ERC32 versus 1750A Memory Management Evaluation Report

ERC32 Products Evaluation Programme

Prepared by: Mike Rennie & Carmen Lomba

Approved by: Mike Rennie

Authorised by: Mike Rennie

DOCUMENT STATUS SHEET

Version	Date	Pages	CHANGE(S)	Signature
1.0	26/1/98	46	First version	

Table of Contents

List of Tables and Figures

1. INTRODUCTION

1.1 PURPOSE

The purpose of this technical note is to present the results of a study, the aim of which has been to analyse and compare the mechanisms and techniques applied to the management of memory in computer systems based on two microprocessor cores used for on-board command and control units, i.e. the 1750 16-bit microprocessor with Memory Management Unit (MMU) versus the ERC32 32-bit microprocessor with Memory Controller (MEC).

1.2 SCOPE

The current version of this note draws on GMV's extensive experience in developing on-board real-time software in Ada, using the TLD 1750 Ada Compilation System, for Instrument Control Units based around the 1750A target processor. The lessons that have been learnt from past applications have been borne in mind when studying the ERC32 memory management and a number of representative small-size sub-applications have been developed using the AdaWorld ERC32-Ada Cross-Compiler and tested using ERC32 target simulators furnished by ESTEC and Spacebel.

1.3 DEFINITIONS AND ACRONYMS

BPU	Block Protection Unit
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
INTEGRAL	International Gamma-Ray Astrophysics Laboratory
MEC	Memory Controller
MERIS	Medium Resolution Imaging Spectrometer
MMU	Memory Management Unit
PROM	Programmable Read-Only Memory
RAM	Random Access Memory

2. REFERENCES

2.1 REFERENCE DOCUMENTS

Reference	Títle	Code	Version	Date
[RD.1.]	Military Standard sixteen-bit Computer Instruction Set Architecture	MIL-STD-1750 (USAF)	2	20/2/88
[RD.2.]	Version Description Document for the sun4/sparc 1750 Ada Compiler System		96T024	15/8/96
[RD.3.]	ERC32 Software Development Toolset User Manual	MCD-LOG-P2-TUM	1/0	15/3/97
[RD.4.]	TSP AdaWorld Development Environment for SPARC based Workstations under Solaris 2 to ERC32 Targets. Apendix F	MCD-ALS-P2-DOC-008	2	29/11/96
[RD.5.]	TSP AdaWorld Development Environment for SPARC based Workstations under Solaris 2 to ERC32 Targets. Cross Development Guide	MCD-ALS-P2-DOC-015	2	12/12/96
[RD.6.]	MEC Rev. A Device Specification	MCD/SPC/0009/SE	4	10/4/97
[RD.7.]	Final Report	MCD/MNT/0015/SE	1	27/5/97
[RD.8.]	ERC32 System Overview Rev. CBA	MCD/TNT/0020/SE	3	10/4/97
[RD.9.]	TSC691E Integer Unit. User's Manual		H	2/12/96
[RD.10.]	TSC692E Floating Point Unit. User's Manual		H	2/12/96
[RD.11.]	Target Simulator User's Manual (Spacebel)	32B-SBI-SUM-0189-003	2 Rev 1	19/12/96

3. OBJECTIVES

3.1 TECHNICAL OBJECTIVES

To compare the ERC32's memory management with that of the 1750A, a prototype application will be developed in Ada for the ERC32 and for the 1750A. Based on exisiting re-usable flight software and executed, respectively, on the TLD 1750 Simulator and ERC32 Simulators, the prototype will aim to evaluate the pros and cons with respect to each processor regarding the ease and efficiency of implementing such functionalities as:

- ❑ background RAM scrubbing

- ❑ memory checksum (CRC) verification

- ❑ etc.

3.2 MOTIVATION FOR THE EVALUATION APPROACH

The motivation for the interest in comparing the ERC32 and 1750 is as follows:

- ❑ Although it is generally recognized that the 1750 is becoming obsolete, there still appears to be a degree of resistance to moving to a 32-bit processor, and the 1750 continues to be specified as the baseline target processor in new projects. This resistance could be diluted if there was more awareness of the drawbacks of the 1750's 16-bit architecture and the advantages of the ERC32's 32-bit architecture.

- ❑ One of the problems with the 1750 is the way it manages extended memory (>64K) via an external device, the Memory Management Unit (MMU). These problems, which have a considerable influence over the resulting software architecture, are not generally foreseen, understood or appreciated by the user community. The aim is to focus on these problems and to show how life would be so much easier for the software developers if they had an ERC32 instead of a 1750.

  Specifically, one of the problems that has been encountered in the INTEGRAL programme exemplifies how a lack of understanding of the drawbacks of the 1750 can have a significant impact on the software development:

  The 1750 has been chosen for the Data Processing Electronics, and this limits the size of arrays to under 64K words. However, the Prime Investigators, charged with developing the science instrument application software were expecting to be able to declare enormous (>64Kwords) arrays for generating histograms of scientifically significant events. Now that they understand that the 1750 memory management makes that very difficult, they have to re-think how to structure the software architecture.

  Another example of the disadvantages of the 1750's extended memory management is the job of correcting memory errors - something which is time consuming in the 1750 because of the need to change MMU page register values to permit a sweep of the RAM, but which is a lot easier with the ERC32. Memory checksum computation shares similar problems because of the need to sweep across extended memory.

