Reducing the Cost of Branches by Using Registers†

JACK W. DAVIDSON AND DAVID B. WHALLEY
Department of Computer Science
University of Virginia
Charlottesville, VA 22903, U. S. A.

ABSTRACT

In an attempt to reduce the number of operand memory references, many RISC machines have thirty-two or more general-purpose registers (e.g., MIPS, ARM, Spectrum, 88K). Without special compiler optimizations, such as inlining or interprocedural register allocation, it is rare that a compiler will use a majority of these registers for a function. This paper explores the possibility of using some of these registers to hold branch target addresses and the corresponding instruction at each branch target. To evaluate the effectiveness of this scheme, two machines were designed and emulated. One machine had thirty-two general-purpose registers used for data references, while the other machine had sixteen data registers and sixteen registers used for branching. The results show that using registers for branching can effectively reduce the cost of transfers of control.

1. INTRODUCTION

Branch instructions cause many problems for machines. Branches occur frequently and thus a large percentage of a program's execution time is spent branching to different instructions. Branches can result in the pipeline having to be flushed, which reduces its effectiveness and makes pipelines with smaller number of stages more attractive. Furthermore, when the target of a branch instruction is not in the cache, additional delays are incurred as the instruction must be fetched from slower main memory.

This paper describes a technique that can eliminate much of the cost due to branches by using a new set of registers. A field is dedicated within each instruction to indicate a branch register that contains the address of the next instruction to be executed. Branch target address calculations are performed by instructions that are separate from the instruction causing the transfer of control. By exposing to the compiler the calculation of branch target addresses as separate instructions, the number of executed instructions can be reduced since the calculation of branch target addresses can often be moved out of loops. Much of the delay due to pipeline interlocks is eliminated since the instruction at a branch target is prefetched at the point the address is calculated. This prefetching of branch targets can also decrease the penalty for cache misses.

†This work was supported in part by the National Science Foundation under Grant CCR-8611653.

2. REVIEW

Due to the high cost of branches, there has been much work proposing and evaluating approaches to reduce the cost of these instructions. One scheme that has become popular with the advent of RISC machines is the delayed branch. While the machine is fetching the instruction at the branch target, the instruction after the branch is executed. For example, this scheme is used in the Stanford MIPS [HENN83] and Berkeley RISC [PATT82] machines. Problems with delayed branches include requiring the compiler or assembler to find an instruction to place after the branch and the cost of executing the branch itself.

Branch folding is another technique that reduces the cost of executing branches. This has been implemented in the CRISP architecture [DITZ87b]. Highly encoded instructions are decoded and placed into a wide instruction cache. Each instruction in this cache contains an address of the next instruction to be executed. Unconditional branches are folded into the preceding instruction since the program counter is assigned this new address for each instruction. Conditional branches are handled by having two potential addresses for the next instruction and by inspecting a static prediction bit and the condition code flag to determine which path to take. If the setting of the condition code (the compare) is spread far enough apart from the conditional branch, then the correct instruction can be fetched with no pipeline delay. Otherwise, if the incorrect path is chosen, then the pipeline must be flushed. The problems with this scheme include the complex hardware needed to implement the technique and the large size of the instruction cache since each decoded instruction is 192 bits in length.

An approach to reduce delays due to cache misses is to prefetch instructions into a buffer [RAU77]. The conditional branch instruction causes problems since either one of two target addresses could be used [RISE72]. One scheme involves prefetching instructions along both potential execution paths [LEE84]. This scheme requires more complicated hardware and also must deal with future conditional branch instructions. Other approaches use branch prediction in an attempt to choose the most likely branch target address [LEE84]. If the incorrect path is selected, then execution must be halted and the pipeline flushed.

3. THE BRANCH REGISTER APPROACH

As in Wilke's proposed microprogrammed control unit [WILK83] and the CRISP architecture [DITZ87a], every instruction in the branch register approach is a branch. Each instruction specifies the location of the next instruction to be executed. To accomplish this without greatly increasing the size of instructions, a field within all instructions specifies a register that contains the virtual address of the next instruction to execute.

Examples depicting instructions in this paper are represented using register transfer lists (RTLs). RTLs describe the effect of machine instructions and have the form of conventional expressions and assignments over the hardware's storage cells. For example, the RTL

```
r[3]=r[1]+r[2]; cc=r[1]+r[2]?0;
```

represents a register-to-register integer addition instruction on many machines. The first register transfer stores the sum of the two registers into a third register, while the second register transfer compares the sum of the two registers to set the condition codes. All register transfers within the same RTL represent operations that are performed in parallel.

