Chapter 5

Understanding the Program Flow Control Unit

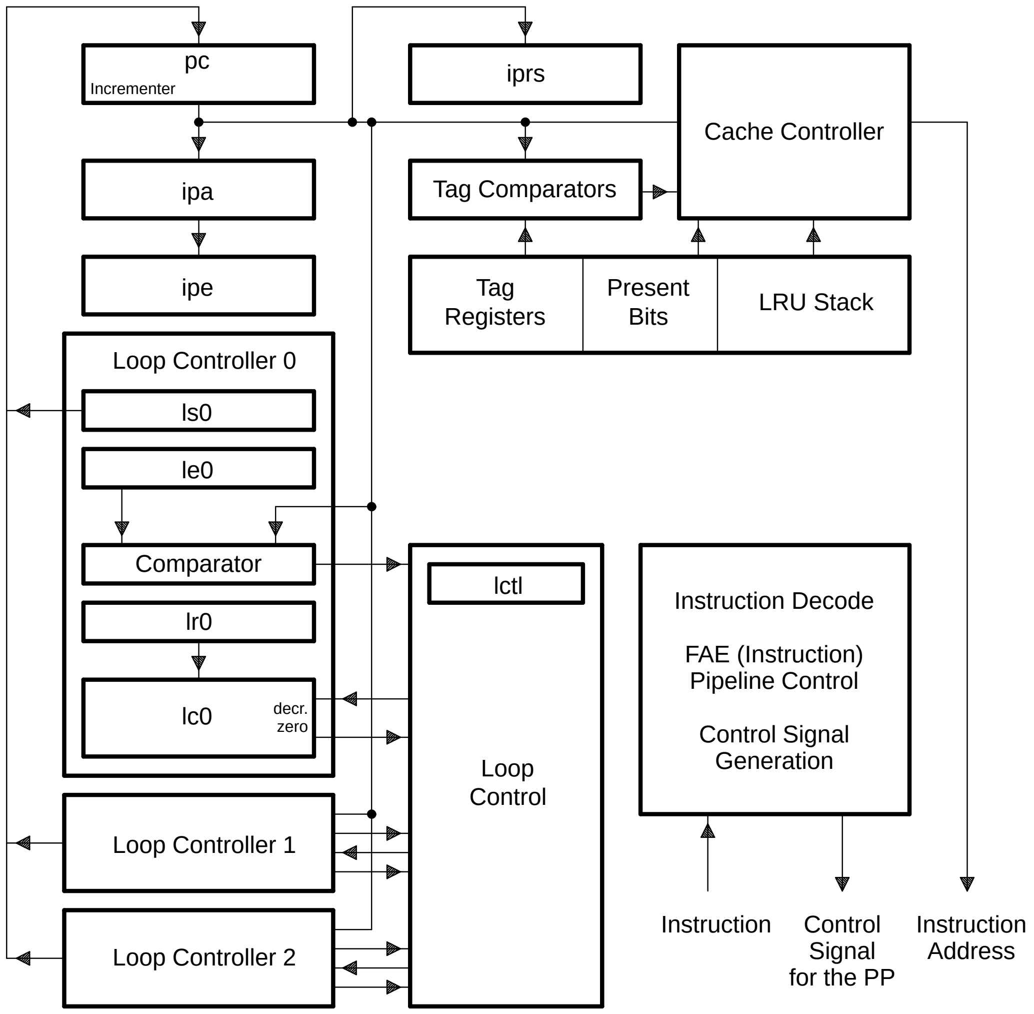
This chapter describes the PP’s program flow control unit and its components. Chapter 9, *Interrupts and Reset*, Chapter 10, *Understanding Branches and Calls*, and Chapter 11, *Understanding the Loop Control Logic*, provide a software-oriented description of interrupts, branching, calls, and zero-overhead looping program flow control elements.

Topics

5.1 Overview of the Program Flow Control Unit

The program flow control unit performs instruction fetching and decoding, handles any necessary handshaking with the transfer controller, and facilitates interrupt response and prioritization. Figure 5–1 shows a block diagram of the program flow control unit.

