Short Description Of The PowerPC 601 Processor (3/94)

Article Created: 8 March 1994

TOPIC ---

This is a short description of the features of the PowerPC 601 processor which will be used for Apple Computer's first RISC products.

The organization of this description follows that of the PowerPC ISA document. Some familiarity with processor architectures, particularly of RISC processors, is assumed in the following. Since the PowerPC 601 is close to "traditional" RISC processors in many ways, the amount of detail in this document will reflect areas which are different. In areas where PowerPC (and/or the 601) are similar to normal chips, very little verbage will be spent.

DISCUSSION ---

Architecture

The 601 is the first chip to implement the new PowerPC architecture (as described in the Version 1.01 documents). However, since it is intended as a bridge chip for IBM, it also includes the existing POWER instruction set. It is, therefore, the union of the two instruction sets.

There are several areas where the architectures differ in a rather fundamental way (for example, virtual address translation); in these areas, PowerPC prevails.

Branch Processor

The Branch Processor is logically responsible for all instruction fetching of the PowerPC architecture. It also decodes instructions to determine to which execution unit they should be sent for execution. The 601 uses pre-fetching to attempt to keep ahead of the execution units. Up to 9 instructions are contained within the Branch Processor, in a so-called "pre-fetch pipeline". In the 601, the bottom stage of the pipeline is what is normally referred to as the "Decode" stage of traditional RISC pipelines.

As instructions fall out of the bottom of the pipe (that is, proceeding from Decode to Execute, by being sent to the Fixed Point and/or Floating Point processors), the pipeline is filled by the pre-fetch logic.

Branch Instruction Processing:

The Branch Processor in PowerPC (as in POWER) "executes" all branch instructions. Note that "executing" a branch simply means to continue fetching from a different location than the current sequential stream implies. The 601 examines the last 4 stages of the pre-fetch pipeline, looking for branches.

Unconditional branches cause the pipeline stages above (and, including) the stage in which the branch is detected to be "flushed". Fetching of instructions to re-fill the pipe are then made from the "target address" of the branch. (Note that a conditional branch whose condition is determined early enough is processes the same as an unconditional branch.)

Conditional branches whose condition is not yet known when processed by the Branch Processor are "predicted". That is, instruction pre-fetching will be attempted along a path (taken vs. not taken) which is "guessed" by the hardware. In the 601, this prediction is based upon the direction of the branch (that is, the sign of its displacement). A "backwards" branch is predicted as taken; a "forwards" branch is predicted as not taken.

When the first instruction of a predicted path reaches the Decode stage (that is, the bottom of the pre-fetch pipe), and the condition is not yet known, the Branch Processor will stall, waiting for an indication of whether its prediction was correct. When the condition becomes known, the Branch Processor will either let the instruction in Decode proceed to Execute (if it predicted correctly) or flush the entire pipeline and re-fetch from the correct path.

When a branch is detected early enough, the pipe can be refilled before it would run "dry". As long as branches can be detected (and, for conditional branches, the proper direction can be determined) early enough, branches "execute" in "zero" time. In the 601, this translates into a general rule that branches which are separated by at least two non-branch instructions will execute without branch-induced delays.

In the 601, only one "predicted" conditional branch can be "outstanding" at any time.

Mini-TLB.

Code fetches have to be translated like any other memory access. In order to minimize the overhead of translating every code fetch, the Branch Processor contains a copy of the last four page translations which were most recently fetched. Any fetch to one of these pages will not cause any explicit address translation. However, when a branch is taken to a page not contained within this mini-TLB, the branch is sent to the Fixed Point Processor, which performs the address translation. The resulting translation will update one of the 4 entries in the mini-TLB.

Accesses to addresses not contained in the mini-TLB will not be attempted until the branch is known to be taken. Thus, conditional branches will incur extra delay if they are not to one of the 4 most recently used pages.

Non-Branches

PowerPC includes a group of instructions which primarily operate upon the Condition Register. They are described within the Branch Processor chapter, because they were processed by the Branch Processor in the original POWER chip-set.

However, in the 601, only branch instructions are processed by the Branch Processor. All of the other "branch processor" instructions are actually executed by either the Fixed Point Processor or the "Sequencer".

