Chapter 10

Understanding Branches and Calls

This chapter discusses branches and calls and how they differ.

Topics

10.1	Software Branching	PP:10-2
10.2	Understanding the Difference Between Branches and Calls	PP:10-3
10.3	Branch Delay Slots	PP:10-4
10.4	Subroutine Call and Return Sequence	PP:10-8
10.5	Absolute Versus Relative Branching	PP:10-11
10.6	Conditional Branches and Calls	PP:10-13
10.7	Branch Tables	PP:10-15
10.8	Two-Input ALU Operations With pc Destination	PP:10-17

10.1 Software Branching

Software branching is performed on the PP by specifying one of the pc register write codes (br or call) as the destination of an ALU operation, load, or move. ALU operations with the pc register as the destination can involve any Boolean or arithmetic combination of two inputs. However, branches are typically specified in one of the following ways:

- An absolute branch

```
br = address
```

- A relative branch

```
br = ipe + offset
```

- A load from the branch table

```
br = *BranchTbl.element
```

- A move from a register

```
br = iprs
```

The following sections discuss issues related to software branching.

10.2 Understanding the Difference Between Branches and Calls

The PP provides two different register codes for the pc register:

- Register code **call** is used for making subroutine calls (a branch with a subsequent return).
- Register code **br** is used for branches that do not have an associated return.

When **call** is specified as the destination of an operation, two actions occur:

- The subroutine start address is written to the pc register, and
- The subroutine return address is written to the iprs (instruction pointer, return from subroutine) register.

The standard call and return sequence using iprs is detailed in Section 10.4.

When **br** is specified as the destination of an operation, only the write to the pc register occurs; the iprs register is not modified. In all other regards (for example, delay slots, etc.), calls and branches behave the same. In the following sections, discussions concerning branches also apply to calls unless a specific distinction is noted.

10.3 Branch Delay Slots

After an instruction that specifies a branch or a call is fetched, two more instructions are fetched (this is normal pipeline behavior) before the branch is executed. These two instructions are referred to as **delay slot** instructions. The PP (unlike the MVP's master processor) does not provide an annul option for these branch delay slot instructions.

Typically, the branch delay slot instructions will be the next two instructions in memory. However, this may not be the case if the branch occupies a branch delay slot for a previous branch or is a loop end address.

Example 10–1 illustrates the delay slot instructions for the branch to LABEL, assuming that the branch does not occur in the delay slot of a branch and that neither the branch nor its first delay slot instruction are a loop end. Thus, the next two instructions (<Instruction1> and <Instruction2>) are the delay slot instructions.

Branch delay slot instructions can perform useful operations. When the branch must be taken before more useful work can be done, you can fill the two delay slots with nop instructions.

Example 10–1. Branch Delay Slot Instructions

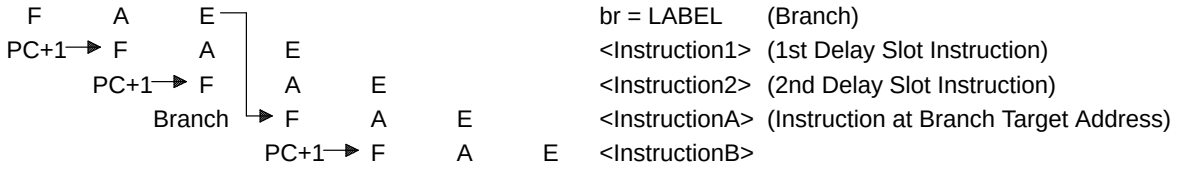
```
   br = LABEL
   <Instruction1>  ;  Delay Slot1 Instruction
   <Instruction2>  ;  Delay Slot2 Instruction
   .
   .
   .

LABEL: <InstructionA>
       <InstructionB>
   .
   .
```

The program flow for Example 10–1 is illustrated in Figure 10–1. This figure shows the overlap between pipeline stages (FAE) of the branch and its delay slot instructions. Progression from left to right corresponds to pipeline stage advances. Progression from top to bottom corresponds to the instruction sequence that is fetched and executed. PC+1→F indicates that the fetched instruction is determined by an increment of the program counter from the previous instruction. This is the case for the two branch delay slot instructions.

