Chapter 11

Understanding the Loop Control Logic

The program flow control unit includes three hardware loop controllers that support up to three simultaneous loops. Since each PP instruction performs so many actions in parallel, key loops often require very few instructions. Thus, overhead is greatly reduced by using hardware looping.

Topics

11.1 Looping

The program flow control unit contains three comparators and associated loop control hardware that support up to three levels of zero-overhead looping. Looping is considered zero-overhead because it occurs in parallel with the normal single-cycle instruction execution so that no looping-related instructions are required within the loop. A loop count decrement and conditional branch to the loop start address occur automatically in hardware each time a loop end is matched. Thus, repetitive operations can be coded in tight loops with no associated loop control overhead.

The loop control logic operates in the fetch pipeline stage. When a loop end address is matched, the corresponding loop start address is loaded to the PC, and that instruction is fetched on the cycle that immediately follows (with zero delay). In contrast, software branches occur in the execute stage of the pipeline. Thus, two delay-slot instructions are fetched between an instruction specifying a software branch and the write to the pc register of the branch target address. Hardware looping greatly reduces the amount of software branching required and therefore makes code more efficient.

In addition to supporting three levels of nested zero-overhead loops, the PP loop control logic supports a variety of features including multiple loop ends associated with the same loop count, hardware branching, and conditional hardware branching.

11.2 Basic Loop Control

This section introduces the basic loop-control mechanisms and describes how they are used to provide looping capability.

11.2.1 Loop Control Registers

Looping on the PP is controlled primarily by the values in the following registers, which are defined further in Chapter 7, *Summary of PP Registers*.

- **Loop End Registers: le2, le1, le0**

  Each loop end register points to the last instruction in a loop. During each instruction fetch, the loop end register is compared to the pc to detect the loop end. When the loop end register matches the pc (when the last instruction in a loop is being fetched), the corresponding loop hardware action is invoked.

- **Loop Start Registers: ls2, ls1, ls0**

  Each loop start register points to the first instruction in a loop. The loop start register is copied over the pc when the looping hardware wants to branch back to the beginning of the loop. This is done in the cycle during which the pc matches the associated loop end register.

- **Loop Counter Registers: lc2, lc1, lc0**

  Each loop count register contains a count of the number of times to copy the associated loop start register over to the pc (the number of times to branch to the starting instruction of the loop). The loop count is decremented each time the associated loop end register matches the pc and the associated loop start register is copied over the pc. Since the loop count is the number of times to branch, it is the number of times to do the loop, minus one.

❑ Loop Reload Registers: lr2, lr1, lr0

Each loop reload register contains an initialization value for the associated loop count register. This reinitialization takes place after the last time through the loop. When the associated loop count register is zero at the end of the loop, instead of decrementing the loop counter, the loop counter is loaded from the reload register. This prepares the loop counter for the next time the loop is entered.

Note:

Writes to a loop reload register automatically write to the loop count register as well, but the converse is not true. This mechanism assists in faster loop initialization.

❑ Loop Control Register: lctl

The loop control register contains three similar 4-bit fields, each field relating to a loop end register. One bit in each field enables the comparator to the address in the loop end register. The other three bits designate the associated loop counter, which can be none, lc0, or the loop counter with the same number as the loop end register.

11.2.2 Loop End Comparison

Each time an instruction is fetched, the PC is compared with the address contained in each loop end register that is enabled (as discussed in subsection 11.3.1.3). When a match is found, the instruction address in the associated loop start register (ls2 for le2, ls1 for le1, and ls0 for le0) is loaded into the PC. Also, the count in the loop counter register, and assigned in the loop control (lctl) register (as discussed in subsection 11.3.1.3) is decremented.

11.2.3 Loop Count Decrement/Reload

A loop counter is decremented each time that a loop end associated with it is encountered, unless the loop count is 0. lc2 and lc1 can be associated only with le2 and le1, respectively. lc0 can be associated with any, some, or all of the loop end registers.

When the PC matches an enabled loop end address and the associated loop count is zero (before decrement), the PC is not loaded with the loop start address and the loop count is not decremented. Instead, the PC increments by 1, thus exiting the loop, and the associated loop counter is loaded with the contents of the loop reload register.

11.3 Setting Up the Loop Control Registers

Looping requires setting up the loop control registers. These registers can be written individually (for general case loops) or by using one of two shortform loop initialization register codes (for special case loops).

11.3.1 General Loop Control Initialization

Loop control values can be initialized by individually loading all of the loop control registers, as described in this section. Since the loop registers are loaded by software in the execute pipeline stage and used by the loop control hardware in the fetch stage, there should always be at least two instructions between the loading of any loop register and the loop end address at which that register will be used. (It is, however, permissible to modify other loop registers not related to that specific loop end. For example, le2 can be modified in the two instructions before le1 is encountered, provided that the loop end address associated with le2 occurs at least two instructions later).

