Low Power Architecture Design and Compilation Techniques forHigh-Performance Processors

Ching-Long Su, Chi-Ying Tsui, Alvin M. Despain

Advanced Computer Architecture Laboratory

ACAL-TR-94-01

February 15, 1994

Keywords: Gray Code, Low Power Design, Memory Addressing, Instruction Scheduling

Abstract

Reducing switching activity would significantly reduce power consumption of a processor chip. In this paper, we present two novel techniques, Gray code addressing and Cold scheduling, for reducing switching activity on high performance processors.

We use Gray code which has only one-bit different in conseuctive number for addressing. Due to locality of program execution, Gray code addressing can significantly reduce the number of bit switches. Experimental results show that for typical programs running on a RISC microprocessor, using Gray code addressing reduce the switching activity at the address lines by 30~50% compared to using normal binary code addressing.

Cold scheduling is a software method which schedules instructions in a way that switching activity is minimized. We carried out experiments with cold scheduling on the VLSI-BAM. Preliminary results show that switching activity in the control path is reduced by 20-30%.

To appear in *Proceedings of the IEEE COMPCON*, Febuary 1994.

Low Power Architecture Design and Compilation Techniques for High-Performance Processors

Ching-Long Su, Chi-Ying Tsui, and Alvin M. Despain

csu@usc.edu, tsui@usc.edu, and despain@usc.edu
Advanced Computer Architecture Laboratory
University of Southern California

Abstract

Reducing switching activity would significantly reduce power consumption of a processor chip. In this paper, we present two novel techniques, Gray code addressing and Cold scheduling, for reducing switching activity on high performance processors.

We use Gray code which has only one-bit different in conseuctive number for addressing. Due to locality of program execution, Gray code addressing can significantly reduce the number of bit switches. Experimental results show that for typical programs running on a RISC microprocessor, using Gray code addressing reduce the switching activity at the address lines by 30~50% compared to using normal binary code addressing.

Cold scheduling is a software method which schedules instructions in a way that switching activity is minimized. We carried out experiments with cold scheduling on the VLSI-BAM. Preliminary results show that switching activity in the control path is reduced by 20-30%.

1. Introduction

With recent advances in microelectronic technology, smaller devices are now possible allowing more functionality on an integrated circuit (IC). Portable applications have shifted from conventional low performance products such as wristwatches and calculators to high throughput and computation intensive products such as notebook computers and cellular phones. The new portable computing applications require high speed, yet low power consumption as for such products longer battery life translates to extended use and better marketability. With the convergence of telecommunications, computers, consumer electronics, and biomedical technologies, the number of low power applications is expected to grow. Another driving force behind design for low power is that excessive power consumption is becoming the limiting factor in integrating more transistors on a single chip or on a multi-chip module due to cooling, packaging and reliability problems. In this work, we will concentrate on how power consumption can be minimized for high performance microprocessors.

1.1 Related work

Power consumption in CMOS has three components: dynamic power consumption, short circuit current power consumption, and static power consumption. With proper circuit design techniques, the latter two components can be reduced and are negligible compared to the dynamic power consumption. Therefore power consumption in CMOS can be described by

$$\mathbf{P_g = f_d\, C_L\, V^2_{dd}}$$

where f_d is the switching activity, C_L is average capacitance loading of the circuit, and V_{dd} is the supply voltage. To minimize power consumption, we can reduce f_d, C_L or V_{dd}.

Many researchers have been studying low power/low voltage design techniques. For example, research is being conducted in low power DRAM and SRAM design. Also aggressive supply voltage scaling and process optimization are used for power consumption reduction for active logic circuits [Chandra 92],[Liu 93]. However, lowering the supply voltage may create other design problems such as reduced noise margin, increased cross talk, etc. Other researchers are exploring instruction set architectures and novel memory management schemes for low power, processor design using self-clocking, static and dynamic power management strategies, etc.

Recently research has been done in minimizing the switching activity of the circuit in order to minimize power. This method is orthogonal to supply voltage reduction and process optimization and thus can be used to further reduce power consumption once the supply voltage and process of the processor are chosen. Currently most of the work has been carried out in the layout and logic levels.

In the layout domain, Vaishnav et al. [Vaishnav-93] propose a low power performance driven placement procedure which minimizes the length, hence the capacitance loading, of the high switching nets and at the same time satisfies the delay constraints.

In the logic level, algorithms to synthesize circuits with minimum switching activities has been developed. Shen et al [Shen 92] present algorithms for reducing power consumption during the technology independent phase of logic synthesis. Prasad et al. [Prasad 93] tackle the low power kernelization problem in multi-level logic minimization. During the factorization process, common sub-expressions which result in a maximum reduction in switching activities are extracted. Roy et al.[Roy 92] propose a low power state assignment method which uses simulated annealing to find the state encoding of a finite state machine which the total probability weighted Hamming distance of the states are minimized. Tsui et. al.[Tsui 93] minimize the weighted switching activity and hence the power consumption during the technology decomposition and mapping phase of logic synthesis. All the above methods assume the switching activities at the circuit inputs are given and minimize the internal switching activities based on this assumption.

1.2 Our approach

In this work, we tackle the problem of minimizing circuit switching activities at a higher level, the architectural level, and we study this in the domain of high performance microprocessors. Instead of minimizing the internal switching activities of each module of the microprocessor, we minimize the switching at the inputs of the modules. Specifically, we minimize the switching activities of the address bus and the instruction bus of the microprocessor using some novel hardware and software techniques. The reasons for focusing on the instruction and address buses are as follows. Switching activity depends on the sequence of the signal values applied at the inputs of the circuit. For the datapath of a microprocessor, the signal values at the inputs depend on the data and hence can only be determined at run-time. For the instruction and address lines, the values are related to the static code which can be determined at complie time. Also for a typical pipelined RISC processor, the instruction line has to drive many modules such as the instruction cache, instruction register, the control path decoder. The address lines also drive many modules such as Memory Address Register, PC chains and the I/O pads. Moreover the instruction and address buses are usually long and have higher routing capacitance. Therefore they are nets with large capacitance loading and minimizing the switching activity of these nets has a significant impact on the power consumption of the microprocessor.