Direct writes to the pc register are prioritized over the program counter increment. Thus, when the branch is executed, the pc is set to the target address for the branch.

The arrow from the execute stage of the branch instruction to the next fetch stage indicates that the next instruction is determined by the branch (instead of the program counter increment). Thus, <InstructionA> is the next instruction fetched after the branch is executed (that is, after the two branch delay slot instructions). After the branch occurs, the program counter resumes incrementing, and <InstructionB> is fetched.

Figure 10–1. Software Branch Program Flow



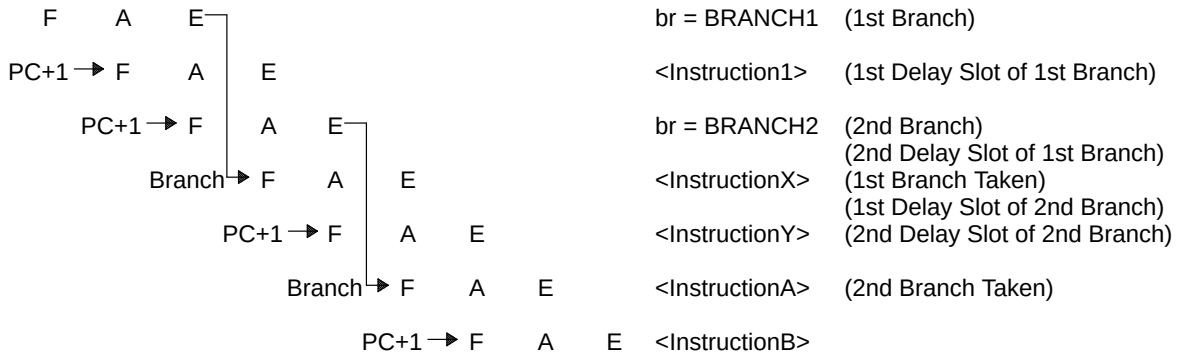
If a branch is specified in the first or second delay slot instruction of a branch, the first branch is taken and 1 or 2 instructions, respectively, are fetched at the target address for the first branch before the second branch target instruction is fetched. In Example 10–2, a branch is specified in the second delay slot instruction of a branch.

Example 10–2. A Branch in the Second Delay Slot of a Branch

```
   br = BRANCH1
   <Instruction1>
   br = BRANCH2
   <Instruction2>
   <Instruction3>
          .
          .
          .

BRANCH1:
   <InstructionX>
   <InstructionY>
          .
          .
          .

BRANCH2:
   <InstructionA>
   <InstructionB>
          .
          .
          .
```

The program flow for Example 10–2 is illustrated in Figure 10–2. Note that the delay slot instructions for a branch are not always the next two instructions in memory. As shown in Figure 10–2, the delay slot instructions for the second branch are the first two instructions at the target address of the first branch (<InstructionX> and <InstructionY>)—not the next two instructions in memory after the instruction specifying the second branch (<Instruction2> and <Instruction3>).

Figure 10–2. Branch in Delay Slot of a Branch



10.4 Subroutine Call and Return Sequence

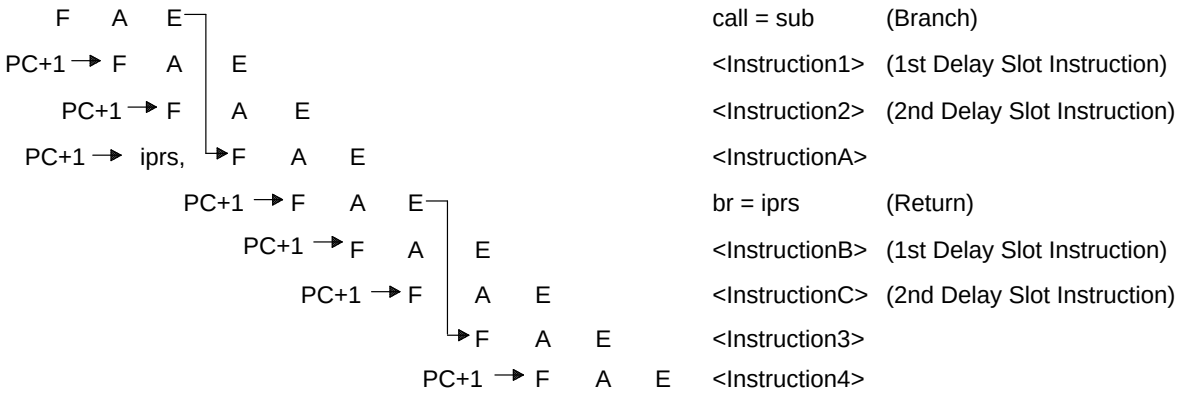
A subroutine call is performed by writing the subroutine address (typically specified by a label) to the call register code for the pc. When a call is executed, not only is the subroutine address written to the pc register, but also the return from subroutine address is written to the iprs register. The return address is determined by PC+1 (the instruction sequentially following the second delay slot instruction) at the time the branch is executed. The return address written to iprs is PC+1, even if the next instruction fetched (had the branch not occurred) would have been different (for example, the second delay slot instruction was a loop end).