For instructions specifying that the next instruction to be executed is the next sequential instruction, a branch register is referenced which contains the appropriate address. This register is, in effect, the program counter (PC). While an instruction is being fetched from the instruction cache, the PC is always incremented by the machine to point to the next sequential instruction. If every instruction is thirty-two bits wide, then this operation can always be performed in a uniform manner. Once an instruction has been fetched, the value of the branch register specified in the instruction is used as an address for the next instruction. At the point the PC is referenced, it will represent the address of the next sequential instruction. An example of this is shown in the RTL below, where b[0] (a branch register) has been predefined to be the PC.

```
r[1]=r[1]+1; b[0]=b[0]; /* go to next seq. inst.
```

Since references to b[0] do not change the address in b[0], subsequent RTLs do not show this default assignment.

If the next instruction to be executed is not the next sequential instruction, then code is generated to calculate and store the virtual address of that instruction in a different branch register and to reference that branch register in the current instruction. Storing the virtual address of a branch target instruction into a branch register also causes the address to be sent to the instruction cache to prefetch the instruction. The prefetched instruction will be stored into an instruction register that corresponds to the branch register receiving the virtual address. The address in the branch register will be incremented to point to the instruction after the branch target. The instruction register i[0], that corresponds to the branch register b[0], which is used as the program counter, is always loaded with the next sequential instruction.

To implement this technique, an organization shown in Figure 1 could be used. During the decode stage of the current instruction, the bit field specifying one of the branch registers is also used to determine which instruction register to use in the decode stage of the next instruction. When a branch register is referenced in an instruction to indicate that a transfer of control is to occur, the next instruction to execute is taken from the corresponding instruction register.

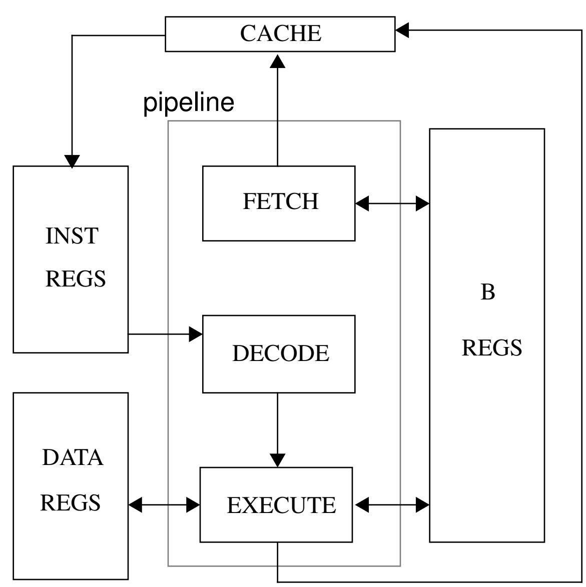

Figure 1: Dataflow for Branch Register Machine

4. CODE GENERATION

The following sections describe how code can be generated to accomplish various transfers of control using branch registers.

Calculating Branch Target Addresses

For all instructions where the next instruction to be executed is not the next sequential instruction, a different branch register from the PC must be specified and the virtual address it contains must have been previously calculated. If we assume a virtual address of thirty-two bits, an address cannot be referenced as a constant in a single instruction. Consequently, most instructions would use an offset from the PC to calculate branch addresses. The compiler knows the distance between the PC and the branch target if both are in the same routine. This is shown in the following RTLs:

```
b[1]=b[0]+(L2-L1);  /* store address of L2
L1:  ...
     ...
L2:  ...
```

For calls or branch targets that are known to be too far away, the calculation of the branch address requires two instructions. One part of the address is computed by the first instruction and then the other part in the second. Global addresses are calculated in this fashion for programs on the SPARC architecture [SUN87]. An address calculation requiring two instructions is illustrated by the following RTLs:

```
r[5]=HI(L1);        /* store high part of addr
b[1]=r[5]+LO(L1);   /* add low part of addr
    ...
L1:  r[0]=r[0]+1;   /* inst at branch target
    ...
```

Unconditional Branches

Unconditional branches are handled in the following manner. First, the virtual address of the branch target is calculated and stored in a branch register. To perform the transfer of control, this branch register is moved into the PC (b[0]), which causes the instruction at the target address to be decoded and executed next. While the instruction at the branch target is being decoded, the instruction sequentially following the branch target is fetched. An example of an unconditional branch is depicted in the following RTLs:

```
b[2]=b[0]+(L2-L1);       /* store addr of L2
L1:  ...
     ...
r[1]=r[1]+1; b[0]=b[2]; /*  next inst at L2
     ...
L2:  .
```

