The Return of the Cray

Author : Andras Tantos

Introduction

Previously I’ve told you about my failed efforts of building a Cray-1. I’ve also told you about how I’ve managed to get my hands on some Cray X-MP software, how I had to recover the data from a raw ‘magnetic photograph’ of the disk, and once I had a reasonably good reconstruction, that I’ve found a mostly complete boot image on the disk.

I originally wanted to dedicate this article to my attempt at writing an X-MP simulator. But in order to do that, I had to introduce the machine I was trying to simulate. That section grew and grew, so I’ve decided to split to topic into two articles: this one about the Cray X-MP series only, and the next one dedicated to the simulator.

After the introduction of the Cray-1 in ‘75, CRI (Cray Research) was busy in two separate development paths. One concentrated on a completely new machine, which culminated in the Cray-2 in ‘81. The other path was to improve the existing Cray-1 design, which lead to the Cray-1S in ‘79 and the X-MP line in ‘82.

The X-MP machines originally came with two CPUs. Later, in ‘84 Cray added a single- and a four-processor configuration as well. Main memory ranged from 1MWord (8MByte) to 16MWord (128MByte). The Model number reflected the number of CPUs in its first digit and the memory size in the second (two for the 16MWord version). So, an X-MP model 22 would have been a dual-CPU machine with 2MWords of memory. My target, a model 14, was a single-processor configuration with 4MWords of memory. Machines come in a wide variety of I/O configurations, peripherals and storage capabilities, so the model number didn’t give you the whole picture. Physically, the X-MP machines looked remarkably similar to the Cray-1s:

The fact that they packed four times as much HW into the same physical size meant that they had to use significantly higher integration. It also meant that the distance between key components got shorter due to the denser setup. These two factors resulted in a modestly higher clock rate of 105MHz, up from the original 80 of the Cray-1s. It's not clear to me from the documentation, but I would expect slightly higher latencies for far-away parts of the system, like the memory.

This marketing document gives a good summary of the series:
http://archive.computerhistory.org/resources/text/Cray/Cray.X-MP.1985.102646183.pdf

With the X-MP tracing back its lineage to the Cray-1 most changes are evolutionary, at least on the mainframe side of things. The biggest change is the redesigned I/O subsystem, so let's take a look at that first!

