EXTERNAL SPECIFICATION
(User Perspective)

TITLE : **SUN-2 Diagnostic Prom (primediag) External Specification**

AUTHOR : Kevin Sheehan

REPORT NO. : 750-1006-01

REVISION NO. :

DATE :

STATUS : Release to Document Control

APPROVALS : DATE ----------

Kevin Sheehan ----------

Document Review Form

Please make note and initial on this page all corrections and/or proposed amendments by page number and/or section number.

Recommendations, Differences, Construction Errors, and comments:

Typographical Errors:

Attach additional sheet(s) as needed.

SUN-2 Diagnostic Prom (primediag) EXTERNAL SPECIFICATION

This document describes the operation and use of the CPU diagnostic proms generally referred to as SUN-2 Architecture Manual SUN-2 CPU Schematics Doc. No. 501-1051-02 SUN-2 CPU Roadmap Doc. No. 270-1051-03 SUN-2 Low Power Memory Board Manual PN 800-1026-?? Manufacturer's Data Sheets for specific devices. The term *leds* refers to the diagnostic LEDs located on the top side of the CPU board. The term *terminal* refers to the RS-232 device connected to port A of the CPU serial I/O connector (J1). MMU stands for Memory Management Unit, which includes the context registers, segment maps, and page maps. The C notation for hexadecimal numbers is used here. A number preceded by 0x (as in 0xabcd) is in hexadecimal.

primediag is used to verify that a CPU board is functioning correctly. It does this by testing the MMU functions, memory function and interface, and the various on-board devices. *primediag* executes each test, and if errors are found it generally goes into a scopeloop for debugging purposes. It maintains a global error flag, and pass counter for extended operation. The following is required to run *primediag* correctly: SUN-2 CPU (rev -01 or later) with all devices in. SUN Low power memory board. At least 128KB of some form of Multibus memory. (possibly another SUN-2 CPU doing DVMA) (Optional) RS-232 device (terminal) to observe test messages. The CPU and Low power memory board need to have a common P2 connector on the Multibus. The Multibus memory cannot have a P2 connector in common with the CPU and memory. The terminal should be connected to port A of the CPU. Error handling is discussed in the descriptions of the tests below. *primediag* takes about 90 seconds to run each pass successfully, most of the time is spent in T48, the memory checkertest.

The user will see test numbers and diagnostic information on the terminal. The leds will advance with the test numbers as *primediag* proceeds. Required input and error handling is described in each of the operation sections below. Assuming that the *primediag* proms are in the CPU, and the required configuration is present, the leds will display a walking off led to indicate that the led register is OK when you turn power on. If the leds stay lit, either the proms are in wrong, or there are other problems too serious to even allow the diagnostic to start. Otherwise, the tests proceed as described below. As each test starts, the test number is put into the leds. After T21, output is also sent to the terminal. If the leds stay at a certain number, and no output appears at the terminal, that test has failed, and a scopeloop is in operation. This test writes the values 7..0 to the supervisor context register, and reads them back. If a value does not read back correctly, we continue to write that value, read it back, and compare until it reads correctly. When all 8 values have been written and read correctly, we go on to.. This test is identical to T11, but does the user context register instead. This test writes the value 7 to the supervisor context register, then does the values 7..0 to the user context register, comparing both to expected values as it does so. If any read back incorrectly, it continues to write and compare both context registers until they read OK. Then we write the value 6 to the supervisor context register, and do the values 7..0 to the user context register. And so on. Assuming all read back correctly, we go on. This test is the same as T13, but the user context register is set to 7, then the supervisor is written with 7..0, and so on. Basically, the supervisor context register is now the most frequently updated. At this point, if things are working well enough to access the UARTs, the revision/identification string is printed on the terminal, and the test numbers start appearing. Each of the tests has a number, and a mnemonic printed out. (Like T21(s0), see Appendix) If we cannot access the UART, the leds will still advance, and the tests will continue. If errors occur, a string with the offending item under test, address information, and observed and expected values is printed. We go to each of the possible contexts, write constant data in each of the segment map entries, and the read them back in the same order. If any of the values does not read back correctly, we print out the address, the observed and expected values, and the context they were found in. We then go on to see if there are any more. The expected values (as indicated by the mnemonics) are 0 for T21, 0xff for T22, 0xaa for T23, and 0x55 for T24. In this test, we write unique values to each of the segment map entries in each context and read them back. Error messages are the same as for the constant tests. If values appear the same on a binary increment, it indicates non-unique decoding of segment map entries. T26 writes the value 0xaa and the inverse into each context segment map using the checkertest algorithm. T27 writes 0x55 using the same algorithm.

