The C400 Superscalar/Superpipelined RISC Design

Lee Hanson and Nathan Brookwood

Intergraph Advanced Processor Division
Palo Alto, California

Abstract

The C400 represents the first complete reimplementation of CLIPPER architecture since Fairchild introduced the original C100 version in 1985. The design incorporates an entirely new pipeline structure that exploits instruction-level parallelism far more than its predecessors, and provides far greater computational performance than earlier CLIPPERs, in both absolute and frequency-adjusted comparisons. The combination of superscalar dispatch and deep floating-point pipelines provides "architectural headroom" that permits performance enhancements over the life of the implementation architecture. This paper discusses the C400's design goals, constraints, and architecture.

Introduction

Intergraph's new C400 CLIPPER is the outcome of a two-year program focused on the design of a high-end CLIPPER processor. Earlier CLIPPERs, the C100[1] introduced in 1985 and the C300[2] introduced in 1988, emphasized a high degree of functional integration on silicon. Both models combined integer and floating-point hardware on one chip, and cache and memory management functions on a second chip. An entire compute engine, including CPU, FPU and dual caches, resided on a small (3 inch by 4.5 inch) module. This high degree of integration forced design compromises with regard to pipeline organization, hardware algorithms and cache size. Intergraph recognized that improvements in process technology alone would not allow these earlier products to achieve the performance levels needed for technical computing systems in the 1990's, and in the fall of 1988, it set out to design a software-compatible high-end CLIPPER. The goals for the 'C4' program were:

- Improve integer performance by a factor of 3 - 4 over that of the C300 operating at 50 MHz.
- Improve floating-point performance by a factor of 6 - 8 over the C300's performance.
- Use several moderately sized die to implement processor functions, rather than one very large die, both to insure good production yields and to lower the program's technology risk.
- Provide an upwardly compatible binary environment that would allow the millions of lines of application code already written for the CLIPPER to continue to execute properly, so that customers could migrate to systems based on the new chipset at their own pace.
- Deliver first silicon to our customers by the end of 1990; i.e., take no longer than two years to design the chips.

Our analysis of performance on earlier CLIPPERs suggested several areas with substantial potential for improvement. The "high integration" focus of the earlier designs resulted in caches that were too small (4 KB) and too slow to support the integer performance that the CPU could theoretically deliver. Since cache access was not pipelined, back-to-back loads or stores introduced wait states, even for cache hits. Pipeline stalls due to branching impacted performance, especially in tight loops. The long latencies of many floating-point operations adversely impacted performance. The single-threaded design of the floating-point logic caused operations like floating-add with short latencies to back up behind ones with very long latencies like divide, and exacerbated the effect this had on performance. The chip's 32-bit datapaths limited the rate at which operands could be read from the 64-bit register file, and throttled double-precision performance. Our compilers paid scant attention to issues of code scheduling and sequencing data accesses, since CLIPPER's internal scoreboard ensured that the CPU would stall, rather than deliver incorrect results, if resources were accessed in the wrong order, or before internal results were written back to the register file.

The C400 system addresses the performance limitations of the earlier CLIPPERs, and achieves the goals outlined above. The processor, as illustrated in Figure 1, includes four major elements:

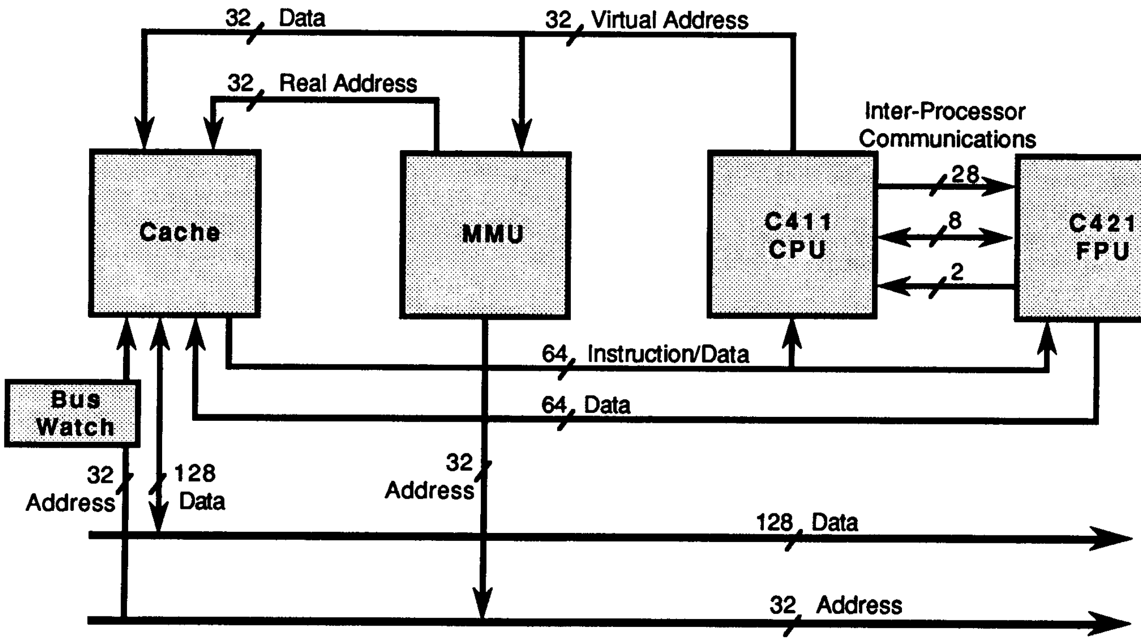
- The C411 Integer Unit (CPU) decodes and issues all instructions, and executes integer operations. It contains approximately 160,000 transistors, on a die measuring 253,000 square mils, and is packaged in a 299-pin ceramic PGA.
- The C421 Floating-Point Unit (FPU) incorporates a sixteen by 64-bit floating register file, along with the execution pipelines for floating add/subtract, multiply and divide. It contains approximately 140,000 transistors, on a die measuring 253,000 square mils, and is packaged in a 299-pin ceramic PGA.
- The Memory Management Unit handles virtual-to-physical address translation, using Translation Lookaside Buffers (TLBs) stored in discrete SRAMs.
- The Cache Unit provides a high-speed (single clock) 128 KB direct-mapped cache that supports the CPU's bandwidth requirements for instructions and data. Cache data and tags are stored in discrete SRAMs.

The entire CPU assembly fits on a printed circuit board with a 9U form factor, with a small daughter-board attached via a private connector.

The system uses 64-bit datapaths to link the CPU, FPU and cache. A single transfer from the cache to the CPU's instruction buffer can contain up to four variable-length instructions. Double-precision data can be moved from the cache to the FPU's register file in a single clock cycle; in general, there is only a minor performance penalty associated with the use of double precision arithmetic, compared with single-precision timings.

The CPU pipelines all accesses to the cache and main memory system. During any given clock cycle, the CPU can generate a new virtual address, while the MMU translates the previous virtual address, and the cache accesses the physical address calculated in the previous clock. Although the

Figure 1. C400 System Processor



Integer Unit coordinates all transfers between the cache and IU or the FPU, floating-point data moves directly between the cache and the FPU register file, and does not pass through the IU.

Although the initial version of the C400 implements its MMU and cache using discrete elements, a future version is planned that incorporates a VLSI MMU/cache mechanism. These 'cache chips,' along with the IU and FPU, will be packaged together in a single multi-chip module (MCM) that behaves in most regards like a large single chip. A companion paper[3] argues that this design technique provides a saner and more economic approach to complex silicon design than the currently fashionable "million-plus transistor chip" approach.

Integer Unit (IU) Design

The processor design includes nine distinct pipelines that handle loads, stores, branches and a variety of integer and floating-point execution elements. The integer unit fetches, decodes and issues all instructions, and manages all but the floating-point execution pipelines. To support the high degree of concurrency within the IU, the chip incorporates a multi-ported 32 by 32 register file with three read and two write ports. This custom-designed file uses an advanced circuit design that provides a 6 ns access time in a 1 μ CMOS process technology.

The integer unit fetches, decodes and issues *all* instructions, including floating-point instructions. The IU also supervises the execution of floating loads and stores. It signals the FPU regarding the appropriate register to load or store, but treats the operation like an integer operation in all other regards. The C400's superscalar dispatch logic can issue one integer and one floating instruction each clock cycle. This allows the IU to start a floating load (or store) and a floating operate instruction in the same clock, if there are no data dependencies in the instructions to be issued. This ability to combine the issue of floating loads with floating operates makes the C400's superscalar capabilities far more effective than they might at first appear. For example, in a hand-coded, unrolled version of the 'Daxpy' inner loop from Linpack, it is possible to calculate six array elements, based on twelve input values (a total of 18 memory accesses), in just 18 clocks; essentially all operate and loop control instructions are 'hidden' by data loads and stores. It is hard to surpass this level of performance in microprocessor designs with a single path between the CPU and cache.

