SUN Microsystems

Graphics Processor Diagnostics Operations

Lee Boylan

August 13, 1985

Revision A

Contents

1. Preface

This document presents a technical description of the tests which run when you invoke the standalone program, "gp1.2.diag".

1.1. Purpose

This document provides technical details for the interested reader.

1.2. Audience

The Technical Publications department produced a gp1.2.diag user guide for the casual user. The reader of this document wants to know the details of the coverage and how gp1.2.diag provides that coverage.

The reader is familiar with common computer science terminology as well as the design of the Graphics Processor board.

2. Revision History

Revision A	July 12, 1985	Initial release of this document.

3. Glossary

FRU - Field Replaceable Unit.

Grappler - The nickname of the GP diagnostic program. It means the same thing as "gp1.2.diag."

Handler - The part of Grappler which runs the tests according to the user's wishes.

Prompter - The part of Grappler which prompts the user for commands and records that input.

Reporter - The part of Grappler which reports test results to the user.

4. Introduction

The program called gp1.2.diag resides in the file */stand* on those Sun workstations which have the optional Graphics Processor board and maybe its optional companion Graphics Buffer board. Candidates for the Graphics Processor options are Model 160 machines.

5. Hardware Requirements

The program expects a good CPU and enough system memory to run in. It needs some way to communicate to the user. The best way is the system's monitor and keyboard, but you can also use an attached dumb terminal. A terminal has a smaller screen than a Sun workstation, so the grappler's big main menu will partially scroll off the top.

Gp1.2.diag does not need a working color video board installed to run successfully. Under normal circumstances, the Graphics Processor receives software commands from the VME P1 bus, processes them and writes to the Sun video buffer over the same bus. It is so microprogrammable that you can tell it to write to any address on the bus, including its own shared memory. So the diagnostic never requests the GP to write to the video board.

6. General Information

Read the *Sun-2/160 Diagnostic Manual* for software version 2.1 or later. It gives a friendly description of how to load and run standalone diagnostics. The chapter called "Graphics Processor Diagnostic" talks about the gp1.2.diag user interface and gives brief descriptions of each test.

7. Grappler Description

7.1. Main()

The first function that runs in gp1.2.diag is, as with all programs written in C, called "main()." Main() sets up some interrupt handler addresses, initializes the global information structures (see below), then invokes the prompter. When the prompter returns, main() invokes the handler. When the handler returns, main() invokes the reporter. When the reporter returns, main() re-invokes the prompter as it loops again.

The effect of this is that the user sees a prompt when gp1.2.diag first starts. After the user entered commands to say how to run the tests and which tests to run, grappler runs them and, unless the user requested otherwise, gives the test results and prompts again.

7.2. Global Structures

The biggest difference between the way grappler works and most other interactive diagnostics work is nearly transparent to the user. When a grappler test finds a hardware fault, it stores the results into the *Result Structure* instead of just printing something to the user and then forgetting it. That way, another test or the grappler can refer to the result structure for decision-making.

Besides the Result Structure, the other global information resource is the *Order Structure*. So called because it stores the orders given by the user when answering the prompter, the order structure has information which the handler and the tests frequently check to decide how to run.

This design allows for artificial intelligence to customize the running and re-running of tests during run-time to zero in on failing hardware.

7.3. The Prompter

The Prompter begins by asking for a command. If user presses the RETURN key, it displays the Main Menu of all possible commands and asks again for a command.

If user enters something, prompter compares that with items in the menu and, if it finds a match, sets a control flag in the order structure according to the command entered. Then, unless the last one was an "action command," Prompter asks for another command.

An "action command" tells the Prompter not to ask for any more commands. Control then returns to main(). These are the action commands available: b, g, qpt and qbt..

7.4. The Handler

The Handler initializes some flags and then enters a loop called the "job loop."\Depending on the order structure, the job loop may only execute once.

Handler then enters the "test selection" loop. That loop goes through the entire list of available tests in the order structure and, when it finds a test marked for running, Handler initializes some more things and enters the "test loop." Depending on the order structure, i.e. what the user requested, the test loop may only execute once. If user wanted to loop the test n times, that would happen in the test loop.

The bottom of the test loop is also the bottom of the test selection loop and the bottom of the job loop. After these loops, handler returns control to main().

7.5. The Reporter

Reporter goes through a loop similar to the test selection loop in Handler. It looks at each test's history in the result structure to see if it ran. If it ran, Reporter checks to see if it found an error. If so, it prints the error message to the user.

If user had asked for summary messages only, Reporter skips printing the error message and instead gives the number of errors found during the last test pass, the last job pass and since user invoked gp1.2.diag (called the "session").

The only other thing to know about Reporter is the case of a test aborting. Most of the gp1.2.diag tests use microcode primitives to cause the gp to do something. If Grappler discovers that the hardware is unable to run that microcode, it aborts that test and sets a flag according to what went wrong. Reporter checks for abortion flags and gives appropriate messages.

8. Test Descriptions

Here are the tests, with technical discussion of how they work. Reference for all of this is the source code, written in C and GP microcode.

8.1. Slave Bus Errors

This is the first test in the group of slave tests, which is the first group.