As shown in Example 11–1, initializing a loop by individually setting the loop control registers requires the following steps:

1) Set the loop end register to the address of the last instruction in the loop. This determines where the loop back occurs.

2) Set the loop start register to the address of the first instruction in the loop. This determines the address that is looped back to.

3) Set the loop reload register to the number of loop backs to be performed each time the loop is entered (the number of times to do the loop, minus one). The same value is also written automatically to the loop counter register.

4) Optionally set the loop counter register (if the desired initial loop counter value is different than the loop reload register value) to the number of loop backs for the first time the loop is entered.

5) Set bits in the lctl register to enable the loop end, and designate the associated loop counter. A loop end should not be enabled (in lctl) until after the correct loop end address has been set up.

Example 11–1. Loop Set Up

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Loop Set Up
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   le0 = Loop_End0       ; Set loop end address.
   ls0 = Loop_Start0     ; Set loop start address.
   lr0 = 99              ; Perform loop instructions
                         ; 99+1 times (loop back to
                         ; ls0 99 times).
   lctl = 0x9            ; Enable le0 and associate
                         ;  it with lc0.

   <Instruction1>        ; 1st Delay Slot Instruction
   <Instruction2>        ; 2nd Delay Slot Instruction
             .
             .
             .
Loop_Start0: .
             .
             .
Loop_End0:   .
```

11.3.1.1 Initializing Loop End and Start Addresses

Loop end and start addresses can be made code-position independent by computing them as offsets with respect to the ipe register, as shown in Example 11–2 (for a general discussion of PC-relative address computations, see subsection 10.5.2, *Relative Branches*).

Example 11–2. Code-Position Independent Loop End Specification

