Prelude

Author : Andras Tantos

I always wanted to build my computer. Not as in ‘going into the store and buy my a motherboard and a CPU’, but as in ‘connecting all the transistors together that make up a computer’. I’ve even ‘designed’ my own computer from 74xx gates back in high-school, but never got around building it. Later, when FPGAs become more available that dream seemed more reachable then ever. However as I grew older, the magnitude of the task also dawned on me. It's one thing to build my own machine, but who’s going to program it? Who’s going to write an assembler, a compiler, libraries, an OS, applications, etc. for it? It’s really not a garage industry any more.

So, I went on looking for ways to re-use what other people have already done. When I looked around on OpenCores and other places, it became clear that most of the interesting CPU architecture have already been re-implemented in FPGA. The ones that are not (most notably ARM) are protected by a fierce pack of lawyers. Re-doing something that others have already done wasn’t terribly appealing, so I kept searching. Then, the idea hit me: what if I re-implemented an old machine, that doesn’t exist any more? Which of course immediately begs the question: which one? A Commodore-64? Has been done many times over. Well, how about a famous machine from the history of computation? No, not that one. This one. A towering monster from the past that is probably one of the most iconic computers of all times. Yes, a **Cray**.

As it turns out there’s a lot of documentation on the old Cray machines out on the net, most notably the Bitsavers archive of scanned manuals. After a few days of researching the history of these machines it became clear that the complexity of these machines sky-rocketed very quickly. The first ones – the Cray-1 series – are relatively modest by todays standards (80MHz clock rate, few 1, 2 or 4 MBytes of memory, about 200,000 gates). The X-MP series is borderline too complex to bite and after that, it is just crazy. (We’ll see later that even the Cray-1 has features that are very hard to accurately replicate with todays HW on a modest budget.)

So it’s settled. I’m building a Cray-1. Now let’s do some reading!

Cray-1 in review

The first Cray-1 machine shipped (well, lent) in 1976 to the Los Alamos National Laboratory. Here’s the Man himself doing an introduction of this machine. This line of machine was a great success, and sold if I’m not mistaken about 80 units. Yes, you read it right. Not 80M, but 80, and yes, that was considered a great success in those days. Of course when each machine costs several million dollars, that’s still a nice cash-flow. In 1981 Cray introduced the new Cray-2s, which weren’t nearly as successful, so it was quickly followed by the Cray X-MP line which was a cleaned-up Cray-1 with extended memories and multi-processing support. In 1988 the new, Y-MP series was announced. Not much after that Seymour Cray left the company to fund Cray Computer Company. Cray Research Inc, or CRI continued to develop machines though, shipping the C90 in 1991 followed by the T90 in 1994 and the J90 in the same year. Finally in 1998 the SV1 was introduced.

