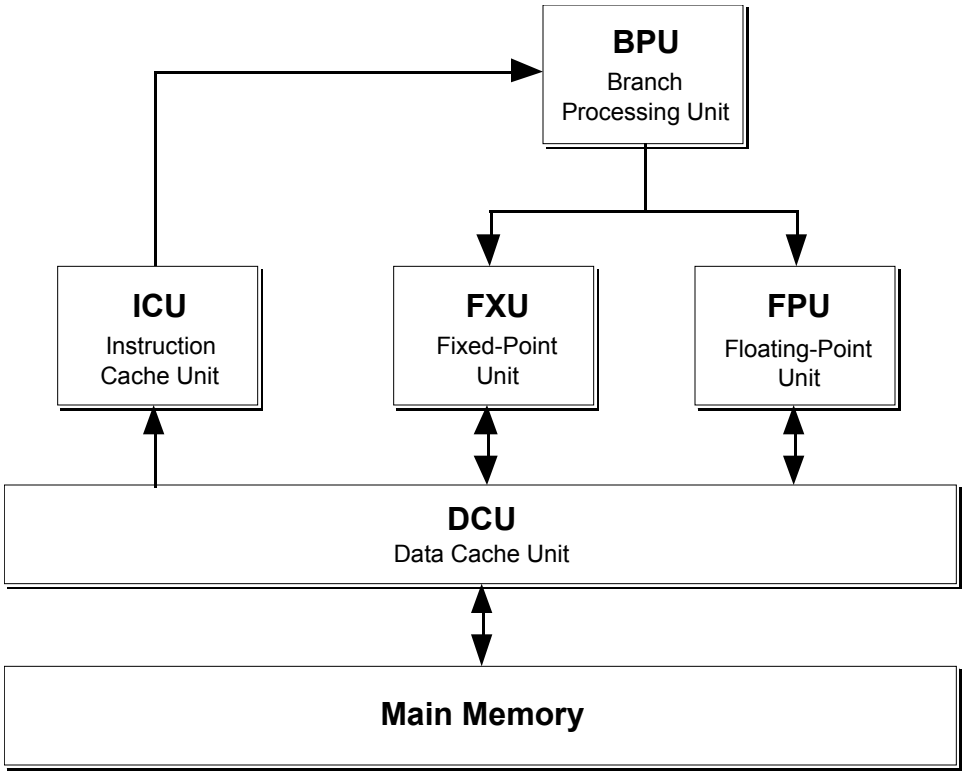
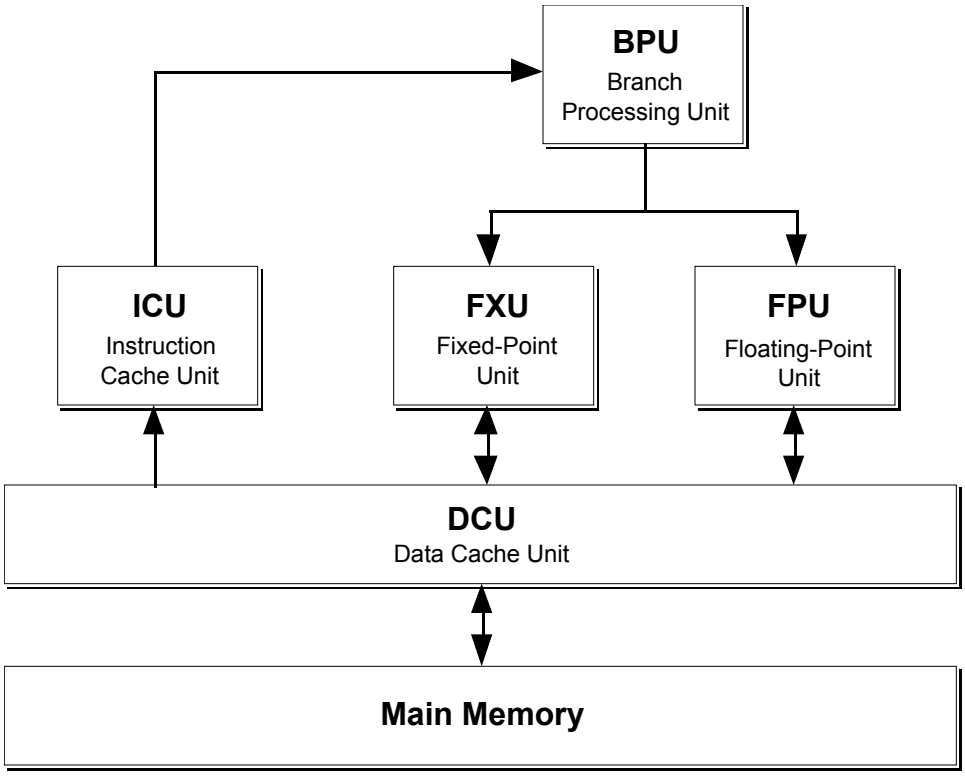


[Figure 2.1] Logical view of CPU functional units

Assembly Language Programming
and
Optimization Techniques
for the
POWER Architecture

platypus@curie.ces.cwru.edu



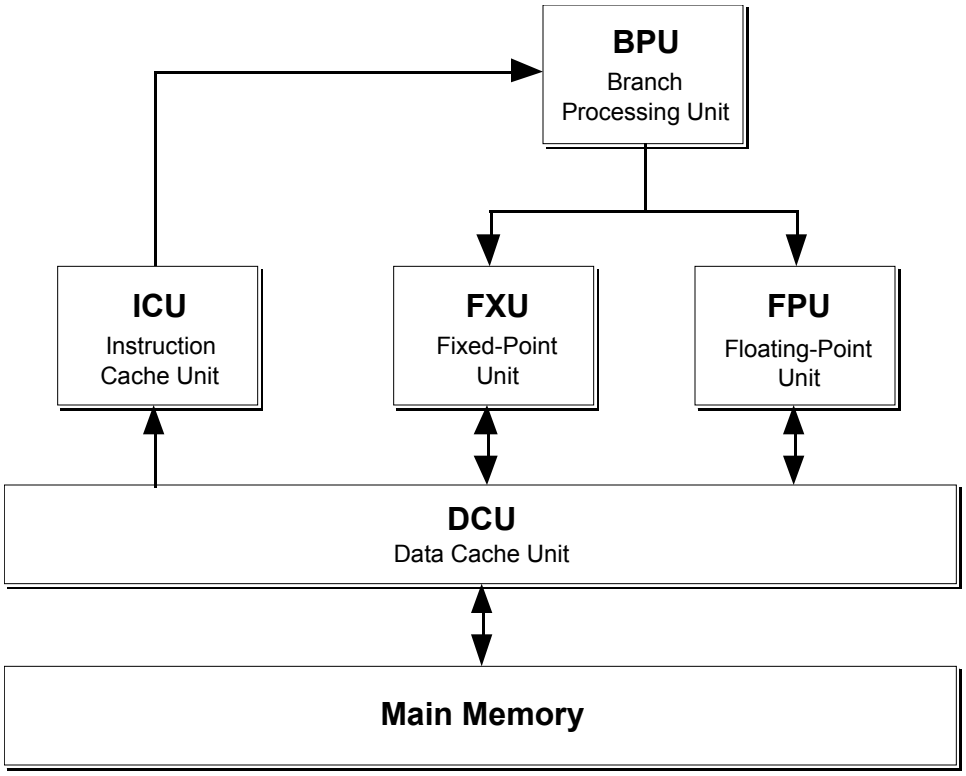
[Figure 2.1] Logical view of CPU functional units

2 Assembly Language Programming and Optimization Techniques for the Power Architecture

Abstract: This paper presents an overview of the POWER architecture and discusses techniques for writing efficient assembly language programs for machines based on this architecture (and its derivatives). IBM's RS/6000 is currently the most commonly available machine which is based on this architecture, and thus, it is the machine which is used as the focus of this presentation. After a brief description of the architecture and the available instructions, optimization techniques which are specific to POWER programs are presented along with a discussion of why they are important. In addition, a set of tools for the Macintosh which allows programmers to assemble, execute and analyze POWER assembly language programs is described. These tools operate on standard RS/6000 XCOFF format `.o` *and* `.obj` *files.*

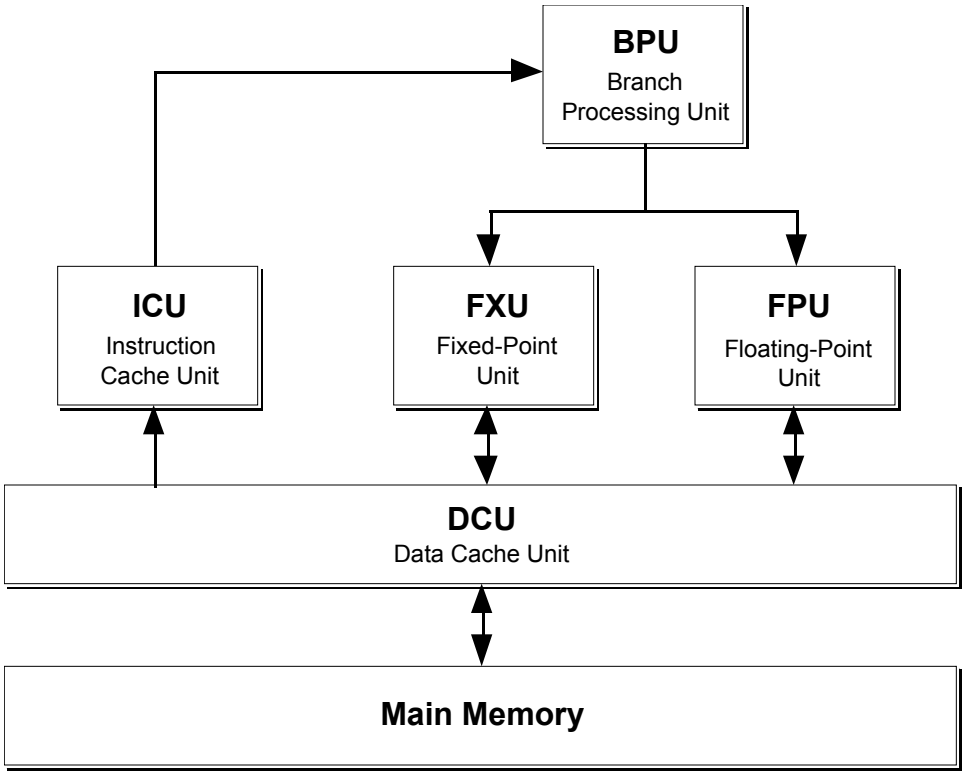
Disclaimer: (from the Introduction)
Modifications to the POWER architecture for the PowerPC Architecture are not discussed in detail and are only presented in this paper when ① the information is important to the topic being discussed, and (most importantly) ② the information has been publicly released by either IBM or Motorola.

Last Minute Note:
Immediately after finishing (and sending off the "final" copy of) this paper, Motorola released the technical information for the PowerPC 601. While time constraints prevented incorporating more detailed timing information about the



[Figure 2.1] Logical view of CPU functional units

601, the instruction set summary in Appendix A has been changed to incorporate the new information.



[Figure 2.1] Logical view of CPU functional units

1.0 Introduction

The POWER Architecture is a descendent of the first RISC architecture, IBM's 801, which was developed starting in 1975. Since then, research groups at IBM have enhanced the architecture and have developed a variety of machines based on it, most notably the IBM RT and the RISC System/6000[1†] .

The RS/6000 is the first machine to use the POWER architecture and is a direct descendent of the AMERICA architecture. The AMERICA architecture was IBM's first attempt at a "superscalar" architecture, i.e., one which could execute more than 1 instruction per cycle.

The PowerPC architecture[2‡] is the next generation of the POWER architecture and includes a variety of modifications which facilitate its usefulness for desktop machines.

This article presents an overview of the POWER architecture and discusses techniques for writing efficient assembly language code. Modifications to the POWER architecture for the PowerPC are not discussed in detail and are only presented in this paper when ① the information is important to the topic being discussed, and (most importantly) ② the information has been publicly released by either IBM or Motorola.

2.0 Architecture Overview

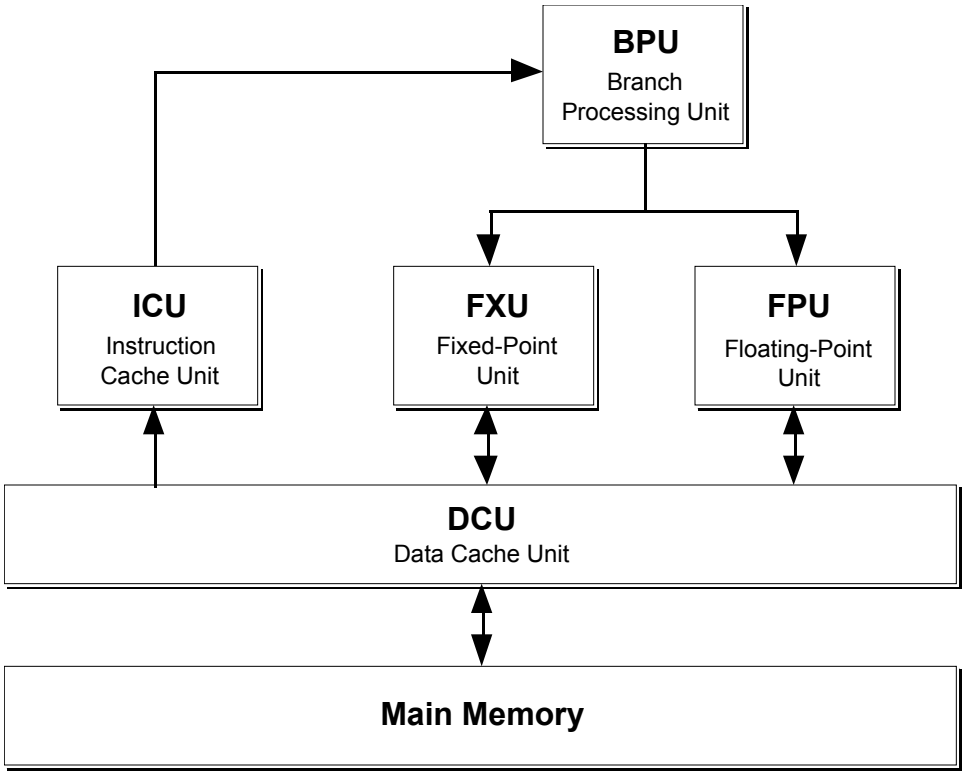
This section presents an overview of the POWER architecture, but first, a few paragraphs about the data organization of the processor are required.

2.1 Data Organization

The POWER architecture is designed around a word size of 32-bits, but it also has support for doubleword (64-bit), halfword (16-bit) and byte (8-bit) operations. Multi-byte data is stored in

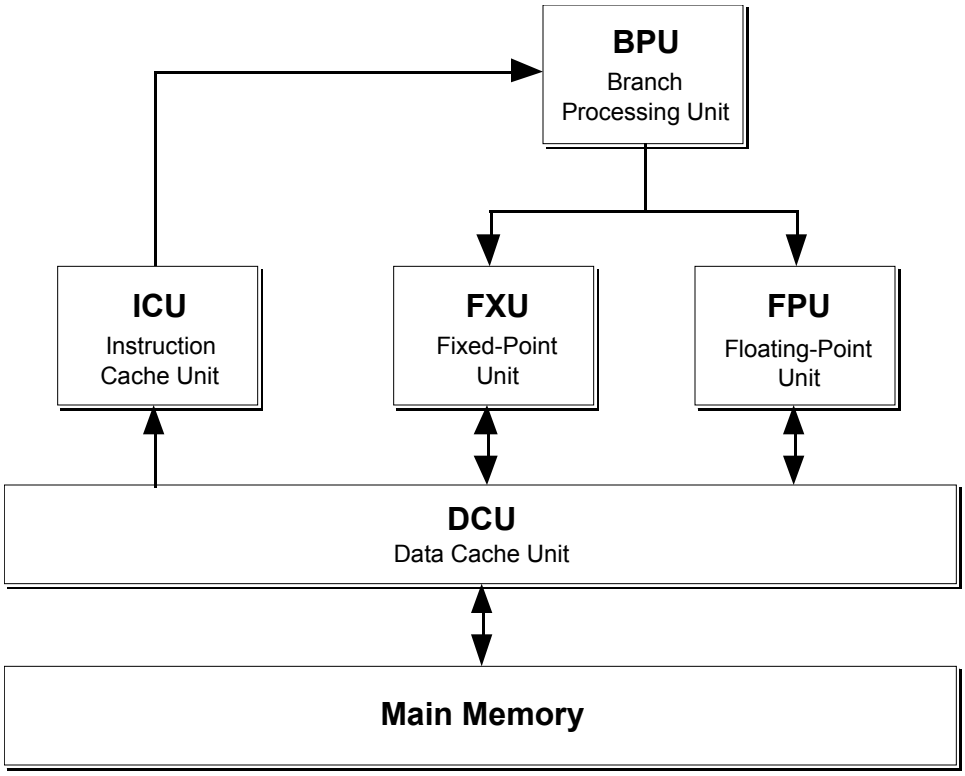
[1†] RISC System/6000 is a trademark of International Business Machines Corp.

[2‡] PowerPC Architecture is a trademark of International Business Machines Corp.



[Figure 2.1] Logical view of CPU functional units

big-endian format (most-significant byte first, like the 680x0), but the architecture does define load and store instructions which support little-endian



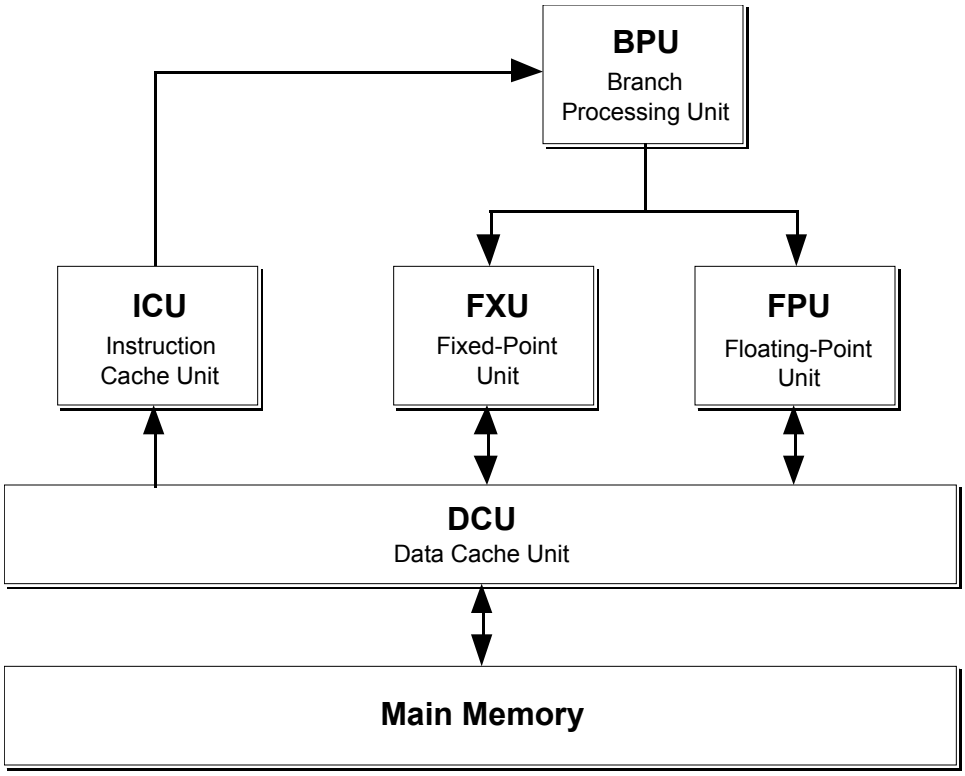
[Figure 2.1] Logical view of CPU functional units

BPU:	Condition Register (CR): Contains 8 fields to store comparison results Link Register (LR): Holds target address for branch instructions Counter Register (CTR): Used for decrement-counter-and-branch loops Machine State Register (MSR): Contains flags describing state of processor
FXU:	General Purpose Registers 0-31 (GPR 0-31): The 32 32-bit fixed-point registers Fixed-Point Exception Register (XER): Contains fixed-point operation results and flags Multiply Quotient Register (MQ): Used by multiply, divide and extended shift op's Real-Time Clock (RTCU, RTCL): The real-time clock Decrementer (DEC): Used to set interrupts after an elapsed time
FPU:	Floating-Point Registers 0-31 (FPR 0-31): The 32 64-bit floating-point registers Floating-Point Status and Control Register (FPSCR): Contains status of FPU

[Table 1.1] Summary of architected registers for the RS/6000.

format data (least-significant byte first, like the 80x86). The PowerPC differs slightly from the RS/6000 in that it has added an addressing mode switch which allows it to operate in either big-endian or little-endian mode ([Diefendorff93] p.4).

Bits within bytes (and words, et al.), however,



[Figure 2.1] Logical view of CPU functional units

are numbered using the little-endian format. This means that bit 0 is the most significant bit (many times interpreted as the sign bit). This is backwards from the standard number scheme that Motorola uses for the 680x0. This bit numbering scheme is important because many instructions require bits to be specified.

2.2 Overview

A POWER CPU is divided into 5 functional units: the Fixed-Point Unit (FXU), the Floating-Point Unit (FPU), the Branch Processing Unit (BPU), and the Instruction and Data Cache Units (ICU and DCU). These units are interconnected as shown in [Figure 2.1].

The BPU handles all instructions involving the Condition Register and all branching instructions. All other instructions are passed on to both the FXU and the FPU. The BPU performs instruction lookahead down both paths of a branch to reduce or eliminate branch delay. The main purpose of the BPU is to resolve/remove branches and provide a steady instruction stream to the Fixed- and Floating-Point Units.

The FXU handles all fixed-point operations and all load and store operations (including floating-point loads and stores). All other instructions passed into the FXU are ignored.

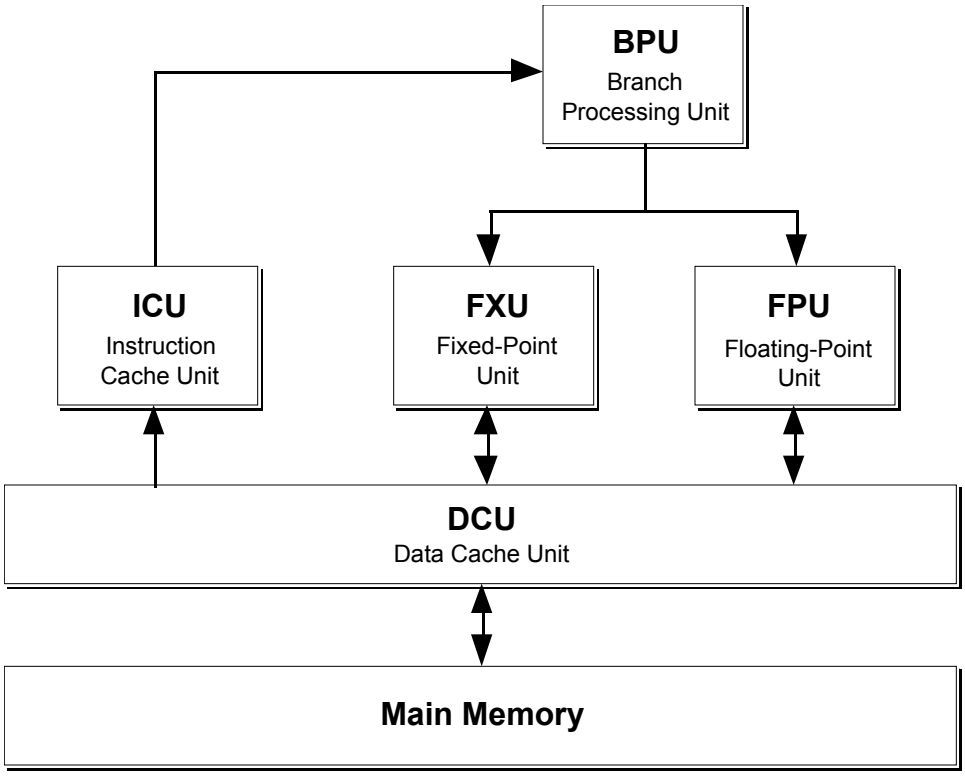
The FPU handles all floating-point operations. Floating-point loads and stores are passed through the floating-point pipeline so that they can be synchronized with the fixed-point pipeline (where the loads/stores are actually executed).

The ICU and DCU provide *n*-way set associative cache[3†] interfaces to the computer's main memory. The ICU feeds instructions to the BPU, and the DCU provides read/write access to memory for the ICU, FXU and FPU.

2.3 The Branch Processing Unit (BPU)

The BPU fetches instructions from the ICU and passes instructions that it can't handle on to the

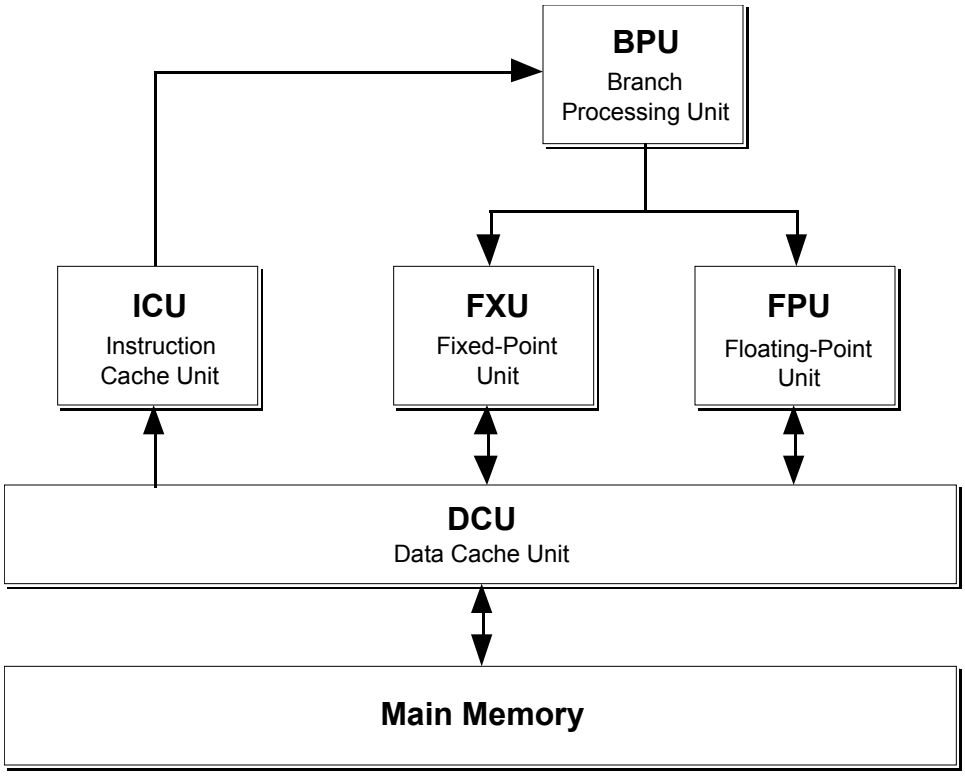
[3†] A set-associative cache limits where a particular block can go by restricting it to a set of blocks in the cache. If there are *n* blocks in the cache set, the cache is *n*-way set associative. (see [Patterson90])



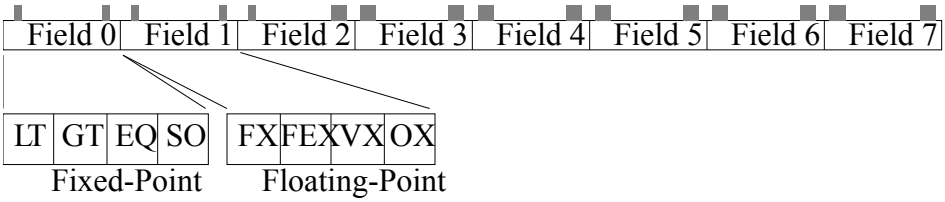
[Figure 2.1] Logical view of CPU functional units

FXU and FPU. The BPU handles all of the instructions which involve branching or the Condition Register.

The BPU consists of 4 architected registers: the Condition Register (CR), the Counter Register (CTR), the Link Register (LR), and the Machine State Register (MSR).



[Figure 2.1] Logical view of CPU functional units



[Figure 2.2] Interpretation of the bits in the Condition Register

The Condition Register consists of 8, 4-bit fields (numbered 0-7) which contain flags that indicate the result of various operations. The interpretation of the bits in each field depend on whether they were set as a result of a fixed- or a floating-point operation.

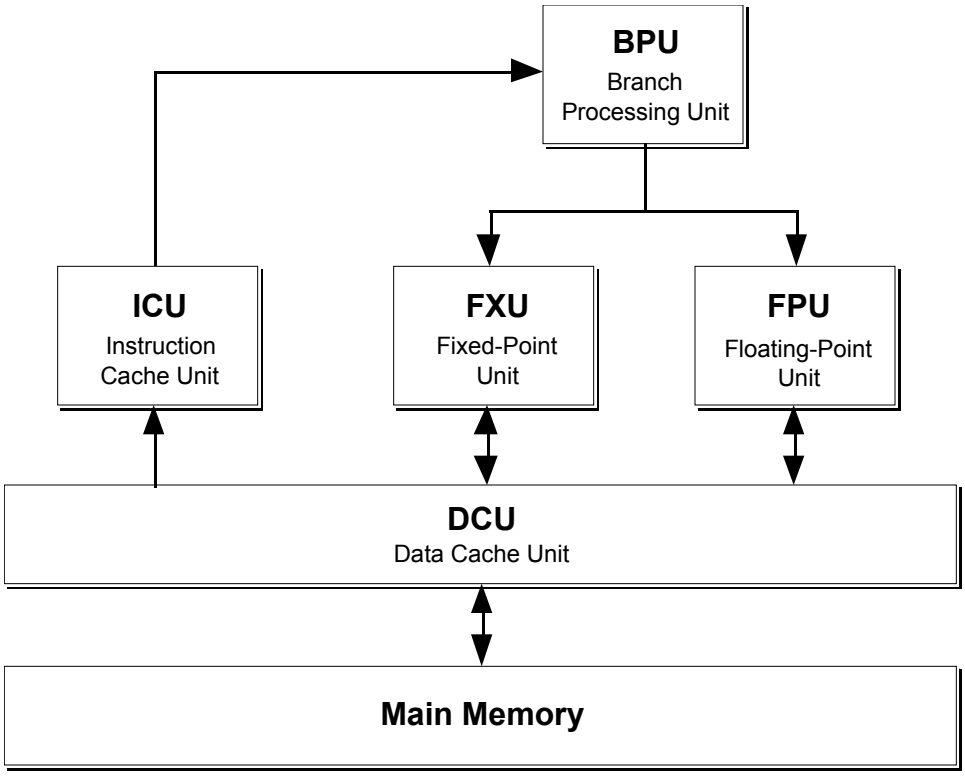
When a CR field is set due to a fixed-point operation, the 4 bits are interpreted as follows:

bit 0: [LT] Result is Less Than 0, or negative.
bit 1: [GT] Result is Greater Than 0, or positive.
bit 2: [EQ] Result is Equal to 0.
bit 3: [SO] Summary Overflow. From XER[SO].

When a CR field is set due to a floating-point operation, the 4 bits are interpreted as follows:

bit 0: [FX] FP Exception.
bit 1: [FEX] FP Exception Enable.
bit 2: [VX] FP Invalid Operation Exception.
bit 3: [OX] FP Overflow Exception.

The values of these bits are taken from the



[Figure 2.1] Logical view of CPU functional units

FPSCR (q.v. FPU) after the instruction has completed.

The organization of the Condition Register is shown in [Figure 2.2]. Note that any of the eight fields can hold the result of either a fixed-point or a floating-point operation.

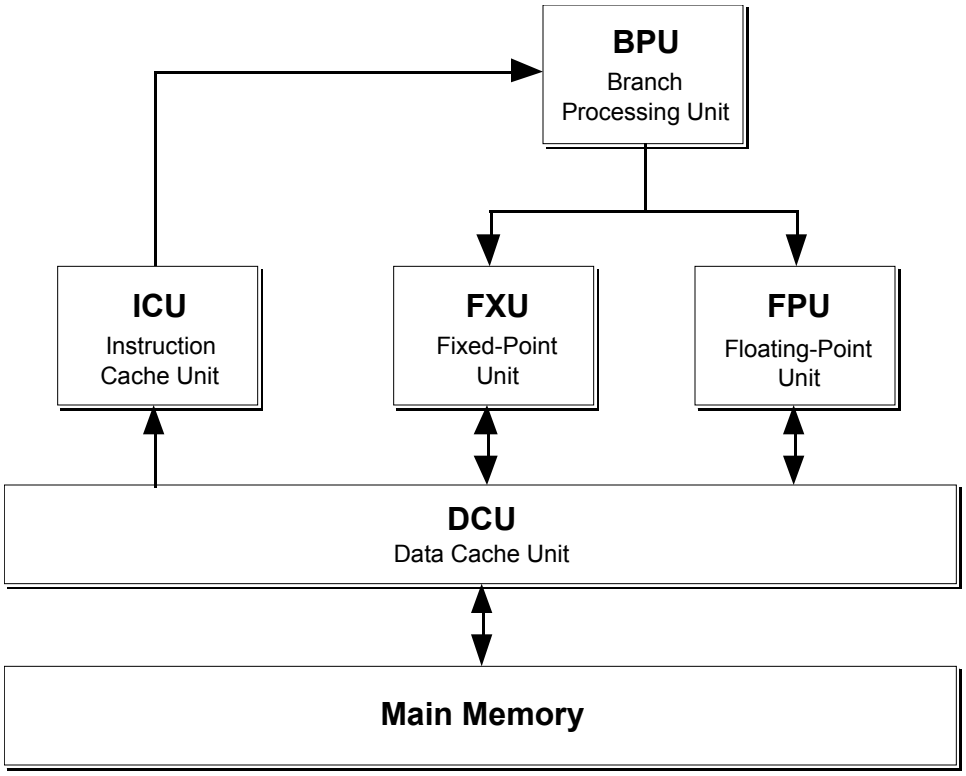
CR Field 0 can be implicitly set by fixed-point operations, and CR Field 1 can be implicitly set by floating-point operations. The other fields can be specified to store the result of compare operations, and operations are provided to move between two CR fields.

The 8 different CR fields allow the programmer to use different fields for different conditional branches. This also provides a mechanism by which the programmer can move the condition upon which a branch is dependent earlier in the code to facilitate zero-cycle branching (q.v. §6.0 Branching Optimization Techniques).

The Counter Register (CTR) is used to efficiently implement branch and count operations. The branch and count operation uses this register in the BPU instead of directly accessing a GPR (located in the FXU) because directly accessing the FXU would incur extra cycle penalties moving the data between the processor units. Requiring the programmer to add an extra instruction to move the count-loop limit into this special register gives the programmer the opportunity to schedule the CTR-loading instruction so that this penalty is executed in parallel with another instruction, thus eliminating any delay.

The Link Register (LR) is used to hold the target address for certain types of branch instructions. Branching through the LR is the standard way to return from a routine. The LR is typically loaded by setting the link bit in a branch instruction, which automatically loads the address of the instruction immediately after the branch into the LR.

The Machine State Register (MSR) contains an array of flags which defines the current state of the processor. These flags include Exception Enable, Instruction/Data Relocate and Alignment Check flags.

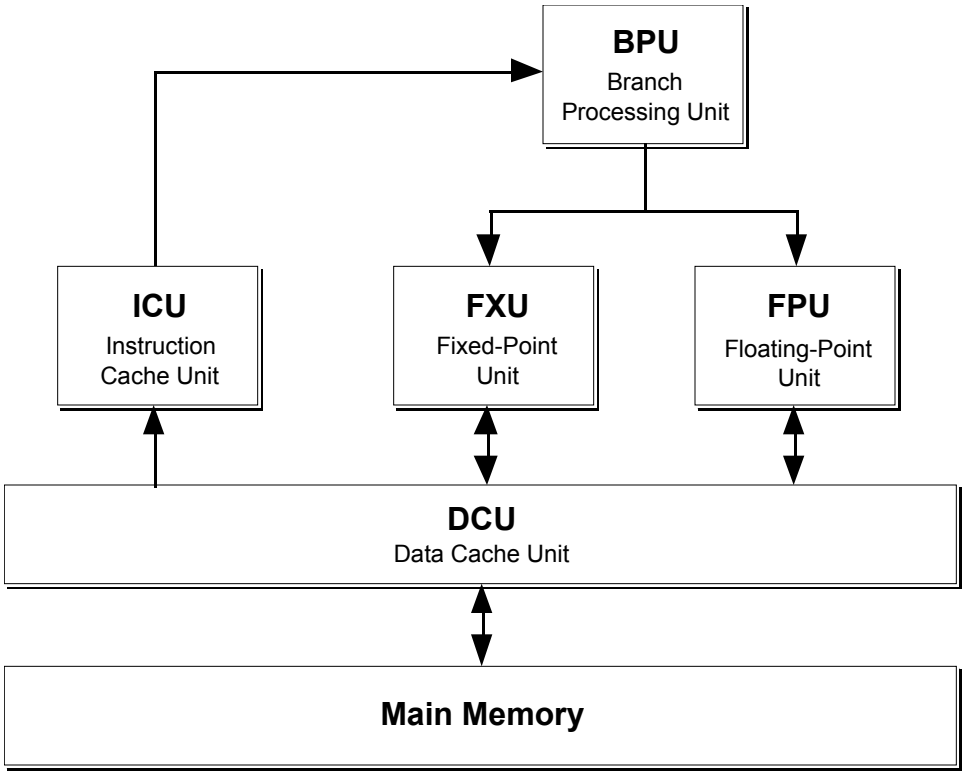


[Figure 2.1] Logical view of CPU functional units

In addition, the BPU also contains the Save and Restore Registers (SRR's) which are used to save and restore the state of the processor when there is an interrupt.

