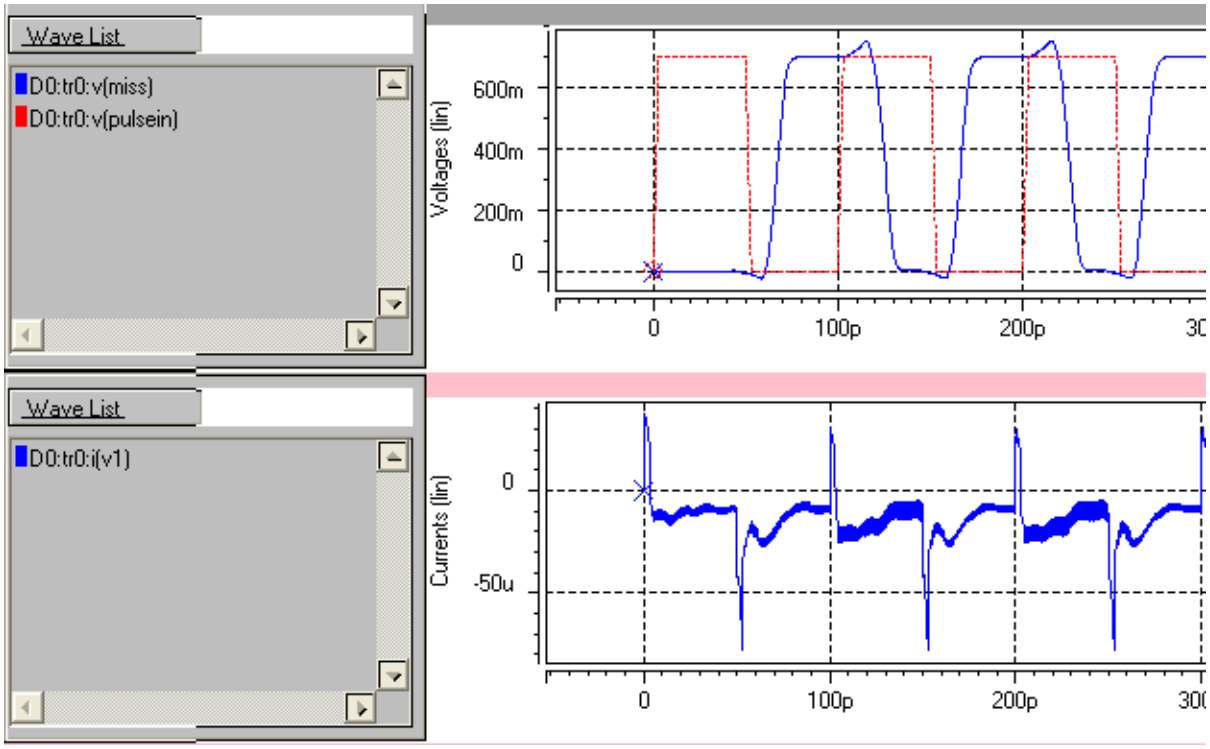


Figure 5-11: 0.7v Simulation waveform for tag comparison.

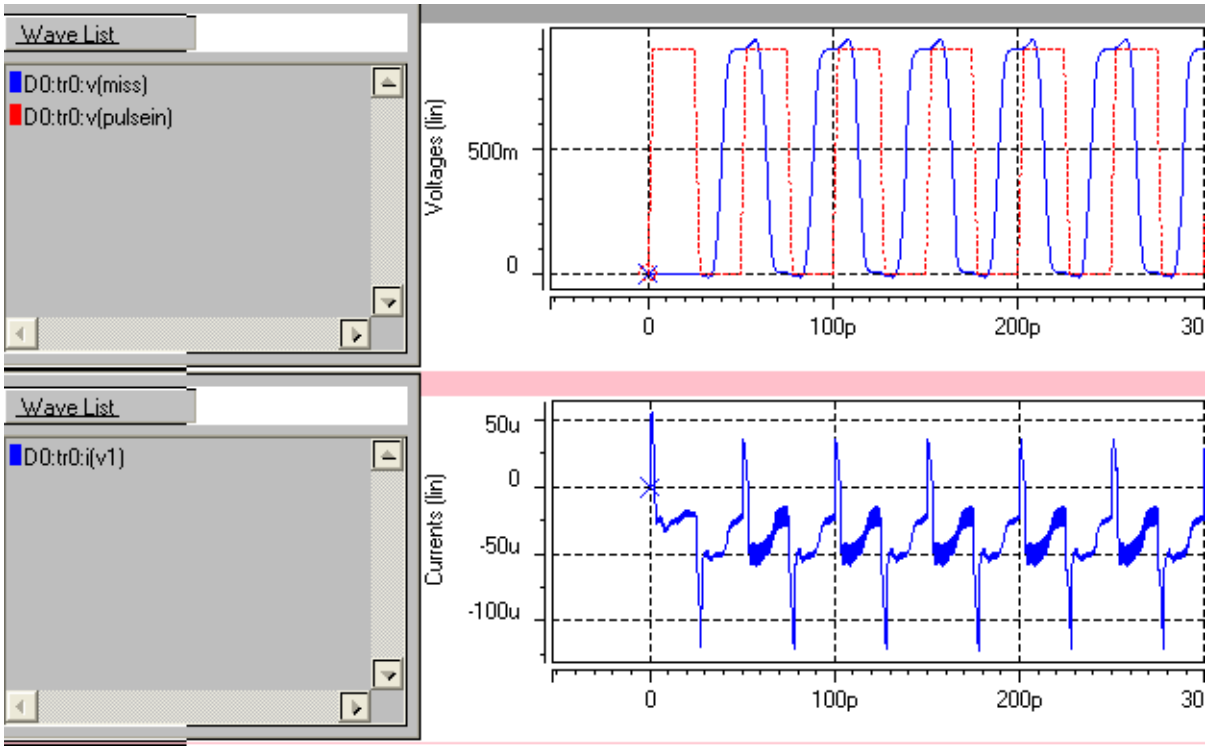


Figure 5-12: 0.9v simulation waveform for tag comparison.

Table 5-3: Total energy consumptions in tag comparator

Total Energy	0.7v	0.9v	0.7v/0.9v (%)
dynamic	2.07699E-14	4.26371E-14	48.71%
static	2.07699E-14	1.02491E-15	2026.51%
time	8E-11	4E-11	200.00%

5.1.2.4 Summary

Table 5-4 summarizes the power consumptions for 0.7v and 0.9v scenarios. At Vdd=0.7v, we need 2 cycles to read or write while we need only 1 cycle at Vdd=0.9v. These numbers are fed into Simplescalar to measure architectural parameters.

Table 5-4: Summary of total power per cycle and cycle time, 2.8GHz

Total Power	0.7v	0.9v	0.7v/0.9v (%)
Dynamic (Write)	0.114385638	0.696793065	16.42%
Dynamic (Read)	0.126148348	0.443836327	28.42%
Static	0.024310838	0.107115685	22.70%
Static (each row)	1.49123E-05	1.47393E-05	101.17%
Time	2 cycles	1 cycle	200.00%

5.1.3 SP2K benchmark results (QOLD)

5.1.3.1 Methodology

We used SPEC CPU2000 benchmark suite [14] to evaluate power and performance impact of the new cache. All benchmarks are compiled with GNU's gcc compiler (flags: -O3-funroll-loops -finline-functions). We used Wattch [15] to simulate our proposed scheme in the context of an out of order microprocessor. Table 5-5 shows the configuration of the processor. We simulated 500M instructions after skipping fast forwarding values generated by the SimPoint toolkit [16].

Table 5-5. Baseline processor configuration

Reorder Buffer Size	128	***L_1 - Data Caches***	32K, 4-way SA, 32-byte blocks, 1 cycle hit latency
Load/Store Queue Size	64	***Unified L_2***	256K, 4-way SA, 64-byte blocks, 12-cycle hit latency
Fetch Unit	Up to 8 instr./cycle. 64-Entry Fetch Buffer	***Main Memory***	Infinite, 100 cycles
OOO Core	8 instructions / cycle	***Memory Port #***	2
L_1 - Instruction Caches	64K, 2-way SA, 32-byte blocks, 1 cycle hit latency	***Branch Predictor***	16K GShare+16K bi-modal w/ 16K selector

We assume 1-cyle delay for L_1 data cache in the baseline configuration. In the baseline configuration, all cache cells use high supply voltage. In our proposed scheme, cache cells have dual supply voltages. Table 5-4 shows static and dynamic power for reads and writes in

the baseline and the proposed schemes. To measure dynamic power, in each cycle, we count the number of reads and writes in low Vdd and high Vdd cache sets and multiply them by the dynamic power reported in Table 5-4. The total energy is sum of power values of all committed cycles. We also take into account the static power of the cache cells. To model latency of low-power cache accurately, we assume one cycle penalty for reads and writes in low Vdd super-sets.

5.1.3.2 Impact of super-set size and counter threshold

Figure 5-13 shows the impact of size of super-sets on both power and performance. The numbers in the plots are the power and performance in low-power cache relative to the baseline scheme. In the baseline scheme, there is no penalty in time. However, in the low-power configuration, there is one cycle penalty if a load/store instruction accesses a low supply voltage cell.

We check 8 instructions in the head of LSQ for criticality and use threshold of 8 for counters. We set the criticality interval to 100000 committed instructions. In Figure 5-13, when the size of the super-set changes from one to 64 rows, performance of the low power cache improves slightly. As the size of super-set increases, it is more likely that critical load/store instructions access a super-set. So, the probability that the value of the counter exceeds threshold increases. As such, more super-sets are configured for high Vdd. This reduces performance loss and also reduces power improvement in the low power cache.

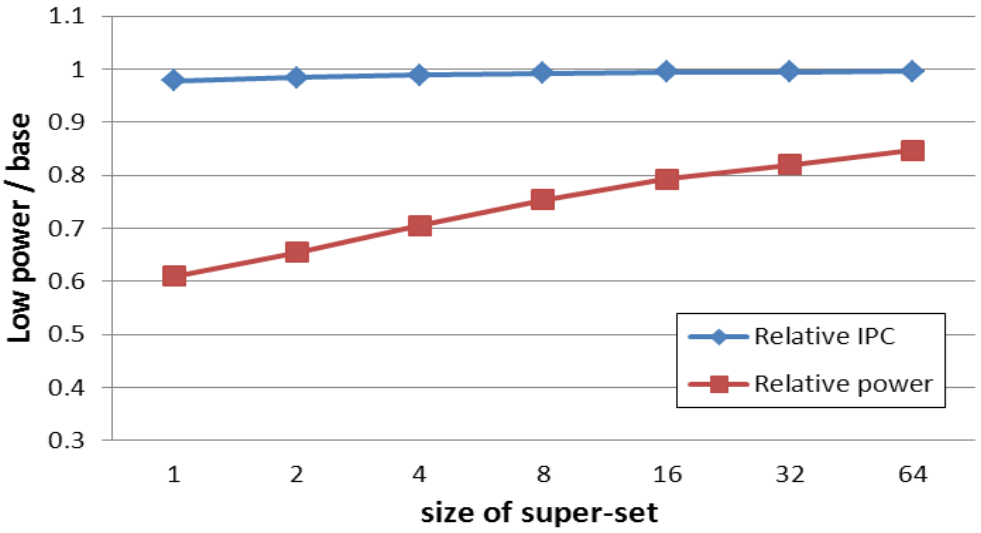


Figure 5-13: Impact of super-set size in Spec2000 benchmarks (N=8, threshold=8 and M=100000).

Figure 5-14 exhibits the impact of counter threshold on power and performance. As the threshold increases, power saving improves but performance decreases. The rationale is that for a given super-set size (8 rows in Figure 4-23), it is less likely to have 64 critical load/store instructions accessing a super-set than just 1. If the number of critical instructions accessing a super-set is less than the threshold, all cache blocks in the super-set are assigned to low supply voltage. This increases latency of the cache but enhances the power. Therefore, the performance reduces and power improves when threshold increases.

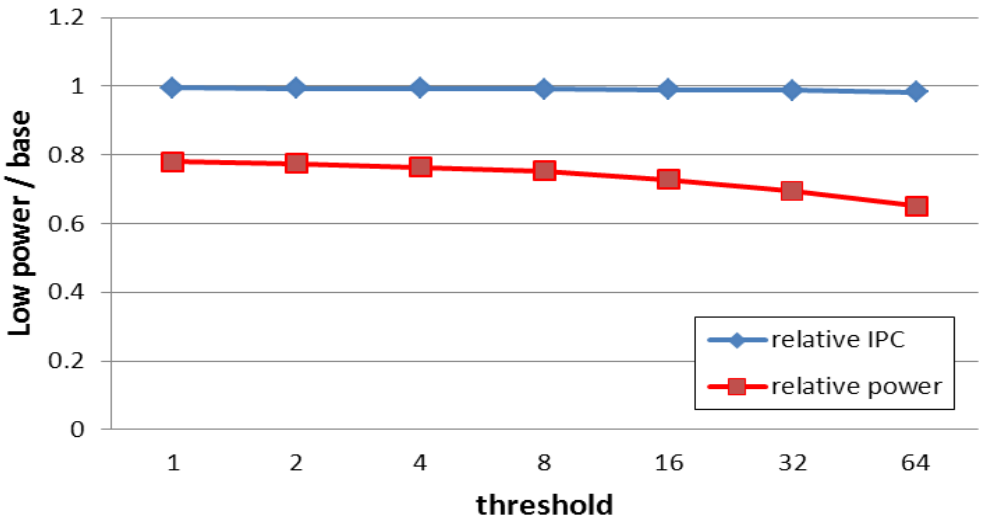


Figure 5-14: Impact of threshold on Spec2000 benchmarks. (super-set=8 rows, N=8, M=100000)

5.1.3.3 Impact of criticality interval (M)

We check criticality counters after a certain number of instructions are committed as this interval impacts the performance and power as well. Figure 5-15 and 5-16 show the impact of criticality interval on power and performance, respectively of selected benchmarks. We ran those Spec2000 benchmarks with criticality interval changes from 1000 to 1000000 instructions while super-set size =8, critical threshold=8 and LSQ instructions (N) =8. As the interval increases the power saving reduces. This is due to the fact that when interval increases it is more likely to have the counters exceeding the threshold. As such, more super-sets are configured to use high Vdd. This reduces power saving and also reduces performance loss.