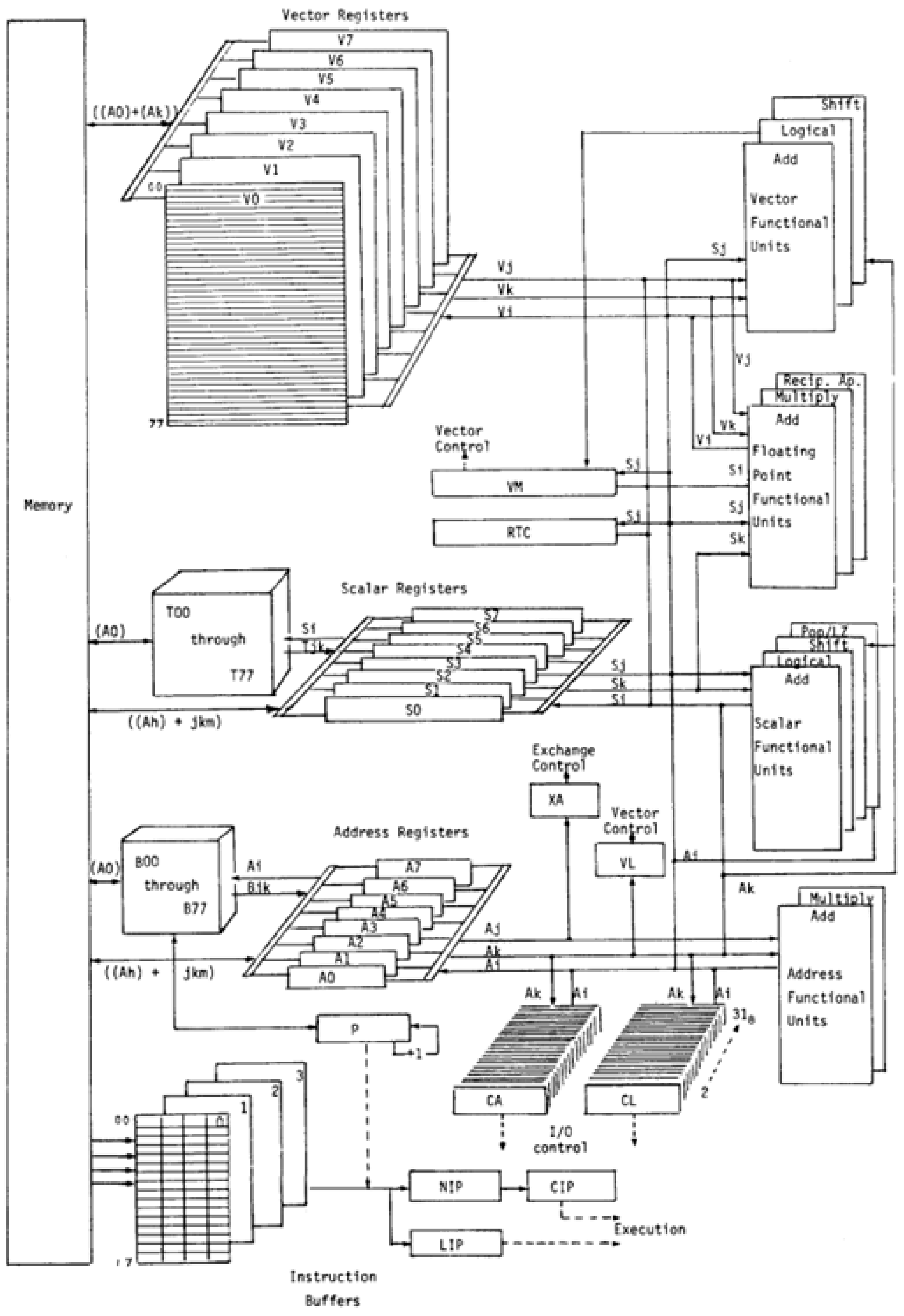
Now, the various companies sporting the Cray brand have released other machines along the years as well – the Cray Computer FAQ pages give you a great list – I've included these here for a reason. All of these machines are parallel vector processors and based on the initial ISA of the Cray-1. Of course things have changed over the years (larger memory, parallel processing, wider addresses, IEEE floats, etc.) but just as the current Core I7-s carry the DNA of the venerable 8086, so did all these machines track their heritage back to the Cray-1. This goes to show just how important instruction-set design is: even in the world of supercomputers, ISAs are carried over for decades, so you better make sure you don't make any mistakes. And more importantly, you anticipate architectural changes for 20 years in an industry that re-invents itself in every 24 months!

In all CPU designs there are two pieces: the programming model, that is the things that the programmer sees or needs to know about, and the hardware architecture, that is the way the processor designers implemented that particular programming model. As an ideal, you want the programming model being influenced by the hardware architecture as little as possible. The main reason is that technology changes, implementation options change, trade-offs shift, but code lives forever. You want to keep backwards compatibility as much as you can, while having freedom in choosing the best implementation as much as possible. This is the way you can crack that very hard nut of maintaining a product line for decades.

The contract with the programmers

So how did the Cray-1 live up to this ideal? It's not stellar, not terrible neither, but let's go though it one by one!

On the surface, the Cray is an honest to God 64-bit vector architecture, with a lot of the usual (by todays standards) suspects and some surprising ones. Here's the computation section diagram from the original Cray-1:



What you can immediately see is that this is not the ‘programming model’. This is not the abstract view that the programmer needs to care about. This is the actual guts of the processor. Red flag No. 1!

