

Chapter 1

Introduction

Nowadays, portable devices such as laptop and notebook computers are very popular. These devices require energy efficient design in order to maximize battery lifetime. Reducing the power consumption of microprocessors has become increasingly important. Many studies have shown that memory accesses account for a noticeably large percentage of the total power consumption in microprocessors, making the power consumption of caches and main memory an important concern [13].

Caches are a significant part of the processor due to the increasing disparity between processor cycle time and memory access time. High performance microprocessors normally have one or two levels of on-chip cache in order to reduce the off-chip traffic as much as possible. Off-chip accesses are not only at least a magnitude slower but also dissipate a large amount of power via highly capacitive I/O pads. Thus, caches are important not only for high performance, but also for low power to help reduce the amount of off-chip communication.

The power dissipated by the on-chip cache itself is often a significant part of the power dissipated by the entire microprocessor. For example, in the StrongARM 110 from DEC and the Power PC from IBM, cache power consumption is either the largest or second largest power-consuming block [13]. In the StrongARM CPU which has the current best SPECmarks/watt rating, 43% of total power is dissipated in the on-chip caches [11]. Another example is the

DEC 21164 microprocessor, whose on-chip cache dissipates 25% of the total power [11]. Hence, to achieve an energy efficient design, it is necessary to reduce the power dissipation in the on-chip caches.

In a typical processor with a split cache architecture, the instruction cache (I-cache) consumes more power than the data cache (D-cache) because the I-cache is accessed for each instruction while the D-cache is accessed only for loads and stores. Since around 25-30% of the executed instructions are loads and stores, the activity of the D-cache is around 25-30% of the activity of the I-cache. Clearly, the I-cache is an attractive target to reduce power consumption.

This thesis, therefore, focuses on reducing the power consumption of the level-1 I-cache by using an in-cache instruction compression technique that uses gated wordlines to reduce the number of bits read for compressed instructions. To accurately estimate cache power, we have developed a cache power consumption model as well as a cache simulator. The analytical model for estimating cache power consumption is based on a cache power consumption model and run time statistics of cache, such as hit/miss rates from a cache simulation. Next, we investigated the effectiveness of two architectural techniques, from previous work, in reducing the power consumed in the I-cache. These two techniques are sub-banking and reducing the frequency of tag compare. We then proposed to reduce the power dissipated in the I-cache, by using gated wordlines to reduce the number of bitline swings per instruction. We evaluated two versions which compress instructions using 2 sizes or 3 sizes. Instead of fetching out the 32 bits fixed-length for all instructions, the 2-size approach uses a gated wordline to read out either a compressed 23-bit medium instruction or an uncompressed 33-bit long instruction. The 3-size approach uses gated wordline to read out either a compressed 17-bit short instruction, a compressed 23-bit medium instruction or an uncompressed 34-bit long instruction. Hence, these two methods allow us to reduce power dissipated from reading or writing the unnecessary bits of the instructions that do not require full 32 bits, thereby reducing the power consumed in the SRAM array of the I-cache. We demonstrate our techniques by applying them to the MIPS-II instruction set. Our dynamic compression for programs in SPECInt95 achieves an average reduction in bits read of 23.73% in the 2-size approach and 29.10% in the 3-size approach.

An overview of the thesis

Chapter 2 is a review of cache structure and operation. We describe the implementation of our cache power consumption model. The main energy dissipating components in SRAM are identified.

Chapter 3 reviews previous work on low power cache. We have selected two techniques, sub-banking and reducing the frequency of tag compares because they both decrease the power consumed in the I-cache by reducing the power dissipated from the SRAM array. Thus, these two techniques can combine with our technique of using the gated wordline to further reduce the power dissipated from the cell array of the I-cache.

Chapter 4 discusses background and motivation in the study of using an in-cache instruction compression technique that uses gated wordlines to reduce the number of bitline swings.

Chapter 5 evaluates the two design methods, the 2-size approach and the 3-size approach, in our technique of using the gated wordlines to reduce the number of bitline swing.

Chapter 6 concludes the thesis.

Chapter 2

Cache Review and Modeling Power Consumption

A cache is a buffer between a fast processor and slow memory. The disparity in speed growth between processor and memory has led to the processor being far faster than the memory. The idea behind caching is to prevent the processor from wasting processor cycles while waiting for information from the memory. Hence, by having recently used data held in a small region of fast memory, the processor can usually get the information it needs quickly and only infrequently access slower main memory.

There are several layers of cache in a modern computer. Each layer acts as a buffer for the next lower level. In this thesis we will concentrate only on level-1 cache, since it is the cache closest to processor. In fact, it is usually built directly onto the processor die itself and runs at the same speed as the processor.

The level-1 cache can be either a unified cache or a split cache. A unified cache is a single cache which handles both instructions and data. A split cache separates instructions from data. In a split cache architecture, the instruction cache is accessed every instruction while the data cache is accessed only for loads and stores. As only 25-30% of all instructions in a typical RISC program are loads and stores, the activity of D-cache is only 25-30% of the activity of I-cache. Therefore, in this thesis will focus on reducing power consumption in the I-cache.

2.1 Cache Review

2.1.1 Review of Cache Organization

Memory transfers the information to the cache in a chunk called a ‘block’. A ‘block’, therefore, is the minimum unit of information that can be present in the cache (hit in the cache) or not (miss in the cache) [17]. Typically there is more than one word in a cache block. For example, in a RISC machine with 32-bit instructions, a cache with 32-byte block size has total of 256 bits in a block. Hence, the block has eight 32-bit instructions or eight words. Each cache block has a tag and a valid bit added to the tag. The valid bit indicates whether the cache block contains valid information. If the valid bit is not set, the address information in that entry is invalid and there can not be a match in that address.

Three types of cache organization are created by restrictions on where a block is placed [17]. If there is only one place that a block can be placed in the cache, the cache is ‘direct mapped’. If a block can appear anywhere in the cache, the cache is ‘fully associative’. If a block is first mapped onto a group of blocks, called set, and can then be placed anywhere within the set, the cache is ‘set associative’.

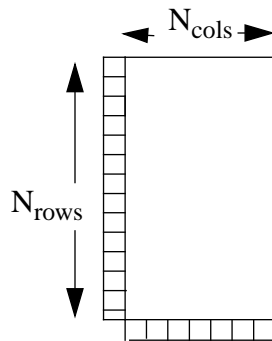


Figure 2.1: Logical organization of cache

Cache parameters are as following:

S: Cache size in bytes

B: Block size in bytes

A: Associativity

$$N_{rows} \text{ (number of rows)} = \frac{S}{BA}$$

$$N_{cols} \text{ (number of columns)} = 8BA$$

The two main components of cache

1. *Data array*: When people say ‘1 KB cache’, they refer to the size of the data array. Data array is where the cached information is stored. The larger the cache, the more information is stored and hence the greater probability that the cache is able to satisfy the request.

2. *Tag array*: This small area in the cache is used to keep track of the location in memory where the entries in the data array come from. The size of tag array, not the size of data array, controls the amount of main memory that can be cached.

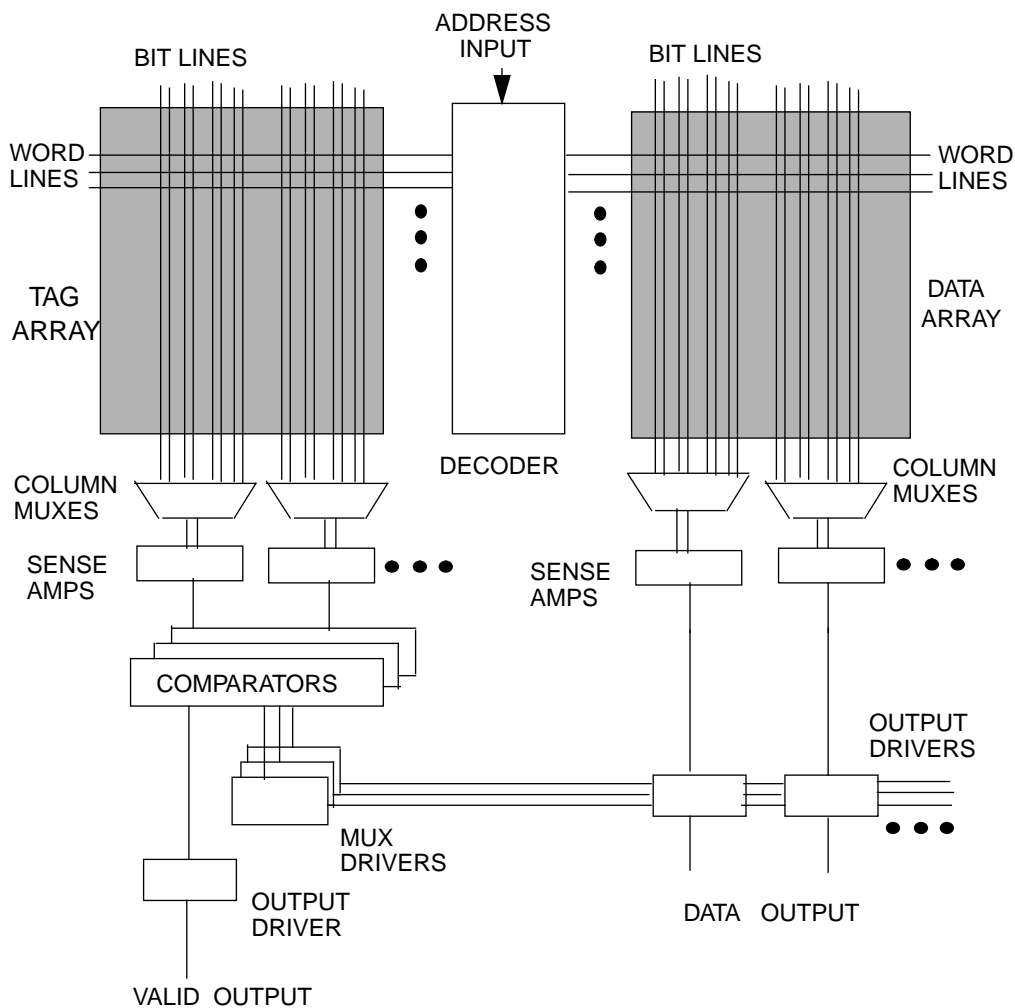


Figure 2.2: Cache structure [21]

2.1.2 Review of I-Cache Operation

Cache read and cache refill accesses are the two most common operations performed on I-caches. Cache invalidate is another operation but does not occur frequently.

2.1.2.1 Cache Read Access [21]

First, the row decoder decodes the index bits of the block address from the CPU. It will select the proper row by driving one wordline in the data array and one wordline in the tag array. There are as many number of wordlines as number of rows in each array and only one wordline is driven at a time. Each memory cell along the selected row is associated with a pair of bitlines which are initially precharged high. When the wordline goes high, one of the two bitlines in each memory cell along the selected row will be pulled down. Which one of the two bitlines will go low depend on the value stored in the memory cell.

The voltage swing of the pulled-down bit line is small, normally around 200 mV. This small differential voltage developed between the bitline pairs will get amplified by the sense amplifier. By detecting which of the two bitlines goes low, sense amps can determine the value stored in the memory cell. It is common to share a sense amp among several pairs of bitlines, using a column multiplexor before the sense amps. The select lines of the column multiplexor are driven by the column decoder.

The comparators compare the information read out from the tag array to the tag bits of the block address. If a tag match occurs and the corresponding valid bit is set, a cache hit signal is generated and the output multiplexors are driven. The output multiplexor selects the appropriate data from the data array, and drives that selected data out of the cache. On the other hand, if a tag mismatch occurs, a cache miss signal is generated. The cache controller then selects a victim block to overwrite with the desired data. After that, the cache controller will fetch the missing block into the victim's slot. During cache miss, the CPU has to wait until the desired block from memory is brought into the cache.

In a direct-mapped cache there is only one tag per index address; therefore, only one tag compare is needed. On the other hand, in a fully-associative or set-associative cache there is more than one tag per index address; therefore, more words can be stored under the same index address but more tag compares are needed.

There are two possible critical paths in a cache read access. If the time to read the tag

array, perform the comparison, and drive the multiplexor select signals is longer than the time to read the data array, the tag side is the critical path. However, if the time to read the data array is longer, the data side is the critical path.

2.1.2.2 Cache Refill Access

The I-cache is normally read only. There is no write access in the I-cache but whenever there is a read miss, a cache controller will refill the victim's slot by fetching the missing block from main memory. The cache refill operation writes the memory cell by driving complementary voltages on the bitlines. When the wordline of the selected row goes high, one of the two bitlines in each memory cell along that wordline will be driven to ground while the other bitline will remain at V_{DD} .

2.1.2.3 Cache Invalidate

Usually I-caches are not kept hardware coherent with updates to main memory. So when instruction memory is changed the I-cache could contain old instructions. Hence, when the processor accesses the I-cache, it might get stale instructions resulting in incorrect program operation.

To prevent this, whenever instruction memory changes, all entries in the I-cache must be invalidated. This forces the processor to refill cache entries with the updated instructions from main memory. Cache invalidate therefore helps solve the problem from stale data and maintain correct operation when instruction memory changes.

2.2 Development of Cache Power Consumption Model

The development of our cache power consumption model is based on the instruction cache of a Torrent-0 (T0) vector microprocessor. Implemented in a 1.0 μm CMOS technology, T0 has a maximum clock frequency of 45 MHz. Its CPU is a MIPS-II compatible RISC processor with a 32-bit integer datapath [3]. T0 has a 1KB direct-mapped I-cache with 16-byte blocks. There are 64 rows (lines), each holding 4 instructions (16-byte). The bitlines of the T0 I-cache design have a full rail swing compared to the low power cache designs where the voltage swing on the pulled-down bitline is small, around 200 mV. Since T0 I-cache, which our power consumption model is based on, is a direct-mapped cache, our cache power consumption model is

developed specifically for direct-mapped cache.

2.2.1 Characterization of Energy Consumption Model

The energy dissipation in CMOS circuits is dominated by charging and discharging of the capacitance during output transitions of the gates. For every low-to-high and high to low transition, the capacitance on the node, C_L , goes through the voltage change. The energy dissipated per transition (a 0->1 or 1->0 transition) is as follows:

$$E_t = 0.5 \cdot C_L \cdot \Delta V \cdot V_{DD}$$

where C_L : load capacitance

Other than memory bitlines and low-swing logic, most nodes swing from ground to V_{DD} , so V can be simplified to supply voltage, V_{DD} . Accordingly, the energy dissipated per transition can be simplified to:

$$E_t = 0.5 \cdot C_L \cdot V_{DD}^2$$

To accurately estimate the power dissipation of cache, we develop a cache power consumption model based on the analysis of energy dissipation in an SRAM cell. The major components in SRAM that dissipate energy for read/write access are decoding path, wordline, bitline and I/O path [9], which are explained below.

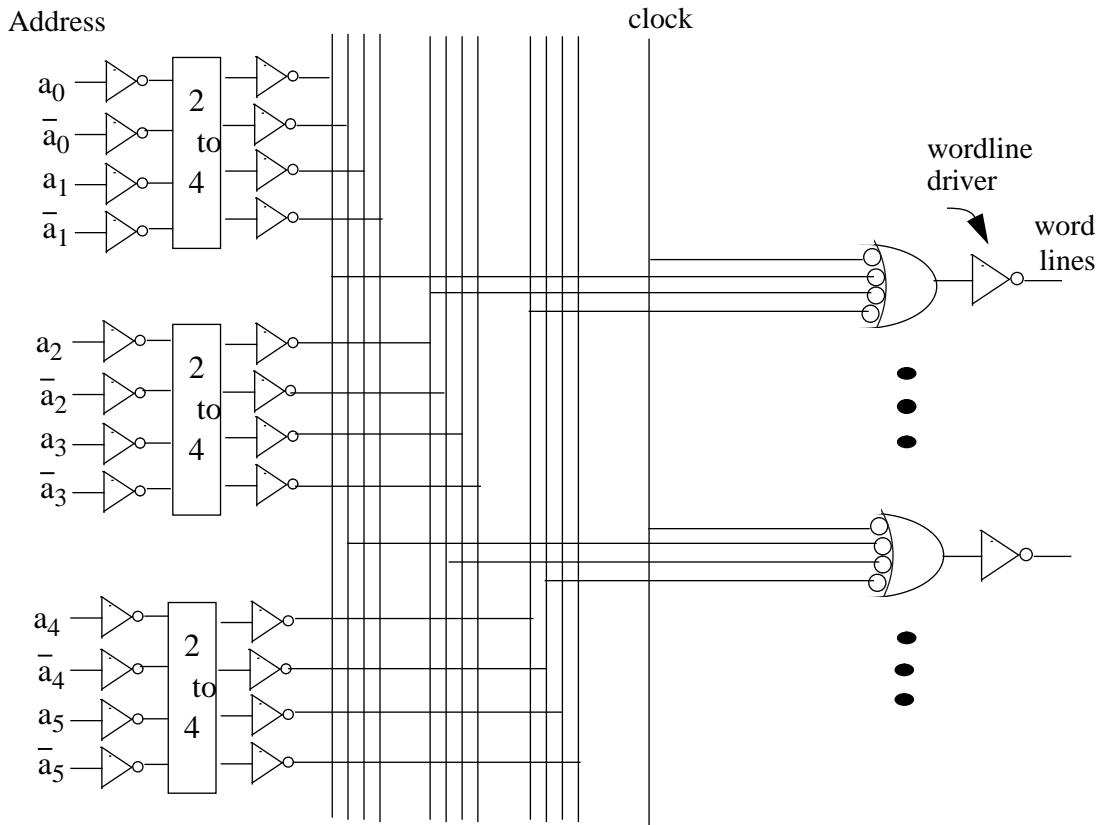


Figure 2.3: Decoder circuit of T0 I-cache

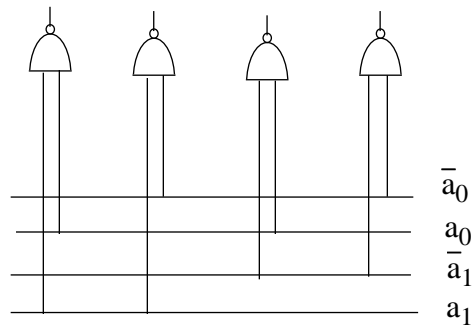


Figure 2.4: A 2-to-4 predecoder block

2.2.1.1 Decoding Path

The decoder architecture shown in Figure 2.3 is based on the decoder of the T0 I-cache. The data array and the tag array share the same decoder, which has four stages. Following the approach in Wilton and Jouppi [21], we have formulated the energy dissipated in

decoding path. Each 2-to-4 block in the second stage takes two address bits (both true and compliment) from the decoder drivers of the first stage and generates a 1-of-4 code from four NAND gates. Cache with S cache size, B block size and A-way associativity, has $\log_2 \frac{S}{BA}$ bits that must be decoded; therefore, the number of 2-to-4 blocks required is:

$$N_{2to4} = \frac{1}{2} \log_2 \frac{S}{BA}$$

Note that 3-to-8 blocks and combination of 2-to-4 blocks with 3-to-8 blocks can also be used for this second stage. Buffering between the second and the fourth stage are the role of the third stage inverters. In the fourth stage, these 1-of-4 codes are combined using NAND gates. One NAND gate is needed for each of $\frac{S}{BA}$ rows. Since, each NAND gate takes one input from each of the 2-to-4 blocks, it has N_{2-to-4} inputs. Finally, each NAND gate in the fourth stage connects to an inverter which drives the wordline driver of each row.

Stage 1: Decoder Driver

The decoder driver drives inputs of NAND gates. As both polarity of the addresses are available, each driver will drive half of the NAND gates in each 2-to-4 block. In short, each decoder driver drives 2 of 4 NAND gates in each 2-to-4 block. Hence, the equivalent capacitance is:

$$C_{stage1} = C_{d, dec_driver} + (N_{addr \rightarrow predec_NAND} \cdot C_{g, predec_NAND}) + (L \cdot C_{wordmetal})$$

where C_{d, dec_driver} : the drain capacitance of decoder driver

$N_{addr \rightarrow NAND}$: the number of NAND gates that each decoder driver drives (2 in our case)

$C_{g, predec_NAND}$: the gate capacitance of NAND gate in 2-to-4 block

$C_{wordmetal}$: the capacitance of metal 1 per bit width

L : length in bit width of metal 1, connecting the decoder driver with the

2-to-4 block, (It is one quarter of the sum of the array width, $\frac{8BA}{4}$ or $\frac{N_{cols}}{4}$ in T0 cache.)

Therefore, the formula can be simplified to

$$C_{\text{stage1}} = C_{\text{d,dec_driver}} + (2 \cdot C_{\text{g,predec_NAND}}) + \left(\frac{N_{\text{cols}}}{4}\right) \cdot C_{\text{wordmetal}}$$

Stage 2: NAND Gates

The equivalent capacitance for stage 2 is:

$$C_{\text{stage2}} = C_{\text{d,predec_NAND}} + C_{\text{g,inv}} + (L \cdot C_{\text{bitmetal}})$$

where $C_{\text{d,predec_NAND}}$: the drain capacitance of NAND gate

$C_{\text{g,inv}}$: the gate capacitance of the inverter buffering between stage 2 and stage 3

C_{bitmetal} : the capacitance of metal 2 per bit height

L : the length in bit height of metal 2 connecting the NAND gate to the inverter. (In our case, it is $\frac{S}{BA}$ rows or 'N_{rows}'.)

Hence, the simplified formula is

$$C_{\text{stage2}} = C_{\text{d,predec_NAND}} + C_{\text{g,inv}} + (N_{\text{rows}} \cdot C_{\text{bitmetal}})$$

Stage 3: Inverter

The equivalent capacitance for stage 3 is:

$$C_{\text{stage3}} = C_{\text{d,inv}} + (N_{\text{inv}\rightarrow\text{NAND}} \cdot C_{\text{g,dec_NAND}})$$

where $C_{\text{d,inv}}$: the drain capacitance of the inverter

$C_{\text{g,dec_NAND}}$: the gate capacitance of NAND gate in stage 4

$N_{\text{inv}\rightarrow\text{NAND}}$: the number of NAND gates in stage 4 that the inverter connects to. (It is

$\frac{S}{4BA}$ or $\frac{N_{\text{rows}}}{4}$, since we use 2-to-4 block in stage 2. If instead, we use 3-to-8 block, this

number will be $\frac{N_{\text{rows}}}{8}$.)

Accordingly, the simplified formula is

$$C_{\text{stage3}} = C_{\text{d,inv}} + \left(\frac{N_{\text{rows}}}{4} \cdot C_{\text{g,dec_NAND}}\right)$$

Stage 4: NAND Gates

The equivalent capacitance of stage 4 is:

$$C_{\text{stage2}} = C_{\text{d,dec_NAND}} + C_{\text{g,wordline_driver}}$$

where $C_{\text{d,dec_NAND}}$: the drain capacitance of NAND gate

$C_{\text{g,wordline_driver}}$: the gate capacitance of the wordline driver

Energy dissipated in the decoding path is estimated by the average number of bit switches on the address bus which result in the switching of the decoding path capacitance.