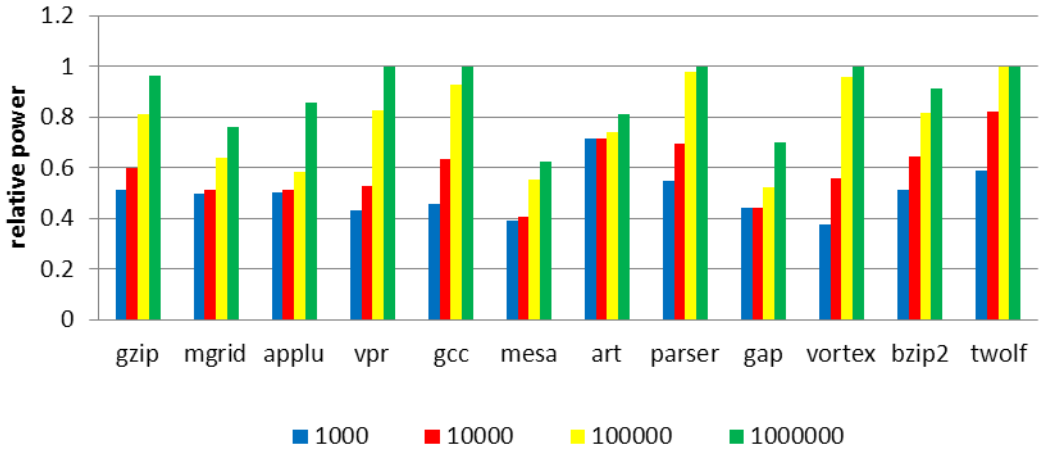


Figure 5-15: Power consumption in Spec2000 benchmarks when criticality interval changes.

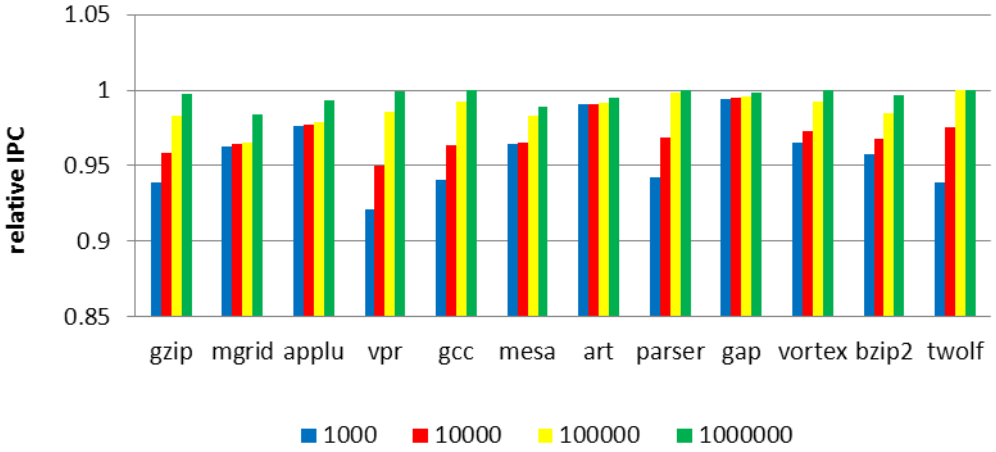


Figure 5-16: Performance in Spec2000 benchmarks when criticality interval changes.

5.1.3.4 Impact of N

To classify instructions into critical and non-critical, the first *N* instructions in LSQ are checked. They are marked as critical if their source operands are not ready. Figure 5-17 and 5-18 show the effect of *N* on power and performance. *N* changes from one to 16 in each benchmark. As *N* increases, it is more likely to mark an instruction as critical. Hence, the chance that counters exceed threshold increases. This reduces power saving and improves performance.

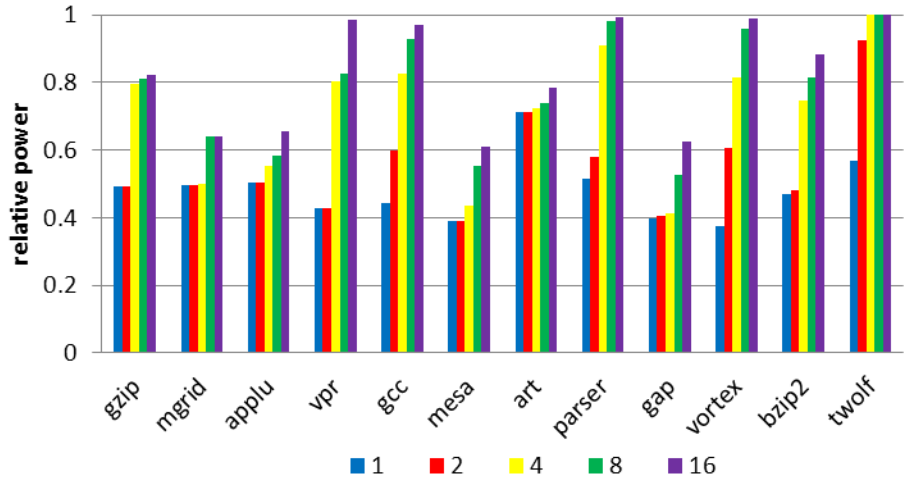


Figure 5-17: Power consumption in Spec2000 benchmarks when N increase.

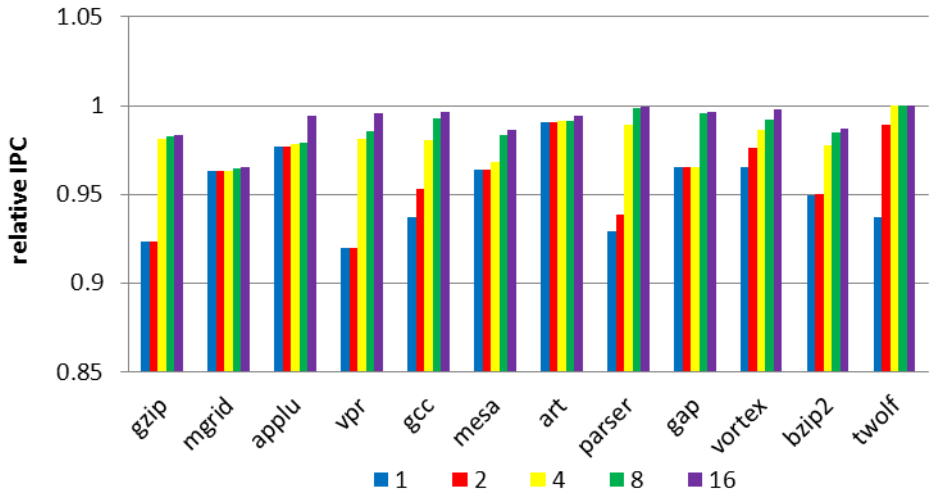


Figure 5-18: Performance in Spec2000 benchmarks when N increase.

5.1.3.5 Summary

We have evaluated the effect of different parameters such as super-set size, counter threshold, criticality interval and N on both power and performance. We found that the optimum configuration is when super-set size is 8, counter threshold is 8, criticality interval is 100000 and N is 8. With this configuration, we sacrifice 1.2% of performance in exchange for 22% power saving.

5.2 Low power register file

5.2.1 Critical instruction policy (QOLD)

Similar to the cache, we partition the registers files into super-sets. All the registers within a super-set have the same supply voltage even if some of them are accessed by critical and some by non-critical instructions. We modeled two register files in this application: integer and floating point. Each register file has 64 registers. The width of each register is 64-bit.

Instructions in issue window are issued when their source operands are ready. If source operand of an instruction is ready in register file, then the instruction accesses register file to read its source operand and so consumes power. However, if the source operand of an instruction is forwarded by a producer instruction, then the instruction does not access register file and so there is no power consumption for the register file.

At each cycle, we inspect the first N instructions in the head of re-order buffer. If an instruction is "not ready", we threat it as a critical instruction. Each super-set has its own counter. We evaluate the counters after M instructions are committed, where M is called criticality interval and is a pre-determined parameter. If the counter exceeds a pre-determined threshold, all SRAM cells in the super-set are assigned to high supply voltage to preserve performance; otherwise, we use low supply voltage to save power. After adjusting the supply voltage of the super-set, all the criticality counters are reset to zero for the next criticality interval (M). With this organization, the supply voltage of registers (both integer and floating point) is dynamically adjusted to enhance performance and reduce power consumption.

We assume an extra cycle delay for low Vdd registers. We take into account the delay of source operands in issue stage and the delay of output operand in commit stage.

In this study, in addition to QOLD policy, we also ran other policies such as QCONS and FREED. The methodology and benchmark results of QCONS are listed and discussed in section 7.1 and that of FREED are in section 7.2.

5.2.2 HSPICE simulations and results

The structure of register file is much simpler than the structure of a cache. In a register file, we only model row decoder and SRAM cells with analog blocks. Tag field and tag comparators are not needed in register files. Most of the processors use register files with 32 registers and we do the same in our simulations [41].

Table 5-6: Register file size [41].

Processor	Integer RF	Floating-point RF
ARM	32	8
Power PC	32	32
Xscale	32	32
Transmeta Crusoe	64	64

5.2.2.1 Decoder

Figure 5-19 shows the simulation setup for a register file row decoder. The register file has 32 registers. Therefore, there are only 5 address bits.

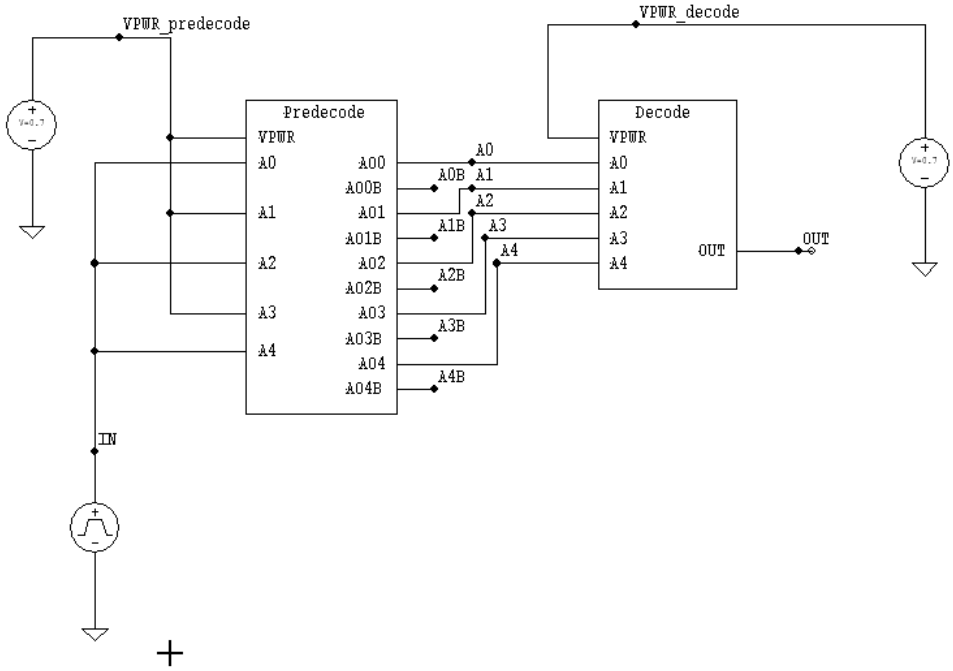


Figure 5-19: Simulation schematic for decoder in register file

The detailed schematics of pre-decoder and final decoder are similar to Figure 5-2 and Figure 5-3 with fewer gates as the register file has only 5 addresses.

Similar to the cache, we toggle half of the input signals to emulate the worst case scenario. All the waveforms are similar to those in Figure 5-4 with lower amplitude (fewer gates in register file simulations). The results are tabulated in Table 5-7.

Table 5-7: Total energy consumptions in register file decoder.

Total Energy	0.7v	0.9v	0.7v/0.9v (%)
dynamic	1.58425E-13	2.58546E-13	61.28%
static	4.74802E-15	1.05814E-14	44.87%
time	1.5E-10	6E-11	250.00%

5.2.2.2 SRAM column

The SRAM column structure in a register file is the same as that of in a cache (Figure 5-5). The only difference is that cache has 256 rows while register file has only 32 rows. In other words, the capacitive loading for BL and BLB is smaller in register file. We used the same time set as those used for cache (Figure 5-6). The simulation waveforms are the same as those in Figure 5-7 and Figure 6-8, only the magnitude is smaller for register file. Table 5-8 tabulates the total energy consumptions in register file.

Table 5-8: Total energy consumptions in register file SRAM columns

Total Energy	0.7v	0.9v	0.7v/0.9v (%)
dynamic write	4.80414E-11	1.37471E-10	34.95%
dynamic read	9.09439E-11	1.53455E-10	59.26%
static	2.05137E-11	3.87018E-11	53.00%
time write	2E-10	1E-10	200.00%
time read	5E-10	2.5E-10	200.00%

5.2.2.3 Summary

Table 5-9 summarizes the power consumptions for 0.7v and 0.9v for a register file. To access the register file, we need 2-cycle at Vdd=0.7v while we need only 1 cycle for Vdd=0.9v. These numbers are fed into Simplescalar to measure architectural parameters.

Table 5-9: Summary of register file's total power and cycle time

Total Power	0.7v	0.9v	0.7v/0.9v (%)
Dynamic (Write)	0.003142854	0.021584258	14.56%
Dynamic (Read)	0.003252489	0.015407699	21.11%
Static	0.000943891	0.005774661	16.35%
Time	2 cycles	1 cycle	200.00%

5.2.3 SP2K benchmark results (QOLD)

5.2.3.1 Methodology

We assume 1-cyle delay for register files in the baseline configuration. In the baseline configuration, all register file cells use high supply voltage. In our proposed scheme, registers have dual supply voltages. Table 5-9 shows static and dynamic power for reads and writes in the baseline and the proposed schemes. To measure dynamic power, in each cycle, we count the number of reads and writes in low Vdd and high Vdd super-sets in both integer and floating point register files and multiply them by the dynamic power reported in Table 5-9. The total energy is sum of power values of all committed cycles. We also take into account the static power of the registers. To model latency of low-power register cells accurately, we assume one cycle penalty for reads and writes in low Vdd super-sets.