Figure 5–1. Program Flow Control Unit Block Diagram



5.2 FAE Instruction Pipeline

The PP has a three-stage instruction pipeline that the program flow control unit controls. This pipeline is different from the MP's three-stage pipeline (the FEA pipeline). The PP's pipeline stages are summarized as follows:

❑ **F**etch instruction stage

The address contained in the program counter (pc) is compared to cache-tag register and present flags.

- If the instruction is in cache, it is fetched, and instruction decoding is started.
- If the instruction is not in cache, the program flow control unit issues a cache service request to the TC and stalls the PP until the cache is loaded with the instruction.

Once the instruction has been fetched, the pc is incremented, unless the pc is written to by a software branch or reloaded with a loop start address by the hardware loop control.

❑ **A**ddress unit computation stage

All address unit computations specified by the previous fetched instruction are performed. The computed addresses are sent to the crossbar at the end of this stage for contention detection/prioritization. The actual memory accesses do not occur until the execute stage.

❑ **E**xecute data unit operations and memory transfers stage

In this stage, operands from the register file are fed to the data unit, the data unit operations are performed, and the results are written back to the register file. Also in this stage, data is transferred between registers or between registers and memory, as specified by the global and local transfers. Conflicts in accessing memory may stall the pipeline at this point.

Since the order of the stages is **f**etch, **a**ddress, and **e**xecute, the operation is referred to as an **FAE pipeline**. Pipeline stages of three instructions overlap, as shown in Figure 5–2. At any given time, one instruction is at the fetch stage, another is at the address stage, and the third is at the execute stage. Thus, when no stall conditions occur, the net throughput is one instruction every cycle.

Figure 5–2. FAE (Instruction) Pipeline

Instruction	T1	T2	T3	T4	T5
One	Fetch	Address	Execute		
Two		Fetch	Address	Execute	
Three			Fetch	Address	Execute

5.2.1 Program Counter-Related Registers

The PP contains program-counter-related registers, including an instruction pointer for each stage of the FAE pipeline and an instruction pointer for returns from subroutine calls. These registers are summarized below:

- **pc** (program counter)

  The pc register points to the instruction currently being fetched. The value in the pc register is incremented by one instruction (eight bytes) each time the pipeline advances, unless the pc is written to by software or by one of the loop controllers, or if an interrupt is taken. When a PP operation writes to the pc register, the program flow changes. You can specify branches and calls by software writes to the pc register. When writing to the pc, you can use one of two different register codes:

  - The **br** register code (used for branches) modifies the pc register.
  - The **call** register code (used for subroutine calls) saves the return address in the iprs register (described below), in addition to modifying the pc register.

- **ipa** (instruction pointer address stage)

  This read-only pipeline storage register tracks the address of the instruction that is at the address stage of the pipeline.

- **ipe** (instruction pointer execute stage)

  The ipe register is read-only and is used for program counter relative addressing because it tracks the address of the instruction being executed. **Do not** use the pc register as a source operand for program counter relative addressing: the pc may bear no relation to the instruction being executed, because of a branch or loop.

- ❏ **iprs** (instruction pointer return from subroutine)

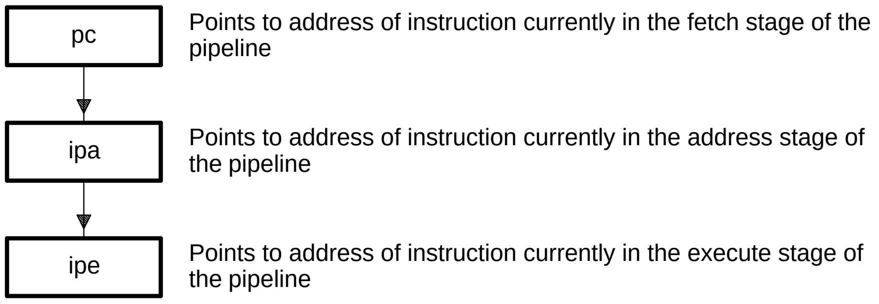
  This register contains an instruction pointer for a return from a subroutine call. Whenever a write is performed to the call register code, the iprs register is loaded with the instruction sequentially following the second delay slot instruction of the call (the two delay-slot instructions are executed before the branch to the subroutine address occurs). When nesting multiple calls, you must save the iprs register on the stack in order to preserve the return address of the prior call. To restore a previously saved value, you can write to the iprs register. To return from a call, move the contents of the iprs register back into the pc register, or load directly off the top of the stack the iprs value that was saved earlier.