Both tests print errors like before. These tests use the first half of context 0 to reference the page map entries, therefore errors in these segment map entries will be reflected here also. These tests write 24 bits of constant data into each page map entry, and compare them to expected values. If errors occur, the offending address, expected and observed values are printed, and the test continues. Expected values are 0 for T31, 0xfff00fff for T32, 0xaaa00aaa for T33, and 0x55500555 for T34. The middle 8 bits do not exist in a page map entry. In this test, we write unique values to each of the page map entries and read them back. Error messages are the same as for the constant tests. If values appear the same on a binary increment, it indicates non-unique decoding of page map entries. T36 writes the value 0xaaa00aaa and the inverse into the page map using the checkertest algorithm. T27 writes 0x55500555 using the same algorithm. Both tests print errors like before. At this point the diagnostic does some setting up of segment and page maps. We assume no more than 4 MB of memory are to be tested here. In order to size memory, we look at the end of each 1MB chunk of memory, and see if it responds with the proper pattern. If memory has bad data lines, no memory will be found. If no memory is found, (for whatever reason) we do write/reads at location 0 for debugging purposes. Once memory is found, if errors occur, the type of test is printed, the physical address of the offense, and the expected and observed values. We then do 64K writes to that location with the expected value, followed by 64K reads. A dot is printed indicating another loop completed. In looping mode, if you type a 'b', *primediag* will start over from the beginning. If you type a 's', *primediag* will skip all further memory pattern tests. If you type any other character, *primediag* will continue with the next test. If nothing is typed, looping continues. If both halves of a pattern have the same bits wrong, it usually indicates a hard failure of either logic or data paths. If one half has incorrect data, and the other does not, it is either a single bit failure in RAM, or a transient (flaky) response. Going on to the next test will confirm or deny this. Parity is turned off on these tests. These tests write a constant pattern into memory as a whole, then read back and compare against the expected value. The expected values as indicated in the mnemonic are: T41(m0) - 0x00000000 T42(mf) - 0xffffffff T43(ma) - 0xaaaaaaaa T44(m5) - 0x55555555 T45(mc) - 0xcccccccc T46(m3) - 0x33333333 Each location in memory is written with a unique pattern (the address), and later read back for comparison. If errors occur that are not bit errors, like 0x00f00000 appearing at 0x00c00000, then memory locations are being decoded incorrectly. Here, memory is being written with the pattern and the inverse according to the checkertest algorithm. This test takes some time, as it does log2(size of memory) -2 passes over all of memory. This translates to about 45 seconds for 1 MB of memory. Be patient, this is a very exhaustive test. Since this is supposed to be Randomly Addressable Memory, we address it randomly for a while. Each location in memory is written with its address in a random sequence. It is then read back and compared in the same order. These tests insure that parity circuitry catches errors when parity is wrong, does not catch errors when ok, and then that parity comes out ok over all of memory. We then do a quick test of ability to address real Multibus memory. This test enables bus errors, then writes even parity, and checks for even parity to have been generated. It then disables parity generation, writes out data in odd parity and checks for bus errors. If all this went well, it then enables parity generation, writes out the data in odd parity, checks for odd parity generation. It then disables parity generation, writes out the data in even parity and checks for bus errors. If bus errors occur while reading with the same (good) parity checking enabled as the data was written, this fact will be indicated by the "unexpected bus error" message. If no bus errors occur while reading data with different (bad) parity, this fact will be indicated by the "buserror not happening" message. If the bus error was generated for the wrong reason, the "bus error wrong" message will be printed. Assuming T50 went ok, we write even parity data throughout memory with parity generation turned on, and read it back. If any bus errors occur, we print information and loop on that location. If even parity went ok, we do the same thing with odd parity data. This is NOT an exhaustive test of Multibus memory. We map in Multibus memory, write unique data (the address) to each location, read it back and compare it to expected results. If bus errors occur, and/or patterns do not match, we print the appropriate information, and enter looping mode on that address. When we size Multibus memory, if no memory is found, we loop on address 0x100000 (MB address 0). If the size is not a multiple of 128KB, we report that fact and continue on as normal. Bus errors are NOT normal here, as the CPU does not store parity for this memory, but generates it on the read cycles instead. Although some assumptions have been made about the SCC chip working in this test, we also do a local loopback test with patterns ranging from 0x00 to 0xff at 9600 baud on channel B. If receive or transmit timeouts occur, or data is read back incorrectly, an error message occurs. NOTE: If output is not appearing on the terminal, reset the terminal. If output still does

