Proceedings of International Conference
on Computer Architecture

Design and Performance Evaluation of Branch-Skipped Reduced Instruction Set Computers

Shyh-Jye Wang, Phen-Lan Lin * and John D. Provence **

ASIC Division
Taiwan Semiconductor Manufacturing Company
Hsin-Chu, Taiwan

* Department of Computer Science and Information Management
Providence University
Shalu, Tai-Chung, Taiwan

** Mixed Signal Products Group
Texas Instruments
Dallas, Texas 75265, U.S.A.

Abstract

A pipelined RISC has been designed to incorporate the branch-skipped feature. Branch instructions are detected in the first stage of the pipeline and branch target instructions are made available early enough to let the RISC skip all branch instructions. The performance of this RISC is significant compared with other RISCs, with or without the branch-skipped feature.

1 Introduction

The design of a pipelined reduced instruction set computer (RISC) is well known [1]. The maximum throughput of a pipeline can only be obtained if the pipeline can be kept full. The execution of a branch instruction will cause the contents of part of the pipeline to be discarded and then reloaded if the branch is taken. Lilja [2] used an equation to calculate the cost of a branch. First, he defined the following parameters:

T_{ave} is the average number of cycles required per instruction,

P_b is the probability that a particular instruction is a branch,

P_t is the probability that a branch is taken, and

b is the branch penalty (that is the number of cycles wasted when a branch is taken).

The average number of cycles per instruction is then

$$T_{ave} = 1 + bP_bP_t. \quad (1)$$

For the simple pipeline model presented here, T_{ave} is determined by the branch penalty b, the percentage of branch instructions executed P_b, and the probability that the branch is taken P_t.

The strategy of reducing the cost of a branch is to reduce any one of the three factors that constitutes the cost. Delayed branch [1], multiple prefetch [3], and the using of branch target buffers [4] are all utilized to reduce the branch penalty. Branch-skipped [5] and branch folding [3, 6, 7] are used to reduce the probability that a particular instruction is a branch instruction. The major difference between the branch folding and the branch-skipped is that the branch-skipped technique detects and resolves all branches, conditional and unconditional, due to its simple architecture, while the branch folding cannot resolve conditional branches all the time. Therefore, branch folding is usually combined with other branch cost reduction techniques such as branch prediction. Branch prediction [4] is used to reduce the probability that a branch is taken to the probability of a wrong prediction. A combination of more than one of the stated techniques have been utilized to reduce the cost of branches by many researchers and companies [2, 6, 8, 9]. For example, both branch folding and branch prediction techniques are used in the AT&T CRISP microprocessor [6]. Another example is made by Gonzalez and Llaberia [8] who have introduced a mechanism called COBRA which incorporates the following schemes: early computation of the target address, multiple prefetch, and delayed branch.

2 The branch-skipped pipelined RISC

We have designed a branch-skipped RISC [5] which implements the instruction set of the 32-bit DLX architecture [1]. Referring to Figure 1, The principal stages of the branch-skipped pipelined RISC are the instruction prefetch stage, instruction fetch stage, instruction decode stage, execution stage, memory stage, and write back stage. These principal stages communicate with each other through signals that represent data, instructions, and control signals during the execution of a RISC instruction.

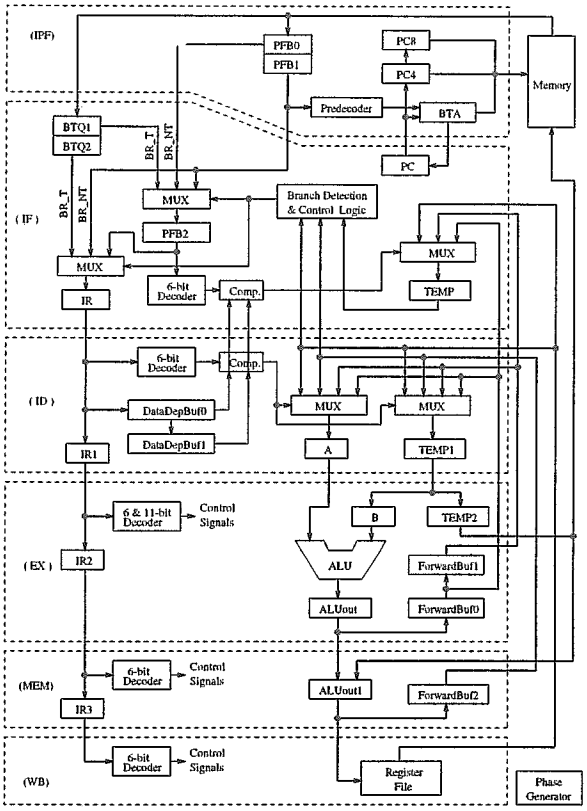

Figure 1: The branch-skipped RISC block diagram.