```
    le0 = ipe + (loopend_label – $)
        .
        .
        .
loopend_label:
        .
        .
```

11.3.1.2 Initializing Loop Reload and Loop Counter Values

When you specify a write to a loop reload register (lr*n*), a write to the associated loop counter register (lc*n*) is also performed automatically. If the desired lc*n* value is the same as the lrn value, then only lr*n* needs to be loaded by software. **If you want different values in lc*n* and lr*n*, you must set lrn first.** Otherwise, writing the lrn value will overwrite the desired lc*n* value.

The value in the loop counter register determines the number of times that the hardware branch to the loop start address is performed. Hence, the value written to the loop reload and/or loop counter register should be one less than the number of times the instructions in the loop should be performed. For example, if a loop operates on one pixel per pass through the loop and it needs to be performed on 100 pixels, the loop counter should be set initially to 99. Since the loop counter is decremented each time a loop back is performed, the loop counter constantly reflects the number of remaining passes through the loop (not including the current one).

11.3.1.3 Enabling Loop Ends and Designating the Associated Loop Counter

The loop control register (lctl) shown in Figure 11–1 enables the comparison between the PC and each individual loop end and associates a loop counter with each loop end. lctl contains three similar 4-bit fields, each field relating to a loop end register. The three enable (E) bits activate the corresponding loop end register comparators. Looping is completely disabled by setting these three bits to 0. When a PP is reset, all lctl E bits are cleared. An lctl E bit should not be set to 1 until the corresponding loop end register has been set.

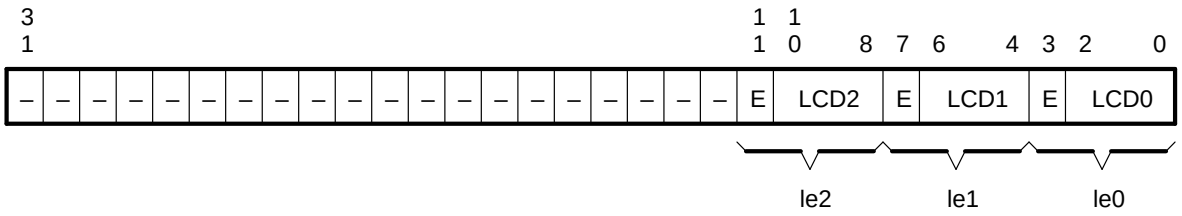
The three LCD*n* fields, each consisting of three bits, designate which loop counter should be decremented when the loop end register is successfully compared against the program counter. The only permitted loop counter assignments are:

- **le0**: lc0 or none
- **le1**: lc1, lc0, or none
- **le2**: lc2, lc0, or none

All other combinations are reserved. If lc0 is designated for two or three loop end registers, multiple ends to the same loop can be implemented (see subsection 11.5.1, *Multiple Ends Associated WIth the Same Loop Counter*).

If the LCD*n* field is 0, no loop counter is checked, decremented, or reloaded, and the loop start copy always occurs. Thus, the loop control logic can be used to perform hardware branches (subsection 11.5.2, *Hardware Branching*) without having to maintain a nonzero loop count.

Figure 11–1. lctl Register



E — Loop end enable

LCD*n* — Loop counter designator for loop end

000 — None. Always load loop start into PC

001 — lc0

010 — lc1

011 — lc2

1XX — Reserved

11.3.2 Shortform Loop Control Initialization

The overhead associated with individually setting all of the loop control registers to initialize a loop that is executed many times is almost insignificant. However, the overhead becomes significant for small short-lived loops. To support faster loop initialization, two shortform methods are provided for two special cases of loop setup performed immediately before entry to the loop.

11.3.2.1 Shortform for a Single-Instruction Loop

Special register codes (lrse2–lrse0) support fast initialization of single-instruction loops. Writing a loop count value to an lrse*n* (where *n* = the loop register number) register performs the following:

- Sets lr*n* (and lc*n*) to the desired count.
- Sets ls*n* and le*n* to the address of the instruction sequentially following the second delay-slot instruction (that is, PC+1).

  Note that a single instruction loop is possible with le*n* equal to ls*n*.
- Sets the corresponding E bit in lctl, enabling le*n*. Sets the loop counter designator to lc*n*.

These operations all occur in a single cycle during the execute pipeline stage. There are two delay-slot instructions between fetching the instruction that writes to lrse*n* and the start/end address of the loop. The loop start and loop end addresses are calculated in relation to the PC value of the second delay-slot instruction. If one of the two instructions fetched immediately **before** the shortform loop initialization operation is a branch, the loop start and loop end are calculated in relation to the PC value after the branch is executed. Shortform loop initialization should therefore be used with care within the delay-slot instructions of a branch.

In Example 11–3, assuming that the instruction that writes to lrse*n* is not in the delay slot of a branch and that neither it nor its first delay-slot instruction (<Instruction1>) occurs at a loop end, the loop start and loop end addresses for the loop are set to the third subsequent instruction in memory (<Instruction3>).

The code shown in Example 11–3 sets up the loop control logic associated with le2 to perform <Instruction3> four times. The value 3 written to lrse2 indicates the number of times to loop back to the start of the loop, <Instruction3>, as illustrated in Figure 11–2. The fourth time <Instruction3> is fetched, lc2 is 0 and therefore, the loop back does not occur. Instead, the PC is incremented by one, le2 is reloaded with 3 from lr2, and <Instruction4> becomes the next instruction fetched.

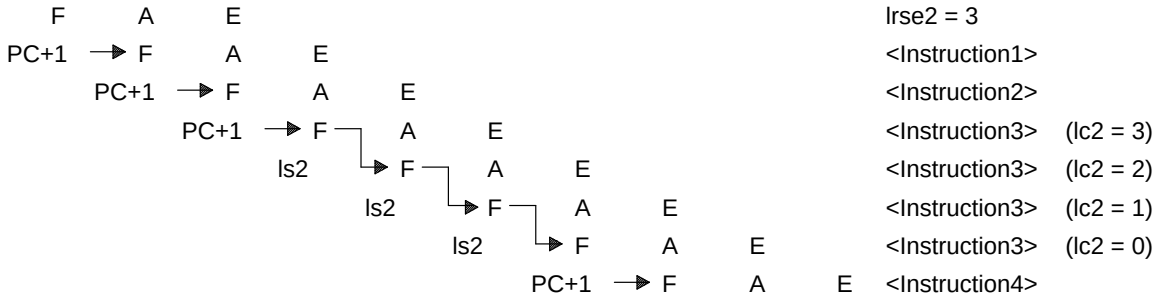
Note:

When editing code between a shortform loop initialization operation and the desired loop start address, be careful to preserve exactly two delay-slot instructions.

Example 11–3. Single-Instruction Loop Shortform Initialization

```
      lrse2 = 3          ; Perform instructions in the
                         ; loop 3+1 times.
      <Instruction1>     ; Delay Slot1 Instruction
      <Instruction2>     ; Delay Slot2 Instruction
loop2:<Instruction3>     ; Single-Instruction Loop.
      <Instruction4>     ;
```

Figure 11–2. Program Flow for Example 11–3



11.3.2.2 Shortform Initialization for Multi-Instruction Loop

Special register codes (lrs2–lrs0) support fast initialization of multi-instruction loops. Fast multi-instruction loop set-up is identical to single-instruction loop set-up except that the associated loop end register is not modified when writing to the lrs*n* register; it must be initialized in a separate instruction. Writing a loop count value to an lrs*n* (where *n* = the loop register number) register has this effect:

1) Sets lr*n* (and lc*n*) to the desired count.

2) Sets ls*n* (loops start address) to point to the instruction sequentially following the second delay-slot instruction.

3) Sets the corresponding E bit in lctl, thus enabling le*n*. Sets the loop counter designator to lc*n*.

These operations all occur in a single cycle during the execute pipeline stage. There are two delay-slot instructions between the fetching of the instruction that writes to lrs*n* and the start of the loop. The loop start address is calculated from the PC value of the second delay-slot instruction. If one of the two instructions fetched immediately **before** the shortform loop initialization operation is a branch, the loop start is calculated in relation to the PC value after the branch is executed. Shortform loop initialization should therefore be used with care within the delay-slot instructions of a branch.

In Example 11–4, assuming that the instruction that writes to lrs*n* is not in the delay slot of a branch and that neither it nor its first delay-slot instruction (<Instruction1>) occurs at a loop end; the loop start address for the loop is set to the third subsequent instruction in memory (<Instruction3>). The instruction sequence shown in Example 11–4 sets up the multi-instruction loop starting at <Instruction3> to be performed four times.

Example 11–4. Multi-Instruction Loop Shortform Initialization