In this paper, we present two techniques which reduce switching activity during program execution. The first technique, Gray code addressing, is a hardware method which uses Gray code for instruction addresses. The second technique, Cold Scheduling, is a software method which schedules instructions during compilation to reduce switching activities.

The advantage of Gray code over straight binary code is that Gray code changes by only one bit as it sequences from one number to the next. In other words, if the memory access pattern is a sequence of consecutive addresses, then each memory access changes only one bit at its address bits. Due to instruction locality during program execution, most of the time memory accesses are sequential in nature. Therefore a significant number of bit switches can be eliminated through using Gray code addressing.

Unlike traditional instruction scheduling which mainly focuses on scheduling instructions for less pipeline hazards, cold scheduling focuses on scheduling instructions for less switching activities while keeping performance as high as possible. Cold scheduling can be easily implemented by modifying a traditional list scheduler with a cost function to minimize the switching activities invoked by executing consecutive pairs of instructions.

1.3 Organization of the Paper

The rest of the paper is organized as follows. Section 2 discusses the significance of the switching activities at the inputs of a circuit on the overall switching activities of the circuit. Section 3 evaluates the use of Gray code as an instruction addressing scheme. We also compare the switching activities using Gray code addressing to traditional binary code addressing. Section 4 presents the cold scheduling techniques and some experimental results. Finally, conclusionary remarks are provided in Section 5.

2. Switching Activity in Pipelined Processors

Figure 1 shows a typical pipelined circuit of which each stage consists of a combinational circuit between two latches. At the beginning of a clock cycle, the input signals of the combinational circuit are first latched in the input latch A. They are then evaluated in the combinational circuit and propagated to the output which are latched in the output latch B at the next cycle. These signals become the input signals for the combinational circuit at the next pipeline stage. The switching activities of the combinational circuits depends on the logic and structure of the circuit and the switching activities at the output of the input latch. Although there is no general theory on the relationship between the switching activities of the inputs and that of the internal nodes of a combination circuit, we believe that if the switching activities are high at the inputs, then the internal switching activities at the combinational circuit also tend to be high and vice versa.

To support this claim, we carried out experiments

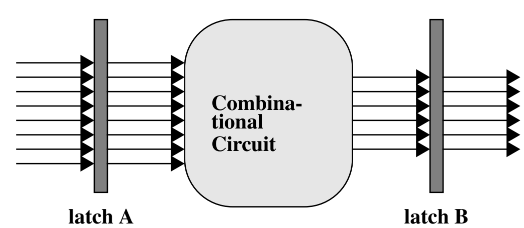


Figure 1 Combinational circuit and latches

on a set of combinational circuit benchmarks obtained from the ISCAS-89 and MCNC-91 benchmark set, and studied the effect on the circuit switching activities if the switching activities at the input lines are reduced. We used the estimation method in [Ghosh 92] to estimate the switching activities. Switching activities are measured as the expected numbers of switching per cycle. First the switching activity of each input is set to 0.5 (Model I). Then it is reduced to 0.42 (Model II) and 0.32 (Model III) respectively. Table 1 summarizes the circuit switching activities under different input switching activities.

	Model I	Model II		Model III	
circuit	switching activity	switching activity	% of reduction	switching activity	% of reduction
9symml	38.11	33.88	11.1	28.31	25.72
alu4	94.91	93.05	1.96	83.80	11.71
apex6	214.28	186.27	13.07	158.50	26.03
cordic	25.23	24.31	3.65	23.03	8.72
count	37.66	32.16	14.60	27.80	26.18
example	92.17	85.80	6.91	76.38	17.13
f51	27.24	23.74	12.85	19.76	27.46
pair	384.02	352.57	8.19	311.33	18.93
s208	26.57	27.07	-1.88	26.30	1.02
s298	31.81	27.38	13.93	22.99	27.73
s344	44.84	38.33	14.53	32.06	28.50
s400	47.80	38.60	19.25	31.13	34.87
s444	48.55	39.50	18.64	31.91	34.27
s526	51.75	45.83	11.44	38.52	25.57
s820	51.87	45.29	12.69	38.49	25.80
s838	99.65	94.19	5.48	85.99	13.71
s953	49.23	39.68	19.40	32.40	34.19
x1	78.71	71.62	9.01	62.36	20.77
x3	213.42	191.90	10.08	165.93	22.25
x4	140.47	122.35	12.90	103.70	26.18
avg.% reduction			10.89		22.84

Table 1Circuit switching activities vs input switching activities

From the results, we see that in general reducing the switching activities at the inputs will reduce the total switching activities of the combinational circuit. The average reductions are about 11% and 23% when the input switching activities are reduced by 16% and 36% respectively.

Switching activities at the staging latches of a pipelined microprocessor are affected by various factors. For latches in the datapath, switching activities are mainly determined by the run-time data sequences. For latches in the control path, switching activities are strongly dependent on the instruction execution sequences. Since run-time data is not well known at compile-time, the impact of cold scheduling on switching activities at the latches in the data path is not clear. However, instruction execution sequences are controlled by the instruction scheduler at compile-time. The impact of cold scheduling on switching activity at the latches in the control path is more direct. In this paper, we will focus on the impact of cold scheduling for reducing switching activities in the control path.

Different instruction sequences can have a significantly different effect on the switching activities and the impact depends on the type of architecture. In a CISC processor, the impact may not be so obvious since one instruction may need several processor cycles to execute. In contrast, for a general pipelined RISC-like processor, in which most instructions can be executed in one processor cycle, the impact can be significant since an instruction scheduler has more instructions to schedule.

To better understand the impact of instruction sequence on the switching activities in general purpose processors, we select a RISC-like processor, the VLSI-BAM [Holmer 90], as an experimental architecture. This microprocessor is pipelined with data stationally control. There are five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execution (IE), Memory access (M), and Write Back (WB). The instruction set of the VLSI-BAM is similar to the MIPS-2000 [MIPS 86] with some extensions for symbolic computation. Figure 2 shows the pipeline stages and the control path of the VLSI-BAM processor. For each pipeline stage, there is an instruction register, a PLA, and a latch for control signals. Instructions are passed through instruction registers and decoded by the PLAs in the pipeline stages. Control signals which are generated from PLAs, are latched before they are sent to the datapath.