Example 10–3 illustrates a call and return sequence. The resulting program flow shown in Figure 10–3 assumes that the call is not in a branch delay slot and that neither the call nor its first delay slot instruction are a loop end. At the time the call is executed, iprs is loaded with the current PC+1. This is the address for <Instruction3>. The return from subroutine is performed by moving the return address contained in the iprs register to br (the branch register code for the pc).

Example 10–3. Call and Return Sequence

```
      call = sub          ;  pc = subroutine address
                          ;     & iprs = return address
      <Instruction1>      ;  1st delay slot instruction.
      <Instruction2>      ;  2nd delay slot instruction.
back: <Instruction3>      ;  Instruction fetched upon
                          ;     returning from the
                          ;     subroutine.
      <Instruction4>      ;
         .
         .

sub:  <InstructionA>      ;  Start of Subroutine.
      br = iprs           ;  Return to back
      <InstructionB>      ;  Return delay slot1
      <InstructionC>      ;  Return delay slot2
```

Figure 10–3. Software Call and Return Program Flow



As shown in Example 10–4, a typical use for call delay slot instructions is to to push registers onto the stack. Likewise, return delay slot instructions can be used to pop these registers off the stack.

Example 10–4. Use of Delay Slot Instructions

```
        call = sub      ;  pc = subroutine address
                        ;  & iprs = return address
        *--sp = d7      ;  1st delay slot instruction.
        *--sp = d6      ;  2nd delay slot instruction.
back:   <Instruction3>  ;  Instruction fetched upon
                        ;  returning from the subroutine.
        <Instruction4>  ;
           .
           .

sub:    <InstructionA>  ;  Start of Subroutine.
           .
           .
           .
        br = iprs       ; Return to back
        d6 = *sp++      ; Return delay slot1
        d7 = *sp++      ; Return delay slot2
```

10.4.1 Nested Calls

When you implement a call within a call, you must save the iprs value of the first call before making the second call; this avoids overwriting the first call's return address.

In Example 10–5, the following events occur to ensure that the iprs value is saved:

1) **main** calls **sub1**, causing the return address to be written to the iprs register.
2) **sub1**, in turn, pushes iprs onto the stack before performing a call to **sub2**.
3) When the call to **sub2** is executed, the contents of iprs are overwritten with the return address from **sub2**.
4) The return from **sub2** is performed by moving iprs to br (the branch register code for the pc register).
5) The return from **sub1** to **main** is performed by popping the stacked return address directly to br. It is not necessary to pop this value to iprs first.

Example 10–5. Saving the iprs Value Before a Nested Call

```
main:     .
          .
          call = sub1     ;  Return address for
                          ;  sub1 written to iprs.
          <Instruction1>  ;  1st Delay Slot Instruction
          <Instruction2>  ;  2nd Delay Slot Instruction
sub1_ret:
          <Instruction3>
             .
             .
sub1:  <InstructionA>
             .
             .
          *--sp = iprs    ;  Push return from sub1
                          ;  address onto stack.
          call = sub2     ;  Return address for sub2
                          ;  overwrites return address
                          ;  for sub1 in iprs.
          <InstructionB>  ;  1st Delay Slot Instruction
          <InstructionC>  ;  2nd Delay Slot Instruction
sub2_ret:
          <InstructionD>
             .
             .
          br = *sp++      ;  Pop sub1 return.
          <InstructionE>  ;  1st Delay Slot Instruction
          <InstructionF>  ;  2nd Delay Slot Instruction

sub2:     .
          .
          br = iprs       ;  Return to sub2.
          <InstructionY>  ;  1st Delay Slot Instruction
          <InstructionZ>  ;  2nd Delay Slot Instruction
```

