Dendou Oni Supercomputing Research & Development Project – software notes

As you can see, I have almost complete list of the hardware to be eventually bought and used in Dendou Oni – some parts needed to be fairly standard so the pains of working on something would be lesser compared to the incurring headaches.

Now, it's ALL about the software (and some hardware). I am still getting hold of what's a yes and what's a no in the Propeller Tool, as it's a bit different than what I am used to be doing on an x86 assembler (it's possible to assemble Propeller ASM in the x86 assembler, but no guarantees: They all varies.) but similar to PowerPC processor programming.

Propeller have few things in common with the IBM PowerPC processors in programming environments. They both requires the usage of so-called Explicit Container Labeling – such as when it's right to label the RAM vectors as a labeled container or not – kinda like labeling the Strawberry jam you just made so it's easy to differentiate between Raspberry jams and the Strawberry jams since they looks nearly the same. PowerPC does the same thing, because there are few vectors that's exactly the same but is therefore a container of something else – like the jams. And, they both are Load-Store RISC processors which relies on the RAM addressing and spooling – they just treat some of their GPRs (General – Purpose Registers) as the RAM memory storage while it's not – instead it's called Vectors. Yet, vectors still end up being saved onto any RAM the CPU want it on. Nearly 30% of times, the vectors contain the integer / floating point instructions, even though Propeller do not have the FPUs. The instructions out of the RAM storage vectors is that things that tells the CPU what to do with the attached user instructions / data and what to except with it.

So I have covered some of the basics. However, it's now clear that some of the explicit GPRs are referenced by the Spin tools and swap that into the machine codes that's being manufactured upon the texts – I have done few experiments: First, I compiled few of the SPINs and then opened with MS Notebook (or MS-DOS EDIT.COM), and read the weird symbols to see how it should be doing its jobs. (Reading strange symbols when I compiled the programs and analyze that in MS editor is a hard-earned talent... It took lot of code-banging and messing around to get the crystal-clear image in my head. I have been doing away entirely in x86 and x86-64 machine codes – you would notice big difference when you view it in your favorite Hex editor of the Long Mode activation example codes plucked from flatassembler.net and compiled with FASM afterward. In the Hex editor [or if you want the hard way, use Notebook or EDIT.COM] window, the top half's totally 16-bit [Real-Mode], middle is 32-bit [Protected-Mode] and last bottom is 64-bit [Long-Mode] – it's the “word” byte size that's profoundly different between three operation modes.)

Now, what to do with Linux? Hmm? Taking 1 Terabyte hard drive into account, I would have to build an ATFS mounter and try to stuff it all in 2MB Magnetic RAM (or 512KB FRAM) hooked up to the MPC8377 PowerPC processor to put ATFS onto the hard drive (to deal with 4KB cluster requirements) – that is if any Linux volume mounters are queasy toward larger hard drives (I heard that SaneFS won't work on the hard drives over 2 Terabytes due to very complex clustering requirements).

And, I would have to include both x86 interpreter (for just in case I run across the x86-only Linux softwares I want to use), and the SPIN interpreter (to deal with the SPIN files to analyze for some of the pin usages and then modify it to prevent the Propeller II arrays from getting smoked – not everyone would know what to except when using Dendou Oni whether in person, on-line or by mail [floppy disks, CD and/or memory cards] so I would have to put in fault toleration protection software in there to protect all of the Propeller II chips, although the user codes would be 90 – 99% untouched, just have to be actually executed by the PowerPC CPU to be sure that it's safe for the Propeller II chips, electrical-wise – it wouldn't hurt having the extra insurance policies, as it will invariably save the lives of Propeller II chips should the pins becomes wrongly used).

While I speak of the insurance policy, the PowerPC processor will just have to check for the pin usage in the SPIN files and convert them into the vector address (to emulate that pin usages – through the AMD Radeon HD4350 or HD5450 video card and through the sound processor chipset), so that way the signal distortion or short-circuit won't happen. Optionally, the MPC8377 CPU can run like an actual Propeller II chip by running the virtual COGs to find out how to spread the resources around on the hypercube card to prevent overloading the Propeller II or impacting their RAM usage – thus cutting the computational times short (as it can actually explode from 700+ MIPS each superscalar COGs up to few hundred GIPS in the array boards altogether). 700+ MIPS is basically achievable with COGs running pure ASM at 200MHz or faster – of course, it won't be broad. But what the heck, it's a moving target anyways. It got four-stage pipelines which is what gives it a berserk computing power for its size, which is a big help for the interpreted SPIN opcodes – six instruction in a pass qualifies the COGs as superscalar engines. A good news for the microcontroller addicts vying for the raw computing horsepowers outta small Silicon die.