2.4 The Fixed-Point Unit (FXU)

The FXU handles all of the fixed-point arithmetic operations and all of the data-address calculations



[Figure 2.1] Logical view of CPU functional units

for itself and the FPU. The FXU controls how the FPU interacts with the data cache and directs the flow of data to and from it.

The FXU contains 32, 32-bit general purpose registers (GPR 0 - GPR 31) and a Fixed-Point Exception Register (XER). Four special purpose registers (MQ, RTCU, RTCU, and DEC) are also included in the FXU.

The XER contains the Carry, Overflow and Summary Overflow bits. The Carry bit is set and cleared based on whether or not a fixed-point arithmetic operation produces a carry out of bit 0. The Overflow bit is likewise set or reset based on whether or not the operation causes an overflow. The Summary Overflow is set whenever the Overflow bit is set, but it is never automatically cleared - the SO bit must be explicitly cleared by the programmer.

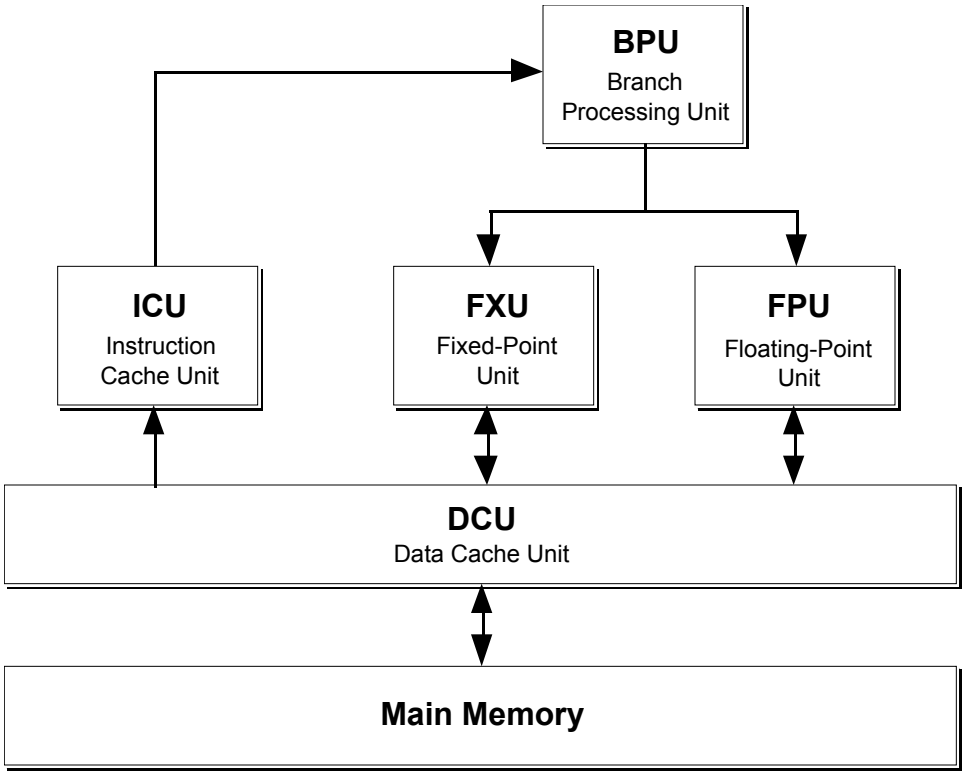
The RS/6000 also includes a Multiply Quotient (MQ) register which is used by the multiply, divide and a few other instructions. This register has been removed from the PowerPC because it is considered a single resource which causes unnecessary conflicts in superscalar implementations ([Oehler92], [Case91]).

The RTCU, RTCL and the DEC registers are used to access the 64-bit Real-Time Clock in the RS/6000. The RTC is divided into an upper (RTCU) and a lower (RTCL) register to make it easier to access the clock. The RTCU contains the time in seconds, and the RTCL contains the time in nanoseconds. The DEC register is used to signal in interrupt after a specified amount of time has passed.

2.5 The Floating-Point Unit (FPU)

The FPU contains 32, 64-bit floating-point registers (FPR 0 - FPR 31) and a Floating-Point Status and Control Register (FPSCR).

The FPR's are used for all floating-point operations and the FPSCR contains a wide array of flags for the floating-point unit, including the current rounding mode, flags indicating which exceptions are enabled, the result of previous



[Figure 2.1] Logical view of CPU functional units

instructions (less than, greater than, equal, unordered) and flags indicating whether an exception has occurred.

The floating-point unit provides the hardware necessary for a POWER system to conform to the *IEEE Standard for Floating Point Arithmetic* (ANSI/IEEE 754-1985), but relies on supporting software for the system to be fully compliant[4†].

Because of IBM's aggressive floating-point engine in the RS/6000, there was no speed advantage to performing single-precision arithmetic and it was therefore not included in the architecture. The RS/6000 performs all floating-point operations in double-precision and rounds to single-precision if needed.

However, many of the PowerPC implementations will not be able to afford the additional cost and complexity of the hardware which is found in the RS/6000's floating-point unit. It is for this reason that single-precision operations were added to the instruction set for the PowerPC.

3.0 Instruction Set

This section presents an overview of the POWER Architecture instruction set. It is intended to provide an introduction to and to complement the instruction set list given in Appendix A.

3.1 Addressing Modes

There are basically five addressing modes in the POWER architecture: Absolute, Absolute Immediate, Relative Immediate, Indexed and Based Addressing.

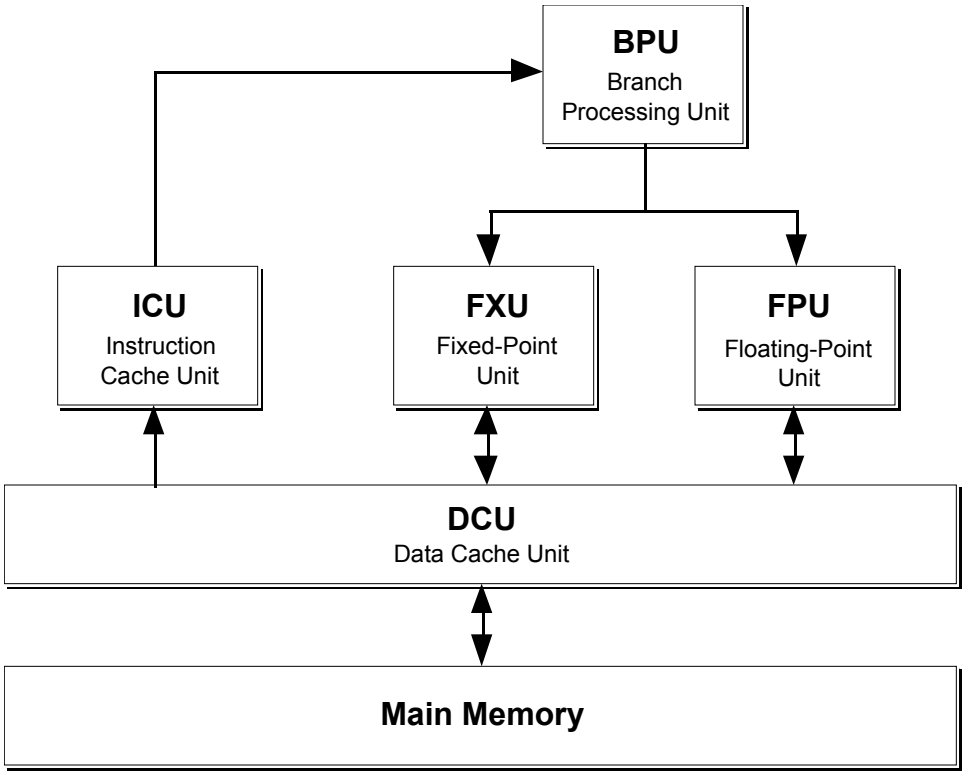
Absolute Addressing specifies an absolute address as the contents of a register. E.g.:

```
br
```

Branch unconditionally to the absolute address stored in the Link Register.

Absolute Immediate Addressing specifies a 26-bit absolute address as an immediate value which is

[4†] For example, the square-root operation (required by the standard) needs to be performed in software.

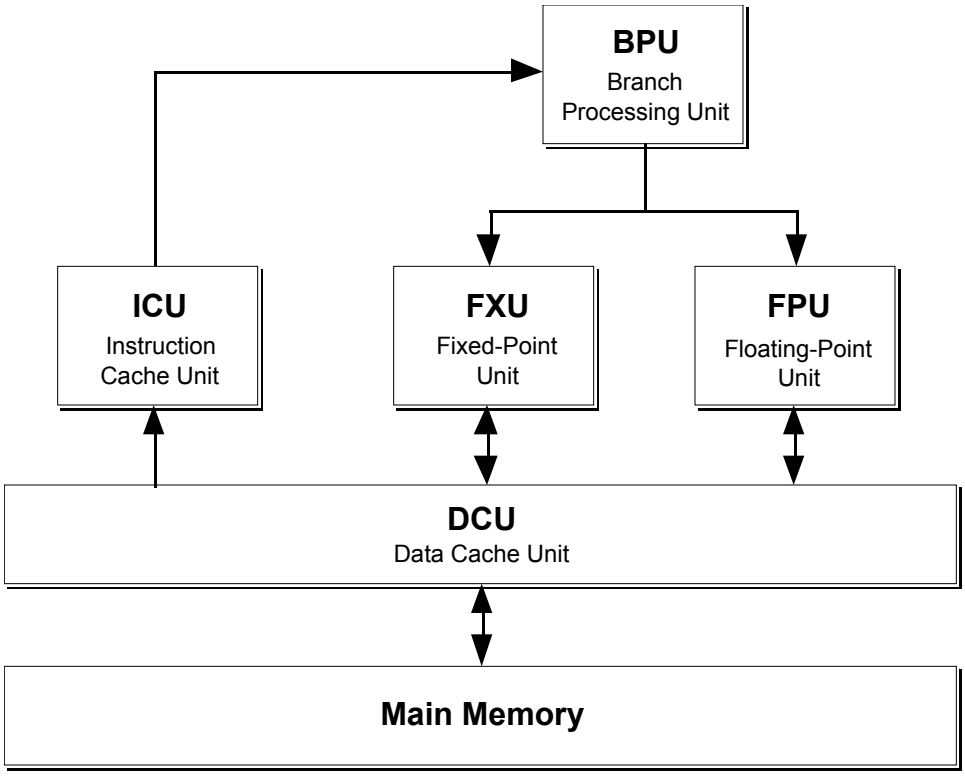


[Figure 2.1] Logical view of CPU functional units

encoded directly in the instruction. E.g.:

```
ba    <label>
```

Branch to the absolute address specified by *<label>*.



[Figure 2.1] Logical view of CPU functional units

Relative Immediate Addressing specifies a 26-bit offset which is added to the current instruction counter to produce an absolute address. The offset is encoded in the instruction.

```
b     <label>
```

Branch to the address specified by *<label>*.

Indexed Addressing calculates an absolute address by adding the contents of two registers together. If the first register specified is GPR 0, the second register is added with 0. E.g.:

```
lx    r31,r20,r2
lx    r30,0,r1
```

Load `r31` with the word from the address calculated from the sum of `r20` and `r2`. Load `r30` with the word from the address that `r1` points to.

Based Addressing calculates an absolute address from a base register and an 16-bit offset. The offset is added to the contents of the base register to produce the address. If GPR 0 is used as the base register, this is interpreted as meaning that no base register should be used. E.g.:

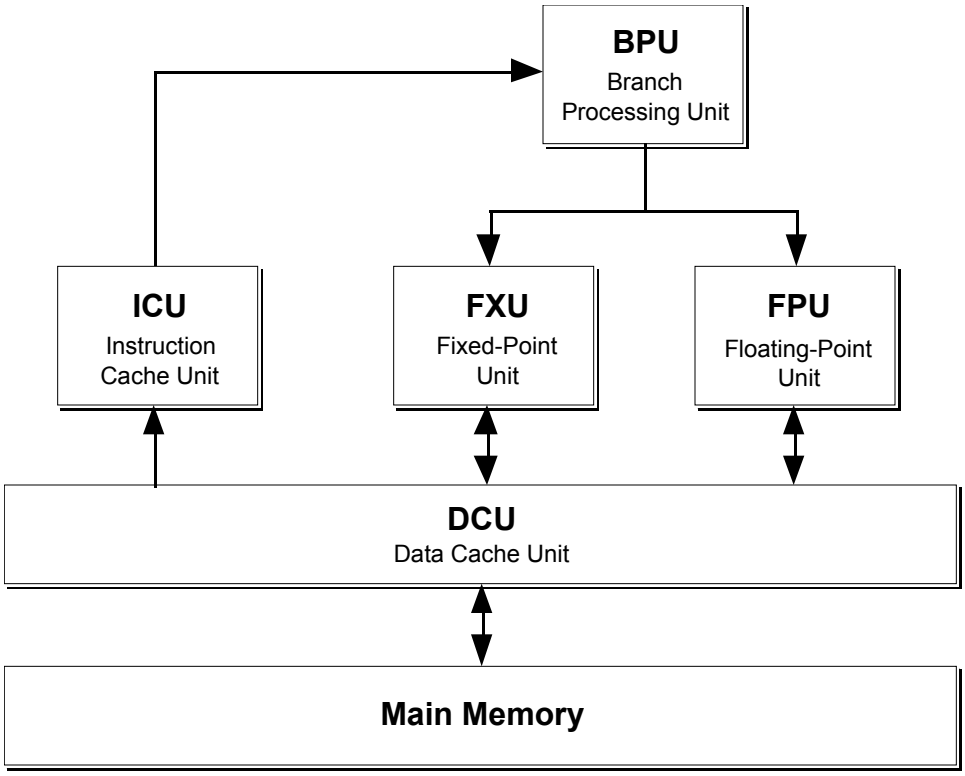
```
l     r31,12(r2)
l     r30,12(0)
```

Load `r31` with the word located 12 bytes from the address that `r2` points to. Load `r30` with the word located at absolute address 12.

3.2 The Instructions

A list of instructions used by POWER processors is given in Appendix A. This table contains a brief description of each instruction and its syntax. Instructions which are being removed for the PowerPC are marked with a '✗' at the beginning of the instruction description. The information for this table was derived from the RS/6000 Assembly Language Reference [IBM92a] with additions from [Oehler92], [Diefendorff93] and [Case91,92].

A summary of the changes made for the PowerPC is given in Appendix B. This is not a



[Figure 2.1] Logical view of CPU functional units

complete list of the modifications made, but attempts to cover the major changes which will affect programmers.

Many of the instructions listed in Appendix A have optional modifiers which can be added to the end of the base mnemonic to change the operation of the instruction. Each instruction mnemonic is followed by the valid suffixes enclosed in square brackets (e.g.: `a[o][.]`). These suffixes are standard across the instruction set and are summarized here:

. Set the Record Bit in the instruction so that the condition codes are set. The condition codes are stored in CR field 0 (for fixed-point) or CR field 1 (for floating-point) based on the results of this instruction.

o Set the overflow (and summary overflow) bits in the XER based on the results of this instruction.

Load and store operations may have the following modifiers:

u Update the source register RA with the calculated address after the load/store operation has been performed.

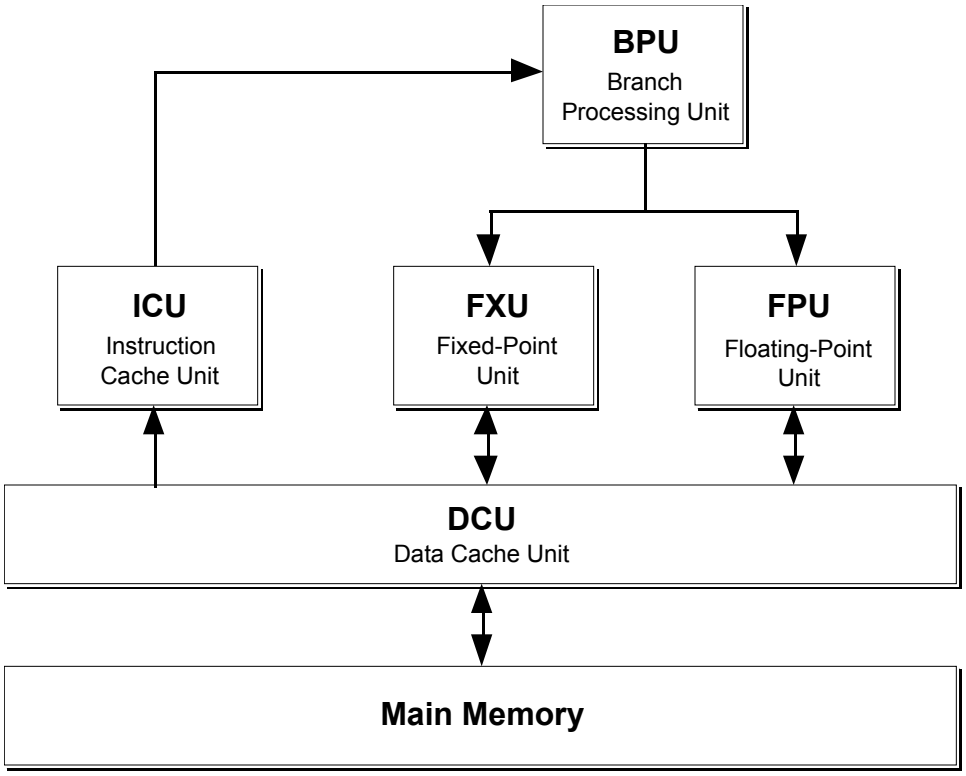
x Use indexed addressing (i.e. sum RA with RB) to calculate the effective address. If this is not specified, the default is Based Addressing.

Branch instructions may have the following modifiers:

l Save the address of the instruction immediately following the branch in the Link Register.

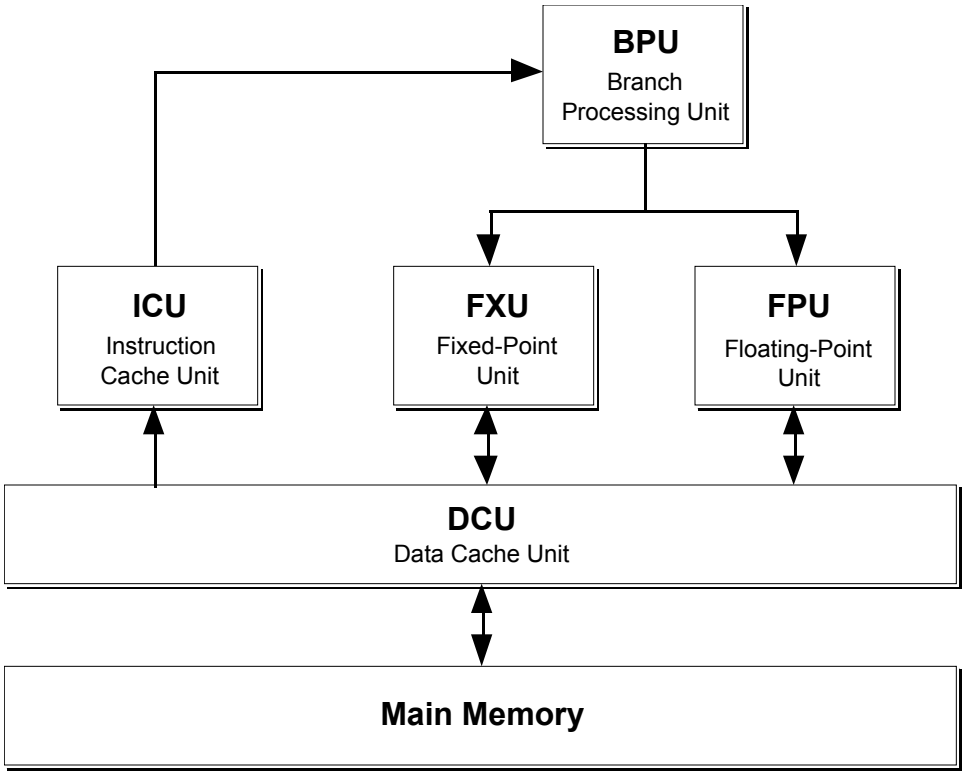
a Consider the specified address to be an absolute address, i.e. don't add the address of the branch to compute the branch target address.

As is evidenced by the size of Appendix A, the POWER instruction set is quite rich and is hardly a *reduced* instruction set. The acronym RISC has actually become somewhat of a misnomer. While many of the original RISC machines did



[Figure 2.1] Logical view of CPU functional units

have a reduced number of instructions (when compared with their CISC counterparts), the commonly accepted tenants of RISC architectures do not *require* that there be a small (or even reduced) number of instructions.



[Figure 2.1] Logical view of CPU functional units

4.0 Architecture Revisited

A more detailed look at the execution units is useful for understanding how the instructions execute and how they interact with each other.

4.1 The Instruction Pipeline

The POWER architecture defines a 8-stage overlapped pipeline which is divided between the BPU, FXU and FPU.

Stage 1: Instruction Fetch (IF) BPU

Instructions are fetched from the instruction cache and placed in the BPU's instruction buffers. The BPU has a Sequential Instruction Buffer (SeqIB: 8 words) and a Target Instruction Buffer (TarIB: 4 words). Instructions are loaded into the SeqIB and, if one of the first 5 instructions in the SeqIB is a branch instruction, instructions starting at the target of the branch are loaded into the TarIB.

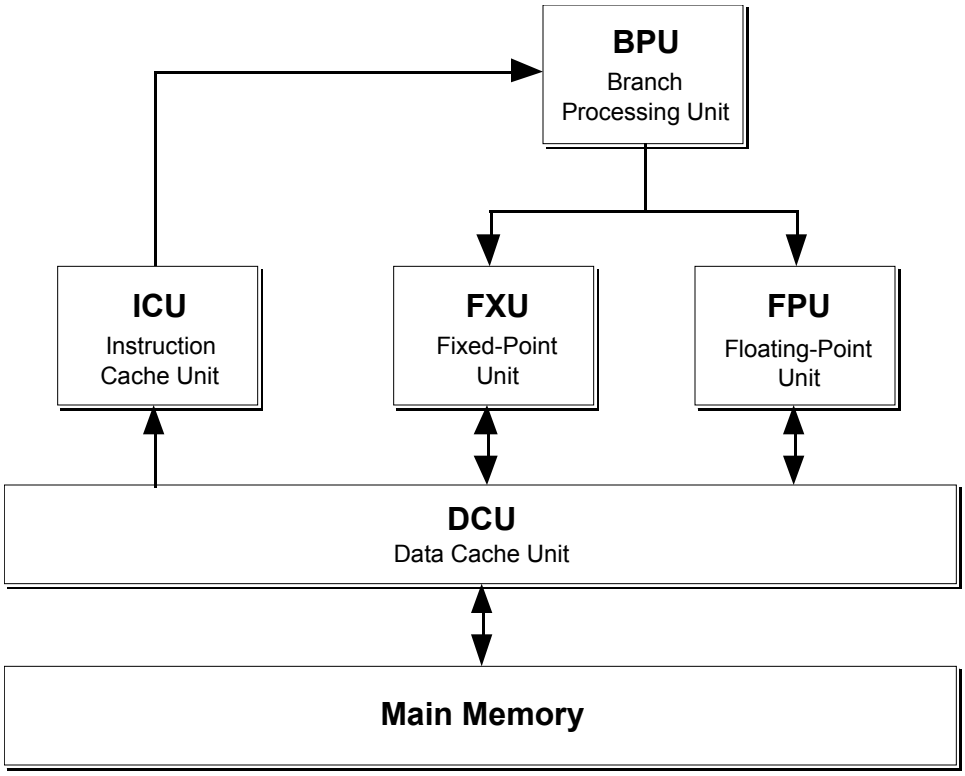
Stage 2: Instruction Dispatch and Branch Execute (IDBE) BPU

Branch and CR instructions are executed during this stage and all other instructions are sent to the next pipeline stage. Instructions which pass from this stage are sent to both stage 3X (decode in the FXU) and stage 3F (predecode in the FPU).

The IDBE stage can handle up to 4 instructions per cycle: dispatching 2 instructions to the arithmetic units and executing one branch and one CR instruction.

Stage 3X: Fixed-Point Decode (FXD) FXU

Instructions are decoded and either sent to the next stage (FXE) or discarded from the fixed-point pipeline. All fixed-point operations and all load/store operations (both fixed- and floating-point) are passed on to the next stage.



[Figure 2.1] Logical view of CPU functional units

Stage 4X: Fixed-Point Execute (FXE) FXU

Fixed-point instructions are executed and load/store operations have their addresses generated and searched for in the TLB[5†].

Stage 5X: Fixed-Point Data Cache Access (FXC) FXU

The data cache is accessed and the results are sent to either the fixed- or floating-point units. During this stage, register-register operations write their results to the register file.

Stage 6X: Fixed-Point Writeback (FXWB) FXU

The results of the (non register-register) instructions are written to the register file.

Stage 3F: Floating-Point Predecode (FPPD) FPU

Instructions are predecoded and either discarded or passed to stage 4F (FPRR). Loads and stores are passed through the pipeline so that the pipelines can be synchronized and so that the registers can be remapped and normalized (if necessary).

Stage 4F: Floating-Point Register Remap (FPRR) FPU

During this stage, the registers for floating-point operations (including loads and stores) are remapped. Remapping is discussed in detail in §4.3 Floating-Point Register Remapping.

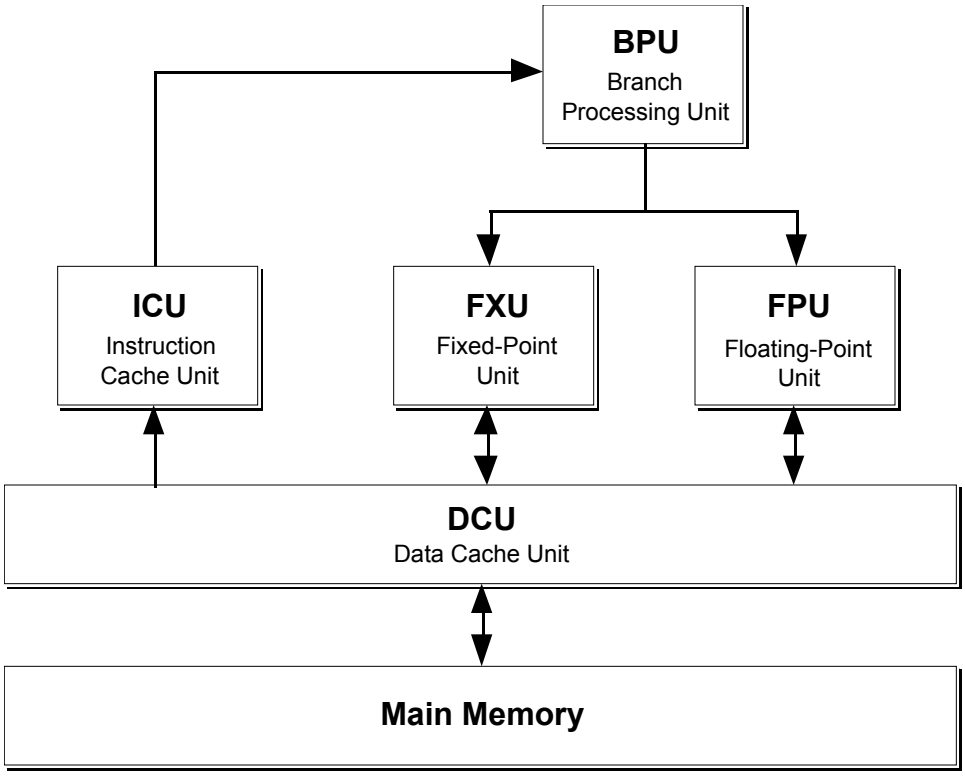
Stage 5F: Floating-Point Decode (FPD) FPU

The instruction is decoded and passed on to the execution phase of the pipeline if all of the required data is available.

Stage 6F: Floating-Point Execute 1 (FPE1) FPU

This is the first stage of the floating-point multiply-add pipeline.

[5†] A translation-lookaside buffer (TLB) is special cache which is used to store address translations when the page table get so large that it must be paged itself. (see [Patterson90])



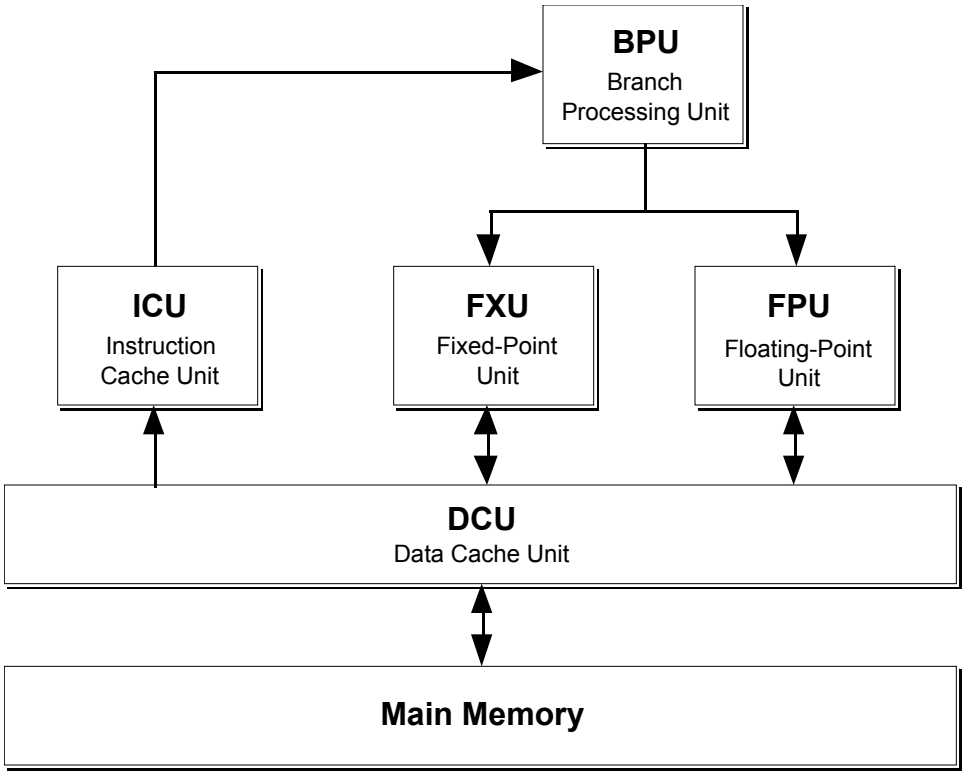
[Figure 2.1] Logical view of CPU functional units

Stage 7F:Floating-Point Execute 2 (FPE2)FPU

This is the second stage of the floating-point multiply-add pipeline. Instructions which depend on the results of this instruction and are currently in decode (Stage 5F) of the pipeline can get the results directly from the pipeline instead of waiting for writeback (Stage 8F) to complete.

Stage 8F:Floating-Point Writeback (FPWB)FPU

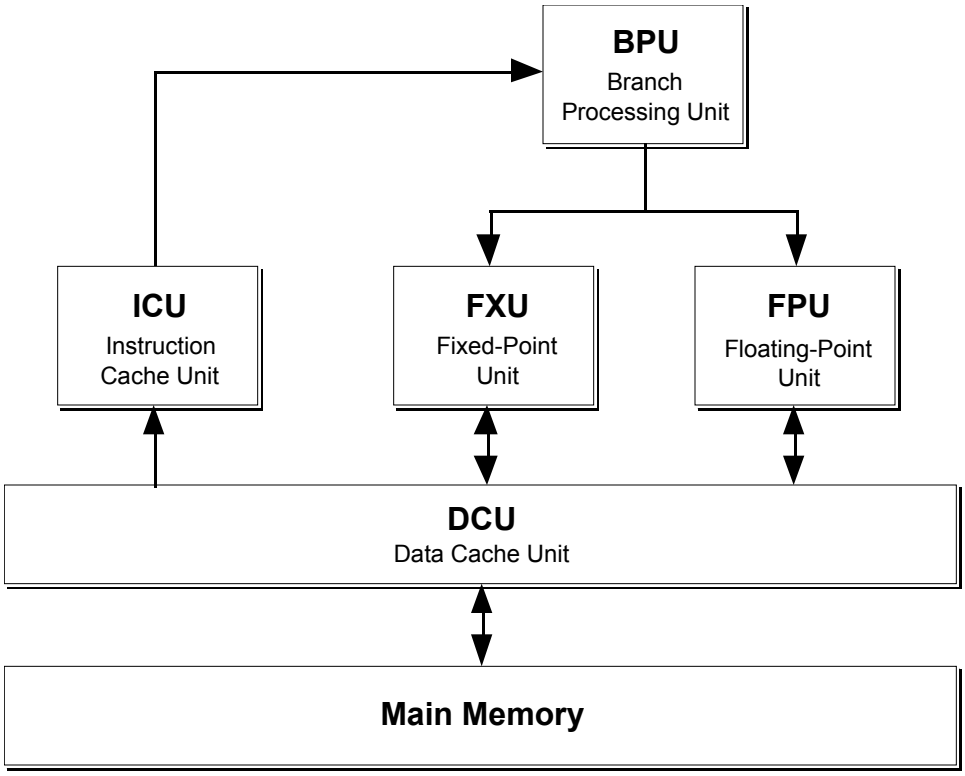
The results from Stage 7F are written into the register file.



[Figure 2.1] Logical view of CPU functional units

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
IF	lfdu f14 lfdu f16 fma f15 fms f17	fa f18 bdn		lfdu f14 lfdu f16 fma f15 fms f17	fa f18 bdn		lfdu f14 lfdu f16 fma f15 fms f17	fa f18 bdn
IDBE		lfdu f14 fma f15	lfdu f16 bdn fma f15	fa f18	lfdu f14 fma f15	lfdu f16 bdn fma f15	fa f18	lfdu f14 fma f15
FXD			lfdu f14,4(r13) fma f15,f14,f1,f2	lfdu f16,4(r14) fms f17,f16,f3,f4	fa f18,f15,f5	lfdu f14,4(r13) fma f15,f14,f1,f2	lfdu f16,4(r14) fms f17,f16,f3,f4	fa f18,f15,f5
FXE				lfdu f14,4(r13)	lfdu f16,4(r14)		lfdu f14,4(r13)	lfdu f16,4(r14)
FXC					lfdu f14,4(r13)	lfdu f16,4(r14)		lfdu f14,4(r13)
FXWB								
FPPD			lfdu f14,4(r13) fma f15,f14,f1,f2	lfdu f16,4(r14) fms f17,f16,f3,f4	fa f18,f15,f5	lfdu f14,4(r13) fma f15,f14,f1,f2	lfdu f16,4(r14) fms f17,f16,f3,f4	fa f18,f15,f5
FPRR				lfdu f14,4(r13) fma f15,f14,f1,f2	lfdu f16,4(r14) fms f17,f16,f3,f4	fa f18,f15,f5	lfdu f14,4(r13) fma f15,f14,f1,f2	lfdu f16,4(r14) fms f17,f16,f3,f4
FPD					fma f15,f14,f1,f2	fms f17,f16,f3,f4	fa f18,f15,f5	fma f15,f14,f1,f2
FPE1						fma f15,f14,f1,f2	fms f17,f16,f3,f4	fa f18,f15,f5
FPE2							fma f15,f14,f1,f2	fms f17,f16,f3,f4
FPWB								fma f15,f14,f1,f2

[Figure 4.1] The instructions from [Listing 4.1] as they flow through the pipeline.