The I/O subsystem

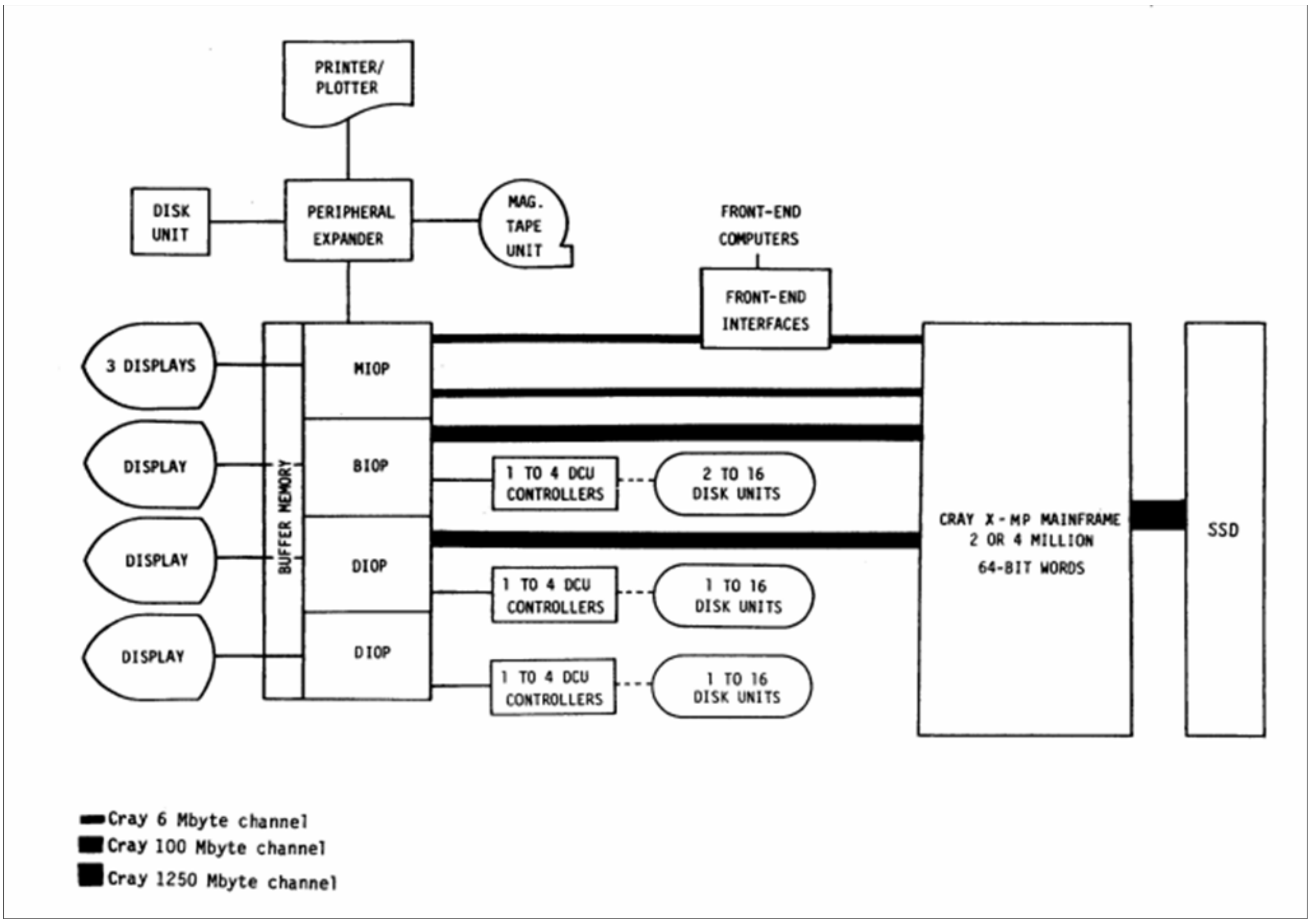
In the original Cray 1 all peripherals attached (more or less) directly to the mainframe. In the new design of the X-MP (in fact already available in the Cray-1S) they introduced a set of I/O processors, or IOPs. These IOPs had their own memory, a different processor (with a different instruction set), and interfaces to actually connect to peripherals, like disk controllers or displays. They are complete computers on their own right. They connected the mainframe though a set of I/O channels – which are more or less the original I/O interfaces from the Cray 1.

On top of that, some of the IOPs also had direct access to the mainframes' memory through separate, high-speed DMA channels.

The mainframe-to-IOP channels were capable of roughly 6MBps of transfer rate. Each side set up its own send and receive buffer and transmitted data from/to those using the channel HW. The actual transmission was done through DMA, but active participation was needed from both sides for a transfer to happen.

In contrast, the DMA channels gave the IOPs direct access to the memory of the mainframe. They could freely transfer any data between the mainframe memory and the IOPs local memory without the mainframe having any say in the transfer. In fact they could do that even if the mainframe is not running at all – which came handy during boot.

Here's how a typical X-MP setup looked like:



There are four IOPs in a fully configured X-MP system: the 'MIOP', 'BIOP' and the two 'DIOP's shown above. They have the same HW, but are hooked up to different peripherals and to different channels on the mainframe. They also execute different SW.

You can also see something called ‘SSD’ on the diagram, which is what you think it is: it’s a solid state drive (yes, from 1982!). It came in several sizes, up to 1GB. It was an expensive, optional component of the system.

You can also see something called a ‘buffer memory’. This was a large chunk (up to 64MB) of memory shared between all IOPs. It is not directly addressable though, there is a DMA controller that can be set up to transfer data to or from local (called I/O) memory to this buffer memory.

The rest (displays, disk units, front end interface, etc.) are the peripherals of the system.

For the curious, there’s actually a little cheat in this diagram: the front-end interface connected to either the MIOP or the mainframe, but not to both. It seems the former configuration was for the new systems, and the latter for legacy systems where compatibility with existing code was important.

The IOPs

With that, let’s take a closer look at the heart of the I/O subsystem: IOP CPUs!

ISA

As far as instruction set and capabilities go, the IOPs are quite simple 16-bit processors. If the mainframe CPUs are more akin to modern RISC processors, the IOPs are distant relatives of the 6502. As far as I can tell, these CPUs were not pipelined, so every instructions took several clock cycles to complete. Almost all instructions are 16-bit long, though there are a few 32-bit ones where a large immediates are needed. They also had a very limited set of registers:

- The 16-bit **A** (accumulator) register used as the target of every computation.
- The 9-bit long **B** register can be used in certain addressing modes.
- Finally, there’s the **C** or carry flag.