In the FAE pipeline, the ipa and ipe registers record the program counter history, as shown in Figure 5–3.

On each cycle in which the pipeline advances (that is, there is no stalling condition), the contents of the ipa register are moved to the ipe register, and the contents of the pc register are moved to the ipa register.

As described in Section 10.1, *Software Branching*, many data unit and address unit operations can specify the pc register (using either the call or br register code) as the destination. This is the way that subroutine calls and branches are performed. Since both data unit and global address unit operations can be made conditional, either the data unit or a global address unit operation can perform a conditional branch.

Figure 5–3. Program Counter History



Points to address of instruction currently in the fetch stage of the pipeline

Points to address of instruction currently in the address stage of the pipeline

Points to address of instruction currently in the execute stage of the pipeline

5.2.2 Pipeline Implications

The PP’s FAE pipeline is designed for fast execution of streams of data that are processed in a tight loop. Data results from one instruction can be used or stored by the next instruction. One instruction can load data from memory to a register, the next instruction can operate on the data, and the following instruction can store the result from a register to memory. Because each PP instruction can define parallel operations, and because zero-overhead looping is supported, three operations like these can be performed in one instruction.

For example, multiply-accumulate operations flow well through the pipeline. With a set of data stored in each of two different RAMs, in a single instruction cycle, the PP can fetch two pieces of data to be multiplied in the next instruction, multiply the data fetched in the previous instruction, and add/accumulate results from the previous instruction’s multiply. The pipeline makes it possible for this same instruction to be executed repetitively in a zero-overhead loop with a net throughput of one multiply accumulate every cycle.

Unlike the MVP’s master processor, no register scoreboarding is performed on the PP.

The major pipeline effects and characteristics are discussed in the following subsections. These conditions hold true, regardless of any stalls in the pipeline due to events such as instruction cache misses or memory access contention. Stalls simply freeze the pipeline until the stall condition is removed.

5.2.2.1 Modifying an Address or Index Register

One of the consequences of the FAE pipeline is that whenever an address or index register is modified by the data unit or loaded by the address unit (both occur in the execute pipeline stage), that register **cannot** be used in the next instruction for address computations (which occur in the address stage). **If this rule is violated, the result of the address computation will be unpredictable**; either the old or new value may be used in the address computation, depending on whether an interrupt occurred or you are single-stepping or running full speed.

- If an interrupt occurs, the execute stage of the instruction modifying the address or index register occurs during the interrupt routine. When the return from interrupt occurs, the address computation of the next sequential instruction is performed using the **new** value of the address or index.

- If no interrupt occurs, the execute stage of the instruction writing to the address register occurs in parallel with the address stage of the next instruction. Thus, the address computation is based on the **old** value of the address or index.

You might think that if you disable interrupts to ensure that the old value of the register is used, you would be safe. However, if you use the debugger and single-step through your code, the new value is used. In summary, simply allow one delay-slot instruction between an instruction that modifies an address or index register in the execute stage and an instruction that uses that register for an address computation.

5.2.2.2 Branch Delay-Slot Instructions

Another consequence of the three-stage FAE pipeline is that two delay-slot instructions are associated with a branch or subroutine call. After an instruction that specifies a branch or call is fetched, two more instructions are fetched before the branch or call is executed. Unlike the master processor, the PP does not support an annul option. Due to possible prior branches or hardware looping, the two delay-slot instructions for a branch are not necessarily the next two instructions in memory.

In most cases, you can structure your code to fill the delay slots with two useful instructions. For example, for subroutine calls, you can use the delay-slot instructions to set up arguments and/or push registers onto the stack. For subroutine returns, you can use the delay-slot instructions to restore registers by popping the registers' previous contents off the stack. For circumstances in which no useful work can be done, you can fill a delay-slot instruction with an instruction that does nothing (nop).

For two primary reasons, PP code tends to require fewer software branches (and their associated delay slots) than most other processors.

