電動鬼 (Dendou Oni)

Dendou Oni is a multipurpose, Propeller II-based supercomputer, which is planned, at first, to have 16 or 32 Propeller II chips tied together in Hypercube arrangement (on the board itself), and on Common Bus arrangement (the seed computer, based on Freescale's PowerPC main CPU, notably e600 [G4], to interface the hypercube boards onto the CPU itself via PCI-express bus). Whenever the sufficient funds are received, the arrangements of up to 1,024 Propeller II chips may be possible in the future. And, if sufficient interests are also shown, I will gladly post the Gerber files of the printed-circuit board masks, and lists of few necessary parts for creating your own boards (seed computer's designs may or may not be released on the Parallax Forums website will depend on many factors, such as patents [like Freescale's IP], difficulties of soldering the finer-pitches Ball-Grid-Array [BGA] chips, like the main CPU with 1,033 solder balls under its high-temperature ceramic package substrate, and the willingness of developing one's downright complex computer. Oh yea, and one must be familiar with PowerPC programming language, but I will post the boot firmware as an easy way out, if the motherboard assembly, assignments are exactly the same. Although, with very careful programming, you can even [re]use your own decommissioned x86 [or PPC] motherboard with PCI-express bus, to talk to the hypercube boards.)

The real purpose of this supercomputer is to understand the basics of neuron-network AI (which several tens, hundreds or thousands of Integer Units (IU) would be working together or individually, just like the neurons in the brains of the living things), and the nature of advanced VLIW (Very Long Instruction Words) floating point architectures in the pseudo-FPU setup (like AltiVec in PowerPC CPUs, although it's a true FPU). The initial interest is in integer, like for AI hardware processing (Conway's Game of Life as an example – J-ALICE, although now abandoned, is also an interesting example too. Will have to find J-ALICE and re-port the source code for my own OS in this supercomputer.) Then, after that, I would test out the number prediction (like the Prime95's n-Prime calculation) using CLUT on Propeller II chips' boot firmware ROM, in VLIW fashion. (Since CLUT is similar to AltiVec and SSE2's RGB calculation, I would use few of the pseudo-FPU methods, with linear but carefully written VLIW floating point scripts, so it would give me an accurate result – which will take a lot of work to get it perfect and to IEEE-754-1985 standards [aka IEEE-80] so it's last on the list.) - which on Integer side, it's going to be very aggressive, and on floating-point, it's a bit iffy...

Now, the more important challenge will be in an attempt in preserving the energy efficiency of the SMPS located on Hypercube board. Since the Propeller generations will be shrunk two time smaller, now that both Propeller I and II are both 180nm (before the die shrinkage, the Propeller I was made on 350nm process), the power efficiency will be more of an issue, as the smaller the transistor becomes, the more leaky the Gate layer becomes, the standing reason the Propeller I and II are made with SOI (Silicon-on-Insulator – basically a gate layer sitting on glass [Silicon Dioxide] layer between the transistor Source / Drain structures), just so it won't suck up precious battery life – a consideration Parallax, Inc. wouldn't dare to ignore, since it was made basically for anything: Solar-powered, battery-powered or wall powered hardwares. However, I have since thought of a possibility of a more dynamic SMPS (so that way the unused COGs would be disconnected from the SMPS, and turned back on when needed so – and DVR [Dynamic Voltage Ramping] is also used for the COG consumption requirement) – that way the vampire tap (+5 V from a computer ATX SMPS) won't be stressed to a point that it would be both a definite fire hazards and wattage-waster. Now, I shouldn't conclude the possibility of the issues in power consumption from the chips of deeper lithography process.

Since Propeller II will eventually be made on either 90nm SOI or 45nm Hi-k Metal Gate [HkMG] process, the issue of keeping Propeller II / III functioning without dropping out will be exacerbated by the fact that transistor leakage will keep getting worse and worse as transistors becomes much smaller – The reason two methods, High-dielectric constants between Gate and Source / Drain (the reason HkMG is the logical next step for all future deeper sub-micron chips like late Propeller II or newer yet-to-be-worked-on Propeller III chips – to reduce the chance of leakage by placing Beryllium Oxide, Zirconium Nitride, Hafnium Dioxide / Nitride, or Tantalum Dioxide between the gate and the rest of transistor, reducing the chance of capacitive leakage), and dynamic LDO / switching regulator are going to be used much more by those smaller-geometry chipsets (and the entire hardwares). Also, HkMG process is the only way out in battery-operated hardwares (be it LED matrix sign, toys, some PDA, and/or industrial - like instrumental and measurement) since the wattage-to-megahertz efficiency is extremely important, the reason HkMG process was researched for 15 years, by semiconductor giants, such as IBM, Intel and AMD – and few others.