Now, just as in the 6502, such a limited number of registers (the 6502 had A, X, Y and some flags) means a lot of spills – loads and stores of registers – into main memory. To help with that problem, the IOPs introduce a set of 512 16-bit registers, called **operand registers**. These registers are fast to get data in/out of, and since there’s only 512 of them, they can be addressed with part of the instruction code (the 6502 equivalent is page-0 addressing). Some operations can be combined so that the source or the target of a computation can be an operand register, saving the extra loads and stores. Still, these operations use the accumulator as their intermediate storage, so even in those cases, the accumulator value is going to change.

The instruction set is also rather simple in terms of addressing modes: it supports direct, indirect (through an operand register) and indexed indirect (operand register + immediate) addressing.

The only noteworthy execution units are the ALU, which supports add, subtract and logical operations and a barrel-shifter. No multiply let alone divide. And of course no floating-point or vector operations either. Simple.

While I kept saying that the IOPs are similar to the 6502s, there's at least one big difference: speed. The IOPs were clocked at 80MHz, while the 6502 was running at a whopping 1MHz at the time. This difference – coupled with the very different price-point – resulted in some differences in key design decisions.

For example, the low speed of the 6502 allowed page-0 addressing the be efficient, while with the IOPs the additional speed-degradation of keep going out to main memory for the operand registers would have been prohibitively slow.

On the other side of the fence the 6502s price-point didn't allow integration of such large memories on die: 256 bytes of 0-page memory would have required at least 512 transistors (in SRAM technology), while the whole CPU consisted of ~3500 transistors. That would have been a 14% size increase, definitely a no-go.

Memory

Please note: the nomenclature used in the documentation describing the organization of the memory is confusing so I'm going to re-word things a bit in the hope to make it easier to understand. Keep in mind though that if you read the original documentation you'll find different names for things.

I/O Memory

Each IOP had 128kByte local (called I/O) memory with a cycle time of 4 clocks for reads and 4-6 clocks for writes.

There were a total of 6 ports to this memory:

- One port for memory read/write accesses
- One port for instruction fetches
- Four ports for four DMA channels

Even though the memory was viewed externally as having 16-bit words, internally it had a 64-bit wide organization. When a read or write operation was made, the appropriate 16-bit word was selected based on the lowest two address bits. At the same time the whole 64-bits was held in a register for quick access to the other three 16-bit sections, should they be needed. This setup allowed for the instruction fetch port to read one instruction (16-bits) every clock cycle, even though the cycle time of the memory was four cycles.

With this setup the instruction fetch port could potentially exhaust all available memory bandwidth. To overcome that, the memory was further organized into four banks which allowed for more bandwidth, provided the parallel requests hit different banks.

Buffer memory

The Buffer memory was a large piece of shared memory between all four IOPs. As I've said before, it's not directly part of the address space of any of the IOPs though: a special peripheral was used to DMA data between the I/O memory and the Buffer memory.

As far as size goes, it was large: 8, 32 or 64 MByte and it was based on DRAM technology, so needed constant refreshing (2ms refresh rate). The memory was organized into 64-bit wide words. As you would expect for DRAM, it was much slower than the I/O memory with a cycle time of 32 clock cycles. The access time was 16 clock cycles – what this means is that you could issue an access every 16th cycle, but you had to wait 32 cycles to get the result. The memory had some pipelining to it.

Just as the I/O memory, the Buffer memory is divided into banks (8 for the smaller two, 16 for the largest configuration). It had four ports, one for each IOP.

Putting the numbers together means that the peak bandwidth available on each port is 40MBps, with the maximum aggregate bandwidth of 160MBps.

At the same time an 8-bank configuration could handle up to 160MBps bandwidth, provided no bank-conflicts. A 16-bank configuration could handle twice of that.

You can see how the numbers for the 8-bank configuration nicely complement each other. You can also see that the extra bandwidth of the largest memory configuration seems to remains unused. Having 16 banks however reduces the chances for a bank-conflict (by having more of them), so the average bandwidth would still be higher.

In some way, you can view the operand registers, the I/O and Buffer memories as an early attempt at a memory hierarchy: operand registers are the fastest but smallest storage. I/O memory is larger, but significantly slower. Buffer memory is yet again larger but slower. As opposed to modern CPUs however where (almost) all the memory hierarchy is managed transparently by the various cache-controllers, here all of this needs to be manually managed by SW. This of course makes for simpler HW, which you pay for in extra instructions – and complexity – on the SW side.