The test code itself is simple, but how grappler behaves is not. So start by understanding how Grappler handles bus errors. First, before even calling the prompter, main() replaces the bus error interrupt handler installed by the workstation's boot PROM with its own version. This handler normally just prints a screen message ~~and~~ starts grappler over.

The slave bus error test begins by testing whether a global flag was set. Call it the "Testing SBE" flag. Since the test itself sets that flag, its already being set means that the test had started before, successfully generated an interrupt, and grappler started it again. So it resets that flag, sets the one that says it ended and returns control to the handler.

If the flag is not set when the test starts, it knows that it is starting for the first time. It sets that flag, sets its standard global flag that says it began (as do all tests) and writes to the board id register. The board id register is read-only. If you write to that address, the GP hardware is supposed to generate a bus error signal. The test then goes into a wait loop, expecting to be interrupted. After the wait loop, it puts a message into the result structure saying no expected bus error interrupt happened and sets the flag that says it found an error. It then resets the "Testing SBE" flag and returns to the handler.

Now to disuss the handler again. The first thing the bus error interrupt handler does is check the "Testing SBE" flag. If set, it quietly restarts grappler *without* putting out a screen message. That's the only difference.

8.2. Board ID

The board ID is one read-only byte at the first address of the GP. The hardware documentation on that byte says only that it is not zero, so that is all the test can do: check that it is not zero. If not in a test loop (see the Handler, above), the test will print the board id on the screen.

8.3. GP Status Register

This test does very little because the GP Status Register is read-only. It uses the write-only GP Control Register to generate a board reset, then reads the GP Status Register and expects bits 8, 9, 10 and 14 to be zeros.

8.4. GP Control Register

The Slave GP Control Register test can do little because you cannot read the register after writing to it. This test, therefore, only covers four bits because you can determine their states by reading some bits in the GP Status Register.

8.4.1. Set Controllable Status Bits To Zero

This test tries to set GP Status Register bits 8, 9, 10 and 14 to zeros by setting GP Control Register bits 15, 9, 4 and 1 to one. It reads and prints out the GP Status Register, then reads the above status register bits, one at a time, and expects zero.

8.4.2. Interrupt Enable

Now it tests control of Interrupt Enable, in Control Register bits 8 and 9. The enabled or disabled status of interrupts is available in the GP Status Register. It should be zero now. The test tries to make it a one by setting bit 8 to 1 and bit 9 to 0. If it goes to 1, tries to *toggle* it by setting both control register bits 8 and 9 to 1. The test then sets it again to zero by setting control reg bits 9 to 1 and 8 to 0.

8.4.3. Software Reset

Control Register bit 6 does a reset of much of the GP hardware. The programmer first sets it to 1 then to zero, in order to give two signal edges to the board logic. GP Status Register bit 10 reflects Control Register bit 6, so that is how the test checks results.

First, the test expects the Reset status bit to be 0. If so, it turns on the reset bit and tests for that. Then it turns it back off and tests again.

8.4.4. VP Running

GP Status Register bit 9 is 1 when the Viewing Processor's 29116 chip is not halted. When the VP is halted that bit is zero. You control the VP with GP Control Register bits 3, 4 and 5.

First, the test expects the VP to be halted. It then sets Control Register bit 3 to 1, which should start the VP. If so, it then sets it back to zero by setting Control Register bit 4.

Control Register bit 5 forces the VP's sequencer to fetch the next microinstruction from location 0, an event to be tested later.

8.4.5. PP Running

Identical to the VP Running test above, for the PP. The Control Register bits are 0, 1 and 2. The Status Register bit is 8.

8.5. VME Shared Memory

Shared memory is accessible from VME and from the Viewing Processor. This test uses VME to test the shared memory RAM.

First it does write-read tests. That means it starts at the first address and goes to the last address, writing, then reading a 16-bit pattern to each location. It repeats, using several different patterns.

Next, the test writes word patterns to every location, then reads every location, expecting the pattern written.

Next, the test writes byte patterns to every location, then reads every location, expecting the pattern written.

Next, the test does address-unique testing. It makes a unique pattern for each memory location and writes to that location. After writing to every location, it reads back and checks for expected. Then it repeats the test using the complements of the patterns.

The test generates random word (16-bit) patterns and writes them to each location of shared memory. Then it reads each location and checks for expected.

The test generates random integer (32-bit) patterns and writes them to each location of shared memory. Then it reads each location and checks for expected.

8.6. Microstore

The VME interface to microstore is two 16-bit registers, called Microstore Address and Microstore Data. You write the desired address into the address register (AR), then write or read the data register (DR) for four times to access a 56-bit microinstruction. During those four visits to the DR, the AR does not change, because you are always accessing the same microinstruction. However, the GP Status Register bits 12 and 13 do change from 00 to 11 to indicate which quarter-microword the DR is touching.

The test checks out the AR by writing and reading various patterns to it.

Next the test writes zeros to the AR, then does writes and reads to the DR, testing GP Status Register bits 12 and 13 and the AR for proper incrementing.

Next the test uses AR and DR to test microstore with constant, 56-bit patterns.

Next, the test checks microstore with address-unique data.