Now that important power efficiency have been discussed above, I now shall emphasis on cooling requirement (that mean overclocking issues too!) - as it's also very important to keep those microcontroller chips happy, or it would be more of a non-trivial, expensive repair. I can't comment on how much you guys can push the core frequency without cooking the Propeller II chip to death, since I haven't tested them yet. When I do have a Propeller II chip to torture so you don't have to find out the hard way, I would construct a test computer rig, to test its upper clock limit (seein' how much it can be overclocked without crashing, or dropping out) – as cooling is also very important here, I will have to cryogenically cool it to -10, to -30 degrees with Thermoelectric Cooler (TEC, aka Peltier Cooler), so that way I can probably reach up to 1.0 GHz without having it overheating – 1 GHz core frequency is not a guaranteed maximum COG frequency, which is why I will have to find out before I ever think about applying it onto my supercomputer's hypercube processor board without accidentally frying an expensive board ($50 or more). And, since TEC is expensive, it's not possible to overclock the hypercube boards to even nearly the twice it's intended core frequency – even though the boards will have the heatsinks mounted the opposite side of Propeller II QFN packages (so that way the heat can be dissipated from the die seat pad in the middle of QFN packages onto the heatsink – that way the Propeller II chips will run cooler) – heatsink isn't good enough for dissipating the flash heat that would occur when overclocking the chips (even a professional overclockers would know that it's not possible unless cryogenically chilled). But, running at its intended core frequency will ensure its longer life expectancy, anyways. Fans are a must for the heatsunk Hypercube boards since it's possible that it will be run for 24/7, so always be sure that it won't overheat without a warning – higher CFM fans are very loud and noisy (the reason it is very loud is because it was constructed nearly like a turbofan jet engine, thus ensuring higher-pressure airflow), but will always ensure that the heatsink will remain cold (resulting in much happier COGs).

Okay, with every design points covered, I am going to talk more of softwares now. There are few things I would like to cover, though. Clocking is one of them. First, I would be planning on using the re-programmable SiTime MEMS (Microelectromechanics Systems) clock generator for giving me the widest frequency selections (I really wish there are violate memory-based MEMS clock generator so there are virtually no wear on PLL selectors like it would with flash ROMs) to be used for the built-in PLL clock generator found on-die inside Propeller II chip for CPU clocking (COG core frequency) so I can actually change the clock or to temporarily overclock it.

And, another reason for the usage of the MEMS resonator is the stability of clock pulses (in large-scale computing), it MUST be accurate, or the worst-case software precision errors will occur like mis-matched HUB accesses among all Propeller II – 72ns latency may be acceptable for some software, but for operating system, and IO interconnection, it's wholly unacceptable. It has to be tight or that would fail, period. That's the reason the current multi-processors (with multi-cores) has to time themselves to the others and, finally, synchronize with each other before the jobs would be distributed among them. One solution is to have a FPGA-based northbridge (Lattice ECP2) and DDR-II DRAM chips to manage the IO flows. Another would be the softwares inside the FRAM boot firmware ROMs, and the PLL buses (inside the COGs), to perform synchronization.