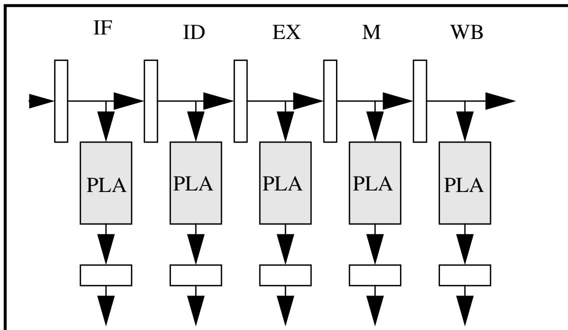


Figure 2 Pipeline structure and control path of VLSI-BAM

A cycle-by-cycle instruction-level simulator is built for collecting the switching activities at the latches in the control path during execution of benchmark programs. Benchmark programs used in this paper are shown in Table 2. The benchmarks are ranging from less than 1,000 cycles to larger than 10,000,000 cycles. These benchmark programs are selected from the Aquarius benchmark suite [Haygood 89]. Applications of these benchmark programs include list manipulation, data base query, theorem prover, and computer language parser. Benchmark programs are first compiled through the Aquarius Prolog compiler [Van Roy 92] into an intermediate code (BAM code), which is target machine independent. The BAM code is then further compiled into machine code of the target machine, the VLSI-BAM.

3. Gray Code Addressing

In traditional von Neumann machines, data is fetched from memory before executed. For binary code addressing scheme, data and instructions that are accessed sequentially are located in the memory with consecutive binary address. For sequential memory access, next address is obtained by doing a binary increment on the current address.

Table 2 Benchmark programs

Benchmark	Cycles	Description
fastqueens	1,138,655	Eight queens problem
qsort	4,560	Quicksort of a 50-element list
reducer	1,064,197	A graph reducer for T-combinator
circuit	4,504,940	VLSI module generator
semigroup	4,487,201	Query a data base
nand	350,761	A circuit generator
boyer	27,494,723	Boyer-Moore theorem prover
browse	18,883,712	Build and query a database
chat	3,303,153	English for database querying

3.1 Binary Code Representation

Binary code addressing system uses base 2. A binary representation 10010110 is interpreted as

$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 170$

Table 3 shows 4-bit binary representations and their corresponding decimal equivalent.

Table 3 Binary code representation and decimal equivalent

Binary code	Decimal Equivalent	Binary code	Decimal Equivalent
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

3.2 Gray Code Representation

A Gray code sequence is a set of numbers, represented as a combination of digits 1s and 0s, in which contiguous numbers have only one bit different. A formal definition of a Gray code sequence is described as follows [Hayes 88],

1. $G_1 = 0, 1$.

2. Let $G_k = g_0, g_1, \ldots, g_2k_{-2}, g_2k_{-1}$. G_{k+1} is formed by first preceding all members of the sequence G_k by 0, then repeating G_k with the order reversed and all members preceded by 1. In other words,

$G_{k+1} = 0g_0, 0g_1, \ldots, 0g_2k_{-2}, 0g_2k_{-1}, 1g_2k_{-1}, 1g_2k_{-2}, \ldots, 1g_1, 1g_0$

For example, G2 = 00, 01, 11, 10 and G3 = 000, 001, 011, 010, 110, 111, 101, 100. Clearly the foregoing construction ensures that consecutive members of a Gray code sequence differ in exactly 1 bit. Table 4 shows 4-bit Gray Code representations and their corresponding decimal equivalent.

Table 4 Gray code representation and decimal equivalent

Gray code	Decimal Equivalent	Gray code	Decimal Equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

3.3 Conversion between Binary and Gray Code

The conversion between binary code and Gray code is defined as follows. Let B and G be the binary and the Gray code representations of the same decimal number respectively represented by:

$$B = \langle b_{n-1}, b_{n-2}, \ldots, b_1, b_0 \rangle$$
$$G = \langle g_{n-1}, g_{n-2}, \ldots, g_1, g_0 \rangle.$$

The conversion of binary code to Gray code is as follows. Let g_i and b_i be the i^{th} bit of G and B respectively. g_i is then equal to the exclusive or of b_i and b_{i+1}. The most significant bit of G and B are the same. The following formula summarize the conversion.

Binary code --> Gray code

$$g_n = b_n$$
$$g_i = b_{i+1} \oplus b_i \quad (i = n-1, 0)$$

For example, let B be a binary number <1,1,0,1> the decimal equivalent of which is 13. The values of b_3, b_2, b_1, and b_0 are 1, 1, 0, 1 respectively. The Gray code representation is then equal to $\langle b_3, b_3 \oplus b_2, b_2 \oplus b_1, b_1 \oplus b_0 \rangle$ which is equivalent to <1,0,1,1>.

Similarly the conversion of Gray code to binary code also uses the exclusive or operation. However, it is more complex. b_i is equal to the exclusive or of g_i and all of the bits of G that preceding g_i, i.e. $g_{i+1}, g_{i+2}, \ldots, g_{n-1}$. The most significant bit of the binary representation and Gray code representation are the same. The following formula summarize the conversion.

Gray code --> Binary code

$$b_n = g_n$$
$$b_i = b_{i+1} \oplus g_i \quad (i = n-1, 0)$$

For example, let G be a Gray code number <1,1,0,1> the decimal equivalent of which is 9. The values of g_3, g_2, g_1, and g_0 are 1, 1, 0, 1 respectively. The binary code representation is then $\langle g_3, g3 \oplus g2, g3 \oplus g2 \oplus g1, g3 \oplus g2 \oplus g1 \oplus g0 \rangle$ which is <1,0,0,1>.