But if you step through that, here's what the programmer is actually seeing of all this:

Base Register Set

The main register set consists of:

- Eight 24-bit long '**A**' registers for indexing into the memory.
- Eight 64-bit '**S**' or scalar registers to support scalar math operations.
- Eight 64-entry 64-bit-wide '**V**' or vector registers for the vector operations.

Today's vector units usually support fewer number of registers, and each vector register is 128 or 256 bits wide. These vector registers have 4096-bit each! Now, a lot of operations don't need such long vectors, so the actual vector size is determined by the '**VL**' or vector-length register. In some cases you only want to execute an operation on certain elements of a vector, or store the result of a comparison of each vector element in a compact form. This is what the vector-mask or '**VM**' register is for.

So far so good, nothing terribly surprising. In review, you would probably take issue with the differentiation of the 'A' and 'S' registers, and you could make the case that 24-bits for addresses even back then was a bit short-sighted, but that's about it. You can't see it from this diagram, but the instruction-set is your average load-store, RISC-like ISA, with very few, simple addressing modes.

Addressing

Being a 64-bit machine, memory addresses are counted in 64-bit increments. There are no byte, word or dword address, if you need a character, first you have to load the full 64-bit word, than using bit-manipulation, extract the proper 8-bit portion. Functional, but makes operation on non 64-bit entities quite slow.

There is one exception though: instructions are either 16- or 32-bit long. This means that the instruction pointer – '**P**' – register needs to be able to address 16-bit quantities. To top it off, P is also a 24-bit register. So even though in theory a 24-bit A register could access 128MBytes of memory, the P register can only address a quarter, or 32MBytes of it. To make things right, the Cray-1 actually truncates addresses in 'A' registers to 22-bits, making them also only able to address 32MBytes of memory. Worse, addresses now come in two flavors: if you use an address for data load or store, it's measured in 64-bit quantities, but if it happens to be an instruction fetch, it's only 16-bits per increment. Why do I care, you ask? Well, instruction and data-addresses get mixed quite often: think of a indirect call (computed GOTO in Fortran), where you have to transfer a value from an 'A' register to the 'P' register. Or when you store a function pointer in memory right next to a buffer address. Yes, it works of course, but it's very confusing and hard to follow.

Interrupts

Interrupts are also handled in an interesting way: whenever an interrupt happens something, called the 'exchange sequence' occurs. There's a special register, called '**XA**', that contains the address of an 'exchange packet'. This is a third type of address by the way, measured in 64 byte quantities and is only 8-bits long; but at least it's aligned such that you load and store normal 64-bit addresses to it, it's just that the lower 4 bits are always 0. The exchange packet contains most of the internal state of the CPU that is stored into memory upon interrupt, and restored from the same location. In other words the 'operational' context in the CPU and the 'inactive' context in the exchange packet pointed to by XA gets swapped on an interrupt.

Which means, you have essentially two contexts. One for normal processing and One for interrupt handling. You better not get another interrupt in the interrupt process, because you would switch right back into your normal processing context. You can of course change the content of the XA register, that way supporting multiple contexts, but you can only do that from 'monitor mode', they Cray-1 equivalent of 'kernel mode'. This means that one of the two contexts – the one that you handle the interrupts in – must be the 'monitor', or the 'kernel', and – before returning from an interrupt – it can decide what context to return to by changing the content of the XA register. So essentially you're kernel mode and your interrupt handler are one and the same.

If it weren't weird enough, the exchange packet contains most but not all context. For starters it doesn't contain any of the vector registers. Now, your kernel probably doesn't do too many vector operations, your user-mode tasks do. This means that every time your kernel does a task-switch (changing the XA register) it has to manually save and restore all V registers. You also have to save all T and B registers, but we'll talk about those later. Overall, that's a huge burden, and this is another point where the architecture shows its age: there's a reason why modern architectures don't have so many large registers. In the defense of the Cray-1 however, it was designed in the age of batch-processing where task-switches were few and far in-between.