10.5 Absolute Versus Relative Branching

Branches (and calls) can be specified in either absolute or relative fashion, as described in the following sections.

10.5.1 Absolute Branch

To perform an absolute branch, write the desired new PC address to the pc register. This is typically done by using a label that is resolved at link time.

Example 10–6. Absolute Call

```
main:     .
          .
       call = sublabel
       <Instruction1>     ;  delay slot1
       <Instruction2>     ;  delay slot2
return:   .
          .

sublabel: .
          .
       br = iprs
       <InstructionA>     ;  delay slot1
       <InstructionB>     ;  delay slot2
```

10.5.2 Relative Branches

Relative branches specify an offset from the current program counter location. On the PP, relative branches should be specified relative to the ipe (instruction pointer, execute stage) register. This is because the ipe register value at the time a call or branch is executed corresponds to the value of the PC at the time the instruction specifying the branch was fetched. The pc register cannot be reliably used to call or branch from because a call, branch, loop, or interrupt may have modified the value of the PC to something other than the desired relative branch base.

Note:

The three LSBs of any immediate or register value added to the ipe register should always be 0 to prevent undesirable carries from the G and L control bits (located in the two MSBs of the pc register) into the PC value.

In Example 10–7, a relative subroutine call is performed. The $ symbol is equivalent to a label for the instruction being executed. Assuming **sublabel** is within the current section of the current file, the assembler can evaluate the expression (**sublabel – $**) and produce the correct displacement between the address of the call instruction being executed and the address of the desired subroutine. This value is then added at runtime to the ipe register, which contains the address of the call instruction being executed.

When the branch offset is +/– 3 instructions, using a relative branch makes it possible to specify the branch with a 5-bit immediate. This, in turn, lets you specify a local transfer in parallel with the branch.

Example 10–7. Relative Call

```
main:       .
            .
         call = ipe + (sublabel – $)
         <Instruction1>        ;  delay slot1
         <Instruction2>        ;  delay slot2
return:
         <Instruction3>
            .
            .

sublabel:   .
            .
         br = iprs
         <InstructionA>        ;  delay slot1
         <InstructionB>        ;  delay slot2
```

10.6 Conditional Branches and Calls

Branches and calls can be performed conditionally just like any other ALU operation, move, or load. Thus, branches can occur conditionally, as specified by any of the 16 supported condition codes (see Table 8–36, *Condition Codes*).

Example 10–8 illustrates how a conditional branch is used to poll the comm register's Q bit until it is 0. As long as the Q bit is nonzero (indicating that a packet transfer request is queued), a branch back to the start of the poll sequence (poll) is performed. When the Q bit becomes 0, the conditional branch is not taken, and therefore, the poll loop is exited. At that point, the comm register's P bit can be set to 1 to submit a new packet transfer request (see subsection 12.2.3, *Step 3: Issuing a Packet Transfer Request to the TC*, for more information about setting the P bit).

ALU operations that write to the pc register do not modify status. This special case for ALU status saving prevents status from being corrupted by branches or calls to routines that require the status information. Also, different branch target addresses can be specified on the basis of different conditions.

Example 10–8. Conditional Branch

```
       a15 = comm & 0x1\\29  ;  Test Q bit.

poll:  br =[nz] poll         ;  Continue polling Q as long
                             ;   as Q=1 (Zero status bit
                             ;   = 0).
       nop

       a15 = comm & 0x1\\29  ;  Test Q bit.

       comm =comm | 0x1\\28  ;  Submit PTR (when poll loop
                             ;   is exited).
```

The code in Example 10–8 can be compacted by moving the instruction that sets the P bit into the polling loop. In Example 10–9, the first delay slot instruction of the conditional branch conditionally sets the P bit according to the opposite condition of the branch (that is, if the Q bit is 0). Because the branch operation does not modify status, the zero status bit is correctly maintained with respect to the previous test of the Q bit (see subsection 12.3.1.1, *Q Bit Polling*, for more information about polling the Q bit).