3.4 Number of Bit Switches in Binary Code vs. Gray Code

The number of bit switches of a sequence of numbers can be significantly different depending on the code representations. Figure 3 shows an example. For the sequence of numbers from 0 to 16, shown in Figure 3(a), there are 31 bit switches when the number are encoded in binary representation while only 16 bits switch when they are encoded in Gray code representation. However, for the following sequence $\langle 1,3,7,15,14,12,13,9,8,10,11,2,6,4,5,0,16 \rangle$, which is shown in Figure 3(b), there are only 17 bit switches for binary representation compared to 29 bit switches for Gray code representation.

(a)

	binary	Gray
0	00000	00000
1	00001	00001
2	00010	00011
3	00011	00010
4	00100	00110
5	00101	00111
6	00110	00101
7	00111	00100
8	01000	01100
9	01001	01101
10	01010	01111
11	01011	01110
12	01100	01010
13	01101	01011
14	01110	01001
15	01111	01000
16	10000	11000
bit changed	31	16

(b)

	binary	Gray
1	00001	00001
3	00011	00010
7	00111	00100
15	01111	01000
14	01110	01001
12	01100	01010
13	01101	01011
9	01001	01101
8	01000	01100
10	01010	01111
11	01011	01110
2	00010	00011
6	00110	00101
4	00100	00110
5	00101	00111
0	00000	00000
16	10000	11000
bit changed	17	29

Figure 3 The bit switches of binary codes vs. Gray codes

For random access patterns, Gray code and binary code have similar number of bit switches. Note that the number sequence in Figure 3(b) is careful selected in favor of the binary representation. In general, this special sequence is rather unlikely to happen. For consecutive access patterns, which occur often in a general processor for executing consecutive instructions in basic blocks, Gray code addressing has fewer bit switches.

3.5 How To Use Gray Code Addressing

In a general uni-processor, an instruction is fetched from an address pointed to by the program counter. After this instruction is fetched, the program counter is increased by one for the next fetch. The program counter is also modified by branch instructions based on branch conditions. In the binary code addressing system, a binary counter is needed for incrementing the program counter. For branch instruction, the calculated target addresses are written directly into the program counter if the branch is taken. In Gray code addressing system, a Gray code counter is needed for incrementing the program counter. For branch instructions, the calculated target address is directly written into the program counter if the branch is taken. The only difference between binary code and Gray code addressing systems is the instruction field for target addresses. In a Gray code addressing system, the instruction field for the target address in a branch instruction is modified such that the calculated target address is the correct target address in Gray code addressing system.

Figure 4 shows an example of branch instructions in binary code and Gray code addressing systems. We represent a binary code as $\langle b_n, b_{n-1},...,b_0\rangle_{\mathbf{2}}$ and a Gray code as $\langle g_n, g_{n-1},..., g_0\rangle_{\mathbf{g}}$, where the bit length is n+1. The correct execution sequence is (*A*,*B*,*br*,*C*,and *D*) where *A*, *B*, *C*, and *D* are instructions and *br* is a taken branch. The target address of the branch is the addition of current program counter and the offset which is specified in the branch instruction. In binary code addressing system, the offset of the branch instruction is $\langle 00111\rangle_2$ and the value of the current program counter is $\langle 01110\rangle_2$. The target address of this branch instruction is the binary addition of $\langle 01110\rangle_2$ and $\langle 00111\rangle_2$, which is $\langle 10101\rangle_2$. In the Gray code addressing system, we can implement this branch operation by modifying the offset value specified in the branch instruction to $\langle 10110\rangle_g$. Since the current program counter is $\langle 01001\rangle_g$, the target address of this branch instruction is then $\langle 11111\rangle_g$, which is the binary addition of $\langle 01001\rangle_g$ and $\langle 10110\rangle_g$.

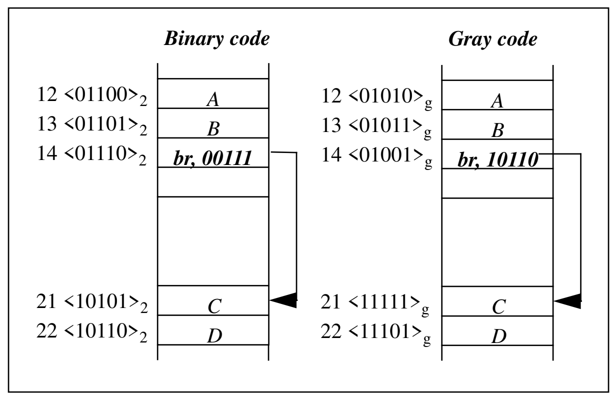


Figure 4 Branch instructions in binary code and Gray code addressi

3.6 Results

To validate the advantage of Gray code addressing, we implement a Gray code addressing scheme on the VLSI-BAM [Holmer 90]. Table 5 summarizes the switching activity at the address bits of the processor. For instruction accesses, compared to the traditional binary code addressing scheme, Gray code addressing significantly reduces the address bits switching activities. For data accesses, switching activity resulting from both schemes are quite close. This is because instruction accesses are more sequential than data addresses for normal program execution. Moreover, the more instruction locality a program has, the more reduction will be.

We measure the performance of the address coding scheme by the number of bit switches per executed instruction, denoted as BPI. Figure 5 shows the BPI of instruction addresses in binary code and Gray code for different benchmark programs. The benchmark program with the most significant reduction of bit switches is *fastqueens*. The BPI of instruction addresses in binary

Table 5 Switching Activities at the address bits

Bench-mark	Instruction Address in Binary Code	Instruction Address in Gray Code	Data Address in Binary Code	Data Address in Gray Code
fastqueens	2,804,797	1,177,861	973,151	1,038,509
qsort	12,057	6,047	6,011	5,701
reducer	2,731,471	1,817,517	1,566,384	1,484,893
circuit	10,477,307	6,638,057	6,005,394	5,308,314
semigroup	12,028,773	8,933,613	6,174,504	6,010,871
nand	848,899	551,512	438,553	406,609
boyer	76,019,503	57,440,477	48,250,617	47,333,577
browse	47,335,397	30,897,487	24,885,247	26,393,123
chat	8,019,831	5,099,811	4,370,555	3,952,065

code and Gray code are 2.46 and 1.03 respectively. The reduction in bit switches is more than half (58.13%). For the worst performance of Gray code addressing, which is *boyer*, the reduction of bit switches is still significant (24.28%). The average BPI for Gray code addressing is 1.60, while the average of BPI in binary code addressing is 2.53. The average reduction in bit switches is then equal to 36.89%.