Conditional Branches

Conditional branches are generated by the following method. First, the virtual address of the branch target is calculated and stored in a branch register. At some point later, an instruction determines if the condition for the branch is true. Three branch registers are used in this instruction. One of two registers is assigned to the destination register depending upon the value of the condition. To more effectively encode this compare instruction, two of the three registers could be implied. For instance, the RTLs in the following example show how a typical conditional branch is handled. The destination branch register is b[7], which is by convention a trash branch register. The other implied branch register, the source register used when the condition is not true, is b[0], which represents the address of the instruction sequentially following the transfer of control instruction. An instruction following this conditional assignment would reference the destination branch register. This is illustrated below.

```
b[2]=b[0]+(L2-L1);       /* store addr of L2
L1:  ...
     ...
b[7]=r[5]<0->b[2]|b[0]; /* set branch register
r[1]=r[1]+1; b[0]=b[7]; /* jump to at addr in b[7]
      ...
L2:
```

Function Calls and Returns

Function calls and returns can also be implemented efficiently with this approach. Since the beginning of a function is often an unknown distance from the PC, its virtual address is calculated in two instructions and stored in a branch register. Then, an instruction at some point following this calculation would reference that branch register. To accomplish a return from a function, the address of the instruction following the call would be stored in an agreed-on branch register (for example b[7]). Every instruction that references a branch register that is not the program counter, b[0], would store the address of the next physical instruction into b[7]. If the called routine has any branches other than a return, then b[7] would need to be saved and restored. When a return to the caller is desired, the branch register is restored (if necessary) and referenced in an instruction. An example that illustrates a call and a return on this machine is given in the following RTLs.

```
r[2]=HI(_foo);      /* store high part of addr
b[3]=r[2]+LO(_foo); /* add low part of addr
   ...
r[0]=r[0]+1; b[0]=b[3]; b[7]=b[0];
/* next inst is first inst in foo
   ...
_foo:
   ...
r[0]= r[12]; b[0]=b[7];   /* return to caller
```

Indirect Jumps

For implementation of indirect jumps, the virtual address is loaded from memory into a branch register and then referenced in a subsequent instruction. The following RTLs illustrate how a switch statement might be implemented.

```
r[2]=r[2]<<2;        /* r2 is index in table
r[1]=HI(L01);        /* store high part of L01
r[1]=r[1]+LO(L01);   /* add low part of L01
b[3]=L[r[1]+r[2]];   /* load addr of switch case
   ...
r[0]=r[0]+1; b[0]=b[3];
/* next inst is at switch case
L01:  .long Ldst1    /* case label
      .long Ldst2    /* case label
      ...
      ...
```

5. COMPILER OPTIMIZATIONS

Initially, it may seem there is no advantage to the branch register approach. Indeed, it appears more expensive since an instruction is required to calculate the branch target address and a set of bits to specify a branch register is sacrificed from each instruction. However, one only needs to consider that the branch target address for unconditional jumps, conditional jumps, and calls are usually constants. Therefore, the assignment of these addresses to branch registers can be moved out of loops. Because transfers of control occur during execution of other instructions, the cost of these branches disappears after the first iteration of a loop.

Since there is a limited number of available branch registers, often every branch target cannot be allocated to a unique branch register. Therefore, the branch targets are first ordered by estimating the frequency of the execution of the branches to these targets. The estimated frequency of execution of each branch is used, rather than the execution of each branch target instruction, since it is the calculation of the virtual address used by each branch that has the potential for being moved out of loops. If there is more than one branch to the same branch target, then the frequency estimates of each of these branches are added together.

After calculating the estimated frequency of reference, the compiler attempts to move the calculation of the branch target with the highest estimated frequency to the preheader of the innermost loop in which the branch occurs. The preheader is the basic block that precedes the first basic block that is executed in the loop (or the head of the loop). At this point the compiler tries to allocate the calculation of the branch target address to a branch register. If the loop contains calls, then a non-scratch branch register must be used. If a branch register is only associated with branches in other loops that do not overlap with the execution of the current loop, then the branch target calculation for the branch in the current loop can be allocated to the same branch register. If the calculation for a branch target can be allocated to a branch register, then the calculation is associated with that branch register and the preheader of that loop (rather than the basic block containing the transfer of control) and the estimated frequency of the branch target is reduced to the frequency of the preheader of the loop. Next, the compiler attempts to move the calculation of the branch target with the currently highest frequency estimate out of the loop. This process continues until all branch target calculations have been moved out of loops or no more branch registers can be allocated.