[Figure 2.1] Logical view of CPU functional units

4.2 A Sample Instruction Stream

[Figure 4.1] shows how the instructions from the loop given in [Listing 4.1] flow through the pipeline.

```
@0:  lfdu    fr14,4(r13)
     fma     fr15,fr14,fr1,fr2
     lfdu    fr16,4(r14)
     fms     fr17,fr16,fr3,fr4
     fa      fr18,fr15,fr5
     bdn     @0
```

[Listing 4.1] A nonsensical sample loop

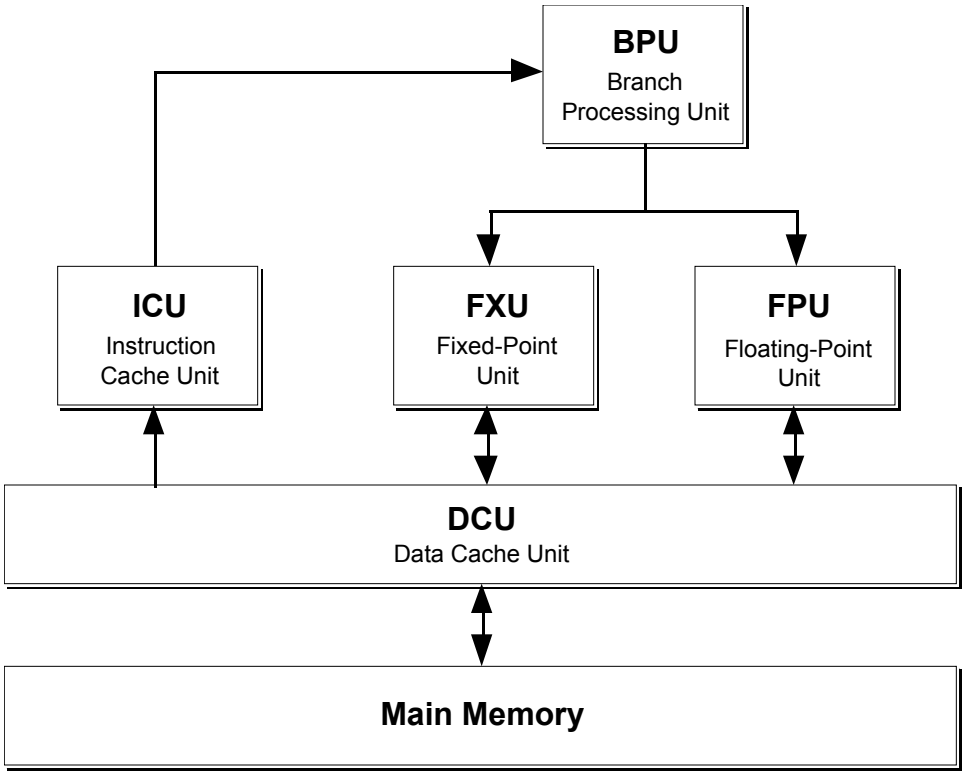
After the pipeline has filled with instructions, this loop requires 3 cycles per iteration to execute. The next few paragraphs will describe each cycle of the pipeline and the operations the processor units perform.

During the first cycle, the first four instructions from the loop are brought into the instruction buffer in the Branch Processing Unit.

The first two instructions from the buffer are dispatched to the FXU and the FPU during the second cycle, and the next four instructions are read into the BPU's buffer. Only 2 instructions are shown because the other 2 are discarded when the branch to the beginning of the loop is taken.

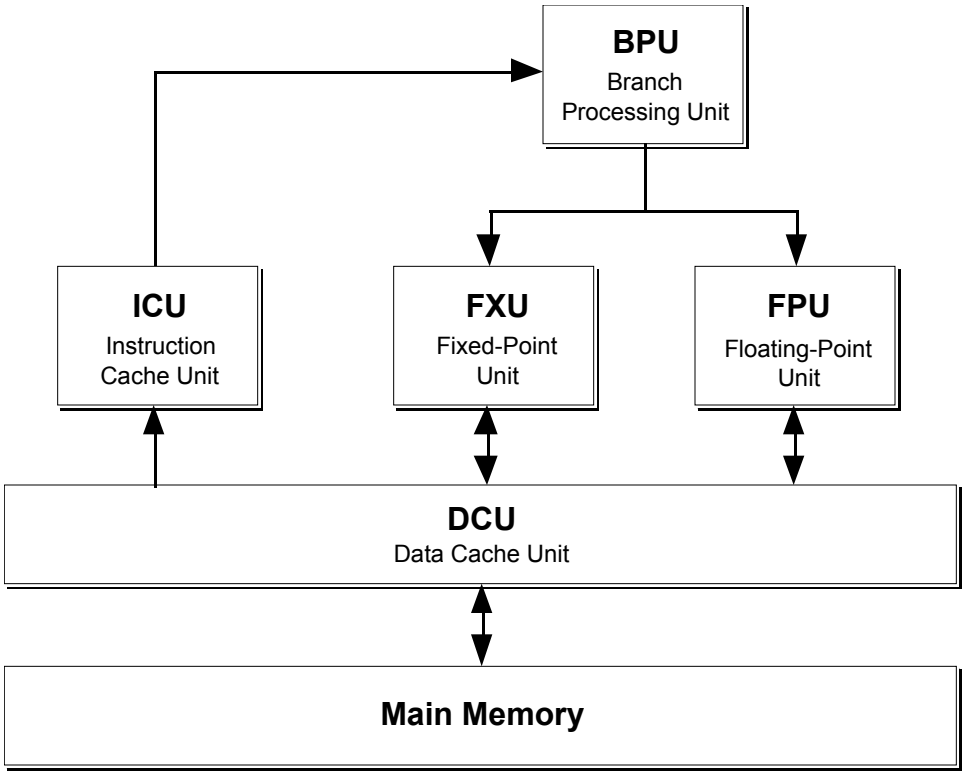
The FXU and FPU receive the 2 instructions from the BPU during the third cycle. The FXU discards the `fma` instruction. The BPU dispatches the next 2 instructions and executes the branch.

During the fourth cycle, the `lfdu` instruction has its address calculated in the FXU and has the floating-point register renamed in the FPU. The registers for the `fma` instruction are remapped. The 2 arithmetic units receive the next 2 instructions (`lfdu` & `fms`) and the FXU discards the `fms` instruction. The branch unit sends the `fa` instruction to the FXU and FPU. The IF stage of the BPU repeats from cycle 1.



[Figure 2.1] Logical view of CPU functional units

The `fma` instruction is in decode of the FPU during cycle 5. If there isn't a cache miss for the data coming back for the `lfdu`, then all of the data is available for this instruction and it passes into the



[Figure 2.1] Logical view of CPU functional units

first execute stage next cycle. During this time, the FPU remaps/renames the `lfdu` and `fms` instructions and predecodes the `fa`. Meanwhile, the FXU is busy getting the data back from the caches for the first `lfdu` instruction and calculating the address for the second. It also discards the `fa` instruction from it's pipeline. The IDBE stage of the BPU repeats from cycle 2.

Cycle 6 has the `fma` instruction executing and the `fms` instruction in decode. Again, if there isn't a cache miss, all of the data for the `fms` is available and it will be sent on to the execute stage for the next cycle. The `fa` instruction is being renamed in the FPRR stage. The FXU is getting the data for the second `lfdu` and sending it to the FPU. The FXE stage of the FXU is idle because there is no fixed-point instruction to execute. The FXWB stage is also idle because there is no result to write to the register store. The FXD stage in the FXU and the FPPD stage in the FPU repeat from cycle 3.

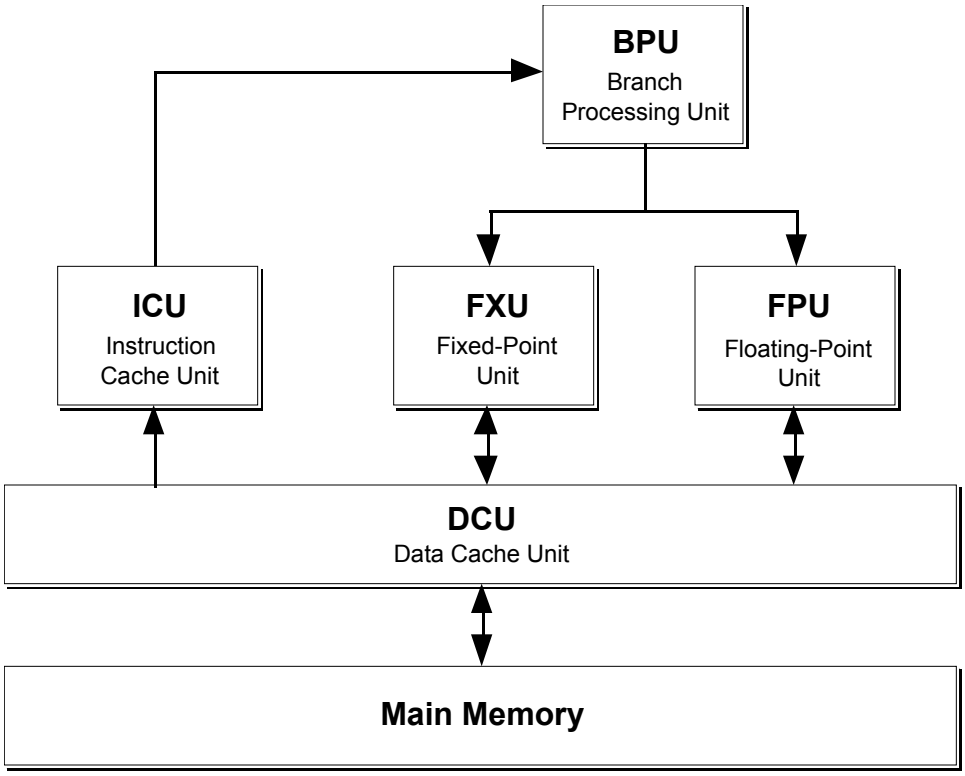
The `fma` and the `fms` instructions are in the floating-point execute stages during cycle 7. The `fa` instruction is in decode and will pass on to execute for the next cycle because `fr15` (the result of the `fma` in stage 7F) can be read directly from the pipeline instead of waiting until it is written to the register store. The FXE (in the FXU) and FPRR (in the FPU) stages are repeated from cycle 4.

The final cycle of this example has the `fma` writing its results to the floating-point register store and the `fms` and `fa` instructions in execute. The FXC and FPD stages are repeated from cycle 5.

Note that an extra 1-cycle fixed-point instruction can be added either before or after the `fa fr18,fr15,fr5` instruction and each loop iteration will still execute in 3 cycles. This can be clearly seen by the empty space in the FXE during cycle 6.

4.3 Floating-Point Register Remapping

Remapping is the process by which the 32 architected registers are mapped into the 38



[Figure 2.1] Logical view of CPU functional units

physical registers of the RS/6000. The remapping is done by looking up the physical register number associated with the architected register in the Map Table.

Initially there are 32 physical registers which are associated with architected registers and 6 physical registers which are “free”. These free registers are used whenever a floating-point load instruction is executed. A floating-point load causes the architected register to be renamed to one of the physical registers on the free list.

This renaming is useful because it divides accesses to a architected register into two physical registers: 1 before the load and 1 after the load. This allows floating-point operations to be overlapped while still allowing instructions before the load to access the old value. Once the load is finished executing, the physical register corresponding to the architected register before the load is added to the free list.

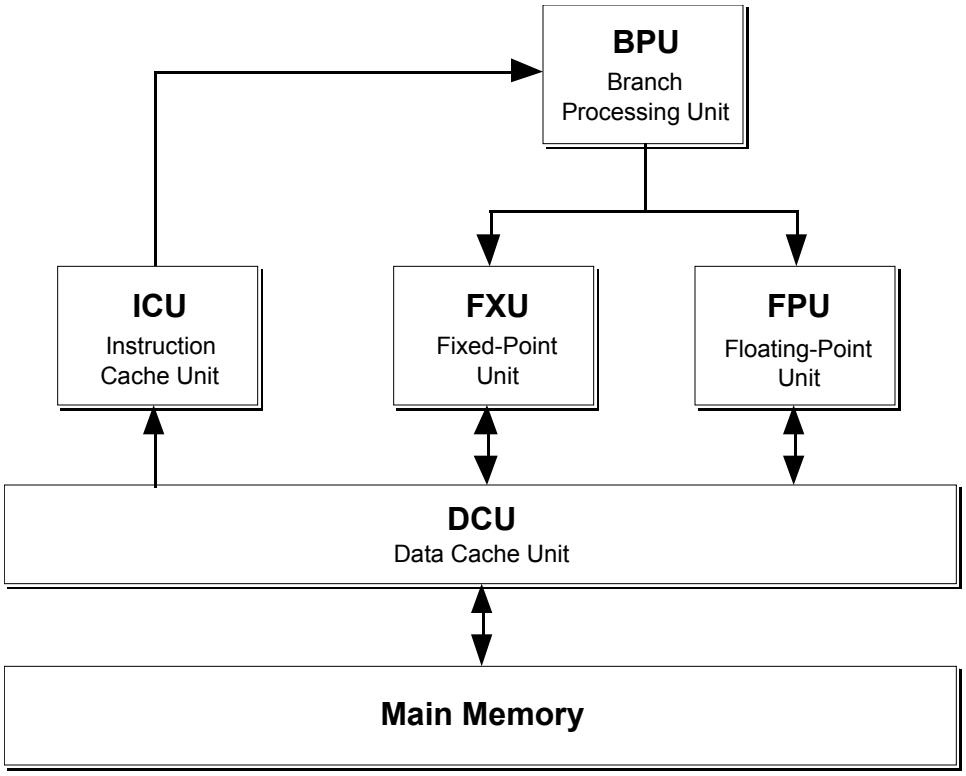
This can be demonstrated on the following code:

```
fa      fr14,fr15,fr16
lfd     fr14,...
fa      fr17,fr14,fr15
lfd     fr14,...
fa      fr18,fr14,fr16
```

This sample of code, while somewhat unrealistic, demonstrates how the remapping/renaming works. Initially, assuming that the Map Table is set to identity, the registers for the first `fa` are mapped into `pr14`, `pr15`, and `pr16` (where `pr` stands for physical register). The first `lfd` causes `fr14` to be renamed to a `pr` on the free list, in this case `pr32` (since `pr0 - pr31` are already allocated). This makes the remapping for the second `fa` become `pr17`, `pr32`, `pr15`. The second `lfd` causes another renaming for `fr14`, this time to `pr33`, which results in the mapping for the third fa being `pr18`, `pr33`, `pr16`.

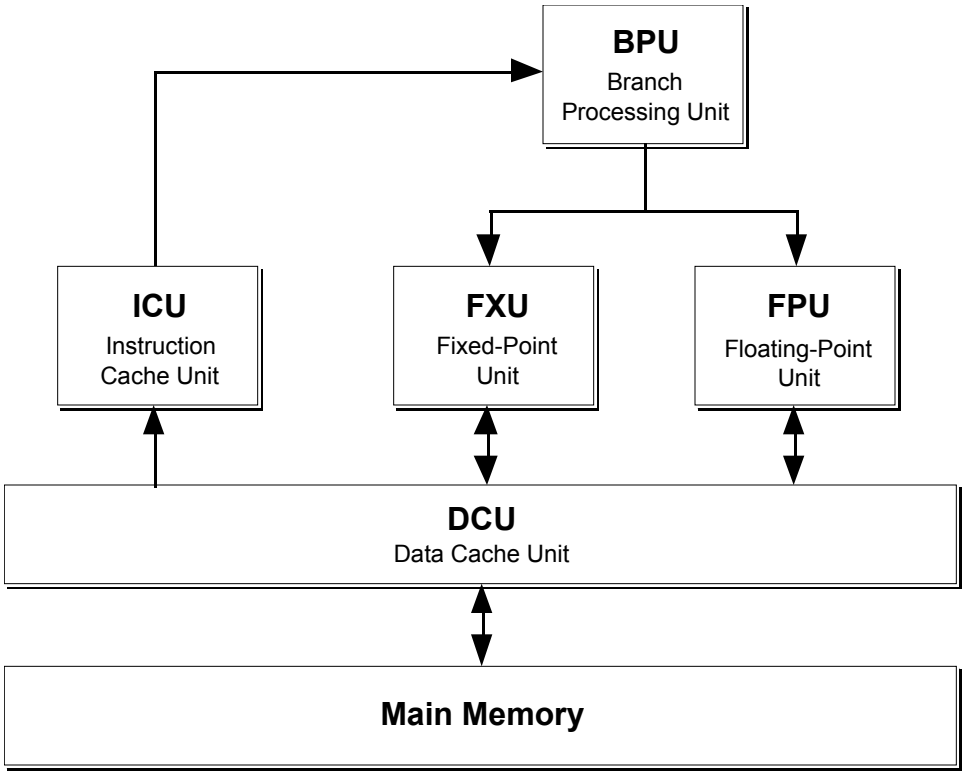
Basically, this code sample maps `fr14` into three different physical registers, which frees up the load and subsequent arithmetic operations from waiting until the `fr14` is no longer being used before loading its new value.

With the large number of floating-point registers,



[Figure 2.1] Logical view of CPU functional units

it might seem like this remapping is not very useful if registers are chosen intelligently. While this may be true to a degree, the remapping scheme is useful when there aren't enough registers to hold all of the values required by a computation and it also provides a mechanism by which more than 32 floating-point registers can be used in a system without changing the instruction coding.



[Figure 2.1] Logical view of CPU functional units

5.0 Basic Block Optimization Techniques

This section presents various techniques for improving the execution speed of a sequence of instructions within a basic block[6†]. There are many standard techniques (e.g.: common subexpression elimination, constant folding, et al.) which are not presented here because they can be applied to any architecture. This section focuses on those techniques which are specific to the POWER architecture.

These techniques are very important for assembly language programmers because one of the philosophies of RISC architectures is that the large amount of the processor resource scheduling is performed at the program level (as opposed to the microcode level). Failure to employ these techniques can result in programs which, while functionally correct, fail to make proper use of the CPU resources.

5.1 Instruction Mixing

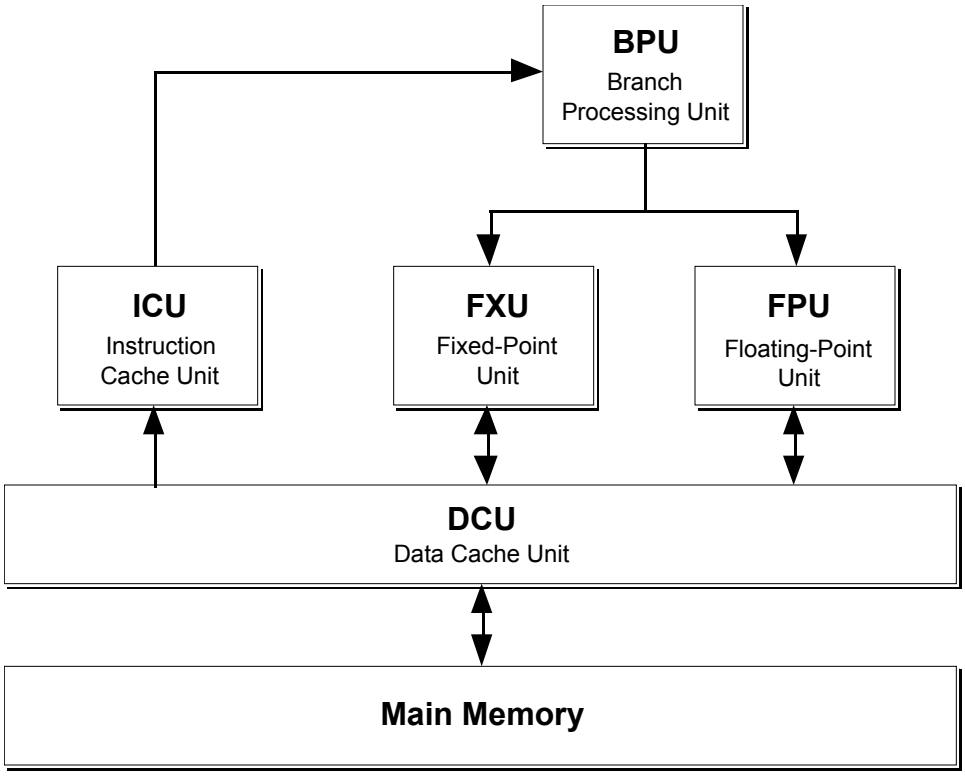
A good way to insure that full use is being made of the processor's resources is to have a good mix of fixed-point, floating-point and "branch related" instructions. The architecture was designed to execute multiple instructions in the same cycle, but only 1 instruction of each type can be executed at a time.

For instruction mixing concerns, there are 4 basic types of instructions: fixed-point, floating-point, branching, and CR-related. An ideal instruction mix would contain 1 of each instruction type for every 4 instructions.

In practice, it is rarely possible to achieve the above "ideal", because instruction streams are typically dominated by either fixed- or floating-point instructions, and CR-related instructions are used infrequently.

The worst situation is where the instruction stream is completely dominated by one

[6†] A basic block is a consecutive sequence of instructions where the flow of control enters only at the beginning and exits only at the end. (see [Aho86])



[Figure 2.1] Logical view of CPU functional units

instruction type. In this case, the other execution units are basically waiting idly.

Fortunately, the Fixed- and Floating-Point Units both have Instruction Prefetch Buffers (IPB's) between the BPU and the decode/predecode stages of their pipelines. These buffers can hold 6 instructions and allow the BPU to run ahead of the arithmetic units.

These buffers give the programmer some slack in arranging the instructions. In general, as long as the number of fixed-point instructions in a row is less than 6 and the number of floating-point instructions in a row is less than 3, there will be few, if any, wasted cycles in the other arithmetic unit.

Another bad situation is when the instruction stream contains a large number of CR-logic or branching instructions in a row. In this case, the BPU is too busy processing its own instructions to fetch and dispatch instructions to the other execution units.

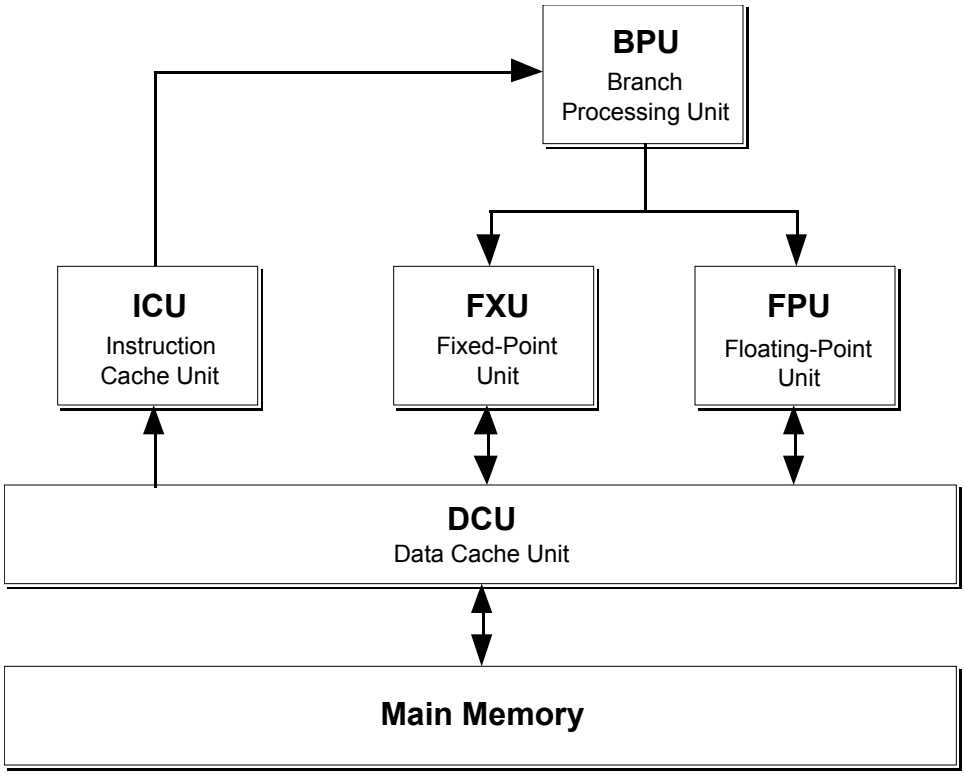
5.2 Multi-Cycle Instructions

Even though almost all of the instructions in the POWER instruction set "execute" in 1 cycle[7†], there are few instructions which do not. If possible, these instructions should have the delay covered by instructions in the other execution units.

The fixed-point multiplication instruction requires between 3 and 5 cycles in the FXE stage of the pipeline to complete execution. The number of cycles required is dependent on the multiplier for the operation. The multiplier is RB for `muls` and the immediate value for the `muli` instruction. If the multiplier is a signed-byte quantity (between -128 and 127), then the multiply will take 3 cycles; if it is a signed-halfword quantity (-32768 to 32767) it will require 4 cycles; and if it is a word quantity, it will require 5 cycles. The `mul` instruction always requires 5 cycles.

Fixed-point division always takes 19 cycles, and

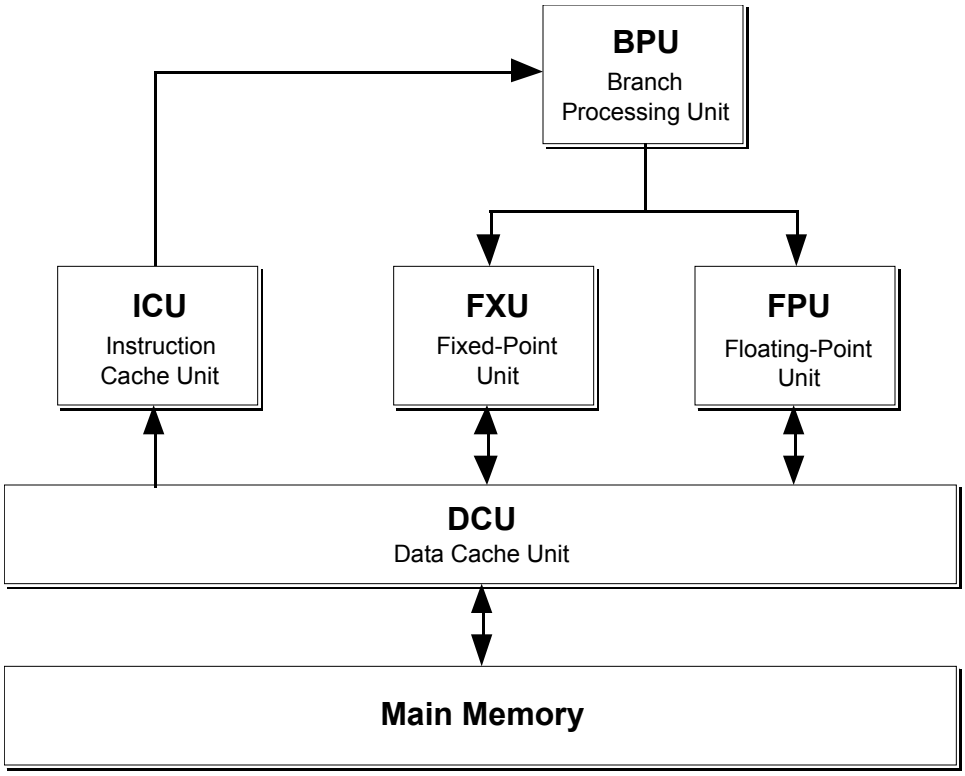
[7†] None of the instructions really execute in 1 cycle, but 1 instruction can be completed each cycle because of the pipeline.



[Figure 2.1] Logical view of CPU functional units

floating-point division takes between 16 and 19 cycles, depending on the current rounding mode. Division is discussed in more detail in §5.9 Avoid Division.

The load and store multiple instructions and the string operations requires multiple cycles depending on the number of words accessed. The `lm` and `stm` instructions require 1 cycle per register moved. The load and store string instructions require 1 cycle for



[Figure 2.1] Logical view of CPU functional units

each word of memory referenced. And finally, the load string and compare byte (`lscbx`) instruction requires 2 cycles plus 1 cycle for each word of memory referenced plus 2 cycles for each cache line crossed.

5.3 Data Alignment

Since misaligned data is generally handled by an interrupt, there are large penalties associated with data that is not aligned properly. Even if the misaligned data is not handled by an interrupt, the penalty will be at least 1 additional cycle. The penalty associated with the interrupt handler is roughly 70 cycles plus about 10 cycles per word accessed.

Data alignment needs to be a concern even when programming in high-level languages. A compiler may take an array of 6-byte structures and round each element up to 8-bytes so that each element is aligned. Be wary of this if you increment pointers yourself – the compiler may still return 6 as the size of the structure (e.g.: with `sizeof()` in C).

On the other hand, if the compiler does not round up to the next word boundary for structures, there is the potential for alignment penalties even if the elements are aligned within the structure. It is generally best to add padding bytes yourself to make sure that the structures and structure fields will be aligned.

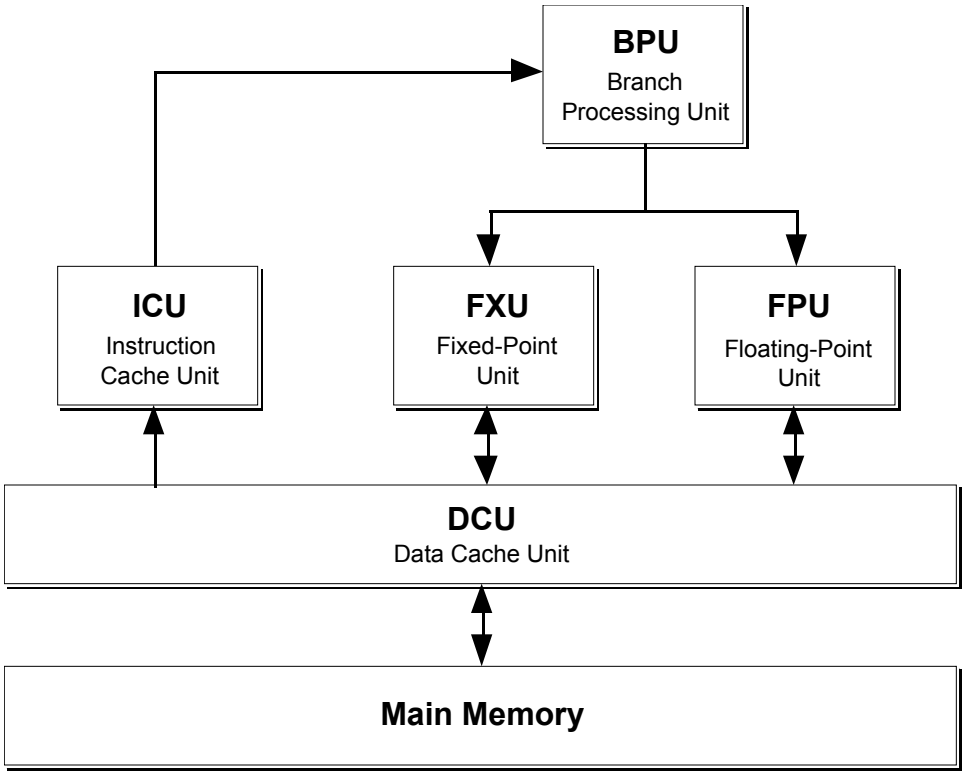
5.4 Load-Use Delay

There is a 1-cycle delay between when a value is loaded into a fixed-point register and when that value is available for use. This delay can be covered by inserting an independent instruction between the load and the use.

Because floating-point loads are executed by the FXU, and the FXU pipeline typically runs ahead of the FPU pipeline, there is not usually a delay between loading a floating-point register and using the new value.

5.5 Store-Load-Use Delay

If the load is preceded by a store to the same



[Figure 2.1] Logical view of CPU functional units

word that doesn't have time to complete, the load will be delayed until the store is completed. This delay is typically between 2 to 4 cycles.

Care must be taken when writing string routines since if they are not written carefully, they may access each word four times incurring multiple store-load-use penalties. E.g.: a routine which performs an inline conversion of a length-encoded string to a null-terminated string should NOT be written as:

```
len = str[0];
for(i=0;i<len;i++)
     str[i] = str[i+1];
str[i] = '\0';
```

5.6 Floating-Point Set-Use

There is a delay of at least one cycle between a floating-point instruction which sets a value, and the first use of the newly computed value. This delay can be covered by inserting an independent floating-point instruction between the set and the use. The delay has been reduced to only 1 cycle because of the bypass pathway between stage 11 (FPE2) and stage 9 (FPD) of the floating-point pipeline.

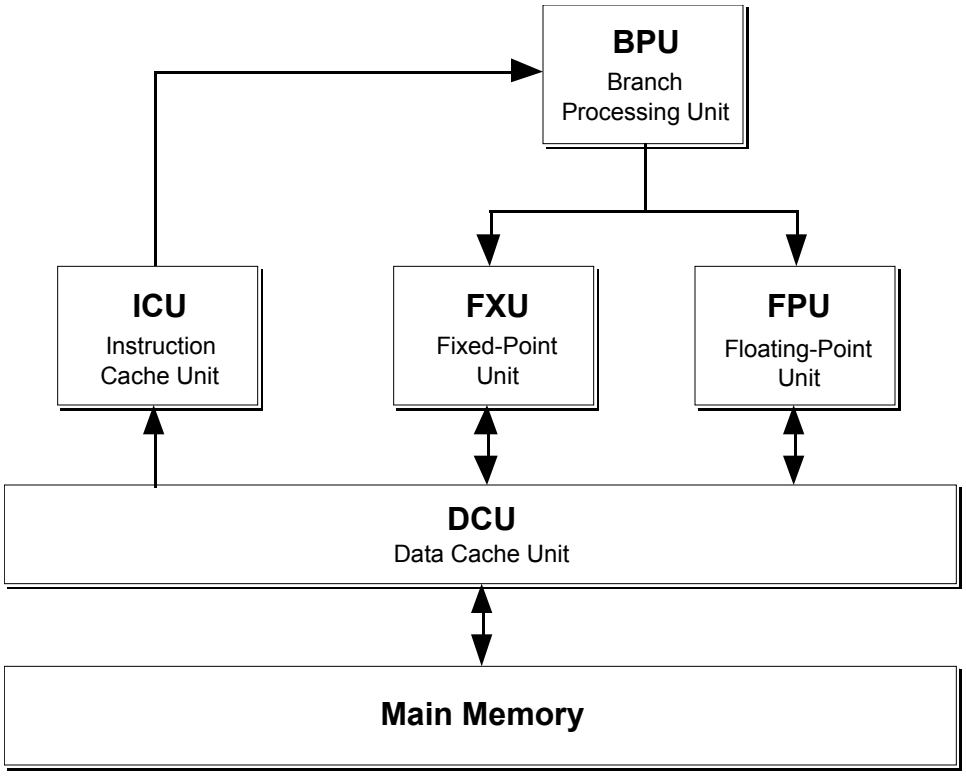
This bypass pathway, however, cannot feed the FRA position of the instruction in stage 9. If the floating-point set feeds the FRA position of the second instruction, the delay is 2 cycles.

With most floating-point instructions, the FRA register can be exchanged with either FRB or FRC without changing the meaning of the instruction. With the two floating-point instructions that are not commutative (`fd[.]` and `fs[.]`) this cannot be done and the delay must be covered with 2 instructions.

5.7 Condition Register

When an instruction that sets a field of the Condition Register is encountered by the BPU, it locks the field until its value can be updated. An instruction which relies on this CR field, must wait until the field is unlocked before it can access the proper value.

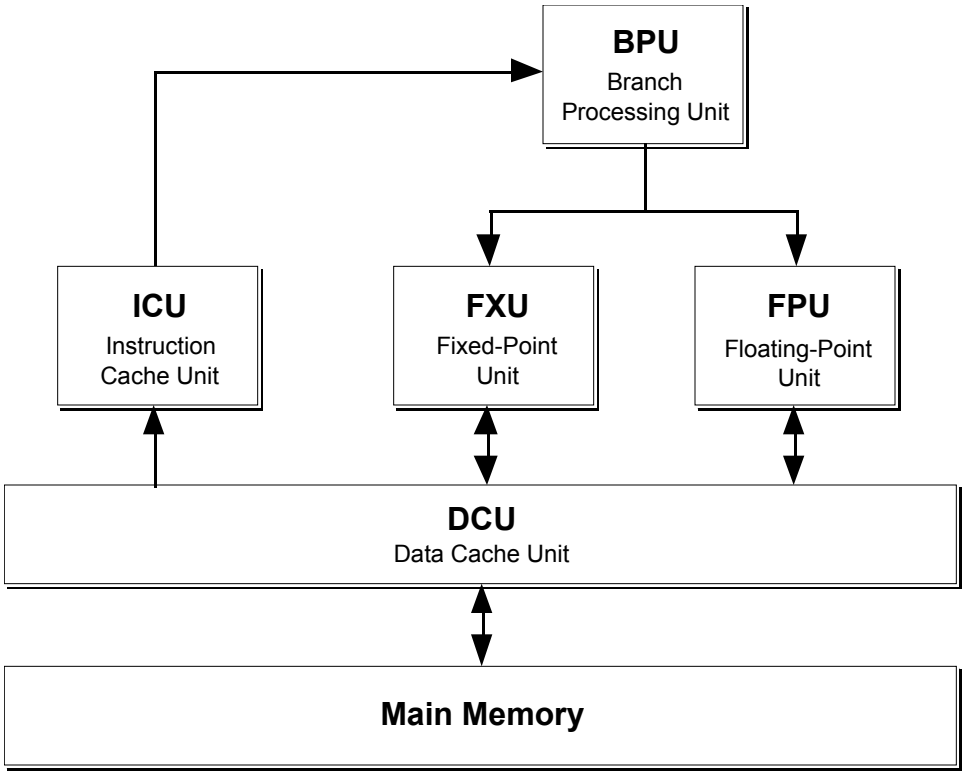
While a CR field is locked, no other instruction



[Figure 2.1] Logical view of CPU functional units

which sets that field can be executed, and the pipeline will stall in the BPU if it encounters another CR-setting instruction for that field.