Superscalar or not, just a CPU execution unit alone just won't cut the mustard, so it's totally desirable to have many Execution units. (Out-of-order processor is kinda like supercomputer, but on transistor levels, it's mostly there to help with the gnarly instructions.) With so many execution units, there come a dilemma in reducing the chance of them fighting over the resources (the reason Hub northbridge circuitry's used in Propeller microcontroller) so I figured that it would be necessary to have the FPGA on top of the “spider webs” of Propeller II's traces to help keep the Propeller II's sanity in check. 70% of what's left out of the Propeller II chips are largely in both software and firmware – the FPGA still can do out-of-order IO pipelining providing the electrical and IO rules aren't violated (FPGA still can annoy a busy Propeller II and it can either ignore the FPGA or spit out “NRDY” packets to tell the FPGA that it's still working) but the data packet “crashing” is not what I would want happening – the data can be gone forever. The packet crashing can partially be avoided by performing the frequency hopping on propeller II chips (also software – using the DAC / ADC as an IO pipeline to squeeze the opcodes into what's become of the single wire / trace and back to the same data before it was “squeezed”, before being demodulated by both COGs and the software to sort out what's good and bad. After good data are collected onto the SDRAM / Hub RAM, it will be executed by the free COGs) to keep the data corruption to the acceptable level (8:10 bit encoding, or more complex and somewhat better – 8:14 bit encoding).

What's the nature of softwares to deal with the DAC / ADC? It is somewhat complex but is mandated to be kept as small as possible to fit on COG RAM, and perform the bit-slicing architecture to ensure the data recovery in the area full of noisy RF sources such as when the CB radios are being used. Bit slicing method in Propeller II should be done in a way: at 100MHz IO frequency (as a metronome reference along with 10MHz MEMS oscillator clocking input), the bytes should be translated in 4:32 and, then add 4 bit into ten bit error recovery packets, and then summed up afterward into 50 bit packets.

Why 50 bit? If you fold it down such as after the butterfly mirroring, there is 25 bit left very, very useful for the 128-bit ECC checksum calculation: It can fill in the holes in bit packets. Therefore, it's very, very complex but is really practical in very high speed IO (cellular phones and the towers always use the bit recovery architecture like this one). How do I get 50 bit packet? There is useful 32-bit data and 4-bit voltage reference, with 4-bit data copied / borrowed into ten-bit ECC checksum packet, all adding up into 50-bit. 1,000 bit packets (20 slices) coming in will contain 80 bytes totally usable, along with 280 bits (35 bytes) for the required ECC checksums. I have been playing with the calculator and thinking in my head to find that value that's best suited for the noisy area.

If it's made way too simple, it won't stand a chance in very noisy area as well – JEITA and FCC policies (and few other foreign RF laws) clearly stated that you're on your own when making a nuke-proof hardware (in term of RF interferences). “Why not make it simple so it's easy to troubleshoot” Well, I wish. But the data packets wouldn't even survive when bombarded by a 3 Watts CB radio. (I have an ancient Technics somewhere in the box – very nasty thing to be put near the electronics that rely on higher speed IO and pressing “Call” button, it can really mess up the CRC checksum in most cases – PC motherboards are pretty smart though and can easily deal with damaging low-frequency radio. I wonder about the 2 Watts 900MHz RF outta the police CB [anyone here in USA can buy one, though expensive] being placed near the digital DSP board.)