To reduce further the number of instructions executed, the compiler attempts to replace no-operation (noop) instructions, that occur when no other instruction can be used at the point of a transfer of control, with branch target address calculations. These noop instructions are employed most often after compare instructions. Since there are no dependencies between branch target address calculations and other types of instructions that are not used for transfers of control, noop instructions can often be replaced.

Figures 2 through 4 illustrate these compiler optimizations. Figure 2 contains a C function. Figure 3 shows the RTLs produced for the C function for a conventional RISC machine with a delayed branch. Figure 4 shows the RTLs produced for the C function for a machine with branch registers. In order to make the RTLs easier to read, assignments to b[0] that are not transfers of control and updates to b[7] at instructions that are transfers of control are not shown. The machine with branch registers had one less instruction (eleven as opposed to fourteen) due to a noop being replaced with branch target address calculations. Since branch target address calculations were moved out of loops, there was only five instructions inside of the loop for the branch register machine as opposed to six for the machine with a delayed branch.

```
strlen(s)
char *s;
{
   int n = 0;

   if (s)
      for (; *s; s++)
         n++;
   return(n);
}
```

Figure 2: C function

```
     r[1]=L[r[31]+s.];   /* load s
     NZ=r[1]?0;          /* compare s to 0
     PC=NZ==0->L14;      /* delay cond. jump
     r[2]=0;             /* initialize n to 0
     PC=L17;             /* jmp to loop test
     NL=NL;              /* no-op required
 L18:r[2]=r[2]+1;        /* increment n
     r[1]=r[1]+1         /* increment s
 L17:r[0]=B[r[1]];       /* load character
     NZ=r[0]?0;          /* compare to zero
     PC=NZ!=0->L18;      /* delayed cond. jump
     NL=NL;              /* no-op required
 L14:PC=RT;              /* delayed return
     r[0]=r[2];          /* delay slot filled
```

Figure 3: RTLs for C Function with Delayed Branches

```
    b[1]=b[7];                /* save ret address
    b[7]=b[0]+(L14-L2);       /* compute exit addr
 L2:r[1]=L[r[15]+s.];         /* load s
    b[7]=r[1]==0->b[7]|b[0];  /* test cond.
    r[2]=0; b[0]=b[7];  /* initialize n and jmp
    b[7]=b[0]+(L17-L1); /* compute entry to loop
 L1:b[2]=b[0]+(L18-L18);b[0]=b[7]; /*compute loop
                    /* header and jump to entry
 L18:r[2]=r[2]+1              /* increment n
     r[1]=r[1]+1;             /* increment s
 L17:r[0]=B[r[1]];            /* load character
     b[7]=r[0]!=0->b[2]|b[0]; /* compute target
     NL=NL;b[0]=b[7];         /* jump
 L14:r[0]=r[2];b[0]=b[1];     /* return
```

Figure 4: RTLs for C Function with Branch Registers

6. REDUCTION OF PIPELINE DELAYS

Most pipeline delays due to branches on conventional RISC machines can be avoided using the branch register approach. For a three-stage pipleline, Figure 5 contrasts the pipeline delays for unconditional transfers of control on machines without a delayed branch, with a delayed branch, and with branch registers. The three stages in the pipeline in this figure are:

1. Fetch (F)
2. Decode (D)
3. Execute (E)

The branch target instruction cannot be fetched until its address has been calculated. For the first two machines, this occurs in the execute stage of the jump instruction. A conventional RISC machine without a delayed branch would have an *N*-1 delay in the pipeline for unconditional transfersof control where *N* is the number of stages in the pipeline. The next instruction for the machine with a delayed branch and the machine with branch registers represents the next sequential instruction following the jump instruction. Thus, a RISC machine with a delayed branch, where the branch is delayed for one instruction, would have an *N*-2 delay in the pipeline. Finding more than one useful instruction to place behind a delayed branch is difficult for most types of programs [MCFA86]. A jump

JUMP	F	D	E			
TARGET				F	D	E

(a) no delayed branch

JUMP	F	D	E			
NEXT		F	D	E		
TARGET				F	D	E

(b) with delayed branch

JUMP	F	D	E	
NEXT		F		
TARGET			D	E

(c) with branch registers

Figure 5: Pipeline Delays for Unconditional Transfers of Control

instruction for the machine with branch registers represents an instruction that references a branch register that is not the PC (b[0]). The branch register referenced is used during the decode stage of the jump instruction to determine which one of the set of instruction registers is to be input as the next instruction to be decoded. While the jump instruction is being decoded, the next sequential instruction is being fetched and loaded into i[0], the default instruction register. If b[0] had been referenced, then i[0] would be input to the decode stage. Since a different branch register is referenced for the jump instruction, its corresponding instruction register containing the branch target instruction would be input to the next decode stage. Thus, assuming that the branch target instruction has been prefetched and is available in the appropriate instruction register, the machine with branch registers would have no pipeline delay for unconditional transfers of control regardless of the number of stages in the pipeline.