Software execution methods are already very important, that I should now point out that the traditional VLIW standpoint are no longer efficient anymore, and that the future generation DSPs will must have Out-of-Order execution VLIW IU (such as those found in Intel Itanium II CPUs) in order to tackle the rapidly expanding complexity of the softwares and Codecs, to be more efficient. COG is one thing I would emphasis, why am I not going to use Propeller I? It's simply the nature of the DSP integer data that is to be fed into hypercube board. In-order execution are more predictable, but would cost the SPIN interpreter way too much CPU cycles, the reason it takes four ALU cycles to complete a so-called "one-cycle execution" of the machine codes – one reason why you cannot even get 80MIPS out of 80MHz COGs, unless you're willing to take the hard-way route, and write 'em in C++ codes – you can only get 20MIPS out of a 80MHz IU inside the COGs with SPIN codes. Out-of-Order execution IU framework is the only efficient method to be able to pipeline the SPIN codes much faster, while there are maybe 4 ALUs (four IUs in one COG unit), or two IUs with two ALUs, to tackle the jobs SPIN interpreter gives to the COG itself, only faster than the one in the first-generation Propeller chip. With tasks being pipelined in both execution flavors, IOE or OoOE, it would be able to complete four instruction dispatch tags in just one ALU cycle, thus reducing the SPIN demands on COGs somewhat, giving you 160 MIPS for 160MHz IU core frequency – 640MIPS (or faster/more) in pure ASM or C++ is possible too. I consider ALU cycles very precious, the reason I am awaiting the Propeller II microcontroller, so I could extract lot of useful computing horsepowers in performing very complex Integer and Floating point data (I would use VLIW programming method of implementing the floating-point capabilities inside Propeller II, so it also will stay predictable and consistent within HUB call-back cycles – and keeping the COG busy without having to confront the pipeline stalling – although there are few clever tricks used by the newer OoOE DSPs, that will be used in Propeller II too, is to keep looping the "if – nop" until the real workloads comes in, to prevent the pipelines from stalling).

Now, how would you pull out the "if – nop" loop then put the COGs back to work? You don't have to do much, since the programmers at Parallax, Inc. have been working on to cut the work out for you, so you don't have to write downright fancy "dog's tricks", which can either make or break your projects, since the interrupt requester's being put in the ROM to really save on RAM / flash EEPROM spaces. In other word, you shall not worry way too much on this "pipeline stall protection", since it will do the jobs for you already, just simply stuff the data onto the HUB RAM and have the COG to fetch them (it will really wake up instantly) then do its job. Me? It really depends on how hard coded the softwares are and what they really call for. VLIW floating-point, on other hands, are really hard-coded, since there are no easy defined predictions of the branches being done inside the COGs, so it's really best to keep the COGs busy without having 'em go to sleep or whatever goes on in there when the ALU request table (aka scoreboarding) goes empty. Integer is really easy to predict since they always contain some predefined words which the scoreboarding LUT (located somewhere on the COG RAMs) keeps track of, so I shouldn't worry too much on Integer jobs, while I really have to with floating-points.

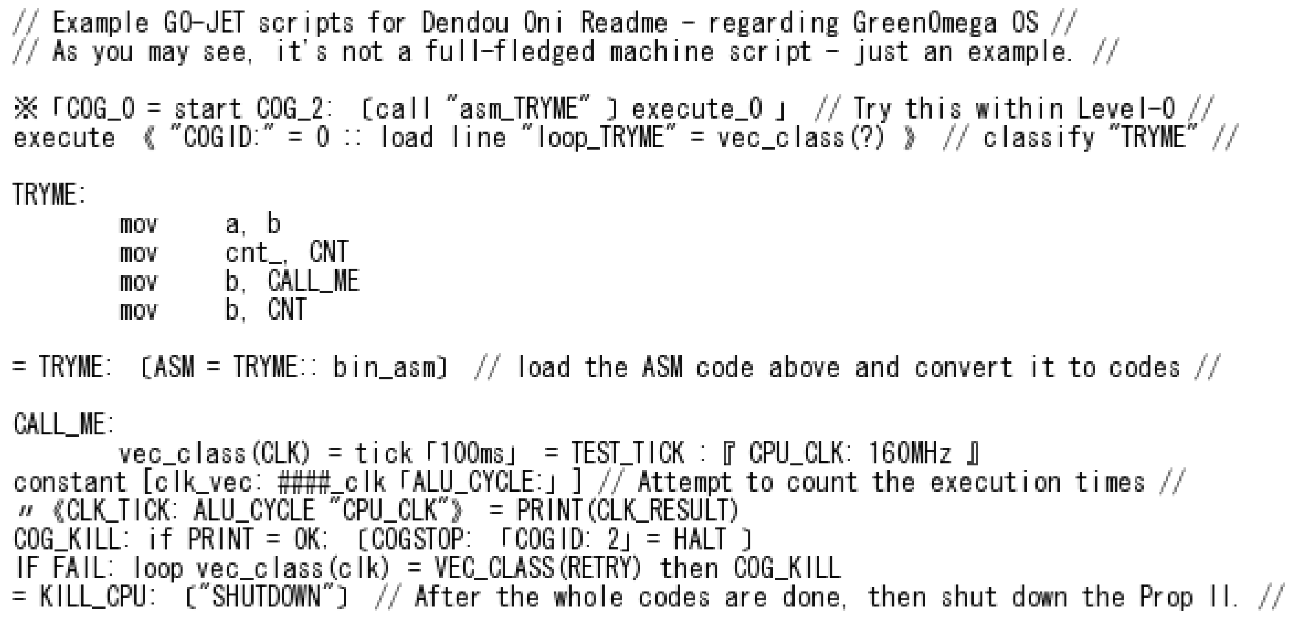
What about branch predictions inside the SPIN codes? 1. I really doubt it, since it's always being interpreted by the boot software, the “SPIN Interpreter”, so there are really no point in using the branch predictor, but there are always a chance that the SPIN Interpreter may ask for some assistance in branch predictions to guess the best, easy way out on the SPIN codes being executed. 2. The branch predictions, due to COG RAM size (2 to 8KB), are pretty much limited to maybe six or so branches to be called in. 3. Many of SPIN files in OBEX are so simple that there are really no point in using the branch prediction. But... The branch predictor inside the COG itself may be available for other type of machine codes, like our favorite, C++, PASAL, and JAVA. The problems with them are addled by larger amounts due to RAM size (although you can always cheat and use “evict cache” request, to save some branch data onto the external RAMs, but that would cost the IU the “wait_cycle_time” red-tapes in having to wait few hundreds of nanoseconds until the HUB allows it to do so...) - the reason you should be real careful not to pollute the SPIN files with too much branch hints, since it will cost COGs dearly. However, it's easy to do the branch predictions, one at a time, so the COGs won't have to flip over due to mis-predictions AND missed HUB windows, and that it can be allowed to evict the stale threads, to continue working on other things (remember the pipeline-stalling I was talking about?) - to provide the results, sans the evicted codes. (Although, the COGs in the propeller II can always retry the branch prediction when the most-recent threads are done with, and nothing to really do, to retry the execution of evicted codes. And, as a side of notice: The COGs will always do what YOUR OS tell it to do, so the retry permissions are always affected by the software being executed, not the boot firmware.)