The motivation for the evaluation approach is the desire to quantify the savings in terms of time and effort that can be gained by developing software for a 32-bit target as opposed to a 16-bit target, because of the simplification of the implementation of the software.

4. DESCRIPTION OF 1750 MEMORY MANAGEMENT (MMU)

The <u>1750 microprocessor</u> is a 16-bit machine. This means that the processor itself understands only 16-bit addresses. That would seem to limit the address range to 64K. However, it is possible to configure the 1750 so as to be able to access memory using a 20-bit address, i.e. with an address range of up to 1Mwords.

How then does the 1750 processor, with its 16-bit addresses, specify a 20-bit address?

The answer is that it can access memory via the <u>Memory Management Unit</u>, or " MMU" .

4.1.1 Logical Memory

The software addresses the physical memory via 16-bit logical addresses as illustrated below:
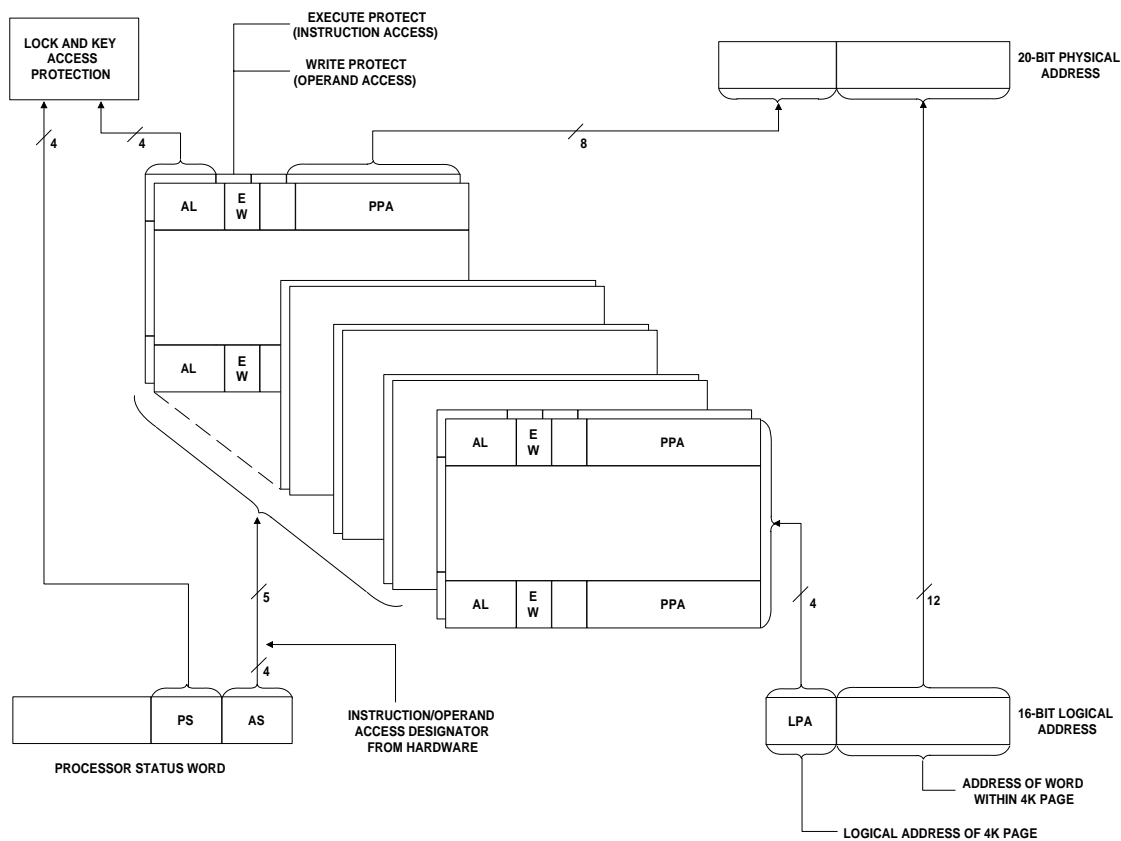
Figure 3.2-1 1750 Extended Memory Mapping (Source: MIL-STD-1750A (USAF) [RD.1.])

The MMU takes the 16-bit address provided by the microprocessor and converts it to a 20-bit address. The 16-bit address is usually referred to as the "<u>logical address</u>", because it is the address generated by the programme logic being executed by the microprocessor, while the 20-bit address is called the "<u>physical address</u>", because it provides the address of the word of memory which the programme will physically access.

The 20-bit physical address is composed of a "physical page address" (PPA) occupying the 8 MSbits, and the "page offset", with a range of 4Kwords (0..4K-1), in the 12 LSbits. The physical memory is thought of as if it were split into 256 consecutive pages of 4Kwords each. The PPA is in effect the number of such a physical page. The first page is number 0, i.e. PPA=0, and the last is PPA=FFh.

In the same way as the physical memory is thought of as being split into 4Kword pages, so too is the logical memory space, i.e. the 64K of memory seen by the microprocessor at any given time is split into 16 pages of 4Kwords each.