❑ First, the PP supports up to three levels of hardware-controlled looping. Hardware looping operates in the fetch stage of in the pipeline so it does not have any associated delay slots for the hardware loop. (Note that the looping-related registers must be loaded at least two instructions before a corresponding loop end address is encountered.)

❑ Secondly, the PP's support of conditionally executed instructions can be used to avoid conditionally branching around a few instructions.

5.2.2.3 Summary of Pipeline Implications

The major pipeline effects and characteristics discussed in this subsection are summarized below. These conditions hold true, regardless of any stalls in the pipeline caused by events such as instruction cache misses or memory access contention. Stalls simply freeze the pipeline until the stall condition is removed.

- ❑ Registers loaded in one instruction can be used as data in the next instruction.
- ❑ Data results from one instruction can be stored by the next instruction.
- ❑ Because the address computations occur ahead of the execute stage, you must allow one delay slot instruction before using an address or index register that is changed by the data unit or by a load from memory. Unpredictable behavior will result if this condition is not met. The assembler cannot ensure that this condition is met.
- ❑ The two instructions fetched following a branch or call (the delay-slot instructions) are always executed, regardless of whether the branch is taken or not. The delay-slot instructions are not necessarily the next two sequential instructions if there are other branches or loops.
- ❑ Zero overhead loops have no delay-slot instructions. The loop controllers work in the fetch pipeline stage to eliminate delay slots.
- ❑ Because the loop-controlling registers work in the fetch stage of the pipeline, loop control registers must be loaded at least two instructions before the corresponding loop end address is encountered.

5.2.3 Pipeline Stall Conditions

Several events can cause the normal instruction fetching operations to cease temporarily. These events, referred to as pipeline stall conditions, are discussed briefly in this subsection. When any of these stall conditions occur, the program flow control unit instructs the address units and data unit to stall. Therefore, the program flow control unit is considered to be master over the other two units. Program flow resumes, once the stall condition is cleared.

These are examples of pipeline stall conditions:

- Cache miss

  Cache misses are detected by the cache control logic, and the transfer controller is automatically signaled when one occurs. The information that the TC needs is supplied by the program flow control unit to allow the TC to service the cache. The PP stalls until its cache request has been serviced.

- Illegal operation detection

  An unimplemented opcode, fetched during the previous fetch cycle, is decoded, causing the pipeline to stall and an interrupt request to be sent to the MP. The PP remains stalled until the MP sends a reset command.

- Crossbar contention

  If one or both of the local and global data ports experience contention with another processor when requesting a memory access, the program flow control unit and data unit stall. The address units also stall, although they continue to request access to the memory. Once contention is resolved, the program flow control unit releases all units to normal operation.

- Local port address miss—diversion to global port

  When a local transfer is performed in parallel with a global transfer, the local transfer access is attempted over the local port. If the local transfer specifies an access to an address that is not in the PP’s local RAM, the transfer cannot occur over the local port, and the pipeline stalls. Once the global transfer has completed (thus freeing up the global port), the local transfer is diverted automatically to the global port, and the access is completed.

- Global port address miss—DEA (direct external access)

  If the address unit detects that it has generated an address that is not in the shared RAMs, then the pipeline stalls and a DEA request is sent to the TC. This is serviced in a manner similar to a cache miss. It finally completes like a normal memory access.

- Halt request

  The MP or the PP itself can specifically halt the operation of a PP under software control. This causes the pipeline to stall until the MP unhalts the PP.

5.3 Loop Controllers

The program flow control unit has three sets of zero-overhead loop controllers. Each controller has a register for the loop start address (ls0 – ls2), loop end address (le0 – le2), current loop count (lc0 – lc2), and loop reload count (lr0 – lr2). A loop control register (lctl) specifies which (if any) loop controllers are active.

5.3.1 Hardware-Looping Mechanism

For a hardware loop, a comparator detects when the pc register is fetching the instruction at the loop end address.

- If the pc register points to the loop end and the loop count is nonzero, then the loop controller loads the pc register with the address in the loop start register to cause the next instruction to be fetched from the top of the loop; also, the loop count is decremented by 1.
- If the pc register points to the loop end address and the loop count is 0, then the pc register is allowed to increment to point at the next sequential instruction (thus exiting the loop). Also, the loop reload count is copied into the loop count register to set up the next pass through the loop (for example, in the case of a hardware loop nested inside another loop).