What about the nested interruption (IRQ)? You're on your own, since Chip Gracey believes in simplistic computer programmings – that way even the machine codes are executed in pretty much predictable manners. (The IRQs are only more usable if the COGs have larger on-die SRAMs, since it has to possibly keep the evicted codes for a while then retry if the COGs have been interrupted by any events – such as incoming data packets, end-of-line call-backs, and/or loop breaks of the codes) – You can try and create such virtual PICs outta few COGs, but it is not an easy decisions, since some of you may want to extract as much computing powers as possible – I will have to sacrifice few COGs in the process too. However, for board-scale Prop II on-die hardware /software requests, I will be using a MIPS-based PIC32MX microcontroller as a PIC for the external interrupt vector controls / requests.

But if it's true that Propeller II's COGs are superscalar, Out-of-Order Execution (OoOE) engines, and you don't like the way it execute your codes, there are few ways you can force it to execute all of your codes in-order – although uCLinux would appreciate that you leave the execution mode alone, since in many cases, it would give Linux kernel a speed boost. Either ways, you can still assign the codes other than uCLinux to be run in IOE mode. Other than the COGs, the CPU inside the HUB northbridge logic will still execute In-Order, as it always has. Since HUB is meant to be giving the COGs the turns every time it “spins”, it doesn't make sense to make the system controller's IU more complex than what's necessary for the COGs, so it's also kept simple and smaller – while on other hands, lots of peoples have been pushing the Propeller I microcontroller chips to the limit, it DOES make sense to make COGs an OoOE processor to off-load the SPIN interpreter so that way it could execute the SPIN codes faster.

And, also there are few new features to be raving about in Propeller II: The built-in SPIN IDE which this chip can also actually compile the machine codes from the text-based SPIN source codes (and even scripts) then execute the resulting machine codes afterward. This compiler will also be very important in the supercomputer applications too – such as in this machine, Dendou Oni.