```
                le1 = loop_end1
                lrs1 = 3            ; Perform instructions in
                                    ;  the loop 3+1 times.
                <Instruction1>      ; Delay Slot1 Instruction
                <Instruction2>      ; Delay Slot2 Instruction
loop_start1:    <Instruction3>      ; 1st Instruction in
                                    ;  loop.
                   .
                   .
loop_end1:      <InstructionN>      ; Last Instruction in
                                    ;  loop.
```

Note that for the multi-instruction loop shortform initialization, the loop end address is loaded separately. It is recommended for safe coding that loop ends never be enabled (by setting lctl directly or with lrs*n*) before the associated loop end register has been set, so no undesired loop-related branches are taken.

Note:

When editing code between a shortform loop initialization operation and the desired loop start address, be careful to preserve exactly two delay-slot instructions.

11.4 Nested Loops

The PP supports up to three levels of nested loops in hardware. Nested loops are frequently useful in image processing. Example 11–5 shows how to initialize three levels of nested looping. Note that shortform loop initialization is used for all three loops, even though there is more than one delay slot between the shortform register assignment and the corresponding loop start. This is done by assigning the desired address to the loop start register before the loop end register is ever encountered. In this example, the outer loop could perform the operations that set up an 8×8 block of data to be processed.

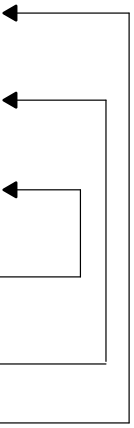
This may include

- Verifying that the block of data to be processed by the middle and inner loops is on-chip by checking that a packet transfer request has completed.
- Submitting a packet transfer request to:
  - Transfer on-chip the block of data that will be processed by the next pass through the nested loops.
  - Transfer off-chip the block of data that was processed by the previous pass through the nested loops.

The middle loop could be the row loop, plus handle special boundary conditions for the first and last pixel in each of the eight rows within an 8×8 block, as well as the address increment from the last pixel within the 8×8 block for one row of data to the first pixel within the block on the next row. Finally, the inner loop would perform the operations that are repeated for each of the six inner pixels within the same row of the block of data.

Example 11–5. Three Levels of Nested Looping

```
loop_setup:
   le1 = middle_end
   lrs1 = 7
   ls1 = middle      ; Loop back to the start of the
                     ; middle loop 7 times per pass of
                     ; the outer loop thus processing
                     ; each of the 8 rows in an 8x8
                     ; block.

   le2 = inner_end
   lrs2 = 5
   ls2 = inner       ; Loop back to the start of the
                     ; inner loop 5 times per pass
                     ; of the middle loop thus
                     ; processing the 6 inner pixels
                     ; in a row of the 8x8 block of
                     ; of data.
   le0 = outer_end
   lrs0 = 5          ; Loop back to the start of the
                     ; outer loop 5 times thus
                     ; processing 6 8x8 blocks.