not appear, check the RS-232 cable, and the J1 connectors. This test does not stop if errors are found. In this test, we write and read unique patterns into each of the 5 timers' mode, load, and hold registers. (See AMD data sheets.) If they do not read back correctly, an error message is printed, and the test repeats. SUN modifications and timer function testing are beyond the scope of this diagnostic, and are tested with boot-only diagnostics. We first check to see if clock status is reasonable. If not, we loop on reading the status back. We then check to see if the clock ticks. If not, we print the error and continue as it might be a data line problem. We then write a value to the millisecond counter (0x70), and see if it matches, or just ticked one (0x70 or 0x80). If it did not, we loop on writing the value, and reading it back. We next write a value to the other registers, read them back and compare them. If one reads back wrong, we do a write/read loop on that register. In these tests, we are trying out the various protection and statistic features of the MMU at the page map level. First we map in a page, clearing the A and M bits (see SUN-2 architecture manual), then we read from the page. If the accessed bit is not set, we print a message and go on. Then, we clear the bits again, and write to the page. If the accessed and modified bits are not set, we print an error message and continue. Here, we set up read, write, and execute permissions for both user and supervisor modes, and try appropriate things to exercise each protection bit. If any of these cause a bus error, we print out the information regarding the offense, and loop on that mode until no bus error is found. Here, no permissions are set up, but the page map is valid. As we do exercises of each protection bit, we check to see if a bus error was found, and that the error information is correct. If either is incorrect, we print the error info and loop on this mode until correct. Now, we set the page invalid, access it, and check to see that the proper bus error occurs. If not, an error is printed, and we loop until it does occur correctly. The encryption chip (am9518) is first reset, and we check for the default setup. If the mode is not as we expected, we print an error, and try some more. If the chip never appears to be ready, we give up after printing 'No Chip?'. If the chip appears to be there, we set it up, and check to see that the setup is OK. If not, we print an error message and loop on setting it up. Next, we load the encryption key, checking status as we do so. If the status looks bad, or the command does not think it finished, we print an error message and try again. Now, for the fun part. We load 8 bytes of data to be encrypted, checking status as we do so. If status is bad, we print an error message and try encrypting again. After we get the data in, we wait for output to appear. If it does not, we print a message, and keep waiting. Once output appears ready, we read in the data and compare it to expected values. If the data differs, we print out expected and observed values, continue reading and start the whole encryption test over when done. If after reading the expected 8 bytes, there is still output ready, we also start over after printing an error message. Here we test the interrupt logic on the board in various ways. There are 7 levels of interrupt on the SUN-2 cpu. Level 7 is reserved for timer #1 of the 9513, level 6 for the UARTs, level 5 for the other timers, level 4 is not tested (is used for video sync), levels 1, 2, 3 are software triggerable, in addition to being used by off-board devices. In this test, when an interrupt either is seen at the wrong level, or does not occur, we print an error message, and then start from the beginning of the test. If an error occurs, consult the following paragraphs to figure out where to trace the problem. In general, an interrupt source is gated thru inverters or buffers, and then thru a priority encoder connected to the 68010. Level 7 is generated from the OUT1 pin of the 9513. We set up timer #1 to raise this line, then enable interrupts to catch it. Level 6 is generated from the SCC INT/ pin. Channel B is set up to generate a transmit interrupt, then we enable interrupts to catch it. Level 5 is generated like level 7, but with OUT2 (timer #2). Level 4 is not tested. Level 3, 2, and 1 are tested by setting the EN.INT3, EN.INT2, and EN.INT1 bits in the system enable register, then enabling interrupts. At this point, the number of passes done is printed. If the high order bits are clear (0x0000nnnn), no errors have occurred. If they are set (0xf000nnnn), errors have occurred at some point during the test which you should have already noted.