The example in Figure 6 shows the actions taken by each stage in the pipeline for an unconditional transfer of control in the branch register machine, assuming that the jump sequentially follows the previously executed instruction. The subscript on the actions denotes the stage of the pipeline. During the first cycle, the jump instruction is fetched from memory and the PC is incremented to the next sequential instruction. In the second cycle, the jump instruction is decoded and the next sequential instruction after the jump is fetched from memory. In the third cycle, the jump instruction is executed, the prefetched branch target in i[4] is decoded, and the instruction sequentially following the branch target is fetched. Since the address in a branch register is incremented after being used to prefetch an instruction from the cache, the branch register contains the address of the instruction after the branch target.

r[1] = r[1] + 1; b[0] = b[4];

	JUMP	NEXT	TARGET	AFTER
(i[0] = M[b[0]];b[0] = b[0] + 4;)$_F$	F			
(DECODE = i[0];)$_D$ (i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	D	F		
(r[1] = r[1] + 1;)$_E$ (DECODE = i[4];)$_D$ (i[0] = M[b[4]];b[0] = b[4] + 4;)$_F$	E		D	F

Figure 6: Pipeline Actions for Unconditional Transfer of Control

Figure 7 contrasts the pipeline delays for conditional transfers of control for the same three types of machines. For unconditional transfers of control, the conventional RISC machine without a delayed branch would have a N-1 pipeline delay and the RISC machine with a delayed branch would have a N-2 pipeline delay for conditional transfers of control. The compare instruction for the machine with branch registers will assign one of two branch registers to a destination branch register depending upon the result of the condition in the compare. It will also make an assignment between the corresponding instruction registers. The conditional jump instruction is performed by the instruction following the compare instruction that references the destination branch register of the compare instruction. The branch register referenced is used during the decode stage of the conditional jump instruction to cause the corresponding instruction register to be input as the next instruction to be decoded. Therefore, the decode stage of the target instruction cannot be accomplished until the last stage of the compare instruction is finished. This results in an N-3 pipeline delay for conditional transfers of control for a machine with branch registers.

COMPARE	F	D	E				
JUMP		F	D	E			
TARGET					F	D	E

(a) no delayed branch

COMPARE	F	D	E				
JUMP		F	D	E			
NEXT			F	D	E		
TARGET					F	D	E

(b) with delayed branch

COMPARE	F	D	E		
JUMP		F	D	E	
NEXT			F		
TARGET				D	E

(c) with branch registers

Figure 7: Pipeline Delays for Conditional Transfers of Control

The example in Figure 8 shows the actions taken by each stage of the pipeline for a conditional transfer of control, assuming that the compare instruction sequentially follows the previously executed instruction. During the first cycle, the compare instruction is fetched from memory and the PC is incremented to the next sequential instruction. In the second cycle, the compare instruction is decoded and the jump instruction is fetched from memory. In the third cycle, the compare instruction is executed (resulting in assignments to both b[7] and i[7]), the jump instruction is decoded, and the instruction sequentially following the jump is fetched. If the condition of the compare is not true, then b[7] and i[7] receive the same values from the fetch operation. During the fourth cycle, the jump instruction is executed, either the target instruction or the next instruction after the jump is decoded, and the instruction after the instruction being decoded is fetched.

b[7] = r[5] < 0 -> b[3] | b[0];
r[1] = r[1] + 1; b[0] = b[7];

	COMPARE	JUMP	NEXT	TARGET	AFTER
(i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	F				
(DECODE = i[0];)$_D$ (i[0] = M[b[0]]; b[0] = b[0] + 4;)$_F$	D	F			
(b[7] = r[5] < 0 -> b[3] | b[0] + 4; i[7] = r[5] < 0 -> i[3] | M[b[0]];)$_E$ (DECODE=i[0];)$_D$(i[0]=M[b[0]];b[0]=b[0]+4;)$_F$	E	D	F		
(r[1] = r[1] + 1;)$_E$ (DECODE = i[7];)$_D$ (i[0] = M[b[7]]; b[0] = b[7] + 4;)$_F$		E		D	F

Figure 8: Pipeline Actions for Conditional Transfer of Control

To avoid pipeline delays, even when the branch target instruction is in the cache, the branch target address must be calculated early enough to be prefetched from the cache and placed in the instruction register before the target instruction is to be input to the decode stage. Assuming there is a one cycle delay between the point that the address is sent to the cache at the end of the execute stage and the instruction is loaded into the instruction register, this would require that the branch target address be calculated at least two instructions previous to the instruction with the transfer of control when the number of stages in the pipeline is three. This is shown in Figure 9.