<delay slot instruction 1>
<delay slot instruction 2>

outer:      .        ; Start of outer loop.
            .
            .
middle:     .        ; Start of middle loop.
            .
            .
inner:      .        ; Start of inner loop.
            .
            .
inner_end:  .        ; Inner loop end.
            .
            .
middle_end: .        ; Middle loop end.
            .
outer_end:  .        ; Outer loop end.
            .
            .
```



11.4.1 Loop End Priority

Loop ends (and/or loop starts) can be coincident. When two or three loops end at the same address, the loop end registers are prioritized from le2 (highest) to le0 (lowest). Thus, le2, le1, and le0 correspond to the inner, middle, and outer loops, respectively. The loop start register associated with the highest priority matched loop end register with a nonzero loop count is loaded into the PC, and the associated loop counter is decremented. Also, the zero-value loop counters of higher priority, PC-matching loop end registers are reloaded from their associated loop reload registers.

In Example 11–6, the outer, middle, and inner loops have the same loop end address, and lctl = 0xBA9. Due to the prioritization scheme, the following events occur when the PC equals the loop end address (that is, the instruction at the loop end address is fetched):

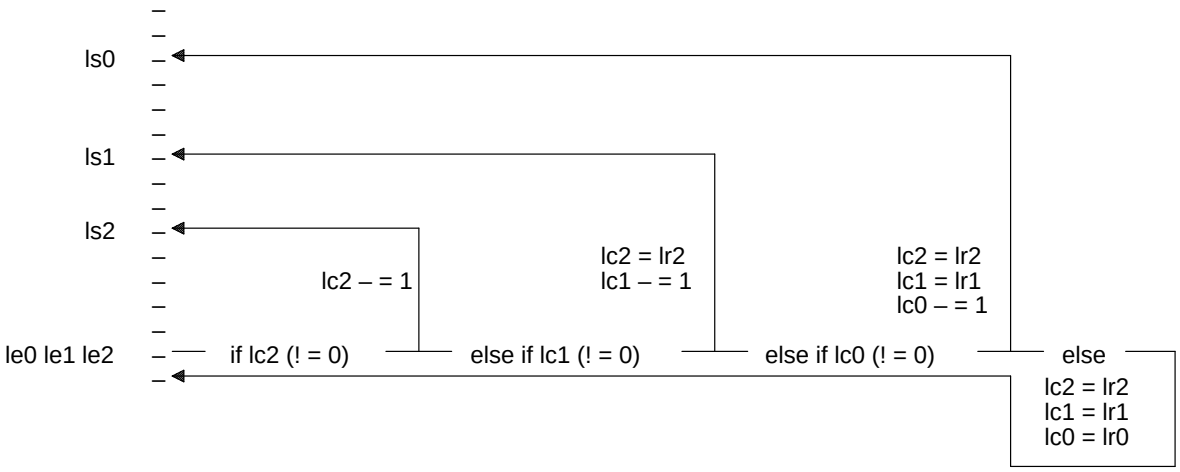
- When lc2 ≠ 0,
  - PC is loaded with ls2 (repeat inner loop)
  - lc2 is decremented
- When lc2 = 0, and lc1 ≠ 0,
  - PC is loaded with ls1 (repeat middle loop)
  - lc2 = lr2
  - lc1 is decremented
- When lc2 = 0, lc1 = 0, and lc0 ≠ 0,
  - PC is loaded with ls0 (repeat outer loop)
  - lc2 = lr2
  - lc1 = lr1
  - lc0 is decremented
- When lc2 = 0, lc1 = 0, and lc0 = 0,
  - PC is incremented by 1 (loop is exited)
  - lc2 = lr2
  - lc1 = lr1
  - lc0 = lr0

Example 11–6. Three Levels of Nested Looping With Same Loop End

```
loop_setup:
   le1 = middle_end
   lrs1 = 7
   ls1 = middle      ; Loop back to the start of the
                     ; middle loop 7 times per pass of
                     ; the outer loop thus processing
                     ; each of the 8 rows in an 8x8
                     ; block.

   le2 = inner_end
   lrs2 = 5
   ls2 = inner       ; Loop back to the start of the
                     ; inner loop 5 times per pass
                     ; of the middle loop thus
                     ; processing the 6 inner pixels
                     ; in a row of the 8x8 block of
                     ; of data.
   le0 = outer_end
   lrs0 = 5          ; Loop back to the start of the
                     ; outer loop 5 times thus
                     ; processing 6 8x8 blocks.

<delay slot instruction 1>
<delay slot instruction 2>

outer:         .     ; Start of outer loop.
               .
               .
middle:        .     ; Start of middle loop.
               .
               .
inner:         .     ; Start of inner loop.
               .
inner_end:
middle_end:
outer_end:     .     ; All 3 loops have same end.
               .
               .
```

The sequence of program flow for Example 11–6 is illustrated in Figure 11–3.

Figure 11–3. Three Loops Ending at Same Address



11.5 Uses of Loop Control Flexibility

The PP loop control logic flexibility can be used for a variety of functions, such as speeding up critical tight loops. This section discusses multiple loop ends associated with the same loop counter, hardware branching, conditional software branching and conditional hardware branching.

11.5.1 Multiple Ends Associated With the Same Loop Counter

The loop control logic allows multiple loop ends to be associated with the same loop counter. This is useful for data-dependent algorithms.

Example 11–7 illustrates how two loop end addresses associated with the same loop counter can be used for performing variable length code (VLC) table look-ups (TLUs). The first step is to perform a TLU on the next eight bits of data. The returned table value indicates whether these eight bits contain a full VLC.

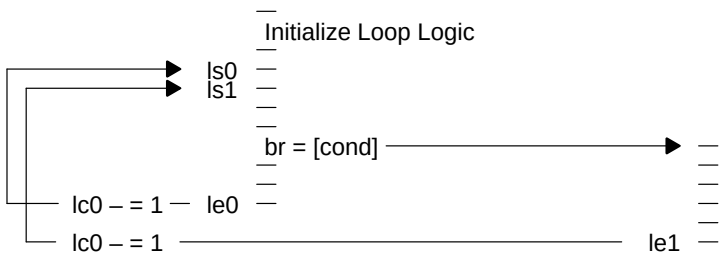
If they do, the routine progresses to le0, at which point lc0 is decremented and the PC is set to ls0. If a full VLC is not contained in the first eight bits, a branch is taken to a routine that performs a second TLU on the next eight bits. The last instruction of the routine that performs the second TLU is set up as le1, which decrements lc0.

The flexibility of the loop control logic is shown in Example 11–7 where the loop back from le1 returns to a different address than le0. With this looping set up, each time a VLC is successfully identified (after either one or two TLUs), lc0 is decremented Figure 11–4).

Example 11–7. Bitstream Decoding With Two Ends to Same Loop

