Cold Scheduling: Schedule Instructions For Less Bit Switches

Ching-Long Su and Alvin M. Despain

CENG Technical Report 93-45

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-9143

September 1993

Cold Scheduling :
Schedule Instructions For Low Power Consumption

Ching-Long Su and Alvin M. Despain

Advanced Computer Architecture Laboratory
University of Southern California
Los Angeles, CA 90089-2562, USA
Phone: (213) 740-9143
E-mail: csu@usc.edu
September 15, 1993

Abstract

Processors for low power consumption are important for future portable computers. In a CMOS processor, bit switching activities consumes the majority of electrical power. Reducing bit switching would significantly reduce power consumption of a processor chip.

In this paper, we provide a novel instruction scheduling algorithm, named cold scheduling, which schedules instructions in a way such that bit switching rate is low. This scheduling algorithm can be used not only in future low voltage processors (e.g. 1~3V processors), but also currently existing processors. For example, we experiment with cold scheduling on a typical RISC processor, the VLSI-BAM. By running through benchmarks using cold scheduling, bit switches in this processor, are only 70~80% of what is seen using a regular instruction scheduling. The results also show that the performance reduction due to cold scheduling is not a concern. Only a 2~4% performance reduction is seen using cold scheduling as compared to the state-of-the-art performance driven instruction scheduler.

1. Introduction

Recent advanced VLSI technology has allowed computers to become portable. Portable computers require not only high performance (high speed and high throughput), but also low power consumption (long battery life time). Reduced power consumption can also help to reduce the cost of the cooling system and add more functionality to portable computers.

In general, the average power consumption of a circuit could be improved by the following methods (1) lower supply voltages (e.g. 1.5 to 3V), (2) lower capacity circuit design by improved technology or redesign of library cells with low capacity, (3) architectural designs (e.g. stop clocking while computation is not activated), and (4) reduce bit switches (e.g. low internal node switches of combinational circuits in logic synthesis). In this paper, we first investigate the bit switching rate of executing programs on a particular machine. We found that the switching rate of executing a regular program is highly related to the sequence of instructions been executed.

Traditional instruction scheduling algorithms mainly focus on reordering instructions in order to reduce pipeline stalls, avoid pipeline hazards, or improve resource usage. More recent instruction scheduling algorithms like trace scheduling [Fisher 81], percolation scheduling [Nicolau 84], and global scheduling [Bernstein 91] mainly schedule instructions cross basic blocks in order to increase instruction-level parallelism. The main goal of these scheduling algorithms is to improve performance. To consider low bit switching, these instruction scheduling algorithms need to be modified in order to adopt this new design metric.

In this paper, we first identify bit switching activities during instruction execution, which includes bit switches among instruction bits and control logic. We then evaluate bit switching rates of programs which were scheduled via traditional instruction scheduling algorithms. We also propose a novel instruction scheduling algorithm, named *cold scheduling*, that is especially designed to reduce bit switching during instruction execution.

To validate the algorithm, we compare the bit switching rate of several programs via cold scheduling and traditional performance driven instruction scheduling on a general pipelined machine. The results show that the bit switching rate can be reduced 20~30% by cold scheduling with minimal sacrifice of the performance.

This paper is organized into six sections. Section 2 presents low bit switching models and hardware assumptions of this study. Section 3 describes our instruction scheduling algorithms. Sections 4 reports the experimental results. Finally, conclusion remarks are provided in Section 5.

2. Bit Switching

In a CMOS design, energy is consumed during the charging and discharging of capacitances [Weste 93], which mainly occurs during bit switches (0 to 1, 1 to 0). For example, a traditional pipelined circuit consists of a combinational circuit between two latches, shown in Figure 1. The input signals of the combinational circuit is first latched in the input latch. These input signals will be evaluated in the combinational circuit. The output signals of the combinational circuit are then latched in the output latch. These output signals of the output latch will become the input signals of input latch for the combinational circuit at the next pipeline stage. While these signals travel through pipeline stages, bits in each pipeline latch are switching. Based on this property of pipeline circuitry, we develop the first hypothesis of this paper.