Memory Access

And there's another reason why the Cray-1 has so many registers: it doesn't have any (data) caches. This means that – even with the relatively fast memory this system had – going out for data all the way to main memory was slow. To remedy that, the processor includes 64 '**T**' and '**B**' registers. Each T register is 64-bit long and each B register is 24-bit long. There are quick load and store instructions to get the content of any A or S register into a B or T register respectively. There are also block-load and -store instructions for the T and B registers. In essence the 'T' and 'B' registers are – or can be used as – SW-managed data-caches.
There's one special-case (why wouldn't there be one?) though: when you do a function-call, the – now – well-known link-register feature is used: instead of putting the return address on the stack, the CPU simply stores it in a register. Upon return, you just reload that register into P, and off you go. This register could have been any A register, just as on modern RISC CPUs, but the designers of the Cray-1 saw it best to put it in B00. That's right: into the first entry of our neat little SW-managed cache. You can also 'return' using any of the B registers as the return address. There's one nice feature about this mess though: you can save your return address to another B register if you have your subroutine calling further subroutines and returning using

directly the saved value, no need to restore it to B00.

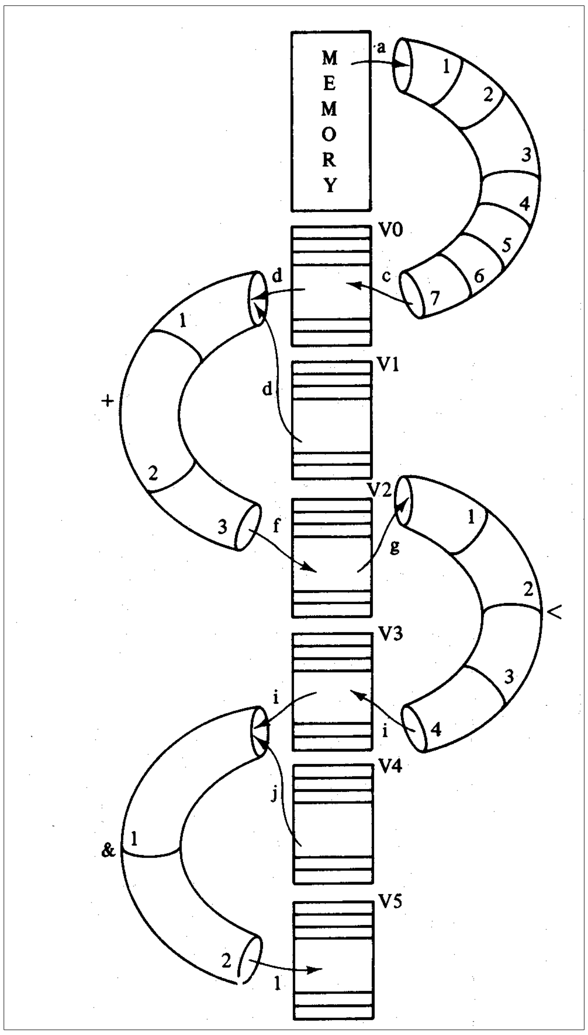
It’s not only T and B registers that can be moved in and out-of memory in blocks. V registers can do the same. To make loading vectors out of matrices efficient, you can also specify the stride of these loads and stores. So for example the first element of a vector can come from memory address 0x001000, the next one from 0x001100, the next one from 0x001200 and so on. Nice on the surface, evil in the details. The problem lies in access control: every memory access is checked against a memory base (**BA**) and limit (**LA**) registers. If the access is outside of the allowed bounds, an interrupt is raised. This is a primitive version of an MMU, that modern processors use to implement virtual memory. In a modern OS, when such an interrupt (exception) happens, the OS swaps in the memory from disk and retries the failing instruction, that then completes. Now, the problem is that the interrupt in this case is generated half way though the processing of the load or – even more evil – the store. Half of the side-effects of the instructions have already taken place, while the rest have not. This is called an ‘imprecise exception’ and is largely frowned upon today. The problem is that it’s next to impossible to build a CPU that supports these instructions, yet features precise exceptions. One might argue of course that the Cray-1 never intended to be used with virtual memory, but it still makes implementing a modern OS on this architecture very challenging. And for the record: this feature is also used for memory-mapped files, not just for virtual memory – which is very much a requirement in any modern system.