```
      le1 = Second_TLU_end
      ls1 = Second_TLU_ret
      le0 = First_TLU_end
      lrs0 = 63             ; Number of VLCs to look up is
                            ;  64.
      lctl = 0x99           ; Enable le1 and le0. Associate
                            ;  both with lc0.
      <Instruction1>        ; Delay Slot2 instruction

First_TLU:
      <Instruction2>
Second_TLU_ret:
      <Instruction3>
      <Instruction4>
      <Instruction5>
      <Instruction6>
      br =[cond] Second_TLU ; Branch to second TLU routine.
      <Instruction8>
      <Instruction9>
First_TLU_end:
      <Instruction10>

          .
          .
          .

Second_TLU:
      <InstructionA>
          .
          .
Second_TLU_end:
      <InstructionZ>
```

Figure 11–4. Two Ends to Same Loop



11.5.2 Hardware Branching

The PP's loop-control logic makes is possible to perform hardware branching. When a loop end is encountered and the corresponding loop counter is nonzero, the PC essentially branches to the address specified in the associated loop start register. In addition, the lctl register can be set up so that the hardware branch is always performed and no loop counter is decremented when the loop end is encountered.

Once the loop end register has been loaded and enabled, a branch occurs each time the instruction at the associated address is fetched; no delays occur between the loop end (pseudobranch request) and the actual branch. Additionally, no instruction is required to request the branch after the looping hardware is configured. Thus, the advantage of hardware branching over software branching is that it requires essentially zero overhead (other than the initial set up of the loop control registers).

Often, hardware branching is associated with a second hardware branch that essentially performs a return. A hardware branch followed by a hardware return typically occurs within an outer loop that initializes the generic aspects of an operation.

For example, most of the coding of various PIXBLT functions is the same, with the exception of the specific pixel-processing inner loop (which is typically two or three instructions). By using hardware branching to enter and return from the pixel-processing subroutine, you can keep code size to a minimum because no delay slots are incurred and no branch instructions are required.

Example 11–8 shows how to set up a hardware branch and hardware return for a generic PIXBLT outer loop. Specific PIXBLT operators such as AND are coded in subroutines.

Hardware branching lets you avoid the instructions required for software branching and their associated delay-slot instructions. Instead, roughly nine instructions are required initially to set up the three sets of loop control registers. If the pixel processing operation is performed on more than a couple of pixels, the loop control initialization (that is required only once) becomes relatively inexpensive, compared to software branching.

Example 11–8. PIXBLT Example With Fast Subroutine Call

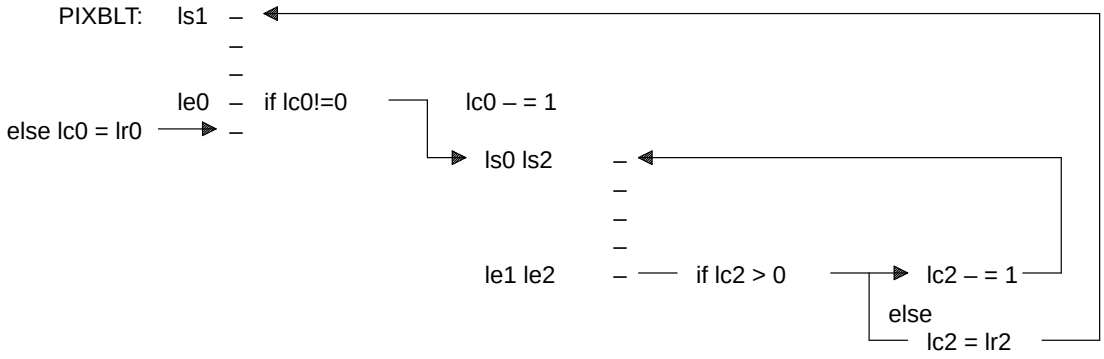
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Set up of loop control registers for a hardware branch
;; and hardware return for desired pixel processing
;; subroutine.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   le0   =  subcall
   ls0   =  xor
   lc0   =  height – 1
   le1   =  xore
   ls1   =  subret
   le2   =  xore
   ls2   =  xor
   lc2   =  width/size – 1
   lctl  =  0xB89

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; PIXBLT Outer Loop.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PIXBLT:
subret:                     ; Zero-delay return address.
      .
      .
subcall:                    ; Zero–delay branch occurs.
   d3 = *a8++               ; Load src2.
      ||d2 = *a0++          ; Load src1.
      .
      .
      .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; AND Pixel Processing Subroutine
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
and:
   d4 = d2 & d3             ; src1 AND src2
    ||d3 = *a8++            ; Load src2
    ||d2 = *a0++            ; Load src1
ande:
   *a9++ = d4               ; Store processed pixel.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; XOR Pixel Processing Subroutine
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
xor:
   d4 =d2 ^ d3              ; src1 XOR src2
    ||d3 = *a8++            ; Load src2
    ||d2 = *a0++            ; Load src1
xore:
   *a9++ = d4               ; Store processed pixel.
      .
      .
      .
```