Next, the test checks microstore with random data.

Finally, the test writes zeros to every location of microstore, just to prevent the microprocessors from accidentally executing garbage.

8.7. VP Status Register

Beginning with the VP status register, most of the tests use microcode to force conditions and pass data to gp1.2.diag. The VP and PP Status Register tests are the simplest of this kind of test. It downloads a primitive into location 0 of microstore. That primitive moves an immediate 29116 value to the VP Status Register. It repeatedly runs that primitive, each time modifying the immediate value before starting it. After each time, the test reads the VP status register to see if the immediate value got there.

This test covers the path between microstore and the 29116 and the bus between the VP 29116 and the VP status register. *If either of these is faulty, virtually none of the rest of gp1.2.diag will work,* because grappler uses the VP (and PP) Status Register to detect the ending (called the end semaphore) of every microcode primitive.

8.8. VP Interprocessor Flags Bit 8

The interprocessor flags (ipflags) are some semaphore bits which are readable by the VP and PP. Some are read-only hardware indicators and some are programmable, for signaling between the VP and PP.

Ipflag bit 8 is the only really crucial one. The one readable by VP is permanently hardwired to be a zero, whereas the one readable by PP is permanently hardwired to be a one. Because microcode for each processor resides in the same place, the programmer must have code to read ipflag bit 8 to determine which processor it is running on. Grappler does not use this feature because grappler only runs one processor at a time. Nevertheless, this is worth testing early, so the technician can fix it. *If this hardware is bad, none of the production VP microcode will run.*

The test downloads microcode to copy the ipflag register to shared memory. It then reads the shared memory and tests bit 8 for zero. It puts a semaphore into the VP Status Register to signal grappler. If 1, it produces the error message, "VP's interprocessor flags bit 8 is a 1, but it should always be a 0."

8.9. VP General Field As Branch Address

16 bits of the VP (and PP) microinstruction are called the General Field, so called because it is used several ways, depending on other parts of the microinstruction. One such use is a branch address.

The test works like this: At almost every location in mircostore, it writes code to turn on an error signal in the VP Status Register. At the one target address, it writes code to turn on the "passed" signal in the VP Status Register. It writes an unconditional branch to general field, in which it has the target address.

Now the test repeats the above for target addresses from 4 to 0x800.

8.10. VP General Field As Immediate Value

Another use of the microcode general field is an immediate value source for a bus data transfer. The test uses the general field to move zero to shared memory and then reads shared memory via VME for the expected zero value. Next it does the same thing with the value 0xffff.

Now the test repeats the above with every possible pattern, from 0 to 0xffff.

8.11. VP N Register

For certain bit-oriented instructions, the 29116 has a 4-bit field called the N field. The programmer uses it to specify which of 16 bit positions to operate on. The N register provides a runtime substitute for that field.

The test moves an immediate value from the 29116 to the n register, then uses it to set the n-th bit of the 29116 accumulator. It moves that accumulator value to shared memory, where grappler compares it with expected.

The test does this for all possible values of n: 0 to 15.

8.12. VP N Assembler Constant

Because the N register test described above ~~does~~ not use the chips that carry the assembled N field value to the 29116, this test exercises that hardware.

The test uses the 29116 mov2n instruction with immediate n field values of zero, walking 1 patterns and 0xf to move values into successive locations of shared memory. It then reads shared memory for the expected patterns.

8.13. VP Branch Register

The branch register holds the possible next microinstruction location, depending on the microcode and runtime conditions.

This test uses the same testing algorithms as in the VP General Field As Immediate Value test, above, except using the VP Branch Register instead of the immediate field.

8.14. VP Bank Select

The 2910A micro-sequencer is capable of addressing 4K locations. That is not enough, so the GP has two banks of 4K microwords available. Hardware detects the 2910A JMAP instruction to cause the 2910A to fetch microinstructions from the second bank. JMAP is the only instruction that enables the switching from bank 0 to bank 1.

This test follows the same testing philosophy as Branch Register test, above. It puts the target instruction, which writes "passed" to the VP Status Register, into bank 1. Other locations write "failed" to the VP Status Register. Then it writes a JMAP to the target address in microstore and starts execution. It uses (almost) every bank 1 location as a target address.

Next, the test uses the same system to test JMAP for selecting bank 0 from bank 1.

8.15. VP Shared Memory

Because Grappler tests the Shared Memory RAM in the Slave Shared Memory test, this test focuses on the VP shared memory pointer and the VP bus data path to shared memory. First, it writes walking-1 patterns to the shared memory pointer and the same pattern to shared memory. Then grappler reads that via VME for correct expected value.

Next, the test does a complete address-unique test from the VP, writing patterns from 0 to the last possible address, both to the VP shared memory pointer and to shared memory. Then it reads shared memory via VME and compares with expected.

The next thing is to check the data path *from* shared memory to the VP bus. The test now has shared memory filled with address-unique patterns. It reads each location, starting from the second one (address 1), and writes the contents to the previous location. When finished, it reads shared memory via VME and compares with expected.

Please note that the last data path test will fail if a fault already caused bad data to be in shared memory before it started. So if you had requested 'continue on error' from the prompter (coe), this test will give an erroneous failing address of one less than the one reported before.