Due to the difference between size of register file and cache, the maximum super-set size and the maximum counter threshold are set differently for register file.

5.2.3.2 Impact of super-set size and counter threshold

Figure 5-20 shows the impact of size of super-sets on both power and performance. The numbers in the plots are the power and performance in low-power register files relative to the baseline scheme. We check 8 instructions in the head of re-order buffer (RUU) for criticality and use threshold of 64 for counters. We set the criticality interval to 100000 committed instructions. In Figure 5-20, when the size of the super-set changes from one to 8 rows, performance of the low power register files improves slightly. As the size of super-set increases, it is more likely that critical instructions access a super-set. So, the probability that

the value of the counter exceeds threshold increases. As such, more super-sets are configured for high Vdd. This reduces performance loss and also reduces power improvement in the low power register files.

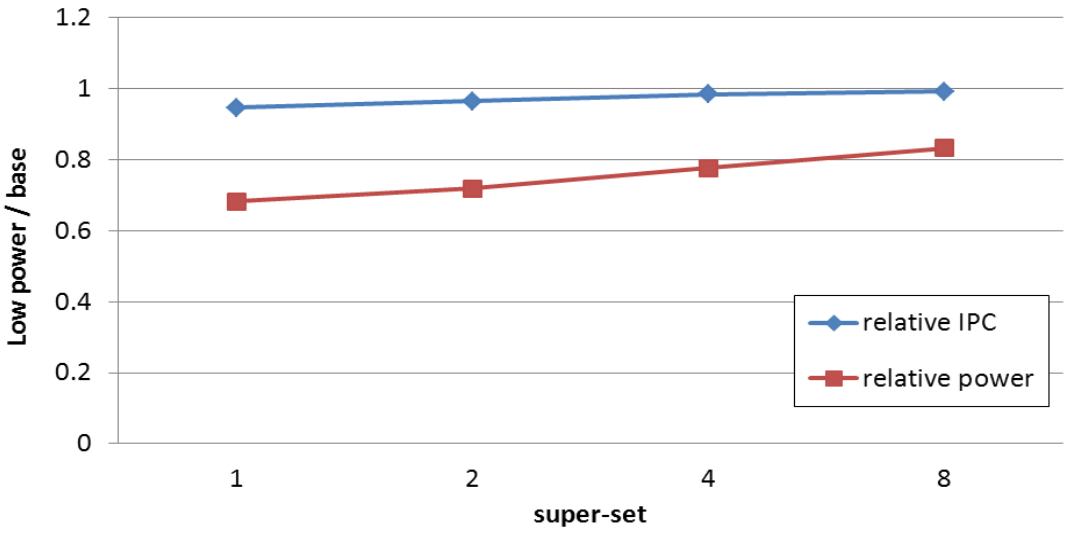


Figure 5-20: Impact of super-set size on power and performance of register files (N=8, threshold=64 and M=100000)

Figure 5-21 exhibits the impact of counter threshold on power and performance. As the threshold increases power saving improves but performance reduces. The rationale is that for a given super-set size, it is less likely to have 1024 critical instructions accessing a super-set than just 1. If the number of critical instructions accessing a super-set is less than threshold, all cache blocks in the super-set are assigned to low supply voltage. This increases latency of the cache but enhances the power. Therefore, the performance reduces and power improves when threshold increases.

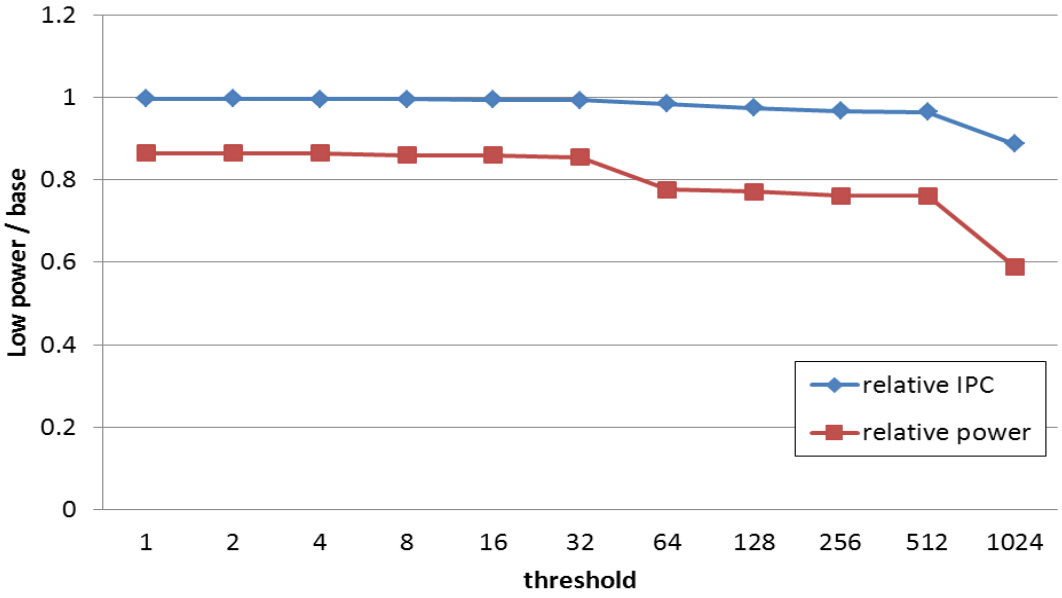


Figure 5-21: Impact of threshold on power and performance of register files. (super-set=4 rows, N=8, M=100000)

5.2.3.3 Impact of criticality interval (M)

Figure 5-22 and 5-23 show the impact of criticality interval on power and performance, respectively. The interval changes from 1000 to 1000000 instructions. As the interval increases the power saving reduces. This is due to the fact that when interval increases it is more likely to have the counters exceeding threshold. As such, more super- sets are configured to use high Vdd. This reduces power saving and also reduces performance loss.

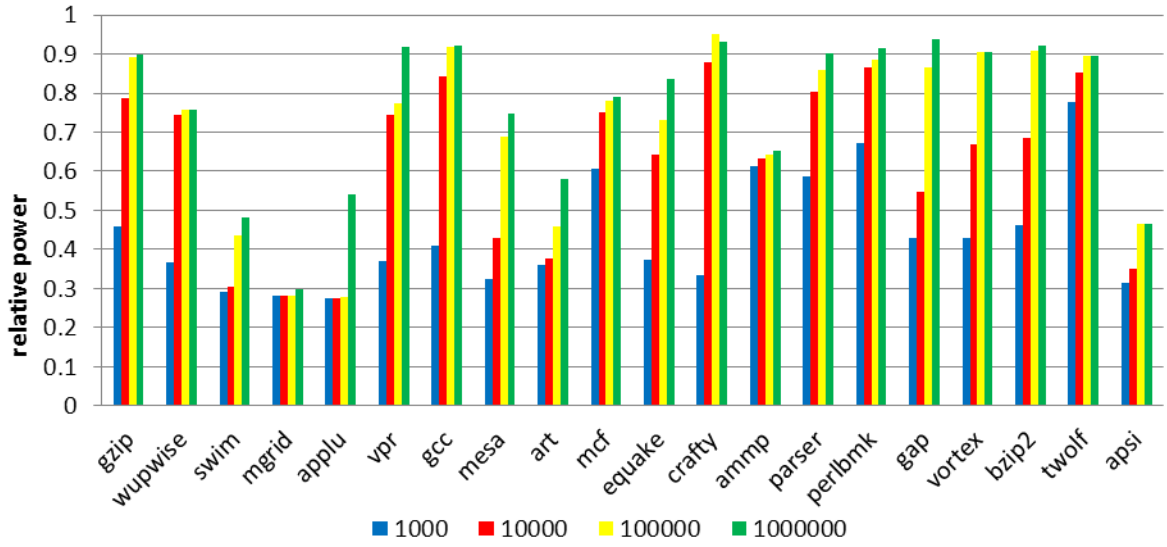


Figure 5-22: Power consumption in register files when criticality interval changes.

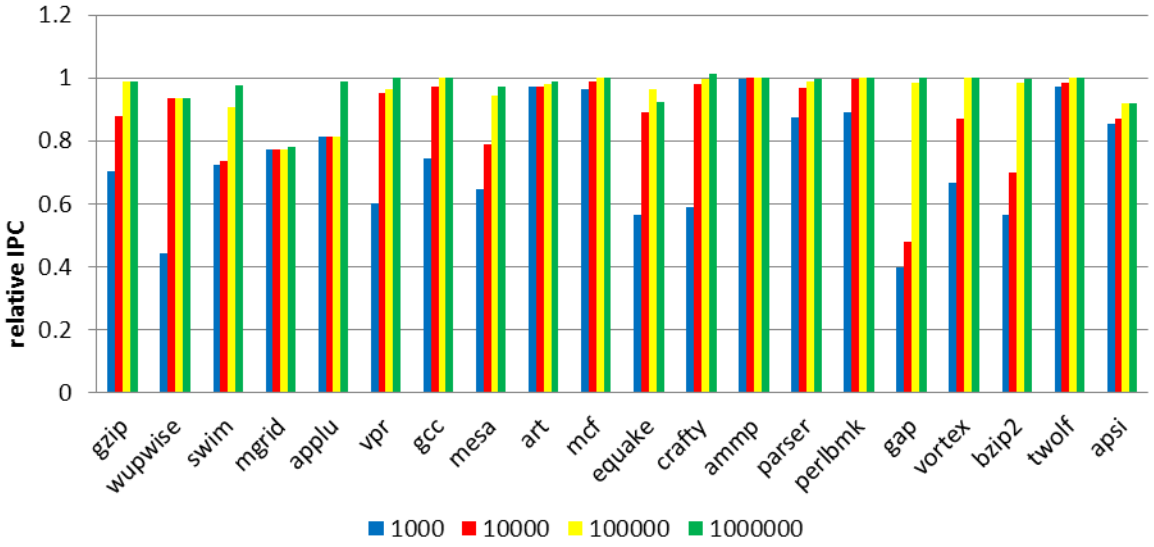


Figure 5-23: Performance in register files when criticality interval changes.

5.2.3.4 Impact of N

Figure 5-24 and 5-25 show the effect of *N* on power and performance. *N* changes from one to 16 in each benchmark. As *N* increases, it is more likely to mark an instruction as critical in the head of re-order buffer. Hence, the chance that counters exceed threshold increases. This reduces power saving and improves performance.

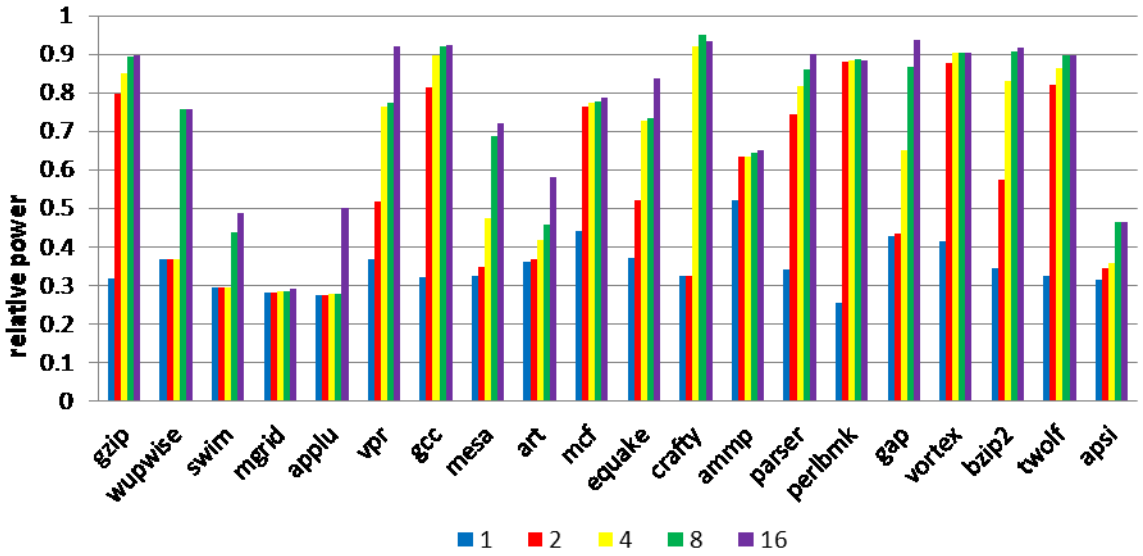


Figure 5-24: Power of register files when N increase.

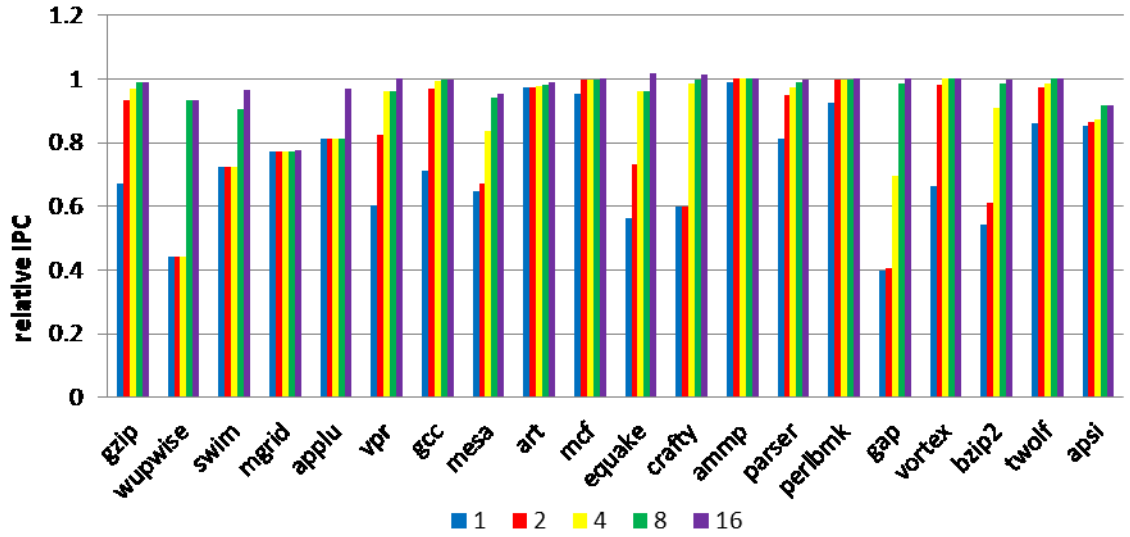


Figure 5-25: Performance of register files when N increase.

