Parallels

Author : Andras Tantos

Introduction

It’s been a while. Not because I haven’t been busy with the project, but because I didn’t have anything complete enough to share (a hint: I know much more about FPGAs now). Recently, while I was waiting on some other things to materialize, I took another look at the simulator and have done some some meaningful – though not revolutionary – updates. It’s nice to be back.

So what’s going on?

The major change I’ve implemented now in the simulator is to break it up into multiple threads for parallel execution. This can be controlled by the MultiThreaded switch in the config file. It is set to ‘yes’ by default, which splits the execution up into on thread per IOP (and their peripherals), one for all the mainframe CPUs and one for all the mainframe CPU peripherals. In the current configuration, this adds up to five threads. Would it be possible to split each mainframe CPU into its own thread? You bet! But there’s no way to test it as the only image I can run is a single-CPU one. So, for now, it’ll stay as it is.

Another related new feature is that you can mark IOPs (I/O processors) as ‘nonexistent’ with the Exists flag, which defaults to ‘yes’. What it does is if an IOP is marked nonexistent, its corresponding thread terminates immediately, not consuming further CPU resources.

While the change sounds rather straightforward (after all all of the processors in the real Cray X-MP were running code in parallel) and in fact the changes needed weren’t too painful, the problem isn’t that simple. There’s a huge difference between a thread and a physical CPU as it turns out even on systems where enough cores are available to dedicate one for each thread. There’s thread scheduling involved in every multi-tasking OS, so threads go in- and out-of execution asynchronously all the time. This means that now simulated CPUs can get out of sync with respect to one another much more frequently then before in the single-threaded model, or in the real HW for that matter. Any synchronization bugs, missing interlocks, tight timing assumptions in the OS image would come to the fore-front in seemingly random, un-reproducible crashes and weird behavior. Sounds fun? Well, it was…

I’ve uncovered two problems so far with who knows how many lurking in the shadows.

Patch or not to patch?

The way the IOPs boot is rather convoluted. Each time an IOP is released from reset, its DMA channel moves 64kWord of data from buffer memory address 0 into the IOPs local memory. After that, execution starts at address 0 in local memory. Essentially, the DMA initializes the local memory before execution. So far, so good. However, we have 3 IOPs to boot, each with a

unique boot image, but all will read from buffer memory address 0. When you power on the machine, only one of the IOPs (IOP0 or MIOP) gets out of reset. This processor is responsible for booting the reset. The way the OS designers solved the boot problem was that the boot image for each IOP was almost identical with the exception of the IOP ID (0 through 3) coded in a word at address 0x0CF0 in the boot image. Execution during boot branches depending on this IOP ID field. So, when it is time for IOP0 to release IOP1 from reset, all it needs to do is to patch up the IOP ID field in buffer memory, then release IOP1. When IOP1 DMA-ed the boot image from buffer memory into it's local memory, IOP0 can patch the buffer memory image back to it's original content. Same for IOP2 and IOP3.

Nice, relatively clean solution. Trouble is however, that IOP0 now needs to know when it's safe to patch back the buffer memory. If it does it too early, IOP1-s DMA would potentially fetch the wrong image (containing the incorrect IOP ID) and would think it's a different IOP then it actually is. So, how did the long-gone coders of the boot process solve this little hick-up? This is how:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Dead start IOP
@ Input: OR[26]: IOP number
0x4904 A = 3539 (0x0DD3)          @ Load IA channel number to target CPU into OR[27]
0x4906 A = A + OR[26]
0x4907 OR[27] = A
0x4908 A = (OR[27])
0x4909 OR[27] = A
0x490A A = OR[27]
0x490B P = P + 59 (0x4946), A = 0    @ Bail if IA channel is not set up
0x490C A = OR[27]
0x490D A = A - 12 (0x000C)
0x490E P = P + 56 (0x4946), C = 1    @ Bail if IA channel number is too large
@ Patch up CPU ID in memory in preparation to transfer it to MOS memory
0x490F A = 3312 (0x0CF0)
0x4911 OR[1] = A
0x4912 A = OR[26]
0x4913 (OR[1]) = A
@ Set up MOS transfer
0x4914 A = 0 (0x0000)             @ Destination address: QWORD around 0x0CF0
0x4915 OR[17] = A
0x4916 A = 3312 (0x0CF0)
0x4918 A = A > 2 (0x0002)
0x4919 OR[18] = A
0x491A A = 3312 (0x0CF0)          @ Source address: QWORD around 0x0CF0
0x491C A = A & 65532 (0xFFFC)
0x491E OR[19] = A
0x491F A = 1 (0x0001)             @ Transfer size: 1 QWORD
0x4920 OR[20] = A
```