4.1.2 Logical-to-Physical Address Conversion Procedure

The principal way in which the MMU converts 16-bit logical addresses to 20-bit physical addresses is by mapping logical pages to physical pages. Basically, the conversion procedure is as follows:

The 12 LSbits of the logical address are used to form the 12 LSbits, or "page offset" of the physical address.

The 4-bit "Address State" field in the Processor Status Word (a special purpose 1750 register) is used to select one group, out of a possible 16 groups, of 16-bit registers in the MMU.

Each such group contains 32 registers, divided by the MMU into two sets of 16 registers each. The MMU automatically goes to one set when the address is that of an operand (a variable or constant) and to the other set when it is an instruction fetch address. An "access designator" signal coming from the microprocessor (without software intervention) lets the MMU know whether it is dealing with an operand or instruction address. So, given the type of address (operand or instruction), and a given value of address state (from 0 to 15), the MMU goes to a specific set of 16 registers in a specific group. Then the 4 MSbits of the logical address are used to select one of the registers in that set. The 8 LSbits of that MMU register contain the PPA which is used to complete the 20-bit physical address.

In other words, each MMU register maps a logical page, containing (logically) either operands or instructions, to a physical page. There can be up to 16 groups of registers in use in the MMU, i.e. as many as one per Address State. If all 16 groups were used, it would be possible to put a unique PPA into each operand address register, and the same again for the instruction address registers, and thus map the entire 256 pages (1Mword) of phyical memory to 256 logical operand pages as well as to 256 logical instruction pages.

Note that using the MMU to access physical memory, without dynamically modifying the initial settings of the MMU page registers, means that it is possible to address up to 2Mwords of RAM, divided equally between Instructions and Operands, i.e. 1Mword of RAM each.

Incidentally, although logical memory is divided into pages of operands and pages of instructions, it is perfectly possible to map any number of logical pages, of whichever type, to the same physical page. Thus, a physical page could contain both operands and instructions, but in that case it would have to be mapped by both a logical operand and a logical instruction page.

It is not absolutely necessary to use all 16 groups of MMU registers, i.e. all 16 address states, to access the whole 1Mword of physical memory, because the MMU register contents can be changed dynamically by the executing programme. In other words, the logical pages in a given address state could be made to map to different physical pages at different times. That is typically how RAM scrubbing processes operate on the 1750: a unique logical operand page in a single address state (usually AS0) is set aside for the purpose of being mapped to each and every

physical page in turn, and the RAM scrubbing code simply reads from that logical page, i.e. the code appears to be reading words from the same logical address range all the time, but really, because the logical page mapping is changed dynamically, the physical address range being read is not the same.

However, it is usual for programmes which access more than 64Kwords to use more than one address state, using as many MMU registers as necessary to be able to set up a static mapping of logical pages to physical pages. A static mapping avoids the need to perform time-consuming External IO (XIO) instructions to modify the MMU registers, and the software is generally less difficult to debug and more reliable when the mapping is static, because only the value of the AS field in the PSW register changes the visibility of physical memory pages.

4.1.3 Impact of MMU on SW Design and Implementation

Now that we have seen how the 16-bit 1750 is able to access memory with a 20-bit address range, we will see that this reliance on the page mapping done by the MMU can have serious consequences for the design and implementation of the software, and makes it unusually difficult and time-consuming to achieve reliable software.

The most important fact that must be borne in mind is that for each setting of the AS field in the PSW register, 64Kwords of physical memory are mapped to the 64Kwords of logical address space which is "visible" to the processor. If the programme, in a given moment, has to access a physical page which is not visible, the usual solution is to switch the AS to an address state which does map the physical page to logical memory. Thus, the programme has to be aware of which address state it is currently using and which physical pages are visible in each address state, and has to take care to switch address states when necessary.

In practice, it is very difficult at the design stage to estimate the size of each of the architectural components of the software, and when the completely implemented programme is ready to be linked, it is often necessary to re-distribute the modules across address states in order to fit the programme into the memory. It is quite common for this to lead to mysterious errors occurring because the re-distribution has, unwittingly, invalidated some logical addresses under certain conditions, e.g. when a procedure running in one AS accesses a variable which used to be in the same AS but has been moved to another page in another addresss state, rendering it physically inaccessible by the procedure.

Previous experience during the integration test campaigns on MERIS and INTEGRAL demonstrates that the additional time spent investigating and correcting problems rooted in the use of the MMU to access extended memory can be around 2-3 elapsed months.