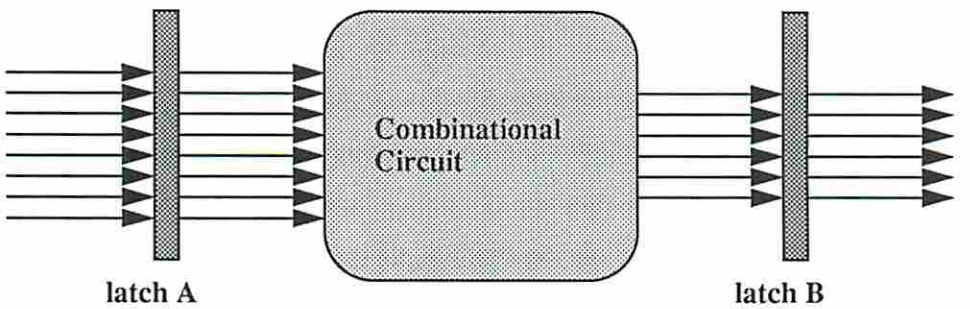


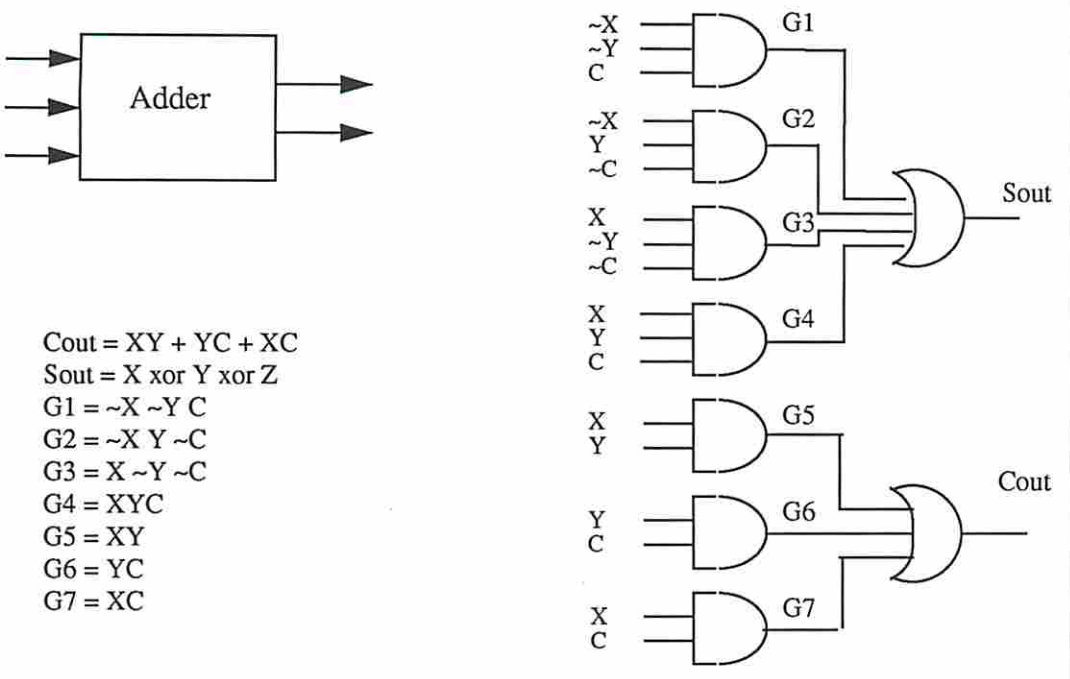
Figure 1 Combinational circuit and latches

Hypothesis 1:

In general, if the bit switching rate of the input latch is high, then the corresponding gate switching rate and transistor switching rate of the combinational circuit tends to be high. In contrast, if the bit switching rate of the input latch is low, then the corresponding gate switches and transistor switches of this combinational circuit tends to be low.