Chaining

Even though the Cray-1 is a vector processor, it doesn’t actually process all operations on a vector in parallel, like modern SIMD engines do. It’s more of a HW implementation of a ‘for’ loop.That’s why you see those vector load and store paths from the V register path to the scalar floating point execution units. The way these operations worked was that if you multiplied a vector with another one, the CPU would take the first two elements of each vector, deliver it to the scalar multiplier, when it’s done with the results, deliver it back to the target V register, and go on to the next element. This is repeated the amount of times, described in the VL register. (In reality the multiplier is pipelined, so new input values are delivered in every clock-cycle, even though it took several clock-cycles to produce the result.) This means that a large vector multiply can take a very long time, more than 64 clock-cycles. Of course the rest of the CPU is not halted during this time, it keeps executing instructions, if it can. So what happens if – during this time – another vector-instruction is issued?

The straightforward approach would be to block execution of the instruction until the resources it needs are free – after all that’s what’s happening in the scalar operations – but the designers decided to take a short-cut and call it a feature.

The technique essentially involves catching the results of one vector operation in flight back to the target V register and start a new operation on it immediately.


MEMORY
a
1
2
3
4
5
6
7
c
V0
d
1
+
2
3
d
V1
V2
f
g
1
2
<
3
4
V3
i
i
V4
j
&
1
2
1
V5

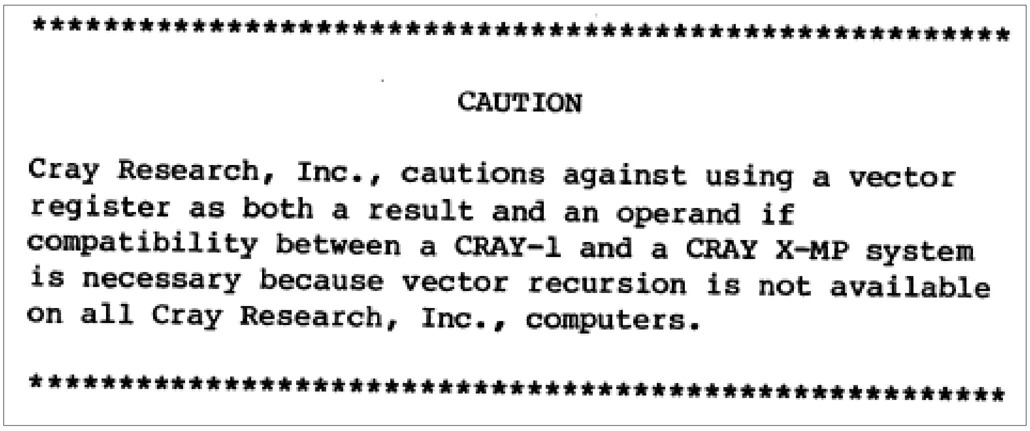
It’s called chaining and it’s described quite well here. The problem here is this: in order to be able to catch element ‘c’ coming out of the pipeline for the first vector operation – the vector load in the example above – you have to know the **exact** latency of the load operation. There’s a point in the instruction stream, called the chain-slot. If you don’t put your vector instruction there, you missed your chance for chaining. If you put it before that, the vector instruction (and the whole CPU as a result) gets stalled until the chain-slot. If you put it afterwards, the instruction gets stalled until the previous vector operation completes. It is not that different form pipeline forwarding, used in modern CPUs, but in this case used for partial vector results.

So far so good, this is a nice feature on paper, in fact it allowed Cray to market the Cray-1 with 160MIPS while it only had an 80MHz clock, and at most one instruction issue per clock-cycle.