Because many instructions can implicitly set CR field 0 or 1, indiscriminate use of the Record Bit (by



[Figure 2.1] Logical view of CPU functional units

adding a '.' after the mnemonic) can lead to a 2-cycle delay between instructions. E.g.:

```
a.      r28,r29,r30
si.     r31,r31,20
cmp     cr0,r27,r26
```

There is a 2-cycle delay between each of these instructions even though they have no register interdependencies. A simple rule to avoid this is never to set the Record Bit of an instruction unless the results are needed.

Even though there are 8 CR fields, the BPU has only 4 internal lock bits. As a result of this, the CR fields are paired so that each lock bit controls 2 fields of the CR. These field pairs are as follows: (0,4) (1,5) (2,6) (3,7)[8†].

This means that with the above example, replacing CR field 0 with CR field 4 would still incur the same delay between the `si.` and the `cmp`. Whereas using field 1, 2, 3, 5, 6, or 7 would eliminate it.

5.8 Delayed-CR Instructions

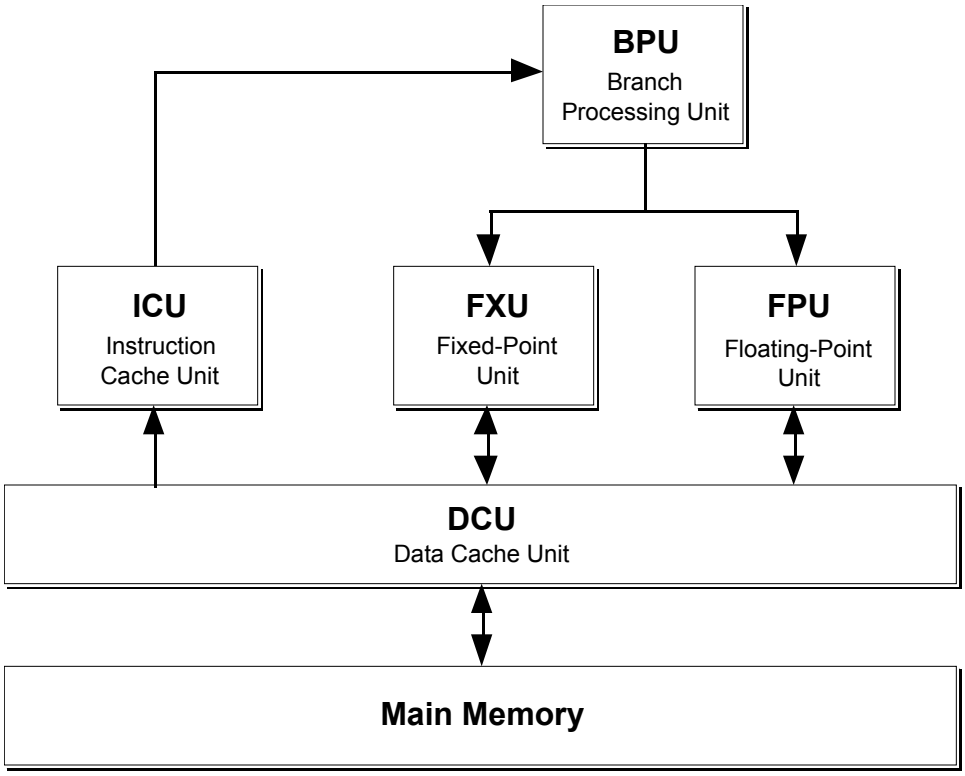
Some instructions which set the Condition Register are known as delayed-CR instructions because they require an additional cycle to send the new CR value back to the BPU. When using these instructions, it is important to place an instruction between the CR loading instruction and the instruction which requires that value or else there will be a pipeline stall.

The delayed-CR instructions are marked in Appendix A with a '•'. All other CR loading instructions are normal and do not require this extra cycle before the updated CR value can be accessed.

5.9 Avoid Division

Division is a slow operation because it is performed by calculating the reciprocal of the

[8†] This "pairing" is required because of timing problems and technology limitations. Future processors may not have this restriction.



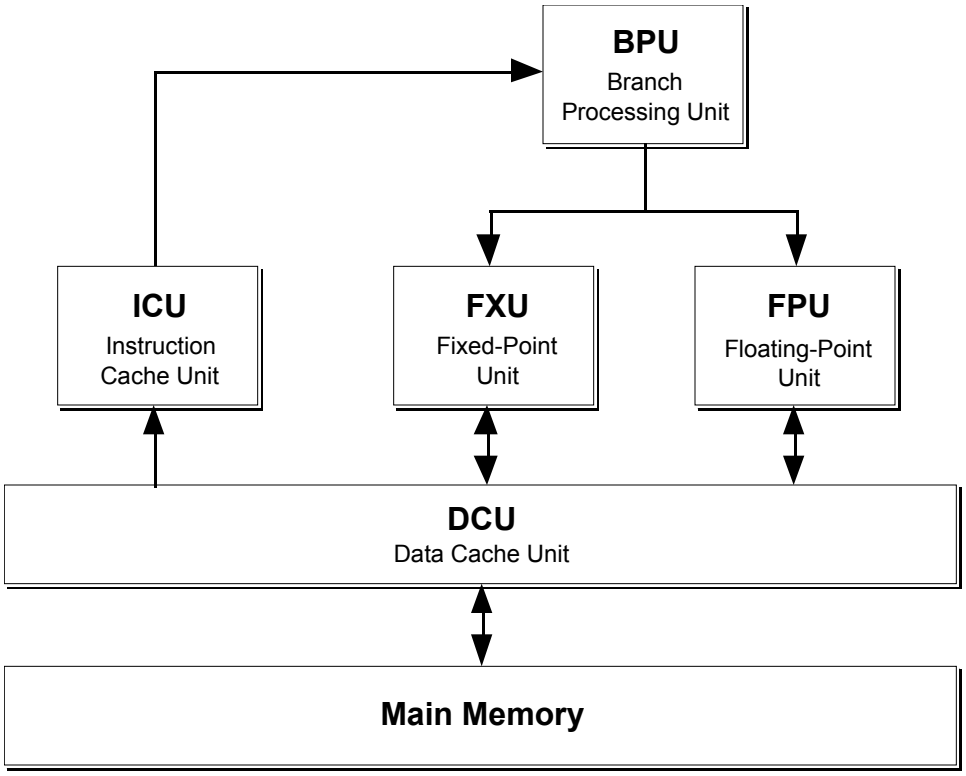
[Figure 2.1] Logical view of CPU functional units

divisor and then multiplying the result by the dividend[9‡] . The multiplication is not a problem because multiplication is a relatively fast operation. The reciprocal, however, is calculated by iteratively applying Newton-Raphson's method, after seeding it with an initial value from a table.

The end result is that division takes between 16 to 19 cycles to execute: 19 cycles for fixed-point divide and floating-point divide when in Round-to-Nearest mode, and 16 cycles for floating-point divides when using any of the other three rounding modes.

The following code demonstrates an instance where performance can be significantly improved by using the `fma` instruction to replace a division operation with a multiplication by the reciprocal, so that the division can be moved outside a loop. This is the code in C:

```
float x[50];
int d;

for(i=0;i<50;i++)
     x[i] /= d;
```



[Figure 2.1] Logical view of CPU functional units

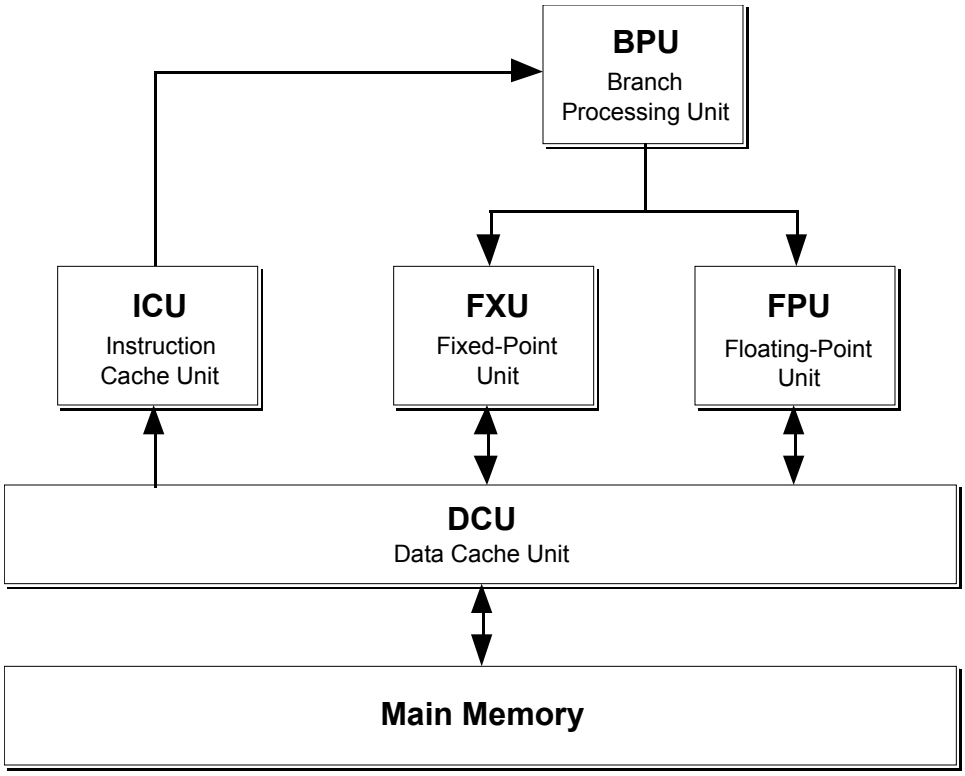
And this is the assembly code before any division specific optimizations have been applied:

```
# assume that r31 points to the
# start of the array and that fr30
# has been loaded with the value d
     # load the CTR reg with the #
     # of times we should iterate
     lil     r30,50
     mtctr   r30
     # offset ptr to array so that
     # we can use load-update
     ai      r31,r31,-4
@0:  # load x[i] into fr31
     lfs     fr31,4(r31)
     # x[i] /= d
     fd      fr31,fr31,fr30
     # save x[i]
     stfsu   fr31,4(r31)
     # dec CTR and branch if !0
     bdn     @0
```

If the division does not need to be exact (i.e.: it can tolerate round-off errors), the body of the loop can be replaced with:

```
     # calculate the reciprocal
     fd      fr29,1.0,fr30
@0:  # load x[i] into fr31
     lfs     fr31,4(r31)
     # x[i] *= (1/d)
     fm      fr31,fr29,fr31
```

[9‡] This is a common technique used in many RISC processors and super-computers.



[Figure 2.1] Logical view of CPU functional units


```
        # save x[i]
        stfsu   fr31,4(r31)
        # dec CTR and branch if !0
        bdn     @0
```

This precomputes the reciprocal and then multiplies by the reciprocal instead of dividing. This moves the 19-cycle divide operation outside the loop and replaces it with a 1-cycle (assuming that the pipeline is full) multiply operation.

But the properties of the `fma` instruction can be exploited to produce code which is functionally equivalent to the above division code and does not introduce round-off errors.

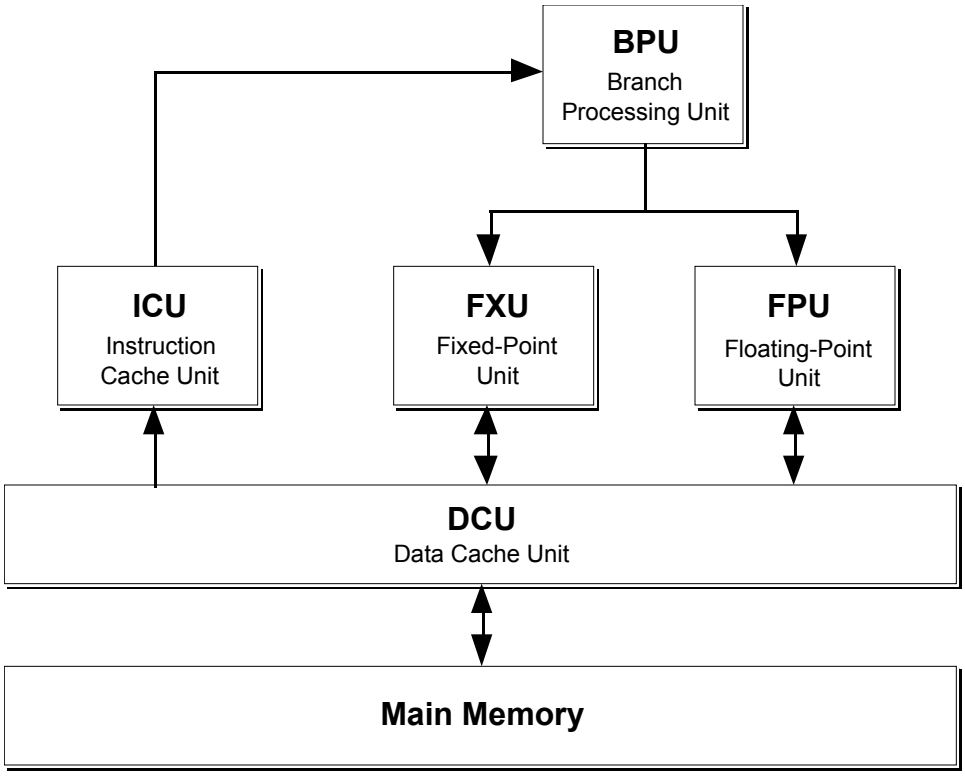
```
        # calculate the reciprocal
        fd      fr29,1.0,fr30
@0:     # load x[i] into fr31
        lfs     fr31,4(r31)
        # temp = x[i] * (1/d)
        fm      fr28,fr31,fr29
        # remainder = temp*d - x[i]
        fms     fr27,fr30,fr28,fr31
        # x[i] = temp + (rem * 1/d))
        fma     fr31,fr29,fr27,fr28
        # save x[i]
        stfsu   fr31,4(r31)
        # dec CTR and branch if !0
        bdn     @0
```

Because of register dependencies, this calculation requires 5 cycles per iteration: 1 (`fm`) + 2 (`fms`) + 2 (`fma`). This code still multiplies by the precomputed reciprocal, but it also calculates an error term and adjusts the result. This adjustment works because the `fma` and `fms` instructions are integral operations which do not perform rounding on the intermediate results.

6.0 Branching Optimization Techniques

The POWER architecture defines a separate branch processor that is very efficient at handling program branches which are structured the proper way. Poorly structured branches, however, can cause up to a 3-cycle delay per branch.

This section begins with a short discussion of how the BPU handles branches and then continues with two additional subsections, each



[Figure 2.1] Logical view of CPU functional units

of which deals with one of the two most common types of branching constructs: loop-continuation and if-then-else branching. Each of these subsections presents common branching structures and discusses the relative strengths and weaknesses of each.

6.1 How the BPU Handles Branches

Whenever the BPU comes across a conditional branch, it fetches the instructions at the target address, and sends the sequential[10†] instructions on to the FXU and FPU. These instructions are sent conditionally, meaning that the arithmetic units know that these instructions may have to be discarded if the branch is taken.

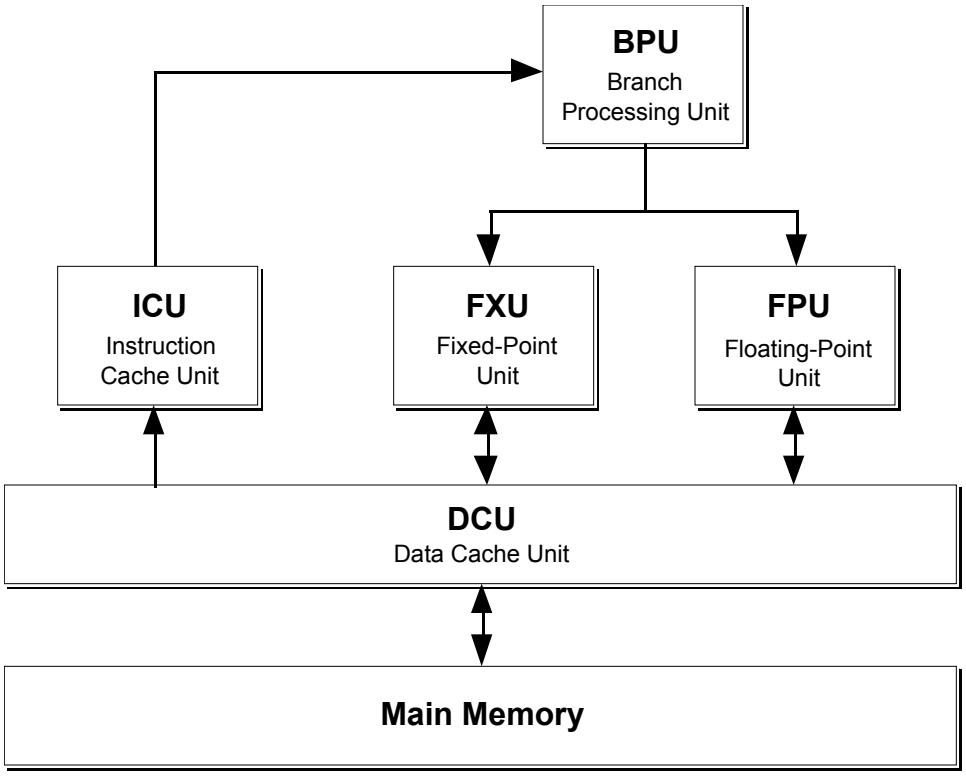
This means that if the conditional branch falls through (i.e.: the branch is not taken), the BPU has done the right thing by sending the sequential instructions to the arithmetic units. It only needs to inform the FXU and FPU that the instructions are no longer conditional and will not be discarded. However, if the branch is taken, the BPU has sent the wrong instructions and must cancel them and send the correct instructions. The potential delay is minimized because the BPU has already fetched the target instructions and only has to start sending them to the arithmetic units.

This delay to refill the pipelines with useful instructions takes 3 cycles (assuming no cache misses).

A useful feature of the BPU is that it calculates the destination of the branch as soon as the condition upon which the branch is dependent has been resolved. This means that by placing at least 3 independent instructions between the instruction setting the condition and the branch dependent on that condition, the programmer can insure that the branch will be resolved before the BPU starts sending post-branch instructions to the arithmetic units.

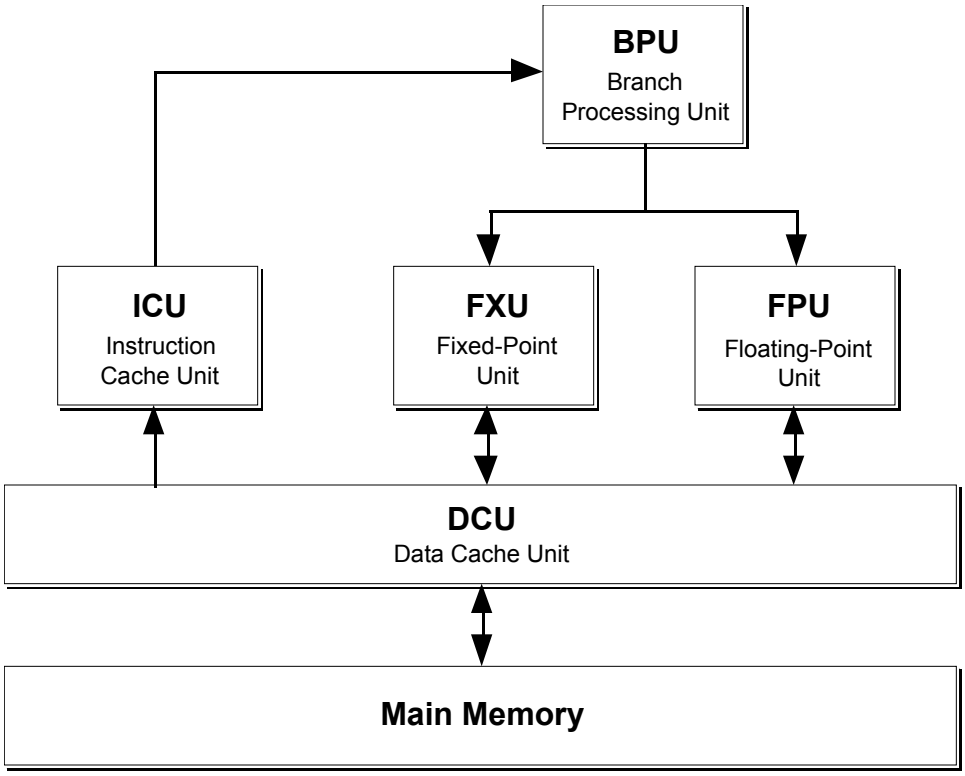
6.2 Common Loop-Continuation Structures

[10†] The target address points to the instruction that should be executed if the branch is taken. The sequential instruction is the instruction immediately following the branch, i.e. the instruction that should be executed if the branch is not taken.



[Figure 2.1] Logical view of CPU functional units

One of the better ways to write a loop in POWER assembly is to make use of the Count Register (CTR). The CTR is part of the BPU and thus, the branch processor always has ready access to the contents of this register.



[Figure 2.1] Logical view of CPU functional units

A standard loop which uses the CTR loads the number of iterations into the CTR before the loop starts, and branches to the beginning of the loop using one of the `dbzXX` or `dbnXX` instructions[11†]. These instructions decrement the CTR and branch if it is zero (`dbzXX`) or non-zero (`dbnXX`) or if the given condition (specified as `XX`) is true. This can be shown as follows:

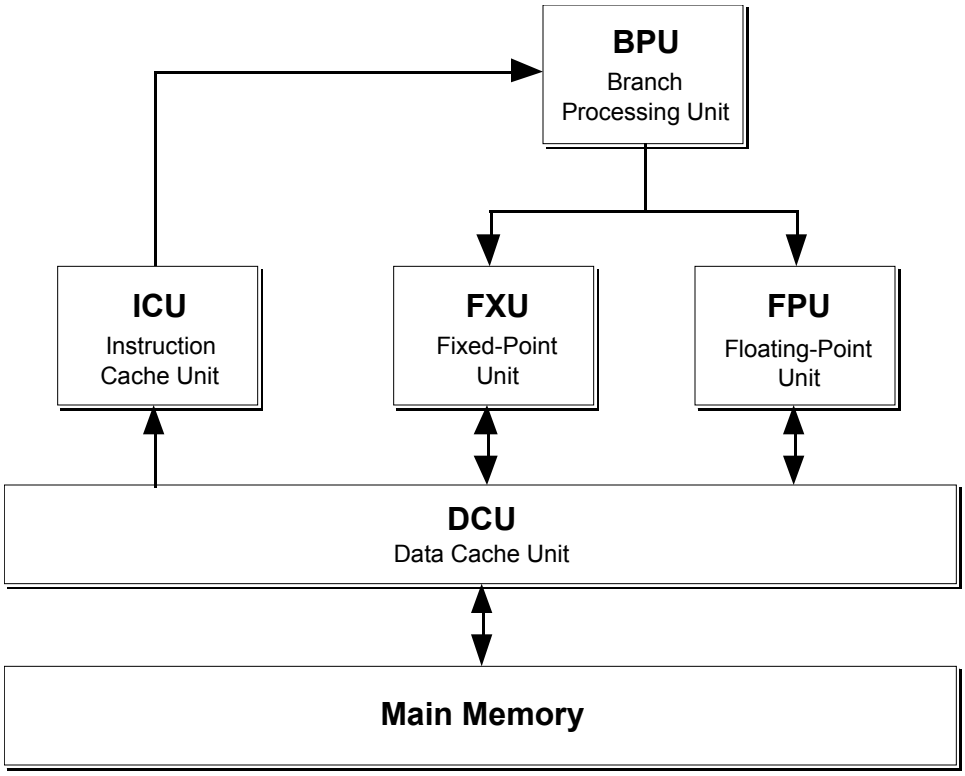
```
      mtctr   r0        # ctr = r0
@0:   xxx
      xxx
      dbn     @0
```

where the *xxx*'s represent the body of the loop (this convention of using "`xxx`" to represent the loop body will be continued throughout this subsection).

The rest of this subsection will discuss loops which either do not make use of the Count Register or which rely on a condition being set in addition to the value of the CTR.

A Standard Loop

When iterating through a loop, the loop-continuation structure determines whether the flow of control passes to the beginning of the



[Figure 2.1] Logical view of CPU functional units

loop, or to the next statement after the loop. Of these two potential statement-flow changes, it is important that the return to the beginning of the loop be more efficient than branch to the next instruction after the loop. This is based on the assumption that the loop is going to be executed more than once.

A standard way of structuring a loop is:

```
@0:  xxx
     xxx
     cmp     cr0,...
     bXXc    cr0,@0
     yyy
```

This has the body of the loop followed by the test to exit the loop followed by the branch back to the beginning of the loop. Unfortunately, this type of loop has characteristics which are exactly the opposite of what we want.

The comparison occurs immediately before the branch, thus the branch condition can't be determined before the BPU needs to start sending instructions which occur after the branch. By default, the BPU assumes that the branch will fall through – a poor assumption for a loop – and starts conditionally sending instructions starting at *yyy*. In most cases, these instructions will need to be discarded when the loop is repeated.

How to Make it Less Efficient

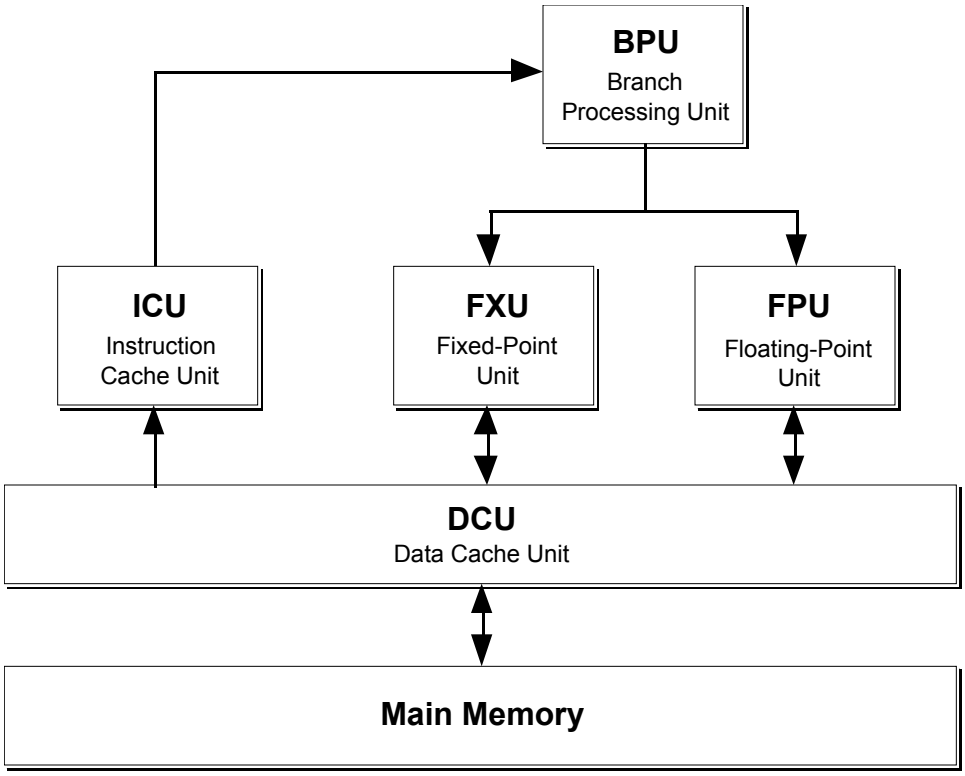
One way or rewriting this loop as an attempt to improve it is:

```
@0:  xxx
     xxx
     cmp     cr0,...
     ¬bXXc   cr0,@1
     b       @0
@1:  yyy
```

where the ¬bXXc is branch with the negation of the original branching condition.

However, because the BPU cannot handle branch instructions (including CR-related instructions)

[11]† These instructions are extended mnemonics for the `bc`, `bcc` and `bcr` branch instructions.



[Figure 2.1] Logical view of CPU functional units

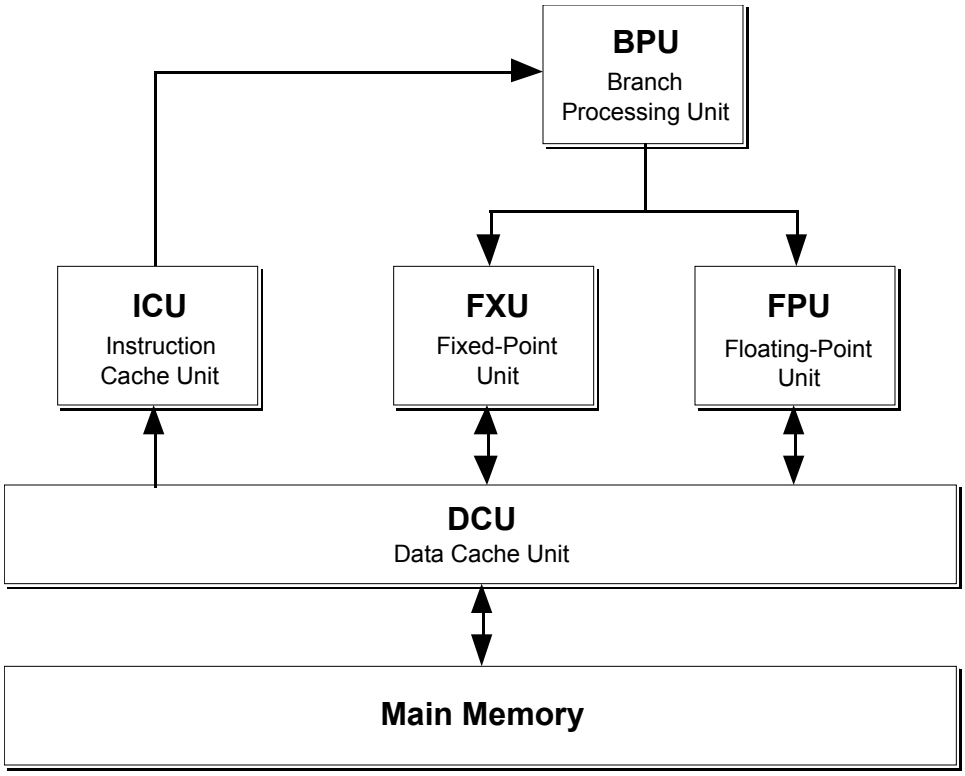
while it is conditionally dispatching instructions from a previous branch, the BPU has an additional stall at the `b` instruction. Note that this stall occurs even for the fall-though case of the `¬bXXc` instruction, so the overall effect is that this code always executes less efficiently than the first loop.

Covering the Delay from the Loop Body

A method which can eliminate the delay is to move instructions from the body of the loop between the comparison and the branch. If three independent instructions can be inserted before the branch, the branch will be resolved and the appropriate instructions will be dispatched without any delay.

```
@0:  xxx
     cmp     cr0,...
     xxx               # 3 indep.
     bXXc    cr0,@0
     yyy
```

This method, unfortunately, cannot always be applied to loops because it is not always possible to find 3 independent instructions to place after the comparison.



[Figure 2.1] Logical view of CPU functional units

Test Condition at Beginning of Loop

Another method which typically eliminates the delay and can always be applied is as follows:

```
      b       @1
@0:  ¬bXXc    cr0,@2
@1:  xxx
     xxx
     cmp      cr0,...
     b        @0
@2:  yyy
```

This can be considered the same as the second looping example with the entire body of the loop inserted between the `¬bXXc` and the `b` instructions and the `cmp` moved before the `b`. Note that this will eliminate the delay only if the body of the loop is at least 2 instructions (to cover the delay between each execution of the `cmp` instruction).

Five-Instruction Loops

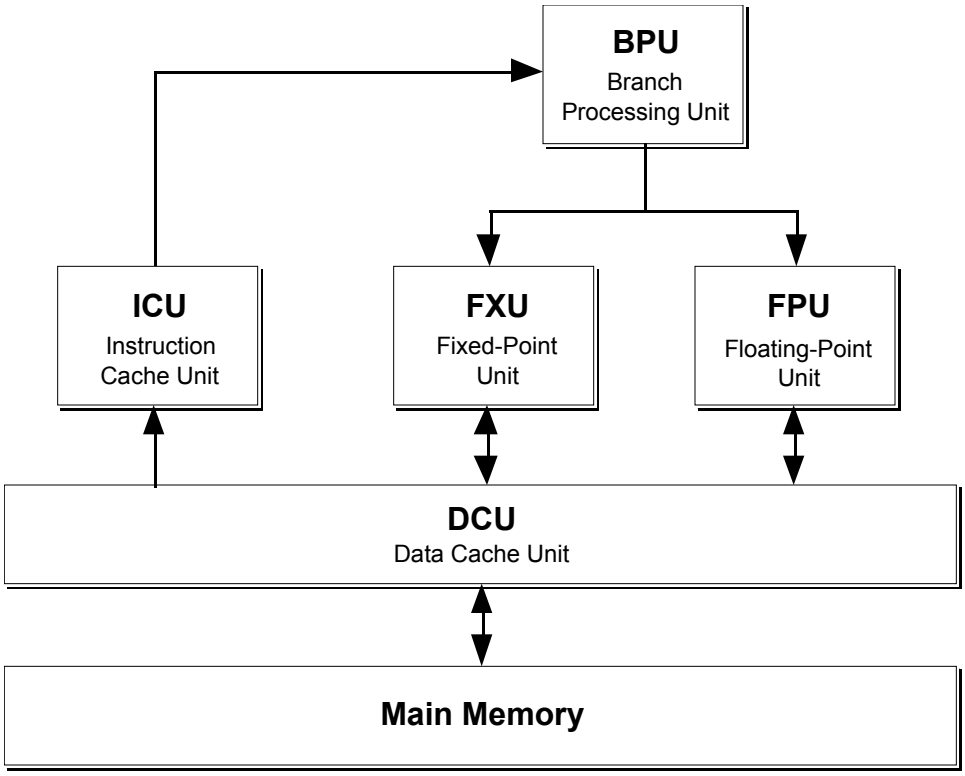
A special case which should be discussed is a loop structure which contains exactly 5 instructions: 4 instructions and a loop closing branch. In this situation, the loop will require at least 3 cycles per iteration due to limitations within the BPU.

During the first cycle, the BPU will fetch the 4 instructions from the loop body. During the second cycle, the BPU will fetch the loop closing branch and the three instructions which follow it (these instructions will be discarded for each iteration except for the last). The branch is detected on the third cycle and the target address is calculated so that, on the fourth cycle, it can fetch the 4 instructions from the loop body, etc.

This situation can be easily avoided by unrolling the loop.

6.3 Common *If-Then-Else* Structures

If-then-else constructs are very common structures which can be manipulated considerably to produce more efficient code. A standard method of implementing an if-then-else sequence is:



[Figure 2.1] Logical view of CPU functional units


```
      cmp     cr0,...
      bXXc    cr0,@0
      # then-clause
      xxx
      b       @1
@0:   # else-clause
      yyy
@1:   zzz
```

```
      cmp     cr0,...
      ¬bXXc   cr0,@0
      # old else-clause
      yyy
      b       @1
@0:   # old then-clause
      xxx
@1:   zzz
```

This code works well in many cases, but is sub-optimal in the situations when ① the then-clause is less than 4 instructions long and thus cannot completely cover the `cmp-bXXc-b` delay, or ② the then-clause is less likely to be chosen than the else-clause.

Swapping the Then- and Else-Clauses

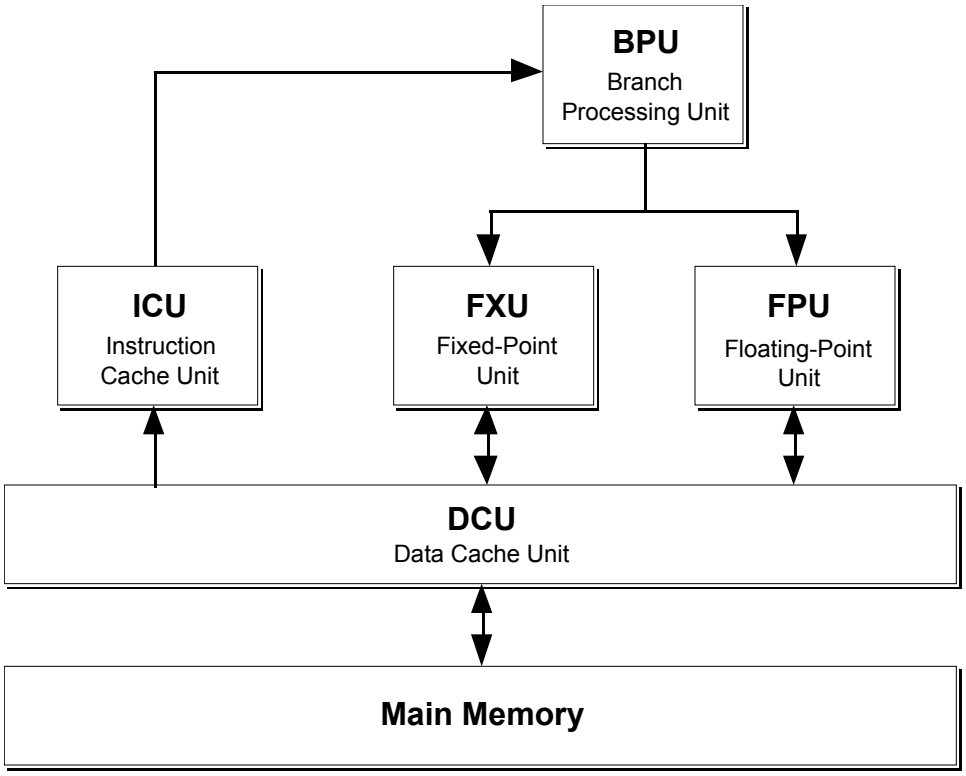
If the then-clause is too short to cover the branching delay, but the else-clause is relatively long, then the condition can be negated and the clauses reversed to eliminate any delay. This is a reasonable method to apply if both clauses have equal probability of executing (or if the else-clause is more likely).