5.2.3.5 Summary

Based on simulation results, when super-set size is 4, counter threshold is 64, criticality interval is 100000 and N is 8, performance is reduced by 1.5% and power is improved by 23%.

Section 6
Conclusions and future work

In this thesis, we proposed and evaluated an integrated architectural and circuit level approach to reduce power of caches and register files while maintaining performance. We proposed an SRAM cell for cache and register file which is able to change supply voltage dynamically. Our SRAM cell provides a trade-off between power and latency. Low voltage cells are power efficient and slow but high voltage cells are power hungry and fast. To minimize performance impact of low voltage cells, we classify stream of instructions into critical and non-critical instructions. The non-critical instructions which are more latency tolerant are assigned to low supply voltage cells. On the other side, critical instructions are assigned to high supply voltage cells. Therefore, this scheme reduces the impact of low power cells on performance.

There are many factors that influence the overall power saving such as size of super-set, threshold, etc. We have evaluated the effect of different parameters on power and performance and found that for N= 8, threshold= 8, super-set size= 8, interval= 100000, on average, the total power of L_1 cache is improved by 22% with 1.2% performance degradation. We also found that in low power register files, when N= 8, threshold= 64, super-set size= 8, interval= 100000, on average, the total power is improved by 23% with 1.5% performance degradation.

In this study, we have used a hardware approach to change supply voltage of cache cells and register files. The alternative approach is that a complier analyses programs and order the hardware to change supply voltage. The compiler can analyze the code and finds regions of code that have significant number of critical instructions. For those regions, supply voltage is set to high. For other regions, supply voltage is set to low. This reduces complexity of the hardware.

The other extension of this study is using low power register file in Graphic Processing Units (GPU). GPUs use register files to hold state of threads. This makes context switching in GPUs much faster than CPUs. To be able to support large number of threads, GPUs exploit large register files. These large register files can be even larger than L_1 caches. Therefore, the static and dynamic power consumption of register files is substantial in GPUs. Our low power cells could be used in GPUs to lower power consumption of register files with negligible impact on performance.

Section 7 APPENDIX

7.1 Methodology and Results (QCONS)

7.1.1 Low Power Cache

7.1.1.1 Methodology

The baseline and experimental hardware configurations of QCONS (Most Consumer in Queue) policy is the same as the one described in section 5.1.3.1 (QOLD policy).

7.1.1.2 Impact of super-set size and counter threshold

Figure 7-1 shows the impact of size of super-sets on both power and performance in QCONS policy. The same as in QOLD policy, the numbers in the plots are the power and performance in low-power cache relative to the baseline scheme. In the baseline scheme, there is no penalty in time. However, in the low-power configuration, there is one cycle penalty if a load/store instruction accesses a low supply voltage cell.

We set the criticality number of dependent instructions (C) to 3 and use threshold of 16 for criticality counters. We set the criticality interval to 100000 committed instructions and sweep super-set size from 1 to 64.

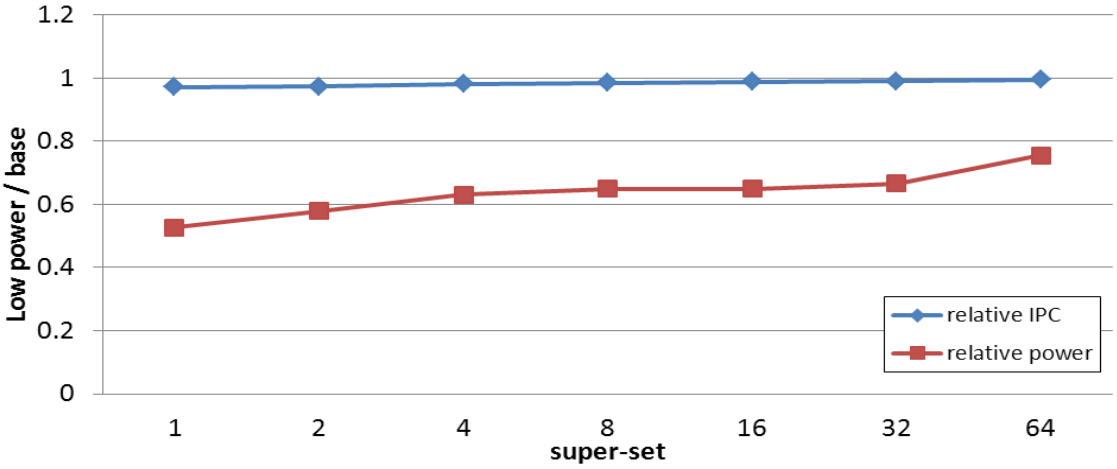


Figure 7-1: Impact of super-set size in Spec2000 benchmarks (QCONS) (C=3, threshold=16 and M=100000).

Figure 7-2 exhibits the impact of counter threshold on power and performance. As the threshold increases, power saving improves but performance reduces. The rationale is the same as those for QOLD policy. With a given super-set size, it is easier to find enough instructions to satisfy smaller threshold value. Therefore, the performance and power consumption are closer to the baseline scheme when threshold is smaller.

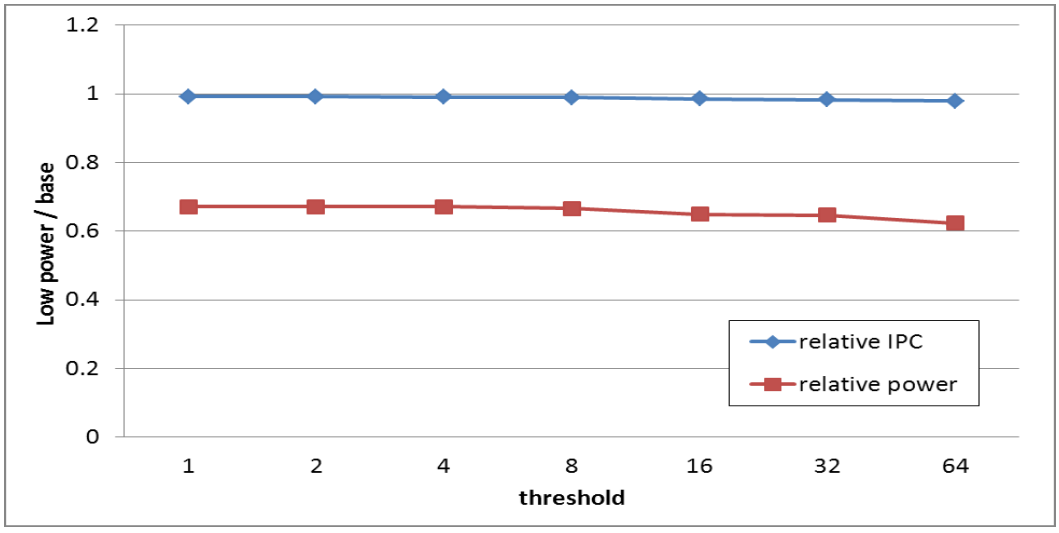


Figure 7-2: Impact of threshold on Spec2000 benchmarks (QCONS) (super-set=8 rows, C=3, M=100000).

7.1.1.3 Impact of criticality interval (M)

Similar to that of QOLD policy, we check counters after a certain number of instructions are committed. Figure 7-3 and 7-4 show the impact of criticality interval on power and performance, respectively. The interval changes from 1000 to 1000000 instructions. As the interval increases the power saving reduces. This is due to the fact that when interval increases it is more likely to have the counters exceeding threshold. As such, more super-sets are configured to use high Vdd. This reduces power saving and also reduces performance loss.

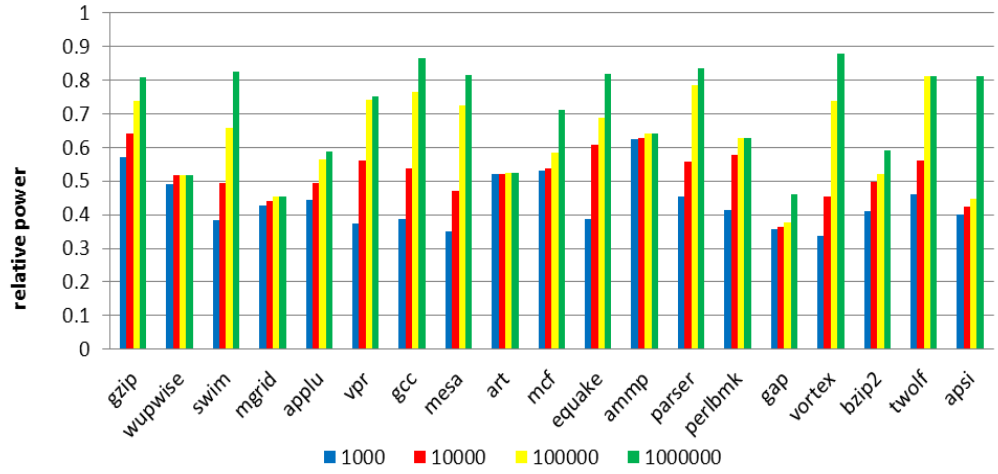


Figure 7-3: Power consumption in Spec2000 benchmarks when criticality interval changes (QCONS).

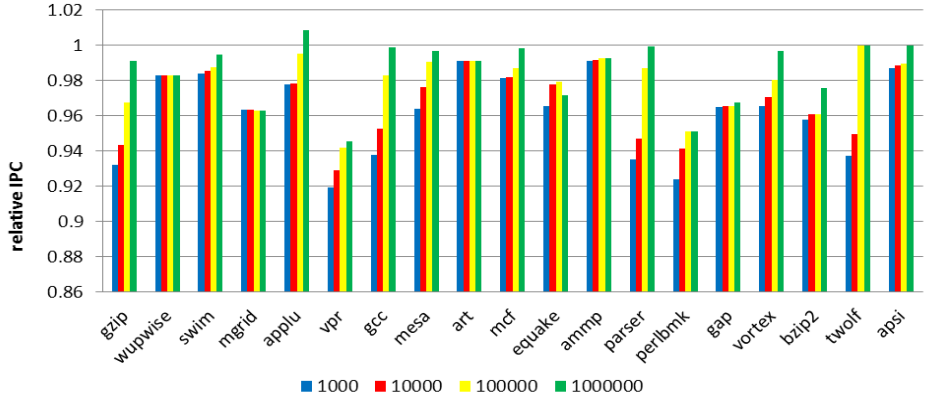


Figure 7-4: Performance in Spec2000 benchmarks when criticality interval changes (QCONS).

7.1.1.4 Impact of C

To classify instructions into critical and non-critical in QCONS policy, we count a Load/Store instruction's number of dependent instructions when it reaches to write back stage. If number of dependent instructions exceeds or equals to a pre-defined number (C), the Load/Store instruction is considered as critical. Figure 7-5 and 7-6 show the effect of *C* on power and performance. C changes from 1 to 5 in each benchmark. As C increases, as opposed to QOLD policy, it is *less* likely to mark a Load/Store instruction as critical. Hence, the chance that counters exceed threshold decreases when C increases. Therefore, when C increases, the performance is degraded and the power ratio is also lowered.

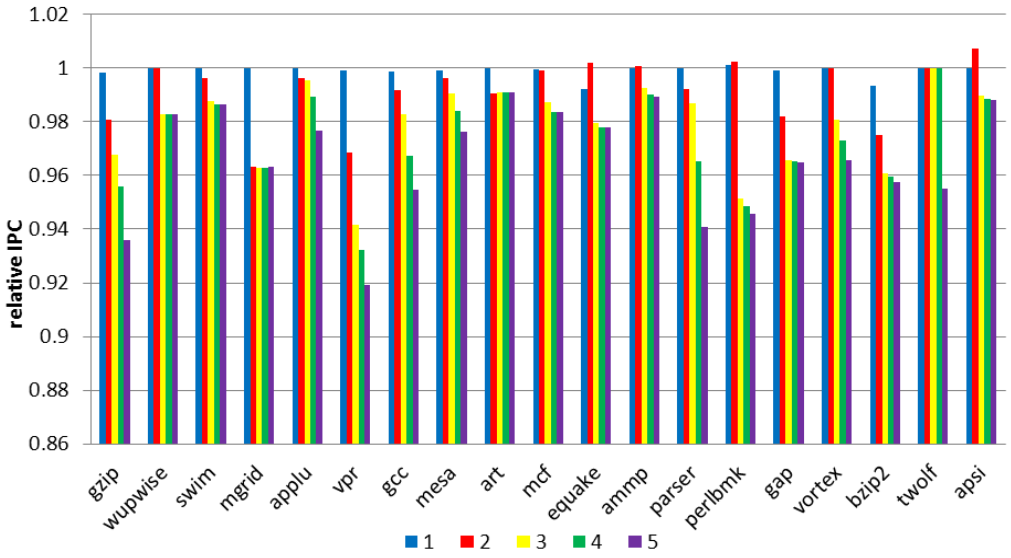


Figure 7-5: Power consumption in Spec2000 benchmarks when C increases (QCONS).

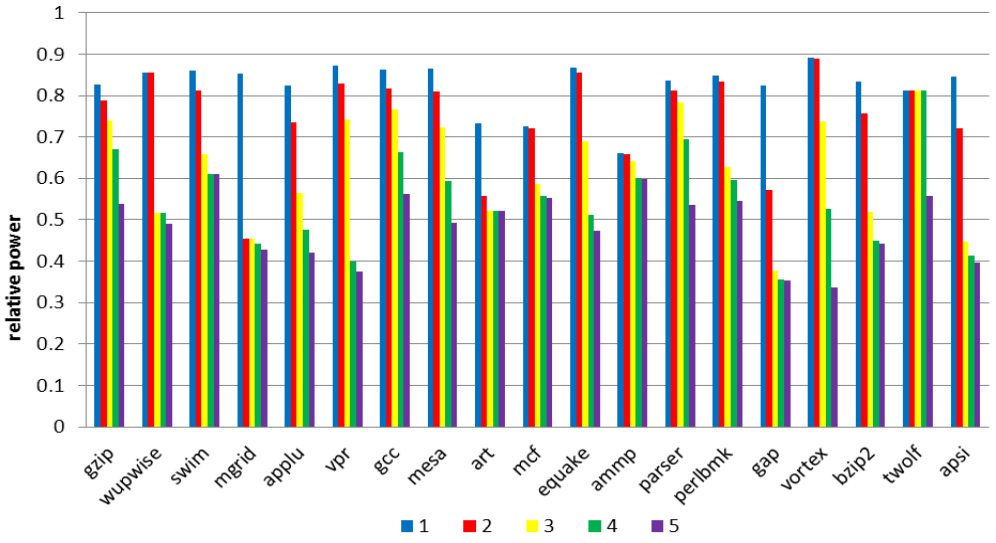


Figure 7-6: Performance in Spec2000 benchmarks when C increases (QCONS).