CLIPPER's instruction set architecture includes a variety of complex addressing modes that greatly complicates the task of effective address generation for loads and stores. The C100/C300 devices used a common ALU for address generation and arithmetic operations, and this slowed performance when the programmer attempted to use these addressing modes. The C400 design uses a dedicated address adder in the load and store pipelines to eliminate this bottleneck.

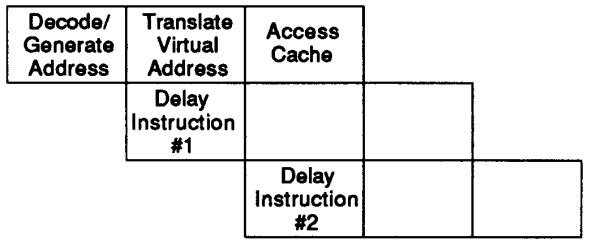
The **Load** pipeline normally requires a clock cycle to generate an effective address, another to translate this address, and a third cycle to access the cache and present the cached data to the IU or FPU. Once the data arrives at the IU or FPU, it can be written to the register file or bypassed to the appropriate functional unit. The overall flow appears as follows:

Decode	Generate Address	Translate Virtual Address	Access Cache	Write Register File

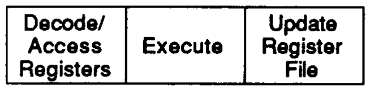
The **Store** pipeline hides the virtual address translation and cache cycles from the rest of the IU:

Decode	Generate Address	Output Data

The new **Branch Pipeline** represents a major enhancement to CLIPPER architecture, and is also one of the few changes visible to application software. The original CLIPPER implementations omitted a delayed branch instruction, primarily because of the difficulty involved in parsing variable-length instructions in the delay slot. The semantics of the old branch and conditional branch instructions were preserved for the sake of software compatibility. A new 'Compare and Branch' instruction tests the value of a general register and branches appropriately, with two delay slots. This approach avoids the use of the condition codes and gives the compiler more flexible choices regarding code scheduling. This instruction also includes a static branch prediction bit supplied by the compiler. When the 'Compare and Branch' instruction is executed, the branch prediction is used to determine whether the address of the branch target should be output to the MMU/cache. If the prediction is correct, then the first instruction at the branch target will arrive in the instruction buffer two clocks later, immediately following the execution of the delay instructions. Bad branch predictions impose a one-clock penalty, since the CPU incorrectly signals the branch target address, and cannot output an updated target address to the MMU until the next cycle. The overall pipeline appears as follows:



The integer unit contains an ALU, a 32-bit barrel shifter, a Wallace-tree multiplier and an integer divider. Most arithmetic and logical operations execute in one clock cycle, and thus do not present any special problems regarding resource management or instruction scheduling. Multiplication and division operations, however, require a variable number of cycles to complete, based on the values of the operands. Because these operations are not pipelined, the instruction issue logic cannot issue a second instruction of the same type until the first one completes. To minimize lost cycles, the issue logic and the multiply/divide function unit communicate via a simple protocol that lets the function unit examine the arguments and inform the issue logic how long the operation will take. The basic flow for most integer and logical operations follows the classic RISC decode/execute/write-back model:



Floating-Point Unit (FPU) Design

The FPU contains separate execution units for addition/subtraction, multiplication and division. The FPU's superpipelined design limits the number of levels of logic in each pipeline stage, and allows the processor to run at a much higher clock rate than otherwise possible, given typical 1 µ CMOS gate delays. This higher clock rate creates more opportunities to issue instructions, and improves overall computational performance. A multi-ported register file with three read and two write ports holds floating-point data; like the IU's register file, it can be accessed in less than 6 ns.

The Integer Unit decodes all floating-point instructions and handles all the address calculations and memory operations involved in floating load and store operations. Most FPU real estate is devoted to the execution units that handle floating addition, subtraction, multiplication and division. The superpipelined add/subtract and multiply execution units incorporate deep pipelines that permit the IU to issue instructions to the same unit on every clock cycle without pipeline stalls. Double-precision multiplication operations require a one-clock interval between back-to-back instructions; a one clock stall occurs if the compiler places two 64-bit floating multiplies in adjacent issue slots. Floating-point divide operations do not occur with sufficient frequency to justify the expense of a pipelined divide operation, and the issue logic will stall the processor if the compiler schedules a second divide operation prior to the completion of an earlier one.