Copying and Pasting Code

If both clauses are very short, or if the most often executed clause is too short, then another technique known as “pasting” or “gluing” must be employed.

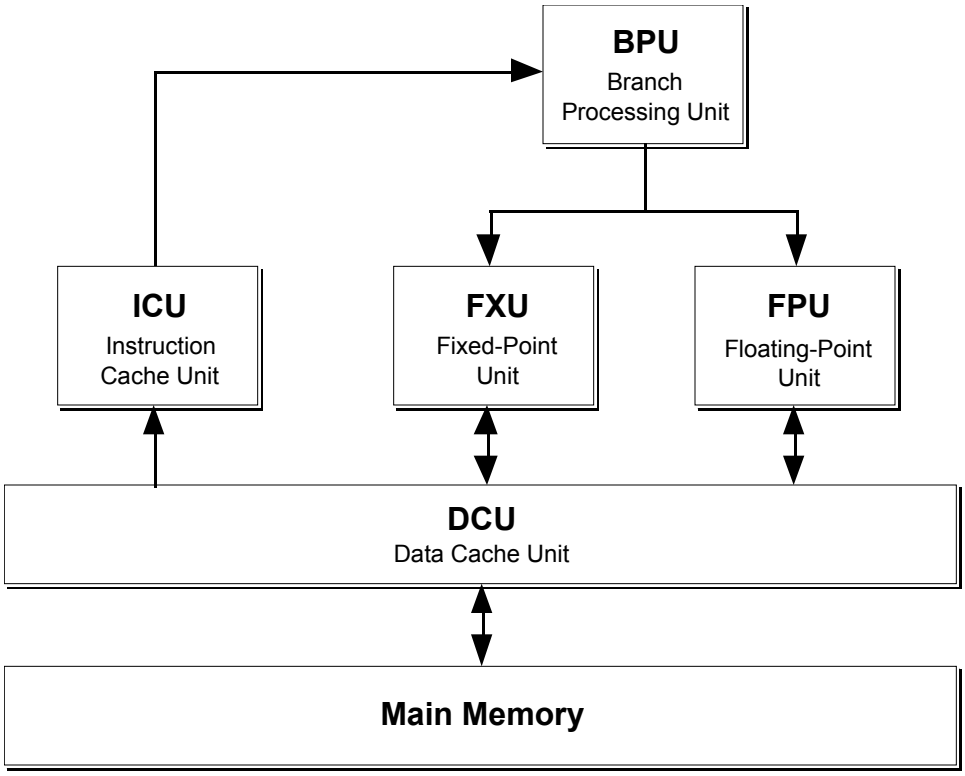
This method basically lengthens both the else- and the then-clause by copying instructions from after the if-then-else statement and inserting them at the end of each of the clauses. The terminating branch of the then-clause is modified to branch to the instruction after the last copied instruction.

```
      cmp     cr0,...
      bXXc    cr0,@0
```



[Figure 2.1] Logical view of CPU functional units


```
      # then-clause
      xxx
      zzz'
      b       @1
 @0:  # else-clause
      yyy
      zzz'
 @1:  zzz''
```



[Figure 2.1] Logical view of CPU functional units

Here, the first few instructions after the if-then-else statement (`zzz'`) have been added to the then- and else-clauses. The label `s2` has been moved so that the then-clause continues at the proper instruction.

If No Else, Then...

If there is no else-clause and the then-clause is not expected to be the most oft executed, then some other transformation tricks need to be applied.

An easy way to eliminate the delay with a seldom-used then-clause is to negate the condition and move the then-clause someplace outside the sequential instruction stream. In the transformation shown below, the then-clause is placed before the condition check (although it could have been placed anywhere). The normal instruction flow jumps over the then-clause, performs the comparison and conditionally executes beyond the if-then statement. Since this is the path that is expected to be executed most often, the instructions will be enabled and there will be no pipeline delays.

To illustrate the above transformation, code of the form:

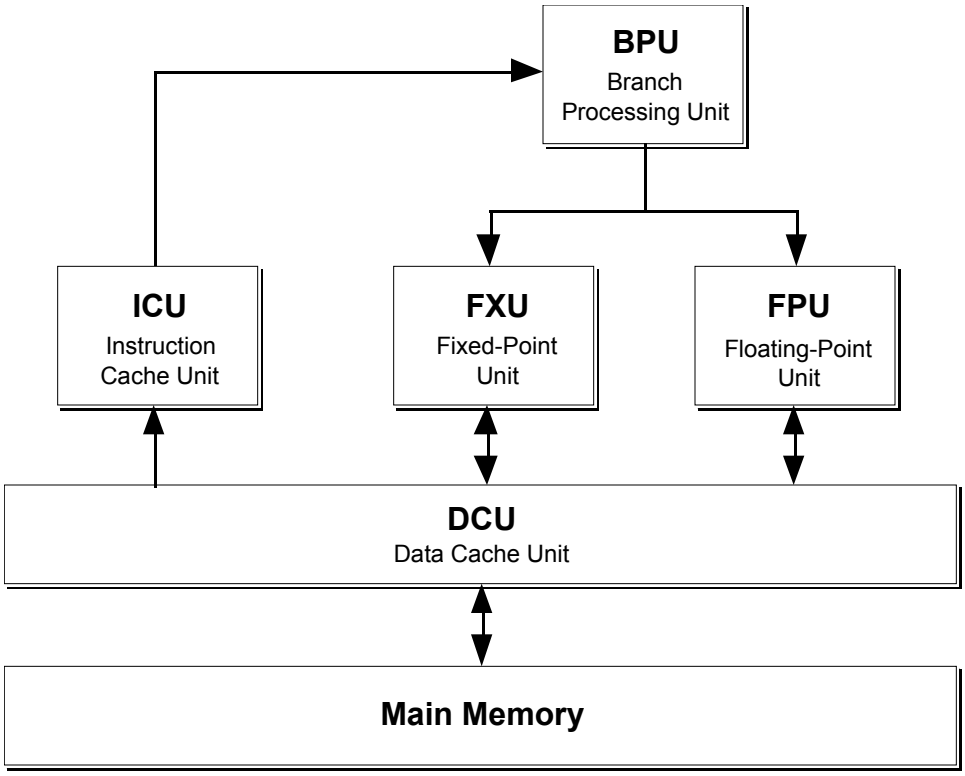
```
        cmp     cr0,...
        bXXc    cr0,@0
        # then-clause
        xxx
@0:     zzz
```

can be transformed into something like:

```
        b       @1
        # then-clause
@0:     xxx
        b       @2
@1:     cmp     cr0,...
        ¬bXXc   cr0,@0
@2:     zzz
```

Optimizing From High-Level Languages

Many of these if-then-else optimizations are such that they can be performed from a high-level language as well as assembly. Hopefully, compilers will be intelligent enough to detect this condition automatically and generate the



[Figure 2.1] Logical view of CPU functional units

appropriate code.

6.4 Changes for the PowerPC

Branching in the PowerPC differs from the POWER Architecture in that the PowerPC has added more complex static branch prediction. This mechanism predicts that backward branches will be taken and that forward branches will not be taken.

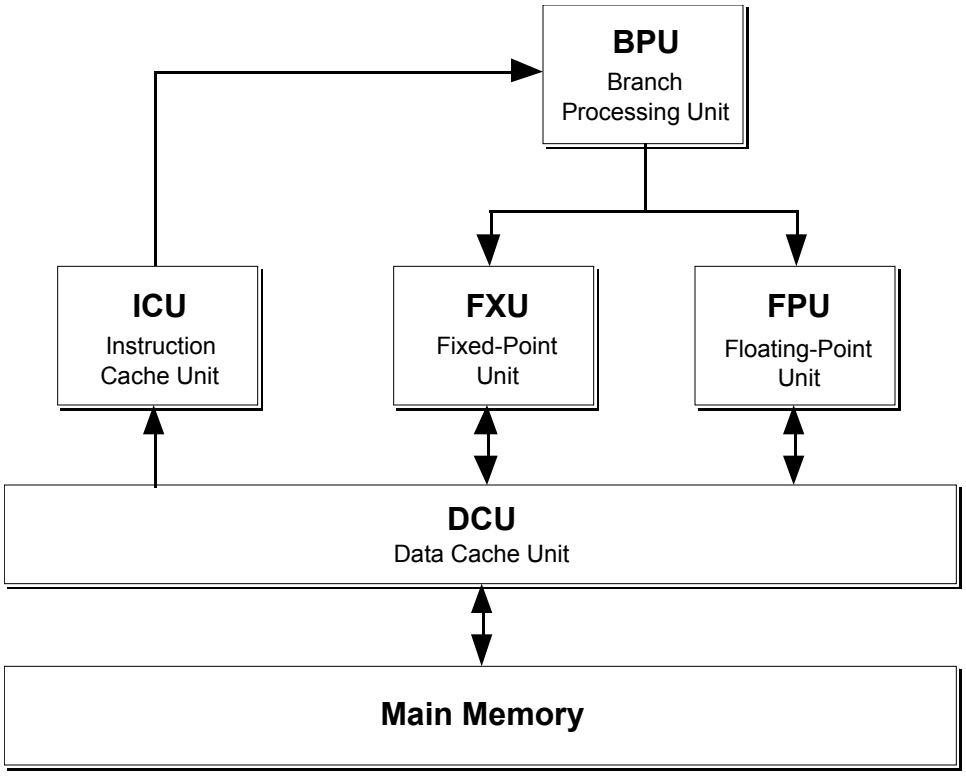
This form of branch prediction is simpler than the dynamic techniques which some processors implement, while providing most of the benefits. A dynamic branch prediction scheme can detect when the prediction is incorrect more than 50% of the time and automatically reverse the prediction.

The fact that the prediction in the PowerPC is static requires that the program be profiled to determine whether the branch is more or less likely to be taken. A reversal bit in the instruction must be set if the most likely direction is not the default direction that would be made by the static branch prediction.

The modified branch prediction algorithm affects loop closing branches more than the if-then-else branches because the loop closing branches are now more likely to be predicted correctly. This means that some of the code contortions presented in §6.1 may not be necessary in all cases, although the basic idea of moving back the condition so that it completes before instructions are dispatched after the dependent branch cannot hurt.

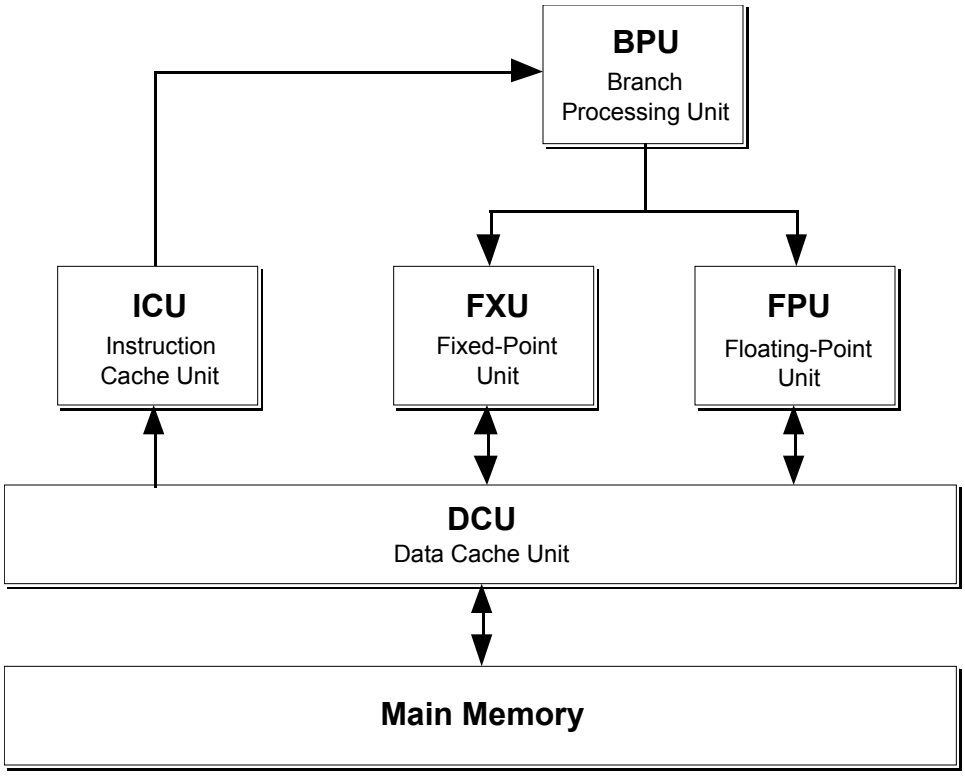
Since forward branches are predicted as being not taken, the techniques for if-then-else structures presented in §6.3 are still valid. However, the technique for when there is a rarely executed then-clause with no else-clause can be modified by either simply setting the reversal bit or by moving the then-clause somewhere *after* the if-then structure (possibly after the end of the current function). If the then-clause is placed before the if-then structure, then the branch back to it will be (erroneously) interpreted as a loop closing branch and predicted as being taken.

Even with the enhanced branch prediction, there



[Figure 2.1] Logical view of CPU functional units

are still some constructs which need to be avoided. It is doubtful that the PowerPC will have (at least for the first few implementations) the ability to predict through multiple unresolved branches. This means that a branch following an unresolved branch should still be avoided if possible.



[Figure 2.1] Logical view of CPU functional units

Of course, relying on branch prediction is unnecessary if the condition upon which the branch is dependent is set far enough ahead of time. Setting the condition far in advance is preferable to relying on the branch prediction because the branch will be optimal no matter which direction the branch takes.

7.0 Writing Assembly Language Programs

Object files for the RS/6000 are stored using a file format known as XCOFF, which is a derivative of a standard UNIX[12†] Common Object File Format (COFF).

An object file is divided into three major sections: the `.text` section, where the program instructions and read-only data are stored; the `.data` section, where initialized read/write data is stored; and the `.bss` section, where the uninitialized read/write data is stored.

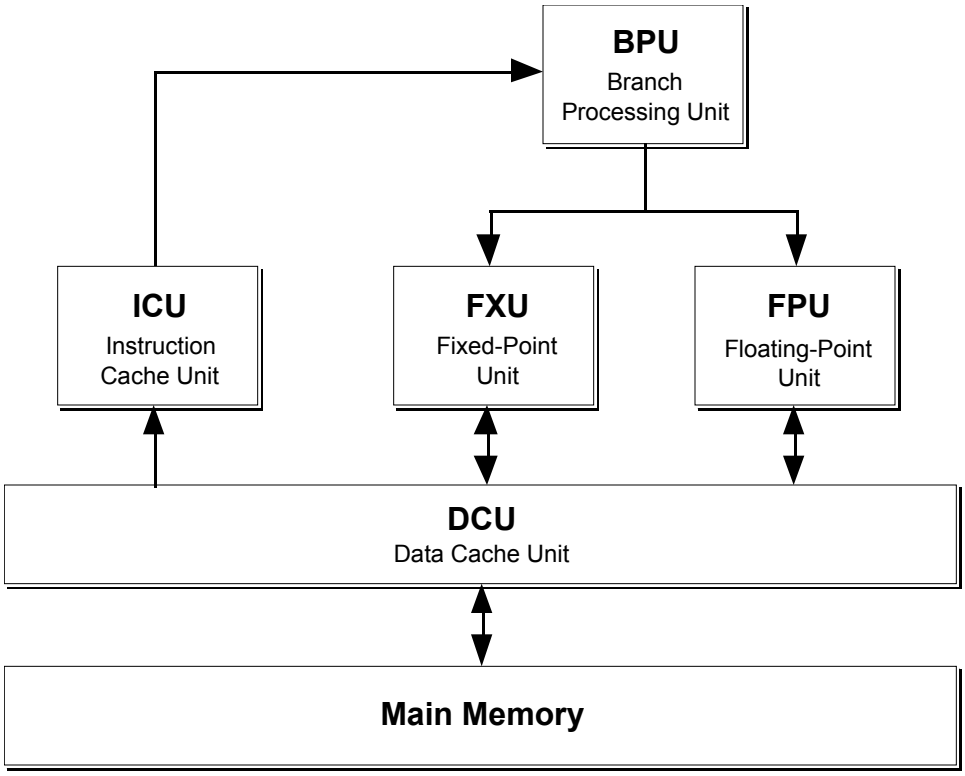
The XCOFF file format also defines other sections which contain loading or debugging (symbol table, line number, et al.) information. Programs can be further broken up into control sections or `csect`'s. When writing assembly language programs, each instruction or data storage directive must belong to a `csect`, and each `csect` must belong to one (and only one) of the XCOFF file sections.

A sample assembly language program which makes use of various `csect`s is given in Appendix C. This simple program demonstrates many of the basic structures required for assembly language programs.

7.1 Register Usage Conventions

As with most systems, there are registers which, while architected as general purpose registers, are assigned special meaning by either the hardware or the operating system. These register conventions allow functions compiled from different languages to be linked together.

[12†] UNIX was a registered trademark of AT&T, but is now (at the time of this writing) a trademark of UNIX[†] Systems Laboratories, which will soon be owned by Novell, Inc.



[Figure 2.1] Logical view of CPU functional units

GPR 0 is used in function prolog and epilog code as storage space for the Link Register.

GPR 1 is used as the stack pointer.
GPR 2 contains a pointer to the Table of Contents (TOC). The TOC pointer is used for all references across `csect`s.

GPR 3 to GPR 10 contain the first 8 words of the argument list and the return value list.

GPR 11 and GPR 12 are special scratch registers which do not need to be saved or restored.

GPR 13 to GPR 31 are non-volatile registers which must be preserved (i.e.: saved and restored) by any routine which uses them. Registers are typically used from the highest (GPR 31) to the lowest so that the `lm` and `stm` instructions can be used to save and restore them easily.

FPR 0 is a scratch floating-point register which is not preserved across calls.

FPR 1 to FPR 13 contain the first 13 floating-point parameters and function results.

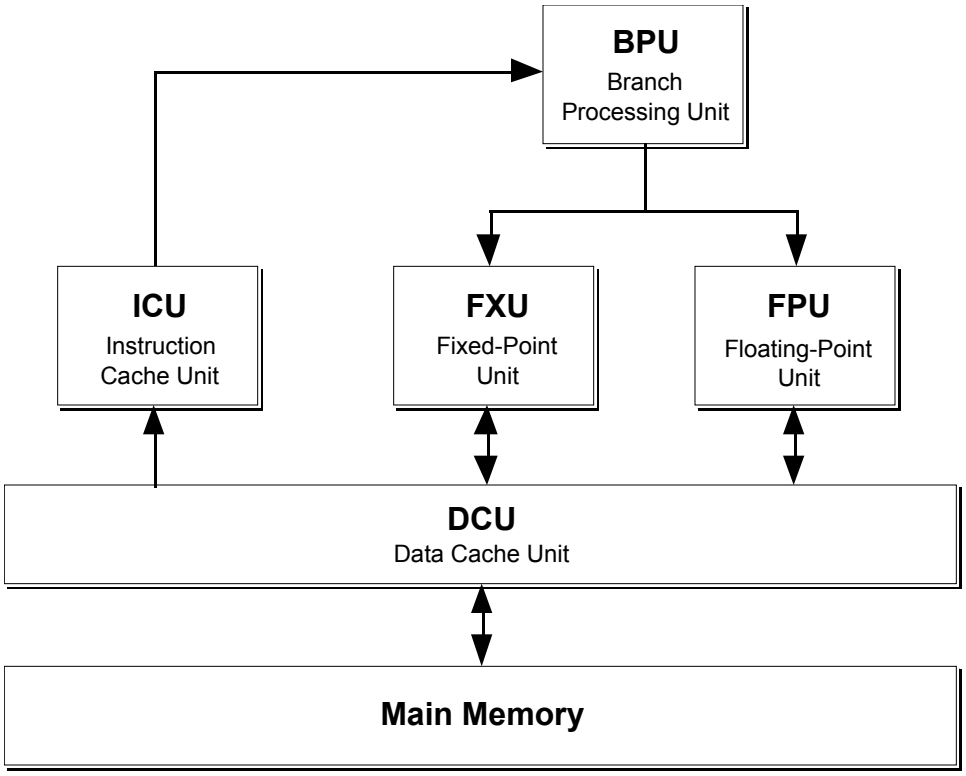
FPR 14 to FPR 31 are non-volatile floating-point registers which must be saved and restored if they are modified in a function.

Condition Register fields 0, 1, 6, and 7 are scratch fields which are not preserved across calls. CR field 5 is reserved for system use. CR fields 2, 3, and 4 must be preserved if they are modified in a function.

All other registers are not generally preserved across function calls.

7.2 Function Calling Conventions

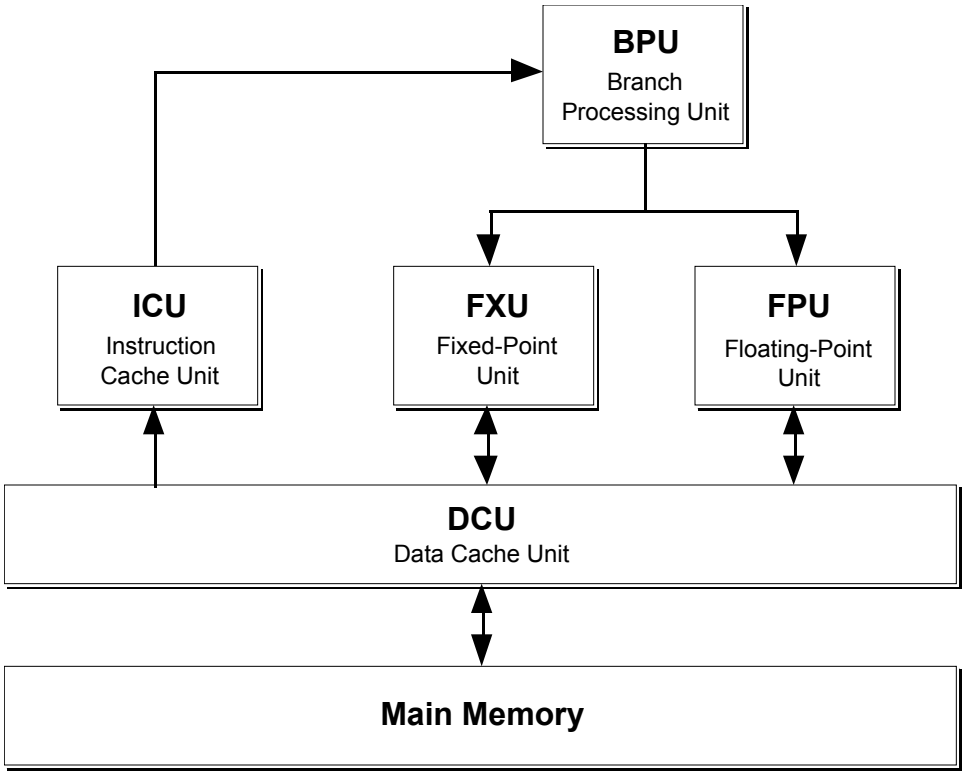
Each function must perform a sequence of tasks upon entry and exit. These instruction sequences are known as prolog and epilog code, respectively. The epilog and prolog are very similar to the `link A6,#-n` and the `unlk A6` instructions used in the 680x0 to build function stack frames, except that the `link`/`unlk` pair was never required on the 680x0.



[Figure 2.1] Logical view of CPU functional units

The stack frame which is built on the RS/6000 contains the saved registers for the function, the local storage area, and sets aside space for the OS to store information as needed during execution.

When a function is first entered, the stack pointer (`r1`) points to the stack frame for the caller function (the function which called the current function).



[Figure 2.1] Logical view of CPU functional units

The stack pointer points to the link area for the caller, which contains the back chain (stack pointer for the function which called the caller), the saved CR and LR, and space for the OS to save other useful items like the function's TOC pointer.

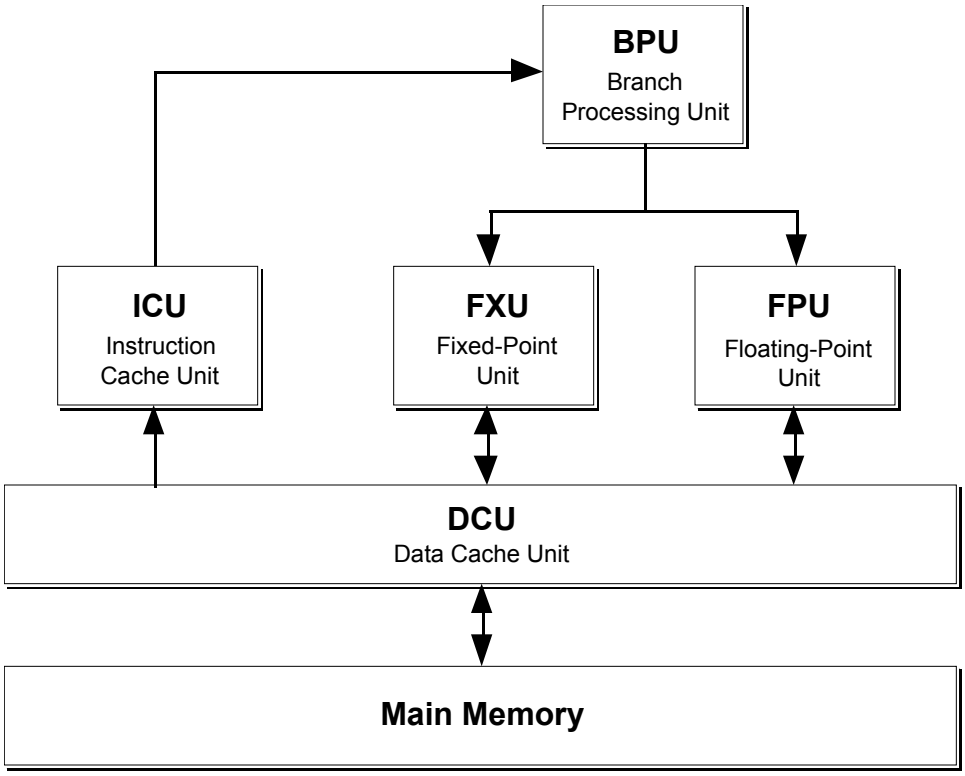
Immediately below the link area is the input parameter area. This is where the caller places parameters for the functions that it calls.

The prolog code for the function must perform the following operations:

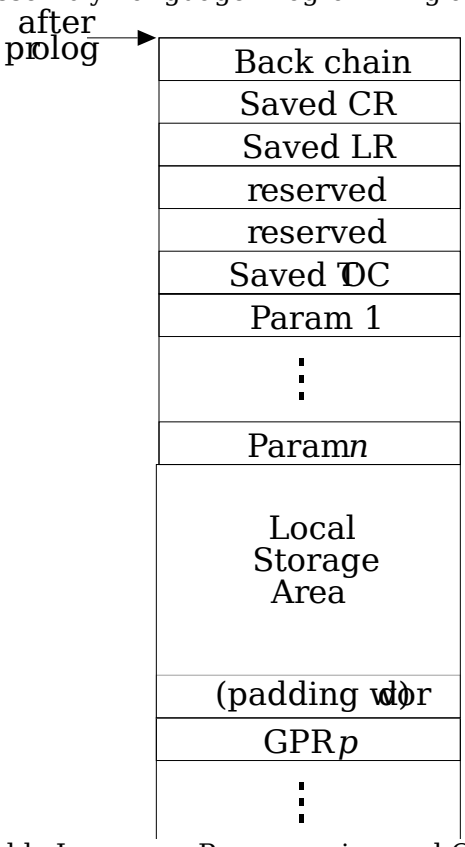
- Save any FPR's and GPR's which are modified by the function on the stack immediately above the caller's back-chain.
- Set aside as much local storage area on the stack as it requires
- Allocate space for an argument area immediately above the local storage area. This area must be at least 8 words long.
- Allocate space for the function's link area. The link area is 6 words long.
- Save the back chain to the caller function in its link area and set the stack pointer to point to the beginning link area.

The epilog code basically undoes what the prolog code set up.

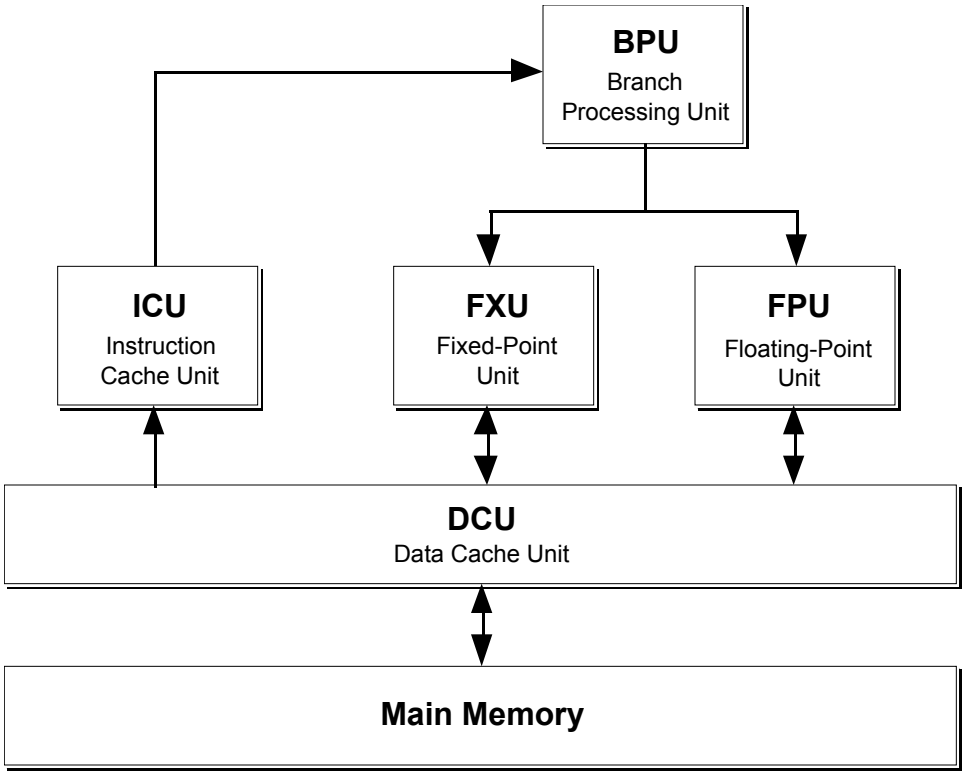
The example program given in Appendix C has comments which point out and describe the code which is associated with the function prolog and epilog operations.



[Figure 2.1] Logical view of CPU functional units



[Figure 7.1] RS/6000 stack frame before and after the function prolog code is executed.



[Figure 2.1] Logical view of CPU functional units

8.0 Teangaire

The Teangaire application is a program which runs on the Macintosh and loads RS/6000 .o files into a POWER architecture emulator. If an assembler source file is specified, the file is first assembled and then the object file is loaded.

There are two main goals of the application. First, the application is intended to be a bridge so that people who do not have access to a RS/6000 can practice writing assembly language programs for a POWER processor before the PowerPC is released.

The second goal is that the program should provide a mechanism for viewing the instructions as they flow through the pipeline so that resource conflicts and pipeline stalls can be readily identified.

8.1 The Emulatrix

The most basic function of Teangaire is that it allows assembly language programs to be loaded into a virtual machine and emulated instruction by instruction. All the machine registers are made available to the programmer in a multi-window environment which is shown in [Figure 8.1].

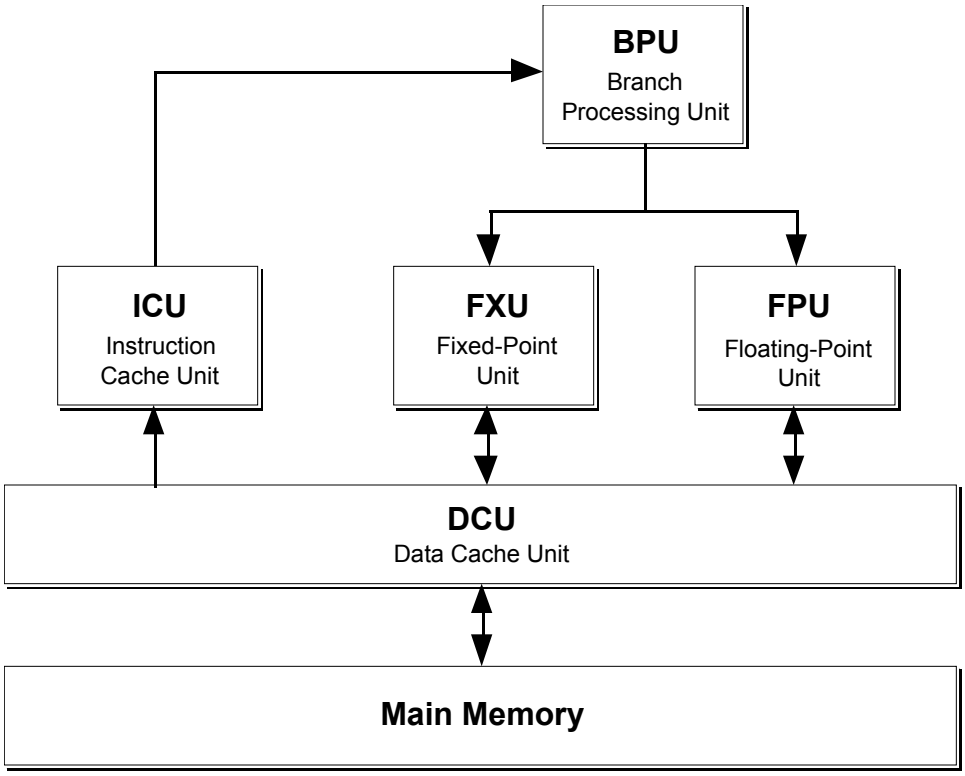
This part of the program provides all of the basic functions of an emulator: registers can be edited, breakpoints can be set, and there is a limited undo facility where the programmer can go back to the machine state before an instruction was executed.

8.2 The Pipeline Analyzer

Teangaire also provides a second level of emulation where a section of code is analyzed as it flows through the RS/6000 pipeline.

After this pipeline analysis is complete, a window shows each stage of the pipeline for each instruction and identifies places in the pipeline where the instruction flow has stalled.

The analyzer currently identifies most of the basic pipeline stalls, but because of its relatively simple pipeline model, it may miss situations



[Figure 2.1] Logical view of CPU functional units

which are subtle or obscure.

8.3 Future Enhancements

As it stands right now, Teangaire is still under development and will most likely undergo a variety of changes.

Eventually, Teangaire will support PowerPC object files and will have special knowledge of the idiosyncrasies of each processor.

There are also plans to improve the pipeline model, and hopefully add emulation support for the instruction and data caches. Currently, the model uses the optimistic assumption that there are no cache misses.

And finally, it is hoped that as people start to use this program, suggestions for various features will be made.

8.4 How to Order

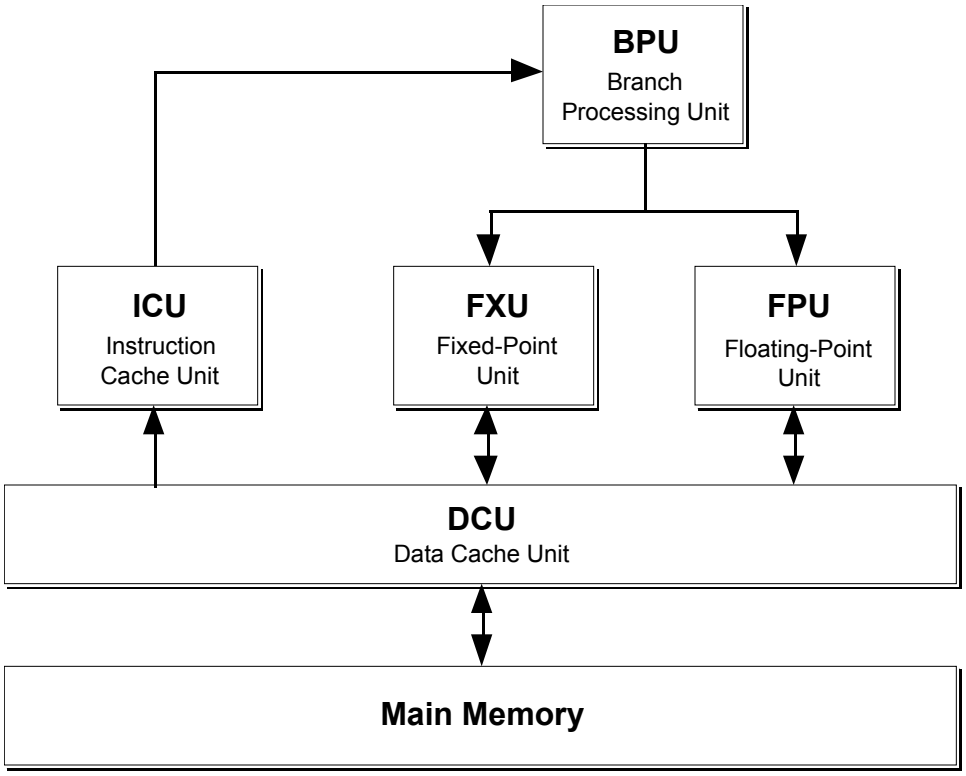
The Teangaire application will be made available on the MacHack server and will be placed on the CD which is distributed after the conference. The latest version can also be anonymously ftp'ed from `ftp.ces.cwru.edu` in the `/pub/larvae` directory.