In Example 11–8, le0 is set up to cause a hardware branch to a pixel-processing subroutine. This hardware branch occurs as long as lc0 is nonzero. ls0 is set to the start address of the desired pixel-processing subroutine.

Once in the pixel-processing subroutine, le2 (which is set to the last instruction of the subroutine) causes the program to loop in the subroutine until lc2 is zero. le1, which is lower priority than le2, is also set to the last instruction of the subroutine. When lc2 is 0 and the last instruction of the subroutine is fetched, a hardware branch back to the PIXBLT outer loop occurs. ls1 is set to the beginning of the PIXBLT outer loop. No loop counters are associated with le1. This allows the hardware return to be implemented without having to maintain a nonzero loop count. The associated loop start register will always be loaded; therefore, the hardware return will always occur.

Figure 11–5 illustrates the program flow for Example 11–8.

Figure 11–5. Program Flow for PIXBLT Fast Subroutine Call



11.5.3 Conditional Software Branching

Software branches (and calls) have priority over looping when they are coincident (the loop end address is the second delay-slot instruction of a branch). Loop counters are not decremented or reloaded during a cycle in which the PC is loaded by a software branch (the loop logic is effectively disabled for that cycle). This allows a software exit from a loop upon a condition becoming true.

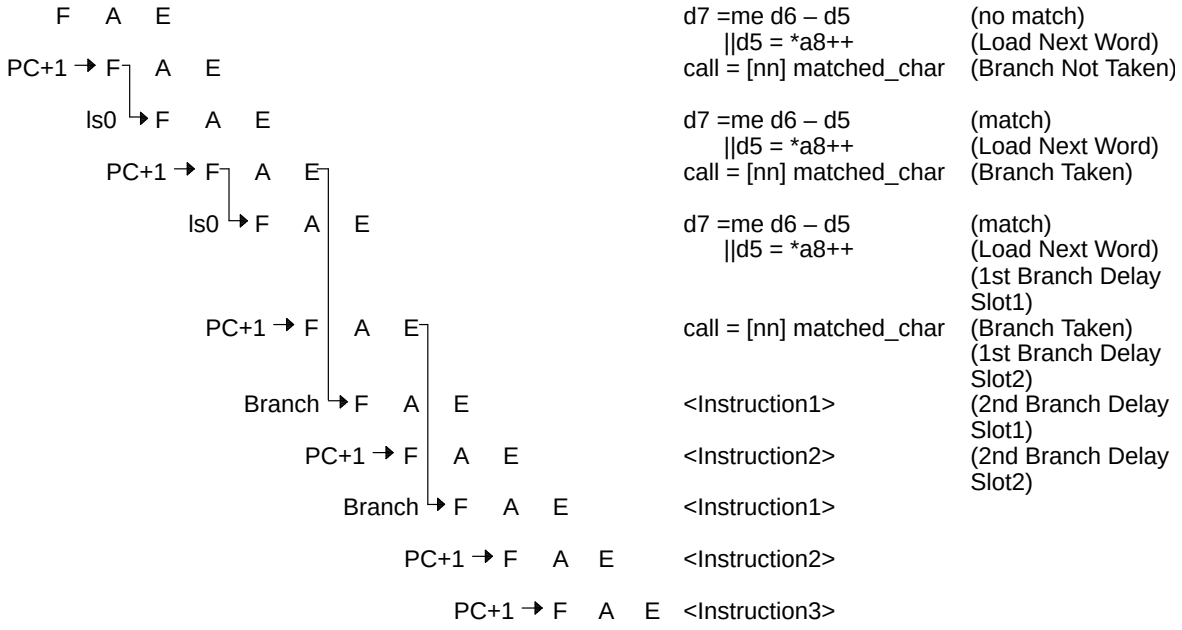
Conditional software branches out of single- or two-instruction loops should be used carefully because the conditional branch occurs in its own delay slot. Example 11–9 uses a conditional branch to exit a loop if a character is matched. Four characters are checked with a multiple-byte compare that saves the split-ALU zero comparisons to the mf register. For multiple arithmetic, the negative status bit records the NOR of the flags saved to mf. Therefore, a single conditional branch can be used to exit the loop if any of the four bytes matches the desired character.