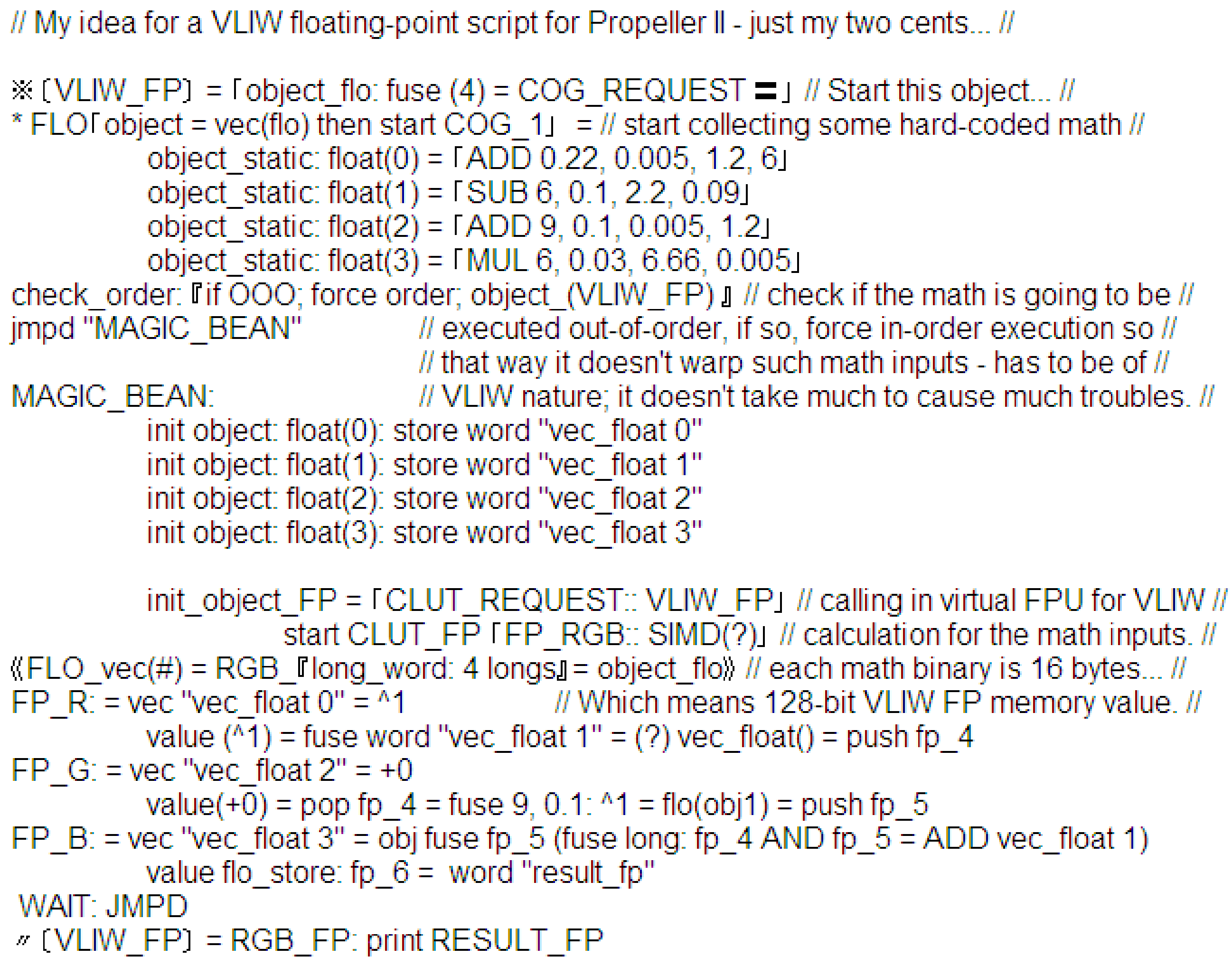
Since texts are hard to corrupt, while anything complex than a simple text – such as pictures, video files and machine codes, can be easily corrupted, thus sometimes executing the text scripts are the only fool-proof methods to get jobs done. However, since I am going to write my own boot firmware that will drop the usage of the on-die primary boot firmware – after being loaded, I may have to also include the same IDE software such as on the on-die firmware, recompiled in either .EXE / .BIN to be run by my own OS kernel in the 128KB FRAM, or maybe implement the BIOS call-back methods to summon forth the IDE software and have it compile the SPIN codes and then load them into idle COG to be executed. Also, while I have done few OS in the past, I may reuse the methods used by my own OS, GreenOmega 32 / 64. While Propeller II is 32-bit, I may build a firmware kernel based on GreenOmega 32 OS – if 64-bit, I will need to compile the kernel that's 64-bit but also written to handle the 32-bit machine codes (so that way ancient Propeller I codes can and will still run). Within GO-based firmware, I may also include the GO-JET (GreenOmega Japanese Electronic Teletyping) script software for the Propeller II to execute the GJS (GO-JET script) files as either BAT files (like in Windows) and/or the executable source codes. It may be strange to you, so it may be relevant to you or not, so you don't need to worry about the usage of GO-JET compiler – like I said, if enough interests are even shown, I will include everything on the Forums. I may also need to write the tutorial on writing the GO-JET scripts if needed to be. BTW, the human-readable scripts don't need to be written in Japanese Kanji, only it shall be written in English with some Japanese teletype symbols being used to tell the Propeller II what to do with the code snippets. Here's an example:



```
// Example GO-JET scripts for Dendou Oni Readme - regarding GreenOmega OS //
// As you may see, it's not a full-fledged machine script - just an example. //

※「COG_0 = start COG_2: 〔call "asm_TRYME" 〕execute_0」 // Try this within Level-0 //
execute 《 "COGID:" = 0 :: load line "loop_TRYME" = vec_class(?) 》 // classify "TRYME" //

TRYME:
        mov     a, b
        mov     cnt_, CNT
        mov     b, CALL_ME
        mov     b, CNT

= TRYME: 〔ASM = TRYME:: bin_asm〕 // load the ASM code above and convert it to codes //

CALL_ME:
        vec_class(CLK) = tick「100ms」 = TEST_TICK : 『 CPU_CLK: 160MHz 』
constant [clk_vec: ####_clk「ALU_CYCLE:」] // Attempt to count the execution times //
〃《CLK_TICK: ALU_CYCLE "CPU_CLK"》 = PRINT(CLK_RESULT)
COG_KILL: if PRINT = OK: 〔COGSTOP: 「COGID: 2」 = HALT 〕
IF FAIL: loop vec_class(clk) = VEC_CLASS(RETRY) then COG_KILL
= KILL_CPU: 〔"SHUTDOWN"〕 // After the whole codes are done, then shut down the Prop II. //
```