5. DESCRIPTION OF ERC32 MEMORY MANAGEMENT SYSTEM

5.1 ERC32 MEMORY CONTROLLER (MEC)

The memory in a computer based on the ERC32 is managed by the Memory Controller (MEC) unit which is an integral part of the ERC32 core, as shown below:
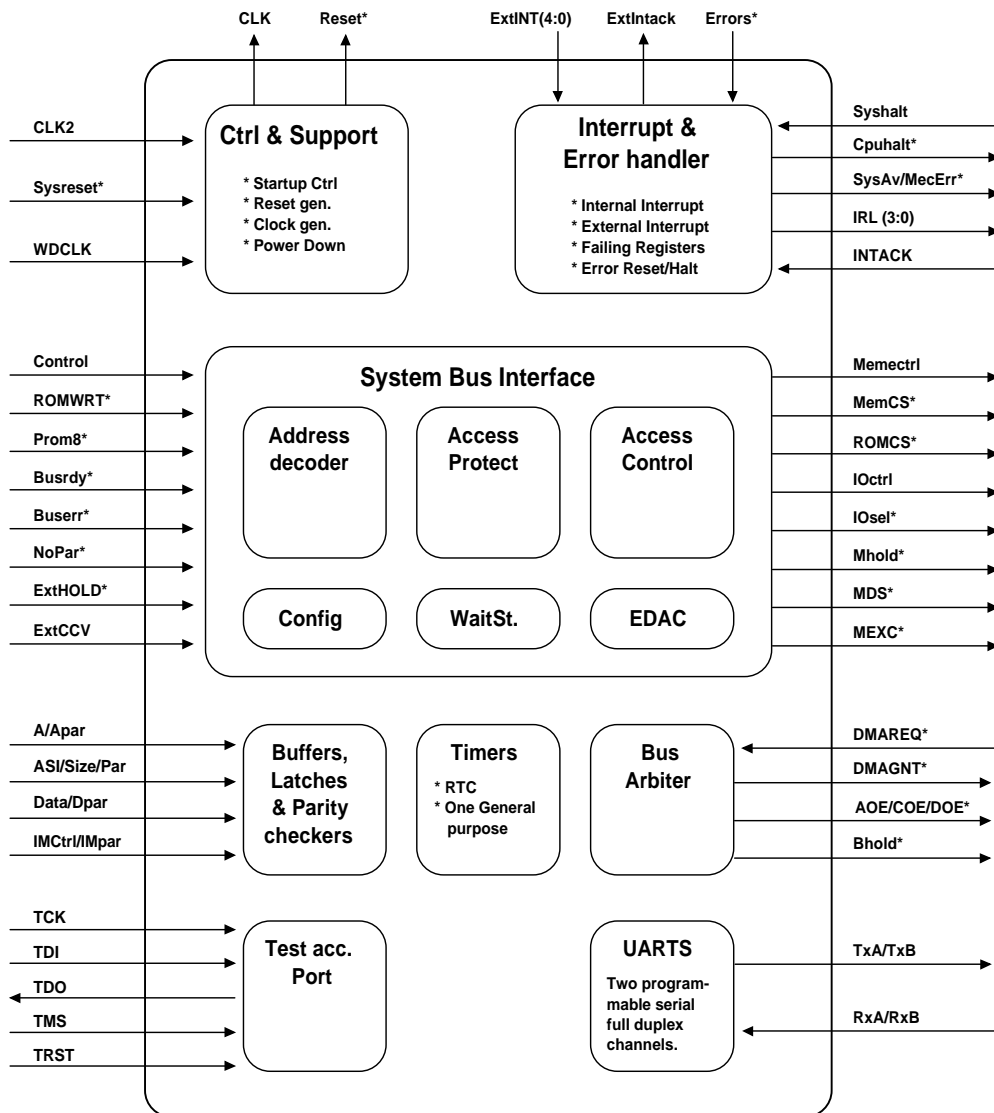
Figure 5.1-1 ERC32 Memory Controller (MEC)

The MEC implements support functions such as memory control and protection, error detection and correction (EDAC), wait-state generation to allow for the different access times related to memory and external devices, timers, interrupt handling, a hardware watch-dog, UARTs for communication with external devices, test and debugging support, as well as concurrent error detection facilities.

The MEC supports configurable memory sizes, programmable through a Memory Configuration Register. Parity and EDAC protection of RAM is programmable in the MEC, both for memory and I/O.

The PROM used for the bootstrap software and for the on-board software code and constants storage can be implemented as normal 32-bit word-size (plus an additional 8 check bits) EDAC protected memory, or the MEC can be configured to access PROM via an 8-bit bus, requiring 4 accesses per 32-bit word but with no check-bit protection. The MEC supports the use of EEPROMs instead of PROMs, which is very useful during system integration, test and even maintenance. The size of PROM is configurable, with values of 2^n bytes (n in the range 17 to 24), i.e. from 128 Kbytes up to 16 Mbytes.

There are no special I/O instructions in the SPARC architecture, unlike the "XIO" instructions of the 1750, and instead all I/O accesses are made via a memory mapped I/O bus.

The ERC32's internal chip select signals are specially defined and decode up to 32 Mbytes of RAM and 16 Mbytes of boot PROM. However, additonal "extended RAM" or "extended PROM" can be included in the system, implemented as I/O units, although with the correspondingly longer access times, with user-provided glue logic implementing the address decoding.

In order to prevent indeterminism in systems that allow concurrent access by external devices to the ERC32 core's local RAM via DMA, the MEC allows an alternative solution using a dedicated 32-bit dual-port memory area called the Exchange Memory. The MEC implements the arbiter for this data exchange area.

6. TLD 1750-ADA EXTENDED MEMORY MANAGEMENT

To understand better the subject of 1750 extended memory, it is useful to study the way that multiple address states are supported by the TLD 1750 Ada Compilation System.

6.1 GENERATION OF THE MMU INITIAL CONFIGURATION BY THE TLD 1750 LINKER

When the expanded memory option is specified, the TLD linker generates MMU page register settings and puts them into the load module. The settings put into the load module are used by TLD's run-time kernel, at initialisation, to perform the sequence of XIO instructions that write to the MMU, initialising the logical to physical page mapping.