So, the software in the Propeller II to deal with the IO should be robust in term of handling with the non-intentional and intentional radio mash-ups, as well as being able to shoot the data through the self-generated “white-noise” present on the traces. “Define white noise, please.” Okay, reconsider that there's a 50 bit packet shooting through a trace within 100MHz time-domain. When I say “time-domain”, 100MHz (or faster) is the maximum clock that the Propeller II can squeeze out of the IO pins in this setup – although I may try doing the highest IO clock to the PLL's design limit to obtain highest speed bandwidth, providing that the SDRAM won't mind. Within 100 MHz, there may be ten to twenty RF channels within the Propeller II's IO traces intended for to talk with each others and with the main PowerPC CPU via the FPGA switcher. Define the bandwidth within 10 channels: 50 bits for 100MHz, 50 bits for 90MHz, 50 bits for 80MHz, 50 bits for 70MHz, and so on to 10MHz, totaling up to 500 bits in one burst – easily approaching up to 500 MB/s each pins for up to 20 5MHz-spaced channels. And with all of that frequency hopping going on, all you would see on the oscilloscope set at 100MHz would be a messed-up jumbles of weird-looking waves – concealing the information within itself, also what you would be excepting with the spread-spectrum clocks and other white noise RF sources.

Also the strict terms in “Time-Domain” here referring to the IO are pointless because the byte width usually varies – if there are nothing to be sent, the Propeller II stops sending out the RF, now basically eavesdropping and wait for the requests. And, if there are not much information in the whole 50 bit packet, there's hole in it, and some partial ECC information, right in the same width as a fully-stuffed 50 bit packets. It only refer to the frequency being summed on the pin (the reason you will see the white noises on the pins right on the screen of the oscilloscope – it's really for transmitting the data rapidly). Also, when the Propeller II flips the state on the pin (same as when it got nothing to send out), it will just absorb the RF, and then demodulated within the processor circuitry and the executing software.

And, I am sure that all of the complex IO delivery architecture would really push the Propeller II chips and the multi-layer PCB to the limit, hence the reason for 50 bit packets. If one 50 bit packet being nibbled is bad and Propeller II finds it distasteful, it can just simply throw it away and ask for new one and piece it in the partially-constructed thread – the same way the Ethernet works its magics.

“Why model it after Ethernet?” This hardware is quite robust. You can inject the garbage packets into the cable or switch, the NIC will just simply ignore it no matter what. But if it's succeed, the NIC will basically sit idle and ignore you until you give up, and then it will catch that opportunity and shoot its packets through at three chosen speeds; 10Mb/s, 100Mb/s and 1Gb/s, then go idle again after the computer's satisfied. It also have up to 64KB RAM (very rare NIC can do up to 16MB buffering), for buffering and CRC checksum analysis – which the Ethernet earns the merit for being robust. The NICs have to succeed one of few task such as this: On the backbone plane (think I2C bus), the computers can shoot the packets at nearly the same times, but to the specific IP and MAC addressing from the computer that's wanted by the sending computer on the same circuit. The computers that's unwanted can just ignore it until the cable activity dies down. The same goes for the wireless Ethernet – although it would perform better than the Copper straight-through Ethernet networking due to the frequency-hopping radio architecture, which it can take a pick and broadcast its data over the airwaves (at extremely high frequency than even the lightning-fast Gigabit Ethernet, about 2.45 Gigahertzs, although a bit slow, from 11Mb/s to 250Mb/s) which the requested hardware will listen for (think of Ham radio or the CB radios – which you ask for whom you want to talk with and both of you guys retune to leave the others out of your conversation).

The data communication is a tough business for the higher-speed processors (from 100MHz and faster) as the data cleanness is very, very important. Even few bytes matters. That's why the SATA, PCI express and even HyperTransport standards IO controllers use ECC a lot to ensure the CPU's happiness. Ever tried talking to the others over a crappy phone switch? Think about it – it affect your PC in the same way. I wouldn't worry about it now, as they have gotten quite sophisticated by now (For example - a 1TB hard drive's CPU got up to 100 million transistors – a 1TB WD Caviar Black I chose for Dendou Oni got a dual-core DSP so that exaggerated transistor counts isn't impossible – in fact, it will help a lot in ECC recovery as that endeavor requires lot of computing power). I would with ancient hardwares, though. (A fun fact: an Intel 4004 CPU made in 1971 is extremely simple, it only got 2,300 transistors while a new and extremely complex Intel Core i7 Sandy Bridge just out this year (2011) got 1.5 billion transistors in it – talk about the rapid paces in the computer manufacturing industry....)