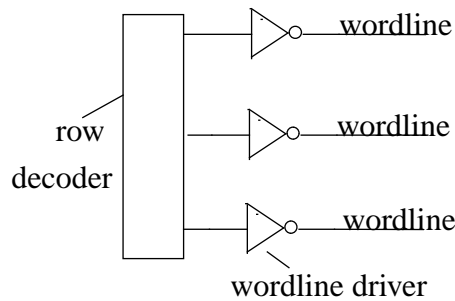


Figure 2.5: Wordline drive logic

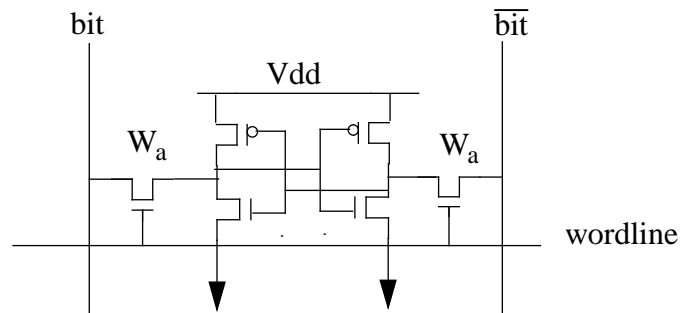


Figure 2.6: SRAM cell

2.2.1.2 Wordline

The wordline driver drives the wordline for read/write access. The energy dissipates due to the assertion of wordline. Based on [11, 19], energy dissipation on wordline can be expressed as:

$$C_{\text{wordline}} = N_{\text{cols}} \cdot (2 \cdot C_{g,W_a} + C_{\text{wordmetal}}) + C_{d,\text{wordline_driver}}$$

where C_{wordline} : the effective load capacitance of wordline

$C_{\text{wordmetal}}$: the capacitance of the wordline metal wire over the extent of a single cell width

$C_{d,\text{wordline_driver}}$: the drain capacitance of the wordline driver

C_{g,W_a} : the gate capacitance of access transistor (W_a)

The factor of 2 for C_{g,W_a} is based on the fact that the wordline drives the gates of two access transistors in each SRAM bit cell.

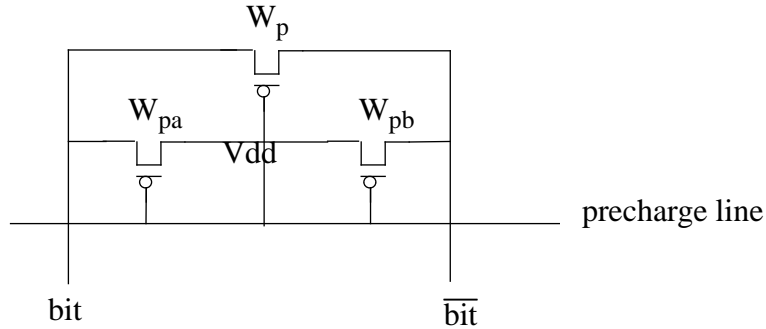


Figure 2.7: Bitline precharge circuit

2.2.1.3 Bitline

In preparation for an access, bitlines are precharged and during actual read/write one side of the bitline pairs are pulled down. Energy is, therefore, dissipated in bitlines due to precharging and during the read/write. Based on [11, 19] the following equations are derived:

$$C_{\text{bit, pr}} = N_{\text{rows}} \cdot (0.5 \cdot C_{d,W_a} + C_{\text{bitmetal}}) + C_{d,\text{bitmux}}$$

$$C_{\text{bit,r/w}} = N_{\text{rows}} \cdot (0.5 \cdot C_{d,W_a} + C_{\text{bitmetal}}) + C_{d,\text{bitmux}} + C_{d,W_p} + C_{d,W_{pa}}$$

where $C_{\text{bit,pr}}$: the effective load capacitance of the bitlines during precharging
 $C_{\text{bit,r/w}}$: the effective load capacitance of the cell read/write
 $C_{\text{d,W}_x}$: the drain capacitance of transistor W_x
 C_{bitmetal} : the capacitance of the bitline metal wire over the extent of a single cell height
 $C_{\text{d,bitmux}}$: the drain capacitance of the column mux

The drain capacitance of access transistor (W_a) in the equations is reduced by half because the drain is shared with another cell in adjacent row.

2.2.1.4 Input / output Lines

Energy dissipation in the input lines is caused by the transitions on the input lines and input latches, while energy dissipation in the output lines is caused by driving the signals from cache to the output lines.

2.2.2 Energy Consumption Model in Direct-mapped Cache

Energy dissipates due to charging and discharging of capacitance. When the cache is accessed, the address decoder first decodes the address (index) bits to find the desired row. The transition in address bits causes the charging and discharging of capacitance in decoder path. This brings about the energy dissipation in decoder path. The transition in fourth stage of the decoder logic triggers the switching in wordline. Regardless of how many address bits change, only two wordlines among all will be switched. One asserts while the other one disasserts. The switching of these two wordline is the source of wordline energy dissipation. The alteration in wordlines induces half of the bitlines in the SRAM array to go low, causing energy dissipation. After the cache access, those bitlines that had previously gone low, will be precharged and will further dissipate energy.

With this in mind, the charging and discharging of capacitance during cache access can be expressed as:

$$C_{\text{rowdec_switching}} = (N_{\text{addr_trans}} \cdot C_{\text{stage1}}) + \alpha \cdot (C_{\text{stage2}} + 2 \cdot C_{\text{stage3}}) + (4 \cdot C_{\text{stage4}})$$

where $N_{\text{addr_trans}}$: the number of transient address bits

α : the switching factor

The factor doubling C_{stage3} is derived from the switching of two rows, one charging and the other discharging. The factor of 4 for C_{stage4} comes from two reason. The first 2 factor is from the same reason as that for C_{stage3} . The other 2 comes from the fact that left and right SRAM arrays in T0 I-cache share the same decoder circuitry; therefore, the capacitance seen in fourth stage is doubled.

How to find the value of switching factor is explained in the following:

T0 I-cache has 64 rows; therefore, six address bits must be decoded. The number of 2-to-4 blocks required to predecode these six address bits is three. If one of six address bits is transient from 0->1 or 1->0, two of four NAND gates in 2-to-4 block will be switching. Hence, the switching factor is two. If two address bits from the same 2-to-4 block change, the switching factor is still two. On the other hand, if two address bits from different 2-to-4 block switch, four NAND gates will be transient; two from each 2-to-4 block. The switching factor will, therefore, be four. For this reason, the maximum switching factor in three 2-to-4 block is six. In conclusion, if the transition in address bits comes from the same 2-to-4 block, the switching factor is always two. However, if the transition of address bits comes from N different 2-to-4 blocks, the switching factor is 2N.

$$C_{\text{wordline_switching}} = 2 \cdot C_{\text{wordline}}$$

$$C_{\text{bitline_switching}} = (N_{\text{cols}} \cdot C_{\text{bit,r/w}}) + (N_{\text{cols}} \cdot C_{\text{bit, pr}})$$

The total energy dissipation in direct-mapped cache based on CMOS SRAM will be given by

$$E_{\text{cache}} = \frac{1}{2} \cdot (C_{\text{rowdec_switching}} + C_{\text{wordline_switching}} + C_{\text{bitline_switching}} + C_{\text{I/O_path}}) \cdot V^2$$

Since all nodes in T0 I-cache swing from V_{DD} to ground including bitlines, V can be simplified to V_{DD} . Energy dissipation in I-cache, therefore, can be simplified to

$$E_{\text{cache}} = \frac{1}{2} \cdot (C_{\text{rowdec_switching}} + C_{\text{wordline_switching}} + C_{\text{bitline_switching}} + C_{\text{I/O_path}}) \cdot V_{\text{DD}}^2$$

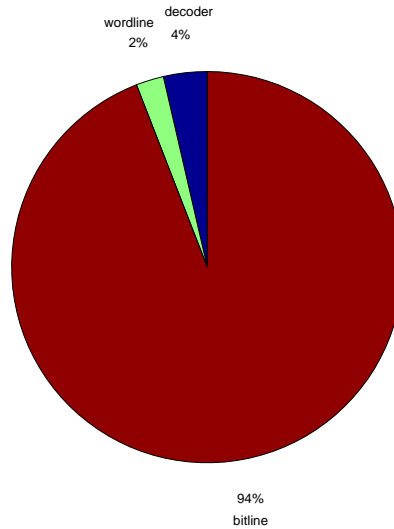


Figure 2.8: Breakdown of switching energy in T0 I-cache

2.2.3 Energy Dissipation in T0 I-cache

Based on the energy consumption model of direct-mapped cache, we have calculated the energy dissipation during cache access in the I-cache of T0 which is implemented in 1.0 μm technology and has the following properties; 1 KB direct-mapped cache with 16-byte blocks assuming a supply voltage of 5 volts and full rail voltage swing on the bitlines. Nevertheless, in our calculation we did not have the information about I/O path of T0 I-cache; therefore, we have included the switching energy dissipated only from decoder path, wordlines and bitlines. The total switching energy dissipated from T0 I-cache is 2.92 nanojoules. We have found that energy dissipated from bitlines dominate other sources of energy. Of all three sources, bitlines account for 94% of the total switching energy, wordlines and decoder circuit account for 2% and 4% respectively as shown in Figure 2.8.

In the study by Ghose and Kamble [8], they used the capacitances for the 0.8 μm CMOS cache, assumed a supply voltage of 5 volts and limited the voltage swing on the bitlines to 500 mV. Their experimental results reveal that around 75% of energy dissipated from the 32KB direct-mapped I-cache with 16-byte blocks is from bitlines. Compared to our study, if we use the 500 mV voltage swing on the bitlines, as in Ghose and Kamble, the energy dissipated in the bitlines will be reduced by the factor of $\frac{1}{10}$. Therefore, 61% of energy dissipated in 1KB direct-

mapped I-cache with 16-byte blocks and implemented in 1.0 μm technology will be from bitlines. The difference in the percentage of power dissipated in the bitlines between our study and their study might have come from the fact that the design implementation, cache size, the CMOS technology used in the implementation are different in the two studies.

As energy dissipated from bitlines dominates the total energy dissipation in cache, low power techniques should focus on reducing the energy dissipated in the SRAM array where the bitlines are located.

Chapter 3

Previous Work on Low Power Cache Design

Many studies have been performed on reducing the power consumption of cache. Previous low-power cache literature suggests several techniques to achieve low power such as sub-banking [19][11][18], block buffering [19][11][13][16], gray code addressing [19] and reducing the frequency of tag compares [16][18]. In this thesis, we have selected to study two of these techniques; sub-banking and reducing the frequency of tag compares. This is because these two techniques and our own technique of using gated wordline, which we will discuss in detail in Chapter 4 and 5, use the same strategy to lower the power consumption in the I-cache; that is by reducing the energy dissipated in the SRAM array. Our technique of using gated wordline reduces bitline swings in the data array. Sub-banking partitions data array and tag array into smaller banks and only accesses the sub-bank where the data is needed. Reducing frequency of tag compares eliminates the power dissipated in the tag array when tag comparison is not necessary.

In this chapter and Chapters 4 and 5 we will discuss these three design techniques to achieve low power by reducing the power dissipation in SRAM array of the I-cache. In this chapter we will present our experimental study on sub-banking and reducing the frequency of tag compares while in Chapter 4 and 5 we will describe our technique on using gated wordlines to reduce the number of bitline swings.

Experimental Techniques

The techniques we have chosen to study in this thesis are sub-banking and reducing the frequency of tag compares. We did not study block buffering and gray code addressing due to the following reason. Block buffer is in fact another level of cache which is closer to processor than level-1 cache. The concept of block buffering is to access a small buffer where frequently required instructions reside and to access the I-cache only when there is a block miss in the buffer. Since smaller cache dissipates less power, cache with block buffer can save power by optimizing capacitance of each cache access [19]. However, as cache is partitioned into several small sub-banks, having yet another smaller cache (buffer) does not help improve the power reduction by much. As we will partition a cache into several small sub-banks in our study and only access needed instructions, block buffering is therefore not applicable. As for Gray code addressing, a Gray code sequence is a consecutive set of numbers having only one bit different. Gray code addressing is more optimal as it can minimize bit switching of the address buses when accessing sequential memory data. Hence, Su and Despain [19] suggested replacing 2's complement representation with gray code one [19]. However, in our study we did not consider changing the 2's complement representation to the Gray code addressing because the energy dissipated from the address bus is only a small part of the overall switching energy, as concluded by Ghose and Kamble [8]. Therefore, we continue with 2's complement representation for our study.

Now, we will cover sub-banking and reducing the frequency of tag compares, the two techniques we have selected to study.

3.1 Sub-banking

3.1.1 Experimental Study

The idea of sub-banking is to partition the cache data array into multiple banks. Each bank can be accessed (powered) individually. Only the sub-bank where the required data is located consumes power in the access. Sub-banking saves power by accessing a smaller array. The amount of power saving depends on the number of sub-banks. The more the sub-banks used the more the power saved. The main draw back is that each sub-bank must have its own decoding logic thereby increasing the overhead area for the decoding circuitry. In addition, more sense amplifiers are needed after sub-banking because fewer number of bitline pairs share the same

sense amplifiers. In other words, the degree of multiplexing by the column multiplexor, that is the number of bitlines that share a common sense amplifier [21], is smaller. In fact, in our study of sub-banking which we will describe later in this section, sense amplifier is needed for each pair of bitlines. As a result there is no need for a column multiplexor before the sense amp.

The logic for sub-bank selection is usually simple and can be easily hidden in the cache address decoder [19]. Block offset bits of block address from CPU are used to select the sub-bank where the requested data is located. Address decoders in each sub-bank, therefore, have extra inputs taken from block offset bits to control the power up of each bank. This way, the address decoder will enable only the desired sub-bank before the wordline is asserted [11]. Since block offset bits are available even before the read or write access is performed, they can be used in the address decoder without causing any delay to the decoder circuit.

In this thesis, we have expanded the study of sub-banking further. In general, there are many instructions in a cache line. In a naive cache design, the whole cache line is read out. Then, the column multiplexor selects the needed instruction. As a result, a lot of energy is wasted in reading out the extraneous instructions in the cache line. Here, we present the technique to read out only the instruction demanded for the instruction fetch by using sub-banking [2].

Each sub-bank in our technique has only two instructions in its cache line; one each on the left and the right side sharing the decoder circuit at the center. The instruction on either side of sub-bank can be accessed separately. Therefore, in each instruction fetch, only one side of one sub-bank is powered on and only the instruction requested for the instruction fetch is read out. Thus, besides saving power from sub-banking, we also save power from not reading the undesirable data in the sub-bank. By reading out only the requested instruction, a lot of energy can be saved. The bigger the cache block size (more instructions/words in a cache line), the more the sub-banks in a cache and hence the more the power saved by this technique.

Aside from sub-banking the data array, the tag array can also be sub-banked by splitting it with horizontal cutlines (causing shorter bitlines). Each sub-bank of the tag array will have two tags in each row; one each on the left and the right side sharing the decoder circuit at the center. As a result, in each tag check, only one side of one sub-bank is powered on and only the tag of the required address is checked. Sub-banking hence helps save power dissipated in the tag array from long bitlines. In addition, the access time also speeds up, since shorter bitlines have smaller load capacitance and thereby faster time to charge and discharge their load capacitance.

3.1.2 Experimental Results

To evaluate the sub-banking approach, we investigated three options of cache organizations. The power model used in this investigation is based on the energy consumption model developed from T0 I-cache.

Option 1: This is a conventional cache organization and T0 is also based on this organization. In this option, the whole cache line, which usually has multiple instructions, is read out from the data array. Only the one instruction needed for the instruction fetch is selected by the offset bits. Tag compare is done for every instruction fetch.

Option 2: In this option, data array is partitioned into several sub-banks. Each sub-bank has only two instructions in its cache line, one each on the left and the right side sharing the decoder circuit at the center. Each sub-bank is powered up individually and the instruction in either side of sub-bank can be accessed separately. The instruction needed for the instruction fetch is the only instruction read out from the data array.

Tag array is also sub-banked by splitting in half with a horizontal cutline. This results in two sub-banks with shorter bitlines. Each sub-bank has two tags on each row; one each on the left and the right side sharing the decoding circuit at the center. Only one side of one sub-bank is powered on at a time.

Tag compare is still done for every instruction fetch as in option 1.

Option 3: The cache organization in here is identical to that in option 2, but there is no tag compare in this option.

Based on the cache energy consumption model in Chapter 2, we compared the switching energy dissipation of various cache sizes and block sizes in each option as shown in Figure 3.1, 3.2 and 3.3.

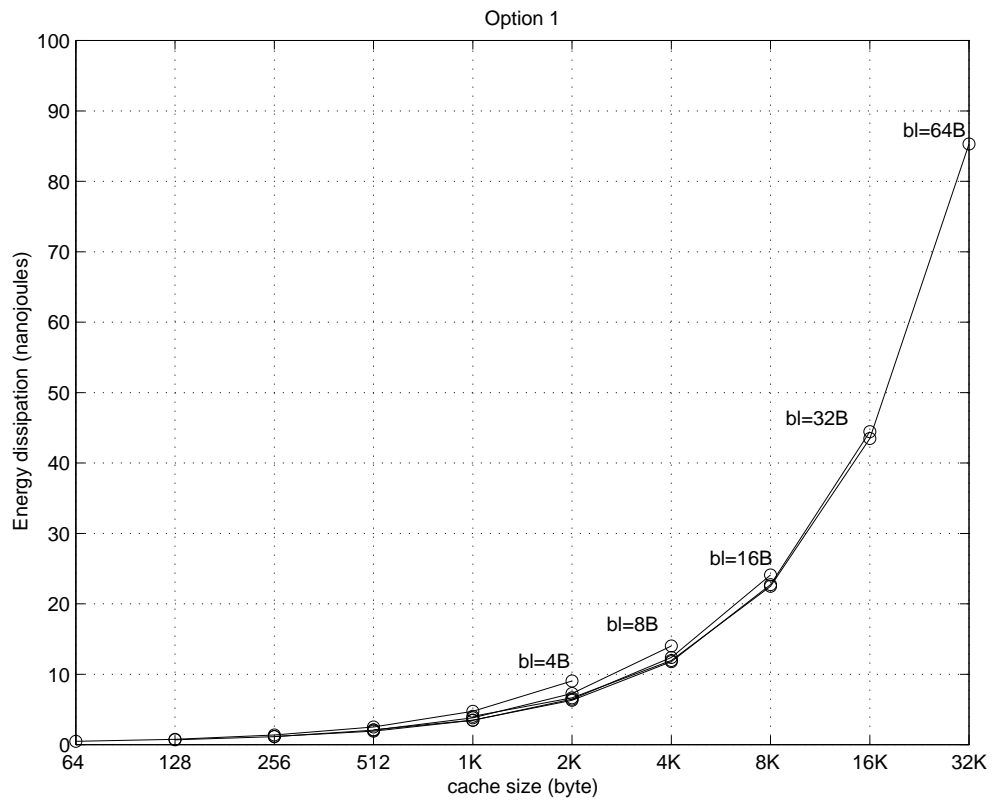


Figure 3.1: Energy dissipation in option 1 where the whole cache line is read out from the data array

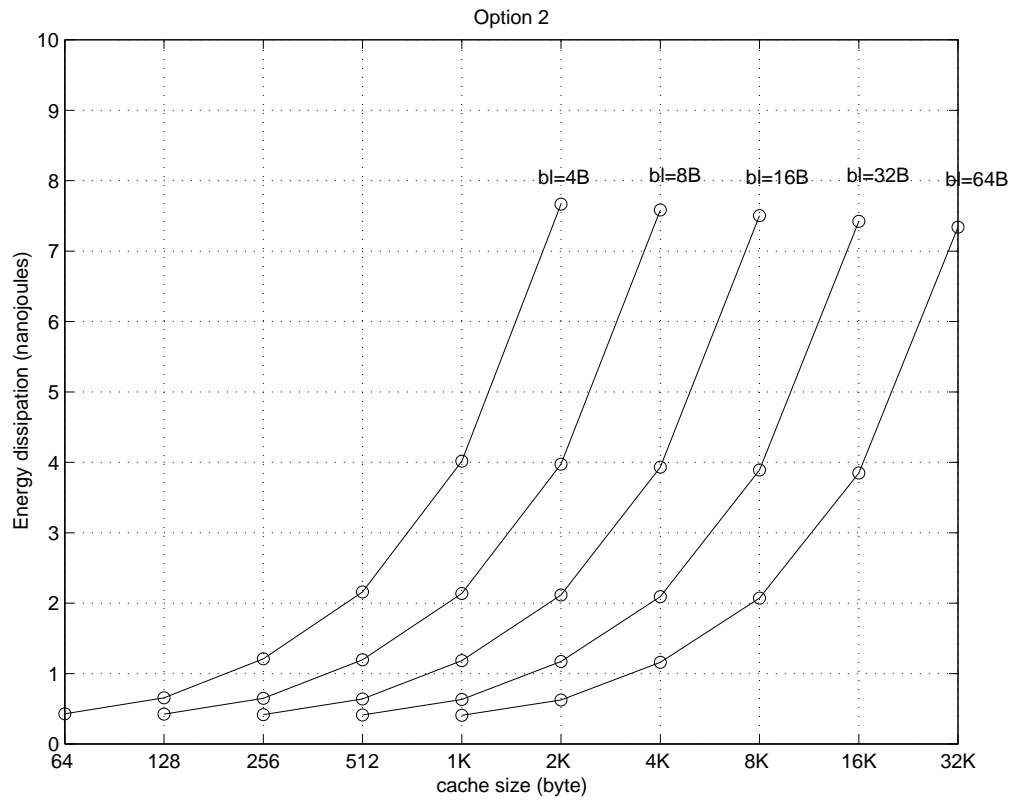


Figure 3.2: Energy dissipation in option 2 where sub-banking is used and only one instruction is read out from the data array