8.16. VP Two Address Operation

The 29116 takes a complete clock cycle to execute one instruction. Some data movement, however, takes place in the first half of the cycle. The GP hardware allows the programmer to take advantage of that fact to put new data on a field of the 29116 instruction inputs during the second half of the cycle, thus allowing for two-register programming of an otherwise one-register instruction.

The test checks this multiplexing hardware this way: first, it loads every 29116 register with zero. Then it puts a 1 into the 29116 accumulator. Now it uses two address operation to add the contents of r[0] to the accumulator and put the results into r[1], add the contents of r[1] to the accumulator and put the results into r[2], etc. Then it moves the contents of every register into shared memory for comparison with actual.

8.17. VP PROM

The hardware provides the vp reciprocal prom, a look-up table for floating point operation. The test dumps the PROM contents to shared memory and does checksum on every location except the last location. The PROM contains the expected checksum in that location. If the checksum fails, the test then uses some diagnostic data in the prom to test the PROM to bus data path. The diagnostic data is walking 1 and walking 0 patterns.

8.18. VP Continue Not At 0

The GP Control Register provides three flags to control the Viewing Processor. They are: VP Halt, VP Continue and Reset PC to 0 When Continue. Because grappler ordinarily uses Reset PC to 0 When Continue whenever it uses VP Continue, this test checks the control hardware when only using VP Continue.

It loads and starts a primitive which infinitely loops in the middle. It then sends VP Halt and modifies that primitive to remove the infinite loop, but right above that loop it puts a new infinite loop which sends a failure semaphore to the VP Status Register. Now it issues VP Continue. If the VP incorrectly started at location 0, it will write the failure semaphore instead of the pass semaphore, which is at the bottom of the original primitive.

8.19. VP Interprocessor Flags ~~and~~ FIFO

This is only a partial test of the interprocessor flags and the FIFO. You need the PP to test ipflags and fifo completely, but right now that is not known to be good. See below for more FIFO and ipflags testing.

As discussed earlier, some of the ipflags are read-only, the others are write-only. This tests two read-only ipflag bits, which indicate FIFO direction (ipflag bit 9) and whether fifo is empty (bit 10). Recall that grappler tested ipflag bit 8 earlier.

First, the test issues a Control Register reset, which should initialize the fifo direction to vp-to-pp and should cause the fifo to be empty. It tests those flags for expected by copying the ipflag to shared memory.

Next, the test writes two words to the fifo from the 29116 and checks ipflag bit 10 for no longer saying the fifo is empty.

Now the test writes 01 to ipflag write-only bits 14 and 15 to set fifo direction pp-to-vp and checks ipflag bit 9 for showing that direction. It reads the two words written above and checks that the second one is as written.

Now the test writes 11 into ipflags bits 14 and 15 to *toggle* the fifo direction to vp-to-pp. It reads ipflag bit 9 for confirmation.

Now the test is satisfied that the VP can write to the fifo, read it and change its direction. So next it writes and reads these patterns to check the data path between fifo and the VP bus: ffff, 0000, 5555 and aaaa.

Next the test writes patterns to the fifo until it is full and verifies that after writing the expected number of patterns, the 2910A received the fifo full condition code. The "expected" number of patterns in this case is one more than the fifo size because hardware pulls the first word out to a buffer.

The final VP fifo test involves moving data from the VP PROM to the fifo. The purpose of the test is to cover the bus control pattern 10 0000, because it happens to be a walking 1 pattern not used anywhere else in grappler.

The test copies the VP PROM into the fifo, and thence to shared memory for a checksum test. Because the fifo is only 512 words long and the VP PROM is 16K words, it takes 32 moves of PROM data through the fifo to get it all into shared memory. See the VP PROM test for a description of the checksum test.

8.20. Floating Point Registers

The two Weitek floating-point chips get their operands from "registers" which are 2K by 32 bits in size. Call them the A and B registers, or the FP registers. You can read either register to the VP bus by setting the A or B pointer and selecting the A or B register as a bus source. You can only write to both of them at the same time, at the same location in each. You set a separate pointer for writing, called the Destination pointer.

The test writes 4k of constant pattern to shared memory, starting at shared memory address 0. Then it causes the VP to fill the FP registers with the pattern from shared memory. Next, it reads the A register into the next 4K locations of shared memory and the B register into the 4K locations after that. It then compares the A and B copies with the original. It uses these patterns: 0000, ffff, 5555 and aaaa.

Now the test has verified the data paths between the FP registers and shared memory. It next uses the same scheme to load, read and compare address-unique data in the FP registers, to test the Destination, A and B pointers.

The last thing this test does is repeat the address-unique test, this time using only multiples of 2 for the pointers. This satisfies a request made to us for a walking 1's test for the pointers.

8.21. Floating Point Functions, Flowthrough Mode

Two Weitek processor chips support two modes: flowthrough and pipeline. Each one (ALU and multiplier) processes every floating point operation code with the same operands from the A and B registers (usually; operands may also come from the floating point outputs). The program chooses which chip's result it wants after the operation.