In investigating the branch path in a pipeline, the only two things that need to be done are determining if the branch is taken or not and determining the branch target address. The idea of the branch-skipped or branch folding techniques is to determine the branch instruction outcomes in the instruction prefetch stage and to prevent branch instructions from being loaded into the later part of the pipeline so that branch instructions can be skipped. It is a waste of time and hardware for the branch instruction to go through other pipeline stages, such as execution, memory access, and write back, since nothing is done in those stages.

The instruction prefetch stage consists of two prefetch buffers, PFB0 and PFB1, and a predecoder. The program counter, PC, is incremented by four and the result, PC4, is used as the memory address for fetching the instruction stored in that address location. The fetched instruction is then stored in PFB1. In the same manner, PC4 is incremented by four and the result, PC8, is used for fetching the next instruction stored in memory. The fetched instruction is then stored in PFB0. The predecoder is used to determine if the instruction in PFB1 is a branch instruction. If a branch instruction is detected, then, the branch target address BTA is generated by a 32-bit adder which sums up PC4 and the displacement indicated in the offset field of PFB1. The branch target address is then used to access the memory and load the branch target instruction and the instruction following the branch target instruction into the branch target queues, BTQ2 and BTQ1, respectively.

The instruction fetch stage is constructed from two branch target queues, BTQ1 and BTQ2, two 32-bit multiplexers, a prefetch buffer, PFB2, an instruction register IR, and a branch detection and control logic. If PFB1 is not a branch instruction, PC is incremented by four. The contents of IR are loaded from PFB2 which are loaded from PFB1 in the previous cycle. When the clock moves to the next machine cycle, we are able to determine if the branch is taken or not by testing the contents of a specific register in the register file or by forwarding the output of the ALU to the control logic if the condition of the branch depends on the current ALU output.

The contents of source register field of PFB2 are compared with the contents of each of the data dependency buffers. If either one is matched, the contents stored in the corresponding forward buffer will be loaded into a temporary register. However, if the comparison between PFB2 and the data dependency buffers is not a match, a specific register in the register file will be accessed and the contents of the register will be stored in the temporary register. If the IPF stage has already detected a branch instruction, then the memory contents pointed by BTA and $BTA + 4$ will be loaded into BTQ2 and BTQ1 respectively. By the end of the current machine cycle, IR is loaded from BTQ2, and PFB2 is loaded from BTQ1, if the branch is taken. Also, PC is loaded by the result of $BTA + 4$ so that a new program sequence path will begin. If the branch is not taken, IR is loaded from PFB1, and PFB2 is loaded from PFB0. In both cases the branch instruction will not be loaded into IR so that the branch instruction is skipped from the later part of the pipeline. PC will increment by four again because the branch instruction is skipped.

Figure 2 is an execution pattern of a branch-skipped pipelined microprocessor. Let BRA be a branch instruction. If branch is not taken, instruction $i + 3$ will succeed instruction $i + 1$ and BRA is skipped. If branch is taken, the new instruction j will succeed instruction $i + 1$ and BRA is skipped as well.

The limitations of the branch-skipped technique are the following: first, a conditional branch cannot depend on the ALU output of the previous instruction. This is because the determination of the branch condition has to be made when the instruction is in the IF stage while its previous instruction is still in the ID stage, and there is no way to know the result of the ALU until the next stage. If we desire to make this possible, extra hardware needs to be added. Second, no consecutive branches are allowed in the machine instructions. This is because when the branch is taken, the instruction is skipped and PFB2 is loaded from BTQ1 which would have never been detected if it were a branch instruction. In this case, the program sequence will be wrong. In the above cases, the compiler can insert an NOP instruction in front of the branch instruction. An NOP instruction costs one machine cycle to execute; however, the average execution time with this technique is still no more than one clock cycle. Let P_c be the probability that

Instruction	Clock number 1	2	3	4	5	6	7	8
i	IPF	IF	ID	EX	MEM	WB		
i+1		IPF	IF	ID	EX	MEM	WB	
BRA			IPF	IF				
i+3				(PFB1)	ID	EX	MEM	WB
j				(BTQ2)	ID	EX	MEM	WB

Figure 2: The execution pattern of the branch-skipped RISC.