To support this hypothesis, we randomly select a combinational circuit and monitor the bit switching rate of it's input signals and internal gate switches. Figure 2 shows an example of bit switches in an one-bit full adder. This adder consists of seven internal gates (*G1* to *G7*). The inputs of this adder are *X*,*Y*, and *C*, where *X* and *Y* are two input sources and *C* is input carry. The output of this adder are the output carry *Cout* and the sum *Sout*. We monitor bit switches of internal gates of this adder based on several input patterns. *Model I* is assumed the bit switching rate of each input is 0.5. The gate switching rate of *G1*,*G2*,and *G3* is 0.375 and the gate switching rate of *G4*,*G5*,*G6*, and *G7* is 0.219. The switching rates of *Cout* and *Sout* are the same, 0.5. When the bit switching rate of each input is reduced to 0.25 (shown in *Model II*), the gate switching rate of *G1*,*G2*,and *G3* is reduced to 0.219 (41% reduction compared to *Model I*); the gate switching rate of *G4*,*G5*,*G6*, and *G7* is also reduced to 0.144 (34% reduction compared to *Model I*). *Cout* and *Sout* is also reduced to 0.296 and 0.438 respectively. Furthermore, when the bit switching rate of each input is further reduced to 0.125 (shown in *Model III*), the gate switching rate of *G1*,*G2*,and

G3 is reduced to 0.117 (46% reduction compared to *Model II*); the gate switching rate of *G4*,*G5*,*G6*, and *G7* is also reduced to 0.083 (42% reduction compared to *Model II*). *Cout* and *Sout* is further reduced to 0.166 and 0.289 (which is a 60% and 34% reduction compared to *Model II*).



Model	X	Y	Z	G1	G2	G3	G4	Sout
I	0.500	0.500	0.500	0.219	0.219	0.219	0.219	0.500
II	0.250	0.250	0.250	0.144	0.144	0.144	0.144	0.438
III	0.125	0.125	0.125	0.083	0.083	0.083	0.083	0.289

Model	X	Y	Z	G5	G6	G7		Cout
I	0.500	0.500	0.500	0.375	0.375	0.375		0.500
II	0.250	0.250	0.250	0.219	0.219	0.219		0.296
III	0.125	0.125	0.125	0.117	0.117	0.117		0.166

Figure2 Switching rates of a one-bit adder

From the above example, we see the relationship between the switching rate of internal gates of a combinational circuit and the switching rate of its input latches. Certainly, a circuit may be found where reducing an individual input signal may not significantly contribute to reducing the bit switching rate of any internal gates. However, generally reducing the switching rate on each bit of the input latch will eventually reduce switching rates of internal gates of a combina-

tional circuit. Therefore, we can conclude that reducing the switching rate of input latches will significantly reduce switches of internal gates in a combinational circuit.

Bit switching rates of latches in a pipelined microprocessor are effected by various factors. For latches in the data path, bit switching rates are mainly affected by run-time data sequences been executed. For latches in the control path, bit switching rates are strongly affected by instruction execution sequences. Since run-time data is hardly known at compile-time, the impact of cold scheduling on bit switching rate of latches in the data path is not clear. However, instruction execution sequences are controlled by instruction scheduler at compile-time. The impact of cold scheduling on bit switching rate of latches in the control path is more directly. In this paper, we only focus on the impact of cold scheduling for reducing bit switching rate of latches in the control path.

Bit switching activities among different instruction sequences can be significantly different. In a CISC processor, this difference may not be obvious since one instruction may be executed for many processor cycles. In contract, for a general pipelined RISC-like micro-processor, in which most instructions can be executed in a processor cycle, this different can be significant since instruction scheduler has more instructions to schedule.

To better understanding the impact of an instruction sequence on switching rate of latches in general purpose processors, we select a RISC-like micro-processor, the VLSI-BAM [Holmer 90], as an experimental architecture. This microprocessor is pipelined with data stational 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 3 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 data path.