Because the test will rely heavily on the ALU operations FLOAT and FIX, it begins by testing them with walking 1 patterns.

Next, the test uses every op code in each chip, using walking 1 patterns in the A and B registers (operands). It selects an operation (op code 4 through 10), then it loads walking 1 pattern operands to the A and B registers. It has the ALU convert that pattern to a floating point number (the FLOAT operation). It performs the selected operation, converts the result to integer format (the FIX operation) and writes it to shared memory. It repeats this process until it has used every walking 1 operand for each operation.

8.22. Floating Point Result Registers and Pipeline Mode

Although the Weitek chips require 12 cycles to complete an operation, they allow pipelining, i.e. starting an instruction two cycles after starting the last one, then receiving results every two cycles.

The GP hardware allows the saving of the output of a floating point operation for use as an operand of a subsequent operation. Call that saving place the Result Register.

This test executes the arithmetic operation, $(A+B) * B$. It allows the use of the result register and pipelining.

The test selects a walking 1 pattern for A and B (different for each one), then uses the FLOAT operation (in flowthrough mode) to convert them to floating point numbers. Then it uses pipeline mode to perform the addition and multiplication. Finally, the test FIXes the results and writes them to shared memory for checking.

8.23. Floating Point Matrix Multiplication

This time the test uses pipeline mode to do inner matrix multiplication. This is a more rigorous use of the pipeline mode and the Weitek processors. It also uses the result register.

It takes these entries of a 1 by 4 matrix and FLOATs them: a1 = 1, a2 = 2, a3 = 3, a4 = 4. It takes these entries of a 4 by 1 matrix and FLOATs them: b1 = 4, b2 = 3, b3 = 2, b4 = 1. It then moves the floating point values to the A and B registers and, using pipelining and the result register, performs the arithmetic operation. The result of that goes to shared memory. Then the test does a FIX of the result and puts that into shared memory where the test compares it with expected. The 1 by 1 result of this operation should be 20.

8.24. Floating Point Status Register

The Floating Point Status Register (FPSR) has bits giving information about the most recent floating point operation and bits which accrue that information until microcode chooses the FPSR as a bus source (reads it).

The test writes invalid (according to the IEEE floating point spec) inputs to the operand (A and B) registers and attempts to add them. It then reads the FPSR and expects the accrued invalid input flag and the invalid input flag.

The test writes inexact (according to the IEEE floating point spec) inputs to the operand registers and attempts to multiply them. Their product should set the overflow result flag. It then reads the FPSR and expects the accrued inexact input flag, the accrued overflow result flag, the inexact input flag and the overflow result flag.

Next, the test provides operands whose product should be a negative underflow result and checks for FPSR accrued sign, accrued underflow result, sign and underflow bits.

8.25. VP Condition Code Select

Because the 2910A has only one input for a condition code (CC), the GP hardware provides a programmable selector to choose which condition code will go to that pin. The CC selector test is almost all microcode. It selects a condition code and asserts it. The test then makes sure that the 2910A correctly branches according to that condition. Then it asserts at least one other condition code but not the one under test and expects the 2910A not to branch as if the target condition code were asserted.

8.25.1. VP Negative Status

First, the test selects 29116 Negative Status and asserts it. It signals an error if that CC failed to assert. Then the test asserts carry, fifo not full and fifo not empty CC's but *not* negative status. The microcode will signal an error if it detects that CC.

8.25.2. VP Status Negation

The programmer may select the logical opposite of the 29116 status for a branch test. For example, instead of branch on zero, you can program to branch on not zero. That logical negation is another function of the Condition Code Select hardware.

The test subtracts 1 from zero and branches to an error signal on not negative.

8.25.3. VP Carry Status

First, the test selects 29116 Carry Status and asserts it. It signals an error if that CC failed to assert. Then the test asserts negative, fifo not full and fifo not empty CC's but *not* carry status. The microcode will signal an error if it detects that CC.

8.25.4. VP Zero Status

First, the test selects 29116 Zero Status and asserts it. It signals an error if that CC failed to assert. Then the test asserts carry, fifo not full and fifo not empty CC's but *not* zero status. The microcode will signal an error if it detects that CC.

8.25.5. VP Overflow Status

First, the test selects 29116 Overflow Status and asserts it. It signals an error if that CC failed to assert. Then the test asserts carry, fifo not full and fifo not empty CC's but *not* overflow status. The microcode will signal an error if it detects that CC.

8.25.6. 29116 Status Select

In order to use a 29116 status result, the programmer must select it in microcode. This test makes sure that the 29116 status only goes to the 2910A when selected.

The test moves a zero in the 29116 with status enabled. It then signals an error if the 2910A did not detect a zero status. Next the test moves a zero in the 29116 with status enabled. Then it moves a one with status not enabled. It then signals an error if the 2910A did not detect a zero status.

8.25.7. FIFO Not Full Condition

The test resets the hardware, which empties the FIFO. Then microcode branches to an error signal if the FIFO Not Full CC wasn't asserted. Next, the test writes to the FIFO until it is full and causes microcode to branch to an error signal if the 2910A detected the FIFO Not Full CC.