The MMU settings generated by the TLD linker can be seen in the .map file, e.g.:

Physical Address (Page Number)	Symbol
0	%A$**PRI00**

"**P**age **R**egister for **I**nstructions in Address State **0**, Logical Page **0**"

This indicates that physical page 0 is mapped to the Logical Page Register 0 for Instructions in Address State 0, by TLDlnk (the TLD 1750 Linker). The symbol A$PRI00 is an external symbol generated by the linker and used by the RTX module RTX_PDG to initialise the MMU at run-time.

6.2 LINKER-GENERATED TRANSIT ROUTINES FOR CALLS ACROSS NODES

6.2.1 Concept of the Transit Routine

The TLD linker identifies calls that are made to a subprogramme which is not visible in the caller's address state, and automatically modifies the call such that it is made instead to a "Transit Routine Instruction Packet", generated by the linker, which in turn passes control to a common "Transit Routine" situated in the RTX. The Transit Routine takes care of modifying the AS field in the PSW such that when the code jumps into the called subprogramme, it does so after it has been made visible, and on return from the called subprogramme, the AS field is restored such that the caller is once again visible. The Transit Routine also manages a chain of calls made via transit routines, i.e. so that a subprogramme called via a transit routine can itself make a call via a transit routine and still be able to return control back to the caller correctly, and similarly makes sure that Ada exceptions raised by a called subprogramme are sent to the caller, across address states, for handling.

6.2.2 Concept of Nodes

How does the linker identify calls that require a transit routine? The answer is that the linker obliges the user to organize his programme control sections, or object files, in a number of "nodes", in a tree-structure with parent-child relationships.

For example, consider the following node structure:
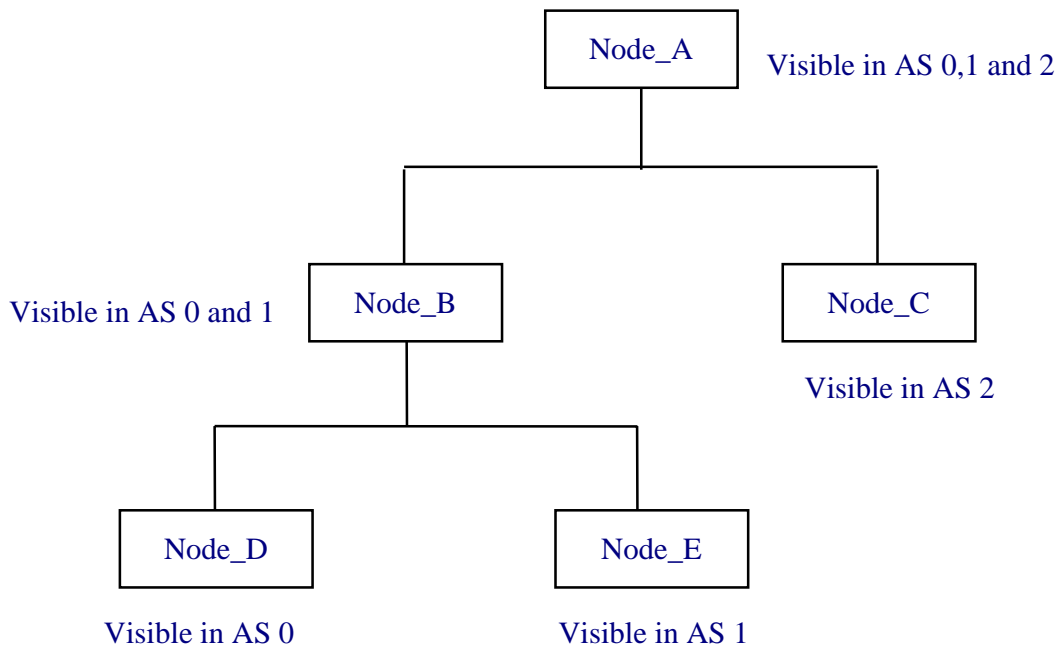
Figure 6.2-2 Example TLD Linker Node Structure

Nodes D and E are children of Node B. Nodes B and C are children of Node A.

The linker begins by allocating programme code and operand control sections to physical memory, making sure that no physical page contains control sections from two or more nodes, i.e. each physical page will contain control sections from one, and only one, node.

The linker then allocates each of the leaf nodes, i.e. D, E and C, to unique address states, i.e. 0, 1 and 2 respectively, and each parent node is allocated to each of the address states to which its offspring have been allocated. Thus, Node_B is allocated to address states 0 and 1, while Node_A (referred to as the "root" node) is allocated to address states 0, 1 and 2.

The aim of allocating a parent node to each of the address states to which its offspring are allocated is to reduce the number of calls that have to be made via a transit routine. The linker sets up transit routine calls where a call is made from one node to another, except when the call is made to a subprogramme in a parent node.

The allocation of a node to an address state means that logical pages in each address state are mapped, by the linker, to the physical pages that contain the code and operands that comprise the control sections assigned to that node. Where a node is allocated to more than one address state, the same logical page to physical page mapping is duplicated in each of those address states.