The instruction issue logic will not issue instructions out of order and will delay the issue of any instruction until all the resources needed for its execution are available. Nevertheless, given the extremely wide range of execution times for floating-point instructions (a 32-bit add takes four clocks and a 64-bit divide takes 30), it is entirely likely that instructions will complete out of order. This creates an interesting problem for compiler writers who must track the sequence in which computations complete if they are to provide optimal code scheduling.[4] It creates an even more challenging problem for the authors of math libraries that need to deal with the variety of IEEE 754 floating-point exception traps that can occur during normal program execution.

Imagine what might happen if a program starts a 64-bit division (a 30 clock operation), proceeds with several additions and multiplications, and then discovers a floating underflow condition on the division. The trap handler will want to sort this out, match the error to the offending instruction, and present the results to the application programmer in a manner consistent with in-order instruction issue and in-order instruction completion. Some IEEE implementations achieve this by disabling out-of-order completion; i.e. they serialize instruction issue and give up any opportunities to exploit parallelism within the code. Other implementations examine the operands at the start of each operation, and signal a trap if there is a possibility a trap might occur; this approach tends to increase overhead as the software processes these potential problems. The C400 FPU achieves IEEE compatibility without sacrificing performance or accuracy. To facilitate the reconstruction of an in-order completion sequence when traps do occur, at the start of each floating operation, the FPU stores the program counter and source operands in a queue known as the Floating Trap Register. As operations complete, their entries are removed from the queue. When the hardware does encounter an IEEE exception, the contents of the FTR are frozen, and an operating system routine is invoked to untangle the situation.

Circuit Design

The C400 VLSI components were designed using a standard cell methodology, with a custom-designed register file and I/O pad ring, using a 1.0 μ CMOS process with two layers of metal. The short design cycle mandated the use of a standard cell approach. The high bandwidth requirements needed to support concurrent operations the IU and FPU necessitated the use of a custom register file. Our APD operation depends on external fabs for wafer production, and we believed the choice of a 1.0 μ process would be a conservative one for the period in 1990-1991 when we planned to go into production. We have been surprised to find that we are closer to the state of the art in this regard than we expected. After reviewing the capabilities of most major semiconductor foundries, we conclude that there is a substantial gap between vendor claims and the reality of ASIC manufacturing today.

Our desire to spread our design over several moderately sized chips, rather than pack it all on one die, led to the need for innovation with regard to interconnect circuitry. We needed to minimize interconnect delays and power dissipation. Much of the power dissipated by CMOS devices goes into driving the I/O pads:

$$P = CV^2f$$

where P is the dissipated power, C is the capacitance of the circuit, V is the voltage swing, and f is the driving frequency. With many I/O pins on each device, and anticipated frequencies in excess of 50 MHz, we were quite concerned about power dissipation. We reviewed the use of a BiCMOS process to obtain a low voltage I/O along with a normal 5 V on-chip operation, but no BiCMOS vendor could support our requirements. Instead, we turned to a unique circuit design that allows us to use low voltage swings (approximately 1 V) for most high frequency I/O lines. We achieve this by feeding a low voltage signal onto the chip, and then using this signal as a reference voltage for input signals, and as a switched output for outbound signals. The use of this one volt signalling method reduces the power used to drive the I/O lines by a factor of 25, and keeps total power requirements under 7 watts.

Conclusions

The C400 development program accomplished a complete redesign of the CLIPPER processor in slightly less than two years, approximately one-half the elapsed time required to implement the original C100 The speed with which this task was performed demonstrates many positive aspects of our industry in general and CLIPPER architecture:

- VLSI development tools have improved vastly over the last five years, especially in the areas of chip layout and simulation.
- The design team was able to incorporate concepts of superscalar dispatch, superpipelining and delayed branching while maintaining software compatibility with existing applications. This demonstrates the robustness of the architecture, and validates many decisions made in CLIPPER's original definition, such as hiding the precise operation of the execution pipeline from applications code.
- The performance gains from our somewhat constrained approach to superscalar dispatch were truly gratifying.

This new CLIPPER delivers a level of performance unimaginable just a few years ago, but future CLIPPERs will likely dwarf the C400's performance, just as it today dwarfs the performance of the original C100.

1 Introduction to the CLIPPER Architecture, Fairchild Camera and Instrument Corp., October 1985

2 W. Hollingsworth, H. Sachs and A. J. Smith, "The CLIPPER Processor: Instruction Set Architecture and Implementation," *CACM* February 1989

3 H. McGhan and H. Sachs, "Future Directions in CLIPPER Processors," COMPCON'91

4 W. Baxter and R. Arnold, "Code Restructuring for Enhanced Performance on a Pipelined Processor," COMPCON'91