ADDR CALC	F	D	E	X		
INST		F	D	E		
JUMP			F	D	E	
NEXT				F		
TARGET					D	E

Figure 9: Prefetching to Avoid Pipeline Delays

7. EXPERIMENTAL EVALUATION

In an attempt to reduce the number of operand memory references, many RISC machines have thirty-two or more general-purpose registers (e.g. MIPS-X, ARM, Spectrum). Without special compiler optimizations, such as inlining [SCHE77] or interprocedural register allocation [WALL86], it is infrequent that a compiler can make effective use of even a majority of these registers for a function. In a previous study [DAVI89a], we calculated the number of data memory references that have the potential for being removed by using registers. We found that 98.5% could be removed by using only sixteen data registers. In order to evaluate the effectiveness of the branch register approach, two machines were designed and emulated. *ease*, an environment which allows the fast construction and emulation of proposed architectures [DAVI89b], was used to simulate both machines. Detailed measurements from the emulation of real programs on a proposed architecture are captured in this environment. This is accomplished by creating a compiler for the proposed machine, collecting information about instructions during the compilation, inserting code to count the number of times sets of basic blocks are executed, and generating assembly code for an existing host machine from the RTLs of the program on the proposed machine. Appendix I lists the set of test programs used for this experiment.

The first machine served as a baseline to measure the effectiveness of the second machine. The baseline machine was designed to have a simple RISC-like architecture. Features of this machine include:

- 32-bit fixed-length instructions
- load and store architecture
- delayed branches
- 32 general-purpose data registers
- 32 floating-point registers
- three-address instructions

Figure 10 shows the instruction formats used in the baseline machine.

Format 1 (branch with disp, i = 0):

opcode	cond	i	displacement
6	4	1	21

Format 1 (branch indirect, i = 1):

opcode	cond	i	ignored	rs1
6	4	1	16	5

Format 2 (sethi, j ignored):

opcode	rd	j	immediate
6	5	2	19

Format 3 (Remaining instructions, i = 0):

opcode	rd	rs1	i	immediate
6	5	5	1	15

Format 3 (Remaining instructions, i = 1):

opcode	rd	rs1	i	ignored	rs2
6	5	5	1	10	5

Figure 10: Instruction Formats for the Baseline Machine

The second machine was a modification of the first to handle branches by using branch registers. Features of the branch register machine that differ from the baseline machine include:

- only 16 general-purpose data registers
- only 16 floating-point registers
- 8 branch registers
- 8 instruction registers
- no branch instructions
- a compare instruction with an assignment
- an instruction to calculate branch target addresses
- smaller range of available constants in some instructions

If one ignores floating-point registers, there are approximately the same number of registers on each machine. Figure 11 shows the instruction formats used in the branch register machine. Since the only differences between the baseline machine and the branch register machine are the instructions to use branch registers as opposed to branches, the fewer number of data registers that can be referenced, and the smaller range of constants available, the reports generated by this environment can accurately show the impact of using registers for branches.

Format 1 (cmp with immed, i = 0):

opcode	cond	bs1	rs1	i	immediate	br
6	4	3	4	1	11	3

Format 1 (cmp with reg, i = 1):

opcode	cond	bs1	rs1	i	ignored	rs2	br
6	4	3	4	1	7	4	3

Format 2 (sethi, inst addr calc):

opcode	rd	immediate	br
6	4	19	3

Format 3 (Remaining instructions, i = 0):

opcode	rd	rs1	i	immediate	br
6	4	4	1	14	3

Format 3 (Remaining instructions, i = 1):

opcode	rd	rs1	i	ignored	rs2	br
6	4	4	1	10	4	3

Figure 11: Instruction Formats for the Branch Register Machine

The branch register machine executed 6.8% fewer instructions and yet performed 2.0% additional data memory references as compared to the baseline machine. The ratio of fewer instructions executed to additional data references for the branch register machine was 10 to 1. Approximately 14% of the instructions executed on the baseline machine were transfers of control. The reduction in the number of instructions executed was mostly due to moving branch target address calculations out of loops. The ratio of transfers of control executed to branch target address calculations was over 2 to 1. Another factor was replacing 36% (2.6 million) of the noops in delay slots of branches in the baseline machine with branch target address calculations at points of transfers of control in the branch register machine. There were also additional instructions executed on the branch register machine to save and restore branch registers. The additional data references on the branch register machine were due to both fewer variables being allocated to registers and saves and restores of branch registers. Table I shows the results from running the test set through both machines.