The two conditional operations in Example 10–9 are thus mutually exclusive; either the branch is performed, indicating that a packet transfer request is still queued, or the conditional branch is not taken and a new packet transfer request is submitted.

Example 10–9. Conditional Branch

```
      a15 = comm & 0x1\\29     ;  Test Q bit.

poll: br =[nz] ipe             ;  Continue polling Q as
                               ;   long as Q = 1 (Zero
                               ;   status bit = 0).
      comm =[z] comm | 0x1\\28 ;  Submit PTR if Q=0 (Zero
                               ;   status bit = 1).
      a15 = comm & 0x1\\29     ;  Test Q bit.
```

10.7 Branch Tables

Branches specified by an ALU operation require a 32-bit immediate source operand (unless the branch is +/– 3 instructions). This precludes any parallel transfers from being specified in the instruction. If a branch to a certain location is performed frequently, such as a conditional branch inside a loop, an efficient technique is to determine the address for the desired branch and store the 32-bit address in a branch table located in the shared RAMs.

In Example 10–10, a branch table is set up in the PP's local parameter RAM. Two operations are required initially to set up each entry in the branch table. Then, each time a branch or call is required to one of target addresses specified in the branch table, it can be performed by a load from the branch table. This allows a data unit operation and/or a local transfer to be specified in parallel with the branch.

Notes:

1) Loads to the pc register must be word data size; halfword and byte loads to the pc register are not allowed.

2) Loads to the pc register from memory should not use an unscaled immediate-offset. The return from interrupt (reti) operations use unscaled immediate offsets for loads to the pc register to indicate special modification of the G and L control bits.

Example 10–10. Branch Table

```
Branch_Tbl_Offset: .set   0x400
Ga_Branch_Tbl:     .set   a10

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Branch Table Set-Up
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   d7 = Sub1              ;  Determine 1st branch address.
   *(Ga_Branch_Tbl = pba + [Branch_Tbl_Offset]) = d7
                          ;  Set branch table base address
                          ;   and store first entry.
   d7 = Branch1
                          ;  Determine 2nd branch address.
   *(Ga_Branch_Tbl + [1]) = d7
                          ;  Store 2nd branch table entry.
      .
      .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; End of branch table set-up
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      .
      .
   <Data Unit Op>
   ||br = *(Ga_Branch_Tbl + [1])  ;  Branch to Branch1
                                  ;   using a load from
                                  ;   the Branch Table.
      <||Local Transfer>

   <Instruction>                  ;  1st delay slot
                                  ;   instruction
   <Instruction>                  ;  2nd delay slot
                                  ;   instruction
      .
      .

Branch1:
      .
      .

Sub1:
      .
      .
```

10.8 Two-Input ALU Operations With pc Destination

The pc register can be specified as the destination of any arithmetic or Boolean ALU operation involving two inputs (base set ALU operation class 7 with the B port input ignored). Note that the inputs cannot be routed through the barrel rotator, mask generator, expander, or bit-detection logic.

Table 10–1 lists the two-input arithmetic and two-input Boolean ALU operations that can written to the pc register (either **br** for branches or **call** for subroutine calls saving a return address).

Table 10–1. Supported ALU Operations to the pc

Two-Input Arithmetic

```
br = src1 – src2          ; allows br = ipe – offset
br = src1 + src2          ; allows br = ipe + offset
```

Two-Input Boolean

```
br = src1
br = ~src1
br = src2                 ; allows br = address
br = ~src2


br = src1 | src2
br = ~src1 | src2
br = src1 | ~src2
br = ~src1 | ~src2


br = src1 & src2
br = ~src1 & src2
br = src1 & ~src2
br = ~src1 & ~src2


br = src1 ^ src2
br = src1 ^ ~src2
```

Note: src1 can be any register. src2 can be any D register or an immediate.