a branch instruction does not follow an NOP instruction. The average execution time of a branch skipped processor is

$$T_{ave} = (1 - P_b)(1) + P_b[0 + (1 - P_c)(1)], \quad (2)$$

which can be rearranged to

$$T_{ave} = 1 - P_b P_c. \quad (3)$$

It can be seen from Equation 3 that T_{ave} is independent from the term P_t in the branch-skipped implementation. This is because a branch instruction is always skipped no matter if it is taken or not. From Equation 3, the worst case is when either there is no branch instruction in the whole machine instructions ($P_b = 0$) or there is always an NOP inserted before a branch instruction, ($P_c = 0$) in which case $T_{ave} = 1$ and is not worse than any other previous scheme. However, whenever $P_b P_c$ is not equal to zero, there is always a branch benefit and T_{ave} is less than one. It can also be seen that the bigger the value of P_b, the lesser the value of T_{ave}, which means the higher the percentage of all machine instructions that are branch instructions, the more the execution time can be reduced.

One issue on implementation is the processor cycle time. In this design, on-chip instruction and data caches must be incorporated. Therefore, the added IPF stage would not be the slowest pipeline stage of the processor.

In order to support the branch-skipping scheme, additional hardware needs to be added to the microprocessor. One adder is needed to calculate PC4 and PC8. Two multiplexers are needed to select the paths of the IR and PFB2. One predecoder and one control logic are also needed to control and make decisions. Without taking the on-chip instruction and data caches into account, the hardware overhead is about 15% when the branch-skipped scheme is incorporated.

The compiler for the branch-skipped microprocessor needs to make sure two things will not happen. First, a conditional branch cannot depend on the ALU output of the previous instruction. Second, no consecutive branches are allowed in the machine instructions.

Like the delay-slot in the delayed branch scheme, we will call the pipeline slot which is placed one step earlier than the branch instruction an **early-slot**. If the instruction in the early-slot conflicts with either one of the previous two rules, the compiler shall find an instruction to fill the early-slot under a condition that the data dependency shall not be changed. If a safe instruction is not found, however, the compiler can always insert an NOP instruction into the early-slot.

The workload of a compiler for the branch-skipped microprocessor should be less than that of a compiler for a microprocessor with the delayed branch feature. The compiler for a microprocessor with the delayed branch feature needs to search for useful instructions to fill every delay-slot and reschedule the machine code. On the other hand, the compiler for the branch-skipped microprocessor only needs to check for any consecutive branches or any data dependencies of the branch. If one of the above conditions is found, the compiler will then find a safe instruction or assign an NOP instruction to fill the early-slot.

3 Performance evaluation

In this section, the performance of the branch-skipped RISC is compared with other existing schemes which include the DLX, CRISP [6], COBRA [8], and a RISC without any feature for reducing the cost of branches. We first modeled the RISCs with Verilog HDL and then extracted the parameters required for the calculation below.

The comparisons are based on the performance of T_{ave}, the average number of cycles required to execute an instruction. The variables are b, P_b, and P_t which have been defined in the first Section. The values of some other variables are fixed so as to keep our results small. However, the variation of those fixed variables will not cause a different outcome of the analysis. Those variables and their assumed values are described below: P_{nop} is defined as the fraction of the b delay slots filled with NOP instruction. Assume that P_{nop} is equal to 0.7 for delayed branch based on the statics shown by Lee and Smith [4]. P_w is the probability that a prediction is wrong. For branch prediction, let P_w equal 1 minus P_t if P_t is greater than or equal to 0.5 and let P_w equal P_t if P_t is less than 0.5. The reason is this: for the case of branching in one direction, a 100% correct prediction is received; for the case when branches alternate direction, only 50% of prediction accuracy is received. P_{con} is the probability that a particular branch instruction is a conditional branch. For CRISP, assume that P_{con} equals 0.2. Finally, assume that P_c equals $(1 - P_b)$ for a branch-skipped RISC. Where P_c is the probability that a branch instruction does not follow an NOP instruction. For a branch-skipped RISC, an NOP is inserted before a branch instruction under two conditions: first, when the branch instruction follows another branch instruction; second, when the branch instruction depends on the result generated by the previous instruction and a useful instruction is not found to fill the slot. The first condition will cause a result as assumed. The second condition can be ignored when useful instructions are found to fill the slot.