Machine	Millions of instructions executed	Millions of data references
baseline	183.04	61.99
branch register	170.75	63.22
diff	-12.29	+1.23

Table I: Dynamic Measurements from the Two Machines

By prefetching branch target instructions at the point the branch target address is calculated, delays in the pipeline can be decreased. In the baseline machine, there were 7.95 million unconditional transfers of control and 17.69 million conditional transfers of control. Assuming a pipeline of three stages, not uncommon for RISC machines [GIMA87], then each branch on the baseline machine would require at least a one-stage delay. Also assuming that each instruction can execute in one machine cycle, and no other pipeline delays except for transfers of control, then the test set would require about 208.83 million cycles to be executed on the baseline machine. As shown previously in Figures 5 and 7, the branch register machine would require no delay for both unconditional and conditional branches in a three stage pipeline assuming that the branch target instruction has been prefetched. As shown in Figure 9, the branch target address must be calculated at least two instructions before a transfer of control to avoid pipeline delays even with a cache hit. We estimate that only 13.86% of the transfers of control that were executed would result in a pipeline delay. Thus, the branch register machine would require about 22.09 million (10.6%) fewer cycles to be executed. There would be greater savings for machines having pipelines with more stages. For instance, we estimate that the branch register machine would require about 30.04 million (12.8%) fewer cycles to be executed due to fewer delays in the pipeline alone assuming a pipeline with four stages.

8. HARDWARE CONSIDERATIONS

An instruction cache typically reduces the number of memory references by exploiting the principles of spatial and temporal locality. However, when a particular main memory line is referenced for the first time, the instructions in that line must be brought into the cache and these misses will cause delays. When an assignment is made to a branch register, the value being assigned is the address of an instruction that eventually will likely be brought into the instruction cache.

To take advantage of this knowledge, each assignment to a branch register has the side effect of directing the instruction cache to prefetch the line associated with the instruction address. Prefetch requests could be performed efficiently with an instruction cache that would allow reading a line from main memory at the same time as requests for instruction words from the CPU that are cache hits are honored. This could be accomplished by setting a busy bit in the line of the cache that is being read from memory at the beginning of a prefetch request and setting it to not busy after the prefetch has completed. To handle prefetch requests would require a queuing mechanism with the size of the queue equal to the number of available branch registers. A queue would allow the cache to give priority to cache misses for sequential fetches over prefetch requests which do not require the execution of the program to wait. Directing the instruction cache to bring in instructions before they are used will not decrease the number of cache misses. It will, however, decrease or eliminate the delay of loading the instruction into the cache when it is needed to be fetched and executed.

The machine must determine if an instruction has been brought into an instruction register and thus is ready to be decoded after the corresponding branch register is referenced in the preceding instruction. This can be accomplished by using a flag register that contains a set of bits that correspond to the set of instruction registers. The appropriate bit could be cleared when the request is sent to the cache and set when the instruction is fetched from the cache. Note that this would require the compiler to ensure that branch target addresses are always calculated before the branch register is referenced.

9. FUTURE WORK

There are several interesting areas involving the use of branch registers that remain to be explored. The best cache organization to be used with branch registers needs to be investigated. An associativity of at least two would ensure that a branch target could be prefetched without displacing the current instructions that are being executed. A larger number of words in a cache line may be appropriate in order to less often have cache misses of sequential instructions while instructions at a branch target are being loaded from memory into the instruction cache. Another feature of the cache organization to investigate is the total number of words in the cache. Since instructions to calculate branch target addresses can be moved out of loops, the number of instructions in loops will be fewer. This may improve cache performance in machines with small on-chip caches.

The exact placement of the branch target address calculation can affect performance. The beginning of the function could be aligned on a cache line boundary and the compiler would have information about the structure of the cache. This information would include

- the cache line size
- the number of cache lines in each set
- the number of cache sets in the cache

Using this information the compiler could attempt to place the calculation where there would be less potential conflict between cache misses for sequential instructions and cache misses for prefetched branch targets. By attempting to place these calculations at the beginning of a cache line, the potential for conflict would be reduced.

Prefetching branch targets may result in some instructions being brought into the cache that are not used (cache pollution). Since most branches tend to be taken [LEE84], we have assumed that this penalty would not be significant. By estimating the number of cycles required to execute programs (which includes cache delays) on the branch register machine and the baseline machine, the performance penalty due to cache pollution of unused prefetched branch targets could be determined.