8.25.8. FIFO Not Empty Condition

The test resets the hardware, which empties the FIFO. Then microcode branches to an error signal if the FIFO Not Empty CC was asserted. Next, the test writes a word to the FIFO and causes microcode to branch to an error signal if the 2910A failed to detect the FIFO Not Empty CC.

8.25.9. Floating Point Negative Condition

The test causes the Weitek ALU to subtract for a negative result. The microcode generates an error signal unless the 2910A detects the Floating Point Negative CC. Next, the test adds two positive numbers in the FP ALU and the microcode branches to an error signal if the 2910A detects the Floating Point Negative CC.

8.26. PP Status Register

This test is identical to the VP Status Register test, above.

8.27. PP Interprocessor Flags Bit 8

This test is identical to the VP Interprocessor Flags Bit 8 test, above, except that the PP ipflag bit 8 is permanently hardwired to be a zero.

8.28. FIFO

This test assumes that most VP hardware works. The first thing it does is test the PP's ipflag register bits 9 and 10 which, just like those in the VP, say which direction the FIFO is in and whether it is empty.

First, the test issues a Control Register reset, which should initialize the fifo direction to vp-to-pp. It reads ipflag register #1 into the 29116 and then writes the high byte back to ipflag register #2 for the VP to pass to shared memory. From there, Grappler expects bit 9 to say fifo direction is vp-to-pp.

Next, grappler tells the VP to change the fifo direction. It repeats the above procedure to read the ipflags register #1 to see if bit 9 shows the direction now is pp-to-vp.

Now the test issues another reset. It reads, using the above technique, ipflag bit 10 to see that it says fifo is empty.

Next, the test uses the VP to move patterns from the 29116 to the fifo. Then it checks the PP's ipflag register bit 10 to see that it does NOT say fifo is empty. Then the VP reads the fifo to shared memory for grappler to compare with expected. That tested the data paths from VP to fifo and back.

Now the test resets the hardware and has the VP move patterns to the fifo. Then it has the PP read those patterns into the PP's 29116 internal registers. Then it resets the hardware, changes fifo direction and moves those patterns from the 29116 back to the fifo. Then the VP reads the fifo to shared memory for comparison by the grappler. That tested the data paths from fifo to PP and back.

Next the test uses the PP to write words to the FIFO until its 2910A receives the fifo full condition code. The test then counts the number of writes that that took and compares with the size of FIFO. Next the test has the VP to read the fifo to shared memory and from there it compares with expected. It repeats this until the VP 2010A receives the fifo empty condition code. It compares the number of reads with fifo size.

8.29. Interprocessor Flags

Although a part of the PP tests, this covers the VP's writeable ipflags too. First, the test uses the PP to write a byte to ipflag register 2 and the VP to read that byte to shared memory where it compares actual with expected. It repeats this for all possible patterns.

Then it has the VP write byte patterns to ipflags 1 and has the PP read ipflags 1 to the 29116 and then back to ipflags 2. Now it has the VP read ipflags 2, which was tested to be good above, into shared memory. Then grappler compares with expected. It repeats for all possible patterns.

8.30. PP General Field As Branch Address

This test is identical to the VP General Field As Branch Address test, above.

8.31. PP General Field As Immediate Value

This test is identical to the VP General Field As Immediate Value test, above. The results go to the FIFO, where the VP copies them to shared memory.

8.32. PP N Register

This test is identical to the VP N Register test, above. The result goes to the FIFO, where the VP copies it to shared memory.

8.33. PP N Assembler Constant

This test is identical to the VP version, described above. The result goes to the FIFO, where the VP copies it to shared memory.

8.34. PP Branch Register

This test is identical to the VP version, described above.

8.35. PP Bank Select

This test is identical to the VP version, described above.

8.36. PP Two Address Operation

This test is identical to the VP version, described above. It uses the VME bus to move the contents of the registers into shared memory for comparison with actual.

8.37. PP Scratchpad Memory

The PP has 4K of fast memory on its bus, called scratchpad memory. First, this test fills it with a constant pattern. Then it starts at the first location and moves that value to fifo. Then VP moves that value to shared memory for comparison with actual. It repeats for every location. It uses these patterns: 0, 0xffff, 0x5555 and 0xaaaa.

Next, the test uses address-unique patterns to cover the data path from scratchpad ram to fifo.

8.38. PP Continue Not At 0

This test is identical to the VP version, described above.

8.39. PP Condition Code Select

This duplicates the VP Condition Code Select test, except it does not have the Floating Point Negative CC but it does have VME Not Ready.

First the test does not cause the VME bus to be busy. It sets up a microcode counter in the 2910A and keeps trying to access the VME bus. If VME stays busy for the entire loop (0x1000 times), it assumes something is wrong with the VME busy CC and so informs the user. Next, the test does a VME write to shared memory and expects, in the very next micro-cycle, to detect the VME Busy CC. If not, it gives an error signal.

8.40. PP Three-Way Branch

Please read a description of 3-way branching in the GP Hardware Reference Manual or the GP Diagnostic Design Description. The hardware allows for testing the usual condition codes plus vme bus busy in the same cycle.