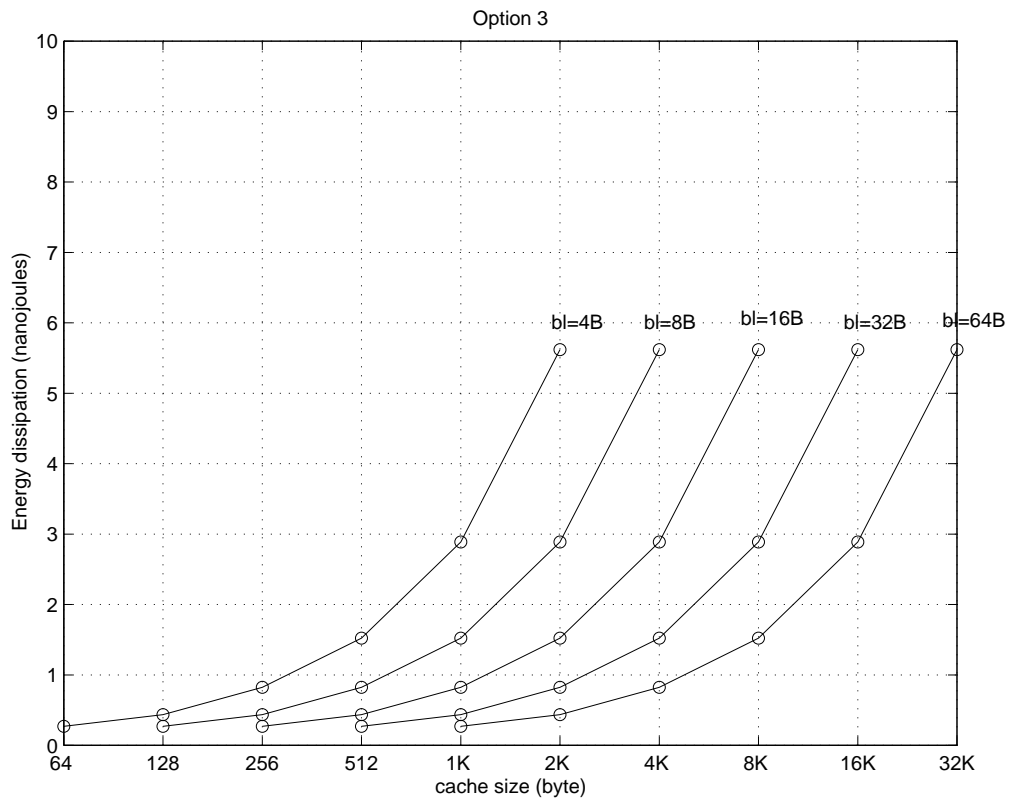


Figure 3.3: Energy dissipation in option 3 where cache organization is identical to option 2 but there is no tag compare

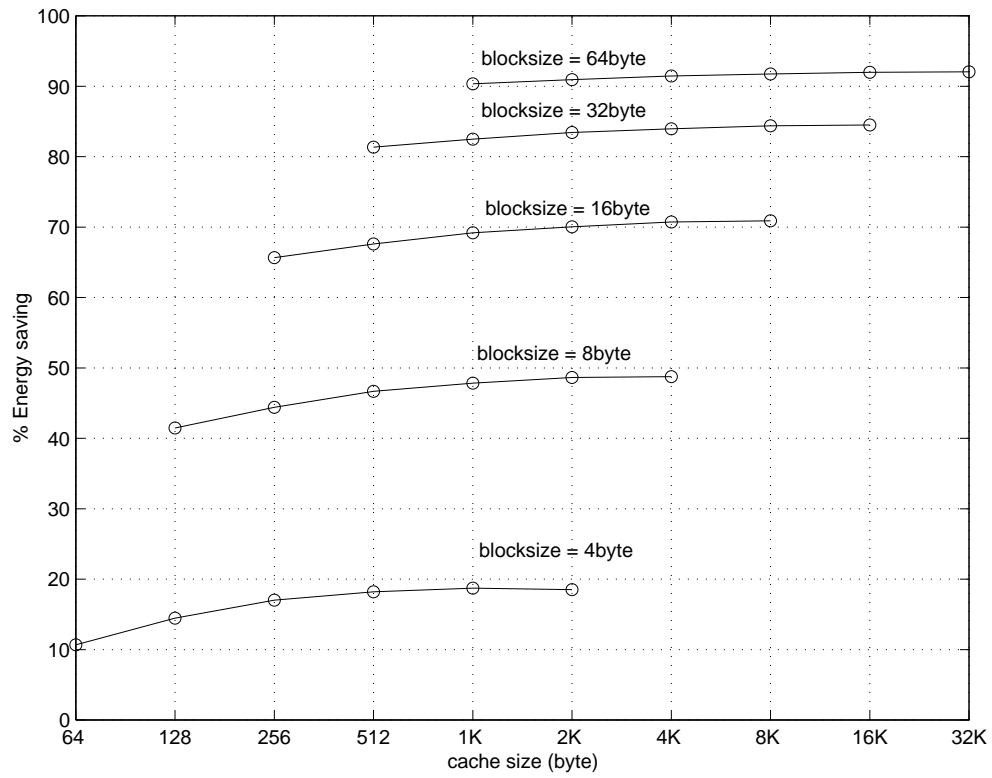


Figure 3.4: Energy saved when using sub-banking (comparing option 2 with option 1)

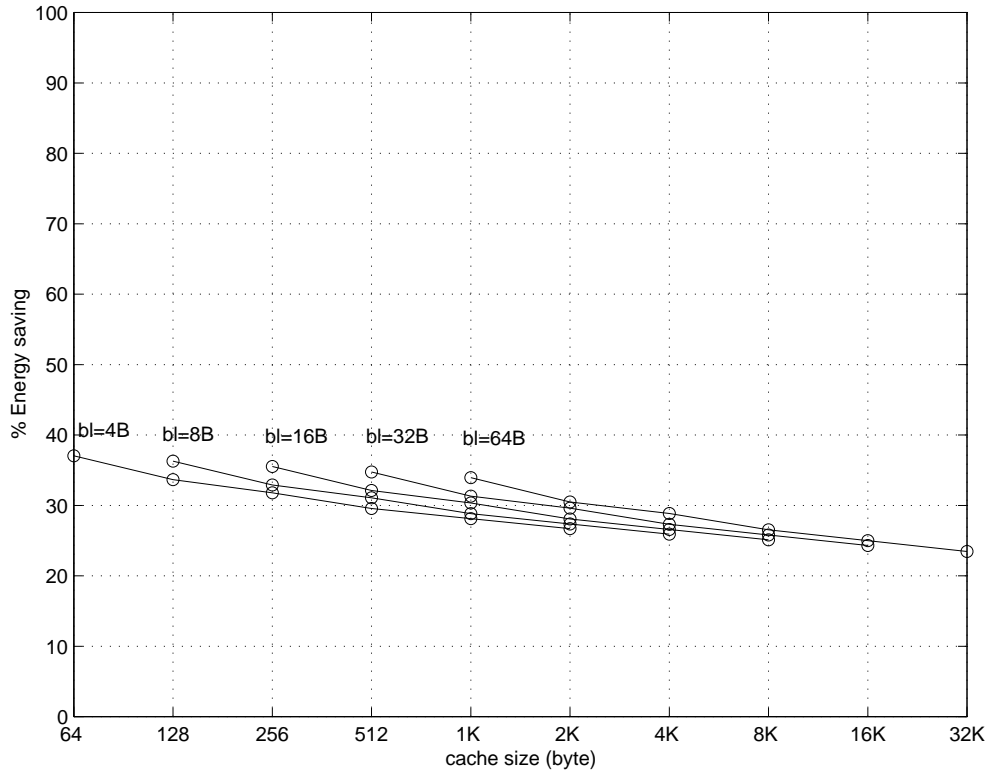


Figure 3.5: Energy saved when there is no tag compare as compared to when there is (comparing option 3 with option 2)

When comparing option 2 with option 1, we can see that sub-banking saves more energy when the cache line size is large. For 1KB direct-mapped cache with block size 4-byte, 8-byte, 16-byte, 32-byte and 64-byte, energy saving from sub-banking is 18.72%, 47.83%, 69.19%, 82.47% and 90.33% respectively as shown in Figure 3.4. In Su and Despain study [19], the energy saving of cache sub-banking is 89.45% in 32KB cache with 16-word block size (64-byte block) compared to 92.05% in our study.

Energy dissipation in option 3 is different from that in option 2 because there is no energy dissipation from tag array. Around 1.8 nanojoules is saved when there is no tag compare. This results in 23-37% saving of energy in cache sub-bank when tag compare is unnecessary.

3.2 Reducing the Frequency of Tag Compares

3.2.1 Experimental Study

Usually the address tag on each cache line is checked on every access to see if it matches the tag field in block address from CPU. As the size of the sub-bank of the data array is smaller, the energy dissipated by the tag array due to tag compare accounts for a larger part of power consumed in I-cache. Therefore, reducing the frequency of tag compares can contribute to considerable power saving.

Unless there is a branch taken or jump instruction, instruction fetch usually happens sequentially. A cache line with block size larger than 4 bytes can hold multiple instructions which share the same tag address. In the sequential flow, if the next instruction and the current instruction are known to be in the same cache line (Intrablock Sequential Flow), tag comparison is not required since both instructions have the same tag address. On the other hand, if the next instruction and the current instruction are in different cache lines (Interblock Sequential Flow), tag comparison is needed as they have different tag addresses.

Taken branches and jumps cause non-sequential flow. In the T0 ISA which is based on the MIPS-II ISA, all jump and branch instructions occur with one instruction delay due to a delay slot [12]. This means that the instruction immediately following the jump or branch instruction is always executed while the target instruction is being fetched. Thus, if the target of a taken branch or a jump resides in a cache line different from the instruction in the delay slot, interblock non-sequential flow occurs and a tag compare is needed.

Consequently, the comparison between the tag from the tag array and the tag bits of the address does not need to be done for all instruction fetches. It is required only when there is interblock sequential flow (taken branches or jumps to a new cache block) or interblock non-sequential flow [16]. Nevertheless, it is too late by the time the result of the condition branch is available to do tag compare because in the pipeline architecture, the result of conditional branch, whether taken or untaken, is not known until the decode stage. Therefore, there must be a way to both detect during instruction fetch whether the conditional branch is taken and to monitor whether the taken branch is still in the same cache line as the delay slot. Panwar and Rennels [16] proposed a technique to solve these two problems by branch tagging the unused space in the opcode as a compiler hint that the branch would transfer the control to an instruction outside the cache block. However, in our technique of using gated wordline to reduce the number of bitline

swings, as discussed in Chapter 4 and 5, we compress and re-encode instructions using those unused spaces in the opcode table; therefore, there is no spare space in the opcode to do branch tagging. As a consequence, we use the simpler scheme here instead.

In our study, we will not wait for the result of conditional branch, whether taken or not taken. Neither will we attempt to detect whether the taken branch or jump transfers the control to a new cache block. We will use a simple technique by doing tag compare in every conditional branch and jump as well as interblock sequential flow. Here we trade simplicity for performing more tag compares in both taken branch/jump to the same cache line and untaken branch.

3.2.1.1 Branch and Jump

Non-sequential instruction fetch is due to the following branch and jump instructions:

BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL, BEQL, BNEL, BLEZL, BGTZL, BLTZL, BGEZL, BLTZALL, BGEZALL, J, JAL, JR, JALR, SYSCALL, BREAK, RFE

The above instructions, which cause interblock non-sequential flow, change the control flow of the program. All instructions occur with one instruction delay due to a delay slot, except SYSCALL, BREAK and RFE which transfer control flow immediately. The frequency of tag compares by control flow instructions is given by f_j , where f_j is the frequency of these instructions which is heavily dependent on the architecture, the compiler and the application [16].

3.2.1.2 Interblock Sequential Flow

An interblock sequential flow can be easily detected by looking at bits of the program counter (PC). Let a cache blocksize be 2^k , and the PC bits be 31, 30,...,1,0 with 0 being the least significant bit. The interblock sequential flow can be found by probing from bit position $k-1$ to the right for $k-2$ bits and checking whether those bits are all 1's. If they are, the next instruction fetch will be an interblock sequential flow given that there is no branch pending. This method of detecting the interblock sequential flow is, therefore, not only simple but will also detect the interblock sequential flow one cycle in advance of when it takes effect.

For example, let's look at a cache with 16-byte blocks. In this case, the cache block size is 2^4 ; therefore, k is equal to 4. By looking from bit position 3 ($k-1$) for 2 bits ($k-2$), which means looking at bit 3 and bit 2, we can easily keep track of the interblock sequential flow. The

interblock sequential flow will be the next instruction fetch after which bit3 and bit2 are both 1's.

```

PC = 00: 00000000
      04: 00000100
      08: 00001000
      0C: 00001100 <--- detect 11
      10: 00010000 <--- Interblock Sequential (tag compare)
      14: 00010100
      18: 00011000
      1C: 00011111 <--- detect 11
      20: 00100000 <--- Interblock Sequential (tag compare)

```

It can be shown that the frequency f_{is} of tag compare due to interblock sequential flow depends on the size of the cache block. The expression is as follows:

$$f_{is} = \frac{1 - f_e - f_j}{S_B}$$

where f_{is} : the frequency of interblock sequential flow

f_j : the frequency of the branch and jump instructions

S_B : the number of instructions in a single block

f_e : the frequency of exceptions

Note: Exception is an event that happens during the execution of a program and as a result disrupts the normal flow of instructions. The events that cause exception are as follows [17]

- I/O device request
- Invoking an operating system service from a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow or underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory access
- Memory-protection violation
- Using an undefined instruction

- Hardware malfunctions
- Power failure

Hardware overhead for the conditional tag compare in interblock sequential flow case is quite small. A signal can be generated in each cycle indicating whether the tag compare is needed. When tag compare is not needed, the tag array can remain in precharge [16].

3.2.1.3 Frequency of Tag Compares

The frequency of tag compare, therefore, is as follows:

$$f_{\text{tag_compare}} = f_j + f_{\text{is}}$$

3.2.2 Experimental Methods

3.2.2.1 Evaluation Environment

The benchmark programs used in this study are as listed in Table 3.1. They are selected from the SPECint95 benchmark suite. Application for these benchmark programs include m88ksim (Motorola 88K Chip simulator), li (lisp interpreter) and jpeg (compression algorithm for images). The benchmark programs are compiled with GCC 2.7.0 using -O3 optimization for the MIPS-II ISA.

Benchmark	Description	Input	Instructions in millions
m88ksim	Motorola 88K chip simulator: run test program	test	518
li	LISP interpreter	boyer	165
		deriv	544
		dderiv	593
jpeg	Graphic compression and decompression	ref	1039

Table 3.1: Benchmark programs

3.2.2.2 Experimental Results

This section presents our study on reducing the frequency of tag compare. A C-program is written to record the statistics of branch and jump instructions as well as interblock sequential flow in benchmark programs. Tag compare statistics can be obtained readily from the summation of their statistics. Table 3.2 shows the frequency of branch and jump instructions in benchmark program.

Application	Input	% of branch and jump instructions (f_j)
m88ksim	test	21.42
li	boyer	21.85
	deriv	20.66
	dderiv	20.74
ijpeg	ref	7.28

Table 3.2: Statistics of branch and jump instructions in benchmark programs

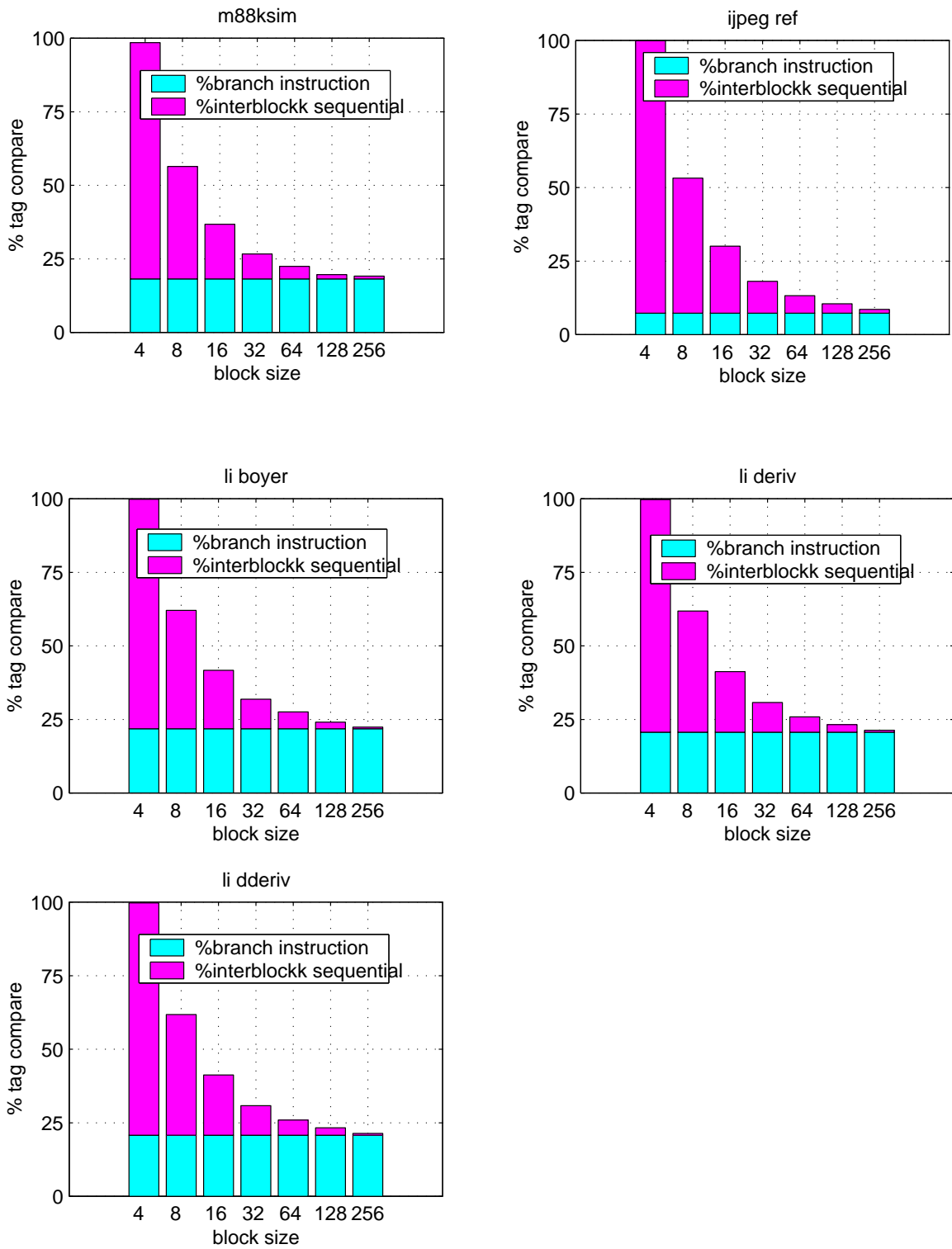


Figure 3.6: Percentage of tag compares in benchmark programs

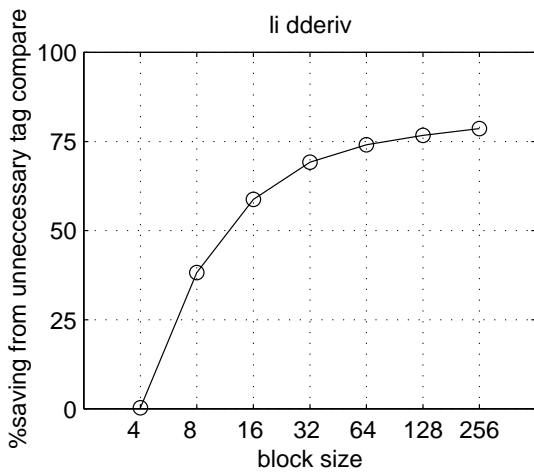
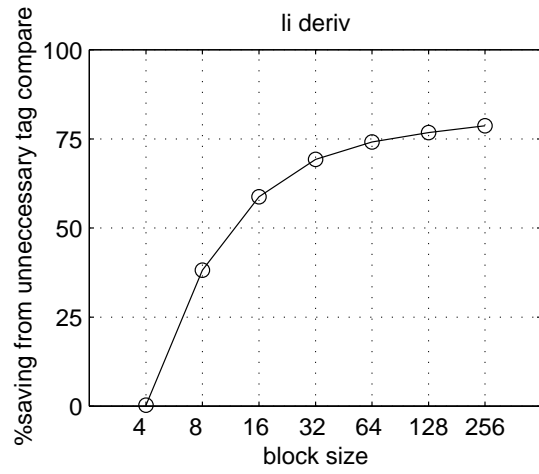
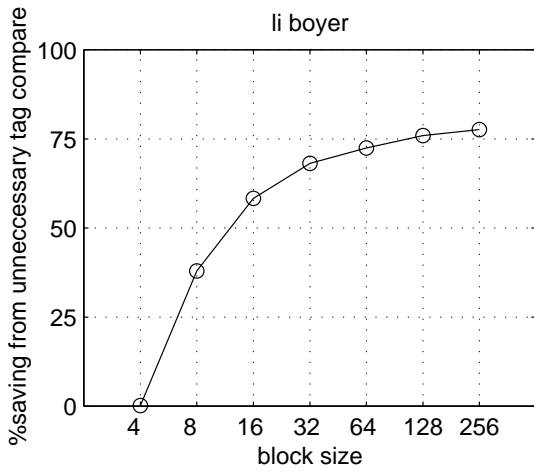
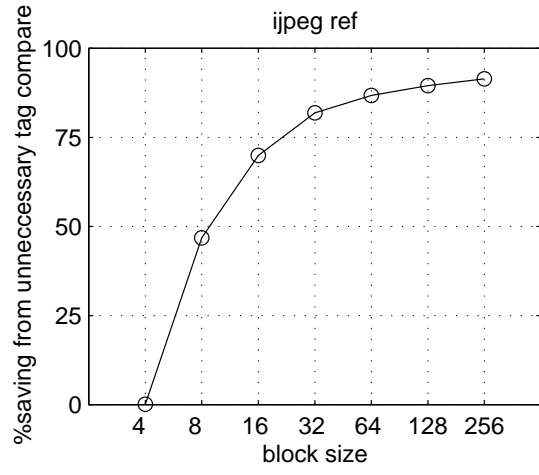
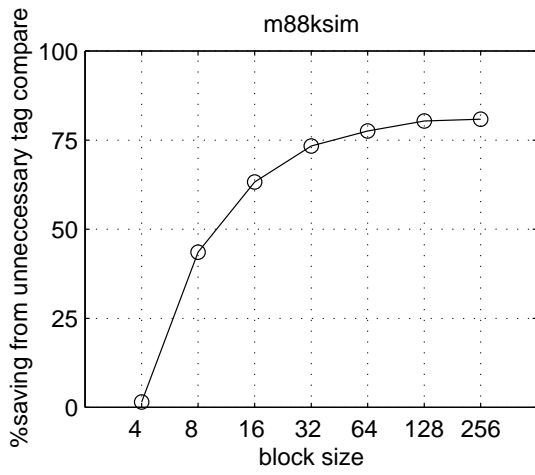


Figure 3.7: Percentage of unnecessary tag compares that has been avoided in the benchmarks

Figure 3.6 shows the percentage of tag compare in each benchmark program and also the breakdown of tag compare statistics into branch and jump instructions as well as interblock sequential flow. We can see that as the number of block size increases, which means more instructions in a cache line, the frequency of tag compare from interblock sequential flow decreases. In contrast, the frequency of tag compare from branch and jump instructions is always the same independent of block size given the same program. As block size varies from 4-byte to 256-byte, the frequency of interblock sequential flow (f_{is}) drops dramatically from 77.24% to 1.09% in m88ksim, from 92.59% to 1.29% in ijpeg, from 77.97% to 0.53% in li boyer, from 79.07% to 0.67% in li deriv and from 78.99% to 0.66% in li dderiv. In Pering et. al. [18], 61% of tag checks is prevented in I-cache using 8-word cache line size (32-byte block). Compared to the results of our simulation, 69.58%, 81.86%, 68.12%, 69.26% and 69.18% of tag checks are avoided in m88ksim, ijpeg, li boyer, li deriv and li dderiv respectively, as shown in Figure 3.7. Hence, the average of 71.6% of tag checks are prevented across the benchmarks. The difference in our result and their result arise because we used benchmark programs different from theirs and because the technique is applied to different ISAs, MIPS-II in our case and ARM8 in theirs.