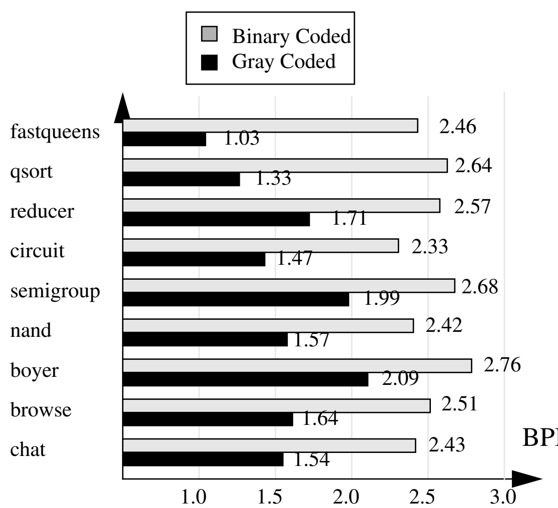


Figure 5 Bit switches of instruction addresses

Figure 6 shows the BPI of data addresses in binary code and Gray code among benchmark programs. The BPI of data addresses in binary code and Gray code are very close. Among the set of benchmark programs, we find that two out of nine benchmark programs (*fastqueen and browse*) have lower BPI in binary code than in Gray code. The other seven benchmark programs have slightly larger BPI in binary code than in Gray code. The average BPI in Gray code addressing is 1.28 while that in binary code addressing is 1.39. The average reduction of bit switches is then 7.91%.

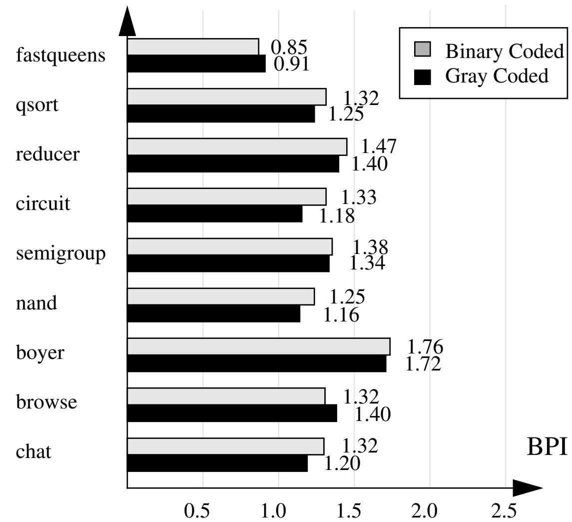


Figure 6 Bit switches of data addresses

4. Cold Scheduling

Traditional instruction scheduling algorithms mainly focus on reordering instructions to reduce pipeline stalls, avoid pipeline hazards, or improve resource usage. More recent instruction scheduling algorithms such as trace scheduling [Fisher 81], percolation scheduling [Nicolau 84], and global scheduling [Bernstein 91] schedule instructions across basic blocks in order to increase instruction-level parallelism. The main goal of these scheduling algorithms is to improve performance. To reduce power consumption, these instruction scheduling algorithms need to be modified to adjust to the new objective.

In this section, we present the details of our cold scheduling algorithm. Basically cold scheduling uses traditional performance-driven scheduling techniques with special heuristics for reducing switching activities. Before we go into the details of cold scheduling, we first review the traditional list scheduling algorithm.

4.1 Scheduling for Performance

Traditional instruction scheduling approaches consist of three steps: 1) partition a program into regions or basic blocks. 2) build a control dependency graph (CDG) and/or data dependency graph (DDG) for each code region or basic block. 3) schedule instructions in CDG and/or DDG within resource constraints.

The main goal of the traditional instruction scheduler is to schedule the instruction sequence such that it can be executed in a target machine as fast as possible with minimal pipeline stalls. Therefore the quality of an instruction scheduler is measured by the amount of pipeline stalls introduced by the output instruction sequence when it is executed on the machine.

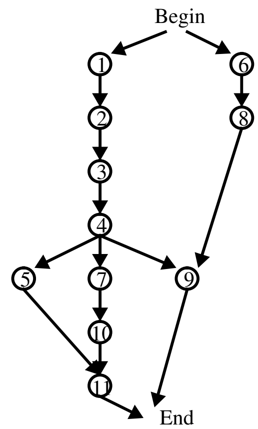
Let $B = I_1, I_2,...$ be an output instruction sequence of a basic block. The number of pipeline stall cycles between execution of instruction I_j and I_{j+1} is denoted as $D(I_j,I_{j+1})$. For example, if there is no pipeline stall cycles between execution of instruction I_1 and I_2, then $D(I_1,I_2) = 0$. Otherwise, if there are m pipeline stall cycles between execution of instruction I_1 and I_2, then $D(I_1,I_2) = m$. The total pipeline stall cycles in the execution of a basic block is denoted as $PS = \Sigma D(I_j,I_{j+1})$, j = 0... n-1. The main objective of an instruction scheduler is therefore to minimize PS. In the case of scheduling a region, the objective is thus to minimize the pipeline stalls of this region, instead of in an individual basic block. For example, if a region consists of basic blocks $B_1,B_2,...B_k$, and the number of pipeline stalls in these basic blocks are PS_1, $PS_2,...PS_k$, then the instruction scheduler tries to minimize $1/k(w_1*PS_1+ w_2*PS_2+... +w_k*PS_k)$, where w_j is a weight of estimated dynamic execution frequency of a basic block B_j.

Figure 7 shows a data dependency graph and various output instruction sequences from a typical instruction scheduler. The target machine is assumed to have a one cycle delay slot for load/store instructions after store instructions due to bus conflicts. These instruction sequences running on the VLSI-BAM introduce different pipeline stalls. Instruction sequence II suffers three cycles of pipeline stalls. Instruction sequences I and III have the best scheduling with only one pipeline stall cycle.