Now, I have been thinking... What shall I do with the floating point? Tried asked around if two Propeller II would be better, yet no answer. So, I figured out a bit on my own. So, I will go down with the speeded-up version of CORDIC and Integer Point (AKA Fixed Point), and do the matrix fusion floating point (as there is 16x16 multiply so I can say it fills up the whole 256-bit multiplier vectors, but actually we would just scratch the bottom of the wooden barrel – doing away with 64-bit vector floating points as it will usually be enough for the Nintendo 64 graphic style rendering). Fusing the CORDIC and fixed point would actually throw away the chance of using the full 256-bit multiplier, but we need to compromise as the processor's just evolved from the first P8X32A family, and we're not exploiting the superscalar COGs in Propeller II chip as we don't have the actual Silicon chip yet.

“Will your idea, “Matrix Fusion Floating Point” be bad for Propeller II chip, and will it eat the memory space for lunch?” I doubt it, because I am only going to use what's already there in each COG's execution units (where the ALUs and Integer Units are located), as well as trying to exploit its inner workings. Memory space? Yes, because the user data dominate the usage of RAM memory: Multiply be 4 to 8, you're givin' away 64 bytes (it's painless in the recent processors as they got huge on-die RAMs as well – from 16KB to 256MB cache RAM space) but at least it can be written onto the external RAM until it gets used. How would I fuse it? First run through the CORDIC, then multiply that result and run it through processor pipeline using accumulators, then finally divide it to get the 64-bit packed results. (That's how certain circuitry of AltiVec FPU in PPC and AVX in x86 CPU works.)

I aim to get at least two words fused together, as accounted within the number of Integer Unit pipelines. It would be nice to have an on-die graphic processor as that thing's inheritably a floating point processor (floating point architecture handles color better than integer ones, as well as when rendering a very complex render object – like a teapot made entirely of liquid Mercury), for providing some speed boost, but unfortunately, we will have to get away with what we have on-die. If very accurate floating point is a must, there's 80 streaming DSP in AMD Radeon HD5450 that I can have assigned to the Propeller II hypercube board's use – the GPU's a superscalar out-of-order processor so it wouldn't matter if it's for the graphic or not – this GPU won't even care at all. (Note – the superscalar out-of-order graphic processor have the definite advantage of overtaking the complex graphic programming, and performing the floating points at the same times, whether it's for graphic or not. Most newer games use the remaining streaming DSP cores for the sophisticated AI architecture of the virtual peoples and the virtual animals, affecting their behaviors as you do something to them. In-order GPU will have trouble dealing with multiple different data, so if it isn't an out-of-order processor, we would have to depend on already straining and sweating multicore x86 processors for AI processing.)

But unfortunately, the Propeller II wouldn't have nice GPU... Even if it do, it's still too simple to do any useful floating point calculation, in other word, I will just forget about depending on the internal GPUs inside the COGs for the usage as a FPU. And, I will have to do the way by software. Ugh...

But still, if I confirm that the Propeller II's COGs can execute the instructions dynamically (out-of-order execution) by running very complex 3D graphic (a surefire way to tell if it is), then what's lost can be made up for its powerful processor logics. Meaning, the software can execute one thing at the time, and this same COG can execute the emulated floating points. (if I take 160MIPS from SPIN execution literally, I would get 100 – 140 MFLOPS each COGs – 800 MFLOPS to 1.12 GFLOPS a Propeller II die (chip) by using ASM-only VLIW-like floating point emulation. Not too bad.)

Here where I can be wrong, but... Suppose I used the speeded-up CORDIC / multiplication morph codes (the self-modifying floating point, similar to what's used by the PowerPC AltiVec and x86 SSE3 / 4 FPUs, in those processors in your own computer), and used the open-paged RAM buffering for floating-point storage and hooks, I suppose I could get up to 220MFLOPS, still measly small compared to just the traditional way, 140 MFLOPS. So, 220 MFLOPS for a COG, multiply by 8 cores = 1.8 GFLOPS. It's alright, but we do the fixed point more than floating point anyways – plentiful of raw horsepowers, up to 5.2 GIPS.

So, how would it compare to AMD's PowerPC-based x86 microprocessor, Phenom II?
To make it fair, say I disable all three cores, leaving only one core executing the SSE4 floating point doing the same floating point morph tricks, compared to 32 Propeller II: 166.4 GIPS outta 32 Propeller II – it beats Phenom II easily. 57.6 GFLOPS outta 32 Propeller II, Phenom II wins. A core outta Phenom II Deneb can execute up to 80 GFLOPS, and 124 GIPS, which is 30x more powerful than a Propeller II COG. I am not being pessimistic, but just showing that it got enough punches, that if the floating point routine is made very clean and truly dynamic, all 32 Propeller II would be able to beat the Phenom II's FPU. All of that contributes to the lack of floating point processor, but meh, we do fixed points more anyways – we can always squeeze out the magic math outta the COGs to the good last drop, as they're much more powerful than the previous generation, P8X32A.