Chapter 4

Background and Related Work on Using Gated Wordlines to Reduce the Number of Bitline Swings

In Chapter 2, we saw that the energy dissipated by bitlines in the SRAM array dominates the other sources of energy in the I-cache. Therefore, in this part of the thesis we will focus on developing design techniques to lower the power consumed in the I-cache, by reducing the energy dissipated from the SRAM array where bitlines are located.

Our concept is based on the fact that several instructions in the RISC instruction set do not require a full 32 bits to represent. However, the 32-bit fixed-length format is used for every instruction to simplify fetch and decode. This results not only in wasted memory space, but also in power wasted from reading out the unnecessary bits in the SRAM array of the I-cache. If the size of the instructions is reduced, the amount of power dissipated from the cell array when reading or writing the instructions will be decreased, thereby lowering the power consumed in the I-cache.

We propose two versions of the design technique that uses gated wordlines to reduce the number of bitline swings which therefore decrease the number of bits read out for compressed instructions. The first version uses instructions of two sizes, medium or long, while the second version uses instructions of three sizes, short, medium or long. Instead of reading out the 32 bits fixed-length for all instructions, the 2-size approach uses a gated wordline to read out either a

compressed medium-size instruction or an uncompressed long size instruction. Similarly, the 3-size approach uses a gated wordline to read out either a compressed short-size instruction, a compressed medium-size instruction or an uncompressed long-size instruction. This technique of using gated wordlines, therefore, lowers the power consumed in SRAM array of the I-cache by reducing the power dissipated from reading or writing unnecessary bits of instructions that do not require a full 32 bits, without any loss in performance.

4.1 Overview of Gated Wordline Technique to Reduce the Number of Bitline Swings

In this proposed technique, we select the instructions in the MIPS-II ISA, on which T0 is based, that do not require the full 32 bits to represent, then reduce the size of these instructions. We investigate two versions for reducing the instruction size of MIPS ISA. In the first technique, instead of using a fixed-length for all instructions, instructions of one of two sizes, medium or long, can be used. If the instruction can be compressed into the medium format, a medium size is used; else, a long size is used. In the second technique we go further by using three different instruction sizes; short, medium and long.

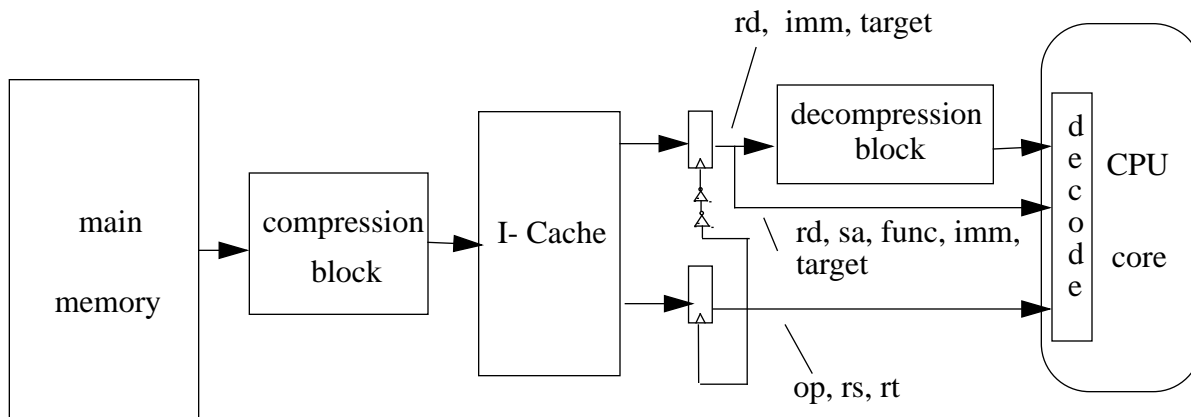


Figure 4.1: CPU layout with compression and decompression blocks

The compression block is located between the memory and the I-cache as shown in Figure 4.1. During the I-cache refill, the 32-bit instructions from memory are passed into the compression block and if possible compressed before they are written into the I-cache. On the other hand, if the instructions can not be compressed, they are stored full-length in the I-cache. An extra bit(s) must also be stored in the I-cache along with the instruction in order to

differentiate among the sizes of instructions that are stored. It should be noted here that all instructions, whether compressed or uncompressed, still take the same space in cache but the bitline swings are reduced in the compressed instructions.

In each RISC instruction, there are critical and non-critical sections. The critical sections are an opcode field (op), a source register (rs) and a target register (rt). The non-critical sections depend on the type of instructions. For the R-type instructions, the non-critical sections are a destination register (rd), shift amount (sa) and a function specifier (funct). For the I-type instruction, the non-critical section is an immediate field (imm). For the J-type instruction, the non-critical section is a target field (target). The critical sections determine the execution time; the opcode needs to be decoded to find the operation of the instruction, source and target registers must access the register file to supply their values for the computation. The time to fetch and decode the critical section hence can not be delayed, else the cycle time will be longer. On the other hand, the non-critical sections do not affect the cycle time. Immediate and target must be signed or zero extended but the time to do the extension is a lot faster than the time to access the register file; therefore, they are non-critical. And since destination register and function specifier are not needed until at the end of the decode stage and the execution stage respectively, they are also non-critical.

During the cache read access, the extra bit(s) controls the number of instruction bits fetched out from the SRAM array by gating the wordline. When the compressed instruction is read out from the SRAM array, its critical-sections proceed directly to the instruction decoder, while its non-critical sections are decompressed before being decoded and executed in the pipeline. In contrast, when the uncompressed instruction is read out, both its critical and non-critical sections can go immediately to the instruction decoder.

As a result, fewer bits will be written to the I-cache and no unnecessary bits will be read out in our proposed technique. The energy dissipated from the bitlines of the SRAM array will be reduced during cache read or cache refill of the compressed instructions. Moreover, in our technique, the critical sections, whether are in the compressed or uncompressed instructions, proceed straight to the instruction decoder; therefore, there is no extra delay added to the cycle time. Since we always fetch the critical sections very fast regardless of whether the instruction is compressed or uncompressed, our technique should not affect the execution speed.

4.2 Background of Using Gated Wordline Technique

Since there are multiple sizes of instructions stored in the I-cache, there must be a way to indicate the bit-size of the instruction to be fetched. We solve this problem by having extra bit(s) to select among multiple sizes by gating the wordline. In the 2-size method, an extra bit is required to choose between medium and long format. We call this extra bit a medium/long bit (M/L bit). Similarly, in the 3-size method, a short/medium bit (S/M bit) is used for instructions that can fit into short-size, and a short/medium bit (S/M bit) and a medium/long bit (M/L bit) are both used when the instructions are either medium-size or long-size.

The values of these extra bits are assigned during the compression. These extra bits are stored along with the compressed/uncompressed instructions in the I-cache and therefore counted as part of the instruction size. During cache read access, the values of these extra bits will be used to gate the wordline in order to select the segments of stored data that should be read out. In other words, these extra bits control the size of the instruction read out from I-cache.

The format layout of 2-size and 3-size instructions are as shown in Figure 4.2 and 4.3 respectively.

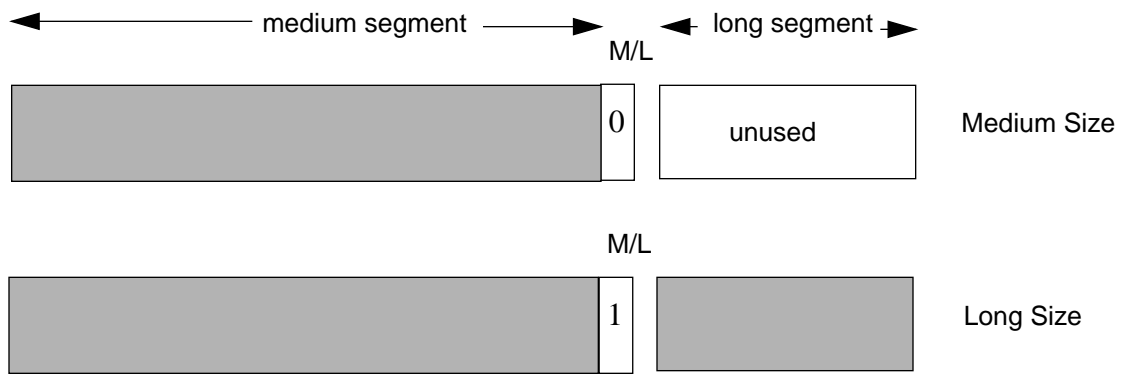


Figure 4.2: Format layout of 2-size instructions

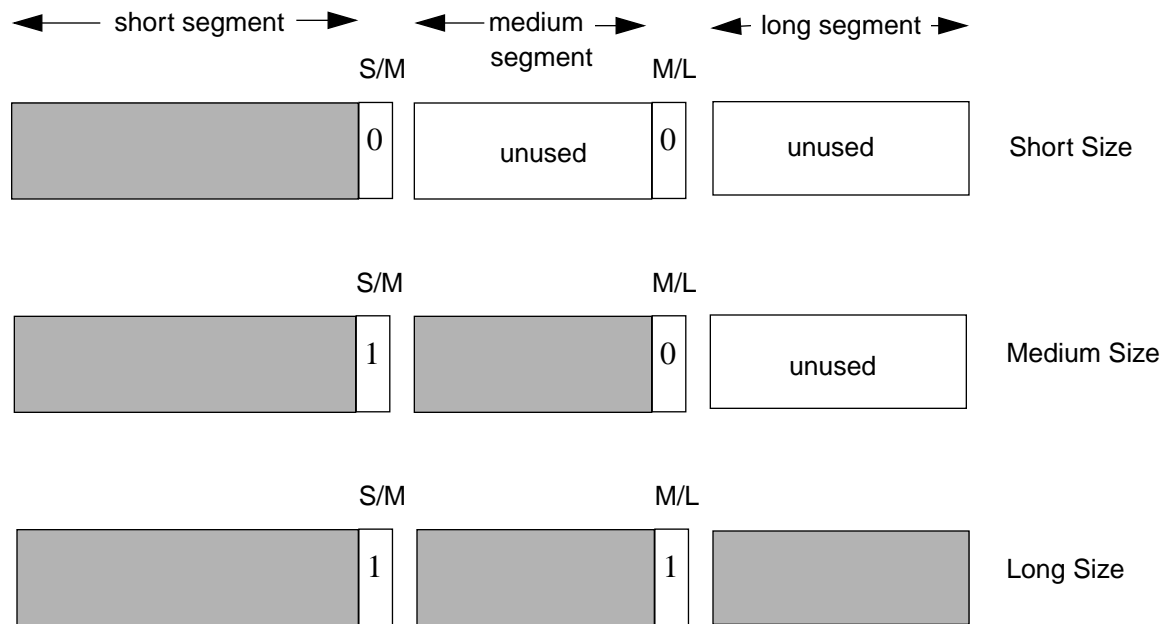


Figure 4.3: Format layout of 3-size instructions

4.2.1 The 2-size Approach

In the two-size approach, a medium-size instruction is composed of a medium segment and an M/L bit. A long-size instruction is composed of all the components of a medium-size instruction with a long segment in addition. A long segment therefore contains bits in the long-size instruction that can not fit into the medium segment.

There are two kinds of wordlines in this approach; a main wordline and a local wordline. The I-cache data array contains as many main wordlines as there are rows in the array,

but only one main wordline is driven at a time. Each main wordline connects to the memory cells storing bits in a medium segment and an M/L bit. Every main wordline is also associated with a local wordline. Each local wordline connects to the memory cells storing bits in a long segment.

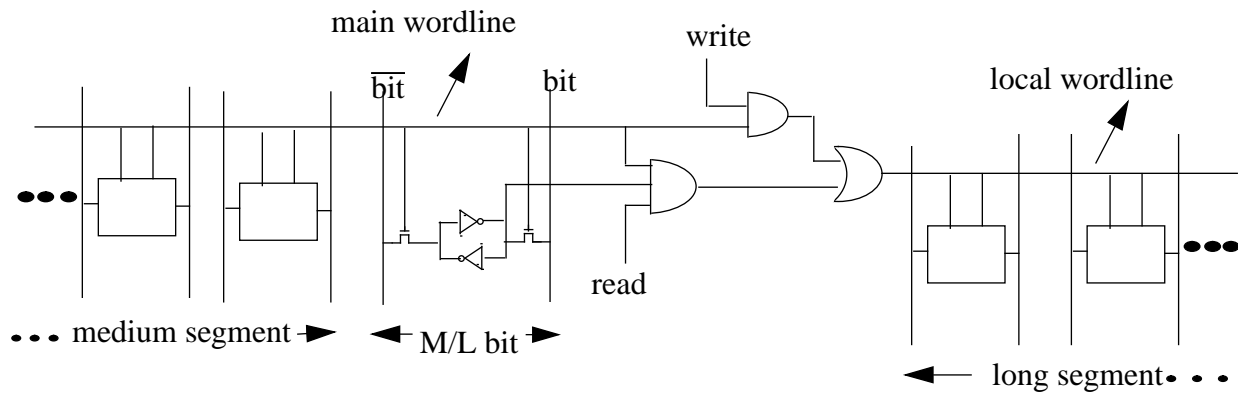


Figure 4.4: Circuit to control the size of the instruction being ‘written into’ or ‘read out from’ SRAM array in the 2-size method

4.2.1.1 The I-cache Refill

During I-cache refill, each 32-bit instruction fetched from memory is passed into a compression block. By first checking the opcode of the incoming instruction, the detection circuit in the compression block determines whether the instruction can be compressed. If the opcode indicates an incompressible instruction, the whole 32-bit instruction will be stored in the I-cache along with a set M/L bit. When the detection circuit encounters a compressible opcode, the instruction will be compressed using the compression technique we will discuss in 5.1.1.1. The compressed instruction is then stored in the I-cache along with a reset M/L bit.

During the compression process, as soon as the value of the M/L bit is determined, it will be used to control the circuitry of the ‘write’ signal which is responsible for storing of the long segment. If the M/L bit is assigned a reset value, which means that the instruction is a medium size, the ‘write’ signal is disabled since there is no need to write the long segment. On the other hand, if the M/L bit is assigned a set value, which means that the instruction is a long size, the ‘write’ signal will be enabled and data will be written into the long segment. Figure 4.4 shows the circuit which uses the ‘write’ signal to control the number of bits stored for each instruction. This control process of the write circuitry happens prior to storing of the bits in the

medium segment and the M/L bit into the I-cache. Therefore, the 'write signal' has already been set up by the time the main wordline is driven. When the main wordline goes high, the M/L bit and the medium segment of the desired instruction are written with the data to be stored. At the same time, the 'write signal' will determine whether additional data will be written into the long segment.

Instead of a separate 'write' signal, the stored value of the M/L bit could be used to control storing of the long segment. However, this is very slow because the M/L bit must be stored in the I-cache first before its value can enable writing of the long segment. As, this will introduce too much delay in storing the long segment, having extra write circuitry for the 'write' signal is a better solution to control the write of the long segment.

4.2.1.2 The I-cache Read Access

The value of a medium-long bit (M/L bit), which has been stored along with the compressed/uncompressed data during I-cache refill, indicates the size of an instruction to be read out during the cache read access. The instruction is medium size when an M/L bit is reset, while the instruction is long size when an M/L bit is set.

During the cache read access, the 'read' signal is enabled and the address decoder of the data array selects the appropriate row to read by driving a main wordline as well. When the main wordline goes high, the medium segment of the desired instruction is retrieved. The value of the corresponding M/L bit then decides whether the additional data in the long segment is also needed.

A local wordline, which connects to the long segment, will be enabled only when the main wordline is driven and the value of the corresponding M/L bit is set. This implies the long-size instruction format; therefore, the remaining bits in the long segment are read out. On the other hand, if the main wordline is driven but the M/L bit is reset, the local wordline will be disabled. Since this suggests the medium-size instruction format, no more data is available beyond what had been retrieved from the medium-size segment. The circuit which uses the M/L bit to control the size of the instruction fetching out is shown in Figure 4.4.

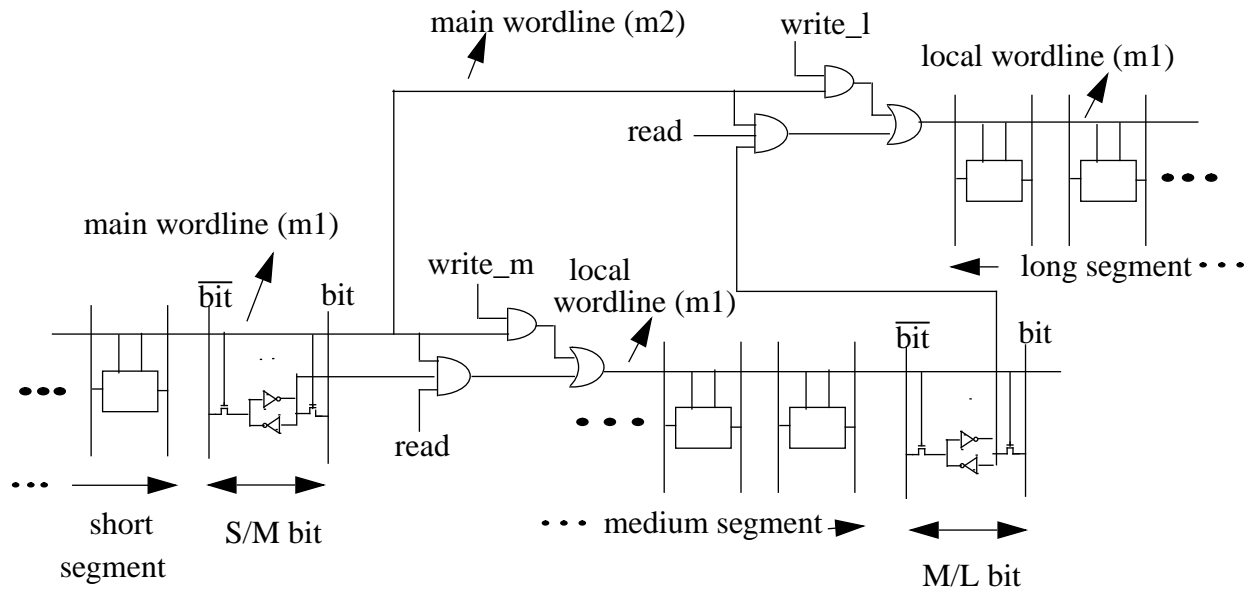


Figure 4.5: Style 1: Circuit to control the size of the instruction being 'written into' or read out from' SRAM array in the 3-size method. In this style, two levels of metal are required.

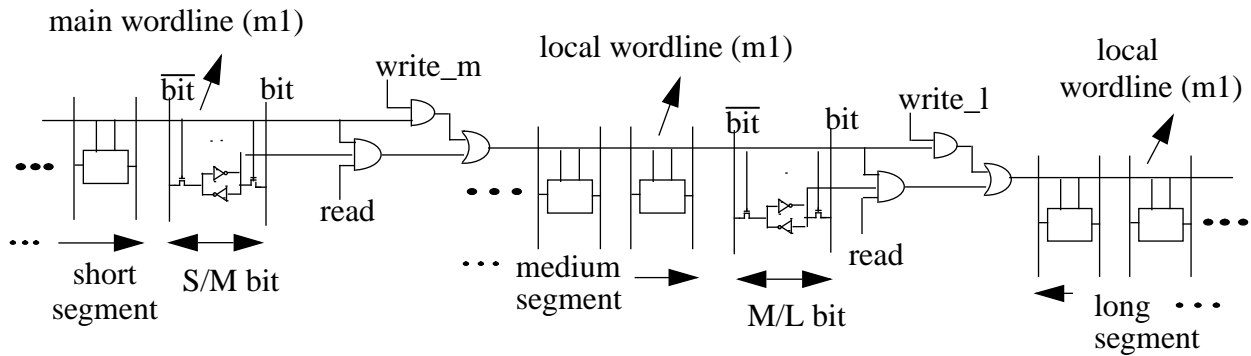


Figure 4.6: Style 2: Circuit to control the size of the instruction being 'written into' or 'read out from' SRAM array in the 3-size method. In this style, only one level of metal is required.

4.2.2 The 3-size Approach

In the 3-size approach, the short-size instruction is composed of a short segment and an S/M bit. The medium-size instruction makes up of all the components in the short-size instruction with an addition of a medium segment and an M/L bit. The long-size instruction comprises of all the components in the medium-size instruction with a long segment in addition.

In the same manner as the 2-size approach, the main wordline connects to the memory cells storing the short segment of the instruction and an S/M bit. Every main wordline is associated with two local wordlines. One local wordline connects to the memory cells storing an M/L bit and the rest of the bits that can not fit into the short size in the medium size (a medium segment). The other local wordline connects to the memory cells storing the rest of the bits in the long size instruction that can not fit into the short size and the medium size (a long segment).

4.2.2.1 The I-cache Refill

Each 32-bit instruction brought from memory is passed into a compression block during I-cache refill. By first checking the opcode of the incoming instruction, the detection circuit in the compression block resolves whether the instruction can be compressed. If the instruction is incompressible, the whole 32-bit instruction will be stored in the I-cache along with a set S/M bit and a set M/L bit. When the detection circuit comes across a compressible opcode, the instruction will be compressed to either a short size or a medium size, using the compression technique we will be discussed in 5.1.2.1. If the instruction is compressed into a short-size, the compressed instruction will then be stored in the I-cache along with a reset S/M bit and a reset M/L bit. On the other hand, if the instruction is compressed into a medium-size, the compressed instruction will be stored in the I-cache along with a set S/M bit and a reset M/L bit.