4.2 Scheduling for Low Power

Switching activities depend on the sequence of input signals. If the input sequence can be reordered in such a way that switching activities are minimized, power consumption can then be minimized. Therefore, the goal of an instruction scheduler for cold scheduling is to reorder the code such that switching activities are minimized without introducing significant performance degradation. The quality of cold scheduling is measured by the reduction in switching activity in the processor when the output instruction sequence is executed on a target machine M. We denote $S(I_j,I_{j+1})$ as the number of bit switches that occur in the processor when instruction I_{j+1} is executed just after I_j on M. The total switching activity is defined as $BS = \Sigma S(I_j,I_{j+1})$, j = 0... n-1. The main goal of cold scheduling is to minimize BS. In the case of scheduling a region which consists of basic block $B_1,B_2,...B_k$, having switching activities BS_1, $BS_2,...BS_k$ respectively, the cost function of the cold scheduler is therefore $1/k(w_1*BS_1+w_2*BS_2+...+w_k*BS_k)$, where w_j is a weight of estimated dynamic execution frequency of a basic block B_j.

We illustrate the significance of instruction scheduling on switching activities using the example in Figure 7. We measure the switching activities for each of the three instruction sequences. The switching activities are normalized with respect to that of the instruction sequence I. The normalized switching activities of instruction sequences II and III are 1.05 and 1.45 respectively. Although instruction sequences I and III are good scheduled codes in terms of performance (only one pipeline stall



(a) data dependencies graph

```
label(l(move/4,7)).
1     mov(e,t0).                 %D(1,2) = 0
2     umax(b,e,e).               %D(2,3) = 0
3     addi(e,4,e).               %D(3,4) = 0
4     pushd(t0/cp,e,2).          %D(4,6) = 0
6     ldi(-1,t1).                %D(6,7) = 0
7     std(r(2)/r(1),e+ -4).      %D(7,8) = 0
8     add28(r(0),t1,r(0)).       %D(8,10) = 0
10    mov(r(3),r(2)).            %D(10,9) = 0
9     st(r(0),e+ -6).            %D(9,5) = 1
```

Pipeline stalls = 1

(b) Instruction sequence I

```
label(l(move/4,7)).
6     ldi(-1,t1).                %D(6,1) = 0
1     mov(e,t0).                 %D(1,2) = 0
2     umax(b,e,e).               %D(2,3) = 0
3     addi(e,4,e).               %D(3,8) = 0
8     add28(r(0),t1,r(0)).       %D(8,4) = 0
4     pushd(t0/cp,e,2).          %D(4,7) = 1
7     std(r(2)/r(1),e+ -4).      %D(7,9) = 1
9     st(r(0),e+ -6).            %D(9,5) = 1
5     st(r(3),e+ -5).            %D(5,10) = 0
10    mov(r(3),r(2)).            %D(10,11) = 0
```

Pipeline stalls = 3

(c) Instruction sequence II

```
label(l(move/4,7)).
1     mov(e,t0).                 %D(1,2) = 0
2     umax(b,e,e).               %D(2,6) = 0
6     ldi(-1,t1).                %D(6,3) = 0
3     addi(e,4,e).               %D(3,4) = 0
4     pushd(t0/cp,e,2).          %D(4,5) = 1
5     st(r(3),e+ -5).            %D(5,8) = 0
8     add28(r(0),t1,r(0)).       %D(8,7) = 0
7     std(r(2)/r(1),e+ -4).      %D(7,10) = 0
10    mov(r(3),r(2)).            %D(10,11) = 0
11    ld(e-4, r3).               %D(11,9) = 0
```

Pipeline stalls = 1

(d) Instruction sequence III

Figure 7 A DAG and its scheduled code

cycle), instruction sequence III has the worst switching activity cost (45% more than that of instruction sequence I). On the other hand, the switching activities of instruction sequence II is quite close to that of instruction sequence I (only 5% more), though instruction sequence II has the worst performance (three pipeline stall cycles).

The above example indicates that there is no clear correlation between performance and switching activities. The design for low power hence does not conflict with the design for performance. In other words, we may be able to achieve both goals at the same time.

4.3 Phase Problem of Instruction Scheduling and Assembly

In order to implement cold scheduling, structure of a traditional compiler backend needs to be modified. In a traditional compiler backend, instruction scheduling and register allocation are processed before assembling code. Instructions and their registers are represented in a symbolic form. Usually, data and control dependency graphs can be easily derived from instructions in a symbolic form. However, it may not be easy to derive the bit switching information between instructions from their symbolic forms due to the following reasons: (1) the target addresses of branch/jump instructions may not be known before instructions are scheduled and registers are allocated, (2) during instruction scheduling and register allocation, the sizes of basic blocks may be changed, (3) peephole optimization may change sequence of instructions in a basic block, and (4) the binary representation of indexes to the symbol table may not be available.

The above problem is called the *phase problem of instruction scheduling and assembly.* Instruction scheduling preceding assembly may degrade the impact of reducing bit switches between instructions. However, when assembly precedes instruction scheduling, the flexibility of instruction scheduling is limited. A similar problem is the phase problem between instruction scheduling and register allocation [Bradlee 91], where instruction scheduling preceding register allocation may increase register pressure and instruction scheduling following register allocation may introduce false dependencies.

A simple solution to deal with the phase problem of instruction scheduling and assembly is to derive or "guess" binary representations of instructions before instruction scheduling. We introduce a novel compiler structure, shown in Figure 8, which divides an assembler into two parts, pre-assembler and post-assembler. The major tasks of the pre-assembler are to calculate target address of branch/jump instructions, indexes to symbol table, and transform instructions to binary form. The major task of the post-assembler is to do the rest of work in an assembler. One advantage of partitioning an assembler is that having binary representations of instructions available before instruction scheduling allows us to proceed cold scheduling. This scheme however will limit the ability of the instruction scheduler to schedule instructions across basic blocks since target addresses of branch/jump instructions are decided before instruction scheduling. For instruction scheduling like trace scheduling, percolation scheduling, and global scheduling, this scheme is hard to apply.