IO

Each of the IOPs have several (up to 40) I/O channels connecting to peripherals (and just to overload terms, these channels are not the same as the 6MB or 100MB channels we've talked in the beginning). You can think of these channels more or less as the x86 I/O space: they are not part of the address space and special instructions are available to transfer data between the channels and the registers. The addressing is split into a channel address and a function address. More or less the channel address determines the peripheral to be selected, and the function address determines the register within that peripheral to be used. There were up to 16 functions (registers) available for each channel. A weird detail is that the function address not only determines the register, but also the direction of the data transfer. Some function codes can only be used to output the value of the accumulator to the peripheral, some only to get a value from it.

Channels also had a **master-clear** (reset) a **ready** and a **done** signal. The master-clear signal could be used to return the peripheral to it’s default state. Ready signals the channels readiness to accept a new command. Done used to signal that the channel completed a previously started operation. If enabled, the done signal also raised an interrupt.

Each IOP had 6 DMA channels sharing the four DMA ports on it’s I/O memory that could be used by high-speed peripherals to transfer data. The DMA channels are statically allocated to peripherals. The actual DMA access generation was the responsibility of the peripheral and consequently was set up potentially differently for each peripheral.

While there is no restriction like that, many I/O channels implement unidirectional transfer between a peripheral and the IOP. If the peripheral needs bidirectional transfers, two I/O channels are assigned to it. This probably has to do with the interrupt generation capability: an I/O channel can only raise a single interrupt. So if simultaneous input and output operations are required, the easiest way of generating the completion interrupts is to use two I/O channels. As a result, consoles (concurrent input and output) have two I/O channels each, while disk drives (no concurrent reads and writes) have only a single one assigned to them.

Instruction sequencing

Call-stack and interrupt handling is rather weird so let’s spend a few minutes on it! When a function call happens, the current instruction pointer is pushed into a HW stack, called the ‘program exit stack’.

This stack is 16 element long. There is an internal ‘stack pointer’ register (called ‘E’), that points to the first empty slot in the stack. Calls push a new value to the stack while returns pop the last entry. There’s only a single interrupt vector, and it’s address is stored at the first location (index 0) of the program exit stack. So far so good, the only unusual part is that the stack is not part of the main memory, it’s a dedicated set of registers – and consequently very limited in size.

The interesting part is the way you can access this stack (other than calls and returns). It’s through one of the I/O channels: the program exit stack is a peripheral as well as an integral part of the CPU. It is through these I/O channel operations you can read or write arbitrary values on the stack, examine or modify the value of the ‘E’ register.

There are a few reasons why these extra operations are important. First of all, you have to be able to set the address of the interrupt vector, which is stored at index 0. Second, since the stack is very limited you’ll have to have means to swap parts of it out into main memory to free up space for deeper function calls. To facilitate that, the HW can raise an interrupt when the hardware stack gets (close to) empty or full and of course the interrupt handler can deal with the swapping of data.

Peripherals

So what’s on the other end of the I/O channels? Peripherals, off course!

Each IOP can have a number of peripherals attached to them. Some are common to all, some are only attached to certain IOPs.

The various peripherals are referred to by a three-character mnemonic. In many cases the 3rd letter simply implies which of the many identical ones are we talking about: ‘A’ for the first one, ‘B’ for the second, etc. Here’s the list:

Channel mnemonic	**Description**
IOR	Interrupt request
PFR	Program fetch request
PXS	Program exit stack
LME	Local memory error
RTC	Real-time clock
MOS	Buffer memory interface
AIA, AIB, AIC	IOP to IOP communication channels, input side
AOA, AOB, AOC	IOP to IOP communication channels, output side
ERA	Error log
EXB	Peripheral expander
CIA	6MB channel to mainframe, input side
COA	6MB channel to mainframe, output side
CIB, CIC, CID	6MB channel to front-end interfaces, input side

COB, COC, COD	6MB channel to front-end interfaces, output side
TIA, TIB, TIC, TID	Console (serial terminal) interface, input side
TOA, TOB, TOC, TOD	Console (serial terminal) interface, output side
HIA	100MB channel to mainframe or SSD, input side
HOA	100MB channel to mainframe or SSD, output side
DKA, DKB…	Disk storage unit (hard drive)
BMA, BMB…	Block multiplexer channel (to tape drivers, punch-card readers, etc.)