There are two ways to implement the circuit which uses both S/M bit and M/L bit to control the size of the instruction to be written. The first way requires two different levels of metal for implementation, as shown in Figure 4.5. Metal 1 is used for the main wordline and the two local wordlines. However, metal 2 is needed for the part of main wordline which crosses over to connect to the local wordline driver of the long segment. In this scheme, read/write operation of both medium and long segments are in parallel. The second way requires only one level of metal for implementation, as illustrated in Figure 4.6. But read/write operation of the long segment is in series with the medium segment. Therefore, this method is slower than the first one.

As soon as the value of the S/M and M/L bits are determined in the compression process, they are used differently depending on the circuit style, to control the circuitry of the 'write_m' and 'write_l' signals. The 'write_m' signal is responsible for storing of the medium segment while the 'write_l' signal is responsible for storing of the long segment. This control procedure of the two write circuitry happen prior to storing of bits in the short segment and the

S/M bit into the I-cache.

In the first circuit style, the M/L bit must always be updated in every cache refill because it directly controls storing of the long segment. Therefore, the 'write_m' signal must always be enabled in every cache refill so that the correct value of the M/L bit will always be written. The S/M bit can not be used to control the 'write_m' bit due to the problem caused by the parallel operations between the medium and long segments in this circuit style.

If the S/M bit is used instead to drive the 'write_m' signal, whenever the instruction is compressed into short-size, the 'write_m' signal will be disabled and the reset value of the M/L bit can not be stored. As a result, if the previously stored value of the M/L bit is set, during read access the long segment will be incorrectly read out and result in excess energy dissipation. Hence, the S/M bit can not be used to control the 'write_m' signal, and the 'write_m' signal must be driven in every cache refill.

Although the 'write_m' signal drives the local wordline which connects to the medium segment and the M/L bit in every cache refill, only the M/L bit is written every time but not the medium segment. The medium segment is written only when the S/M bit is assigned a set value (medium-size or long-size instruction). Otherwise, all the bitlines in the medium segment remain driven high so as not to change the SRAM state.

The 'write_l' signal on the contrary is controlled by the S/M and the M/L bit. The only time that the 'write_l' signal is enabled is when both S/M and M/L bits are assigned set values, which indicate a long-size instruction. In this case, the 'write_l' signal is driven as data needs to be stored in a long segment. In other cases, the 'write_l' remains disabled.

In the second circuit style, the 'write_m' signal is controlled by the S/M bit. If the S/M bit is assigned a reset value during compression time, which means that the instruction is a short size, the 'write_m' signal is disabled since there is no need to write the medium segment. On the other hand, if the S/M bit is assigned a set value, which means that the instruction is either a medium or a long size, the 'write_m' signal is enabled as the data will be written to the medium segment.

Whenever the S/M bit is reset and hence disables the 'write_m' signal, the M/L bit can not be stored. However, this is not a problem in this circuit style because the long segment is in series with the medium segment. During the read access with reset S/M bit, the local wordline

which connects to the medium segment and the M/L bit will not be driven; therefore, the local wordline of the long segment can not be driven as well. On the other hand, when the S/M bit is set, the M/L bit can be stored normally.

The 'write_1' signal is only enabled when the M/L bit is assigned a set value, which means long instruction, since the data will be stored in a long segment. Otherwise, the 'write_1' signal is disabled.

4.2.2.2 The I-cache Read Access

Similar to the 2-size approach, this method uses a short-medium bit (S/M bit) and a medium-long bit (M/L bit) to designate the size of the instructions to be retrieved. The instruction is a short size, when an S/M bit is reset. When an S/M bit is set and an M/L bit is reset, the instruction size is medium, and when both an S/M bit and an M/L bit are set, the instruction size is long.

During the cache read access, the 'read' signal is enabled and the address decoder of the data array selects the appropriate row by driving a main wordline. The data stored in the memory cells along the main wordline, which composed of bits in a short segment, will be read out. If the corresponding S/M bit is reset, only bits from a short segment are retrieved. If the S/M bit is set but the M/L bit is reset, one local wordline will be enabled. This local wordline drives the memory cells which store a medium segment. If the S/M bit and the M/L bit are both set, two local wordlines will be enabled. One local wordline is the same wordline that drives the memory cell storing the medium segment. The other local wordline drives the memory cells storing the long segment.

As mentioned previously, there are two ways to implement the circuit which uses both S/M bit and M/L bit to control the size of the instruction being fetched out, as shown in figure 4.5 and 4.6. In the first way, the signal in the local wordline of the long segment is fast since it is in parallel with the medium segment, but two levels of metal are needed. The second way requires only one level of metal for implementation; however, the signal in the local wordline of the long segment is slower than the first one since it is in series with the medium segment.

As we are concerned with the speed of the read/write operation we will use the first technique, which uses two different levels of metal for the local wordlines, to implement the circuit to control the size of the instruction.

Before discussing in detail about how we compress the instruction size in Chapter 5, we will review the MIPS RISC instruction set and at the same time examine the strategy on compressing the instruction size. Then, we will discuss the previous studies on code compression methods, on which our technique is based.

4.3 Review of MIPS Instruction Set

4.3.1 Type of Instructions

The MIPS-II instruction set is divided into the following types [12].

- *Computational* instructions perform arithmetic, logical, shift operations on values in register.

- *Load/store* instructions move data between memory and registers.

- *Jump and branch* instructions change the control flow of the program.

- *Coprocessor 0* instructions perform operations on CP0 registers. They handle memory management and exception handling.

- *Special* instructions move data between special and general registers, perform system calls and breakpoints.

4.3.2 Instruction Formats

There are three instruction formats in MIPS-II ISA, R-type (register), I-type (immediate) and J-type (jump) as shown in Figure 4.7.

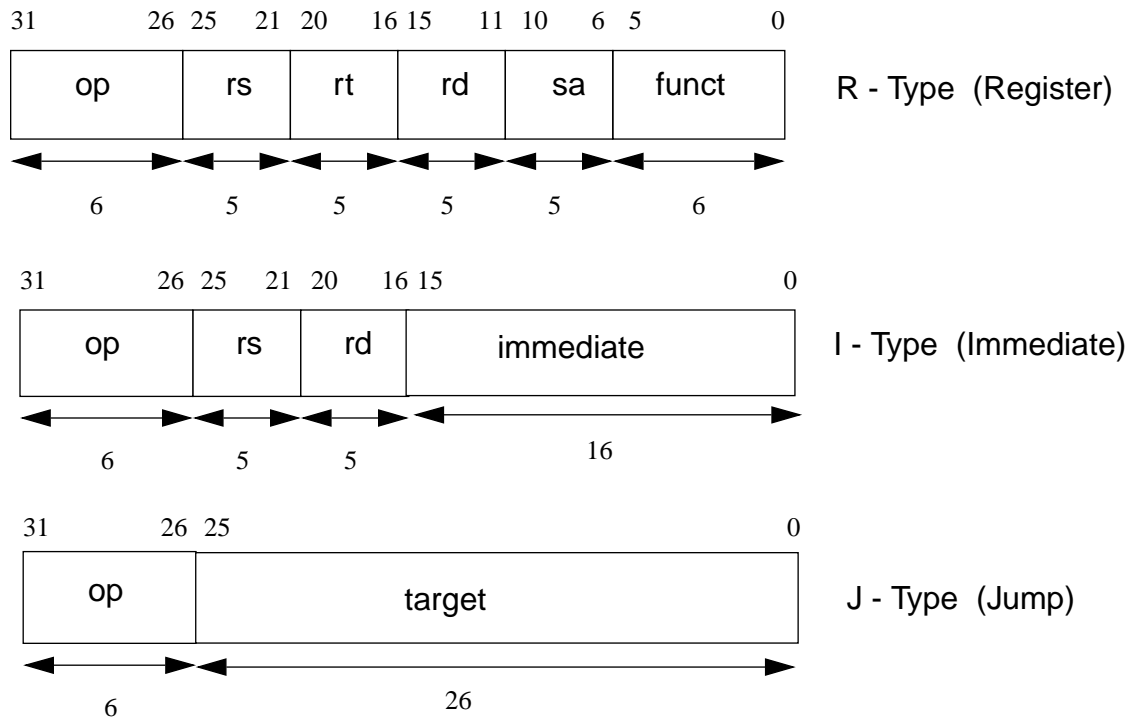


Figure 4.7: MIPS RISC instruction formats

Subfield	Definition
op	6-bit major operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Table 4.1: Subfield definition

		Opcode							
31...29	28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ	
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI	
2	COPO	*	*	*	BEQL	BNEL	BLEZL	BGTZL	
3	*	*	*	*	*	*	*	*	
4	LB	LH	*	LW	LBU	LHU	*	*	
5	SB	SH	*	SW	*	*	*	*	
6	*	*	*	*	*	*	*	*	
7	*	*	*	*	*	*	*	*	

		SPECIAL function							
5...3	2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV	
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC	
2	MFHI	MTHI	MFLO	MTLO	*	*	*	*	
3	MULT	MULTU	DIV	DIVU	*	*	*	*	
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR	
5	*	*	SLT	SLTU	*	*	*	*	
6	*	*	*	*	*	*	*	*	
7	*	*	*	*	*	*	*	*	

		REGIMM rt							
20...19	18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*	
1	*	*	*	*	*	*	*	*	
2	BLTZAL	BGEZAL	BLTZALIBGEZAL		*	*	*	*	
3	*	*	*	*	*	*	*	*	

Figure 4.8: MIPS-II instruction encodings of integer subset [3]

4.3.2.1 R-type Instructions

All R-type (Register) instructions except MFCO and MTCO have a 6-bit opcode modified by a 6-bit function code. MFCO and MTCO do not use the function specifier, as a result the contents of their function codes are zero. If the instructions are re-encoded so that no function specifier is needed for modification of opcode, six bits can be saved. Moreover, every instruction in this format has at least one unused 5-bit subfield. For example in ADD instruction, shift amount subfield (sa) is unused. Further bits saving is possible if the unused subfield(s) in the instructions is(are) discarded with re-encoding. Hence, R-type instructions can be compressed by re-encoding to get rid of the function specifier and unused subfield(s).

We separate the R-type instructions into the following sub-groups as shown in Table

Type	Abbr	Used Subfield(s)	Unused Subfield(s)	Instructions
R1 Arithmetic instructions (3-operand R-type)	R1	rs,rt,rd	sa	ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
R2 Shift Instructions	R2.1	rt, rd,sa	rs	SLL, SRL, SRA
	R2.2	rs, rt, rd	sa	SLLV, SRLV, SRAV
R3 Multiply/divide Instructions	R3.1	rs, rt	rd,sa	MULT, MULTU, DIV, DIVU
	R3.2	rd	rs, rt,sa	MFHI, MFLO
	R3.3	rs	rt,rd,sa	MTHI, MTLO
R4 Jump Instructions	R4.1	rs	rt,rd,sa	JR
	R4.2	rs, rd	rt,sa	JALR
R5 Special Instructions	R5.1	none	rs,rt,rd,sa	SYSCALL
	R5.2	(rd and sa are coded)	rs,rt	BREAK
R6 System Control Coprocessor Instructions	R6.1	rt, rd, (rs encoded)	sa	MTCO, MFCO
	R6.2	(rs encoded)	rt,rd,sa	RFE
	R6.3	no info	no info	ICINV
R7 Load/Store Instructions	R7	none	rs,rt,rd,sa	SYNC

Table 4.2: Sub-groups of R-type instructions

4.3.2.2 I-type Instructions

All I-type (Immediate) instructions include a 16-bit immediate. These immediates are either zero-extended or sign-extended. Consequently, by removing leading zeros or leading sign bits, the immediates can be shortened to contain only significant bits. Like R-type, some of the I-type instructions contain unused subfields. Re-encoding to discard the unused subfields, therefore, helps with bit saving in some I-type instructions. Accordingly, I-type instructions can be compressed by reducing the size of immediate field along with re-encoding to remove the unused subfields.

I-type instructions are divided into the following sub-groups as shown in Table 4.3.

Type	Abbr	Used subfield(s)	Unused subfield(s)	Instructions
I1 Arithmetic Instructions (ALU immediate)	I1.1	rs,rt,imm	none	ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI
	I1.2	rt,imm	rs	LUI
I2 Branch Instructions	I2.1	rs,rt,imm	none	BEQ, BNE, BEQL, BNEL
	I2.2	rs, imm	rt	BLEZ, BGTZ, BLEZL, BGTZL
	I2.3	rs, imm (rt encoded)	none	BLTZ, BGEZ, BLTZL, BGEZL, BLTZAL, BGEZAL, BLTZALL, BGEZALL
I3 Load/Store Instructions	I3	rs, rt, imm	none	LB, LH, LW, LBU, LHU, SB, SH, SW

Table 4.3: Sub-groups of I-type instructions

Table 4.4 summarizes how the immediates of I-type instruction are extended.

Type	Abbr	Instructions	Extension of Immediate
I1 Arithmetic Instructions (ALU immediate)	I1.1	ADDI, ADDIU, SLTI, SLTIU ANDI, ORI, XORI	signed zero
	I1.2	LUI	zero
I2 Branch Instructions	I2.1	BEQ, BNE, BEQL, BNEL	signed
	I2.2	BLEZ, BGTZ, BLEZL, BGTZL	signed
	I2.3	BLTZ, BGEZ, BLTZL, BGEZL, BLTZAL, BGEZAL, BLTZALL, BGEZALL	signed
I3 Load/Store Instructions	I3	LB, LH, LW, LBU, LHU, SB, SH, SW	signed

Table 4.4: Immediate extension of the I-type instructions

4.3.2.3 J-type instructions

J-type (Jump) instructions have 26-bit targets which are zero-extended. By eliminating leading zeros, the target can be shrunk to retain only significant bits. Thus, J-type instructions can be compressed by reducing the size of the target field.

Table 4.5 presents the only sub-group of J-type instructions. The extension of target field is summarized in Table 4.6.

Type	Abbr	Used Subfield(s)	Unused Subfield(s)	Instructions
J1 Jump Instruction	J1	target	none	J, JAL

Table 4.5: J-type instructions

Type	Abbr	Instructions	Extension of Target
J1 Jump Instruction	J1	J, JAL	zero

Table 4.6: Target extension of J-type instructions

4.4 Previous study on Code Compression

Embedded systems are highly constrained by cost, power and size. A considerable part of circuitry is used for instruction memory. As larger memory means bigger and more expensive die, smaller program size can therefore reduce the cost of embedded systems. Nowadays, the code size of a program has grown and become considerably large. The growth in code size impacts embedded system not only on increasing cost but also by lower performance due to delays from I-cache misses [15]. This problem will only become worse as the disparity in performance between processor and memory increases. One solution to reduce the I-cache misses as well as reduce the cost from larger memory is to decrease the code size of a program.

4.4.1 Approaches for Code Size Reduction

Many approaches have been proposed to solve the code-size problem including [1]:

- *Hand code in assembler*: This technique requires the designer to hand-code assembler for code-size optimization. This approach, however, is impractical because of the amount of time required to produce code.

- *Improve the compiler*: Another option is to improve the compiler technology. Nevertheless, the code size from this method will never be smaller than the code size from hand-coding.

- *Use compressed code*: This method requires compressing the code and decompressing it back in the decode stage of the pipeline. The draw back of this technique is the slower execution speed due to compression and the additional hardware overhead required.

Our study on using gated wordlines to reduce the number of bitline swings, as a way to reduce power dissipation in the SRAM array of the I-cache, is based on the technique of using

compressed code. In the next section we will, therefore, discuss other techniques on using compressed code; short instruction encodings and Dictionary compression. However, while our study is focused on reducing the instruction size to lower the dynamic power dissipated from the bitlines of the SRAM array in the I-cache, their studies emphasize on reducing the code size to decrease the delay from I-cache misses, provide higher instruction bandwidth and reduce size and cost of program memory.

4.4.1.1 Short Instruction Encodings

The concept is based on the fact that several instructions in the RISC instruction set do not require full 32-bit to represent. However, the 32-bit fixed-length format is used for every instruction because of the ease of fetch and decode but this results in wasted memory space. If the size of instructions is reduced, more instructions can be stored in memory as well as cache.

MIPS16 and Thumb are two instruction set modifications which reduce instruction sizes in order to reduce the total size of the compiled program [15]. Thumb and MIPS16 are defined as an extension of ARM architecture and MIPS-III architecture respectively.

In our study on using gated wordline to reduce the number of bitline swings, we based our development on MIPS16. Therefore, in this part we will cover MIPS16 in detail.

4.4.1.1.1 MIPS16

One proposed idea for reducing the size of the instructions is the MIPS16 architecture, the subset of MIPS-III architectures, by MIPS Inc. It is designed to be fully compatible with 32-bit (MIPS I/II) and 64-bit (MIPS-III) architecture.

MIPS16 architecture reduces the size of the 32-bit instructions to 16-bit. Each MIPS16 instruction has a one to one correspondence with MIPS-III instruction. The translation from MIPS16 to MIPS-III uses simple hardware between I-cache and the decode stage as in Figure 4.9

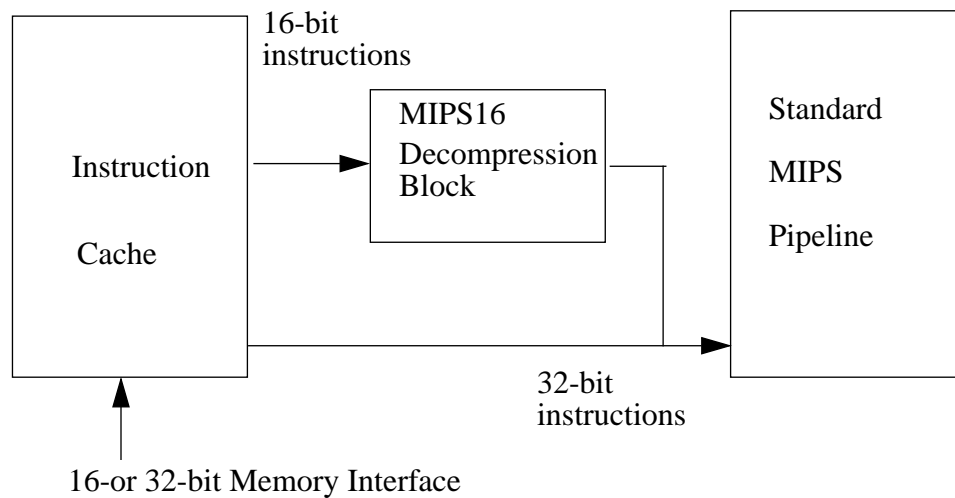


Figure 4.9: MIPS 16 Decompression [14]

MIPS16 reduces the instruction size by shrinking all three components of the instruction word, opcodes, register numbers and immediate size. Opcode and function modifier are both reduced from original 6 bits to 5 bits, and define 79 instructions in total. The number of registers in use are restricted to only 8 registers out of 32 base registers. This, therefore, allows MIPS16 to use 3 bits for register specifier instead of 5 bits. In R-type instruction format, only 2 operands, are supported in MIPS16, compared to 3 operands in MIPS-III. This means that one source register must also be used as the destination register.

The constraint on the size of immediate contributes to the most saving. I-type instructions have immediate size ranges from 3-bit, 4-bit and 5-bit, with the majority being 5-bit immediate, compared to 16-bit in MIPS-III.

Switching between MIPS16 compressed instruction mode to 32-bit MIPS-III mode is done by the JALX instruction (Jump And Link with eXchange). JALX instruction toggles the state of the instruction decode logic between 32-bit mode and 16-bit mode [14].

The instruction of MIPS16 is shrunk with the price of reducing the number of bits representing registers and immediate values. The programs are limited to use only 8 registers of the base architecture and the range of immediate values are reduced substantially. Consequently, MIPS16 are unable to use the full capability of the underlying processor.

MIPS16 achieves the net code saving of 40% across a range of embedded and desktop

codes [14]. It is evident from these statistics that MIPS16 is effective in reducing the code size. However, the statistics also suggest that the number of instructions in a program increases. If the number of the instructions did not increase, it would be possible to obtain 50% in code reduction. This observation is indeed correct as reported in Kissell [14], who observes that in MIPS16 more instructions are required to perform some of the operations. The increase in the number of the instructions necessitates a program to execute more instructions; therefore, reducing the performance [15].

MIPS16 reduces the size of instructions in order to lower the memory capacity requirement of the hardware, while our technique compresses the size of instructions and uses gated wordlines to reduce the number of bitline swings in the SRAM array of the I-cache. We can combine our proposed technique with MIPS16 so that both the memory requirement and the I-cache power dissipation can be reduced. MIPS16 provides the run-time switching between the 16-bit compressed mode and the 32-bit mode. By applying our technique to the 32-bit mode, when the processor is in the 32-bit mode and hence not using MIPS16, power dissipated in the I-cache can be lowered without reducing performance.

4.4.1.2 Code Compression Using a Dictionary

Generally, object code generated by a compiler is repetitive. Therefore, a compiled program can be compressed by using a post-compilation analyzer to find the redundant instruction sequences in the instruction stream and replacing each complete sequence with a single codeword, as proposed in Lefurgy et. al. [15]. In their technique, the sequences of instructions which are replaced with codewords are saved in a Dictionary. During program execution, the instruction decoder in the decode stage checks the opcode of the instruction fetched from the I-cache. If the opcode indicates a compressed codeword, the Dictionary will expand the codeword in the instruction stream back to the original sequences of instructions which are issued to the execution stage sequentially. Instruction fetch from the I-cache is stalled until the sequences are complete. On the other hand, if the opcode detects an uncompressed instruction, the instruction decode and execution proceeds in a normal manner.

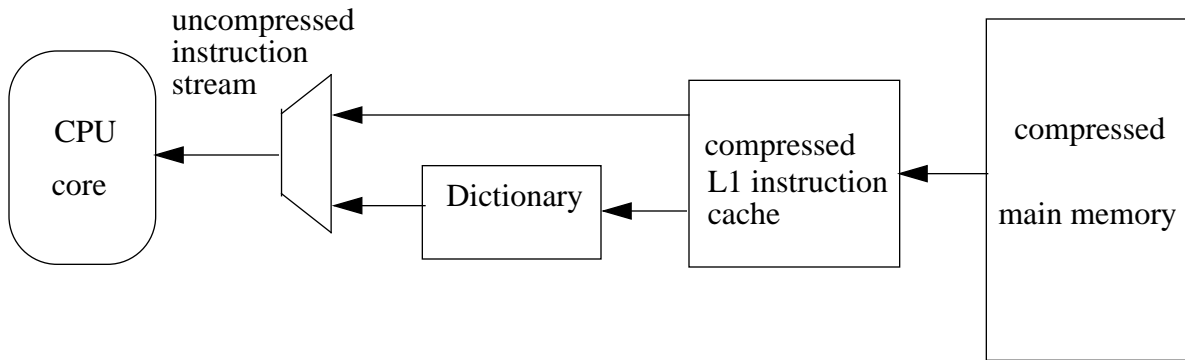


Figure 4.10: CPU layout with Dictionary for expansion of instructions [15] [6]

This method is applied to the Power PC, ARM and i386 ISAs using SPEC CINT95 benchmark programs. An average code size reduction of 39%, 34% and 26% is attained respectively. Unlike MIPS16, this approach does not increase the number of instructions in a program. The compiled program is compressed and translated back to the original uncompressed program. As a result, the number of instructions executed in the program is not modified. In addition, this technique has the complete access to the resources of the underlying processor. The compressed program can access all the registers, operations and modes available as in the base architecture [15].