A cycle-by-cycle instruction-level simulator is built for collecting bit switching counts of latches in the control path during execution of benchmark programs. Benchmark programs used in this paper are shown in Table 1. The benchmarks are ranging from less than 1,000 cycles or larger than 10,000,000 cycles. These benchmark programs are selected from the Aquarius benchmark suite [Haygood 89]. Applications of these benchmark programs are various, including list manipulation, a data base query, a theorem prover, and a 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 benchmark program in BAM code is further compiled into machine code of the target machine, the VLSI-BAM.

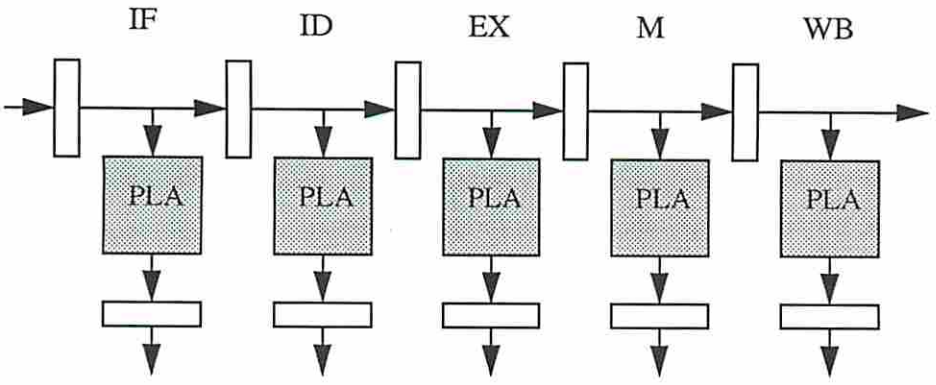


Figure 3 Pipeline structure and control path of VLSI-BAM

Benchmark	Line	Cycles	Description
nreverse	145	4,287	Naive reverse of a 30-element list
qsort	225	4,560	Quicksort of a 50-element list
query	397	97,259	Query a static database
circuit	1315	4,504,940	VLSI module generator
semigroup	4395	4,487,201	Query a data base
zebra	985	2,071,780	A logical puzzle based on constraints
boyer	9918	27,494,723	An extract from a Boyer-Moore theorem prover
browse	1295	18,883,712	Build and query a database
chat	114312	3,303,153	A small subset of English for database querying

Table 1 Benchmark programs

3. Cold Scheduling

In this section, we present the details of cold scheduling algorithm. Basically cold scheduling use traditional performance-driven scheduling techniques with heuristics especially for reducing bit switching rates. Before we go into the details of cold scheduling, we first review a traditional list scheduling algorithm.

3.1 Metric For Performance

A traditional instruction scheduling approach consists 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.

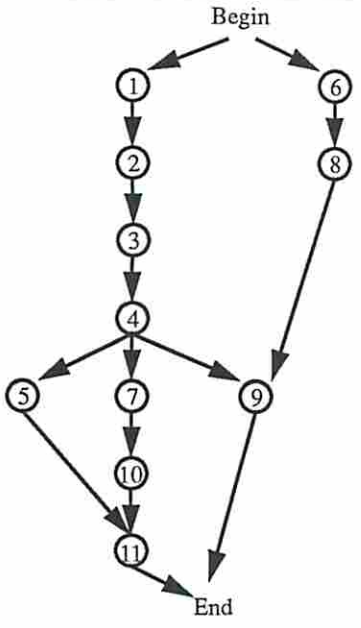
The main goal of a traditional instruction scheduler is to schedule the instruction sequence such that it can be executed in a target machine as fast as possible with minimal pipeline stalls. Therefore the quality of an instruction scheduler can be defined as the amount of pipeline stalls introduced 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. This instruction sequence B is executed in a machine. The number of pipeline stall cycles between execution of instruction I_j and I_{j+1} is recorded 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 design metric of an instruction scheduler is therefore to minimize PS. In the case of scheduling a region, the design metric of an instruction scheduler is to minimize the pipeline stalls of this region, instate of 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 design metric of an instruction scheduler for a region is therefore 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 4 shows a data dependency graph and varies output instruction sequences from a typical instruction scheduler. The target machine is assumed to have 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 sequence I and III have the best scheduling with only one pipeline stall cycle.