```
0x4921 A = 3 (0x0003)              @ Direction: to MOS, blocking
0x4922 OR[16] = A
0x4923 R = OR[0]+6049 (0x17A1)     @ Transfer to MOS memory
@ Release the target IOP from reset
0x4925 A = OR[27]
0x4926 A = A + 1 (0x0001)
0x4927 B = A
0x4928 A = 3 (0x0003)
0x4929 IOB , fn001
@ Reverse the CPU ID patch above
0x492A A = 3312 (0x0CF0)
0x492C OR[1] = A
0x492D A = 0 (0x0000)
0x492E (OR[1]) = A
@ Set up another MOS transfer to undo the patch in there as well
0x492F A = 0 (0x0000)              @ Start setting up MOS address
0x4930 OR[17] = A
0x4931 A = 3312 (0x0CF0)
0x4933 A = A > 2 (0x0002)
0x4934 OR[18] = A
0x4935 A = 8 (0x0008)              @ Wait some (loop through below 8 times)
0x4936 OR[27] = A
0x4937 A = OR[27]
0x4938 P = P + 3 (0x493B), A = 0
0x4939 OR[27] = OR[27] - 1
0x493A P = P - 3 (0x4937)
0x493B A = 0 (0x0000)              @ Loop terminated: remove the reset from target IOP
0x493C IOB , fn001
0x493D A = 40000 (0x9C40)          @ Wait quite a bit for the initial MOS transfer of the target
IOP to complete
0x493F A = A - 1 (0x0001)
0x4940 PASS
0x4941 P = P - 2 (0x493F), A # 0
 0x4942 A = 3 (0x0003)             @ Direction: to MOS, blocking
0x4943 OR[16] = A
0x4944 R = OR[0]+6049 (0x17A1)     @ Finish up the unpatching of the MOS memory
0x4946 EXIT
```

They didn't. They simply set up a loop, saying, surely, this many clock cycles is enough for the DMA (MOS in Cray terminology) to complete. It in fact is in real HW, where memory timing is very predictable.

Not so much in a multi-threaded SW environment. To fix this, I decided to halt IOP0 when it releases one of the other IOPs from reset (address 0x4929 above) and only allow it further progress, once the target IOP started execution as well. Not a terribly nice solution, but this guarantees proper interlocking between the different IOPs.

Out of buffers

Are we done yet? Well, not quite. The next problem came up when I tried to start the mainframe CPU (using the START command in the IOP console). It would – occasionally – result in a HALT 015, a fatal condition. Some digging around turned address 0x09c2 up as the culprit:

0x09C2 R = OR[52], A # 0
0x09C3 PASS (with d field set to 015)

At this point all I knew was that A somehow became non-0, so the call happened (operand register 52 or OR[52] contains the address of the fatal handler and the word after this call contains the error-code conveniently encoded as one of the many PASS instruction options, so if the condition is false, execution simply flows through).

The line immediately before the check is a call:

0x09C0 R = OR[0]+5756 (0x167C)

The routine called attempts to allocate a buffer to transfer data between the MIOP and the BIOP. The reason for why this is necessary in this case worth its own discussion, so let's save that for later. For now, lets assume it's needed. This call fails because all data-transfer buffers are currently in use. That is, they are awaiting for the BIOP to get around and process them. Some further digging showed (and we're deeply in heisenbug territory here so I had to be very careful what kind of diagnostics I could use) that it's not that the BIOP simply stopped accepting data transfers from the MIOP. It's just that the MIOP ended up flooding the poor thing with so many requests that eventually there was no more space left in the pool.

Obviously the designers didn't think that could happen, the code above is the equivalent of a kernel-level assert.

I've been wrecking my brain on ways to prevent this from happening but I couldn't come up with a solution. The only path I could see forward was to patch up the IOP kernel code so that this condition either never happens or is handled properly.

The call-stack for this point in the execution starts in the overlay START3. This is a relatively simple piece of code responsible for copying the mainframe boot image from the MIOP expander disk to the mainframe memory. You can check out the disassembly for this overlay in START3.asm. Essentially, it opens the file containing the disk image, reads each sector (512 Words) of data out of it and transfers those 512 word pages to the mainframe memory with a system call to HPSW:

```
0x0131 A = 35 (0x0023)          @ Kernel function code (HPSW)
0x0132 OR[294] = A
0x0133 A = 0 (0x0000)           @ Target memory type: Central memory
0x0134 OR[295] = A
```

```
0x0135 A = OR[281]              @ Target memory address high
0x0136 OR[296] = A
0x0137 A = OR[282]              @ Target memory address low
0x0138 OR[297] = A
0x0139 A = OR[286]              @ Local memory address
0x013A OR[298] = A
0x013B A = OR[293]              @ Buffer length
0x013C OR[299] = A
0x013D A = 1 (0x0001)           @ NOWAIT: return immediately
0x013E OR[300] = A
0x013F A = 294 (0x0126)
0x0140 B = A
0x0141 A = 0 (0x0000)
0x0142 R = OR[9]                @ Call kernel function
```

Notice the parameter ‘NOWAIT’. This means that the kernel just sends the transfer request over the BIOP but doesn’t actually wait for completion. The loop that wraps around this call contains no synchronization with the BIOP, so if for some reason the MIOP goes around this loop too quickly, we can exhaust all the available buffers. That’s exactly what’s happening.

The documentation of the HPSW call states that it can be called in a ‘blocking’ fashion as well, (NOWAIT cleared above). This is what I need! If this call blocks until the BIOP takes the data, I can’t run out of buffers. All I need to do is to change the A = 1 operation into A = 0 at address 0x013D and I’m golden.

Well, almost. See, the problem is that the call in that case is not really blocking. What it does is it puts this ‘activity’ into a waiting queue and goes on to deal with other activities. In modern terms, it removes the executing thread from the ready-to-run queue, puts it in one of the many wait queues and context-switches to another thread. (Yes, this is on the I/O subsystem of a batch-processing mainframe from the early ‘80-s!) The trouble is that the context-switch needs some assistance from the caller on this system. Notice the A = 0 at address 0x0141 just before the call!

On kernel calls, where the activity could be suspended, 'A' describes the part of the huge operand register set that needs to be preserved as part of the context. The lower 9 bits contain the index of the first OR register to be saved and the upper 7 bits contain the number of OR registers to save. So, what to set A to? Well, there are other kernel calls in START3, some of which could result in context switches, so we have some examples to go with. As it turns out, the magic value is 0x1D18 or 016430. This means that we have to save 14 registers starting at OR[280].

Good, so I have to change the A = 0 into A = 0x1D18 as well. Right? Right, except it’s easier to be said then done. You see, A = 0 is encoded in 16-bits, but A = 0x1D18 needs 32-bits. I need to somehow free up on extra word around the call to fit the changed instruction in. Luckily the compiler used for these IOPs wasn’t very smart (in fact it wasn’t much more then a glorified macro-assembler), so the code after the call looks like this:

```
0x0143 A = OR[281]              @ Test if target address is zero. If it is, patch back the size and local address fields we've fixed up just before the call
0x0144 P = P + 2 (0x0146), A = 0
0x0145 P = P + 7 (0x014C)
 0x0146 A = OR[282]
0x0147 P = P + 2 (0x0149), A = 0
0x0148 P = P + 4 (0x014C)
0x0149 A = OR[286]
0x014A A = A - 32 (0x0020)
0x014B OR[286] = A
0x014C A = OR[282]              @ Move target address by the transfer length
```

Check out how the conditional jump is done around 0x0144! If A = 0 we simply jump over the unconditional jump. This could be written as this:

```
A = OR[281]             @ Test if target address is zero. If it is, patch back the size and local address fields we've fixed up just before the call
P = P + 7 (0x0146), A # 0
 A = OR[282]
P = P + 2 (0x0149), A = 0
P = P + 4 (0x014C)
A = OR[286]
A = A - 32 (0x0020)
OR[286] = A
A = OR[282]             @ Move target address by the transfer length
```

And this requires one less word to encode. All in all, this is a larger then anticipated change, but here it goes:

1. Change NOWAIT to WAIT as the last parameter to kernel call HPSW
2. Set up context information properly for the call to HPSW, which requires an extra word to encode
3. Re-code conditional check right after HPSW call to free up the extra word needed
4. Find the overlay data on the boot tape image, and patch it up.

All in all, this is the end result that you can find in xmp_sim.cfg:

```
0x25BCB { Size 2 Value 0x1000 } ; A = 0
0x25BCF { Size 2 Value 0x1800 } ; A = ...
0x25BD0 { Size 2 Value 0x1D18 } ;     0x1D18
0x25BD1 { Size 2 Value 0x7C09 } ; R = OR[9]
0x25BD2 { Size 2 Value 0x2119 } ; A = OR[281]
0x25BD3 { Size 2 Value 0x8607 } ; P = P + 7, A # 0
```

And with that – at least as far as I can tell, things started to work again even in multi-threaded mode.

How many DMAs does it take to move a byte?

Let's come back to the discussion of why the MIOP needs to talk to the BIOP during mainframe boot to begin with. The way the I/O subsystem was designed was that the IOPs are largely independent. They have their local memory only they can access. There is a shared 'buffer' memory, which they have DMA access to through something called the MOS channel. This is to say that the buffer memory is not part of the addressable space of any of the IOPs. If they need something from it, they have to use a DMA to copy the data between buffer memory and local memory.

The IOPs have another peripheral (a pair of dedicated channels for each IOP pair) that they can use to pass information between one another. It is sort of a single-entry mail box. A 16-bit register with a valid bit and interrupt generation capability. The source IOP can put a word in there and get an interrupt when it was consumed by the destination IOP. The destination also can get interrupted any time the source put something in the register.

As you can imagine, this type of communication is not terribly fast. It's good for passing alerts, but actual data transfer would happen through the buffer memory: the source IOP would DMA the desired data into a portion of the buffer memory, through the inter-IOP communication alert the destination IOP about it's existence then the destination IOP would DMA the data back into it's local memory for consumption.

Now, for that to work, you need to somehow manage the buffers that are used for this communication in buffer memory. You need to know which pieces are available and which are used up. Of course you would also need to know when the destination IOP is finished DMA-ing the data locally so you can free up the allocated buffer. You need in total two DMA transfers and at least four interrupts to transfer any sizable message between two IOPs. If this sounds a bit too convoluted, you are not alone, but who am I to criticize the design of this machine.

Now, the next problem: if we want to pass information between the I/O subsystem and the mainframe, we have a similar setup: a low-speed communication channel, mostly used to pass alerts, and a high-speed communication channel for bulk data transfers. The low-speed channel is hooked up to the main IOP (MIOP) since that's the one coordinating all the work. Bulk data mostly comes from the disk drivers though and those are hooked up to the BIOP (or DIOP if it exists) so the high-speed channel is connected to the BIOP.

But this is not all! The I/O subsystem has its own set of peripherals of it's own. They are connected to something called a peripheral expander, and they consist of a printer, a tape drive and a hard disk. This is the hard disk that contains the boot image and is separate from the hard drives that the mainframe normally uses. Better yet, the expander is connected to the MIOP.

So, to put it all together, when you try to boot the mainframe, you have to read the data from the expander disk connected to the MIOP and transfer it to the mainframe memory through a communication channel connected to the BIOP. In order to do so however, you have to transfer the data between the MIOP and the BIOP which involves copying it out to buffer memory first

from the MIOP and back into the local memory of the BIOP.

Easy!

What else?

Expander disk geomtry

I've changed the way the expander disk emulation works. As it turns out, the expander disk had 5 heads, 823 tracks and 35 (512 Byte) sectors per track, making it an 80MByte disk. The expander disk controller had the ability to read multiple sectors at a time and the wise designers of the IOP subsystem decided to always read in 8 sectors at a time. Since 35 is not divisible by 8, they simply decided to only use the first 32 sectors of each track, letting the last 3 going to waste, resulting in a usable capacity of ~64MByte. Probably not a big deal for any of you, but in case you had custom expander disk-images made, you'll have to re-create them, the old ones are incompatible with this release.

Build changes

I've cleaned up the build process quite a bit. Now GCC and boost versions are auto-detected. I've also tested and checked in 64-bit Visual Studio projects and generated binaries. 64-bit Linux also works, most likely 64-bit Cygwin and MinGW do as well, though those are untested. The new makefiles however need many more posix features, so for MinGW you'll have to use MSys to build. Cygwin comes with its own bash prompt, use that.

Downloads

You can download the latest version from here: cray_xmp_sim-0.95.zip