APPENDIX

primediag REV 2.9 12/16/83 Copyright Sun Micro
T21(s0)T22(sf)T23(sa)T24(s5)T25(sA)T26(sCa)T27(sC5)
T31(p0)T32(pf)T33(pa)T34(p5)T35(pA)T36(pCa)T37(pC5) 0x00100000 bytes of memory found
T41(m0)T42(mf)T43(ma)T44(m5)T45(mc)T46(m3)T47(mA)T48(mCa)T49(mR) T50(Pf)T51(Pv)T52(mb)
0x00020000 bytes of P1 memory found T61(scc)T62(timer)T63(clock)
T71(pageAM)T72(pageon)T73(pageoff)T74(valid)T75(des) T80(Ints) end of pass 00000002

Test takes about 90 seconds to run with 1 MB P2 memory, and 128 KB P1 (Multibus) memory. Each of the error messages is preceded by the test it will appear in. All numbers are in hex. Percent signs (%) are number positions. items in square brackets indicate that the expressions may appear there, but only if called for.

The following are generated whenever they happen, and we are not expecting them. More information is not printed, as there is no known good memory to record it.

Unexpected bus error!! (%) (bus error register is printed) Unexpected address error!! Unexpected exception error!!

Tests T21(s0), T22(sf), T23(sa), and T24(s5) may generate:

bad cons seg reg @ % exp % obs % cx %

T25(sA) may generate:

bad addr seg reg @ % exp % obs %

Tests T26(sCa) and T27(sC5) may generate:

seg check err @ % exp % obs % cx %

Tests T31(p0), T32(pf), T33(pa), and T34(p5) may generate:

bad cons page map @ % exp % obs %

T35(pA) may generate:

bad addr page map @ % exp % obs %

T36(pCa) and T37(pC5) may generate:

bad check page map @ % exp % obs %

When we look for memory, either we find it and print:

% bytes of memory found

or,

NO MEMORY FOUND .. looping on 0

Tests T41(m0), T42(mf), T43(ma), T44(m5), T45(mc), and T46(m3) may generate:

bad cons memory @ % exp % obs % .. looping

T47(mA) may generate:

bad addr mem @ % exp % obs %

Test T48(mCa) may generate:

bad check memory @ % exp % obs %

Test T49(mR) may generate:

bad rand memory @ % exp % obs %

In T50(Pf) you may see the following:

unexpected buserror berr(%) pattern(%) buserror not happening pattern (%) got(%) busseror reg wrong berr (% != 83) pattern (%)

In T51(Pv), the following may be generated:

bus error @ (0x%) berr (0x%)

followed by one or more of:

[lower parity] [upper parity] [timeout] [protection error] [P1 master] [page invalid]

In test T52(mb) you will see either: 0x% bytes of P1 memory found

or,

NO P1 memory found .. looping

and maybe,

P1 memory not multiple of 128k by 0x% bytes no bus error on P1 memory size .. skipping

Then during the test, these may appear:

data mismatch @ 0x% exp (0x%) obs(0x%) .. looping

bus error on P1 memory berr (0x%) .. looping

followed by one or more of:

[lower parity] [upper parity] [timeout] [protection error] [P1 master] [page invalid]

Test T61(scc) can report the following:

tx timeout rx timeout data error exp (0x%) obs (0x%)

For test T62(timer) these are possible error messages:

mode reg error timer % exp 0x% obs 0x% load reg error timer % exp 0x% obs 0x% hold reg error timer %

exp 0x% obs 0x%

For T63(clock), you may see:

bad status (%) looping msec not counting, got 0x% twice bad value on port % exp(%) obs(%) looping

In T71(pageAM), one of the these may appear:

access test exp (0x%) obs (0x%) modify test exp (0x%) obs (0x%)

In T72(pageon), we may see:

buserror berr(%) space(%) write(%) page(0x%)

In T73(pageoff), one of these may be seen:

mismatch berr(%) exp(%) fc(%) w(%) page(0x%) no buserror!! fc(%) w(%) page(%)

In T74(valid), you may see:

mismatch berr(%) exp(%) page(0x%) no buserror!! page(%)

For T75(des), the following may appear:

mode wrong after reset obs(%) exp(%) No chip?? mode wrong after setting obs(%) exp(%) .. looping command not pending on loading of key .. retrying command still pending after loading of key .. retrying input not ready on encryption #% .. retrying input still ready after encryption (%) .. retrying output not ready on readback (%) .. retrying output still ready after readback .. retrying bad data read back exp (%) obs (%) .. restart

In the lase test, T80(ints), we may see:

Got level % instead of % No interrupt on level %

And finally, after each pass you will see:

end of pass %