3.2 Metric For Bit Switching

Unlike a traditional instruction scheduler whose main goal is to minimize pipeline stalls between instructions, an instruction scheduler for cold scheduling reorders the code such that bit switches are minimized without introducing significant performance degradation.Therefore, the quality of cold scheduling can be defined as the amount of bit switches in the output instruction sequence can be saved when it is executed on a target machine. Similar to the definition of the quality of traditional instruction scheduling which defined $D(I_j, I_{j+1})$ as pipeline stall cycles between execution of I_j and I_{j+1}, we defined bit switches $S(I_j,I_{j+1})$ as the number of bit switches in latches of a control path between execution of I_j and I_{j+1}. Let $B = I_1,I_2,...$ be an output instruction sequence of a basic block. This instruction sequence B is executed on a machine M. The total bit switches is defined as $BS = \Sigma S(I_j,I_{j+1})$, $j = 0.. n-1$. The main design metric of cold scheduling is to minimize BS. In the case of scheduling a region, the design metric of cold scheduling is to minimize the bit switches of this region, instate of individual basic block. For example, if a region consists of basic block $B_1,B_2,...B_k$, and the number of bit switches in these basic block are BS_1, $BS_2,...BS_k$, then, the design metric of an instruction scheduler for a region is therefore to mini-



(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
5    st(r(3),e+ -5).             %D(5,11) = 0
11   ld(e-4,r3).

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
11   ld(e-4,r3).

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
9    st(r(0),e+ -6).             %D(9,10) = 0

Pipeline stalls = 1
```

(d) Instruction sequence III

Figure 4. Sample input/output code sequences and their dependency DAG

mize $1/k(w_1*BS_1, w_2*BS_2,...,w_k*BS_k)$, where w_j is a weight of estimated dynamic execution frequency of a basic block B_j.

Using DDG and the instruction sequences in Figure 4 as an example, we show the switching activities among these instruction sequences in Figure 5. The normalized bit switching is defined

as bit switches of an instruction sequence against bit switches of instruction sequence I. Although instruction sequence I and III are good scheduled codes in terms of performance (only one pipeline stall cycle), instruction sequence III has the worst bit switching rate (45% more than that of instruction sequence I). On the other hand, the bit switching rate of instruction sequence II is quite close to instruction sequence I (only 5% more), though instruction sequence II has the worst performance (three pipeline stall cycles).

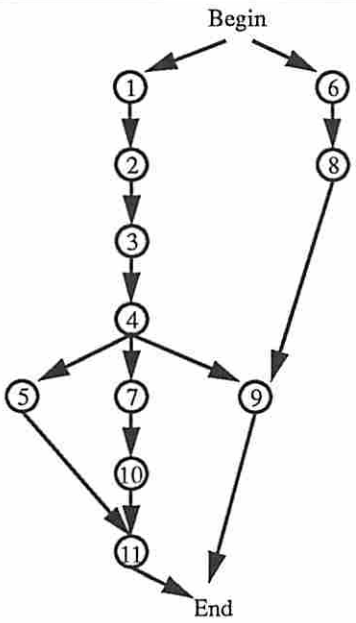
The above example indicates that the relation of performance and bit switching is not clear. The design for low bit switching does not conflict with the design for performance. In other words, we may be able to achieve both goals at the same time.

3.3 Phase Problem of Instruction Scheduling and Assembly

In order to achieve the new design metric, reduction of bit switches, 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 for scheduler and register allocator 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 possible to derive the bit switching information between instructions from their symbolic forms. For example, (1) the target addresses of branch/jump instructions may not be known before instructions are scheduled and their 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 that proceeds assembly may degrade the impact of reducing bit switches between instructions.owever, when assembly proceeds 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 proceeding 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 6, 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 post-assembler is to do the rest of work in an assembler. One advantage of partitioning an assembler is that binary representations of instructions are available before instruction scheduler in order to process cold scheduling. This scheme however will limit the ability of the instruction scheduler to schedule instructions cross basic blocks; since target addresses of branch/jump instructions are decided before instruction scheduled. For instruction scheduling



(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
5     st(r(3),e+ -5).           %D(5,11) = 0
11    ld(e-4,r3).

Pipeline stalls = 1
Normalized Bit Switching = 1.0
```

(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
11    ld(e-4,r3).