First, the test creates the conditions of vme bus not busy with zero condition code. The microcode will show the correct semaphore in the PP status register if ended correctly and grappler will know that part passed. The possibilities are: correctly detected the zero CC, failed to detect the zero CC, falsly detected VME busy.

For the next case, the test does NOT set the zero condition or VME bus busy. The possibilities are: falsly detect zero CC, falsly detect vme busy, correctly detect neither condition.

Next, the test does make the VME bus busy and the CC not true. The possibilities are: failed to detect VME busy, falsly detect CC or correctly detect VME busy.

8.41. PP Interrupt Generation

The Graphics Processor is capable of generating a bus interrupt, detectable by the workstation CPU as a user interrupt. Specifically, the microcode generates that interrupt, but the workstation, via the GP Control Register, can suppress that interrupt signal from actually going over the VME bus. In that case, the GP is supposed to set a bit in the GP Status Register saying an interrupt is "pending." The GP hardware has a copy of that bit in the VME Status Register. This test checks both of those bits.

First, the test sets the GP Control Register Clear Interrupts bit and reads the GP Status Register and the VME Status Register to verify that their Rupts Pending bits are off.

Next, it disables interrupts with the GP Control Register and causes PP microcode to (attempt to) generate an interrupt. If the GP incorrectly generates an interrupt, the grappler's unexpected interrupt handler will so report. Now the test examines both the GP Status Register and the VME Status Register and, if either fails to show a "pending" interrupt, reports to the user. (Pending is in quotes because that interrupt will never happen, even if the GP Control Register later enables interrupts. See GP literature.)

Finally, the test sets the Clear Interrupts bit of the GP Status Register and checks again that it cleared both Interrupt Pending bits.

8.42. VME Bus Master Write

As a bus master, the GP (specifically the PP) can request the bus and write to or read from any VME addressable location. This test tries to write to GP Shared Memory over the VME bus.

First, the test has the PP request VME bus and write wallking 1 word patterns over it to shared memory. Then it makes a comparison between actual and expected data. Next it uses byte mode to write walking 1's to odd bytes and walking zeros to even bytes in shared memory and compares for expected.

Finally, it tests the VME Low Address Register by putting walking 1's into it and writing 0xffff to shared memory at those locations. It compares with expected.

8.43. VME Bus Master Read

This is the other half of VME Bus Master testing. The PP is supposed to be able to read any VME addressable location.

First, the test initializes shared memory with walking one data for the PP to read. Then it sets up a loop, using shared memory locations as the loop counter. Each time through the loop, the test reads shared memory from the VME bus to the FIFO. Then it goes from the fifo to shared memory at location zero, where the test compares with expected. The loop continues for all walking one patterns previously written to shared memory.

Next, the test uses byte locations of shared memory to do the same thing. This time, the big loop includes writing the byte to shared memory at location zero from the GP. Then the test moves that pattern via VME bus to the fifo, then back to shared memory location 1 for comparison with expected.

Because part of the GP hardware only handles data of word length, the byte test must do some fancy stepping with the data. For example, byte 0 in shared memory is the high half of word zero. So the test writes 0x100 as the first walking 1 pattern. When the test moves the byte read by the PP into the fifo, that same byte is now the low half of a word there. So the test must shift and mask to compare actual with expected. For details of this see the code.

The test repeats the byte test above with walking zero patterns.

8.44. VME Status Register

The VME Status Register has several bits which give information to the PP microcode concerning the current status of the bus. Grappler diagnostics cover two of those bits, 4 and 15. Bit 15 is the Interrupts Pending bit, tested above. This test covers bit 4, Illegal Access. The diagnostic cannot test the other bits, because it is unable to generate the necessary bus conditions.

First, the test generates a board reset, to clear the fifo. Then it sets fifo direction to pp-to-vp and sets up the vme control register for a bus word write to shared memory on a byte (odd) boundary. It has microcode write to vme and then read the VME Status Register to fifo. Then the VP moves that word to shared memory for comparison of the illegal access bit with expected.

8.45. VME Interrupter

This test uses the VME bus to generate interrupts to the workstation. Although the designers only intend for the GP to generate one interrupt vector, it is capable of generating all possible ones according to the Motorola 68010 specifications.

The test creates a big loop, with interrupt vectors from 64 to 255 as the counter. For each vector not under test, it sets up an unexpected interrupt handler. For the vector under test, it sets up an expected interrupt handler. It then enables interrupts via the GP Control Register and writes the vector to the VME Interrupt Register.

The expected interrupt handler merely causes the test to end the loop. The unexpected interrupt handler generates an error message giving the expected interrupt vector and saying that it received a different one.

8.46. Graphics Buffer DRAM Constant Patterns

The Graphics Buffer board is mostly dynamic RAM. The first thing to do is test it with constant patterns. The test must use the GP's shared memory to move data from the GB to the workstation for checking. For microcoding convenience, it moves data in 4K word chunks.

The test moves the target address to the GB Hi and Low Address Pointers and the constant pattern as a 29116 immediate value to the GB Write Data Register. Then it increments the pointers until it has filled 4K words.

Next the test moves the 4K words to shared memory via VME and compares with expected. It repeats this until it has tested all 2M-bytes of DRAM. The first patterns are 0, 0xffff, 0x5555 and 0xaaaa.