These interfaces are attached to the IOP channels in different configurations for the various IOPs. On top of that, some of these interfaces have further connections externally:

The Peripheral Expander (EXB) can connect to hard drivers, printers, tape drivers and real-time clocks. As I later figured this out, the peripheral expander appears to be designed to connect to Data General Eclipse and Nova peripherals. This of course is never admitted anywhere in the documentation, but by comparing the programming model, it quickly becomes quite obvious.

The Block multiplexer channels in contrast are connecting to IBM equipment, like tape drivers or card readers.

The front-end interfaces (as far as I could gather) are what we would call these days network cards: they provide remote access to the mainframe through some sort of a computer network.

In many cases the actual peripheral and it's programming is not very well documented, and can vary within the same device class. For example, the programming of a disk drive depends on the actual disk drive and it's capacity. Just because it's called 'DKA', you don't know how to talk to it. This is especially true for equipment connected to the peripheral expander or the block multiplexer: those devices have zero documentation.

The mainframe

The mainframe portion of the X-MP is similar to a Cray-1, but – mostly due to the multi-CPU setup – it had a some important changes. Most of what I've written about in the introductory article, still applies, so I'll concentrate on the differences.

Shared registers

To support multi-processor development, a set of shared registers are added to help with interprocessor synchronization:

There’s a set of 24-bit shared address (**SB**), and 64-bit shared scalar (**ST**) registers as well as a set of 1-bit semaphore (**SM**) registers. These registers are grouped into so-called **clusters**. Each cluster contain 8 SB, 8 ST and 32 SM registers. The number of available clusters depend on the number of CPUs. For four-CPU configurations, there were 5 clusters, for dual-CPU setups, there were 3. Each CPU had an associated cluster stored in the **CLN** register. The value of this register could be changed by a special (monitor-mode only) instruction, or during a exchange sequence. CPUs with the same assigned cluster number would share one set of SB, ST and SM registers. The new **SMjk 1,TS** (test-and-set) instruction supported atomic operations on the SM registers to facilitate inter-CPU synchronization. Talking about synchronization, a new interrupt was added for one CPU to interrupt another.

Memory

More CPUs mean more memory requests, so the memory subsystem got greatly beefed up with higher peak bandwidth and multiple ports. The CPUs also received two additional memory ports. In a four-core system, the connection of memory and CPU ports was the following:

I/O and DMA channels (these are the mainframe I/O channels, connecting to the IOPs) shared the ports of the CPUs. CPU path selection units steered traffic from the various CPU ports to one of the memory ports. The memory itself got cut up into four separate sections, each independently having 16 banks. This coupled with the 64-bit organization results in a peak **13GBps** (!) memory bandwidth even with the relatively slow 38ns access-time SRAM chips used. This bandwidth is shared by the four CPUs, and even all that isn't always enough: if all four CPUs constantly do instruction fetches, they could generate twice that bandwidth. That is of course only a theoretical number and in fact the CPUs are prevented in HW from doing just that.

The various operations on the CPUs have a variety of access times ranging from 14 to 17 clock cycles. The data transfer rate per clock cycle is also highly variable from 32-bits to 512 bits.

A dual-CPU configuration has half of most of these things:

The rest

There is a new real-time clock addition to help with time-sharing. The X-MPs could run UNICOS, a variant of Unix on top of COS, the old batch operating system.

To help system-level performance evaluations, the system included a set of performance counters.

While the internal organization of the CPUs remained largely the same, they received a second vector logical execution unit. This unit could be disabled in SW for backwards compatibility.

Sources of the images

By Jens Ayton (Ahruman) (Own work) [CC-BY-SA-2.5 (http://creativecommons.org/licenses/by-sa/2.5)], via Wikimedia Commons

http://bitsavers.informatik.uni-stuttgart.de/pdf/cray/HR-0032_CRAY_X-MP_Series_Model_22_24_Mainframe_Ref_Man_Jul84.pdf

http://bitsavers.informatik.uni-stuttgart.de/pdf/cray/HR-0097_CRAY_X-MP_Series_Model_48_Mainframe_Ref_Man_Aug84.pdf

http://bitsavers.informatik.uni-stuttgart.de/pdf/cray/HR-0808_CRAY-1S_HWref_Nov81.pdf