Pipeline stalls = 3
Normalized Bit Switching = 1.05
```

(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
9     st(r(0),e+ -6).           %D(9,10) = 0

Pipeline stalls = 1
Normalized Bit Switching = 1.45
```

(d) Instruction sequence III

Figure 5. Sample input/output code sequences and their dependency DAG

like trace scheduling, percolation scheduling, and global scheduling, this scheme is hard to apply.

A more inexpensive scheme could be the following. Since some of symbolic representations of an instruction can be exactly predicted (e.g. opcode, register operand, indexes to symbol table

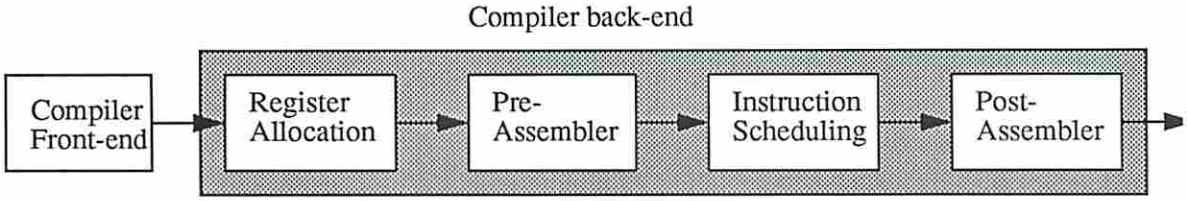


Figure 6. Pre-assembler, instruction scheduler, and post assembler

and displacement), we still can implement cold scheduling without the pre-assembler. For symbolic representations of an instruction which cannot be correctly predicted, an estimation of binary representation is applied. The advantage of this scheme is that the current compiler structure will not be changed much. Most of the existing compile backend can still be applied. The disadvantage of this scheme is that the estimation of binary representation for unpredictable target addresses before instruction scheduler may not be highly accurate. This may degrade the effects of cold scheduling.

Since we want to see how many bit switch activities can be eliminated with cold scheduling, only the first scheme is investigated in this paper. The other scheme is also quite interesting. For future studies, we would like to see how accurate the binary representations can be predicted from the symbolic representation of instructions before assembly.

3.4 Inputs of Cold Scheduling Algorithm

Two inputs are needed for cold scheduling: (1) data dependency graphs for benchmark programs and (2) a bit switch table for each opcode.

The data dependency graph can be easily constructed based on 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 records a set of instructions which are going to use and define the resource R respectively. Resources are defined as general registers, special registers (the program counter, true/fault bit,...,etc,), and memory. We show the DAG construction algorithm in Figure 7.

The bit switch table is a mapping table of active control signals at each pipeline stage and opcode. This table can be constructed when the target processor was built. Based on this table and some operands associated with the instruction, the number of bit switches of control signals in pipeline stages between two instructions can be calculated. Table 2 shows the bit switch table of the opcodes in VLSI-BAM. A control array of an instruction is defined as a list of control signals associated with a given instruction. To efficiently calculate the number of bit switches between two instructions, we simply apply exclusive or on control arrays of both instructions. The number of '1' shown in the result is the number of bit switches when both instruction are executed in the pipeline stages.

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 7. DAG construction algorithm

		Control signals													
		c1	c2	c3	c4	c5	c6	● ● ●	c89	c90	cr91	c92	c93	c94
opcode	add	1	1	1	0	1	1	● ● ●	1	1	1	1	0	1
	jmp	1	1	1	1	0	1	● ● ●	1	0	0	1	1	1
					●						●			
					●						●			
					●						●			
	load	0	1	0	1	0	0	● ● ●	0	1	0	0	0	0
	store	0	1	0	0	1	0	● ● ●	0	0	0	0	1	0

Table 2 Bit switching table

3.5 Cold Scheduling Algorithm

One simple implementation of cold scheduling is a list scheduling with heuristics target 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 in the executed next cycle. The priority of an instruction in the ready list is defined as the amount of bit switches introduced when this instruction is selected to be executed in the next cycle. The less amount of bit switches introduced by an instruction, the higher priority of this instruction is. The priority of an instruction in the ready list could be changing 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 into the priority list. A new instruction with the highest priority in the list will be selected as the new instruction to be executed in the next cycle. 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 have been executed. Figure 8 shows the algorithm for cold scheduling.