Now, we will use Amdahl's Law [10] to quantify

our measurement. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Let T_{old} be the execution time for the entire task without using the enhancement and T_{enh} be the execution time for the entire task using the enhancement when possible. Amdahl's Law defines the speedup that can be gained by using a particular feature.

$$Speedup = \frac{T_{old}}{T_{enh}}. \quad (4)$$

Figure 3 to 5 are the performance comparisons when $b = 1$, with $P_b = 15\%$, 25%, and 35%. It can be seen that the performance of the branch-skipped scheme is always the best one. Since the DLX is implemented with the same features the COBRA has, the performance of the COBRA and the DLX are the same.

Lee and Smith found that the average number of branch instructions in a program is about 25% among all instructions [4]. Therefore, when the branch penalty of the normal RISC is one clock cycle, the speedup gained by using branch-skipped feature is from 1.24 to 1.55 depending on the probability that a branch is taken. the higher the probability that a branch is taken, the higher the speedup. Under the same condition, the speedup gained by the CRISP is from 1.06 to 1.30 and the speedup gained by the COBRA is from 1 to 1.09.

The speedup gained by using any feature to reduce the cost of branch is more significant when the percentage of branch instructions in a program is higher. In the case of $P_b = 35\%$, the speedup gained by using branch-skipped feature is from 1.30 to 1.75.

Figure 6 to 8 are the performance comparisons when $b = 3$, with $P_b = 15\%$, 25%, and 35%. The speedup gained by using any feature to reduce the cost of branch is even more significant when the branch penalty of the normal RISC is three clock cycles. In this case, the speedup gained by using branch-skipped feature is from 1.25 to 2.70 when $P_b = 35\%$.

When the branch penalty is three clock cycles, the COBRA outperforms the CRISP in the range between 15% to 75% of P_t. This is because the CRISP incorporates a branch prediction scheme. The probability of a correct prediction is low when the number of taken branches and not taken branches are about the same.

4 Conclusion

The idea that a branch instruction can be skipped from the pipeline stimulated this research. We have shown the design of a branch-skipped RISC in a register transfer level.

The performance of this technique is significant in that while other approaches are targeted on reducing the branch penalty to zero, this approach reduces the branch penalty to a negative value. Therefore, the more branch instructions there are, the more execution time is reduced. When the branch penalty of the normal RISC is one clock cycle and the average number of branch instructions in a program is about 25% among all instructions, the speedup gained by using branch-skipped feature is from 1.24 to 1.55 depending on the probability that a branch is taken. The higher the probability that a branch is taken, the higher the speedup. The technique of branch-skipping shown in this paper can be applied to almost every RISC.

References

[1] D. A. Patterson and J. L. Hennessy, Computer Architecture A Quantitative Approach. 1990, Morgan Kaufmann.

[2] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processor," IEEE Computer, vol. 21, no. 7, July 1988, pp. 47-55.

[3] PowerPC 601 RISC Microprocessor Technical Summary. IBM Microelectronics, Essex Junction, VT, November, 1993.

[4] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, Jan. 1984, pp. 6-22.

[5] S.-J Wang and J. D. Provence, "Branch-Skipped Pipelined Microprocessor," IEE Electronics Letters, vol. 30, no. 14, July 7, 1994, pp. 1122-1123.

[6] D. R. Ditzel and H. R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proc. of the 14th Symposium of Computer Architecture, 1987, pp. 2-9.

[7] J. E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," IEEE Computer, vol. 22, no. 7, July 1989, pp. 21-35.

[8] A. M. Gonzalez and J. M. Llaberia, "Reducing Branch Delay to Zero in Pipelined Processors," IEEE Transactions on Computers, vol. 42, no. 3, March 1993, pp. 363-371.

[9] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," Proc. 13th Symposium on Computer Architecture, 1986, pp. 396-403.

[10] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," Proc. AFIPS 1967 Spring Joint Computer Conf., April 1967, pp. 396-403.

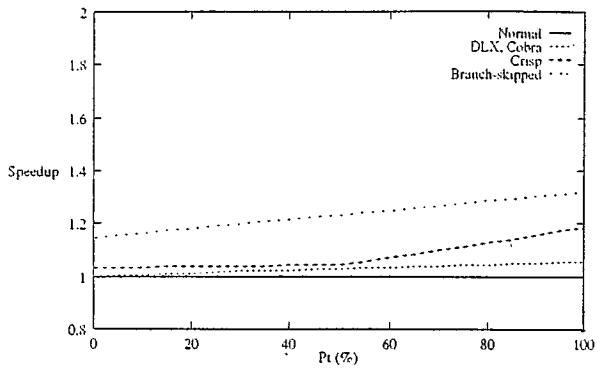

Figure 3: Speedup comparison with $b = 1$, $P_b = 15\%$.