I can be reached at:

Internet:
`platypus@curie.ces.cwru.edu`
AppleLink:
`platypus@curie.ces.cwru.edu@internet#`
US Mail:
4972 Leafy Mill West
North Ridgeville, Ohio 44039

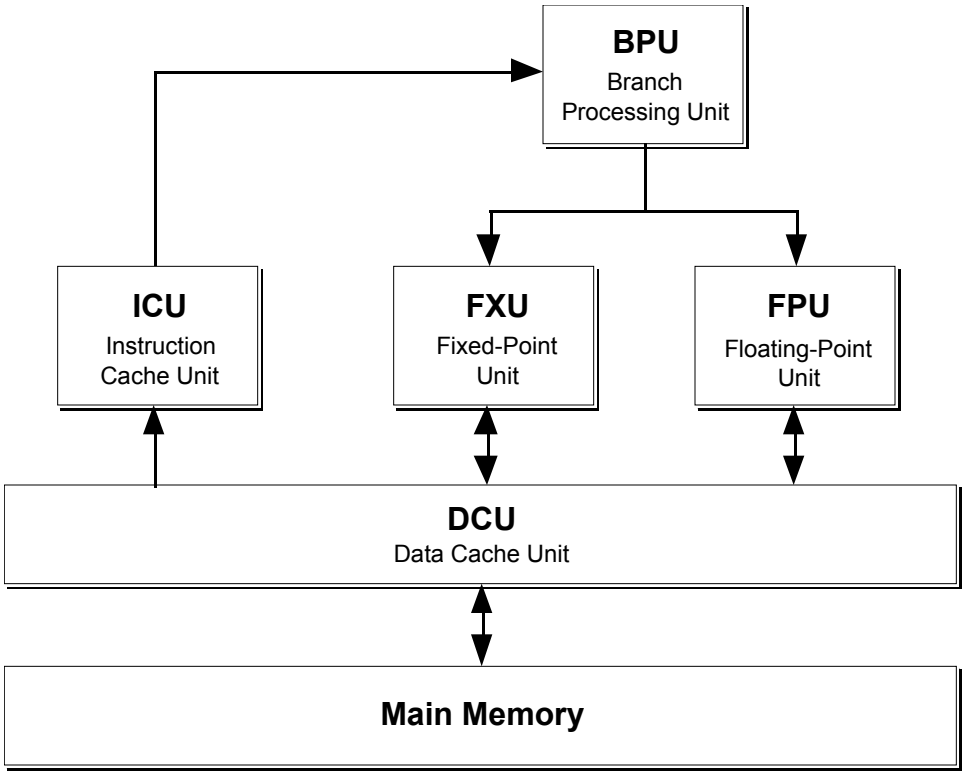
If you enjoyed this paper, you might also enjoy the following works by the same author:

- "A Neuroethological Model of the Cockroach Escape Response" [Kacmarcik91]
- "A Model of Distributed Sensorimotor



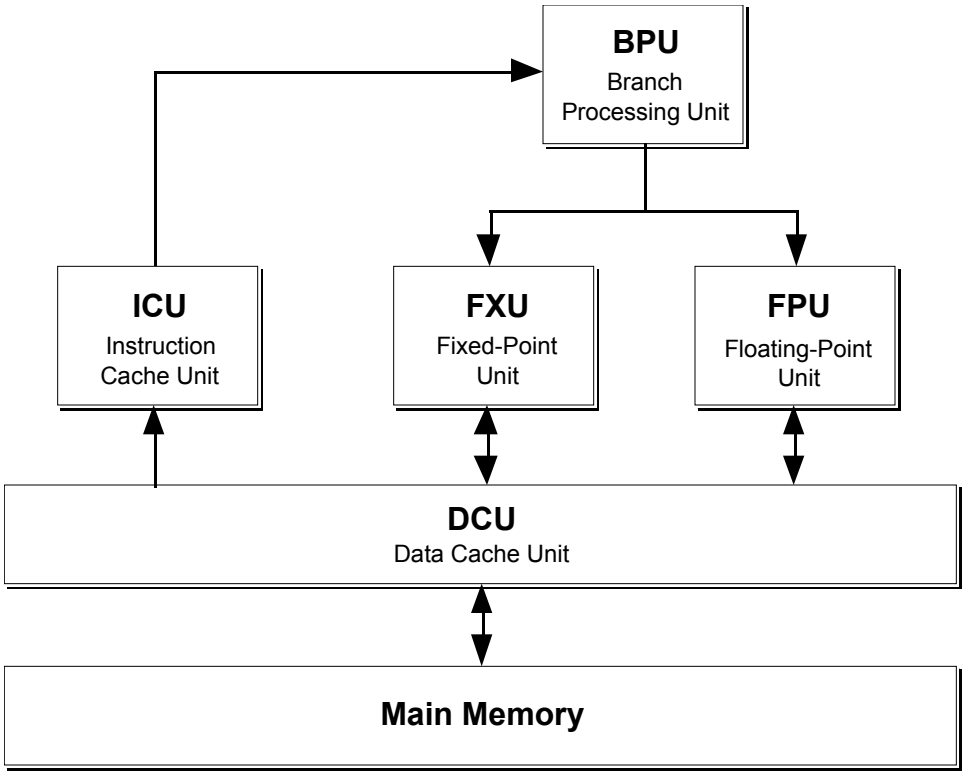
[Figure 2.1] Logical view of CPU functional units

55 Assembly Language Programming and Optimization Techniques for the Power Architecture Control in the Cockroach Escape Turn" [Beer91]



[Figure 2.1] Logical view of CPU functional units

ith a sample object file loaded



[Figure 2.1] Logical view of CPU functional units

References

[Aho86] Aho, A.V., Sethi, R., Ullman, J.D., "Compilers: Principles, Techniques, and Tools", Addison-Wesley Publishing Company, Reading, Massachusetts, ISBN 0-201-10088-6, QA76.76.C65A37 1986.

[Beer91] Beer, R.D., Kacmarcik, G.J., Ritzmann, R.E., Chiel, H.J., in "Advances in Neural Information Processing Systems 3", Lippmann, R.P., Moody, J., Touretzky, D.S. (ed's), Morgan Kaufmann Publishers, 1991.

[Case91] Case, B., "RS/6000 Architecture Fine-Tuned for PowerPC", The Microprocessor Report, Vol. 5 #24, pp. 9-12, 26 December 1991.

[Case92] Case, B., "IBM Delivers First PowerPC Microprocessor", The Microprocessor Report, Vol. 6 #14, pp. 1,6-10, 28 October 1992.

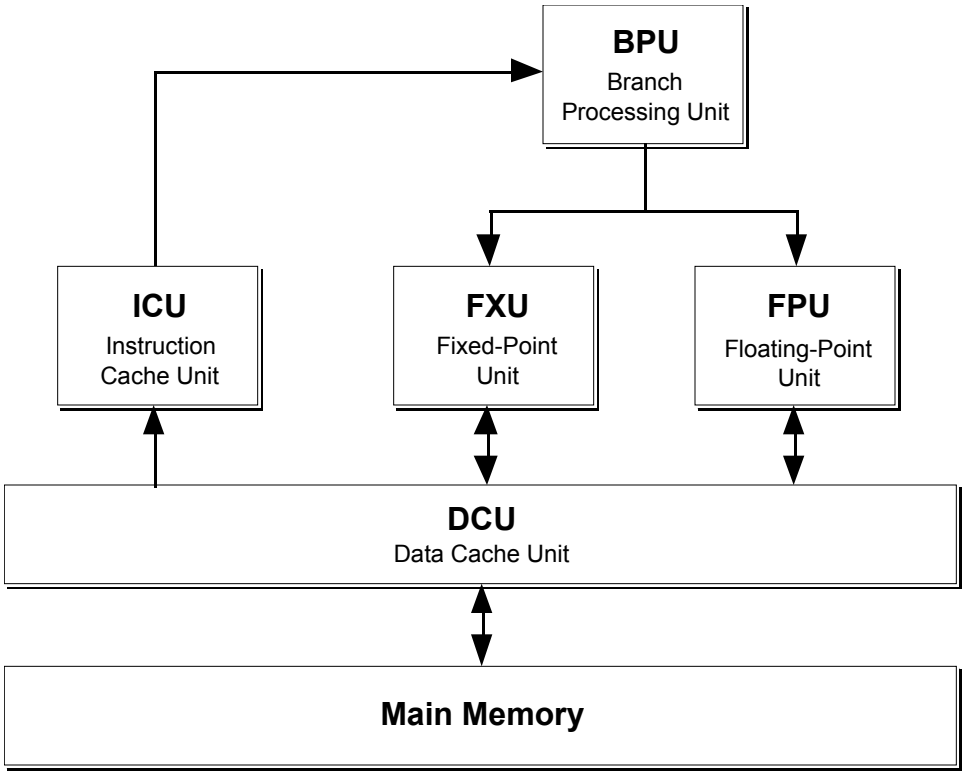
[Diefendorff93] Diefendorff, K., "The PowerPC Architecture", document included with PowerOpen Association, Inc. literature, Billerica, Mass., 1993.

[Gircys88] Gircys, G.R., "Understanding and Using COFF", O'Reilly & Associates, Inc., Sebastopol, California, 1988.

[Golumbic90] Golumbic, M.C., Rainish, V., "Instruction Scheduling Beyond Basic Blocks", IBM J. Res. & Develop., Vol. 34 #1, pp. 93-97, IBM Corporation, G322-0169-00, Jan 90.

[Grohoski90a] Grohoski, G.F., "Machine organization of the IBM RISC System/6000 processor", IBM J. Res. & Develop., Vol. 34 #1, pp. 37-58, IBM Corporation, G322-0169-00, Jan 90.

[Grohoski90b] Grohoski, G.F., Kahle, J.A., Thatcher, L.E., Moore, C.R., "Branch and Fixed-Point Instruction Execution Units", IBM RISC System/6000 Technology, pp. 24-32, IBM Corporation, SA23-2619, 1990.



[Figure 2.1] Logical view of CPU functional units

[IBM91] Course Outline Notes from "RISC System/6000 Hardware Architecture", IBM Corporation, IBM966-8211, 9 January 1991.

[IBM92a] "AIX Version 3.2 for RISC System/6000 Assembler Language Reference", IBM Corporation, SC23-2197-01, January 1992.

[IBM92b] "POWERstation and POWERserver Hardware Technical Information - General Architectures", IBM Corporation, SA23-2643-02, 1992.

[Kacmarcik91] "A Neuroethological Model of the Cockroach Escape Response", Center for Automation and Intelligent Systems Research, Case Western Reserve University, Technical Report TR 91-133, June 1991.

[Kim92] Kim, J-H., Huang, W., "The AIX Binder System", AIXpert, pp. 29-36, IBM Corporation, August 1992.

[Motorola93] "PowerPC 601 RISC Microprocessor User's Manual", Motorola, MPC601UM/AD, 1993.

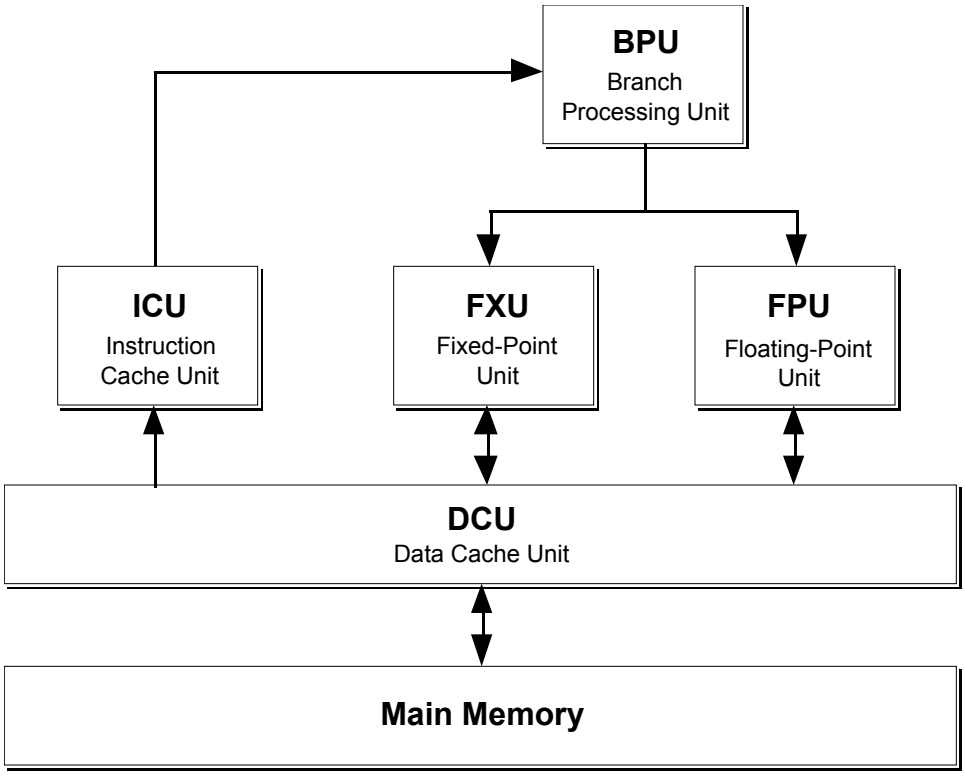
[Oehler92] Oehler, R.R., Outline Notes from "PowerPC Architecture", IBM Corporation, IBM966K-3635, 27 February 1992.

[Patterson90] Patterson, D. A., Hennessy, J. L., "Computer Architecture: A Quantitive Approach", Morgan Kaufmann Publishers, San Mateo, California, ISBN 1-55880-069-8, QA76.9.A73P377 1990.

[Press92] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., "Numerical Recipes in C: The Art of Scientific Computing - 2nd Ed.", Cambridge University Press, Cambridge, UK, QA297.N866 1992.

[Warren90] Warren, H.S.Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor", IBM J. Res. & Develop., Vol. 34 #1, pp. 85-92, IBM Corporation, G322-0169-00, Jan 90.

[Warren91] Warren, H.S.Jr., "Predicting Execution Time on the IBM RISC System/6000", IBM Corporation, GG24-3711-00, July 1991.

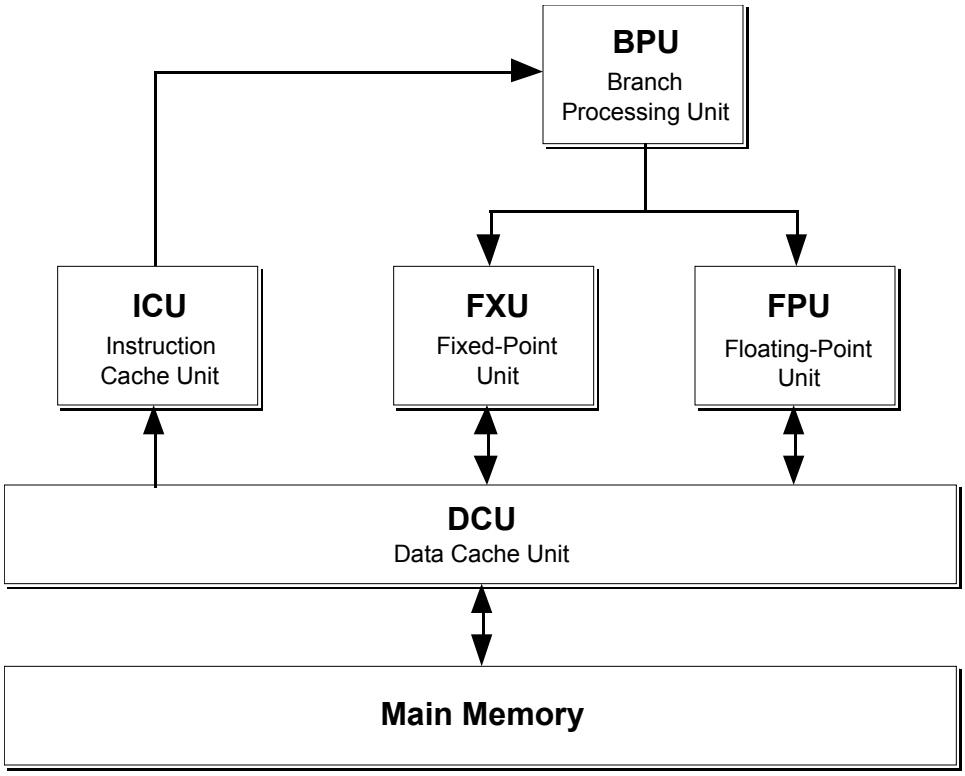


[Figure 2.1] Logical view of CPU functional units

Appendix A: RS/6000 & PowerPC Assembly Language Instruction Set Summary

The following notation is used in this appendix for the description of the instruction operation:

###	a decimal number
0x###	a hexadecimal number
Rn	the contents of GPR Rn
(RA|0)	the contents of GPR RA if RA!=0, or 0 if RA==0
FRn	the contents of FPR Rn
x[y]	bit y of register x
x{y}	bitfield y of register x (range of bits between x[y*4] to x[y*4+3])
|x|	absolute value of x
`	sign-extend to 32-bits (eg: `0xA7E3 == 0xFFFFA7E3)
~	one's complement (eg: ~11001010 == 00110101)
&	logical and (eg: 0101 & 0011 == 0001)
|	logical or (eg: 0101 | 0011 == 0111)
^	logical xor (eg: 0101 ^ 0011 == 0110)
!	bit concatenate (eg: 0110 ! 1101 == 01101101)
==	test for equality
!=	test for inequality (not equal)
a?b:c	if (a) then (b) else (c)
rrot	right rotate
lrot	left rotate
%	remainder
<<	left shift



[Figure 2.1] Logical view of CPU functional units

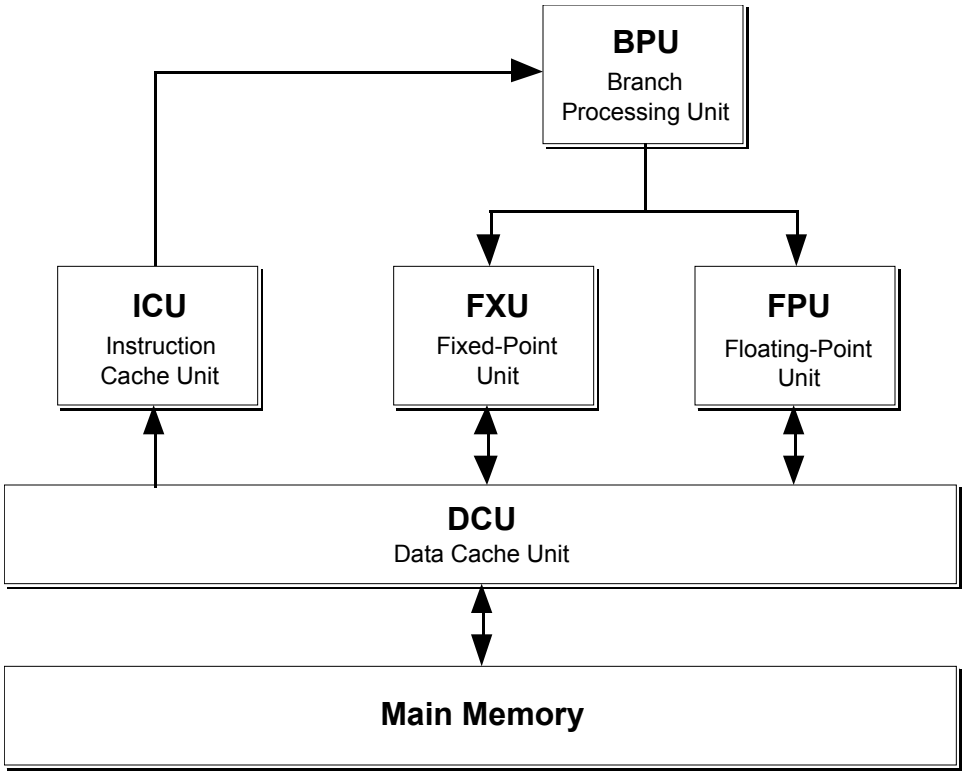
60 Assembly Language Programming and Optimization Techniques for the Power Architecture
>> right shift

Parentheses may be used to group operations, and 't' is used as a temporary swap space when the contents of two registers depend on each other's initial conditions.

Many of the instruction mnemonics have changed between the POWER and PowerPC. When the mnemonic has changed, the PowerPC form is the main entry and the POWER form is given in *italics* after the instruction description.

In addition, the instruction may be marked with one or more of the following symbols:

✗ Identifies an instruction which has been removed from the PowerPC.
✓ Identifies an instruction which has been added for the PowerPC.
✧ Identifies an instruction which is implemented in the 601, even though the instruction is not part of the PowerPC specification.
✂ Identifies an instruction which is not implemented in the 601, even though the instruction is part of the PowerPC specification.
• Identifies a delayed-CR instruction (see §5.8 Delayed-CR Instructions).



[Figure 2.1] Logical view of CPU functional units

Arithmetic Instructions

Addition:

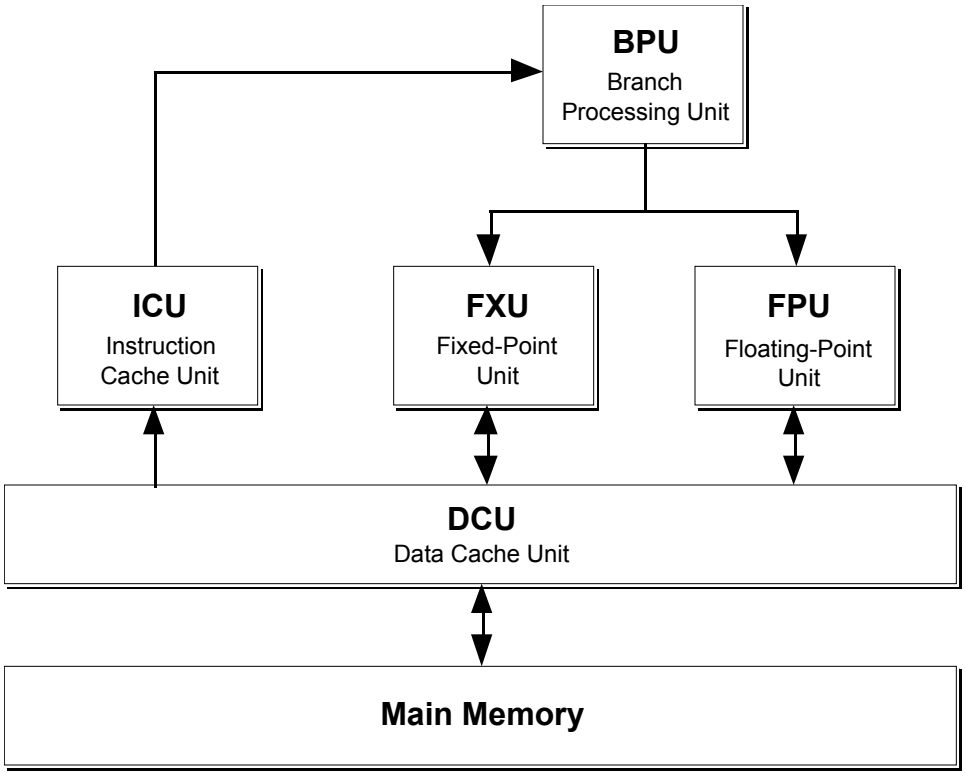
`add[o][.]`	`RT,RA,RB`	RT = RA + RB	Add *`cax[o][.]`*
`addc[o][.]`	`RT,RA,RB`	RT = RA + RB	Add Carrying *`a[o][.]`*
`adde[o][.]`	`RT,RA,RB`	RT = RA + RB + carry	Add Extended *`ae[o][.]`*
`addi`	`RT,D(RA)`	RT = D + (RA|0)	Add Immediate *`cal`*
`addic[.]`	`RT,RA,SI`	RT = RA + SI	Add Immediate Carrying *`ai[.]`*
`addis`	`RT,RA,UI`	RT = (UI ! 0x0000) + (RA|0)	Add Immediate Shifted *`cau`*
`addme[o][.]`	`RT,RA`	RT = RA + -1 + carry	Add to Minus One Extended *`ame[o][.]`*
`addze[o][.]`	`RT,RA`	RT = RA + 0 + carry	Add to Zero Extended *`aze[o][.]`*

Subtraction:

`si[.]`	`RT,RA,SI`	RT = RA - SI	Subtract Immediate
`subf[o][.]`	`RT,RA,RB`	RT = RB - RA	✓ Subtract From (without modifying CA)
`subfc[o][.]`	`RT,RA,RB`	RT = RB - RA	Subtract From Carrying *`sf[o][.]`*
`subfe[o][.]`	`RT,RA,RB`	RT = RB + ~RA + carry	Subtract From Extended *`sfe[o][.]`*
`subfic[.]`	`RT,RA,SI`	RT = SI - RA	Subtract From Immediate Carrying *`sfi`*
`subfme[o][.]`	`RT,RA`	RT = -1 + ~RA + carry	Sub. From Minus One Extended *`sfme[o][.]`*
`subfze[o][.]`	`RT,RA`	RT = 0 + ~RA + carry	Subtract From Zero Extended *`sfze[o][.]`*

Multiplication:

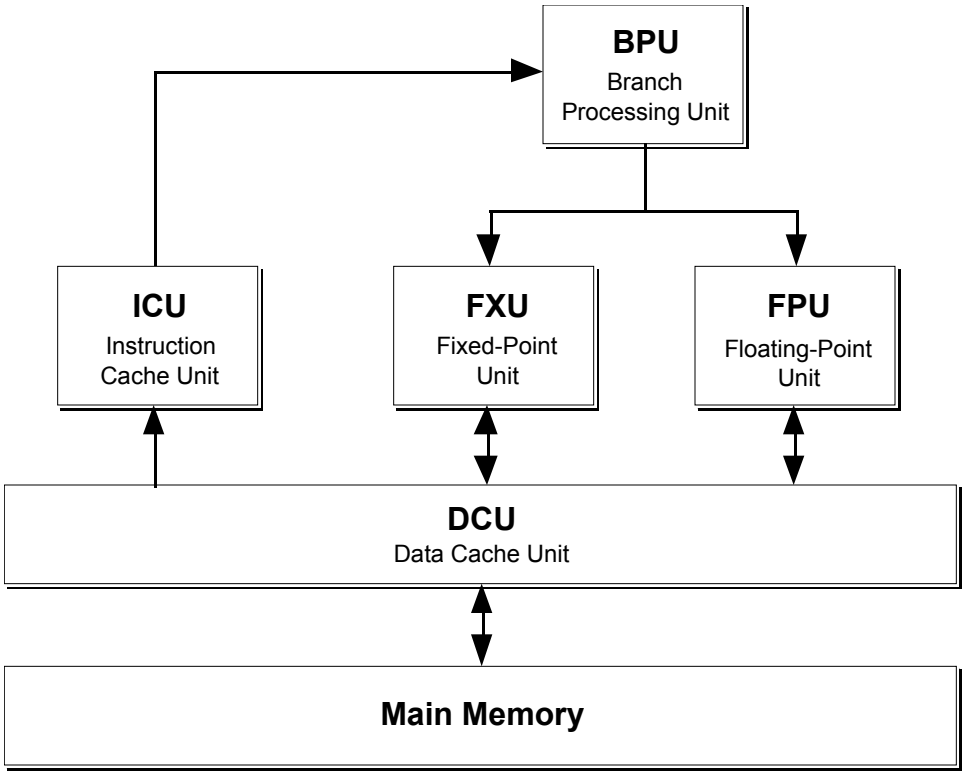
`mul[o][.]`	`RT,RA,RB`	(RT!MQ) = RB * RA	✗✧• Multiply
`mulhd[.]`	`RT,RA,RB`		✓✁ Multiply High Doubleword
`mulhdu[.]`	`RT,RA,RB`		✓✁ Multiply High Doubleword Unsigned
`mulhw[.]`	`RT,RA,RB`		✓ Multiply High Word



[Figure 2.1] Logical view of CPU functional units

`mulhwu[.]`	`RT,RA,RB`		✓ Multiply High Word Unsigned
`mulli`	`RT,RA,SI`	RT = RA * SI	Multiply Immediate *muli*
`mulld[.]`	`RT,RA,RB`		✓✁ Multiply Low Doubleword
`mullw[o][.]`	`RT,RA,RB`	RT = RA * RB	• Multiply Short *muls[o][.]*
Division:			
`div[o][.]`	`RT,RA,RB`	RT = (RA!MQ)/RB MQ = (RA!MQ)%RB	✗✧• Divide
`divd[o][.]`	`RT,RA,RB`		✓✁ Divide Doubleword
`divdu[o][.]`	`RT,RA,RB`		✓✁ Divide Doubleword Unsigned
`divs[o][.]`	`RT,RA,RB`	RT = RA / RB MQ = RA % RB	✗✧• Divide Short
`divw[o][.]`	`RT,RA,RB`	RT = RA / RB	✓ Divide Word
`divwu[o][.]`	`RT,RA,RB`	RT = RA / RB	✓ Divide Word Unsigned
Miscellaneous:			
`abs[o][.]`	`RT,RA`	RT = |RA|	✗✧• Absolute Value
`doz[o][.]`	`RT,RA,RB`	t = RB - RA RT = (t < 0) ? 0 : t	✗✧• Difference or Zero
`dozi`	`RT,RA,SI`	t = SI - RA RT = (t < 0) ? 0 : t	✗✧ Difference or Zero Immediate
`nabs[o][.]`	`RT,RA`	RT = -|RA|	✗✧• Negative Absolute Value
`neg[o][.]`	`RT,RA`	RT = -RA	Negate

[Table A.1] Instruction Set Summary (cont.)



[Figure 2.1] Logical view of CPU functional units

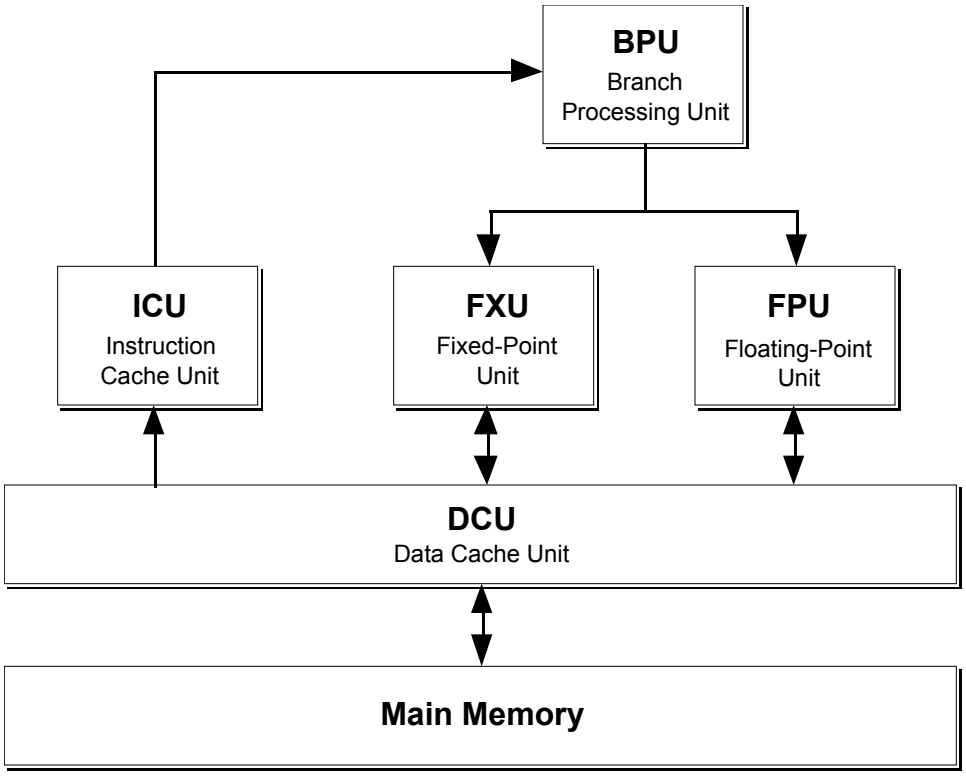
Condition Register Instructions

Boolean CR Operations:

`crand`	`BT,BA,BB`	CR[BT] = CR[BA] & CR[BB]	CR AND
`crandc`	`BT,BA,BB`	CR[BT] = CR[BA] & ~CR[BB]	CR AND with Complement
`creqv`	`BT,BA,BB`	CR[BT] = ~(CR[BA] ^ CR[BB])	CR Equal
`crnand`	`BT,BA,BB`	CR[BT] = ~(CR[BA] & CR[BB])	CR NAND
`crnor`	`BT,BA,BB`	CR[BT] = ~(CR[BA] | CR[BB])	CR NOR
`cror`	`BT,BA,BB`	CR[BT] = CR[BA] | CR[BB]	CR OR
`crorc`	`BT,BA,BB`	CR[BT] = CR[BA] | ~CR[BB]	CR OR with Complement
`crxor`	`BT,BA,BB`	CR[BT] = CR[BA] ^ CR[BB]	CR AND

Move CR Field:

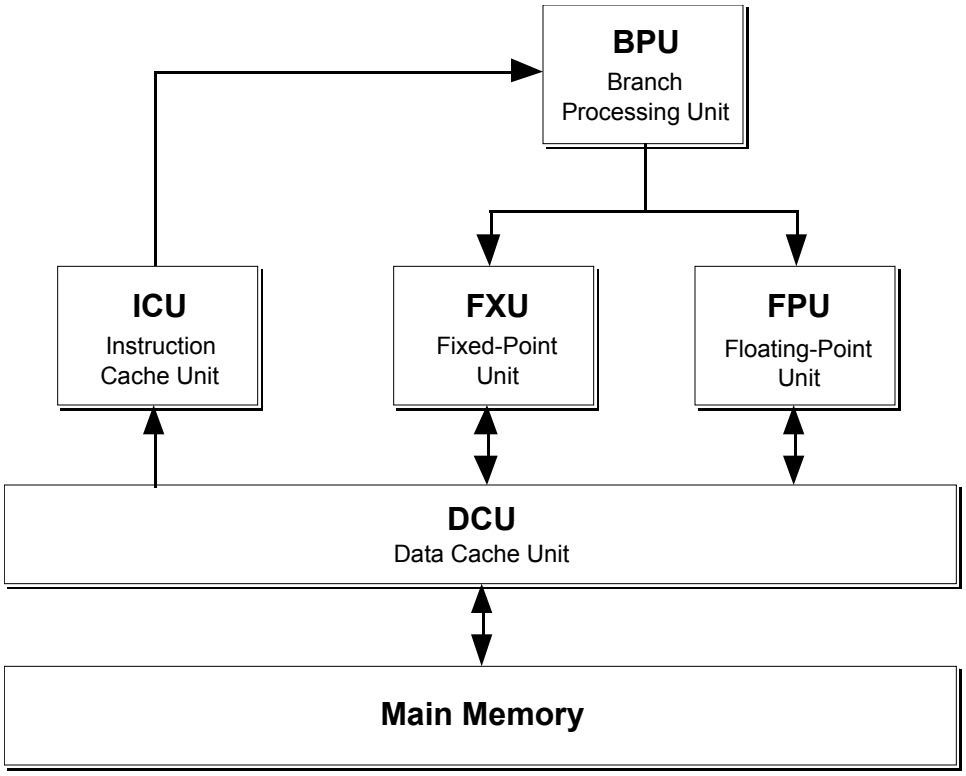
`mcrf`	`BF,BFA`	CR{BF} = CR{BFA}	Move CR Field
`mcrfs`	`BF,BFA`	CR{BF} = FPSCR{BFA}	Move to CR Field from FPSCR
`mcrxr`	`BF`	CR{BF} = XER[0-3]	• Move to CR Field from XER
`mfcr`	`RT`	RT = CR	Move From CR
`mtcrf`	`FXM,RS`	CR = RS (under control of FXM)	Move To CR Fields



[Figure 2.1] Logical view of CPU functional units

Logical/Boolean Instructions

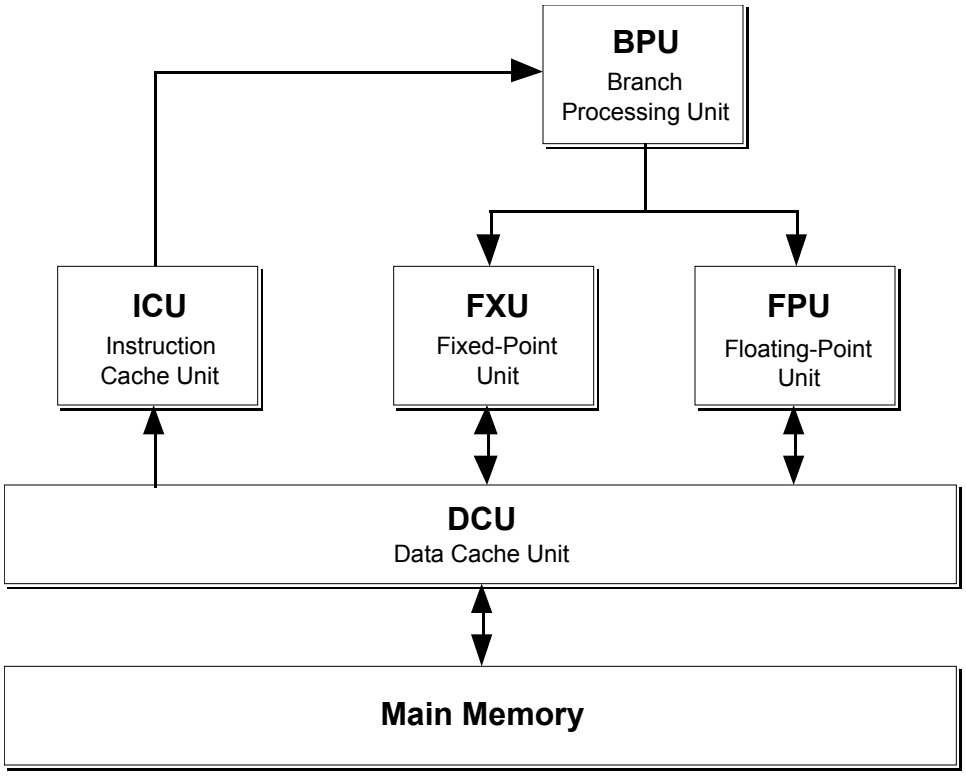
AND:			
`and[.]`	`RA,RS,RB`	RA = RS & RB	AND
`andc[.]`	`RA,RS,RB`	RA = RS & ~RB	AND with Complement
`andi.`	`RA,RS,UI`	RA = RS & (0x0000!UI)	AND Immediate Lower *`andil.`*
`andis.`	`RA,RS,UI`	RA = RS & (UI!0x0000)	AND Immediate Shift *`andiu.`*
OR:			
`or[.]`	`RA,RS,RB`	RA = RS | RB	OR
`orc[.]`	`RA,RS,RB`	RA = RS | ~RB	OR with Complement
`ori`	`RA,RS,UI`	RA = RS | (0x0000!UI)	OR Immediate Lower *`oril`*
`oris`	`RA,RS,UI`	RA = RS | (UI!0x0000)	OR Immediate Upper *`oriu`*
XOR:			
`xor[.]`	`RA,RS,RB`	RA = RS ^ RB	XOR
`xori`	`RA,RS,UI`	RA = RS ^ (0x0000!UI)	XOR Immediate Lower *`xoril`*
`xoris`	`RA,RS,UI`	RA = RS ^ (UI!0x0000)	XOR Immediate Upper *`xoriu`*
Miscellaneous:			
`cntlzw[.]`	`RA,RS`	RA = # of leading 0's in RS	• Count Leading Zeroes *`cntlz[.]`*
`cntlzd[.]`	`RA,RS`	RA = # of leading 0's in RS	✓✁ Count Leading Zeroes Doubleword
`eqv[.]`	`RA,RS,RB`	RA = ~(RS ^ RB)	Equivalent
`extsb[.]`	`RA,RS`	RA[0-23] = RS[24] RA[24-31] = RS[24-31]	✓ Extend Sign Byte (8-bit -> 32-bit)



[Figure 2.1] Logical view of CPU functional units

`extsh[.]`	`RA,RS`	RA[0-15] = RS[16] RA[16-31] = RS[16-31]	Extend Sign (16-bit -> 32-bit) *`exts[.]`*
`extsw[.]`	`RA,RS`	RA[0-31] = RS[32] RA[32-63] = RS[32-63]	✓✁ Extend Sign Word (32-bit -> 64-bit)
`nand[.]`	`RA,RS,RB`	RA = ~(RS & RB)	NAND
`nor[.]`	`RA,RS,RB`	RA = ~(RS | RB)	NOR
`not[.]`	`RA,RS`	same as “n`or RA,RS,RS`”	✓ Complement Register

[Table A.1] Instruction Set Summary (cont.)