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,
   Calculate the number of bit switches between current LSI and I.

3. Remove an instruction with the least number of bit switches with current LSI 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 8 Cold Scheduling Algorithm

4. Experimental Evaluation

We implement this simple list scheduling with heuristics for low bit switching described above in the Aquarius compiler system. The original instruction scheduler in the Aquarius compiler system is used for comparison. Figure 9 shows the reduction of bit switching rates of benchmark programs scheduled by cold scheduling comparing to that by a regular performance-driven

instruction scheduler. In average, using cold scheduling, about 20 ~ 30% of bit switches can be avoid. Although the study in this paper is limited to circuits in the control path, the amount of bit switches been saved by cold scheduling is still significant.

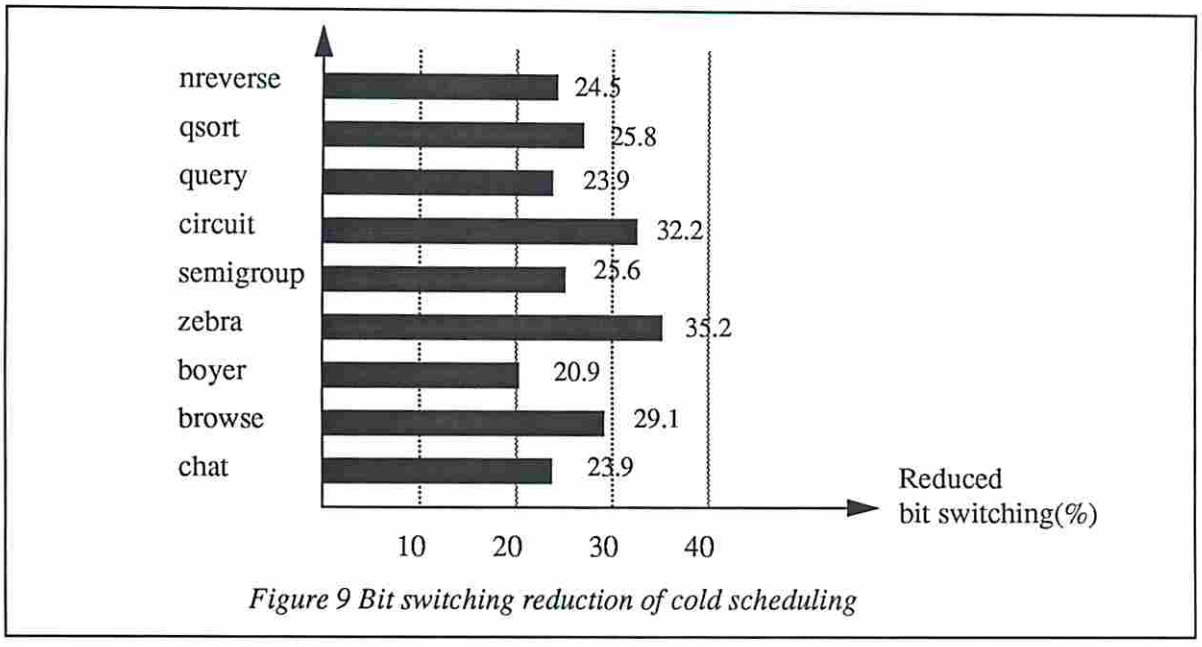


Figure 9 Bit switching reduction of cold scheduling

Figure 10 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 miner performance degradation is mainly due to trading bit switches for performance.