5.3.2 Multiple Nested Loops

Multiple nested hardware loops with a common end address are supported by prioritizing the three loop controllers. Loop controller 2 has the highest priority (and thus controls the innermost loop), and loop controller 0 has the lowest priority (thus controlling the outermost loop).

When the pc register points to the common end of loop, the highest priority loop start address register with a nonzero loop count is copied into the pc register. If, at the common end address, a higher priority loop counter is 0, that loop counter is reloaded by its corresponding loop reload register. If all loop counters are 0 at the loop end address, then the pc register increments and all the corresponding loop counters are reloaded.

5.3.3 Hardware Branching

In addition to specifying which, if any, loop controllers are active, the lctl register associates a loop counter register with each loop end. Using a loop controller with no associated loop counter essentially sets up a hardware-controlled branch. The loop end register (le*n*) specifies where the branch will occur, and the loop start register (ls*n*) specifies where to branch to. Since hardware detects the branch conditions, these branches occur with no overhead, once the loop start and end registers are set up.

Hardware branching can be particularly useful for supporting run-time options. The address that is loaded into the loop start register can point to the option. For example, graphics has different pixel-processing options for algorithms such as line drawing and BitBLT that are often selected at runtime. With the three sets of loop controllers, hardware branches can be combined with hardware looping; one controller can be used to control the loop, a second controller the branch out (to the option), and the third controller the branch back. This example is described more thoroughly in subsection 15.1.3, *Pixel Block Transfers (PIXBLTs)*.

You can use the loop controllers to perform zero-overhead conditional branches by loading the loop counter (lc*n*) register with either a zero or nonzero value before the corresponding loop end address is reached. If the value loaded into the loop counter is nonzero, then the program will branch to the loop start address. Furthermore, this technique of zero-overhead conditional branching can be combined with zero-overhead looping; a higher priority controller is used for the conditional branch, and a lower priority controller is used to control the loop.

5.3.4 Software Branching Within a Loop

Software branches or calls (that is, software writes to the pc register) take precedence over the hardware loop controllers when the branch or call instruction is executed in the same cycle in which an instruction at a loop end address is fetched (that is, the instruction at the loop end address is the second delay-slot instruction of a branch or call). When the pc register is written to by software, loop ends are ignored and loop counts are not updated. This allows loops to be exited conditionally (regardless of the loop count).

When you place loop ends in conditional branch delay slots, take special care in order to get predictable behavior. Section 11.1, *Looping*, discusses this, gives several examples, and contains a more detailed description of the loop controllers operation and capabilities.

5.4 Cache Controller

The instruction cache controller inside the program flow control unit contains the address tag registers, address tag comparators, present flags, least recently used (LRU) stack, and control hardware for managing the PP's instruction cache.

As discussed in subsection 3.1.1, *Cache Architecture*, the instruction cache RAM contains 2K bytes or 256 64-bit doubleword instructions, broken into four sets of 64 doubleword blocks. Each block in turn is broken into four sub-blocks of 16 doublewords each.

Each block has a tag register that contains the 23 MSBs of the address for the instructions held in that block of the cache. The tag comparators associated with each block compare the address of the instruction being fetched with the tag register to see if the address is in the range of the block. Each time an instruction is fetched from a block, that block becomes the most recently used and is put at the back of the LRU stack.

Each of the subblocks has a present flag to indicate whether the instructions in that subblock have actually been transferred from off-chip memory into the cache. Whenever there is a cache miss (the instruction being fetched is not in the cache), a cache service request is submitted to the transfer controller (TC) to transfer the entire subblock (16 instructions). Thus, the PP caching method always pulls in 16 instructions when the next instruction needed is not already present in the cache. This reduces the number of cache misses and makes better use of the available off-chip bandwidth. After the subblock has been brought into the cache, the subblock's present flag is set and the PP resumes instruction execution.

A more detailed explanation of the PP's instruction cache operation is given in Chapter 3, *PP Instruction-Cache Operation and Interprocessor Communications*.