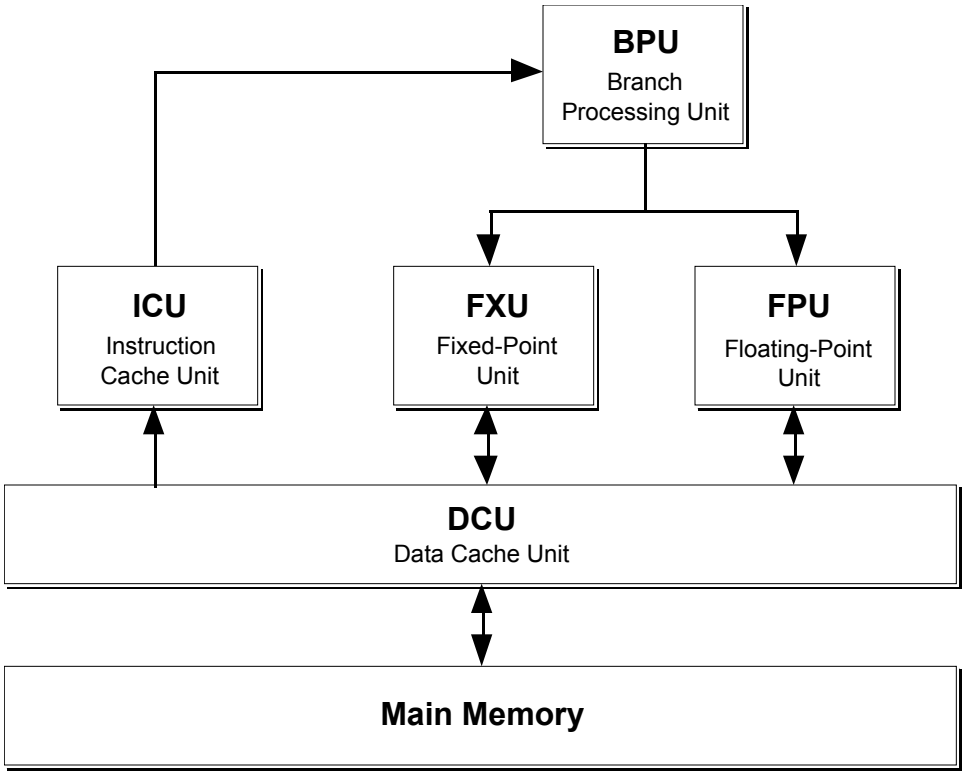


[Figure 2.1] Logical view of CPU functional units

Floating-Point Instructions

Floating-Point Arithmetic:

`fabs[.]`	`FRT,FRB`	FRT = |FRB|	F Absolute Value
`fadd[.]`	`FRT,FRA,FRB`	FRT = FRA + FRB	F Add *`fa[.]`*
`fadds[.]`	`FRT,FRA,FRB`	FRT = FRA + FRB	✓ F Add Single-Precision
`fcfid[.]`	`FRT,FRB`		✓✁ F Convert From Int (Double)
`fctid[.]`	`FRT,FRB`		✓✁ F Convert To Int (Double)
`fctidz[.]`	`FRT,FRB`		✓✁ F Conv To Int (Dbl) Rnd to Zero
`fctiw[.]`	`FRT,FRB`		✓ F Convert To Int (Word)
`fctiwz[.]`	`FRT,FRB`		✓ F Conv To Int (Wd) Rnd to Zero
`fdiv[.]`	`FRT,FRA,FRB`	FRT = FRA / FRB	F Divide *`fd[.]`*
`fdivs[.]`	`FRT,FRA,FRB`	FRT = FRA / FRB	✓ F Divide Single-Precision
`fmadd[.]`	`FRT,FRA,FRC,FRB`	FRT = FRA * FRC + FRB	F Multiply Add *`fma[.]`*
`fmadds[.]`	`FRT,FRA,FRC,FRB`	FRT = FRA * FRC + FRB	✓ F Multiply-Add Single
`fmr[.]`	`FRT,FRB`	FRT = FRB	F Move Register
`fmsub[.]`	`FRT,FRA,FRC,FRB`	FRT = FRA * FRC - FRB	F Multiply Subtract *`fms[.]`*
`fmsubs[.]`	`FRT,FRA,FRC,FRB`	FRT = FRA * FRC - FRB	✓ F Multiply-Subtract Single
`fmul[.]`	`FRT,FRA,FRC`	FRT = FRA * FRC	F Multiply *`fm[.]`*
`fmuls[.]`	`FRT,FRA,FRC`	FRT = FRA * FRC	✓ F Multiply Single-Precision
`fnabs[.]`	`FRT,FRB`	FRT = -|FRB|	F Negative Absolute Value
`fneg[.]`	`FRT,FRB`	FRT = -FRB	F Negate
`fnmadd[.]`	`FRT,FRA,FRC,FRB`	FRT = -(FRA * FRC + FRB)	F Neg. Multiply Add *`fnma[.]`*
`fnmadds[.]`	`FRT,FRA,FRC,FRB`	FRT = -(FRA * FRC + FRB)	✓ F Neg. Multiply-Add Single
`fnmsub[.]`	`FRT,FRA,FRC,FRB`	FRT = -(FRA * FRC - FRB)	F Neg. Mult-Subtract *`fnms[.]`*



[Figure 2.1] Logical view of CPU functional units

`fnmsubs[.]`	`FRT,FRA,FRC,FRB`	FRT = -(FRA * FRC - FRB)	✓ F Neg. Multiply-Sub. Single
`fres[.]`	`FRT,FRB`	FRT = 1 / FRB	✓✁ F Reciprocal Est. Single
`frsp[.]`	`FRT,FRB`	FRT = (single)FRB	F Round to Single Precision
`frsqrte[.]`	`FRT,FRB`		✓✁ F Reciprocal Square Root Est.
`fsub[.]`	`FRT,FRA,FRB`	FRT = FRA - FRB	F Subtract *`fs[.]`*
`fsubs[.]`	`FRT,FRA,FRB`	FRT = FRA - FRB	✓ F Subtract Single-Precision
`fsel[.]`	`FRT,FRA,FRC,FRB`	FRT = (FRA>=0)?FRC:FRB	✓✁ F Select
`fsqrt[.]`	`FRT,FRB`		✓✁ F Square Root
`fsqrts[.]`	`FRT,FRB`		✓✁ F Square Root Single-Precision

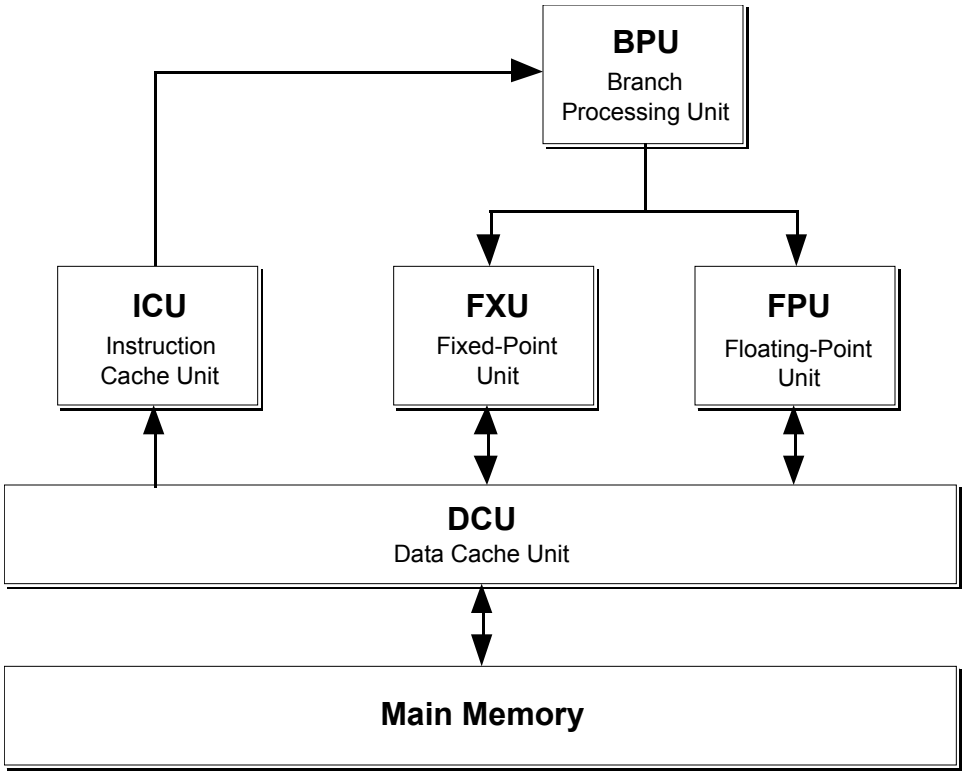
Floating-Point Compare:

`fcmpo[.]`	`BF,FRA,FRB`	CR{BF} = ordered compare of FRA and FRB	F Compare Ordered
`fcmpu[.]`	`BF,FRA,FRB`	CR{BF} = unordered compare of FRA and FRB	F Compare Unordered

FPSCR Operations:

`mffs[.]`	`FRT`	FRT = (0xFFFFFFFF ! FPSCR)	Move From FPSCR
`mtfsb0[.]`	`BT`	FPSCR[BT] = 0	Move To FPSCR Bit 0
`mtfsb1[.]`	`BT`	FPSCR[BT] = 1	Move To FPSCR Bit 1
`mtfsf[.]`	`FLM,FRB`	FPSCR = FRB (under control of FLM)	Move To FPSCR Fields
`mtfsfi[.]`	`BF,I`	FPSCR{BF} = I	Move To FPSCR Field Immediate

[Table A.1] Instruction Set Summary (cont.)

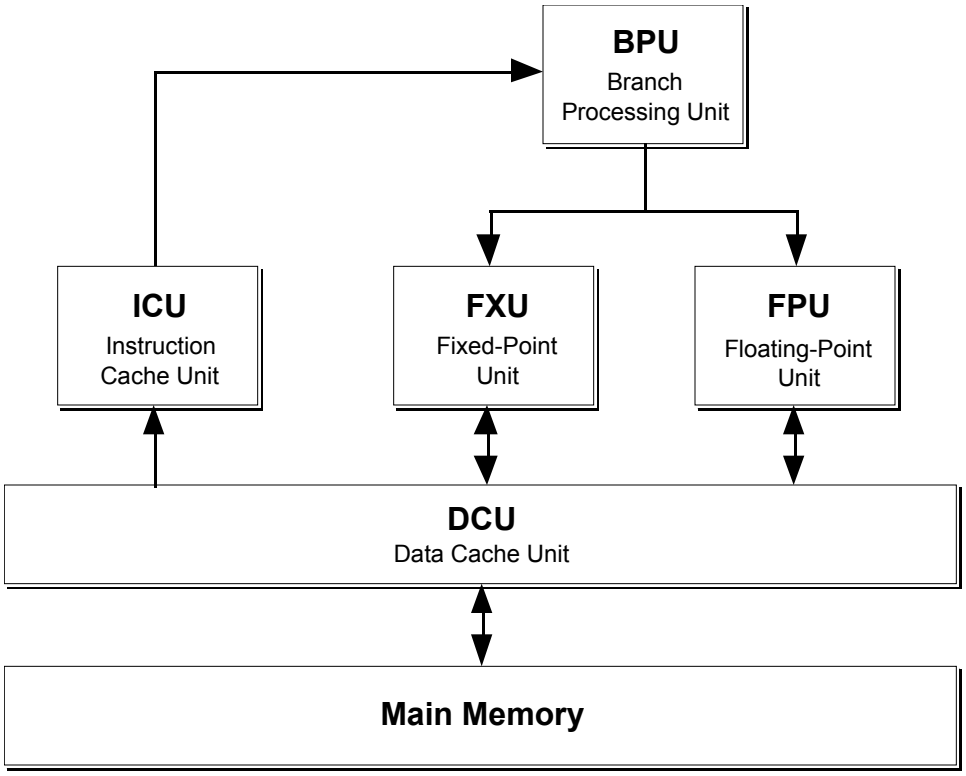


[Figure 2.1] Logical view of CPU functional units

Load/Store Instructions

Load:

`lbz[u]`	`RT,D(RA)`	RT <= Byte from D+(RA|0)
`lbz[u]x`	`RT,RA,RB`	RT <= Byte from RB+(RA|0)
`ld[u]`	`RT,D(RA)`	✓✁ Load Doubleword
`ld[u]x`	`RT,RA,RB`	✓✁ Load Doubleword Indexed
`ldarx`	`RT,RA,RB`	✓✁ Load Doubleword and Reserve Indexed
`lfd[u]`	`FRT,D(RA)`	FRT <= Doubleword from D+(RA|0)
`lfd[u]x`	`FRT,RA,RB`	FRT <= Doubleword from RB+(RA|0)
`lfs[u]`	`FRT,D(RA)`	FRT <= Word from D+(RA|0)
`lfs[u]x`	`FRT,RA,RB`	FRT <= Word from RB+(RA|0)
`lha[u]`	`RT,D(RA)`	RT <= Halfword from D+(RA|0)
`lha[u]x`	`RT,RA,RB`	RT <= Halfword from RB+(RA|0)
`lhbrx`	`RT,RA,RB`	RT <= Byte-reversed halfword from RB+(RA|0)
`lhz[u]`	`RT,D(RA)`	RT <= Halfword from D+(RA|0)
`lhz[u]x`	`RT,RA,RB`	RT <= Halfword from RB+(RA|0)
`li`	`RT,SI`	RT = SI *`lil`*
`lis`	`RT,UI`	RT = (UI ! 0x0000) *`liu`*
`lmw`	`RT,D(RA)`	Load consecutive words from D+(RA|0) to RT...R31 *`lm`*
`lwa`	`RT,D(RA)`	✓✁ Load Word Algebraic
`lwa[u]x`	`RT,RA,RB`	✓✁ Load Word Algebraic Indexed
`lwarx`	`RT,RA,RB`	✓ Load Word and Reserve Indexed
`lwbrx`	`RT,RA,RB`	RT <= Byte-reversed word from RB+(RA|0) *`lbrx`*
`lwz[u]`	`RT,D(RA)`	RT <= Word from D(RA) *`l[u]`*



[Figure 2.1] Logical view of CPU functional units

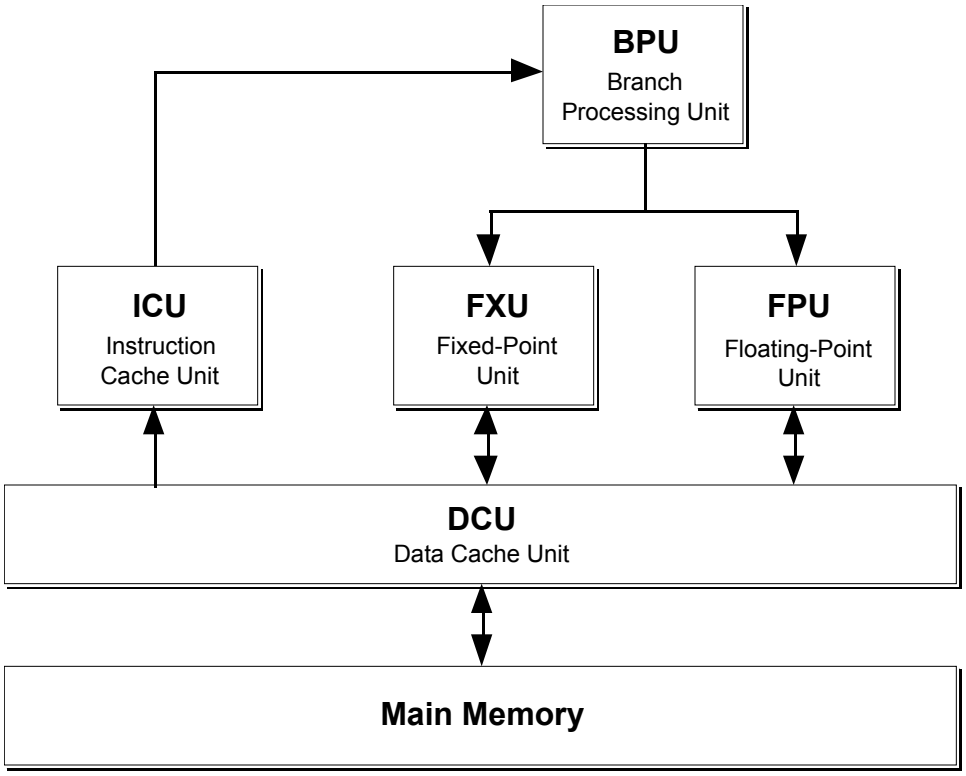
`lwz[u]x`	`RT,RA,RB`	RT <= Word from RB+(RA|0) *`l[u]x`*

Store:

`stb[u]`	`RS,D(RA)`	Store byte in RS into D+(RA|0)
`stb[u]x`	`RS,RA,RB`	Store byte in RS into RB+(RA|0)
`std[u]`	`RS,D(RA)`	✓✁ Store Doubleword
`std[u]x`	`RS,RA,RB`	✓✁ Store Doubleword Indexed
`stdcx`	`RS,RA,RB`	✓✁ Store Doubleword Conditional Indexed
`stfd[u]`	`FRS,D(RA)`	Store doubleword in FRS into D+(RA|0)
`stfd[u]x`	`FRS,RA,RB`	Store doubleword in FRS into RB+(RA|0)
`stfiwx`	`FRS,RA,RB`	✓✁ Store FP as Integer Word Indexed
`stfs[u]`	`FRS,D(RA)`	Store word in FRS into D+(RA|0)
`stfs[u]x`	`FRS,RA,RB`	Store word in FRS into RB+(RA|0)
`sth[u]`	`FRS,D(RA)`	Store halfword in FRS into D+(RA|0)
`sth[u]x`	`RS,RA,RB`	Store halfword in FRS into RB+(RA|0)
`sthbrx`	`RS,RA,RB`	Store byte-reversed halfword in RS into RB+(RA|0)
`stmw`	`RS,D(RA)`	Store RS...R31 into consecutive mem starting at D+(RA|0) *`stm`*
`stwbrx`	`RS,RA,RB`	Store byte-reversed word in RS into D+(RA|0) *`stbrx`*
`stwcx.`	`RS,RA,RB`	✓ Store Word Conditional Indexed
`stw[u]`	`RS,D(RA)`	Store word in RS into D+(RA|0) *`st[u]`*
`stw[u]x`	`RS,RA,RB`	Store word in RS into RB+(RA|0) *`st[u]x`*

String Operations:

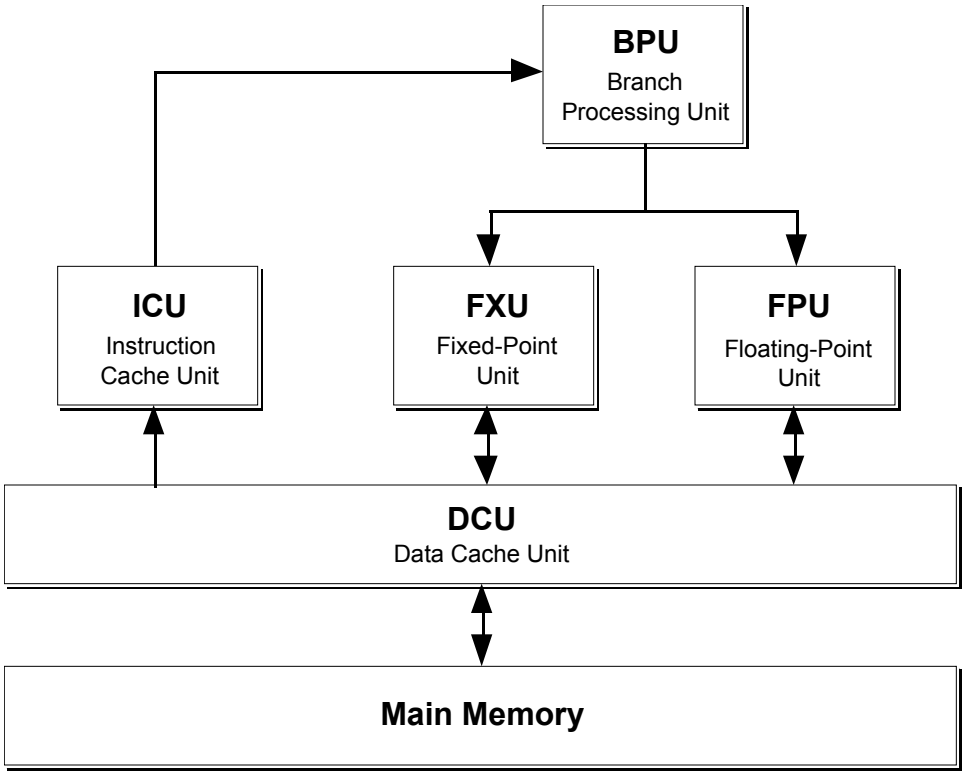
`lscbx[.]`	`RT,RA,RB`	✗✧• Load string and compare byte indexed
`lswi`	`RT,RA,NB`	Load string immediate *`lsi`*
`lswx`	`RT,RA,RB`	Load string indexed *`lsx`*



[Figure 2.1] Logical view of CPU functional units

`stswi`	`RS,RA,NB`	Store string immediate *stsi*
`stswx`	`RS,RA,RB`	Store string indexed *stsx*

[Table A.1] Instruction Set Summary (cont.)



[Figure 2.1] Logical view of CPU functional units

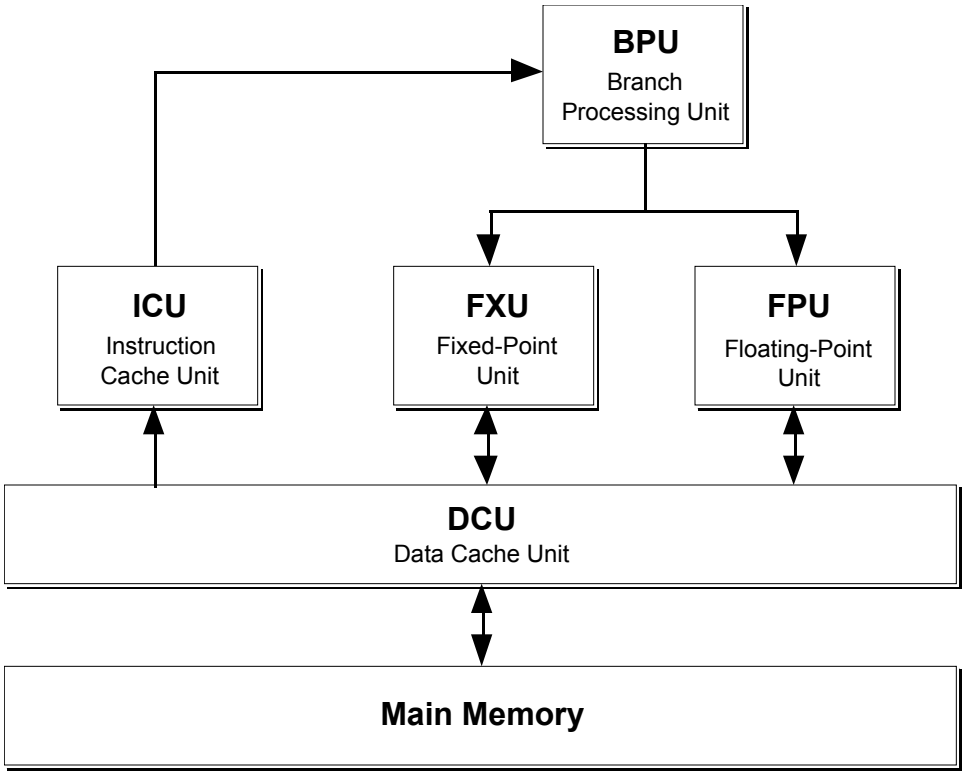
Shift, Rotate and Mask Instructions

Shift Left:

`sld[.]`	`RA,RS,RB`		✓✁ Shift Left Doubleword
`sle[.]`	`RA,RS,RB`	RA = (RS << RB) ; MQ = (RS lrot RB)	✗✧• Shift Left Extended
`sleq[.]`	`RA,RS,RB`	RA = (RS!MQ << RB) ; MQ = (RS lrot RB)	✗✧• Shift Left Extended with MQ
`sliq[.]`	`RA,RS,SH`	RA = (RS << SH) ; MQ = (RS lrot SH)	✗✧• Shift Left Immediate with MQ
`slliq[.]`	`RA,RS,SH`	RA = (RS!MQ << SH) ; MQ = (RS lrot SH)	✗✧• Shift Left Long Immed w. MQ
`sllq[.]`	`RA,RS,RB`	RA = (RS << SH) | (MQ & mask)	✗✧• Shift Left Long with MQ
`slq[.]`	`RA,RS,RB`	RA = (RS<<RB)&mask ; MQ=(RS lrot RB)	✗✧• Shift Left with MQ
`slw[.]`	`RA,RS,RB`	RA = (RS << RB)	• Shift Left *`sl[.]`*

Shift Right:

`srad[.]`	`RA,RS,RB`		✓✁ Shift Right Algeb Doubleword
`sradi[.]`	`RA,RS,SH`		✓✁ Shift Right Algeb Dbl Immed
`sraiq[.]`	`RA,RS,SH`	RA = (RS >> SH) ; MQ = (RS rrot SH)	✗✧• Shift Right Algeb. Imm. w. MQ
`sraq[.]`	`RA,RS,SH`	RA = (RS >> SH) ; MQ = (RS rrot SH)	✗✧• Shift Right Algebraic with MQ
`sraw[.]`	`RA,RS,RB`	RA = (RS >> RB)	• Shift Right Algebraic *`sra[.]`*
`srawi[.]`	`RA,RS,SH`	RA = (RS >> SH)	• Shift Right Algeb. Imm. *`srai[.]`*
`srd[.]`	`RA,RS,RB`		✓✁ Shift Right Doubleword
`sre[.]`	`RA,RS,RB`	RA = (RS >> RB) ; MQ = (RS rrot RB)	✗✧• Shift Right Extended
`srea[.]`	`RA,RS,RB`	RA = (RS >> RB) ; MQ = (RS rrot RB)	✗✧• Shift Right Extended Algebraic



[Figure 2.1] Logical view of CPU functional units

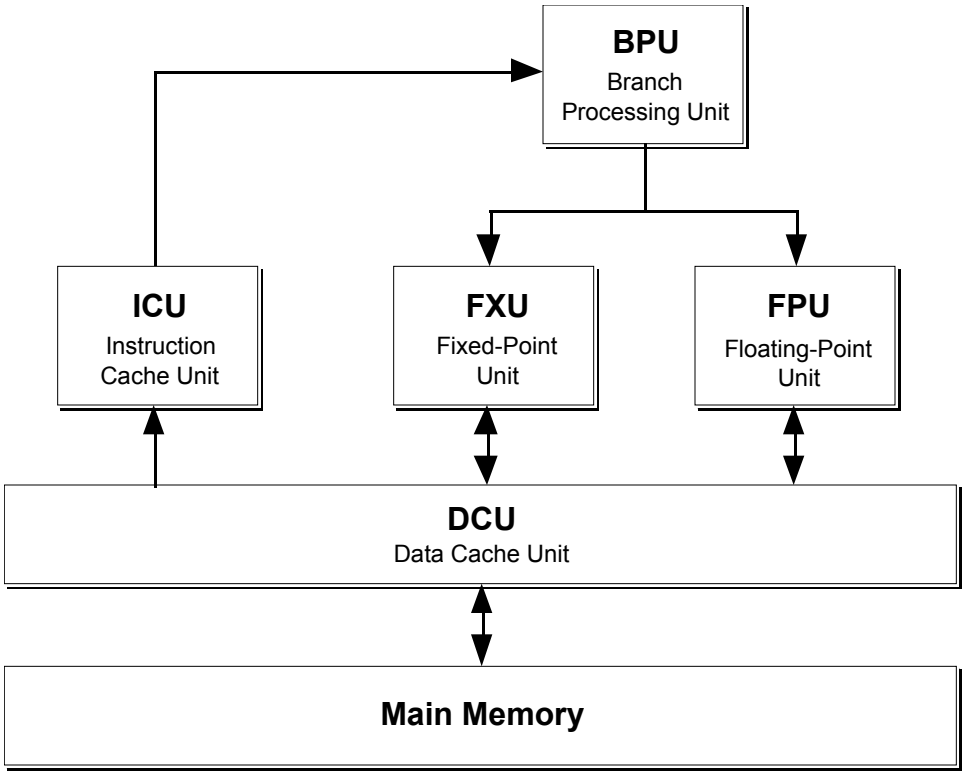
`sreq[.]`	`RA,RS,RB`	RA = (RS >> RB) ; MQ = (RS rrot RB)	✗✧• Shift Right Extended with MQ
`sriq[.]`	`RA,RS,SH`	RA = (RS >> SH) ; MQ = (RS rrot SH)	✗✧• Shift Right Immed with MQ
`srliq[.]`	`RA,RS,SH`	RA = (RS >> SH) ; MQ = (RS rrot SH)	✗✧• Shift Right Long Imm. w. MQ
`srlq[.]`	`RA,RS,RB`	RA = (RS >> RB)	✗✧• Shift Right Long with MQ
`srq[.]`	`RA,RS,RB`	RA = (RS >> RB) & mask	✗✧• Shift Right with MQ
`srw[.]`	`RA,RS,RB`	RA = (RS >> RB)	• Shift Right *`sr[.]`*

Rotate:

`rldcl[.]`	`RA,RS,RB,BM`		✓✁ Rotate Left Dblword; Clear Lft
`rldcr[.]`	`RA,RS,RB,BM`		✓✁ Rotate Left Dblword; Clr Right
`rldic[.]`	`RA,RS,SH,BM`		✓✁ Rotate Left Dblword Imm; Clr
`rldicl[.]`	`RA,RS,SH,BM`		✓✁ Rotate Left Dbl Imm; Clear Lft
`rldicr[.]`	`RA,RS,SH,BM`		✓✁ Rotate Left Dbl Imm; Clear Rt
`rldimi[.]`	`RA,RS,SH,BM`		✓✁ Rotate Left Dbl Imm; Mask Ins.
`rlmi[.]`	`RA,RS,RB,BM`	RA = ((RS << RB) & BM) | (RA & ~BM)	✗✧• Rotate Left; Mask Insert
`rlwimi[.]`	`RA,RS,SH,BM`	RA = ((RS << SH) & BM) | (RA & ~BM)	• Rot. L. Imm.; Mask Ins. *`rlimi[.]`*
`rlwinm[.]`	`RA,RS,SH,BM`	RA = (RS << SH) & BM	• Rot. L. Imm.; & Mask *`rlinm[.]`*
`rlwnm[.]`	`RA,RS,RB,BM`	RA = (RS << RB) & BM	• Rotate L.; & w. Mask *`rlnm[.]`*

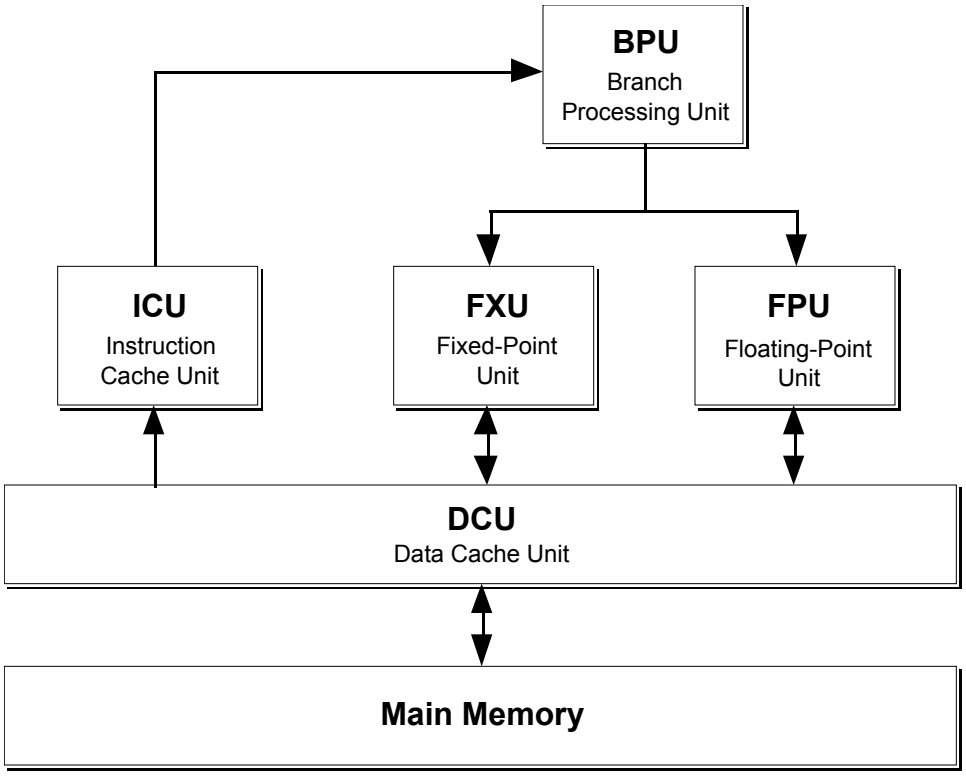
Bit Mask:

`maskg[.]`	`RA,RS,RB`	RA = mask from RS and RB	✗✧• Mask Generate
`maskir[.]`	`RA,RS,RB`	RA = (RS & RB) | (RA & ~RB)	✗✧ Mask Insert from Register
`rrib[.]`	`RA,RS,RB`	RA[RB] = RS[0]	✗✧• Rotate Right and Insert Bit



[Figure 2.1] Logical view of CPU functional units

[Table A.1] Instruction Set Summary (cont.)