Other code generation strategies could be investigated. For instance, if a fast compare instruction could be used to test the condition during the decode stage [MCFA86], then the compare instruction could update the program counter directly. A bit may be used in the compare instruction to indicate whether to squash [MCFA86] the following instruction depending upon the result of the comparison. Eight branch registers and eight instruction registers were used in the experiment. The available number of these registers and the corresponding changes in the instruction formats could be varied to determine the most cost effective combination.

10. CONCLUSIONS

Using branch registers to accomplish transfers of control has been shown to be potentially effective. By moving the calculation of branch target addresses out of loops, the cost of performing branches inside of loops can disappear and result in fewer executed instructions. By prefetching the branch target instruction when the branch target address is calculated, branch target instructions can be inserted into the pipeline with fewer delays. By moving the assignment of branch registers away from the use of the branch register, delays due to cache misses of branch targets may be decreased. The performance of a small instruction cache, such as the cache for the CRISP architecture [DITZ87a], could also be enhanced since the number of instructions in loops will be fewer. Enhancing the effectiveness of the code can be accomplished with conventional optimizations of code motion and common subexpression elimination. A machine with branch registers should also be inexpensive to construct since the hardware would be comparable to a conventional RISC machine.

11. ACKNOWLEDGEMENTS

The authors wish to thank Anne Holler and Ron Williams for providing many helpful suggestions.

12. REFERENCES

[DAVI89a] J. W. Davidson and D. B. Whalley, *Methods for Saving and Restoring Register Values across Function Calls*, Tech. Rep. 89-11, University of Virginia, November 1989.

[DAVI89b] J. W. Davidson and D. B. Whalley, *Ease: An Environment for Architecture Study and Experimentation*, Tech. Rep. 89-08, University of Virginia, September 1989.

[DITZ87a] D. R. Ditzel and H. R. McLellan, *The Hardware Architecture of the CRISP Microprocessor*, Proceedings of the 14th Annual Symposium on Computer Architecture, Pittsburg, PA, June 1987, 309-319.

[DITZ87b] D. R. Ditzel and H. R. McLellan, *Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero*, Proceedings of the 14th Annual Symposium on Computer Architecture, Pittsburg, PA, June 1987, 2-9.

[GIMA87] C. E. Gimarc and V. Milutinovic, *A Survey of RISC Processors and Computers of the Mid-1980s*, IEEE Computer **20**,9 (September 1987), 59-69.

[HENN83] J. Hennessy and T. Gross, *Postpass Code Optimization of Pipeline Constraints*, ACM Transactions on Programming Languages and Systems **5**,3 (July 1983), 422-448.

[LEE84] J. K. F. Lee and A. J. Smith, *Branch Prediction Strategies and Branch Target Buffer Design*, IEEE Computer **17**,1 (January 1984), 6-22.

[MCFA86] S. McFarling and J. Hennessy, *Reducing the Cost of Branches*, Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986, 396-403.

[PATT82] D. A. Patterson and C. H. Sequin, *A VLSI RISC*, IEEE Computer **15**,9 (September 1982), 8-21.

[RAU77] B. R. Rau and G. E. Rossman, *The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units*, Proceedings of the 4th Annual Symposium on Computer Architecture, Silver Spring, MD, March 1977, 80-89.

[RISE72] E. M. Riseman and C. C. Foster, *The Inhibition of Potential Parallelism by Conditional Jumps*, IEEE Transactions on Computers, **21**,12 (December 1972), 1405-1411.

[SCHE77] R. W. Scheifler, *An Analysis of Inline Substitution for a Structured Programming Language*, Communications of the ACM **20**,9 (September 1977), 647-654.

[SUN87] *The SPARC Architecture Manual*, Sun Microsystems, Mountain View, CA, 1987.

[WALL86] D. W. Wall, *Global Register Allocation at Link Time*, Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction, Palo Alto, CA, June 1986, 264-275.

[WILK83] M. Wilkes and J. Stringer, *Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer*, Proceedings of the Cambridge Philosophical Society, Cambridge, England, April 1983.

13. APPENDIX I: TEST PROGRAMS

Class	Name	Description or Emphasis
Utilities	cal	Calendar Generator
	cb	C Program Beautifer
	compact	File Compression
	diff	File differences
	grep	Search for Pattern
	nroff	Text formatter
	od	Octal dump
	sed	Stream editor
	sort	Sort or merge files
	spline	Interpolate Curve
	tr	Translate characters
	wc	Word count
Benchmarks	dhrystone	Synthetic Benchmark
	matmult	Matrix multiplication
	puzzle	Recursion, Arrays
	sieve	Iteration
	whetstone	Floating-Point arithmetic
User code	mincost	VLSI circuit partitioning
	vpcc	Very Portable C compiler