7.1.1.5 Distribution of number of dependent instructions

We also measure the distribution of dependent instructions across all benchmarks which are shown in Figure 7-7. It can be seen that averagely over 90% of load/store instructions have 1-2 dependent instructions.

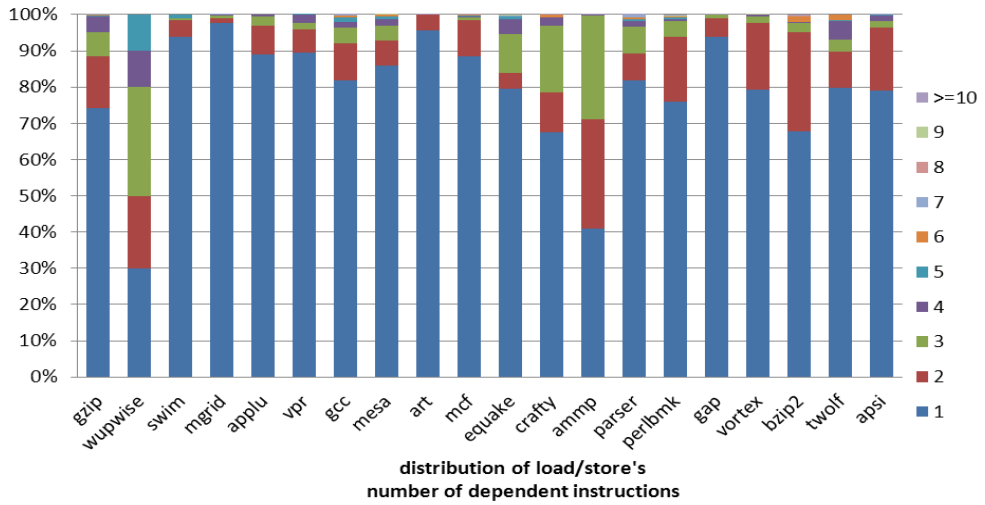


Figure 7-7: Distribution of load/store's number of dependent instructions (QCONS).

7.1.1.6 Summary

We have evaluated the effect of different parameters such as super-set size, counter threshold, criticality interval and C on both power and performance. We found that the optimum configuration is when super-set size is 8, counter threshold is 16, criticality interval is 100000 and C is 3. With this configuration, we sacrifice 1.5% of performance in exchange for about 35% on power saving.

7.1.2 Low Power Register Files

7.1.2.1 Methodology

The baseline and experimental hardware configurations of QCONS policy is the same as the one described in section 5.2.3.1 (QOLD policy).

7.1.2.2 Impact of super-set size and counter threshold

Figure 7-8 shows the impact of size of super-sets on both power and performance in QCONS policy. The same as in QOLD policy, the numbers in the plots are the power and performance in low-power cache relative to the baseline scheme. In the baseline scheme, there is no penalty in time. However, in the low-power configuration, there is one cycle penalty if an instruction accesses a low supply voltage cell.

We set the criticality number of dependent instructions (C) to 3 and use threshold of 16 for criticality counters. We set the criticality interval to 100000 committed instructions and sweep super-set size from 1 to 8.

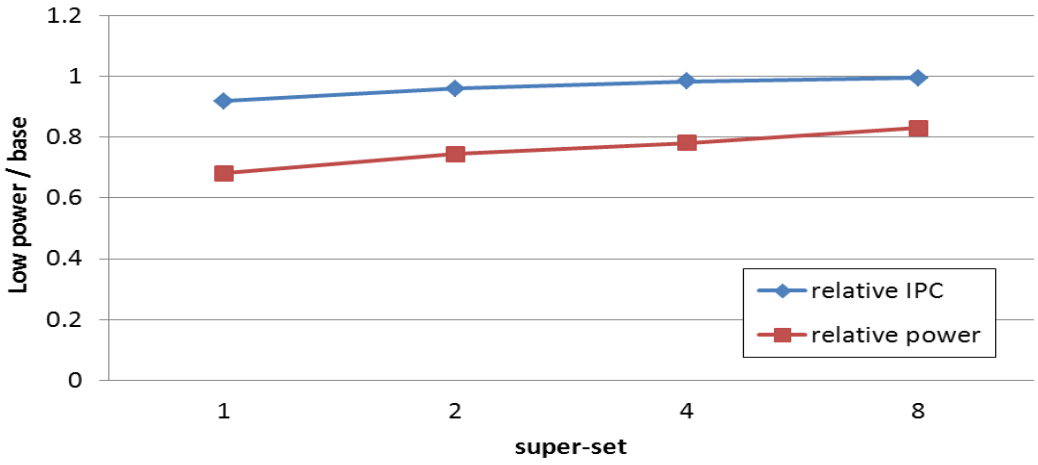


Figure 7-8: Impact of super-set size on power and performance of register files (QCONS) (C=3, threshold=16 and M=100000).

Figure 7-9 exhibits the impact of counter threshold on power and performance. As the threshold increases, power saving improves but performance reduces. The rationale is the same as those for QOLD policy. With a given super-set size, it is easier to find enough instructions to satisfy smaller threshold value. Therefore, the performance and power consumption are closer to the baseline scheme when threshold is smaller.

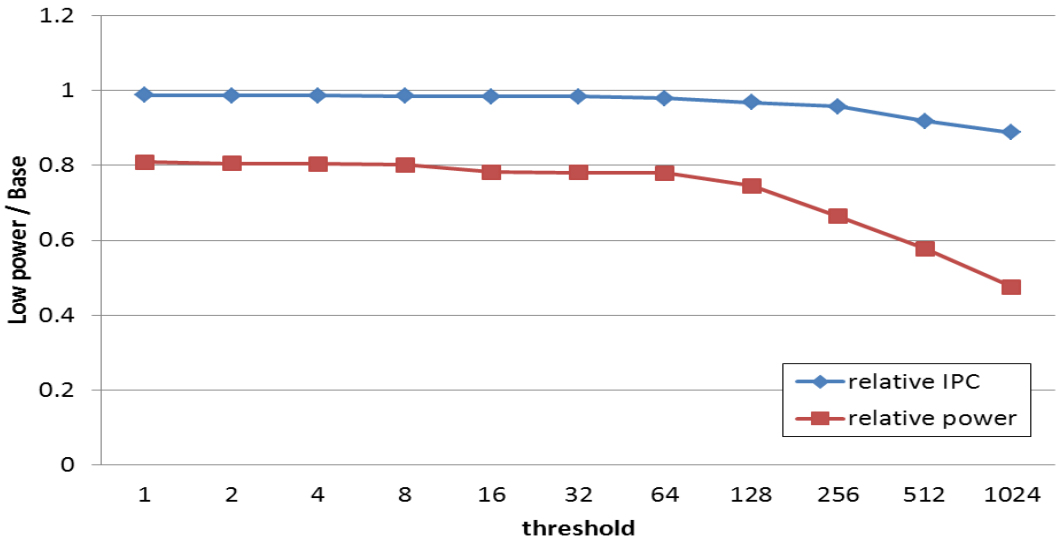


Figure 7-9: Impact of threshold on power and performance of register files (QCONS) (super-set=8 rows, C=3, M=100000).

7.1.2.3 Impact of criticality interval (M)

Similar to that of QOLD policy, we check counters after a certain number of instructions are committed. Figure 7-10 and 7-11 show the impact of criticality interval on power and performance, respectively. The interval changes from 1000 to 1000000 instructions. As the interval increases the power saving reduces. This is due to the fact that when interval increases it is more likely to have the counters exceeding threshold. As such, more super-sets are configured to use high Vdd. This reduces power saving and also reduces performance loss.

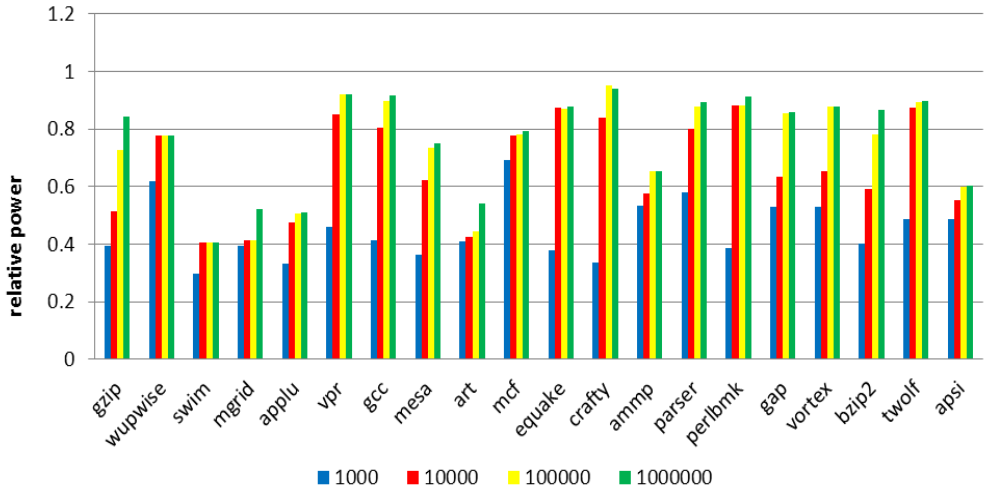


Figure 7-10: Power consumption of register files in Spec2000 benchmarks when criticality interval changes (QCONS).

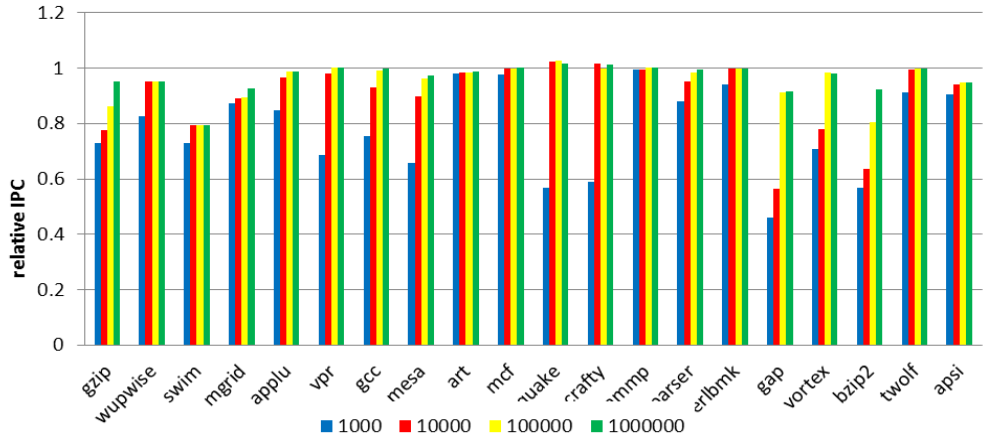


Figure 7-11: Performance of register files in Spec2000 benchmarks when criticality interval changes (QCONS).

7.1.2.4 Impact of C

To classify instructions into critical and non-critical in QCONS policy, we count an instruction's number of dependent instructions when it gets to write back stage. If number of dependent instructions exceeds or equals to a pre-defined number (C), the instruction is considered as critical. Figure 7-12 and 7-13 show the effect of *C* on power and performance. C changes from 1 to 5 in each benchmark. As C increases, as opposed to QOLD policy, it is *less* likely to mark an instruction as critical. Hence, the chance that counters exceed threshold decreases when C increases. Therefore, when C increases, the performance is degraded and the power ratio is also lowered.

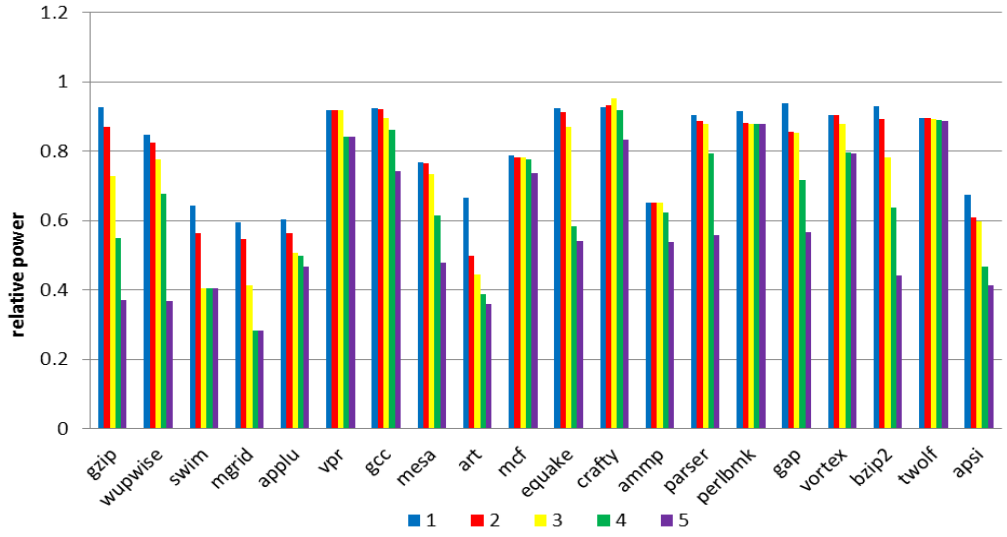


Figure 7-12: Power consumption of register files in Spec2000 benchmarks when C increases (QCONS).