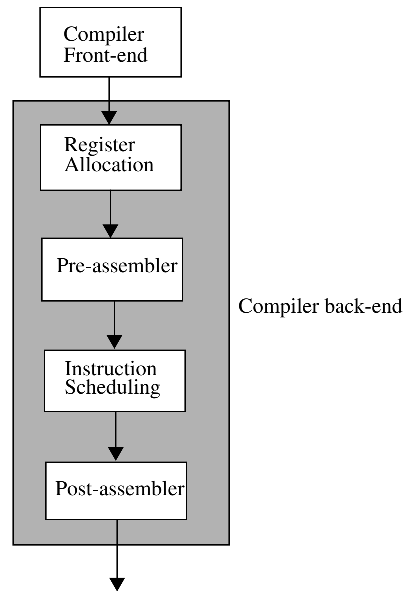


Figure 8 Pre-assembler, instruction scheduler, and post-assembler

4.4 Inputs of Cold Scheduling Algorithm

Two inputs are needed for cold scheduling: (1) data dependency graphs for benchmark programs and (2) a power cost table for each pair of instruction.

The data dependency graph is constructed based on the instruction-level dependencies. Given an instruction stream, a DAG is built by backward pass construction, in which the instruction stream is scanned backwards. For each resource R, R.input and R.output represent sets of instructions which use R as input and output resources respectively. Typical resources are general registers, special registers (the program counter, true/fault bit,...,etc,), and memory. The details of the DAG construction algorithm are shown in Figure 9.

Every entry, $S(I_i,I_j)$, in the power cost table represents the switching activities invoked in the processor by executing the pair of consecutive instructions I_i,I_j. Since the instruction bus mainly drives modules inside the control path, we only include the switching activities of the control path in the power cost entry. In Figure 2, we can see that the pipelined control path contains the instruction registers, PLAs and control signal output latches. For every pair of instructions (I_i,I_j), $S(I_i,I_j)$ is then obtained by summing up all the gates in the instruction registers, PLAs and control signal output latches that are going to switch (0->1 or 1->0) when I_j is executed right after I_i. If the capacitance loading of each gate is known, the actual power cost can be obtained by summing up the capacitance loading of all switching gates.

DAG construction

INPUT: An instruction stream.
OUTPUT: DAG representation.

0. Reverse the instruction sequence from an instruction stream.
1. For any resource *R*,
   set *R.input* to be *{}*
   set *R.output* to be *{}*
2. Visit an instruction *I*, identify its input and output resources, *INs* and *OUTs*.
3. For each element *Y* in *OUTs*,
   IF there is any instruction *J* associated with *Y.input*,
      THEN create an arc *(I,J)*.
   IF there is any instruction K associated with *Y.output,*
      THEN create an arc *(I,K)*.
   set *Y.output* to be *{I}*.
   set *Y.input* to be *{}*.
4. For each element *X* in *INs*,
   IF there is any instruction *L* associated with *X.output*,
      THEN create an arc *(I,L)*.
   add *I* into *X.input*.
5. IF there is any instruction yet to be scheduled,
   THEN go to step 2.
   ELSE return.

Figure 9 DAG construction algorithm

4.5 Cold Scheduling Algorithm

One simple implementation of cold scheduling is list scheduling with heuristics targeted for low bit switching. Given a DAG, cold scheduling first selects instructions which are ready to be executed. An instruction is defined as ready to be executed if all its operands and resources are ready. All ready instructions are collected in a ready list. Instructions with the highest priority in the list will be selected to be executed in the next cycle. The priority of an instruction in the ready list is measured by the power cost when this instruction is selected to be executed in the next cycle. The less the power cost, the higher the priority of this instruction. The priority of an instruction in the ready list has to be recalculated for each cycle until the instruction is selected.

After executing the selected instruction, some instructions may become ready if they depend on the selected instruction. These new ready instructions are then added to the priority list. The instruction that has the highest priority in the list will be selected as the next instruction to be executed. If there are still instructions that haven't been scheduled and the ready list is empty, we simply put a NOP (No OPeration) instruction for the next execution cycle. If the target machine has the ability to detect pipeline hazards, then we just ignore the next execution cycle. This process is continued until all instructions are scheduled. Figure 10 shows the algorithm for cold scheduling.

4.6 Results

We have implemented the cold scheduling algorithm in the Aquarius compiler system. The original instruction scheduler in the Aquarius compiler system is used for comparison. For the sake of simplicity, we only

Cold Scheduling

INPUT: DAG representation and bit switching table
OUTPUT: A scheduled instruction stream

0. Set ready list RL to be { }
   Set the last scheduled instruction LSI = NOP
1. Remove ready instructions from DAG and add these ready instruction into RL.
2. For each instruction I in RL,
   find S(LSI,I).
3. Remove an instruction I with the smallest S(LSI,I) from RL. The removed instruction becomes the current LSI. Write out LSI.
4. IF there is any instruction yet to be scheduled,
   THEN go to step 1,
   ELSE return.

Figure 10 Cold Scheduling Algorithm

include the switching activities at the input and output latches in the power cost entry. Figure 11 shows the reduction in switching activities for different benchmark programs using cold scheduling over that uses the original performance-driven instruction scheduler. The results show an 20 ~ 30% reduction in switching activities. Although the study in this paper is limited to circuits in the control path, the amount of switching activities saved by cold scheduling is still significant.

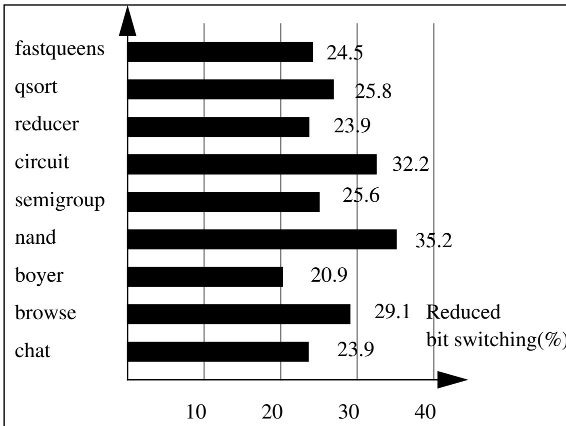


Figure 11 Bit switching reduction of cold scheduling

Figure 12 shows the impact of performance using cold scheduling. Compared to a regular performance-driven instruction scheduling, cold scheduling has a 2~4% performance degradation. This minor performance degradation is mainly due to trading bit switches for performance.