6.2.3 Impact of the Transit Routines on Timing and Sizing

The work done by the Transit Routine to make sure that visibility is maintained when calls are made across address has a cost in terms of both memory and timing budget. For each operation that requires a transit routine to make it visible before it is called, the linker generates a 2-word data packet and a 5-word instruction packet. The overhead caused by the Transit Routine has been measured on a host simulation of a MA31750 CPU running at 13Mhz with 1 wait-state as approximately 70µsecs.

6.2.4 Impact of Nodes on Sizing

One of the most serious consequences of the duplication of logical-to-physical page mapping, for parent nodes, is that if the parent nodes, and especially the root node, become too large, it can lead to a situation in which a large number of address states have to be used in order to get enough logical pages to permit the child nodes to be completely mapped to physical memory. Then there would be a higher overhead if the child nodes frequently had to call each other subprogrammes. In an extreme situation, there might not even be enough logical pages left to map the child nodes, in which case the parent nodes would somehow have to be reduced in size, by changing the actual software architecture and then re-assigning control sections to child nodes.

This problem can be illustrated by a simple example:

Suppose the operands belonging to the packages that have been assigned to the root node occupy 12 pages (equals 48Kwords of physical memory), and that the root node has to be visible in all 16 address states. That means that the root node's operands occupy 48*16=768Kwords of logical operand memory, leaving only 4 operand pages per address state (48 operand pages) for the rest of the nodes. Now suppose that the control sections belonging to the rest of the nodes amount to 127 pages (508Kwords) of operands (this might be the case if the 1750 were performing the telemetry data processing for a science instrument). Clearly, these 127 pages will not fit into the 48 that are available. If at least 4Kwords of operands could be moved from the root node to a child node so that the root occupied only 8 pages of operands, there would be 8 operand pages per address state (128 operand pages) available outside the root, and then it would be possible to allocate the child nodes operand control sections to the availble address states. However, moving the 4Kwords of operands out of the root node might imply a significant restructuring of the software architecture, e.g. creating new packages, and new operations, to manage that 4Kwords of operands.

Note that if a subprogramme being executed in one address state needs to access operands declared in a control section which is not visible in that address state (because it has been allocated to a node which isn't in the caller's node's ancestry), the access has to be made via a subprogramme, not by direct memory reference. Neither the compiler nor the Ada run-time will detect an error if a reference is made to a variable while executing in an address state which doesn't have visibility of that variable. The compiler will generate references to the variable, without regard to its valid address state(s), using 16-bit (logical) addresses, and the processor will access the word at that logical memory address, without regard to whether the variable is logically visible. In practice, this leads to very strange errors that are often difficult to resolve. The programme appears to be mis-reading or corrupting variables, just because it is accessing the correct logical address but the wrong physical address.

6.2.5 Example of a Typical MMU Initial Configuration

A typical MMU initial configuration is illustrated in the following diagram, which shows the mapping used for the INTEGRAL DPE Common Services Software, which has a root node called ROOT and two child nodes called AS0 and AS1:

	Address State 0		Address State 1	
	Operand Pages	Instruction Pages	Operand Pages	Instruction Pages
0	0	0	0	0
1	801	F	1	14
2	2	10	2	FFF
3	3	11	815	FFF
4	4	4	16	FFF
5	5	12	FFF	FFF
6	6	13	FFF	FFF
7	7	FFF	FFF	FFF
8	8	FFF	FFF	FFF
9	9	FFF	FFF	FFF
A	A	FFF	FFF	FFF
B	B	FFF	FFF	FFF
C	C	FFF	FFF	FFF
D	D	FFF	FFF	FFF
E	E	FFF	E	FFF
F	FFF	FFF	FFF	FFF

Key:

= ROOT node

= AS0 node

= AS1 node

Figure 6.2-3 Typical MMU Configuration in 2 Address States

Note that unused logical pages are given the value FFFh, i.e. they are mapped by default to the very last physical page (FFh) with Execute/Write Protection enabled.

The programme occupies 23_{10} physical pages (68Kwords) but it occupies 21_{10} pages (84Kwords) of logical operand memory and 9 pages (36kwords) of logical instruction memory. More logical than physical memory is occupied because physical pages 0, 1 and 2 are visible to two address states, and because page 0 contains both for operands and instructions.

Physical page E contains the programme heap space. The heap has to belong to the root node because the heap is used by the Ada run-time which can be executing in any address state. The data on the heap has to be visible to the Ada run-time regardless of switching between address states.

One of the reasons for the root node taking up a large amount of space is when a subprogramme called via a transit routine uses operands that are also visible to the caller. If the operand is a global variable, the control section to which it belongs has to be allocated to node that is a parent of the caller and callee. If the variable is on the stack of the task that is making the call, then the

stack too has to be visible in both address states. This usually means that the entire task stack has to be visible in all the address states in which the task could execute, and that in practice often forces the user to put the task stacks into the root node.

6.3 GUIDELINES FOR LINKING A PROGRAMME IN EXTENDED MEMORY

GMV have drawn on experience in the development of 1750-targetted on-board software, such as the application software for the MERIS instrument aboard the ENVISAT platform, and the operating system for the Data Processing Electronics of the science instruments aboard the INTEGRAL platform, to develop a number of guidelines that may help to avoid some of the problems commonly encountered when trying to link a complex programme that has to use extended memory.