The disadvantage of this approach, however, lies in the fact that more bits need to be fetched, including not only the codeword but also the original sequences of instructions translated back from that codeword. Hence, more power is consumed.

This technique of code compression using a Dictionary can be further improved if combined with our method of using gated wordline to reduce the number of bitline swings. Our approach can be used to compress the size of the uncompressed instructions and the codewords that will be stored in the I-cache as well as the sequences of instructions that will be stored in the Dictionary. Our method will prevent the storing of unnecessary bits into the SRAM arrays of the I-cache and the Dictionary, and as a result eliminate the power dissipated from reading or writing those unused bits. With the integration of our scheme, this technique can not only effectively compress the code size but also can reduce dynamic power dissipation from its I-cache.

Chapter 5

Experimental Study on Using Gated Wordlines to Reduce the Number of Bitline Swings

In this last part of the thesis, we have developed the technique of using gated wordline to reduce the number of bitline swings which hence decrease the number of bits read out from the I-cache, as was discussed in Chapter 4. Our approach reduces the instruction size, a method that is also used in MIPS16. However, unlike MIPS16 in which every instruction is shortened to 16 bits, we shrink the instructions only when we are still able to retain full access to the resources of the underlying processor. Otherwise, the instructions will continue to be full-length. The extra bit(s), which is stored along with the instructions during the cache refill, will be used to indicate the size of the instruction to be read out by gating wordline. Our technique, therefore, allows us to reduce power dissipated in the SRAM array of the I-cache from reading or writing the unnecessary bits of the instructions that do not require full 32 bits.

In the next sections we will discuss the compression techniques used for reducing the size of instructions in the 2-size and the 3-size versions of our gated wordline technique. Then, we will discuss our experimental results obtained from applying our technique to the MIPS-II instruction set.

5.1 Experimental Study

5.1.1 The 2-size Approach

5.1.1.1 Compression Technique for Medium Size

In order to achieve high bits saving, it is imperative to use medium size for the instructions as much as possible. This requires re-encoding to eliminate unnecessary bits for some R-type and I-type instructions as well as reducing the immediate size and target size for I-type and J-type instructions respectively.

In R-type instructions, both the 6-bit function modifier and one 5-bit unused subfield can be discarded with the new encoding. Some of the I-type instructions are composed of an unused subfield while some have the source register equal to the target register which can hence be combined to use only one register for both source and target. Therefore, these I-type instructions can also use re-encoding to get rid of a 5-bit subfield. However, by consulting MIPS-II opcode bit encoding table shown in Figure 4.8, we found that there are only thirty-four entries available for instruction re-encoding. Hence, only thirty-four instructions are selected for re-encoding based on the frequency of occurrence in benchmark programs.

From the statistics of the benchmark program, we have selected the following frequently occurred instructions for re-encoding.

R1: ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU

R2_1: SLL, SRL, SRA

R2_2: SLLV, SRLV, SRAV

R4_1: JR

R4_2: JALR

I1_1: ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI

I2_1: BEQ, BNE, BEQL, BNEL

I1_2: LUI

R3_1: MULT, MULTU

R3_2: MFHI, MFLO

		Opcode							
31...29	28...26	0	1	2	3	4	5	6	7
0		SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1		ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2		COP0	JR(R)	*	JALR(R)	BEQL	BNEL	BLEZL	BGTZL
3		ADD(R)	ADDU(R)	SUB(R)	SUBU(R)	AND(R)	OR(R)	XOR(R)	NOR(R)
4		LB	LH	SLL(R)	LW	LBU	LHU	SRL(R)	SRA(R)
5		SB	SH	SLLV(R)	SW	SLTU(R)	SLT(R)	SRLV(R)	SRAV(R)
6		MULT(R)	MULTU(R)	MFHI(R)	MFLO(R)	BEQ(R)	BNE(R)	BEQL(R)	BNEL(R)
7		ADDI(R)	ADDIU(R)	SLTI(R)	SLTIU(R)	ANDI(R)	ORI(R)	XORI(R)	LUI(R)

		SPECIAL function							
5...3	2...0	0	1	2	3	4	5	6	7
0		*	*	*	*	*	*	*	*
1		*	*	*	*	SYSCALL	BREAK	*	SYNC
2		*	MTHI	*	MTLO	*	*	*	*
3		*	*	DIV	DIVU	*	*	*	*
4		*	*	*	*	*	*	*	*
5		*	*	*	*	*	*	*	*
6		*	*	*	*	*	*	*	*
7		*	*	*	*	*	*	*	*

		REGIMM rt							
20...19	18...16	0	1	2	3	4	5	6	7
0		BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1		*	*	*	*	*	*	*	*
2		BLTZAL	BGEZAL	BLTZAL	BGEZAL	*	*	*	*
3		*	*	*	*	*	*	*	*

Figure 5.1: CPU instruction encodings of integer subset for the 2-size method (Note: (R) stands for re-encoded)

It should be noted that in order to re-encode instructions for compression purpose, we have used all the illegal opcodes which cause exception (previously designated by asterisk “ * ”), as shown in Figure 4.8. Therefore, every time we encounter an illegal opcode before doing a compression, we have to re-encode them to use BREAK as their opcode instead.

Among the thirty-four instructions that we have chosen, twenty-two instructions are R-type. Therefore all twenty-two R-type are re-encoded so that both 6-bit function modifier and the unused 5-bit subfield can be removed. After re-encoding in which 11 bits are saved, these R-type instructions need only 22-bit to represent (21 bits of instruction with an M/L bit). The twelve remaining instructions are I-type. I1_2 (LUI) instructions always have rs field equal to

zero; therefore, after re-encoding only *rt* field remain. From the instruction analysis, we discovered that I1_1 (ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI) instructions often have source register (*rs*) equal to target register (*rt*). Consequently, when I1_1 instructions have *rs* equal to *rt*, they are re-encoded to have only 5-bit *rs* field without the redundant *rt* field. Another discovery is that the majority of I2_1 (BEQ, BNE, BEQL, BNEL) instructions have target register (*rt*) equal to zero. Accordingly, when I2_1 instructions have *rt* equal to zero, they are re-encoded with only *rs* field left. As a result, these twelve instructions with the right condition can eliminate the insignificant subfield, thereby saving 5-bit even before immediate field is reduced.

As mentioned previously, I-type and J-type instructions have immediate and target that can be shortened to contain only significant bits. We use the 22-bit instruction size from the re-encoded R-type instructions as a baseline to find the suitable immediate size for the medium-size instructions. Accordingly, the 22-bit I-type instruction has a 5-bit immediate; therefore, we examine each potential immediate size ranging from 5-bit up to 15-bit to find the immediate size which results in the largest bits saving. Note that for the twelve re-encoded I-type instructions, their baseline size for the immediates is 10-bit due to 5-bit saving achieved from removing the unnecessary subfield. As for other instructions which re-encoding and reducing the immediate and target size can not cover, they remain long size which are 33 bits; 32 bit instruction with a set M/L bit. Section 5.2 covers in detail our method to find the size of the immediate field of the medium-size instruction that results in the biggest bits saving.

5.1.2 The 3-size Approach

Aiming for bits saving higher than that achieved by the 2-size method, we proceed with the technique of having instructions in one of three sizes; short, medium and long. We followed MIPS 16 by using 16 bits to represent a short instruction, but required an additional reset S/M bit for size indication. Therefore, our short-size instruction is 17-bit, one bit longer than the MIPS16 instruction. The long size instruction stays 32-bit full-length with a set S/M bit and a set M/L bit. As a result, a long-size instruction is 34 bits in total, a bit more than the one in the 2-size approach. The optimized size of the medium instruction will be determined in section 5.2.

5.1.2.1 Compression Technique for Short Size

In addition to reduce the immediate size and the target size, as previously demonstrated in the 2-size method, better re-encoding strategy is required for short-size compression.

In the 2-size method, the re-encoded R-type instructions have both the 6-bit function modifier and the 5-bit unused subfield discarded. In order to reduce the R-type instruction to 16-bit, an additional 5 bits must also be removed. Instruction analysis has been done on some of the R-type and the I-type instructions which have been targeted for re-encoding to further eliminate the 5-bit unused subfield or the 5-bit redundant subfield, for which the source register is equal to the destination register.

Based on the statistics of the benchmark program, we have selected the following twenty-nine frequently occurring instructions for re-encoding.

R1: ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU

R2_1: SLL, SRL, SRA

R2_2: SLLV, SRLV, SRAV

R4_1: JR

R4_2: JALR

I1_1: ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI

I2_1: BEQ, BNE, BEQL, BNEL

		Opcode							
		28...26							
31...29		0	1	2	3	4	5	6	7
0		SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1		ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2		COP0	SB	SH	SW	BEQL	BNEL	BLEZL	BGTZL
3		LBU	LB	LH	LW	LHU	*	*	*
4		*	*	BEQL(R)	BNEL(R)	BLEZL(R)	BGTZL(R)	JR(R)	JALR(R)
5		SLL(R)	SRL(R)	SLT(R)	SLTU(R)	SRA(R)	SLLV(R)	SRLV(R)	SRAV(R)
6		ADD(R)	ADDU(R)	SUB(R)	SUBU(R)	AND(R)	OR(R)	XOR(R)	NOR(R)
7		ADDI(R)	ADDIU(R)	SLTI(R)	SLTIU(R)	ANDI(R)	ORI(R)	XORI(R)	*

		SPECIAL function							
		2...0							
5...3		0	1	2	3	4	5	6	7
0		*	*	*	*	*	*	*	*
1		*	*	*	*	SYSCALL	BREAK	*	SYNC
2		MFHI	MTHI	MFLO	MTLO	*	*	*	*
3		MULT	MULTU	DIV	DIVU	*	*	*	*
4		*	*	*	*	*	*	*	*
5		*	*	*	*	*	*	*	*
6		*	*	*	*	*	*	*	*
7		*	*	*	*	*	*	*	*

		REGIMM rt							
		18...16							
20...19		0	1	2	3	4	5	6	7
0		BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1		*	*	*	*	*	*	*	*
2		BLTZAL	BGEZAL	BLTZALB	BGEZALI	*	*	*	*
3		*	*	*	*	*	*	*	*

Figure 5.2: CPU instruction encodings of integer subset for the 3-size method (Note: (R) stands for re-encoded; The opcode for Load/Store instructions and the opcode for COP0 have changed places so that the new encoded opcodes can locate in the bottom half of the table.)

We have summarized the techniques we have used to compress the size of the instruction into 16-bit as follows:

5.1.2.1.1 Technique to compress R1 (ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU) and R2_2 (SLLV, SRLV, SRAV)

Out of thirteen instructions in these two sub-groups, register ordering matters for seven instructions; SUB, SUBU, SLT, SLTU, SLLV, SRLV, SRAV. The position of the rs field and the rt field in these instructions can not be altered; otherwise, the result is incorrect. For example, ‘SUB r1, r2, r3’ means that the contents of r3 (rt) being subtracted from the contents of r2 (rs) to form a

result. When the position of r2 and r3 are exchanged to ‘SUB r1, r3, r2’, the result is changed to the contents of r2 (rt) being subtracted from the contents of r3 (rs).

With detailed examination, we have found that often source register (rs) is equal to destination register (rd) and also that target register (rt) is equal to rd. Therefore, the rd subfield is redundant in both cases. However, when register ordering matters, we have to choose between the two cases; either use an rs subfield for both rs and rd, or use an rt subfield for both rt and rd. Further analysis reveals that rs equal to rd is more frequent than rt equal to rd. Consequently, for those seven instructions, in which register ordering matters, rd subfield will be discarded when the rs and rd subfields are equal. As for the rest of the instructions in these two categories, rd subfield can be removed in both cases. With the elimination of 5-bit redundant rd subfield, 6-bit function specifier and 5-bit unused sa subfield in the new encoding, the instructions can shrink to 16-bit. Other instructions in these two sub-groups with no redundant rd subfield remain medium-size.

5.1.2.1.2 Technique to compress R2_1 (SLL, SRL, SRA)

Instructions in this type frequently have the target register (rt) same as the destination register (rd). With this condition, rt and rd can be compressed into one subfield, 5 more bits are saved besides the 6 bits from function specifier and the 5 bits from rs subfield. The instructions with rt equal to rd, therefore, can be re-encoded to shorten the size to 16-bit. On the other hand, when rt is not equal to rd, the instructions can not be compressed any further and the instruction stays medium-size.

5.1.2.1.3 Technique to compress R4_1 (JR) and R4_2 (JALR)

It is relatively easy to compress JR, since there are as many as three unused subfields. Yet, we discard only two unused subfields, rd and sa, so that the short size is 16 bits, same as the others. JALR also has two unused subfields which are thrown away during re-encoding. Hence, these two instructions are always short-size.

5.1.2.1.4 Technique to compress I1_1 (ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI)

As mentioned previously in the 2-size approach, I1_1 instructions often have source register (rs) equal to target register (rt). When this is true, they are re-encoded to have only the 5-bit rs subfield without the redundant rt subfield. With this condition of rs equal to rt, another 11 bits must still be eliminated before the instruction can be short-size. As a consequence, if the

instructions have both rs equal to rt and also immediate size not exceeding 5 bits, they can fit into a short-size format.

When rs is equal to rt, but the immediate size exceeds 5 bits, the instructions can be either a medium size or a long size depending on the size of the immediate. If the immediate size does not surpass the sum of the immediate size of the medium instruction and the additional 5 bits, the instructions are medium-size; else, they are long-size.

On the contrary, when rs is not equal to rt, the instructions can be either a medium size or a long size based on the size of the immediate. If the immediate size does not exceed the immediate size of the medium instruction, the instructions are medium-size; otherwise, they are long size.

Size	Condition
short	$RS = RT \ \& \ \text{immediate size} \leq 5\text{-bit}$
medium	<ol style="list-style-type: none"> 1. $RS = RT \ \& \ 5\text{-bit} < \text{immediate size} \leq 5\text{-bit} + \text{medium immediate size}$ 2. $RS \neq RT \ \& \ \text{immediate} \leq \text{medium immediate size}$
long	<ol style="list-style-type: none"> 1. $RS = RT \ \& \ \text{immediate} > 5\text{-bit} + \text{medium immediate size}$ 2. $RS \neq RT \ \& \ \text{immediate} > \text{medium immediate size}$

Table 5.1: Compression condition for I1_1 (ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI)

5.1.2.1.5 Technique to compress I2_1 (BEQ, BNE, BEQL, BNEL)

As discovered in the 2-size method, the majority of these branch instructions have zero-valued rt. They are, therefore, re-encoded without the 5-bit rt field. The instructions in this type can be expressed in short format, only when the instructions have zero-valued rt and their immediate size does not exceed 5 bits.

If instructions have zero-valued rt, though the immediate size exceeds 5 bits, the instructions can be either a medium size or a long size contingent upon the size of the immediate. If the immediate size does not surpass the sum of the immediate size of the medium instruction and the additional 5 bits, the instructions are medium-size; else, they are long-size.

In contrast, when the instructions do not have zero-valued rt to begin with, the instructions fall into either a medium size or a long size depending on the size of the immediate.

If the immediate size does not exceed the immediate size of the medium instruction, the instructions are medium-size; otherwise, they are long size.

Size	Condition
short	$RT = 0 \ \& \ \text{immediate size} \leq 5\text{-bit}$
medium	1. $RT = 0 \ \& \ 5\text{-bit} < \text{immediate size} \leq 5\text{-bit} + \text{medium immediate size}$ 2. $RT \neq 0 \ \& \ \text{immediate} \leq \text{medium immediate size}$
long	1. $RT = 0 \ \& \ \text{immediate} > 5\text{-bit} + \text{medium immediate size}$ 2. $RT \neq 0 \ \& \ \text{immediate} > \text{medium immediate size}$

Table 5.2: Compression condition for I2_1 (BEQ, BNE, BEQL, BNEL)

5.1.2.1.6 Technique to compress I3 (LB, LH, LW, LBU, LHU, SB, SH, SW)

The word size in 32-bit architecture is 4 bytes. Load/Store allow byte offset but there is almost always word offset for LW, SW and halfword offset for LH, LHU, SH. Thus, the two least significant bits in the immediate of LW and SW are almost always zero due to word offset. Likewise, the least significant bit of LH, LHU and SH are almost always zero due to halfword offset. In SPECint95 benchmark programs that we used, we found that there are always word and halfword offsets. Accordingly, we disregard bit 0 in the immediate of LH, LHU, SH and start probing the immediate from bit 1 as a starting point. In the same manner, bit 0 and bit 1 can be omitted in the immediate of LW and SW. This technique of using word and halfword offsets is also used in MIPS16. The compression conditions for I3 instructions are as shown in Table 5.3.

Size	Condition
medium	LB, LBU, SB: $\text{immediate size} \leq \text{medium immediate size}$ LH, LHU, SH: $\text{immediate size} \leq \text{medium immediate size} + 1$ LW, SW: $\text{immediate size} \leq \text{medium immediate size} + 2$
long	LB, LBU, SB: $\text{immediate size} > \text{medium immediate size}$ LH, LHU, SH: $\text{immediate size} > \text{medium immediate size} + 1$ LW, SW: $\text{immediate size} > \text{medium immediate size} + 2$

Table 5.3: Compression condition for I3 (LB, LH, LW, LBU, LHU, SB, SH, SW)

The rest of R-type instructions besides R1, R2_1, R2_2, R4_1 and R4_2 are 34-bit long format. I-type instructions other than I1_1, I2_1, I3, which are I1_2, I2_2 and I2_3, are either medium format or long format based on whether it exceeds the immediate size of the medium instructions. Table 5.4 summarizes the compression condition for I1_2, I2_2 and I2_3 instructions.

Size	Condition
medium	immediate size \leq size of medium immediate size
long	immediate size $>$ size of medium immediate size

Table 5.4: Compression condition for I2_1, I2_2, and I2_3 instructions (LUI, BLEZ, BGTZ, BLEZL, BGTZL, BLTZ, BGEZ, BLTZL, BGEZL, BLTZAL, BGEZAL, BLTZALL, BGEZALL)

5.2 Experimental Methods

5.2.1 Simulation

Table 3.1 shows the applications we have used in our evaluations. We compiled each benchmarks with GCC 2.7.0 using -O3 optimization. The benchmarks were simulated using an ISA-level simulator written in C++ which integrates the techniques used to reduce the size of the instruction set in both approaches. The frequency distribution of each instruction, dynamic compression ratio and the percentage of reduction in bits read out were recorded.

The dynamic compression ratio is defined by the following formula [15]

$$\text{Dynamic compression ratio} = \frac{\text{bits read out after compression}}{\text{original bits read out}}$$

In order to determine the immediate length of the medium-size instruction that achieves the largest bits saving in both approaches, we use the variable-length immediate in the simulation, as shown in Figure 5.3 and 5.4. We examine the potential immediate length ranging from 5 bits to 15 bits. This results in the medium-size instruction varying from 22 bits to 32 bits in the 2-size method and 23 bits to 33 bits in the 3-size method. The long-size instructions are fixed-length 33 bits and 34 bits in the 2-size and the 3-size techniques respectively. The short-size instruction for the 3-size approach is fixed at 17 bits.