5. Conclusion

In this work, we demonstrated that significant reduction in power consumption can be achieved by an architectural decision and compiler techniques. In particular, we presented two novel techniques to minimize switching activities for high-performance processors.

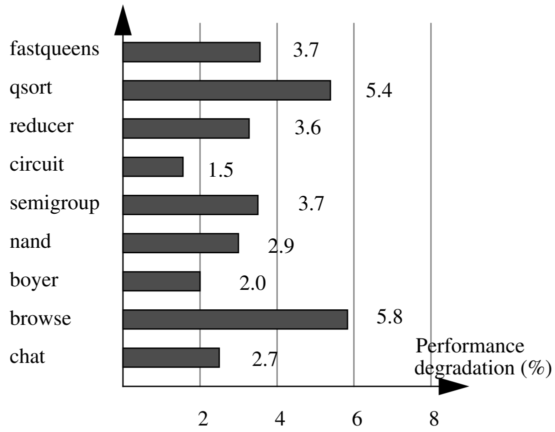


Figure 12 Performance degradation due to cold scheduli

First Gray code is used for the memory addressing scheme instead of the traditional binary coding. Because of the characteristic of Gray code and the consecutive memory referencing nature of processors due to program locality, significant reduction in switching activities is obtained at the address lines. Experimental results show an average 36.9% reduction. In addition, we described a simple scheme which modifying the current optimizing compiler backend to work with the Gray code addressing system.

Second, we developed a new instruction scheduling algorithm, cold scheduling, which minimizes the switching activity using a simple list scheduling algorithm. We implemented it in the Aquarius compiler system. Experimental results show that by using cold scheduling, about 20~30% of the switching activity in the control path are reduced with only a 2~4% degradation in performance comparing with the regular performance-driven instruction scheduler.

In the future, we want to further explore the impact of architectural decisions and compiler techniques on power consumption. In particular, we are working on the instruction set architecture design for a low power processor and hardware/software co-design for low power.

Acknowledgments

The authors would like to thank Chen-Chiu Teng for the valuable discussion on using Gray codes for addressing and John Barr and Steve Crago for reviewing early drafts and valuable comments. The work was supported by ARPA under grant No. J-FBI-91-194.

References

[Bradlee 91] D.G. Bradlee, S.J. Eggers, and R.R. Henry, “Integrating Register Allocation and Instruction Scheduling for RISCs,” the 4th International Conference on Architectural Support for Programming Languages and Operating System, 1991.

[Bernstein 91] D. Bernstein, and M. Rodeh. “Global Instruction Scheduling for Superscalar Machines,” Proc. of the ACM SIGPLAN ‘91 Conf. on Programming Language Design and Implementation, June. 1991.

[Chandra 92] A.P. Chandrakasan, S. Sheng and R.W. Brodersen, “Low-power CMOS digital design,” IEEE J. Solid-State Circuits, Vol. 27, No4, 1992.

[Fisher 81] J.A. Fisher. “Trace Scheduling: A Technique for Global Microcode Compaction,” IEEE Transactions on Computers, Vol. 30, No. 7, 1981.

[Ghosh 92] A. Ghosh, S. Devadas, K. Keutzer, and J. White, “Estimation of Average Switching Activity in Combinational and Sequential Circuit,” the 29th DAC, 1992.

[Haygood 89] Haygood, “A Prolog Benchmark Suite for Aquarius,” Technical Report, Computer Science Department, University of California, UCB/CSD 89/509, 1989.

[Hayes 88] J.P. Hayes, “Computer Architecture And Organization,” McGraw-Hill Int. Editions, 1988.

[Holmer 90] B. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. Bush, and A. Despain. “Fast Prolog with an Extended General Purpose Architecture,” the 17th Annual International Symposium on Computer Architecture, May 1990.

[Hwu 92] W.W. Hwu and P.P. Chang, “Efficient Instruction Sequencing with Inlining Target Insertion,” IEEE Transactions on Computers, Vol. 41, No.12, Dec. 1992.

[Jouppi 89a] N.P. Jouppi, and D.W. Wall. “Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines,” the 3rd International Conference on Architectural Support for Programming Languages and Operating System, 1989.

[Liu 93] D. Liu, and C. Svensson, “Trading Speed for Low Power by Choice of Supply and Threshold Voltages,” IEEE J. of Solid State Circuits, Vol. 28, No. 1, 1993.

[MIPS 86] “MIPS language programmer’s guide,” MIPS Computer Systems, Inc., 1986

[Nicolau 84] A. Nicolau, J.A. Fisher, “Measuring the Parallelism Available for Very Long Instruction Word Architectures,” IEEE Transactions on Computers, Vol. 33, No. 11, 1984.

[Prasad 93] S. Prasad and K. Roy, “Circuit activity driven multilevel logic optimization for low power reliable operation,” EDAC, February, 1993.

[Roy 92] K. Roy and S. Parsad, “SYSLOP: Synthesis of CMOS logic for low power application,” ICCD, October, 1992.

[Shen 92] A. Shen, A. Ghosh and S. Devadas,”On Average Power Dissipation and Random Pattern Testability of CMOS Combinational Logic Networks”, IEEE ICCAD,Nov, 1993.

[Su 92] C.L. Su, “An instruction Scheduler and Register Allocator for Prolog Parallel Microprocessors,” International Computer Symposium, 1992.

[Tsui 93] C.Y. Tsui, M. Pedram, and A.M. Despain, “Technology Decomposition and Mapping Targeting Low Power Dissipation, “the 30th DAC, 1993.

[Vaishnav 93] H. Vaishnav and M. Pedram, “Pcube: A Performance driven placement algorithm for low power designs,” EURO-DAC, September,1993.

[Van Roy 92] P. Van Roy and A. M. Despain, “High-Performance Logic Programming with the Aquarius Prolog Compiler,” Computer, January 1992.

[Weste 93] Neil H.E.Weste and K. Esharaghian, “Principles of CMOS VLSI Design, A Systems Perspective,” Addison Edition 1993.