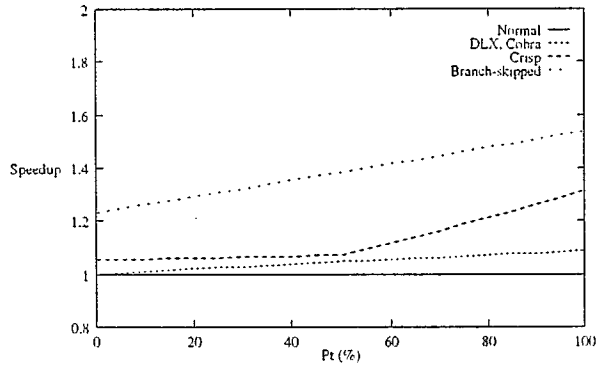

Figure 4: Speedup comparison with $b = 1$, $P_b = 25\%$.

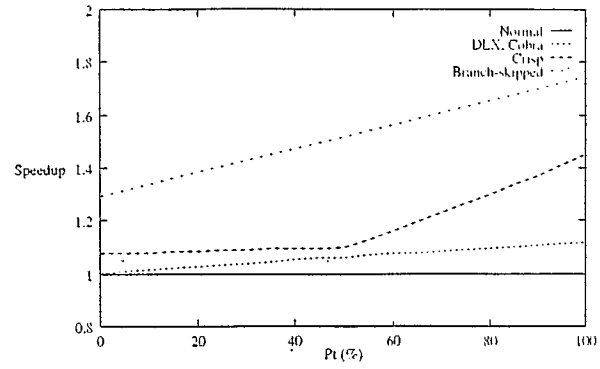

Figure 5: Speedup comparison with $b = 1$, $P_b = 35\%$.

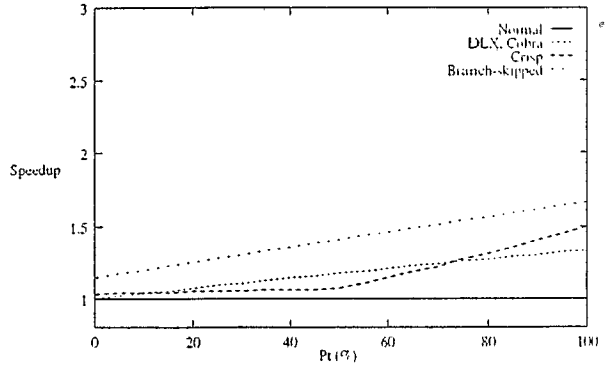

Figure 6: Speedup comparison with $b = 3$, $P_b = 15\%$.

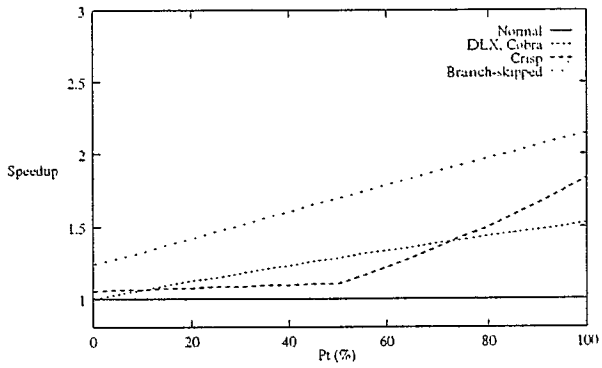

Figure 7: Speedup comparison with $b = 3$, $P_b = 25\%$.

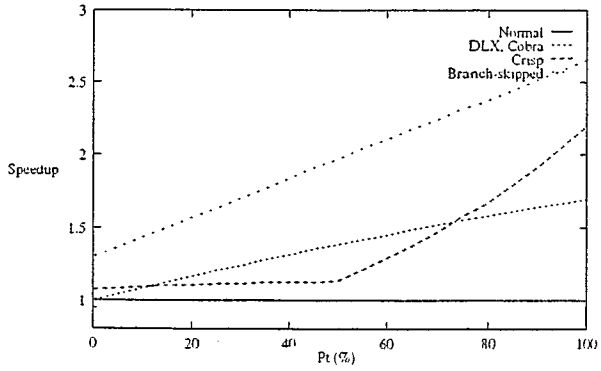

Figure 8: Speedup comparison with $b = 3$, $P_b = 35\%$.