As you can see, I like to category every single of the machine codes into a spot where they belong, so that way the COGs' IUs would know what the scripts want for the text-based codes inside it. That way it's a bit easier for me to be able to write the GO-JET scripts (and programming the executing Propeller II chips in the process) and also be able to easily find out what went wacko, saving me some headaches and troubleshooting time. Although I do know what's easier for me could be hard for you, thus if enough interests are shown for the GO-JET console / IDE software, I will try my best to write the tutorial on GO-JET script programming, just for you to be able to code them the same way I do.

Anyways, I am going to say for my two cents (or my two Yens) about the VLIW floating-point processing methods in GO-JET script – resultant machine codes: What I am going to do is to try and use all four pipelines (or four ALUs) to attempt to satisfy the need for a result of a 128-bit VLIW SIMD floating points. However, my explanation of the CLUT is still crude anyhow (until I really get inside the Propeller II chips' minds later on to get the clear idea of how the CLUT is doing and what DSP functionality will it have) – so the RGB idea oughta do for now. So, here's my script:



```
// My idea for a VLIW floating-point script for Propeller II - just my two cents... //

※〔VLIW_FP〕 = 「object_flo: fuse (4) = COG_REQUEST 〓」 // Start this object... //
* FLO「object = vec(flo) then start COG_1」  = // start collecting some hard-coded math //
            object_static: float(0) = 「ADD 0.22, 0.005, 1.2, 6」
            object_static: float(1) = 「SUB 6, 0.1, 2.2, 0.09」
            object_static: float(2) = 「ADD 9, 0.1, 0.005, 1.2」
            object_static: float(3) = 「MUL 6, 0.03, 6.66, 0.005」
check_order: 『if OOO; force order; object_(VLIW_FP) 』 // check if the math is going to be //
jmpd "MAGIC_BEAN"              // executed out-of-order, if so, force in-order execution so //
                               // that way it doesn't warp such math inputs - has to be of //
MAGIC_BEAN:                    // VLIW nature; it doesn't take much to cause much troubles. //
            init object: float(0): store word "vec_float 0"
            init object: float(1): store word "vec_float 1"
            init object: float(2): store word "vec_float 2"
            init object: float(3): store word "vec_float 3"

            init_object_FP = 「CLUT_REQUEST:: VLIW_FP」 // calling in virtual FPU for VLIW //
                        start CLUT_FP 「FP_RGB:: SIMD(?)」 // calculation for the math inputs. //
《FLO_vec(#) = RGB_『long_word: 4 longs』= object_flo》 // each math binary is 16 bytes... //
FP_R: = vec "vec_float 0" = ^1                  // Which means 128-bit VLIW FP memory value. //
            value (^1) = fuse word "vec_float 1" = (?) vec_float() = push fp_4
FP_G: = vec "vec_float 2" = +0
            value(+0) = pop fp_4 = fuse 9, 0.1: ^1 = flo(obj1) = push fp_5
FP_B: = vec "vec_float 3" = obj fuse fp_5 (fuse long: fp_4 AND fp_5 = ADD vec_float 1)
            value flo_store: fp_6 =  word "result_fp"
 WAIT: JMPD
〃〔VLIW_FP〕 = RGB_FP: print RESULT_FP
```