To bring up why we use fixed point more than floating point: the cheap pocket calculator always do fixed point as it's cheaper, (but $20-40 calculator do floating point too), so there's some difference in how we apply our input and what we except from it as it spits out our data.

And, there are going to be 8 Propeller II chips on a board, so the integer data is basically to be distributed around the card carefully as the throughput can be as high as 70Gb/s (worst case scenario), so the processors don't gridlock each other. The software in both PowerPC processor and the FPGAs must compromise with each other, otherwise, the packets won't be going anywhere, or just going somewhere and disappear in other dimension... And then magically reappear onto the PowerPC processor's DDR II memory totally untouched, not what you want happening.

And, the firmware ROMs in the Propeller II chips are exactly the same, same operating system kernel, same drivers and, etc... With a difference, the Propeller II chips are to be called by the set-up interruption vectors (we can always do software emulation of nested interruption vectors), and the ROM ID. After all, the Propeller II chips will always ignore what's not matched to the ID, but one will snag it and process it. Kinda like Ethernet. That way, if the FPGA messed that one up, the packet will just fly around til it arrive to its destination, reducing the chance of having to re-request for the packets (Propeller II chips are also responsible for requesting the packets to piece up to what they have), so that way the processors wouldn't have to deal with some of the software latency issues of their own.

“I am curious... What's the Propeller II chips' maximum addressable nodes and memory?” Good question. They're same as the older 32-bit x86 processors, addressable for up to 4 Gigabytes of memory spaces (Actually 3.97 Gigabytes, because of the Power of 2), so it can address the memory big enough for the SPIN, but there were some catch: How can we address it explicitly? Hub's limited to 128 – 256KB RAM space. Luckily for us, we CAN actually force that by popping in explicit memory addressing into the intermediate RAM vector, so the processor would do what we want it to do: Expand its RAM space. Only you have to be able to have some RAM size pointer already ready because in some case we can't use the I2C SDRAM configuration EEPROM, it's non-existent. “How come?” We just have to solder what we can onto the board. SDRAM modules are sometimes out of question since they got full 64-bit pipeline (a waste of money, bandwidth, and size unless it's a old, decrepit SDRAM module outta your obsolete laptop) – we can only get away with 32-bit pipeline, sometime lesser.

And, nodes? Maybe up to 1,024 – maybe 4x (4,096) – depending on how well the Propeller II's executing OSes keep journals of what's going on, and how well you configure your FPGA. “Why FPGA?” It's very fast, and some of them are dirt cheap. (a dirt-cheap SRAM-based FPGA, Altera Cyclone III is also made on 65nm, good enough for higher speed switching) It also allows the hardware bug tracking and fixes (after all, SRAM FPGAs download the logic configuration data from the I2C / SPI flash memories). Over 4,096 nodes, the main processors and FPGAs may have to use either 36-bit or 40-bit RAM addressing since they just view the Propeller II as “memories” - Propeller II are Load-Store RISC processor, so it could make something difficult for something as simple as themselves to be to address the huge numbers of processor. 1,024 is what I am viewing at as my maximum design limit.

“So, how fast if just 1,024?” Very good question... Barring the FPGAs' electrical wiring latency (1 inch of wire equates to 1.2 ns latency), and memory contention – it can hit 5.3 TIPS and 1.8 TFLOPS. All of that speeds can be all too true, due to the latency from the node to node, but it can get to this exact number if we depend on it being mass-requested as in AMD Radeon HD graphic processors, all of its DSP cores – all of them – being requested and fed at the same time. But then we will run into two thing; RAM overload (that's what the hard drive's for...), and power consumption – 307 Amps out of all 300mA chips – up to 568 Amps Max. so we will have to use the separate ATX power supply to safely distribute the power to the chips without having the wiring insulators melting and dripping. It also depends on HOW well is the software written. Poorly written softwares will just waste extra CPU cycles getting' done – that's the reason there's OBEX in the first place – to study.