[Figure 2.1] Logical view of CPU functional units

Comparison Instructions

`cmpw`	`BF,RA,RB`	CR{BF} = compare of RA and RB	Compare *`cmp`*
`cmpwi`	`BF,RA,SI`	CR{BF} = compare of RA and SI	Compare Immediate `cmpi`
`cmplw`	`BF,RA,RB`	CR{BF} = unsigned compare of RA and RB	Compare Logical *`cmpl`*
`cmplwi`	`BF,RA,UI`	CR{BF} = unsigned compare of RA and UI	Compare Logical Immed *`cmpli`*

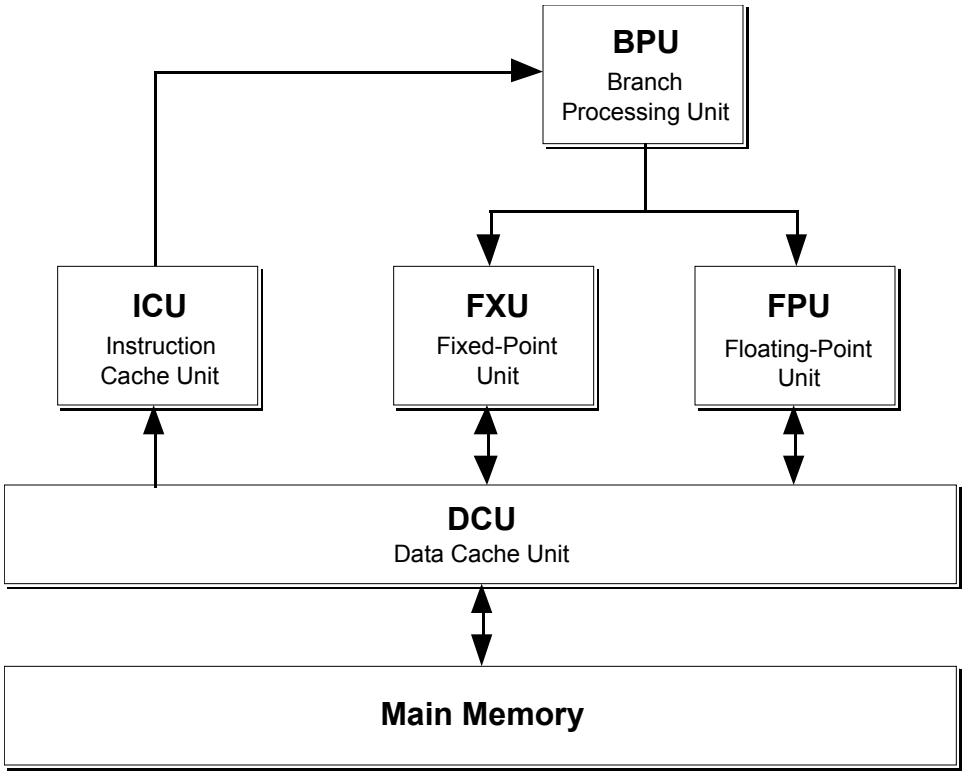
Flow of Control Instructions

Branch Instructions:

`b[l][a]`	*`<addr>`*	branch to address
`bbt[l]`	`BIT,`*`<addr>`*	branch if BIT of CR is set
`bbf[l][a]`	`BIT,`*`<addr>`*	branch if BIT of CR is clear
`bc[l][a]`	`BO,BI,`*`<addr>`*	branch on BO condition
`bcctr[l]`	`BO,BI`	branch to address in CTR on BO condition *`bcc[l]`*
`bclr[l]`	`BO,BI`	branch to address in LR on BO condition *`bcr[l]`*

Trap & Supervisor Instructions:

`rfi`		return from trap/interrupt	Return From Interrupt
`rfsvc`		return from svc	Return From Supervisor Call
`svc[l]`	`LEV,FL1,FL2`	generate supervisor call interrupt	✗ Supervisor Call
`sc`		generate supervisor call interrupt	Supervisor Call Absolute *`svc[l]a`*
`td`	`TO,RA,RB`	trap when TO condition is true	✓✁ Trap Doubleword
`tdi`	`TO,RA,SI`	trap when TO condition is true	✓✁ Trap Doubleword Immediate



[Figure 2.1] Logical view of CPU functional units

`tw`	`TO,RA,RB`	trap when TO condition is true	Trap Word *t*
`twi`	`TO,RA,SI`	trap when TO condition is true	Trap Word Immediate *ti*

System Instructions

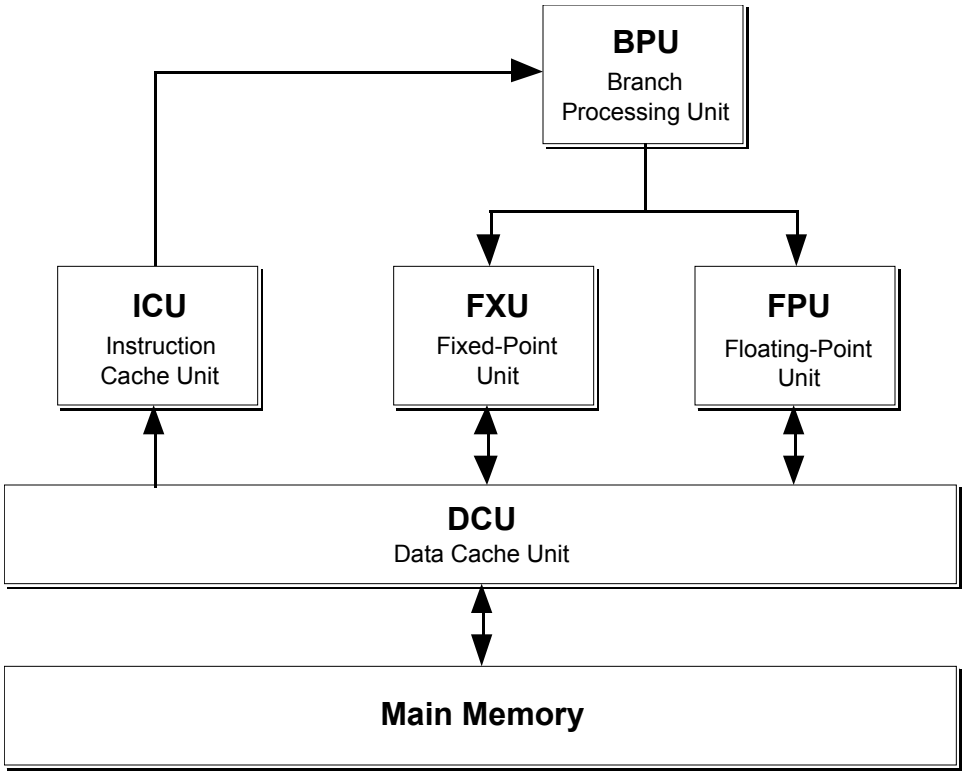
Look-aside Buffer Instructions:

`slbia`		invalidate entire SLB	✓✁ SLB Invalidate All
`slbie`	`RB`	invalidate SLB entry	✓✁ SLB Invalidate Entry
`slbiex`	`RB`	invalidate SLB entry	✓✁ SLB Invalidate Entry by Index
`tlbia`		invalidate entire TLB	✓✁ TLB Invalidate All
`tlbie`	`RB`	invalidate TLB entry	TLB Invalidate Entry *tlbi*
`tlbiex`	`RB`	invalidate TLB entry	✓✁ TLB Invalidate Entry by Index
`tlbsync`		invalidate TLB entry	✓✁ TLB Synchronize

Move To/From Special Registers:

`mfmsr`	`RT`	RT = MSR	Move From Machine State Register
`mfspr`	`RT,SPR`	RT = SPR	Move From Special Purpose Register
`mfsr`	`RT,SR`	RT = SR	Move From Segment Register
`mfsrin`	`RS,RB`	RS = SR selected from RB	✗✧ Move From SR Indirect *mfsri*
`mftb`	`RT,TBR`		✓ Move From Time Base
`mftbu`[13]†			✓ Move From Time Base Upper
`mr`	`RA,RS`	same as "`or RA,RS,RS`"	✓ Move Register
`mtmsr`	`RS`	MSR = RS	Move To Machine State Register

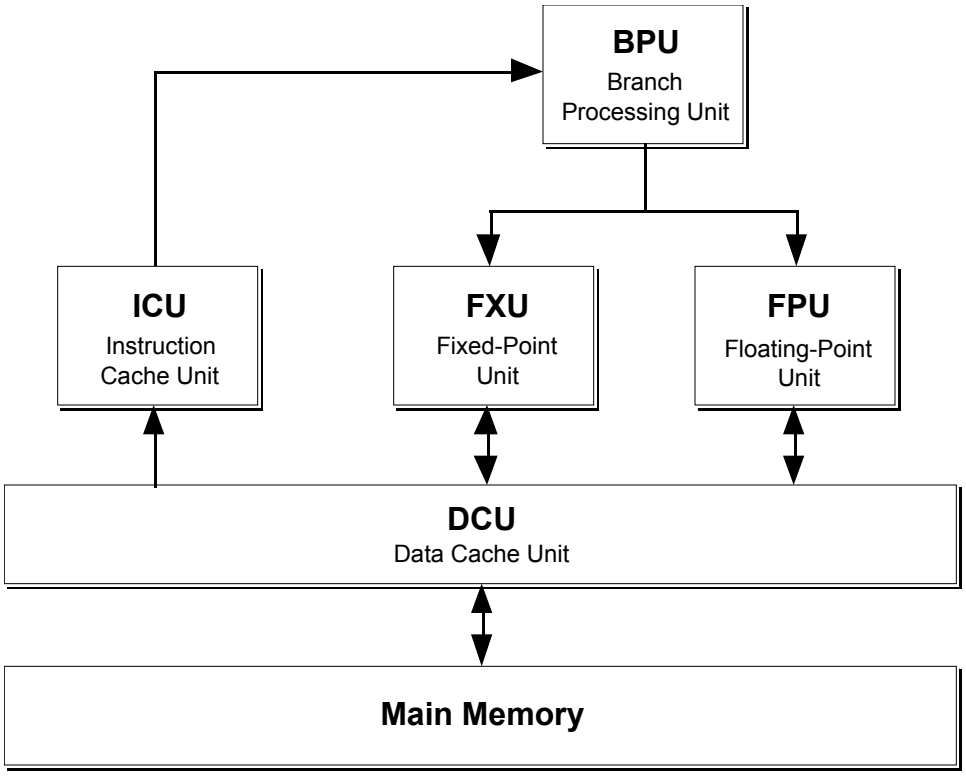
[13]† The `mftbu` instruction is listed as a new instruction in [Case92], but is not included in the PowerPC User's Manual.



[Figure 2.1] Logical view of CPU functional units

`mtspr`	`SPR,RS`	SPR = RS	Move To Special Purpose Register
`mtsr`	`SR,RS`	SR = RS	Move To Segment Register
`mtsrin`	`RS,RB`	SR selected from RB = RS	Move To SR Indirect *`mtsri`*

[Table A.1] Instruction Set Summary (cont.)

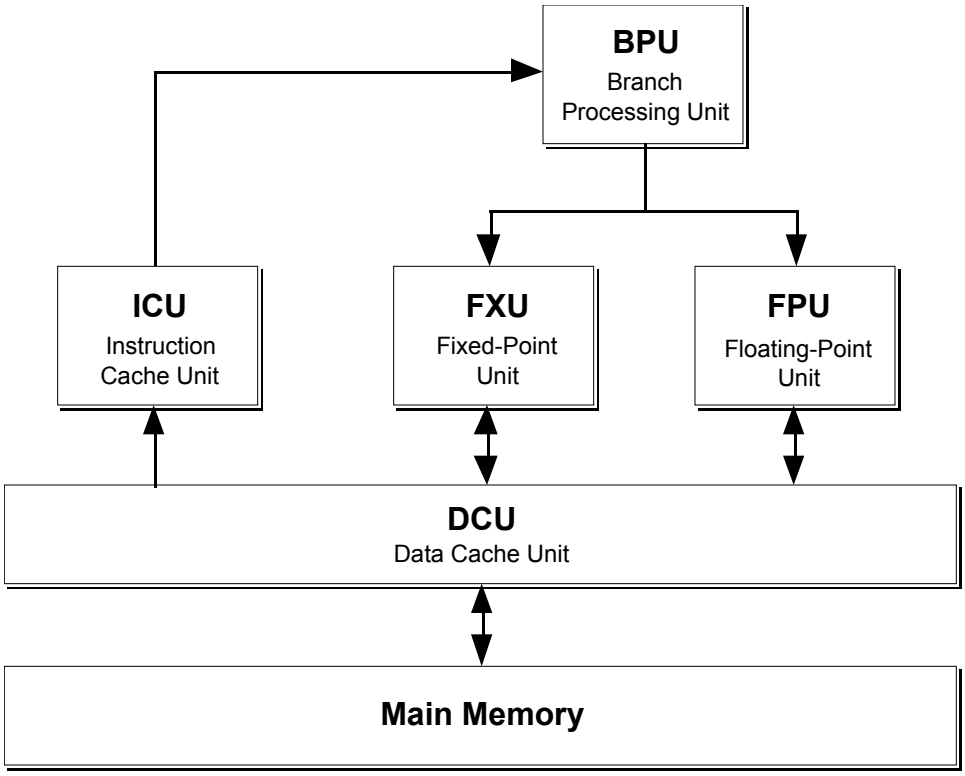


[Figure 2.1] Logical view of CPU functional units

Cache & I/O Operations:

clcs	RT,RA	RT = cache line size	✗✧ Cache Line Compute Size
clf	RA,RB	flush a data cache line	✗ Cache Line Flush
cli	RA,RB	invalidate a data cache line	✗ Cache Line Invalidate
dcbf	RA,RB	flush a data cache block	✓ Data Cache Block Flush
dcbi	RA,RB	invalidate a data cache block	✓ Data Cache Block Invalidate
dcbst	RA,RB	store block from cache into memory	✓ Data Cache Block Store
dcbt	RA,RB	prefetch block into cache	✓ Data Cache Block Touch
dcbtst	RA,RB	prefetch block into cache	✓ Data Cache Block Touch For Store
dcbz	RA,RB	set cache block to all 0's	✓ Data Cache Block Set to Zero
dclst	RA,RB	store data from cache into memory	✗ Data Cache Line Store
dclz	RA,RB	set bytes of the data cache to 0	✗ Data Cache Line Set to Zero
eciwx	RT,RA,RB		✓ External Control Input Word Indexed
ecowx	RT,RA,RB		✓ Ext. Control Output Word Indexed
eieio			✓ Enforce In-order Execution of I/O
icbi	RA,RB	invalidate an instruction cache block	✓ Instruction Cache Block Invalidate
isync		synchronize the instruction cache	Instruction Cache Synchronize *ics*
rac[.]	RT,RA,RB	RT = RB + (RA|0)	✗• Real Address Compute
sync		synchronize the data cache	Synchronize Data Cache *dcs*

[Table A.1] Instruction Set Summary (cont.)



[Figure 2.1] Logical view of CPU functional units

Appendix B: Summary of Instruction Set Changes to POWER for PowerPC

This information was derived from [Case91,92], [Oehler92], [Diefendorff93], and [Motorola93].

- All instructions which rely on the MQ register have been eliminated. These 18 instructions are: `mul[o][.]`, `div[o][.]`, `divs[o][.]`, `sle[.]`, `sleq[.]`, `sliq[.]`, `slliq[.]`, `sllq[.]`, `slq[.]`, `sraiq[.]`, `sraq[.]`, `sre[.]`, `srea[.]`, `sreq[.]`, `sriq[.]`, `srliq[.]`, `srlq[.]`, and `srq[.]`.

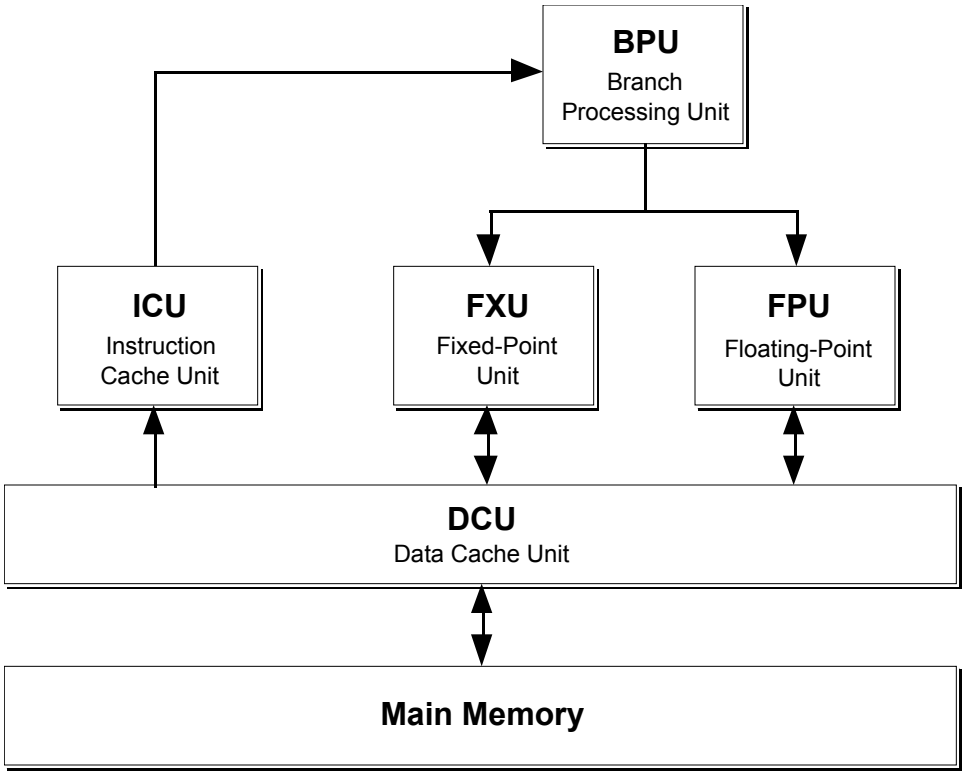
- Instructions whose operation was data dependent have been eliminated. These instructions would require an additional multiplexer which would increase the critical path of the processor. These 4 instructions are: `abs[o][.]`, `doz[o][.]`, `dozi`, and `nabs[o][.]`.

- Instructions which required three reads from the general register file have been removed. These instructions include: `maskir[.]`, `rrib[.]`, and `rlmi[.]`.

- Various instructions which loaded/stored multiple words were either eliminated or removed from microcode and trap-emulated instead. These affected instructions are: `lscbx[.]` (removed), `lsi`, `lsx`, `stsi`, and `stsx` (trap-emulated).

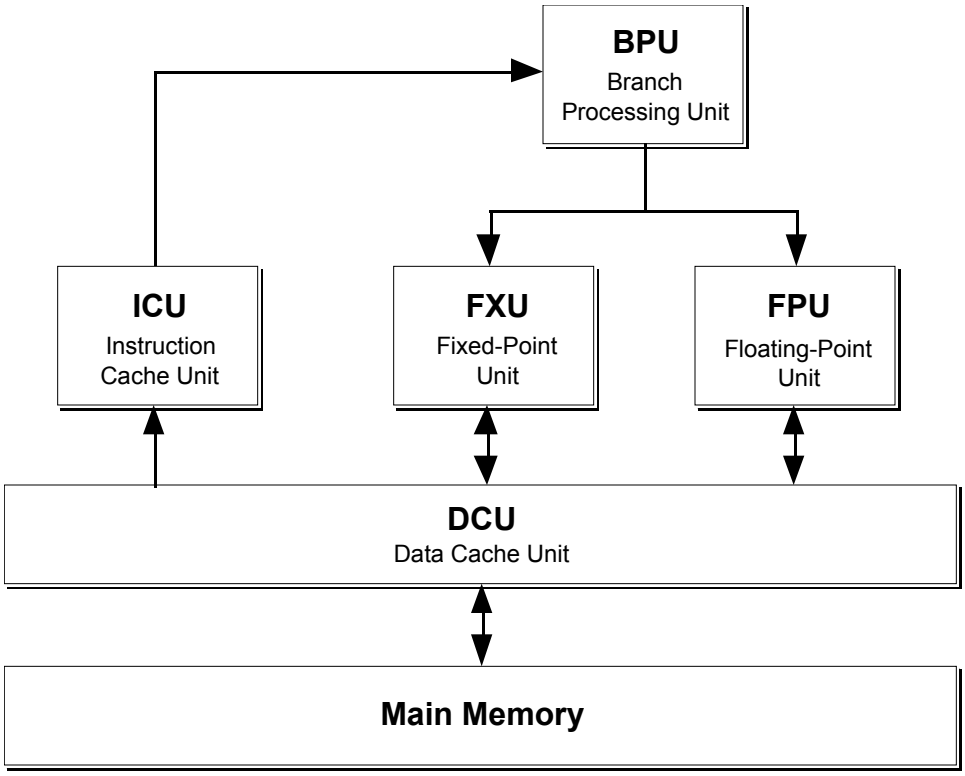
- The mask generation (`maskg[.]`) instruction has been removed to reduce mask complexity.

- Hardware checks for rarely used “corner cases” have been removed. For example, the results of a `lu r14,4(r14)` are now undefined. Under the POWER architecture, this condition is detected and the register is loaded with the data and not with the updated address.



[Figure 2.1] Logical view of CPU functional units

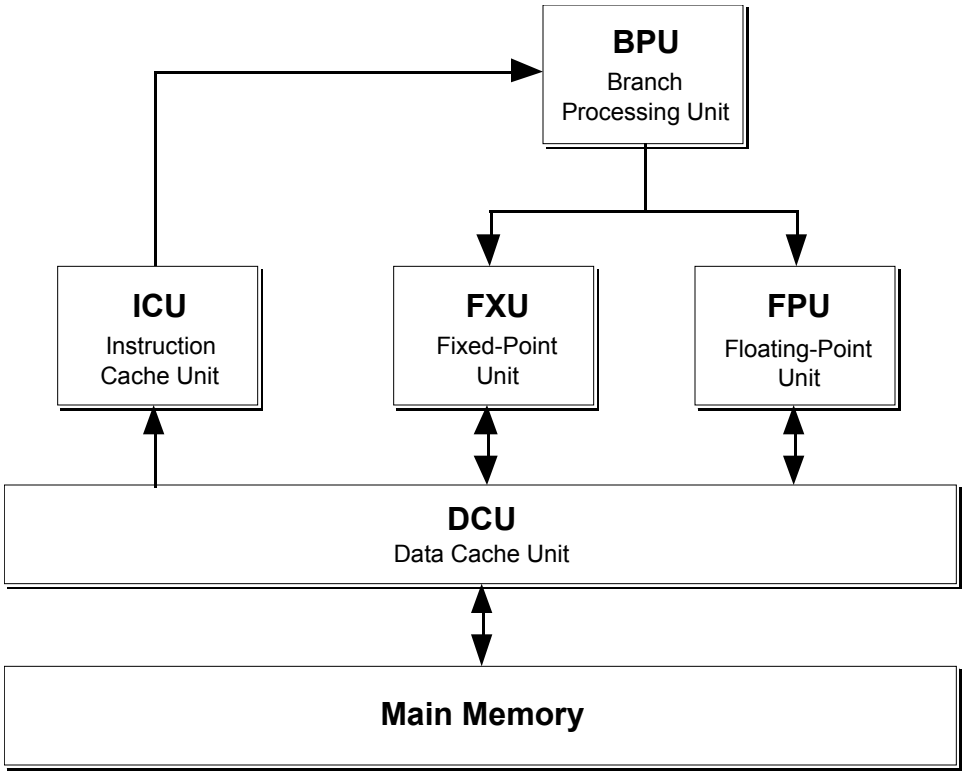
- The requirement that all instructions execute in 1 cycle (with a few exceptions, of course) has been relaxed so that a larger number of instructions now require multiple cycles.
- A fixed-point subtract instruction which does not update the Carry (CA) bit of the XER was added (`subf[o][.]`)
- New signed multiply and divide operations were added to replace the ones removed, and unsigned integer multiply and divide instructions were added.
- Single-precision floating-point instructions were added.
- A floating-point square root instruction was added to make IEEE conformance easier.
- Floating-point to integer (word and doubleword) instructions were added.
- Instructions were added to assist the programmer with synchronizing multi-threaded applications. Additional operations include: load and reserve (`lXarx`) and store conditional (`stXcx`).
- Instructions were added to control data moving into and out of the processor cache.
- A weak-ordered storage model was defined. This means that the programmer must explicitly synchronize shared memory before accessing the data.
- Extensions were made to the architecture to support both 32-bit and 64-bit memory models. These extensions include a large number of instructions to support doubleword operations.



[Figure 2.1] Logical view of CPU functional units

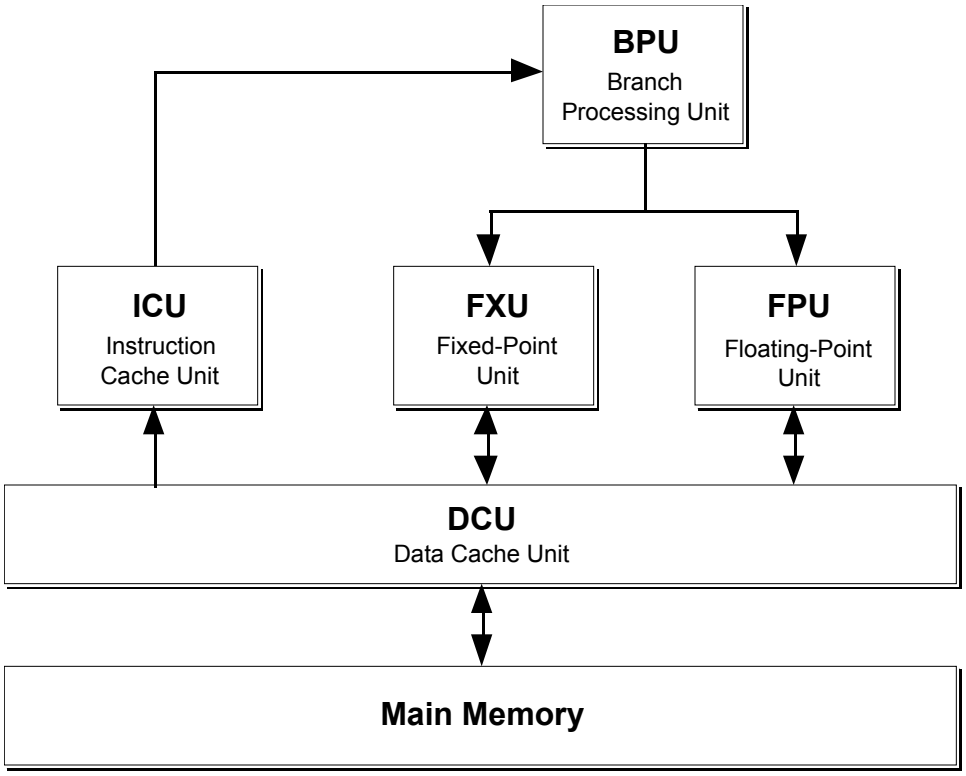
Appendix C: Sample RS/6000 Assembly Program

```
# sample.s
# sample RS/6000 assembly language program
# Copyright 1993 Gary J Kacmarcik
#
# this is sample RS/6000 assembly code which is meant to be linked with
# the standard C libraries.
#
# build the executable using:
#       cc -o sample sample.s

# because we are using the C compiler (so that we can reference the .printf
# function), we must make our main routine globally visible to the linker
# the "pr" in the brackets after the symbol name identifies this symbol
# as belonging to the .text section (program area)
        .globl   .main[pr]

# since printf is defined elsewhere, we must declare it
        .extern  .printf[pr]

# define our entries in the Table Of Contents
# here is where we define entries for functions, function descriptors, and
# external variables
        .toc
T.data:  .tc      data[tc],data[rw]              # TOC for data area
T.main:  .tc      .main[tc],main[ds]      # TOC for main function descriptor
```



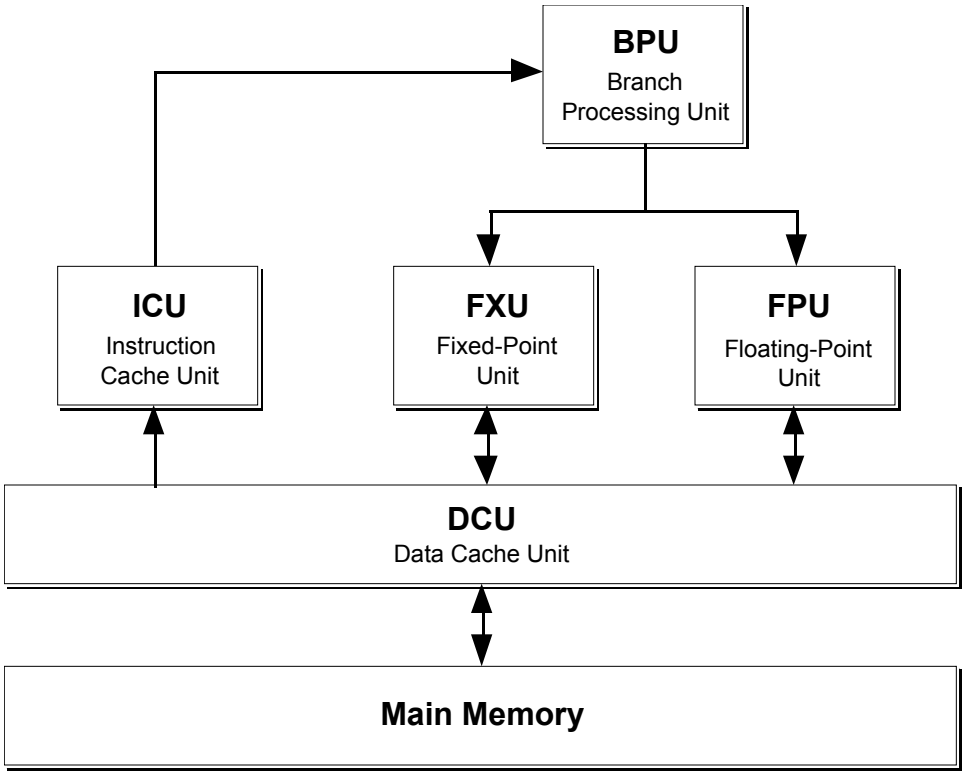
[Figure 2.1] Logical view of CPU functional units


```
# define the function descriptor for main
# these belong in a csect of type "ds"
# function descriptors are used to describe function pointers in various
# high-level languages (C, FORTRAN)
        .csect   main[ds]
        .long    .main[pr],TOC[tc0],0

################################
# the main function (finally) #
################################

        .csect   .main[pr]

        # define a few assembler variables which define our stack area
        .set     ngprs,1          # of gpr's we need to save
        .set     nfprs,0          # of fpr's we need to save
        .set     regarea,8*nfprs+4*ngprs       # size of register save area
        .set     argarea,32       # size of output argument area
        .set     linkarea,24      # size of link area
        .set     localarea,0      # size of local stack area
        .set     totalsize,regarea+argarea+linkarea+localarea

        # function prolog
        # these actions must be performed at the beginning of each function
        # save the LR if this is not a leaf routine
        mflr     r0               # get the Link Register
        st       r0,8(r1)         # save the LR on stack
        # save the CR if it is modified in this routine
        mfcr     r0               # get the Condition Register
        st       r0,4(r1)         # save the CR on the stack
```



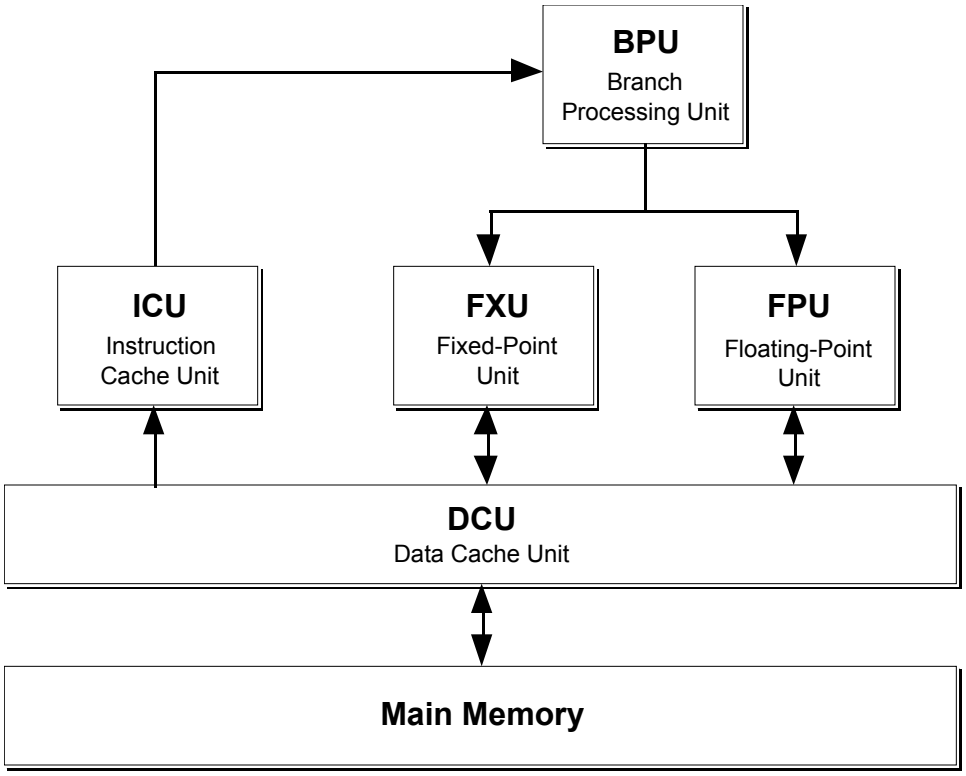
[Figure 2.1] Logical view of CPU functional units


```
# save any GPR's and FPR's which are used by this routine
# gpr’s get saved at -regarea(r1)
# fpr’s get saved at -8*nfprs(r1)
stm     r31,-regarea(r1) # we only use register r31
# decrement the stack pointer and save the stack back-chain
stu     r1,-totalsize(r1)
```



[Figure 2.1] Logical view of CPU functional units


```
# load the address of our string from the .data section
# to do this we must first get the address of the our area in
# .data section.  we accomplish this by using the TOC entry
# that we set up for the data[rw] csect
l       r14,T.data(r2)
# now r14 points to the beginning of our csect in the .data section.
# get the address of our string by adding the offset from the start
# of the csect to the strin
cal     r3,_str1(r14)

# call the printf function
# because the function is not defined in the same module as
# this function, the linker will first call some global linkage
# code to set up the called routine's proper TOC.
# We must add a dummy instruction (cror 31,31,31) after the
# function call which will be overwritten with an instruction
# to restore our local TOC. (l r2,20(r1))
bl      .printf[pr]
cror    31,31,31

# load the address of the second string
cal     r3,_str2(r14)

# call printf again
bl      .printf[pr]
cror    31,31,31

# function epilog
# these actions must be performed at the end of each function
```



[Figure 2.1] Logical view of CPU functional units


```
        # restore the LR that we saved earlier
        l       r0,totalsize+8(r1)
        mtlr    r0
        # restore the stack pointer
        ai      r1,r1,totalsize
        # restore any registers that we saved
        lm      r31,-regarea(r1)

        # end of function - return
        brl

# data area
# since all of the r/w data needs to be stored in the .data section, we start
# a new csect
        .csect  data[rw]
        .align  2               # make sure its aligned properly

# the first string
_str1:  .byte   "hell"
        .byte   10,0

# the second string
_str2:  .byte   "oh world"
        .byte   10,0
```