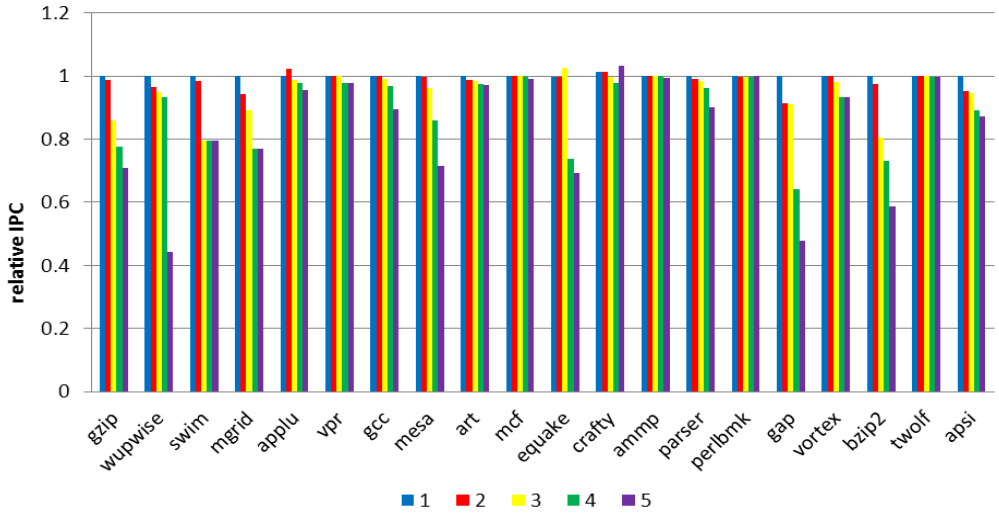


Figure 7-13: Performance of register files in Spec2000 benchmarks when C increases (QCONS).

7.1.2.5 Distribution of number of dependent instructions

We also measure the distribution of dependent instructions across all benchmarks for both integer register file and floating point register file. Figure 7-14 shows the distribution of integer registers and Figure 7-15 shows that of floating point registers. In Figure 7-15, those benchmarks without any stack percentage bar such as gzip and mcf, etc. don't use floating

point registers. Figure 7-16 shows the distribution of dependent integer registers vs floating point registers.

Figure 7-14: Distribution of number of dependent integer registers.

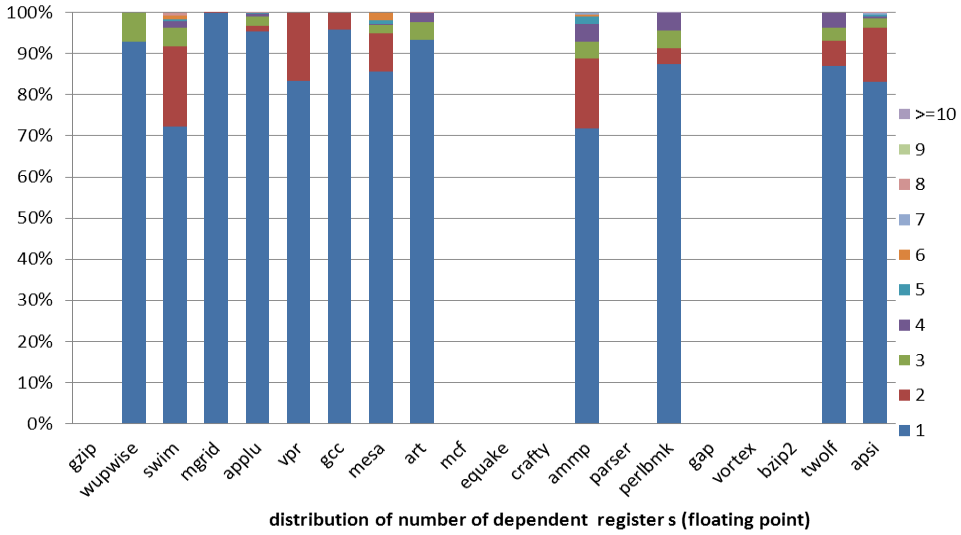


Figure 7-15: Distribution of number of dependent floating point registers.

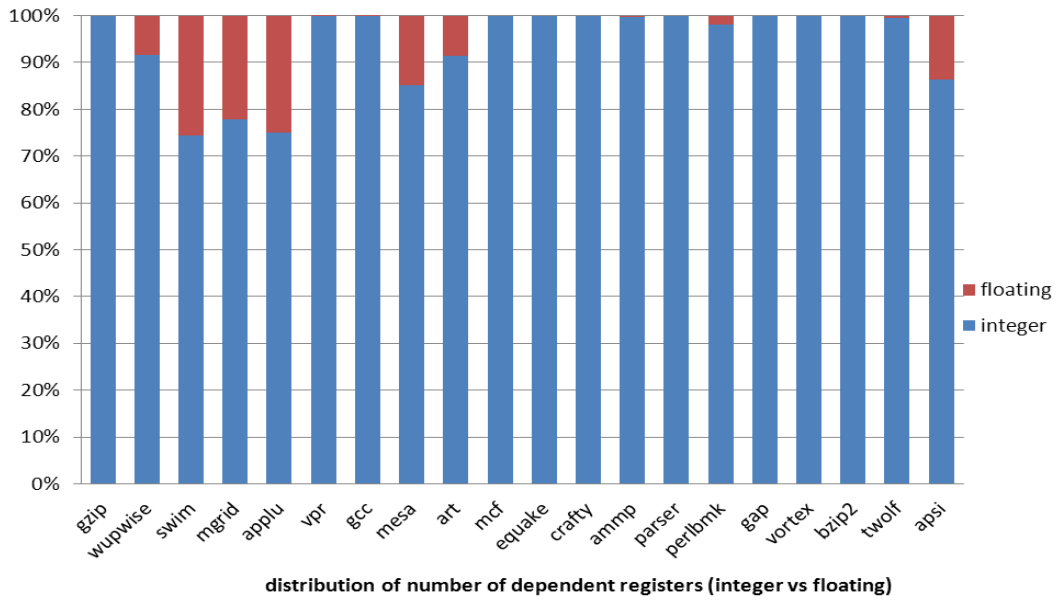


Figure 7-16: Distribution of dependent registers (integer vs floating point).

7.1.2.6 Summary

Based on simulation results, when super-set size is 4, counter threshold is 16, criticality interval is 100000 and C is 3, performance is reduced by 1.5% and power is improved by 22%; integer and floating point registers combined.

7.2 Methodology and Results (FREED)

7.2.1 Low Power Cache

7.2.1.1 Methodology

The baseline and experimental hardware configurations of FREED (Number of instructions are freed) policy is the same as the one described in section 5.1.3.1 (QOLD policy).

7.2.1.2 Impact of super-set size and counter threshold

Figure 7-17 shows the impact of size of super-sets on both power and performance in FREED policy. The same as in QOLD policy, the numbers in the plots are the power and

performance in low-power cache relative to the baseline scheme. In the baseline scheme, there is no penalty in time. However, in the low-power configuration, there is one cycle penalty if a load/store instruction accesses a low supply voltage cell.

We set the criticality number of freed instructions (F) to 2 and use threshold of 8 for criticality counters. We set the criticality interval to 10000 committed instructions and sweep super-set size from 1 to 64.

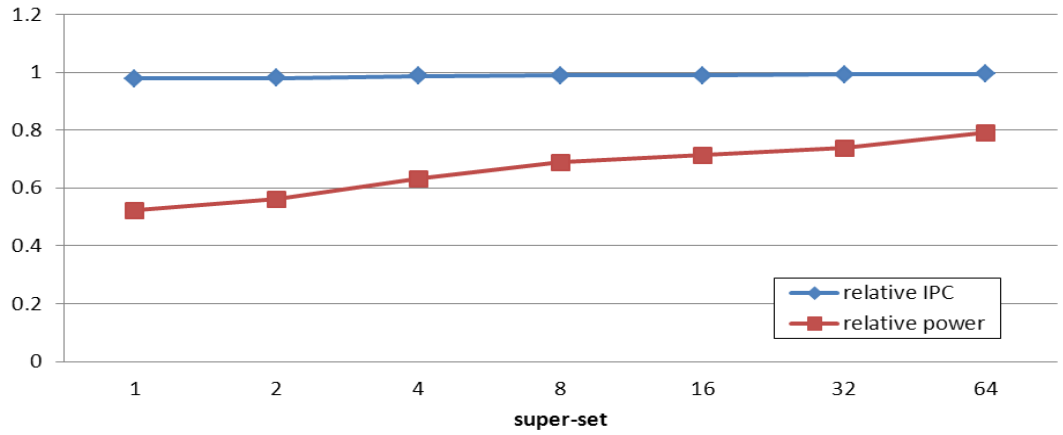


Figure 7-17: Impact of super-set size in Spec2000 benchmarks (FREED) (F=2, threshold=8 and M=10000).

Figure 7-18 exhibits the impact of counter threshold on power and performance. As the threshold increases, power saving improves but performance reduces. The rationale is the same as those for QOLD policy. With a given super-set size, it is easier to find enough instructions to satisfy smaller threshold value. Therefore, the performance and power consumption are closer to the baseline scheme when threshold is smaller.

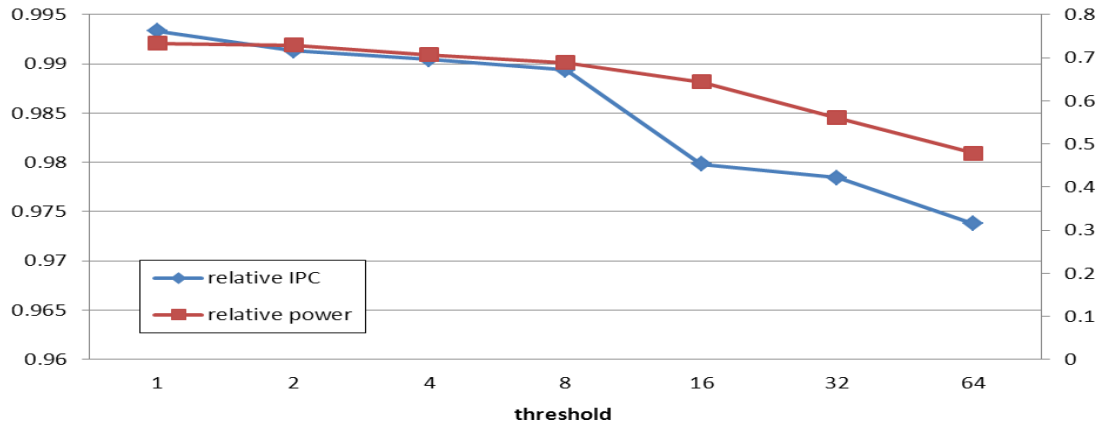


Figure 7-18: Impact of threshold on Spec2000 benchmarks (FREED) (super-set=8 rows, F=2, M=10000).

7.2.1.3 Impact of criticality interval (M)

Similar to that of QOLD policy, we check counters after a certain number of instructions are committed. Figure 7-19 and 7-20 show the impact of criticality interval on power and performance, respectively. The interval changes from 1000 to 1000000 instructions. As the interval increases the power saving reduces. This is due to the fact that when interval increases it is more likely to have the counters exceeding threshold. As such, more super-sets are configured to use high Vdd. This reduces power saving and also reduces performance loss.

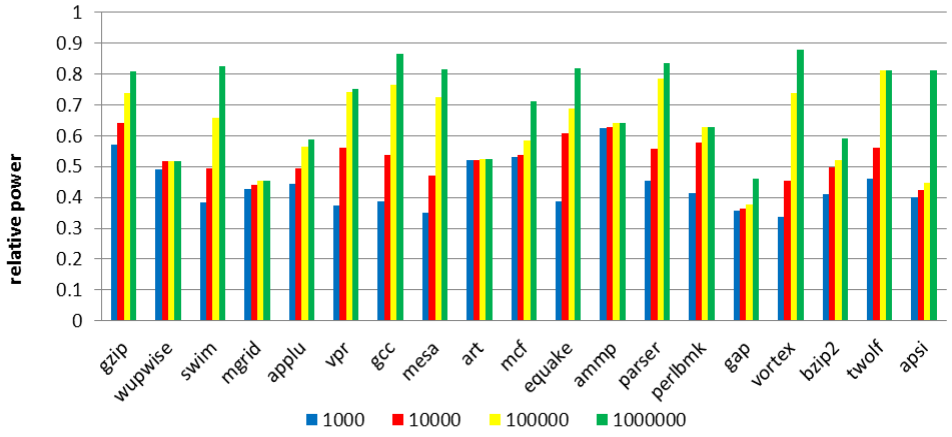


Figure 7-19: Power consumption in Spec2000 benchmarks when criticality interval changes (FREED).

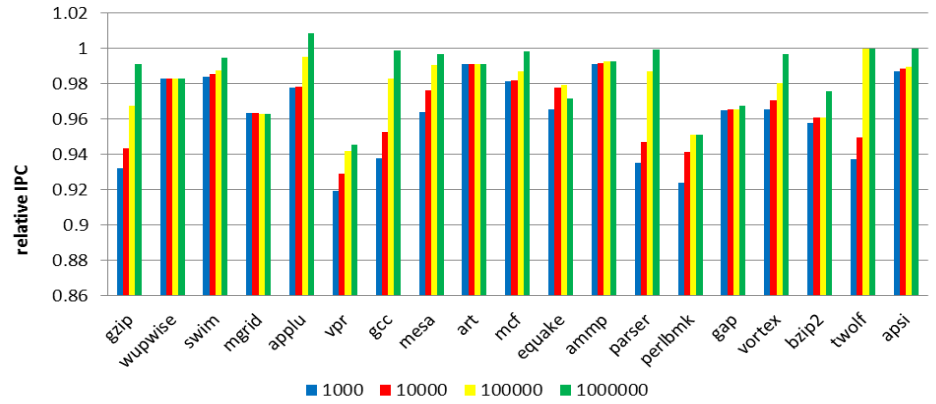


Figure 7-20: Performance in Spec2000 benchmarks when criticality interval changes (FREED).

7.2.1.4 Impact of F

To classify instructions into critical and non-critical in FREED policy, we count a Load/Store instruction's number of freed instructions when it gets to write back stage. If number of freed instructions exceeds or equals to a pre-defined number (F), the Load/Store instruction is considered as critical. Figure 7-21 and 7-22 show the effect of F on power and performance. F changes from 1 to 5 in each benchmark. As F increases, as opposed to QOLD policy, it is *less* likely to mark a Load/Store instruction as critical. Hence, the chance that counters exceed threshold decreases when F increases. Therefore, when F increases, the performance is degraded and the power ratio is also lowered.