Fixed Point Processor

The Fixed Point Processor is primarily intended to execute the Fixed Point instructions of PowerPC. However, as mentioned above, the 601's Fixed Point Processor also executes most of the non-branch instructions described in the PowerPC Branch Processor chapter.

The Fixed Point Processor also performs all address translation. Thus, all loads and stores (both Fixed and Floating Point) have their address generation (that is, computing the Effective Address) and translation (converting from EA->VA->RA) executed within the Fixed Point Processor. (A complete description of Address Translation appears as a separate section in this document).

In general, all Fixed Point instructions execute in one clock, and the result of the operation is immediately avaiable to a successive instruction. The exceptions to this rule are Loads (discussed below), Multiplies (which take 5 clocks for short results and 9 clocks for long) and Divides (which take up to 36 clocks). Multply and Divide instructions actually "stall" in the Execute stage of the Fixed Point Processor, thereby preventing execution of any following Fixed Point instructions until they complete.

The important timing number of Loads is its "latency". For example, if one has a "dependent" operation, how many extra clocks are required. In the 601, assuming that the data is aligned and in the cache, one "extra" clock is required to load the register with the correct data. This implies a general rule that one should have at least one instruction between a load and any dependent instruction in order to eliminate extra clocks.

Floating Point Processor

The Floating Point Processor executes all of the Floating Point instructions of PowerPC. In general, Floating Point instructions take 3 clocks to produce a result, with the exception being FP Divide, which takes up to 31 clocks. Thus, sequences of dependent Floating Point computations will execute at 1 every 3 clocks. However, sequences of Adds, Subtracts and Multiplys which can overlap their computations so that no dependencies are within the 3 clock latency will execute at 1 per clock.

A Floating Point Add or Subtract can be "issued" every clock, as long as its sources are available. Floating Point Multiplys can be issued every other clock. A Floating Point Divide stalls the entire unit.

Note that all Floating Point Loads and Stores require processing by the Fixed Point Processor (for address computation and translation). Thus, they fill a

slot in both units.

Address Translation

The 601 implements the PowerPC translation mechanism, which is different from that of the original RS/6000s and the RSC. (See the Storage Control chapter in the PowerPC Operating Environment Architecture document for details.)

The PowerPC documents define the memory structure which is used for address translation (the Hash Table). Like most processors, the 601 uses a Translation Lookaside Buffer (TLB) to "cache" recently used translations to minimize the overhead of a full "table lookup" for translations. Addresses which translate to "recently used" pages will be found within the TLB, thereby circumventing the complete process of table walking.

The 601's TLB is organized as a 2-way set-associative cache with 128 sets, using LRU updating. Thus, up to 256 translations are available with no extra processing time required. (Note: if an operating system uses the full capabilities of the PowerPC translation mechanism, no explicit "flushing" of the TLB is necessary.)

If a translation is not available within the TLB, the Sequencer is invoked to perform the actual Hash Table walking.

Cache

The 601 contains a 32 KB Unified cache, organized as 8-way set-associative, with 64 sets, using LRU updating. Each cache line is 64 Bytes, divided into two "sectors"; a sector is the unit which is processed as a single burst transaction on the bus. (The term Unified means that the cache is shared between Code and Data.)

The cache is normally run in "Store-In" (CopyBack). This means that stores are performed by updating the cache contents and marking that cache sector as "dirty". Subsequent re-use of the cache line will cause dirty sectors to be written to memory.

Sequencer

In addition to the "architected" functional units described in the PowerPC documents, the 601 also contains a micro-coded "Sequencer". This sequencer performs any of the "hard" tasks which the hardware can't. This includes such things as Loads and Stores to "I/O" space (that is, accesses where the Segment Register has its T-bit == 1), TLB misses, exception processing (that is, processing interrupts).

Note that the Sequencer is normally inactive. When it does become activated (for example, by a TLB miss), all processing in the Branch and Fixed Point and Floating Point Processors is suspended. In other words, the Sequencer "takes over" the hardware.

(The only reason for mentioning the Sequencer is that it is referred to in various other documents.)

Keywords: kppc

==========

This information is from the Apple Technical Information Library.

19960215 11:05:19.00
Tech Info Library Article Number: 14834