As you may see, there are some delayed jumps – even a speculative, OoOE integer unit will still have to allow its pipeline to finish first – there are few pipelines which it can assign to several instructions at once BUT here, the COG's IU is forced to process the floating-point data in explicit order fashion, so it doesn't derail the whole operation – hence the reason why I am giving it a delay so it won't have to suffer some of “intentional” code bloats, or the pipeline freezes, so that way it can finish that FPU operations just in time, for to be able to do the next instructions or to do something else.

And, here too, you can see that the COG is programmed to do rather fancy floating-point tricks which it will have to take a while to calculate the fused object floating-points which in itself are very complex, four-words vectors which is 4 longs (indicating 128-bit floating-point values) – but I can also use only one word to do 32-bit floating-point, but then that isn't VLIW unless there are two words or more in an integer instruction.

And, regarding the GreenOmega OS in the Dendou Oni, there will be few GO-JET scripts (which can be user-modifiable) on the FRAM connected to the individual Propeller II chips. But, why every FRAM chips for Propeller II chips – why not just a chip? That's impossible, because the ROM ID must be unique for the bootstrap GJS to be set to identify the chip processing this script, ensuring the correct IRQ request ID to be requested for the thread packets to be given to the particular chip – WITHOUT interrupting the others. Also, if the wrong IRQ is requested and a busy Propeller II chip is interrupted, it can just simply ignore it (if ignored, the northbridge logic will check again and retry with corrected IRQ requests). And, if the ROM is virtually the same – all Propeller II chips are tied to an I2C FRAM, there will be a disaster waiting to happen: If the packet is forcibly shoved into a Propeller II chip doing something, it will simply crash requiring cold reset (pulling down the RST pin until it resets). And, that is not good: It will simply result in wasted wattage and GIPS. You would want it to do something REALLY meaningful, thus the reason for PIC32MX chip and the unique hardware ID / address for the Propeller II chips. However, you do not have to use the I2C FRAM or even a MRAM if you can't afford them, you can use cheaper EEPROM flash – but be prepared to sacrifice the freedom of trying to modify the data content – you only get thousands of rewrites so you only get few retries ONCE. That's the reason I am using the non-violate RAM, so I can have approximately the same rewrite cycles as a regular hard drive – that way I can always fiddle with the OS settings in every Propeller II chips until something fruitful comes out of it.

But how do I supply the information into Dendou Oni's clusters? There are couple ways I can really do. Here's the lists I could do away with:

- Ethernet uplink via seed computer (can be over the Internet)
- CD / DVD / BD – rewritable disc if possible
- Removable hard drive (via SATA or USB)
- Flash-based Jump Drive via USB or Compact Flash via EIDE bus

However, I would still affix the hard drive into Dendou Oni's case permanently (although it will still be removed when needed so as a post-mortem examination to really find out what went wrong and fix it then reinstall the hard drive) – mostly for RAM vectors and files to be used in the system itself. For rapidly-changing data, the Hypercube board will have its own shared / dedicated RAM chips as well as the DDR-II modules on seed computer.

And, lastly... What about the video? Seed computer always will talk to the clusters, via bi-directional links (PCI-e 1x pipelines each boards), so at the host frontend, the seed computer will display everything through the PCI-e video card. (Since AMD Radeon HD series video cards are friendly with PowerPC CPU, I am also going to use it to display the results onto the monitor or a plasma TV.) And, it will be run in VESA or DRI mode as far as the OS in seed computer is concerned.