What is important in the scope of this report is that these guidelines also give an insight into the kind of problems that occur when trying to implement a large programme in extended memory:

1. Place each task's stack in a node that is an ancestor of all the nodes which contain code that could be executed by the task (it usually means the root node). This will ensure that if a local variable, i.e. one that is placed on the stack, is passed as a parameter to an operation which is in a module in a different AS, it will still be visible at the same logical address.

2. The control sections containing the roots of tasks, i.e. the procedures whose addresses are used by the operating system as the starting addresses for the tasks, must be in the same address state in which the operating system executes (usually AS0 only). That is because the OS does not normally have any way of knowing in which AS the task entry point is situated - it assumes that it is visible.

3. Unless great care is taken over the assignment of logical pages, the code should avoid calling operations with parameters that reference static (not on the stack) arrays. Instead, those arrays should be declared on the stack of the calling task, and the stack should be declared in an object which is allocated to the root node. That will avoid problems caused by making a call with a parameter referencing an array which is not visible (at the same logical address) in the AS in which the called subprogramme executes .

4. Try to keep the packages small, so that they allow greater flexibility in moving bits of the programme from one node and/or AS to another. A large software decomposed into a large number of small packages, instead of a small number of large packages, may be more time-consuming to develop and maintain from a design point of view, but will be allow much greater flexibility when it comes to allocating control sections to nodes, and ultimately will make it easier to get the programme to fit correctly into logical memory.

5. The packages that contain data that have to be visible across address states should be kept as small as possible, i.e. including only that part of the code and/or data that really needs to be visible across address states.

The aim is to make the parent nodes as small as possible. It should be borne in mind that the parent nodes take up logical memory space, i.e. use up logical pages, in every AS in which they have to be visible.

If, for example, the root node were to occupy n pages of logical operand memory, that would leave only 16-n logical operand pages free for the other nodes in each AS.

6. If variables or constants of a composite (record) type are passed as parameters, make sure that they are visible in the AS of both the calling and called operations. For example, declare them in a package which is in a node that is visible to, i.e. is an ancestor of, the node containing the caller.

7. Don't make the task stacks unnecessarily big. The stacks usually have to be put into the root node, and that means they can end up starving all the address states of much needed logical memory space in which to place the rest of the nodes.

8. If an operand has to be situated at a specific known physical address, it is recommended to give the linker a user-generated linker directive to assign the relevant physical page to the same number logical page, in the address state *n* where *n* is equal to the 4 MSbits of the physical address. In that way, the logical address given by the Ada attribute 'ADDRESS will be valid as the physical address while executing in address state *n*. For greater reliability, that logical page should belong to the root node, so that the address is valid across address states.

9. All elaboration code generated automatically by the compiler has to be visible in the address state in which the programme entry point is placed, because the TLD linker does not generate transit routines for compiler-generated elaboration code.

10. Beware of passing static aggregate values as parameters to operations that are provided by a package which is in a different node to the calling operation.

Consider the following example of a call, made from package "TOM", assigned to a node in AS1, that passes an aggregate parameter to another procedure, "PUT_DATA", provided by module "FRED" that is in AS0:

```
FRED.PUT_DATA (DATA => (A => TRUE,
                        B => 474,
                        C => 10));
```

The compiler will see that the aggregate is really a constant and will therefore generate a constant, which it will put into the $CONS control section of TOM. However, the constant will be passed to FRED.PUT_DATA by reference, i.e. by logical address. The problem is that if TOM and FRED are in different AS, and furthermore TOM's node is not an ancestor of FRED's node, then TOM's $CONS is not visible to FRED. In other words, the logical address passed to FRED won't have the same meaning in FRED as in TOM. In fact, FRED will receive rubbish in the parameter DATA, even though TOM has sent correct data!

One possible solution is to declare the aggregate as a variable on the calling task's stack (which must be in a node which is a common ancestor of TOM's and FRED's nodes). Another solution is to put the constant declation into a package which is in the common ancestor node.

11. There must be enough room on the main programme stack for the chain of transit routine packets which are created/deleted dynamically when calls are made across nodes.

7. LIST OF TESTS CARRIED OUT

The tests that have been carried out are as follows:

- Periodic RAM scrubbing over an address range of 256 Kbytes

- Cyclic Redundancy Check (CRC) using the CCITT algorithm

These algorithms have been implemented in Ada cross-compiled for both the ERC32 and 1750 processors.

For the elaboration of these tests, the following tools have been used:

- ERC32:

  - AdaWorld-ERC32 Cross-Compilation System

  - ERC32 Target Simulator SIS

  - ERC32 Target Simulator from Spacebel.

  Since the remote use of the cross compiler prevents use of the Ada source-level cross debugger, which is x-windows based, assembly-level debugging provided by the target simulators was used instead.

- 1750A:

  - TLD Ada-1750 Cross-Compilation system, version 97Sa027, including:

    - Ada-1750 Cross-Compiler (TLDada)

    - 1750 Assembler (TLDasm)

    - 1750 Linker (TLDlnk)

    - MS1750 Target Simulator (ms1750)

    - Ada Symbolic Debugger (TLDdbg)

    - Environment Simulator (TLDenv) running the DPE HW Simulator developed by GMV for the INTEGRAL programme. The DPE HW Simulator provides improved timing accuracy.