Example 11–9. Character Search Using Conditional Software Branch Exit

```
sr = 4                  ; Set Asize for byte multiple
                        ; arithmetic and MSS for setting mf
                        ; bits according to zero compares.
loop0: d7 =me d6–d5 ;Test 4 bytes for character match.
   ||d5 = *a8++         ; Load next word of 4 bytes.
loop0e:  call =[nn] matched_char
                        ; Exit if character was matched.
       .
       .
       .

matched_char:
   <Instruction1>
   <Instruction2>
   <Instruction3>
       .
       .
```

Figure 11–6 illustrates that in Example 11–9, the second delay-slot instruction of the conditional branch coincides with the loop end. Also, the conditional branch is in its own second delay slot; therefore, if the condition remains true (after the first time it is met), the branch may actually occur twice.

Figure 11–6. Conditional Software Branch in its Own Delay Slot



11.5.4 Conditional Hardware Branching

Conditional hardware branching can be implemented by loading a value into the designated loop counter at least two instructions before a loop end address is encountered.

When the instruction at a loop end address is fetched, the associated loop start address is loaded into the PC by the loop control logic only if the designated loop counter value is nonzero. Thus, conditional hardware branching can be implemented by writing to a loop counter either with a nonzero value (in which case the loop is taken when the instruction at the loop end address is fetched) or with zero (in which case the loop is not taken when the loop end address is fetched).

Note that software writes to a loop counter dominate over the loop count decrement associated with a loop end address.

The character search example shown in the previous section can be optimized by using conditional hardware branching to exit the loop when a character match is found instead of by using conditional software branching.

In Example 11–10, the character search example is implemented with a single-instruction loop. In each cycle, four bytes are compared against the character that is to be matched. The zero compares for each of the split-ALU results are written to the mf register.

If no match is found, all of the zero compares will be false, so a zero value is stored in mf. If a match is found, at least one of the zero compares will be true, so a nonzero value will be stored in mf. In the next cycle, this value will be written to lc2 (the loop counter associated with le2).

Two delay-slot instructions are fetched before the execute stage of the write to lc2 with the nonzero value occurs; then the character search inner loop is exited.

Example 11–10. Character Search Using Conditional Hardware Branch

```
      sr = 4          ;  Set Asize for byte multiple
                      ;     arithmetic and MSS for setting mf
                      ;     bits according to zero compares.
      d5 = 0xF0F0F0F0 ;  Put in reg. Packed Character
                      ;     that is being searched.
      le2 = loop2_end
                      ;  Set address for conditional
                      ;     hardware branch out of the
                      ;     character search loop.
      lrs2 = 0        ;  Zero loop count (lc2) associated
                      ;     with conditional branch and
                      ;     enable le2. Note that ls2
                      ;     still needs to be set to the
                      ;     correct target address for
                      ;     the conditional hardware branch.
      lrse1 = 511     ;  Set loop1 to search up
                      ;     to 512 words (2k bytes).
      ls2 = matched_char
                      ;  Correct the target address for
                      ;     the conditional hardware branch.
      mf = 0          ;  prezero mf
         ||d6 = *(a0=dba)
                      ;  Load first word to be searched
                      ;     after setting the pointer.

loop1_start:
loop1_end:
loop2_end:
      d7 =me d6 – d5  ;  Compare 4 bytes to the
                      ;     character being searched for.
         ||lc2 = mf   ;  Set lc2 to value of the multiple
                      ;     zero flags for previous compare.
                      ;     If match is found (mf has
                      ;     set zero flags), then after 2
                      ;     delay–slot instructions, the pc
                      ;     will be set to ls2 thus exiting
                      ;     the search inner loop.
         ||d6 = *a0++ ;  Load next word.

         .
         .
         .

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; After the character search loop is exited by the
;; conditional hardware branch, the data pointer needs
;; to be decremented by 4 words to reload the first word
;; that matched the character to determine which byte
;; matched.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

matched_char:
      <Instruction1>
      <Instruction2>
         .
         .
```

Figure 11–7 shows that after a word with a matching character is loaded, the character search loop is executed four more times:

- Two times because of latency. The mf move into lc2 occurs two cycles after the data is loaded.
- Two pipeline delay-slot instructions. Two additional instructions have already been fetched when the mf move to lc2 is executed.

To identify the exact byte that matched, the pointer must be decremented by four, and the compare must be re-executed.

Figure 11–7. Conditional Hardware Branch out of a Single-Instruction Loop