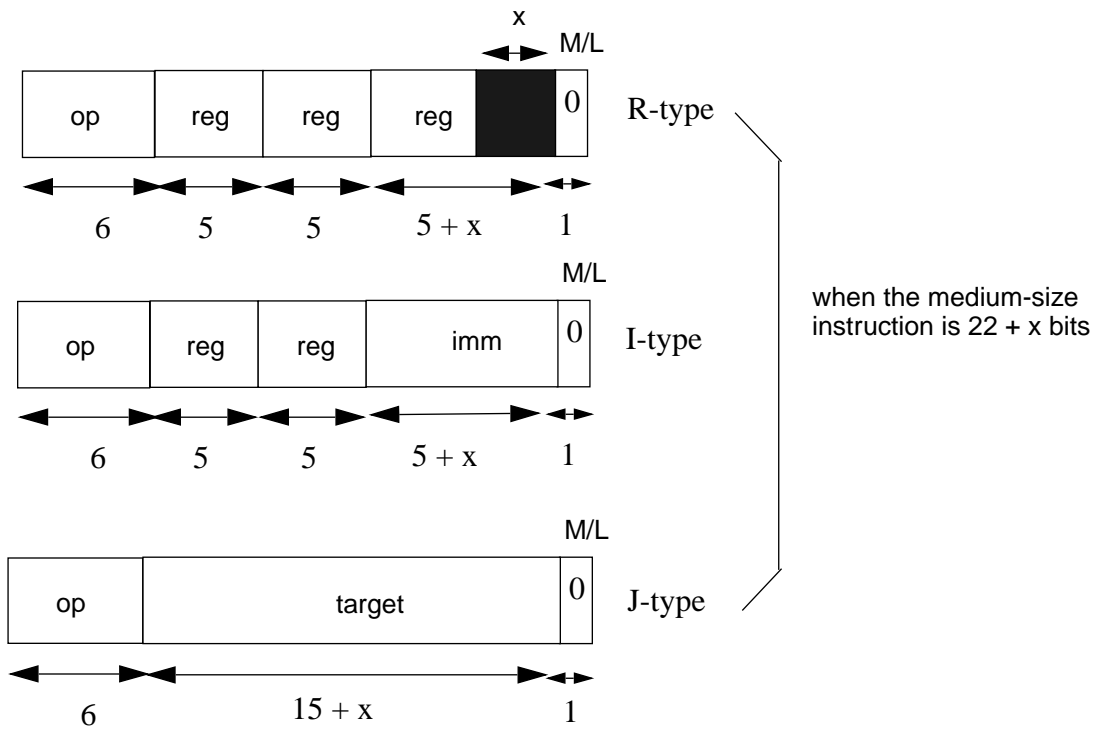


Figure 5.3: Variable-length immediate of medium-size instruction in the 2-size approach

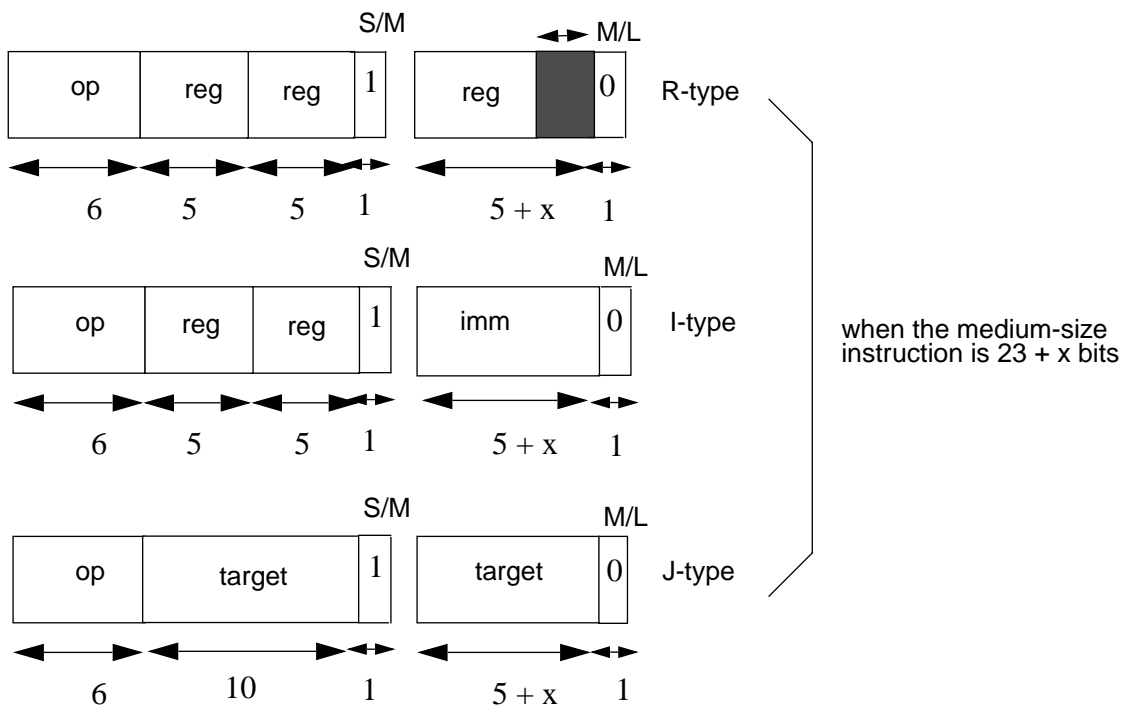


Figure 5.4: Variable-length immediate of medium-size instruction in the 3-size approach

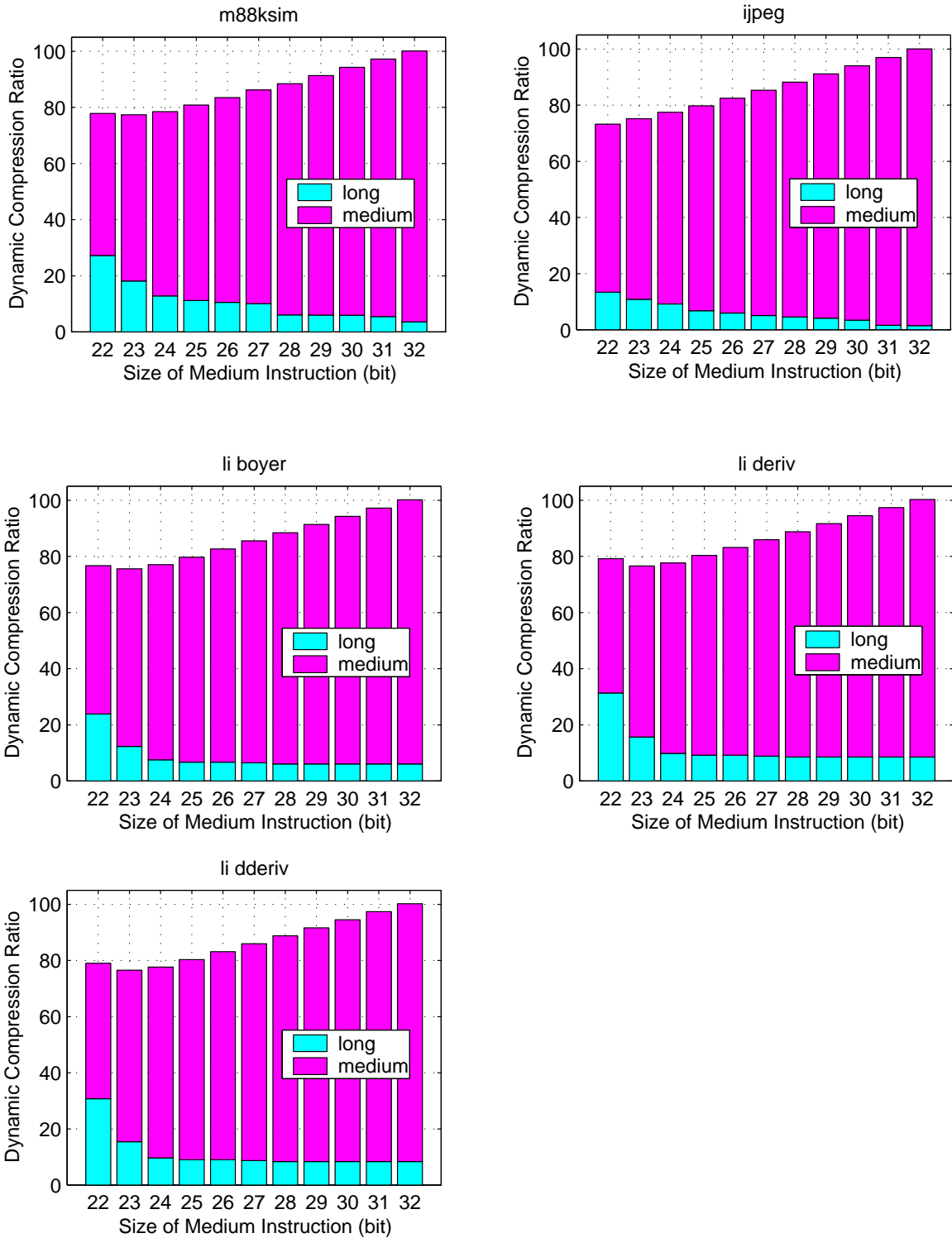


Figure 5.5: Dynamic compression ratio of various benchmark programs using the 2-size approach

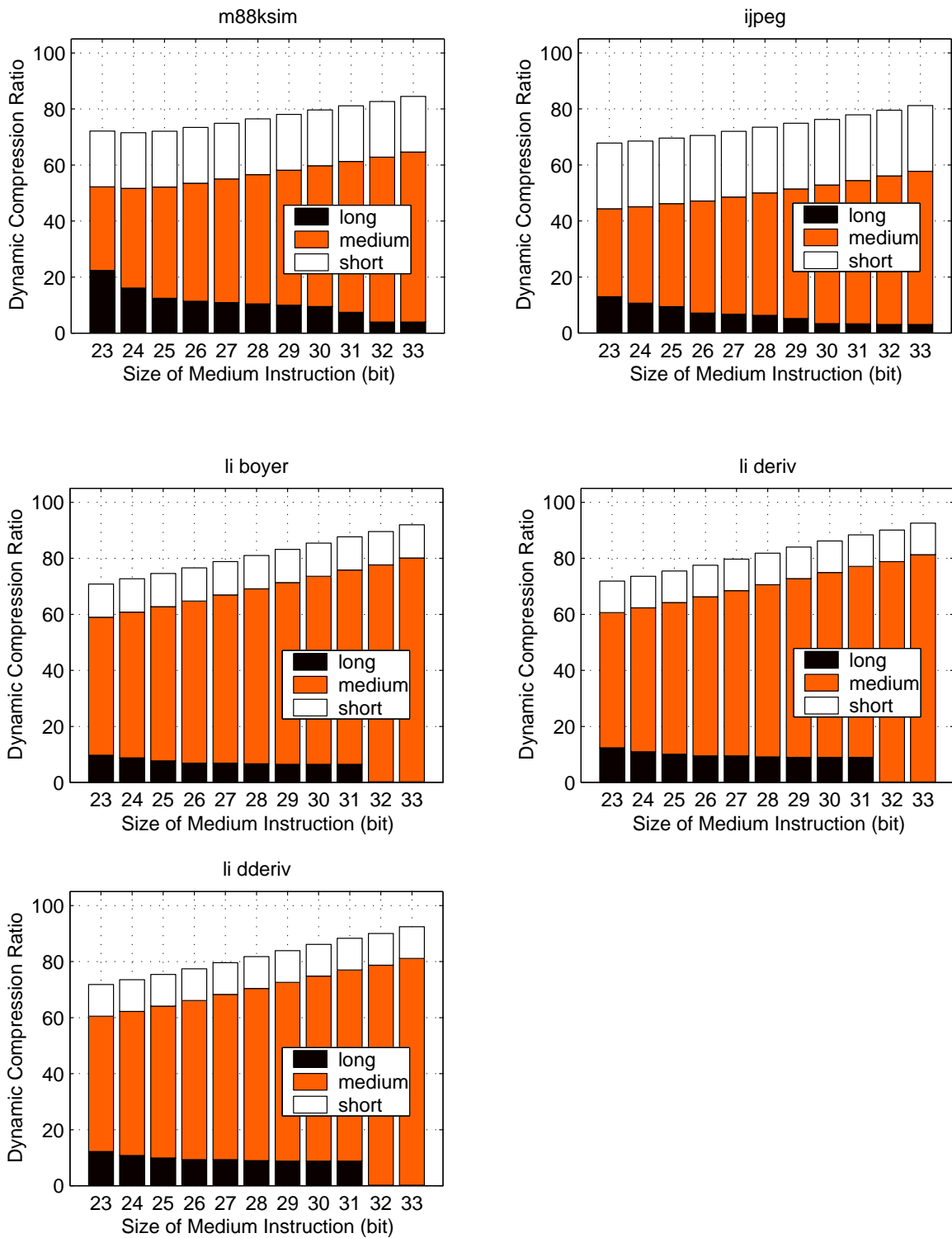


Figure 5.6: Dynamic compression ratio of various benchmark programs using the 3-size approach

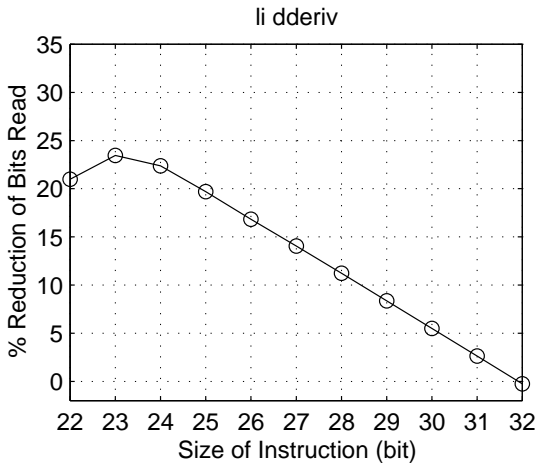
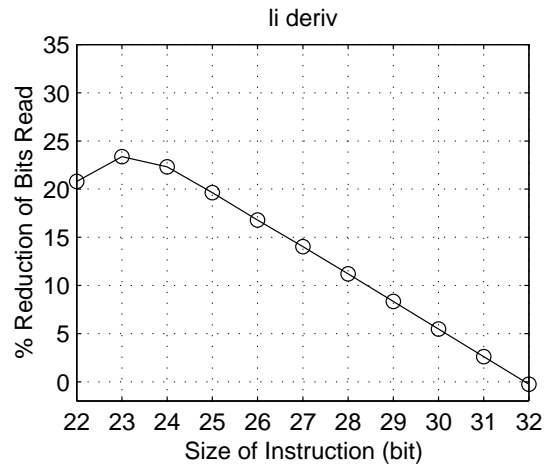
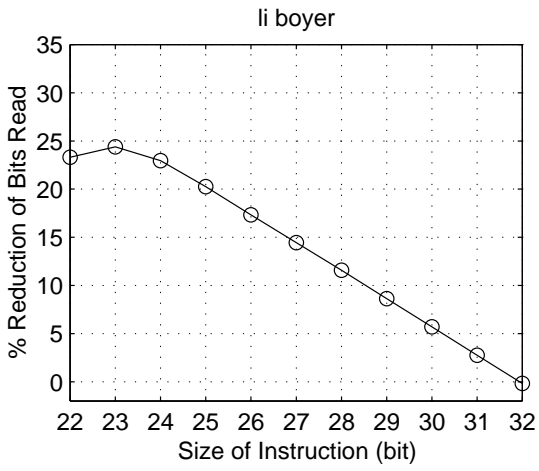
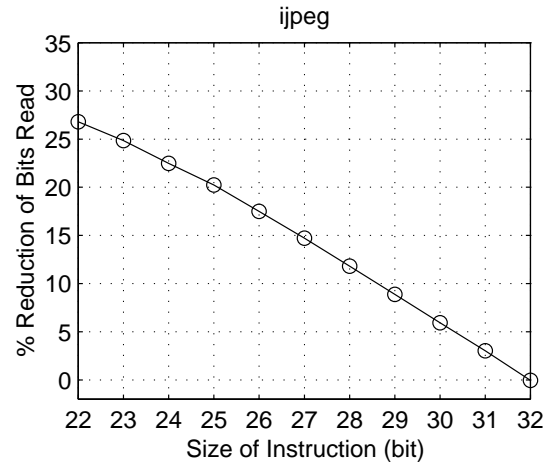
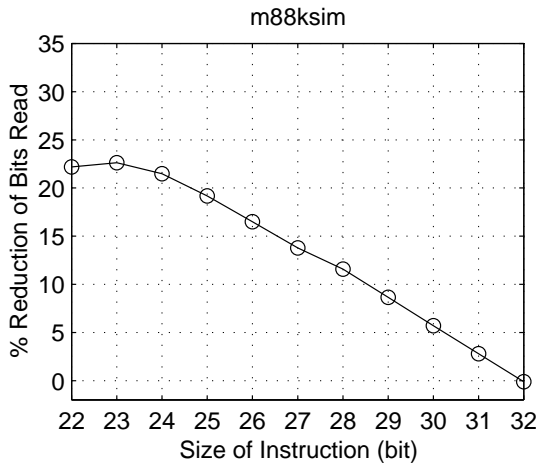


Figure 5.7: Percentage of reduction (saving) in bits read out in the 2-size approach

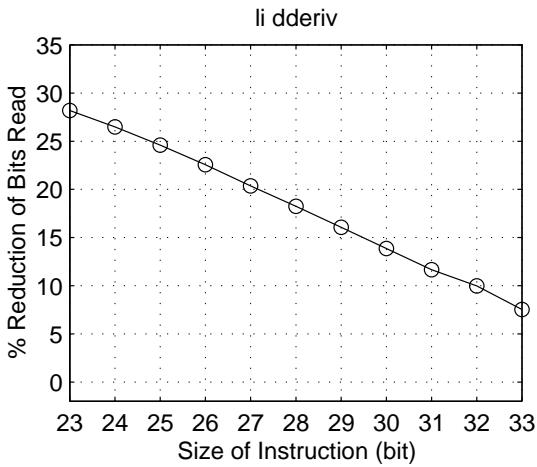
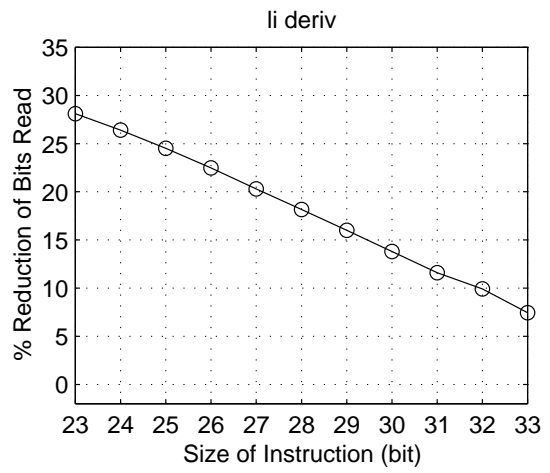
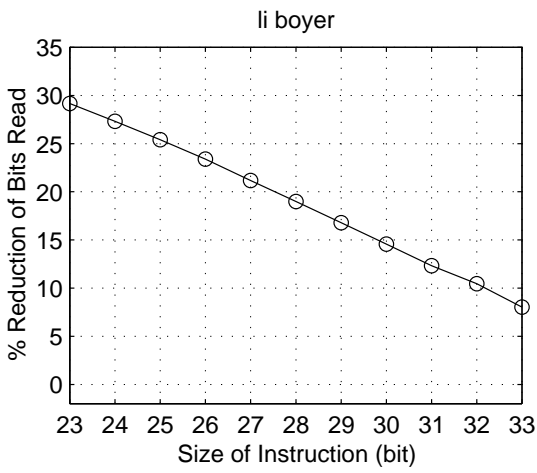
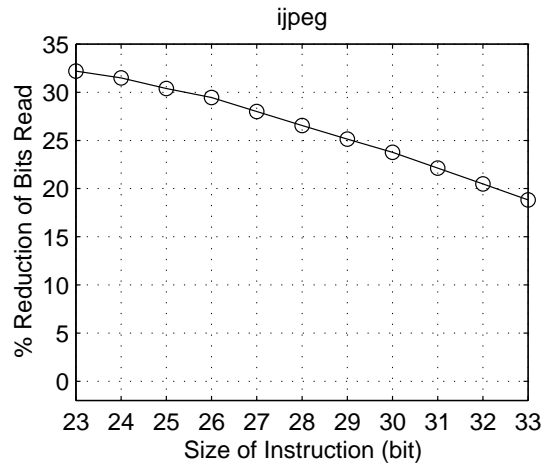
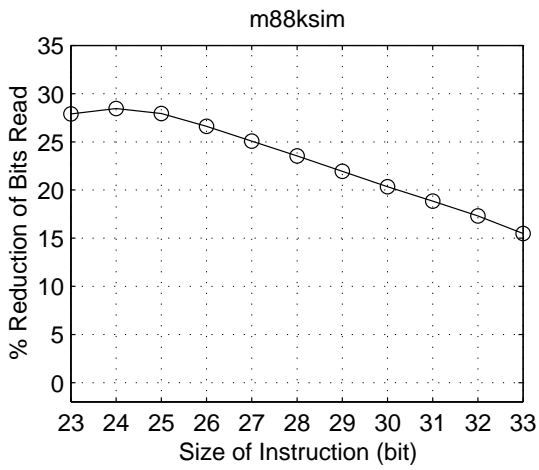


Figure 5.8: Percentage of reduction (saving) in bits read out in the 3-size approach

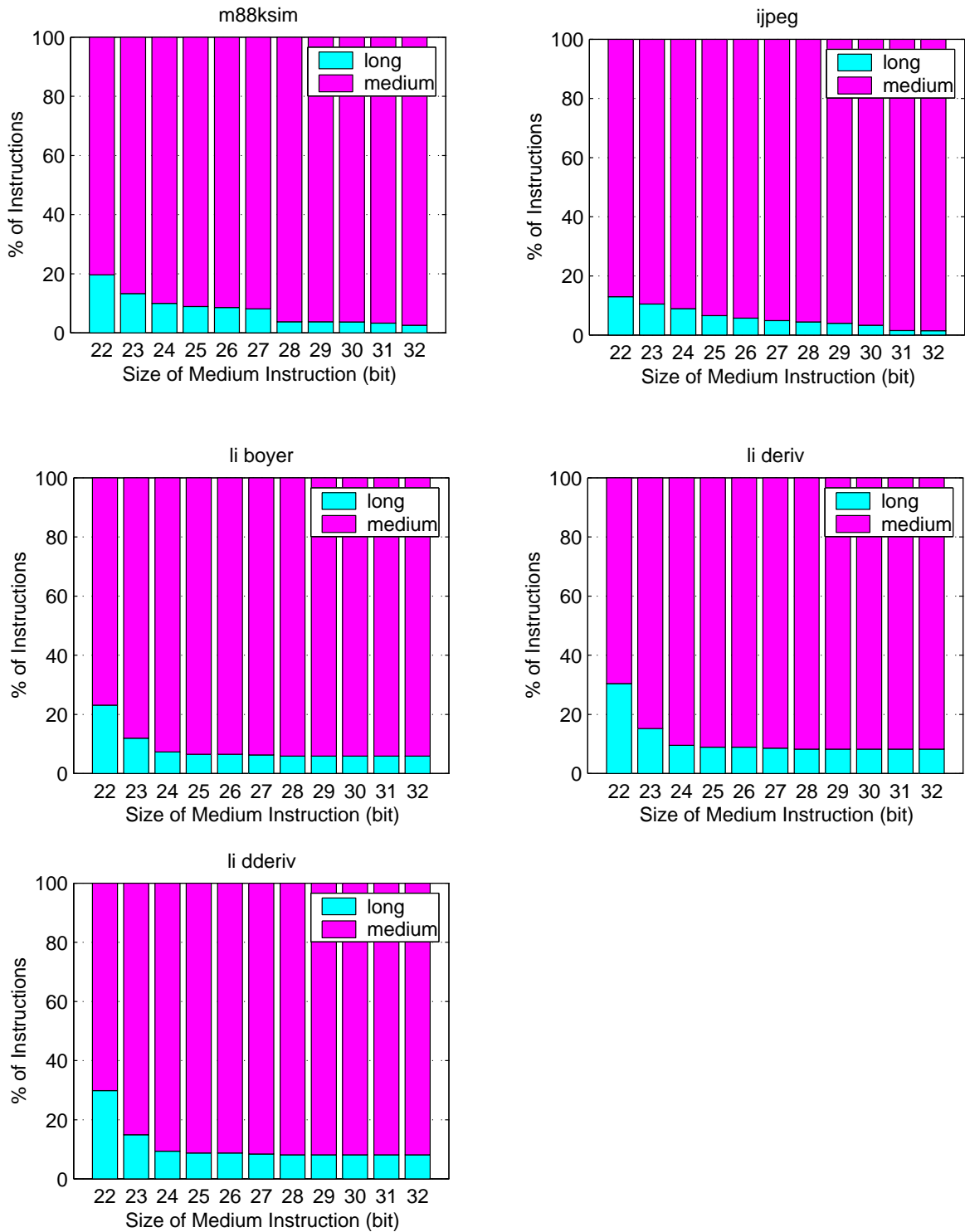


Figure 5.9: Instruction composition breakdown into medium and long instructions for the 2-size approach