The problem arises when you set the target of the vector operation to be the same as one of its source. With chaining you can set up loops now! In order to use these loops properly, the programmer has to know the following properties of the implementation:

- That vector operations don’t happen in parallel on each element, but in a loop
- The exact cycle-count of each operation
- The exact scheduling rules for when subsequent vector instructions can execute
- The number of vector execution units and their allocation rules

This is bad. In fact, you can see how bad it was from this note in the Cray X-MP manual:



```
*********************************************************

                        CAUTION

Cray Research, Inc., cautions against using a vector
register as both a result and an operand if
compatibility between a CRAY-1 and a CRAY X-MP system
is necessary because vector recursion is not available
on all Cray Research, Inc., computers.

*********************************************************
```

In conclusion

The programming model of the Cray-1 has some problems, but in general it’s not horrible. Apart from one side-effect of chaining – which might even have been unintentional – the separation between programing model and architecture is rather strong. The main problems are:

- Chaining with source and target being the same exposes architecture. Should be avoided.
- Large register sets were defined as a result of having no caches. This impacts context-switching performance in modern environments.
- Vector-loads with non-unit strides raise imprecise exceptions that are hard to retry. Makes memory mapped files and virtual memory implementation hard.
- No 8- 16- or 32-bit addressing makes it hard (slow) to handle smaller than 64-bit data-types

The rest are either easy to fix or just look weird by todays standards, but not necessarily bad.

Finally, let's take a look at the more modern incarnations of the programming model:

http://docs.cray.com/books/SR-3108_9.1/html-SR-3108_9.1/fqwqfdcsmg.html

It's striking how similar they are still. Yes, they implement IEEE floats (remind you, when the Cray-1 came out there **was no** IEEE floating-point standard), they vary the length of the vector registers, but the rest is pretty much identical. Just to show how deep the heritage is, here's the instruction encoding page from the same manual. They still use octal numbers!

The Architecture

Now, that we went through what the programmer sees, let's see what's under the hood that we haven't covered so far!

Caches

Did I say the Cray-1 didn't have any caches? Well, that's exactly not true. At least not for code. That's what those 'instruction buffers' at the bottom-left corner of the image are there for. They are more or less a four-way associative instruction cache, each containing a single 128-byte cache-line. These cache-lines can be filled faster – sort of like a burst – then individual instruction-fetches form main memory would take.

Execution units

Looking back at the block-diagram just above, you'll see the usual execution units with a few notable oddities. There is the separate set of 'address' and 'scalar' execution units. This is most likely due to the buses connecting the execution units to the register files. There is no way to execute a scalar and an address arithmetic operation in one instruction.

The other oddity is the missing vector multiply function unit. There are probably size reasons for this: a floating-point multiplier is large. Another reason might be that they didn't expect scalar and vector multiplications to overlap in time so a shared functional unit might have made sense.

There is also the notable omission of a scalar-integer multiplier. The address multiplier can only handle 24-bit entities, and the floating-point multiplier is also only capable of dealing with 24-bit integers (when used in integer mode). This means that even multiplying 32-bit integers take quite a few instructions. For 64-bit integers, it’s even worse.

Peripherals

Data movement in- and out-of the computer is done exclusively through DMA channels. Each channel (12 in total in the Cray-1) has a channel-address (**CA**) and a channel-limit (**CL**) register. These channels are unidirectional, half of them are input, the other half are output channels. The channels raise an interrupt when they complete the data transfer. While setting up the channels is the responsibility of the central processor, the actual transfer is timed by the peripheral attached to the other end. External equipment, like hard-drives, or front-end computers connected to these channels, interpreted the data presented on them and replied accordingly.

The only ‘internal peripheral’ available is a simple cycle counter, called the real-time clock. This can be used to measure time (as the name suggests) but can also be used as a primitive performance measurement device as well. The curious thing is that there is no periodic interrupt source. This clock can count (clock) pulses but can’t generate interrupts. The rest of the peripherals aren’t like that either, they can only generate interrupts on completion of a memory transfer. This means that the ‘kernel’ – called monitor in these machines – had no way to interrupt a user-mode application. Pre-emption was impossible. This was not seen as a problem in the days of batch-processing machines, like the Cray-1, and of course was quickly remedied in further generations.

Memories

Even back in those days there were two kinds of memory: SRAM and DRAM. In fact there were many more. There were magnetic drum memories (no, not as storage, but as main memory), delay-line memories and ferrite-core memories. Others too. By the mid 70’s however RAM became more and more common-place, and if you were to build the worlds fastest machine – where cost is not an issue – you choose SRAM. So that’s what Cray did. He used 1024 bit (yes, bits!) memory chips for the main memory. To build the largest Cray-1 configuration (512kWords of memory, 64-bits per word), you need 32,768 of these chips. In reality even more since you need to implement ECC as well.