A second test does these constant patterns in the same way: 0x6666, 0x9999, 0x1818, 0xe7e7, 0x7171, 0x8e8e, 0xc3c3 and 0x3c3c.

8.47. Address Test 1, Address-Unique Data

The test moves unique 29116 immediate values to every location of GB DRAM. Then it goes back to the beginning of DRAM and moves it 4K words at a time to shared memory and compares with expected.

8.48. Address Test 2, Galpat

The test is a modified galpat test. First, it writes 0 to every location. Then it writes 0xffff into every third location of GB DRAM starting at location 0 and reads every location, comparing with expected. Then it writes the same pattern to every third location starting at location 1 and reads every location, comparing with expected. Then it writes the same pattern to every third location starting at location 2 and reads every location, comparing with expected.

Next, the test repeats the above, using 0 for the new pattern.

8.49. Address Test 3, Surround Disturb

This is a modified surround disturb test. First, it writes 0 to every location. Then it writes 0xffff to location 0 256 times and checks all locations. Next it writes 0 to that location (so now all are zero again) and writes 0xffff to location 0xfffff 256 times and checks all locations. Next it writes 0 to that location (so now all are zero again) and writes 0xffff to location 0x55555 256 times and checks all locations. It repeats this kind of testing for these addresses: 0xaaaaa, 0x66666, 0x99999, 0x18181, 0xe7e7e, 0x71717, 0x8e8e8, 0xc3c3c and 0x3c3c3. If one of the tests fails and testing is supposed to continue, it will write zeros to every location instead of just the one with the last pattern.

8.50. GP DRAM Refresh

The test writes 0 to even address and 0xffff to odd addresses of DRAM. Then it waits for enough cycles to allow non refreshed memory to lose data and checks every location for expected.

Now the test repeats by writing 0xffff to even addresses and 0 to odd ones.

8.51. GB DRAM Fill Mode

Fill mode allows the micro-programmer to write to four word locations at once. The test writes 0 to the first 4k word locations using fill mode and then moves the data via VME to shared memory for comparison with actual. Then it repeats with 0xffff, 0x5555 and 0xaaaa.

8.52. GB RMW Mode

The GB hardware allows Read Modify Write (RMW) mode, which means the ability to set a location in the GB High and Low Address Registers, read that location and write to it in the minimum number of cycles.

The test, using RMW mode, writes 0 to the first 4K-words of GB DRAM. Then, using RMW mode, reads it to shared memory via VME and compares with expected. It repeats with 0xffff, 0x5555 and 0xaaaa.

8.53. Integer Multiplier

The GB contains an AM29517, a special-purpose chip for integer multiplication.

First, the test multiplies 0 times 0 and checks for the expected result. Then it tests for all possible operands from 1 to 0x4000 and compares the result with the same product from the workstation's CPU.

Next, the test uses walking 1 values from 1 to 0x4000 in one operand and minus 1 in the other. It compares the result with expected. It then repeats this test, reversing the roles of the two operands. Then it puts minus 1 into both operands and expects 1 as the result. Next it puts minus 1 into one operand and 0x8000 in the other and compares the result with expected, then repeats with the operands switched.

Finally, the test puts 0xe000 into one operand and 0xc000 into the other one. It compares with expected as generated by the workstation CPU.

8.54. PP PROM

This test is identical to the VP PROM test, described earlier. It uses VME bus to move result information to shared memory.

9. Scope Loops

Grappler provides scope loops as a separate item under the main menu. The user enters "sl" after the command prompt. Menus provide for scope loops in shared memory; microstore; microstore address register; VP, PP and GP source-dest data movement and Graphics Buffer DRAM.

10. Error Handling

In virtually every case, Grappler error messages name the failing test, describe what they tried to do (if they can do that tersely) and give actual and expected results.

Because the tests assume that hardware tested previously must be good, they do not list all the possible failures outside of the area under test which might cause that failure to occur.

10.1. Field Replaceable Units

The two FRU's are the Graphics Processor board and the Graphics Buffer board.

11. Recommended Test Procedure

At the command prompt, enter "pm" for progress messages and "gp" for all graphics processor tests. If the workstation has a GB, enter "gb." Then enter "b."

12. Summary

Neither the Graphics Processor nor the Graphics Processor diagnostics package is simple. Gp1.2.diag starts out by trying to talk to the addressable slave locations on the board, then it uses those to download and control microcode to test the rest of it.

The gp1.2.diag structure provides for the user to select a range of running modes and test subsets. It also provides for maintainability when users require changes.

13. References

"Graphics Processor Hardware Reference Manual" by John Fetter, Sun Microsystems.

"Graphics Processor Design Overview" by John Fetter and Jerry Evans, Sun Microsystems.

"Graphics Processor Schematics" by John Fetter and Serdar Ergene, Sun Microsystems.

Notes on AMD2910 and AMD29116 processors in *Bipolar Microprocessor Logic and Interface* by Advanced Micro Devices, Inc.

Graphics Processor Diagnostics Source Code by Lee Boylan and Gus Wang, Sun Microsystems.