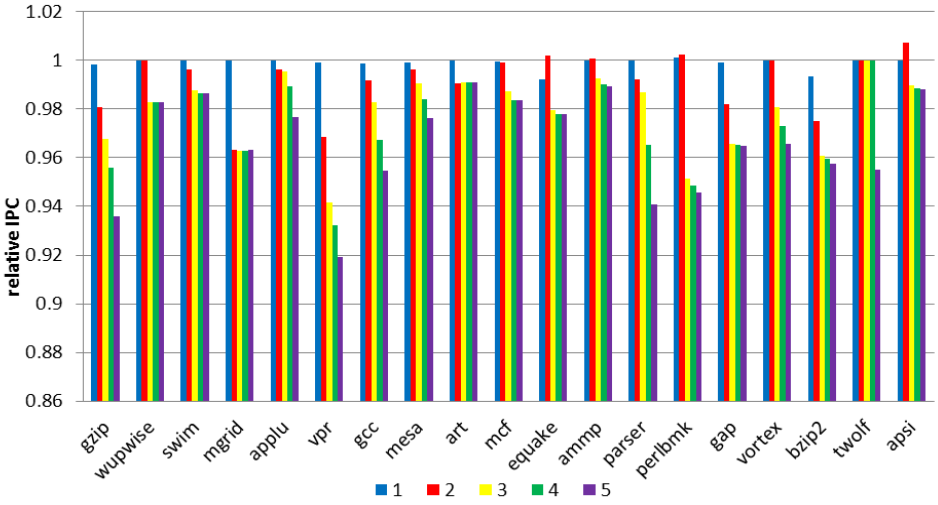


Figure 7-21: Power consumption in Spec2000 benchmarks when F increases (FREED).

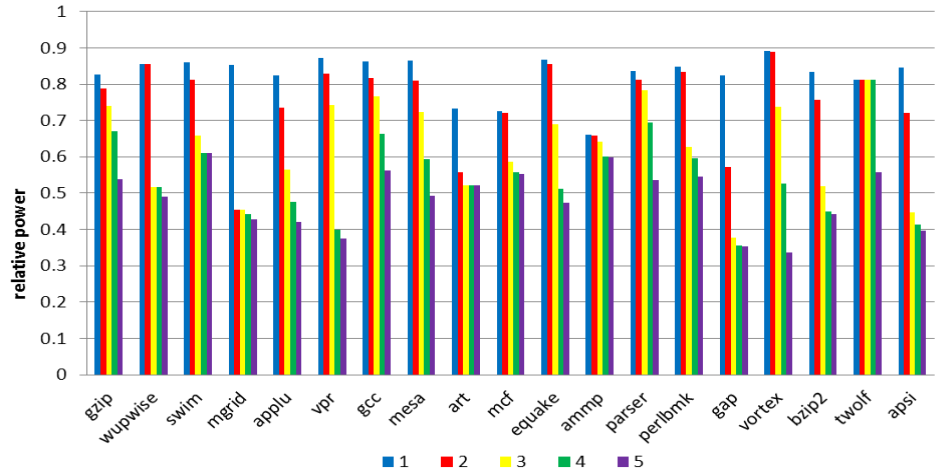


Figure 7-22: Performance in Spec2000 benchmarks when F increases (FREED).

7.2.1.5 Distribution of number of dependent instructions

We also measure the distribution of dependent instructions across all benchmarks which are shown in Figure 7-23. It can be seen that averagely over 90% of load/store instructions have 1-2 freed instructions.

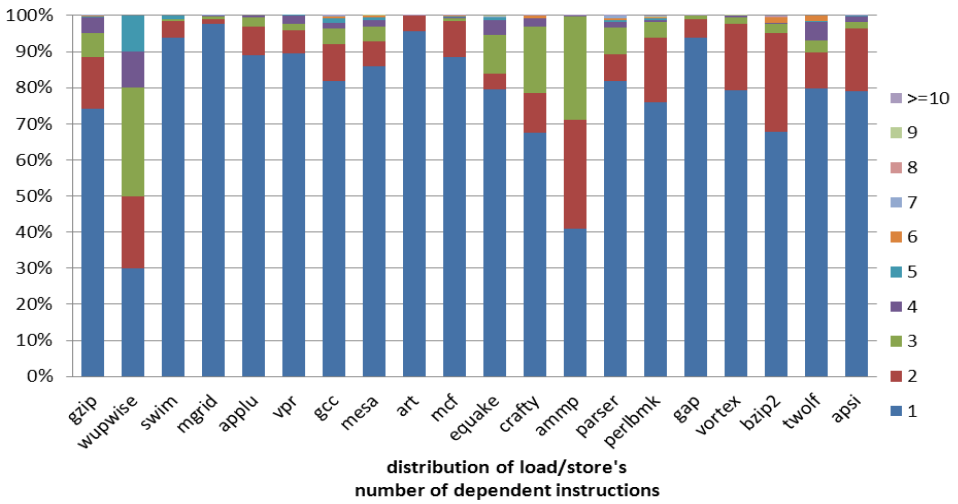


Figure 7-23: Distribution of load/store's number of dependent instructions (FREED).

7.2.1.6 Summary

We have evaluated the effect of different parameters such as super-set size, counter threshold, criticality interval and C on both power and performance. We found that the optimum configuration is when super-set size is 8, counter threshold is 8, criticality interval is 10000 and F is 2. With this configuration, we sacrifice ~1% of performance in exchange for about ~31% on power saving.

7.2.2 Low Power Register Files

7.2.2.1 Methodology

The baseline and experimental hardware configurations of FREED policy is the same as the one described in section 5.2.3.1 (QOLD policy).

7.2.2.2 Impact of super-set size and counter threshold

Figure 7-24 shows the impact of size of super-sets on both power and performance in FREED policy. The same as in QOLD policy, the numbers in the plots are the power and performance in low-power cache relative to the baseline scheme. In the baseline scheme,

there is no penalty in time. However, in the low-power configuration, there is one cycle penalty if an instruction accesses a low supply voltage cell.

We set the criticality number of freed instructions (F) to 2 and use threshold of 16 for criticality counters. We set the criticality interval to 10000 committed instructions and sweep super-set size from 1 to 8.

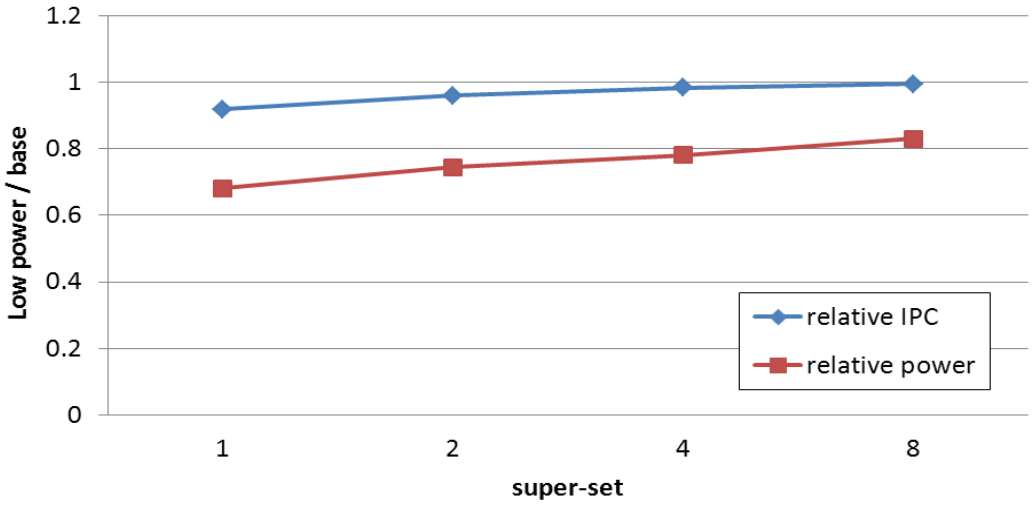


Figure 7-24: Impact of super-set size on power and performance of register files (FREED) (F=2, threshold=16 and M=10000).

Figure 7-25 exhibits the impact of counter threshold on power and performance. As the threshold increases, power saving improves but performance reduces. The rationale is the same as those for QOLD policy. With a given super-set size, it is easier to find enough instructions to satisfy smaller threshold value. Therefore, the performance and power consumption are closer to the baseline scheme when threshold is smaller.

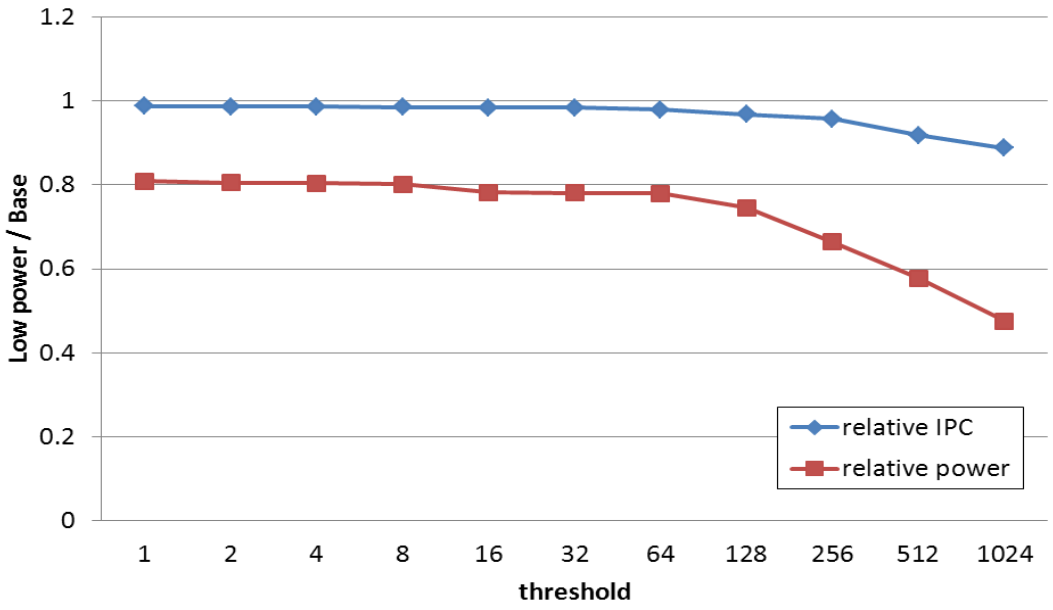


Figure 7-25: Impact of threshold on power and performance of register files (FREED) (super-set=8 rows, F=2, M=10000).

7.2.2.3 Impact of criticality interval (M)

Similar to that of QOLD policy, we check counters after a certain number of instructions are committed. Figure 7-26 and 7-27 show the impact of criticality interval on power and performance, respectively. The interval changes from 1000 to 1000000 instructions. As the interval increases the power saving reduces. This is due to the fact that when interval increases it is more likely to have the counters exceeding threshold. As such, more super-sets are configured to use high Vdd. This reduces power saving and also reduces performance loss.

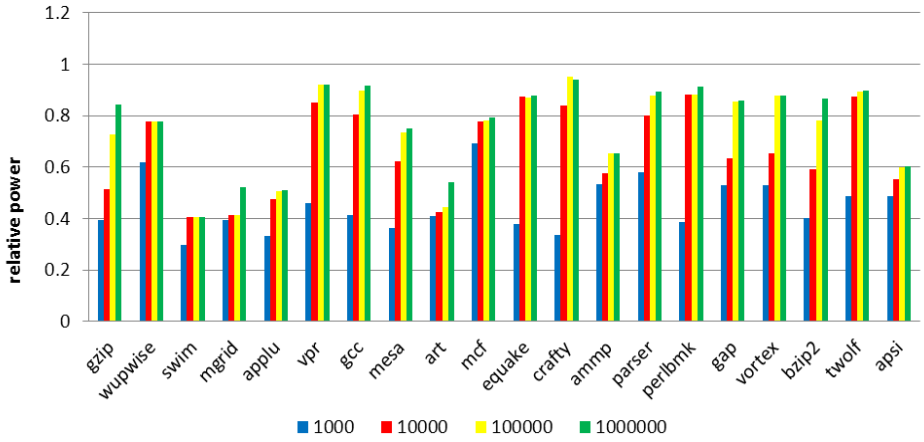


Figure 7-26: Power consumption of register files in Spec2000 benchmarks when criticality interval changes (FREED).

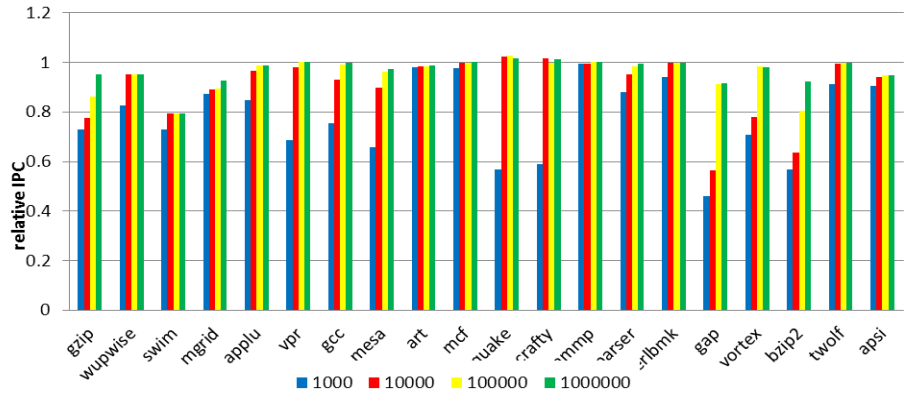


Figure 7-27: Performance of register files in Spec2000 benchmarks when criticality interval changes (FREED).

7.2.2.4 Impact of F

To classify instructions into critical and non-critical in FREED policy, we count an instruction's number of freed instructions when it gets to write back stage. If number of freed instructions exceeds or equals to a pre-defined number (F), the instruction is considered as critical. Figure 7-28 and 7-29 show the effect of F on power and performance. F changes from 1 to 5 in each benchmark. As F increases, as opposed to QOLD policy, it is *less* likely to mark an instruction as critical. Hence, the chance that counters exceed threshold

decreases when F increases. Therefore, when F increases, the performance is degraded and the power ratio is also lowered.