This huge bag of chips got organized into 8 or 16 banks. The banks are straddled so that consecutive addresses fall into different banks. While the memory ran at half-speed of the CPU, each of the banks could deliver one data per this 40MHz clock cycle. The peak bandwidth of the memory subsystem is consequently 5.1GBps. In comparison this is roughly the bandwidth a 128-bit DDR2 system, that was used in PCs just a few short years ago.

And this is the point where we run into accurate re-implementation on todays HW: the peak bandwidth of todays DDR2 and DDR3 memories is higher than the bandwidth of the Cray-1 memory subsystem, the **timing** of these memories is quite different. DDR devices are very slow to respond to a read request, but once they start going, they're quite fast in delivering a burst of consecutive data. The 16-bank SRAM setup of the Cray-1 was relatively quick to respond and continued to be quick in subsequent beats of a burst. When non-sequential data was requested, their performance varied a bit depending on the stride (remember, V registers support this addressing mode), but never more then by a factor of 2. In DDR, non-contiguous bursts are not supported, and restarting a new burst for each of the reads is several times slower then a contiguous burst would be. The only possible way to re-create a Cray-1 memory in a cycle-accurate way is to use SRAMs, which is expensive. Several hundred dollars just for RAM chips.

Plan of attack

After a few weeks of tinkering, I've come to the conclusion that the most complete documentation available for the Cray-1S variant. It also adds some nice features, like the interval timer to support pre-emptive multitasking. I decided to concentrate on that machine.

My plan was to start with a simple (not cycle-accurate) instruction-set simulator to learn more about the platform. It doesn't have to be fast, but this would be the playground for making sure all instructions execute as they were intended, that I have gotten the non-IEEE floating-point implementation right and so on.

Once that is done, I thought, I would move on to the much-harder-to-write cycle-accurate software simulator. This would be used to find and patch holes in my understanding of how the machine worked. I could use the previous simulator as the reference platform and – when I find differences – figure out which one is right. It would also be relatively easy to experiment with which parts of the system need to be cycle-accurate, and which ones do not. Maybe I can get away with a non-cycle accurate memory after all…

Finally, when I have the second simulator done, I would go on to an FPGA implementation. At that point I have two simulators as reference so – while debugging the HW is much harder – I have things to compare it against.

Is there anybody out there?

However there lies the problem: in order to validate any of these implementations, I need code. Original code, that executed on the original machine, preferably with known expected results. Without that I can build the HW – there's enough documentation on that – but I can't tell if my machine is doing even remotely the same thing that the original did!

So the hunt for SW began. With very little success. I even contacted Cray both formally and informally through a friends-friends-friend but nothing. I could only find a small piece of code that listed a few floating-point numbers in Cray format and their real value. This was enough to write a somewhat accurate floating-point class. But that was it.

I have found one organization, the Computer Museum-Muenchen, who supposed to have an operational Cray Y-ML EL machine. What's more, they have these machines connected to the internet, so you can log-on to them remotely. This is something! As the architectures are close to one another, this could help ferret out some of the problems. If I'm lucky, they have a compiler on it, that can generate Cray-1 binaries too. The trouble was however that their machine was out of commission at the time. Check them out here! They seem to have gotten the machine back online by now.

At about the same time, Chris Fenton started publishing his results of building a Cray-1 in an FPGA. His project made some waves though the net, so I though, great! I'm at least not alone! I've contacted Chris and asked him about the SW situation. Unfortunately his reply was just as disappointing as my experience was up until then: no code to be found.

Is this the end?

This is where the project was in the summer of 2010. I had no code and no hope to get some. Without code, the project was doomed. In fact without substantial amounts of original code, even if I succeeded in building an accurate Cray-1 implementation, it's a rather useless piece of junk; not much more than the existing giants collecting dust in the various computer history museums of the world. It only makes sense to build such a thing if I can fire it up as it was originally intended in all it's batch processing glory.

So is this the end? You've probably guessed it's not, but this is where I left the project in 2010.