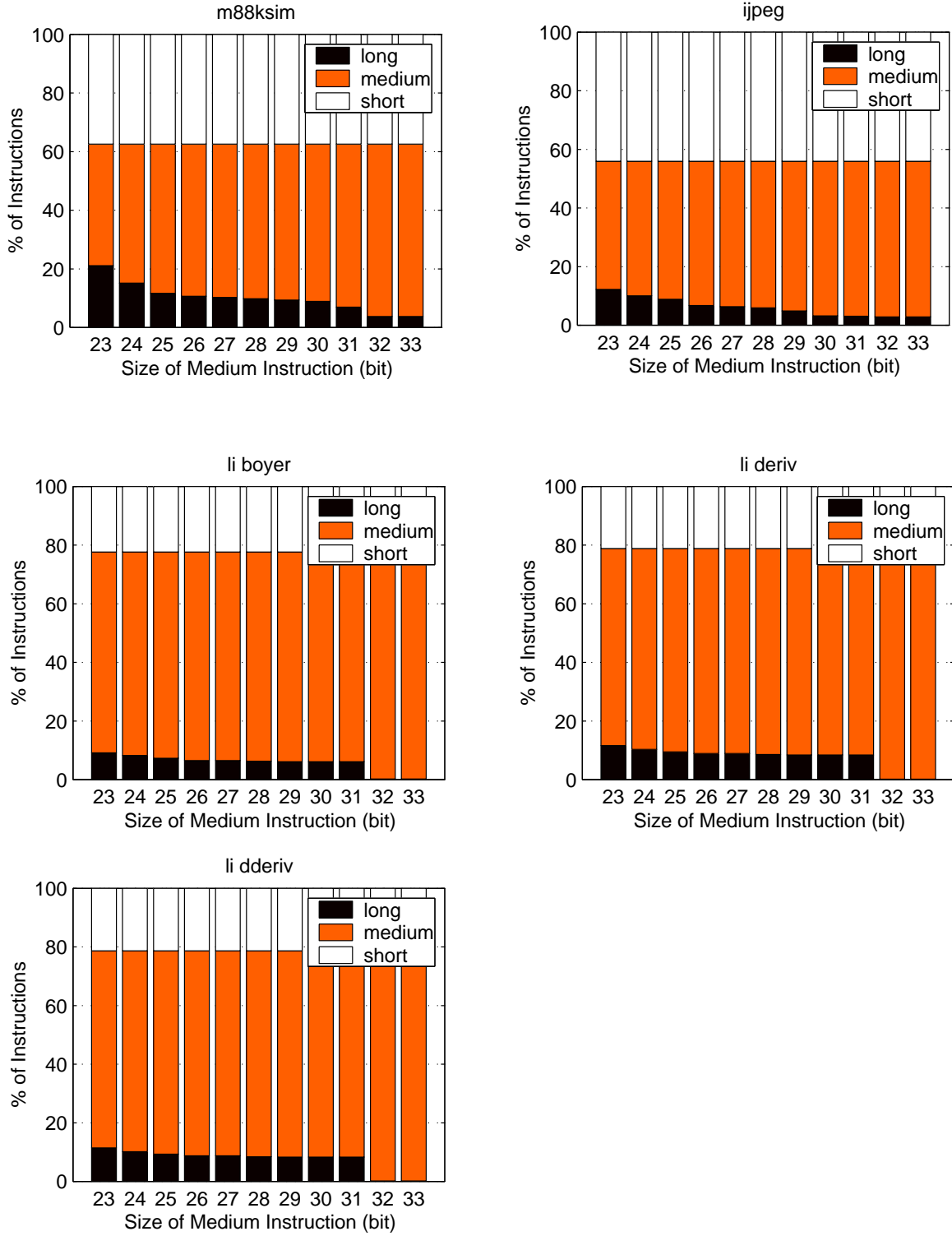


Figure 5.10: Instruction composition breakdown into short, medium and long instructions for the 3-size approach

5.2.2 Experimental Results

Our experiments reveal that the dynamic compression ratio and the percentage of reduction (saving) in bits read out are very similar across all inputs of the same benchmark program, and also comparable across the different benchmark programs in both methods. Figure 5.5 and 5.6 illustrate the dynamic compression ratio of the benchmarks in the 2-size and the 3-size approaches respectively. Figure 5.7 and 5.8 exhibit the percentage of reduction (saving) in bits read out from each benchmark in the 2-size and the 3-size versions in the order given.

As can be seen from the figures, when the dynamic compression ratio increases, the percentage of reduction in bits read out decreases because more bits are fetched out in the compressed program. Our simulations on the 2-size approach indicate that the compressed programs are composed of bits mostly from the medium-size instructions, as shown in Figure 5.5. This is how bits saving originates because we no longer use the 32-bit fixed-length format for all instructions. When the length of the medium-size instruction increases, the bits composition comprises more of the medium-size instructions and less of the long-size ones, as more instructions fit into the medium size. This is confirmed in Figure 5.9. However, the increment in the instructions captured into the medium size is less than the increase in total bits due to the longer size of the medium instruction; therefore, the dynamic compression ratio rises and the percentage of reduction (saving) in bits read out drops.

Our simulations on the 3-size approach also follow the same trend. The majority of bits are still from the medium-size instructions but a large number of bits are now from the short-size ones. The percentage of short-size instructions in the benchmarks is as follows; 37.44% in m88ksim, 44.07% in ijpeg, 22.33% in li boyer, 21.17% in li deriv and 21.31% in li dderiv. This 3-size version compresses the program size further since several instructions previously in the medium size can now be assigned to the short size. Figure 5.10 presents the breakdown of instructions in each benchmarks into short, medium and long instructions.

We have to understand here that R-type instructions get affected differently from I-type and J-type instructions when the length of medium-size instruction increases. The R-type instructions that can be compressed into the medium size are all fit into 21 bits. Therefore, the minimum medium-size length, which is 22 bits (21 bits with additional M/L bit) in the 2-size approach and 23 bits (21 bits with additional S/M bit and M/L bit) in the 3-size one, is required for the medium-size R-type instructions. As a result, when the length of the medium-size

instruction increases, no additional R-type instructions get captured into the medium size, since all have already been caught with the minimum length of the medium size instruction. Not only do the number of R-type instructions which can fit into the longer instruction not increase, but the longer medium size also introduces unused bits in the medium-size R-type instructions. Hence, more bits are wasted unnecessarily. In contrast, as the length of the medium-size instruction increases, more I-type and J-type instructions get captured into the longer medium size. This is because the longer the instruction size is, the more the instructions with longer immediate length can fit in. However, the increase in the number of the I-type and the J-type instructions that can fit into the longer medium-size instructions is not enough to offset the bits increase from using the longer size. The negative impact of longer medium size on the R-type, I-type and J-type instructions; therefore, results in the rise in the dynamic compression ratio and the drop in the percentage of reduction (saving) in bits read out as the length of the medium-size instruction increases, as seen in Figure 5.5 - 5.8. Nevertheless, this is not the case in some benchmarks when the length of the medium-size instruction increases from 22-bit to 23-bit and from 23-bit to 24-bit in the 2-size and the 3-size approaches respectively. In these exceptional cases, the increase in the number of the I-type and the J-type instructions that can fit into the longer medium-size instructions dominates the bits increase from using the longer size. Therefore, we can see the dynamic compression ratio drops and the percentage of reduction in bits read out rises, as illustrated in Figure 5.5 - 5.8.

The effect of longer medium-size instruction on dynamic compression ratio and reduction (saving) in bits read out is more pronounced in the 2-size approach than in the 3-size one. For example in *ijpeg ref*, as the immediate-length increases from 5-bit to 15-bit, the dynamic compression ratio of the 2-size approach increases from 73.21% to 100.05% compared to the dynamic compression ratio of the 3-size approach which increases from 67.81% to 81.18%.

When the immediate length of the 2-size approach is 15-bit, the compressed program size is always bigger than the uncompressed program size; therefore, dynamic compression ratio exceeds 100% and no bits are saved, but actually wasted. On the other hand, the compressed program size of the 3-size approach is always smaller than the uncompressed program size regardless of the immediate length. This is due to the fact that the number of short-size instructions in a program is constant and does not vary with the immediate size. Therefore, it helps separating the bits increase from longer medium-size instruction.

We have discovered in the 2-size approach simulation that 6-bit immediate, which results in a 23-bit medium-size instruction, gives the smallest dynamic compression ratio and maximum percentage of reduction (saving) in bits read out in the majority of the benchmarks. Similarly in the 3-size approach, the 5-bit immediate, which results in 23-bit medium-size instruction, achieves the smallest dynamic compression ratio and maximum percentage of reduction (saving) in bits read out. The 23-bit length for the medium instruction is therefore chosen primarily for both methods.

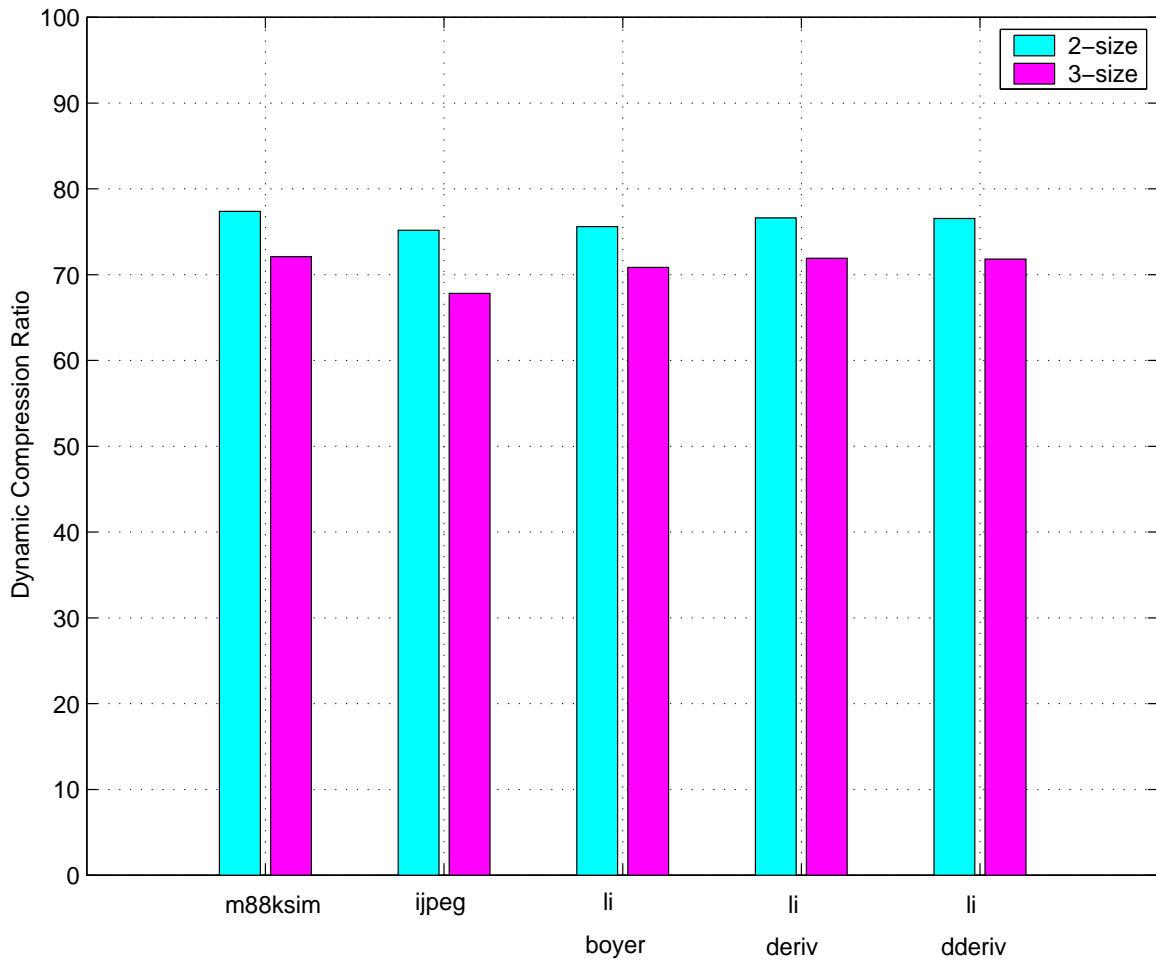


Figure 5.11: Summary of dynamic compression ratio in the benchmark programs when using 23-bit length for medium-size instruction

With the 23-bit medium-size instruction, the 2-size approach obtains an average dynamic compression ratio of 77.38% for m88ksim, 75.16% for jpeg, 75.60% for li boyer,

76.63% for li deriv and 76.54% for li dderiv. Thus, the average dynamic compression ratio across the benchmarks for the 2-size approach is 76.26%. On the other hand, with the 23-bit medium-size instruction in the 3-size approach, we achieved average dynamic compression ratios of 72.09% for m88ksim, 67.80% for jpeg, 70.83% for li boyer, 71.90% for li deriv and 71.82% for li dderiv. The average dynamic compression ratio across the benchmarks for the 3-size approach is hence 70.88%. These dynamic compression ratio statistics are summarized in Figure 5.11. One clear observation is that compressed programs in the 3-size approach saves more bits than in the 2-size approach as seen in the smaller average dynamic compression ratio.

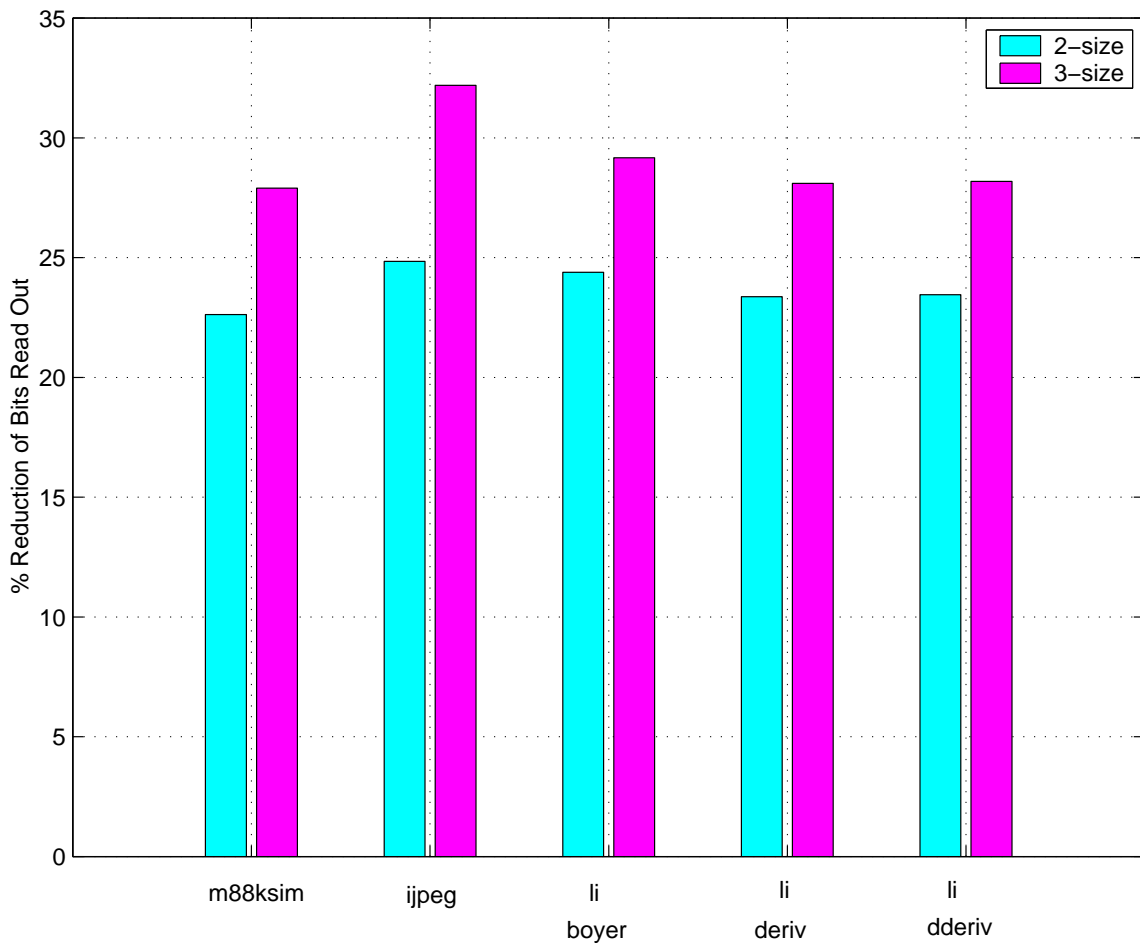


Figure 5.12: Summary of the percentage of reduction (saving) in bits read out in the benchmark programs when using 23-bit length for medium-size instruction

Compression of benchmark programs results in the reduction in bits read out of 22.62% in m88ksim, 24.84% in jpeg, 24.39% in li boyer, 23.37% in li deriv, 23.45% in li dderiv;

an average of 23.73% for the 2-size approach. For the 3-size approach, the reduction in bits read out is even greater; 27.97% in m88ksim, 32.01% in ijpeg, 29.29% in li boyer, 28.06% in li deriv, 28.16% in li dderiv; an average of 29.10%. These statistics of percentage of reduction (saving) in bits read out are summarized in Figure 5.12.

5.2.3 Conclusion

We have proposed a technique to reduce the number of bits read out from the I-cache by using gated wordlines. Our dynamic compression for programs in SPECInt95 achieves an average of reduction in bits read out of 23.73% in the 2-size approach and 29.10% in the 3-size approach.

The 3-size approach when compared to the 2-size one, achieves lower dynamic compression ratio and higher percentage of reduction in bits read out across the benchmarks. Therefore, it is clear that the 3-size approach is the better between the two in reducing the number of bitline swings and hence better in lowering the power dissipated in the SRAM array of the I-cache.

In our method of using gated wordline, the compression and decompression techniques are simple and fast; therefore, there should be little impact on the overall performance. We can also apply our technique of using gated wordline to MIPS16 and code compression technique using a Dictionary. These two techniques, when combined with our proposed method, will not only achieve higher static code size reduction but will reduce the dynamic power dissipation in the I-cache as well.

Chapter 6

Conclusion

In conclusion, we have proposed and developed a technique to lower the power consumption in the I-cache by using an in-cache instruction compression technique that uses gated wordlines to reduce the number of bitline swings.

To accurately estimate cache power, we developed a cache power consumption model based on the direct-mapped I-cache of Torrent (T0) Vector Microprocessor. From the analysis of the energy dissipated during cache access in the T0 I-cache, we have discovered that bitlines account for 94% of the total switching energy. As energy dissipated from bitlines dominates the total energy dissipation in cache, to achieve low power, the design techniques should focus on reducing the energy dissipated in the SRAM array where the bitlines are located.

Next, we investigated two previous studies of low power cache design; sub-banking and reducing the frequency of tag compare. These two techniques and our own technique of using gated wordlines, use the same strategy to lower the power consumption in the I-cache; that is by reducing the energy dissipated in the SRAM array. The simulation results show for a 1KB direct-mapped cache with block size 4-byte, 8-byte, 16-byte, 32-byte and 64-byte, energy saving from sub-banking as compared to the cache without sub-banking is 18.72%, 47.83%, 69.19%, 82.47% and 90.33% respectively. As for reducing the frequency of tag compares, the experimental results reveal that by doing tag compare only when there is conditional branch or

jump or interblock sequential flow, an average of 71.6% of tag checks are prevented across the benchmarks.

We then developed two design versions to lower the power dissipated in the I-cache, by using gated wordlines to reduce the number of bitline swings which therefore decrease the number of bits read out. We investigated two methods which compress instructions using 2 sizes or 3 sizes and evaluated our technique by applying them to the MIPS-II instruction set. Our dynamic compression for programs in SPECInt95 achieves an average reduction in bits read of 23.73% in the 2-size approach and 29.10% in the 3-size approach.

Bibliography

- [1] Advanced RISC Machines Ltd., “An introduction to Thumb”, March 1995.
- [2] B. S. Amrutur and M. Horowitz, “Techniques to Reduce Power in Fast Wide Memories”, *Symposium on Low Power Electronics*, vol. 1, October 1994, pp. 92-93.
- [3] K. Asanovic and J. Beck, T0 Engineering Data, Revision 0.1.4. Technical Report CSD-97-931, Computer Science Division, University of California at Berkeley, January 1997.
- [4] K. Asanovic and D. Johnson, “The Programmer’s Guide to SPERT”, International Computer Science Institute.
- [5] T. D. Burd and R. W. Brodesen, “Processor Design for Portable Systems”, *Journal of VLSI Signal Processing*, vol. 13, August-September 1996, pp. 203-222.
- [6] I. Chen, P. Bird and T. Mudge, “The Impact of Instruction Compression on I-cache Performance”, Technical Report CSE-TR-330-97, University of Michigan, 1997.
- [7] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson and K. Yelick, “The Energy Efficiency of IRAM Architectures”, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 327-337.
- [8] K. Ghose and M. B. Kamble, “Energy Efficient Cache Organizations for Superscalar Processors”, *Power Driven Microarchitecture Workshop at ISCA98*, June 98.
- [9] P. Hicks, M. Walnock and R. M. Owens., “Analysis of Power Consumption in Memory Hierarchies”, *Proceedings of the International Symposium on Low Power Design*, 1995, pp. 239-242.
- [10] M. Hill, “Dinero III cache simulator” online document available via <http://www.cs.wisc.edu/~markhill>, 1989.
- [11] M. B. Kamble and K. Ghose, “Energy-Efficiency of VLSI caches: A Comparative Study”, *Proceeding of IEEE 10th International Conference on VLSI Design*, January 1997, pp. 261-267.

- [12] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1992.
- [13] J. Kin, M. Gupta and W. H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure", *IEEE Micro-30*, December 1997.
- [14] K. D. Kissell, "MIPS16: High-density MIPS for the Embedded Market", Silicon Graphics MIPS Group, 1997
- [15] C. Lefurgy, P. Bird, I. Chen and T. Mudge, "Improving Code Density Using Compression Techniques", *Proceedings of Micro-30*, December 1-3 1997.
- [16] R. Panwar and D. Rennels, "Reducing the frequency of tag compares for low power I-cache design", *Proceedings of the International Symposium on Low Power Design*, 1995, pp. 57-62
- [17] D. Patterson and J. Hennessy, *Computer Architecture - A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.
- [18] T. Pering, T. Burd and R. Brodersen, "Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System", *Power Driven Microarchitecture Workshop at ISCA98*, June 1998.
- [19] C. Su and A. M. Despain, "Cache Design Tradeoffs for Power and Performance Optimization: A Case Study", *Proceedings of the International Symposium on Low Power Design*, 1995, pp. 63-68.
- [20] T. Wada, S. Rajan and S. Przybylski, "An Analytical Access Time Model for On-chip Cache Memories", *IEEE Journal of Solid-State Circuits*, vol. 27, No. 8, August 1992.
- [21] S. E. Wilton and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-chip Caches", *DEC WRL Research 93/5*, July 1994.