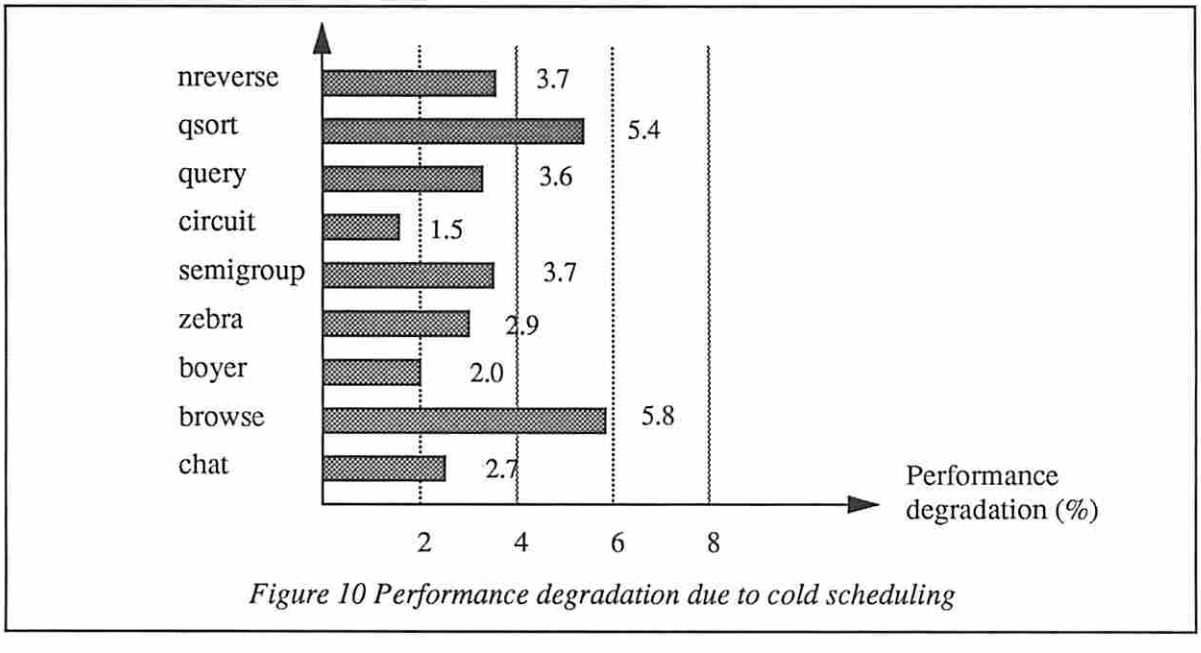


Figure 10 Performance degradation due to cold scheduling

5. Conclusion

This paper describes an instruction scheduling algorithm, call cold scheduling, that is appropriate for low power consumption computers. In a CMOS circuit, electrical power is consumed while capacitors are charging and then discharging. This charging and discharging activity is defined as a bit switch. Therefore, reducing bit switches during execution of programs can significantly save power. For a combinational circuit, if the bit switching of input signals is low, then the bit switch of internal gates is also low. In a pipeline processor, bit switches of latches in the control path are strongly effected by the sequence of instructions been executed. In other words, reducing bit switches through instruction scheduling (cold scheduling) could significantly impact the power consumption.

We experimented the cold scheduling through a simple list scheduling with heuristics target for low bit switches. We implement cold scheduling in Aquarius compiler system. The results are quite encouraging. By using cold scheduling, about 20~30% of the bit switches can be avoid. Comparing performance of benchmark programs scheduled by a regular performance-driven scheduler, cold scheduling suffers only a 2~4% performance degradation. For portable computers where power is more of a concern than performance, cold scheduling is a good choice.

This work offers the encouraging evidence that compiler techniques can have significant contribution of reducing power consumption. This paper only focused on the impact of instruction scheduling to bit switch activities in the control path of a particular RISC instruction set processor. The impact of some other compiler techniques (e.g. register allocation) to bit switching has not been verified yet. We believe that the impact of other compiler techniques to bit switching could be also significant.

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.

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

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

[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.

[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.

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

[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.