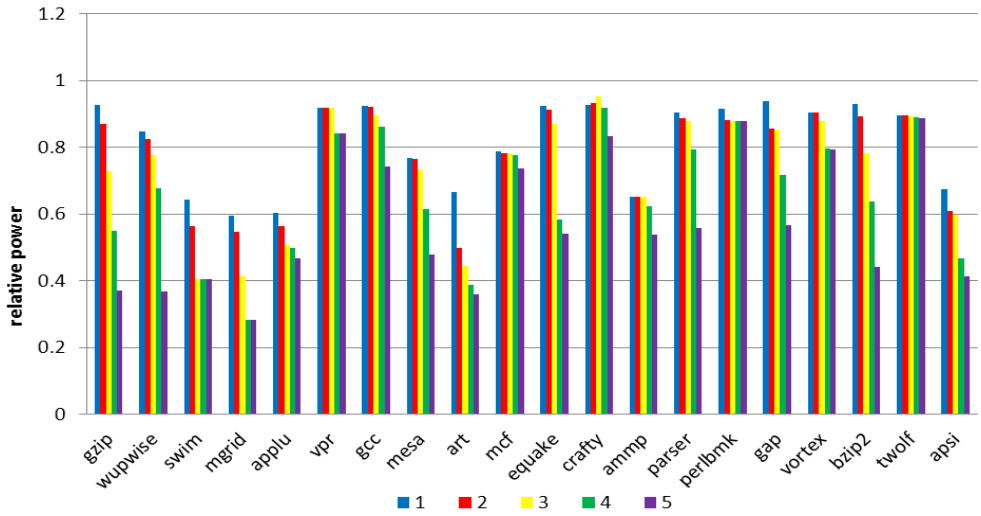


Figure 7-28: Power consumption of register files in Spec2000 benchmarks when F increases (FREED).

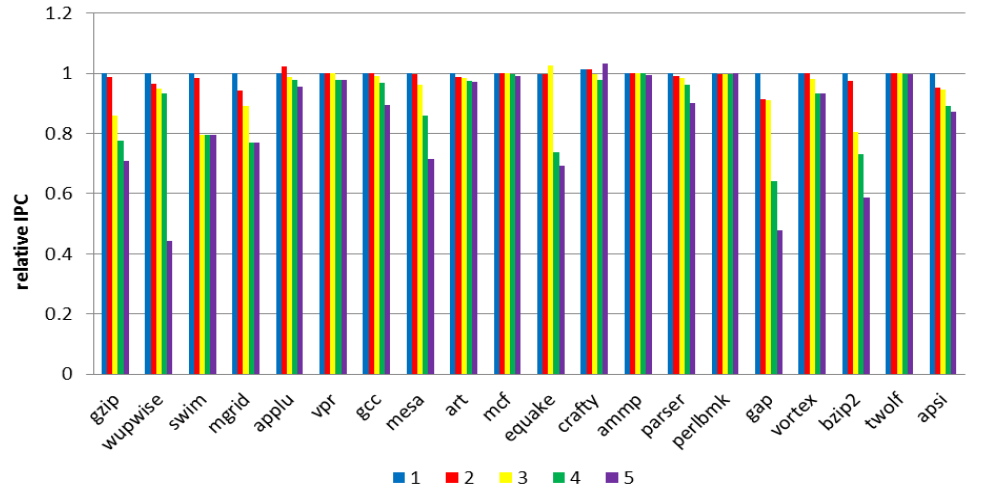


Figure 7-29: Performance of register files in Spec2000 benchmarks when F increases (FREED).

7.2.2.5 Distribution of number of dependent instructions

We also measure the distribution of freed instructions across all benchmarks for both integer register file and floating point register file. Figure 7-30 shows the distribution of integer

registers and Figure 7-31 shows that of floating point registers. In Figure 7-31, those benchmarks without any stack percentage bar such as gzip and mcf, etc. don't use floating point registers. Figure 7-32 shows the distribution of dependent integer registers vs floating point registers.

Figure 7-30: Distribution of number of freed integer registers.

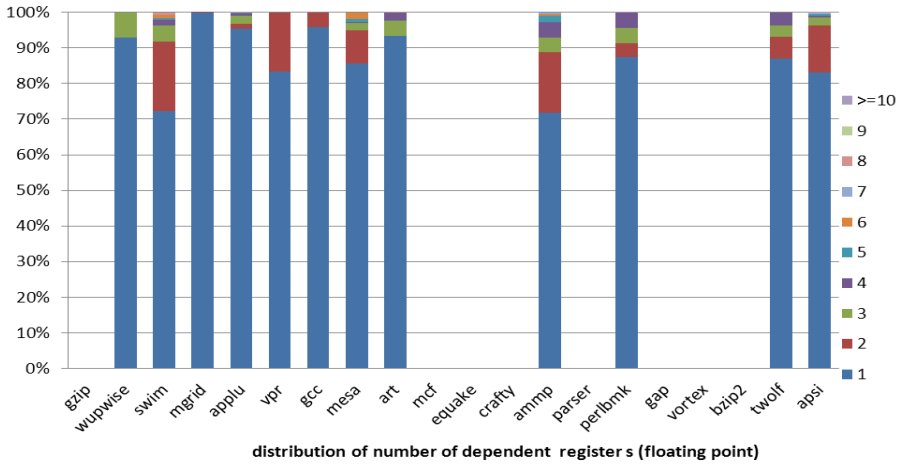


Figure 7-31: Distribution of number of freed floating point registers.

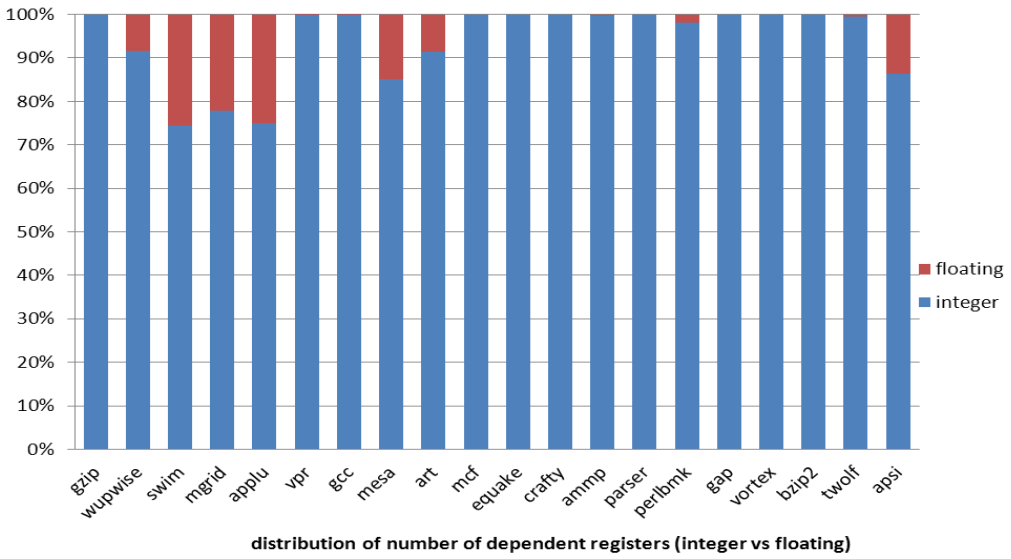


Figure 7-32: Distribution of freed registers (integer vs floating point).

7.2.2.6 Summary

Based on simulation results, when super-set size is 4, counter threshold is 16, criticality interval is 10000 and F is 2, performance is reduced by 1.5% and power is improved by 22%; integer and floating point registers combined.

References

[1] R. Stevenson. Changing the Transistor Channel. IEEE Spectrum, pages 34-39, July 2013.

[2] M. Gowan, L. Biro, and D. Jackson. Power Consideration in the Design of the Alpha 21264 Microprocessor. In 35th Design Automation Conference, pages 726-731, June 1998.

[3] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In Proceedings of the Seventh International Symposium on High Performance Computer Architecture, pages 185–196, January 2001.

[4] Arizona State University. Predictive Technology Model. URL: http://ptm.asu.edu/modelcard/HP/32nm_HP.pm, 5 April, 2009

[5] H. Yang, R. Govindarajan, G. Gao, and K. Theobald. Power-Performance Trade-offs for Energy-Efficient Architectures: A Quantitative Study. Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02), pp.174-179, September 16-18, 2002

[6] J. Seng, E. Tune, D. Tullsen, and G. Cai. Reducing Processor Power with Critical Path Prediction. In proceedings of MICRO-34, Dec 2001.

[7] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98), pages 64–69, New York, Aug. 10–12 1998. ACM Press.

[8] B. Fisk and I. Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency. In IEEE International Conference on Computer Design, October 1999.

[9] R. Balasubramonian, V. Srinivasan, and S. Dwarkadas. Hot-and-Cold: Using Criticality in the Design of Energy-Effcient Caches. In Workshop on Power-Aware Computer Systems, in conjunction with MICRO-36, December 2003.

[10] V. Agrawal. CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies. Springer Science, 2008.

[11] T. Haraszti. CMOS Memory Circuits, Kluwer Academic Publishers, 2000.

[12] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction. In The 34th Annual IEEE/ACM International Symposium on Microarchitecture, 2001.

[13] S. Yang, B. Falsafi, M. Powell, K. Roy, T. Vijaykumar, An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches, Proceedings of

the 7th International Symposium on High-Performance Computer Architecture, p.147, January 20-24, 2001.

[14] SPEC Benchmark Suite. Information available at http://www.spec.org.

[15] D. Brooks, V. Tiwari M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", In Proceedings of International Symposium on Computer Architecture, 2000.

[16] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in: 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.

[17] H. Hanson, M. S. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger. Static Energy Reduction Techniques for Microprocessor Caches. 2001 International Conference on Computer Design, September 2001.

[18] B. Razavi. Design of Analog CMOS Integrated Circuits. 2001.

[19] J. Kin, G. Gupta, W. Mangione-Smith. The filter cache: an energy efficient memory structure. In proceedings of the 30th Annual International Symposium on Microarchitecture, pages 184-193, 1997

[20] C. Su, A. Despain. Cache designs for energy efficiency. In proceedings of the 28th Hawaii International Conference on System Sciences, January 1995.

[21] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass aware instruction scheduling for register file power reduction. volume 41, pages 173–181, June 2006.

[22] J. H. Tseng and K. Asanovic, "Energy-efficient register access," in Proc. 13th Symp. Integr. Circuits Syst. Des. (SBCCI), 2000, pp. 377–382.

[23] I. Bennour, E. Albouhamid. Lower bounds on the iteration time and the initiation interval of functional pipelining and loop folding. Design Autom. for Emb. Sys. 1(4):333-355 (1996)

[24] Werner Bucholz, ed., Planning a Computer System. New York: McGraw-Hill, 1962.

[25] Class note, ENGI-5231, Computer Architecture, Fall 2012.

[26] J. Smith, G. Sohi, The Microarchitecture of Superscalar Processors. Aug. 1995.

[27] Patterson D., Anderson T. et al.: A Case for Intelligent RAM: IRAM. IEEE Micro (1997)

[28] http://www.csc.lsu.edu/~busch/courses/comparch/spring2013/slides/appendixB.ppt

[29] J. Abella and A. Gonzalez. Power Efficient Data Cache Designs. In Proceedings of ICCD-21, Oct 2003.

[30] A. Chandrakasan, W. Bowhill, F. Fox, editors. The Design of High Performance Microprocessor Circuits, p.290, IEEE Press, 2001.

[31] J. L. Ayala and A. Veidenbaum. Reducing Register File Energy Consumption using Compiler Support. In Workshop on Application-Specific Processors, 2002.

[32] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, Mudge T. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In ISCA, 2002.

[33] X. Guan and Y. Fei, ‘‘Reducing Power Consumption of Embedded Processors through Register File Partitioning and Compiler Support,’’ Proc. 2008 Int’l Conf. Application-Specific Systems, Architectures and Processors, IEEE CS Press, 2008, pp. 269-274.

[34] W, Shieh, and C. Chen. Saving Register-File Leakage Energy by Register-usage Exploiting, Journal of Information Science and Engineering 24, 1429-1444, 2008.

[35] N. Gong, and J. Wang, R. Sridhar. Low power tri-state register files design for modern out-of-order processors. SOC Conference (SOCC), 2011 IEEE International. IEEE, 2011.

[36] A. Agarwal, H. Li, and K. Roy, “DRG-Cache: A Data Retention Gated Ground Cache for Low Power”, Design Automation Conf., June 2002.

[37] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high performance circuits. In IEEE Symposium on VLSI Circuits, pages 40–41, 1998.

[38] A. Kang, Y.Leblebici. CMOS Digital Integrated Circuits 2nd. McGraw-Hill, 1999

[39] http://bgu.uniclass.co.il/ee08/uploader.php?file=Lecture06+-+SRAM.pdf&id=73

[40] http://soc.cs.nchu.edu.tw/pllai/NCUT/95(%E4%B8%80)/%5B02%5D%20VLSI_PPT/%5B12%5D%20Chapter13_Memory%20and%20Programmable%20Logic.ppt

[41] K. Koo, J. Seo, M. Ko, J. Kim. “A new level-up shifter for high speed and wide range interface in ultra deep sub-micron”, 2005. ISCAS 2005. IEEE International Symposium on Circuits and Systems

[42] J. L. Ayala, M. López-Vallejo, A. Veidenbaum, and C. A. López, Energy Aware Register File Implementation Through Instruction Predecode, International Conference on Application-Specific Systems, Architectures and Processors (2003).

[43] http://www.nature.com/nature/journal/v479/n7373/fig_tab/nature10676_F3.html

[44] http://spectrum.ieee.org/semiconductors/processors/transistor-aging

[45] http://www.singularity.com/charts/page61.html

[46] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, 5th ed., 2011

[47] http://en.wikipedia.org/wiki/Branch_predictor

[48] J. Hennessy, D. Patterson, Computer Organization and Design Hardware /Software Interface. 3rd ed., 2007.

[49] http://www.edn.com/design/systems-design/4397051/2/Memory-Hierarchy-Design-part-1

[50] M. Abdel-Majeed and M. Annavaram, Warped Register File: A Power Efficient Register File for GPGPUs, Proc. IEEE HPCA, 2013.

[51] B. Batson, T, Vihaykumar, Reactive-Associative Caches, Int. Conf. on Parallel Architectures and Compilation Techniques, Sep. 2001.