However, regarding the seed computer, you don't have to really be sticking with PowerPC CPUs – you can really use x86 motherboard to serve as a seed computer. However, due to the nature of the Propeller II chip itself and the sheer numbers of them, I would personally recommend the 7x86 RISC CPU such as AMD Athlon K7 (aka Athlon Classical), but 8x86 Athlon 64 would sound right due to the PCI-e required to interface to the hypercube boards. How about x86-compatible PowerPC CPUs, you say? Yes, they exist – AMD Phenom II (or III) is of the PowerPC internal. However, it's up to you, the seed computer construction is your decision – be it Pentium 4 Prescott, Athlon 64 X2, or even G5. But, if you want to make the same computer I am going to build for it, ask me, and I will think about it.

And, lastly – Chip Gracey, and Beau Schwabe, if you are reading this PDF, I would want to thank you for making the Propeller family microcontrollers. They're great and REALLY handy for everything, like the Propeller II to be used in my supercomputer, Dendou Oni. And, I would also compliment you guys at Parallax, Inc. for making such wonderful chip, since the realization of the necessity of a powerful multi-core microcontrollers for complex application, such as a smart stabilized Helium-Neon laser, car radar, photolux (lumen) meters and so much more, is fulfilled. Now, I hope its applications will keep grow (such as DIY supercomputing) – since this Propeller II in the making, as far as I can see, holds lots of promises and a bright future throughout the life expectancy of its marketing.

And, I hope my subsequent PDF reports would be as meaningful as the Dendou Oni's first boot-up! Since very few peoples are essentially interested in owning a supercomputer, let alone building one from scratch, I figured that I would glean a bit of my plan here in this PDF, to really sow the seed of DIY supercomputing. If I didn't succeed in sowing the seed, oh well, as it's still meaningful that peoples will see lot of potentiality in Propeller II as well as its predecessor, Propeller I that started the whole shows in the Propeller Forums and everywhere on the Internet, as well as in the real life.

And, for few peoples, including their projects – like Humanoido's UltraSpark 40, as well as KiloCore project, that really inspired me to build my own supercomputer – THANK YOU GUYS! Since supercomputing is really important that there are so much things I have been wondering, such as why would water expand in the coldest temperature, let alone watching the Di-Hydrogen Oxide (H2O) molecules doing its magics on the screen. Also, I am a tech freak, that I liked to tear apart the tossed-away electrical appliances to see how they work (thus learning how they really work in the process), now that I started to build electronic projects a decade ago, and that didn't do much enough to satisfy my desires and not so much challenges, that was also the reason I wanted to get into the Computer Science. Now, I am going to see that my goal and desire for this supercomputer be met!

And, why the name, "Dendou Oni?" Why not Electro-Tower or something? Aha, that was where my still-virgin Japanese vocabulary comes in (I am still learning on my own pace! ^____^) and that I have some strange likeness for the Japanese folklore about the demons (Oni or Akuma, take your pick), another reason I liked to watch Addams Family, and never got tired of watching it again and again... Dendou simply mean electric (with the electricity Kanji in the first letter – dendou has 6 meanings), and that Oni means demon (the ones with four fangs sticking out of their mouth, two for jaw and two for upper teeth, and have two horns on their forehead – they can be of green or blue skin color, although they're not limited to just being green or blue – they can be of any colors too. They're infamously being portrayed in the Japanese folklore and myth) – for some reason, that name, Dendou Oni sounded right for this supercomputer, since each Propeller II can do up to 5 billions instructions per seconds (5.2GIPS) in pure C++ codes EACH chip, and in large number, the whole mainframe can get real aggressive (faster integer computing), with 32 Propeller II (two hypercube boards) running up to 40 GIPS in SPIN language and 162 GIPS in pure machine codes, such as ASM and / or C++ programming languages - thus Dendou Oni simply means Electric Demon.

And, if anyone are wondering, what the number will the Dendou Oni supercomputer have with over a thousands of Propeller II chips? 1.2 TIPS (trillions instructions per seconds) in SPIN language, while it will run at heart-stopping speed of 5.2 TIPS while executing the pure machine codes – particularly useful for even calculating the physics of the light of any wavelengths (from FIR to hard X-ray) and what effect will the light have on the surface of any inputted material such as wood or glass.