8. COMPARISON OF TEST RESULTS

Table 8-1 below shows the results of the measurements of simulated time for the CRC-CCITT and RAM scrubbing algorithms, implemented for the 1750 and the ERC32. For the ERC32 implementation, two available simulators have been used, i.e. the SIS Simulator developed in-house by ESTEC and the Target Simulator from Spacebel.

It is to be remarked that, obviously, the input data for the CRC-CCITT algorithm are the same in both implementations, i.e. for the 1750 and the ERC32.

Target	Simulator	CRC-CCITT				RAM Scrubbing			
		Time (msecs)	Instruction Count	Cycle Count	CPI	Time (msecs)	Instruction Count	Cycle Count	CPI
1750A	TLDenv	18.89	21852	–	–	970.31	1442207	–	–
ERC32	SIS (ESTEC)	4.36	28432	43639	1.53	243.24	1446600	2432449	1.68
	Target Simulator (Spacebel)	7.45	50605	74524	1.47	246.30	1468422	2463007	1.68

Table 8-1: Comparison of Test Results - Simulation Time and Instruction & Cycle Count

Some conclusions can be drawn from the above measurements:

1. The ERC32 executes the same functionality in less time than the 1750.

The 1750 suffers the overhead of the time-consuming XIO instructions required to read and modify the contents of the MMU page registers.

2. The SIS and Spacebel simulators do not give exactly the same results.

9. ADAWORLD/ERC32 VERSUS TLD/1750

9.1 RANGE OF MEMORY

The most obvious advantage of the ERC32 is that it is capable of supporting more memory than the 1750.

However, the conclusion that can be drawn fron this report is that even if a software system's memory requirements are within the capabilities of the 1750, if the programme is going to need more than 64Kwords of RAM, then life is going to be more difficult for the designer, implementors and test team than if the system was targetted to an ERC32. This is principally because the job of fitting the programme into extended memory successfully, i.e. in such a way that the system's performance is acceptable and all possible errors due to generating and using erroneous logical addresses are avoided, is difficult and time-consuming, particularly because it is easy to make a mistake that not even the TLD Ada-1750 compiler or TLD 1750 linker will spot.

The 1750 extended memory management has to be taken into account when designing the software architecture and assigning data and processes to objects. The overhead of the TLD transit routines has to be taken into account when designing the node structure and sharing out the code and operands among logical address states. A poor analysis of the calls that will be made via transit routines can lead to impaired performances.**These problems are virtually non-existant in the ERC32**. The only comparable problem is that of deciding in which areas of memory to place each piece of the programme, based on the different access times for different areas of memory.

9.2 PLACING PROGRAMME SECTIONS IN MEMORY

The AdaWorld-ERC32 Binder "SEGMENTATION" option provides flexibility when it comes to arranging the different parts of the programme in memory.

The TLD-1750 ACS is also very flexible in this respect - the compiler switch "SINGLE_MODULE" allows the user to choose whether to create just one object module per compilation unit or for each package body and specification. The TLD linker, using the "SET" directive, then allows the user to specify logical or physical locations for each of the modules.

9.3 MEMORY PROTECTION

Both the ERC32 MEC and 1750 MMU detect accesses to unimplemented or illegal memory addresses, signalling the error to the microprocessor.

The ERC32 MEC allows the user to define memory segments for which write access is allowed, further qualified by permission in Supervisor and/or User Mode, and the MEC will detect access violations and signal the error to the microprocessor. A maximum of two segments with write access can be defined.

The 1750 MMU, used in conjunction with the optional Block Protection Unit (BPU), allows the user to specify execution access rights for individual logical instruction pages and write access rights for individual logical operand pages In addition, the optional Access Lock and Key feature can be used to specify a 4-bit code in each MMU page regsister. The microprocessor receives an error signal if it attempts to access a page when the PS field of the Processor Status Word has not

been set to a value that is compatible with the Access Lock and Key code. This feature of the 1750 is intended to be used together with an operating system which has sole responsibility for changing the PS field. For example, the OS could prevent the application level software from accessing certain pages of memory which are used exclusively by the OS.

Thus, it seems that the 1750 memory protection scheme is more sophisticated than that of the ERC32, although it can only be fully utilized at the expense of employing the BPU. However, for reasons of mass and power consumption, the BPU is not widely used in space applications. When the BPU is not used, the only protection provided by the MMU is that given implicitly by the logical-to-physical page mapping.

10. TLD/1750 SOURCE CODE

10.1 CRC-CCITT CYCLIC REDUNDANCY CHECK

10.1.1 Ada Package Specification "TYPES"

```
with UNCHECKED_CONVERSION;

package TYPES is

   type BYTE is range 0 .. 2**8 - 1;
   for BYTE'SIZE use 8;
   -- byte type of 8 bits


   type T_UNSIGNED_16 is range 0..(2**16)-1;
   for T_UNSIGNED_16'Size use 16;
   -- Unsigned 16-bit integer

   type WORD is range 0 .. 2**31 - 1;
   for WORD'SIZE use 32;
   -- word type of 32 bits

   type WORD_LOGICOS is array( 1 .. 32 ) of BOOLEAN;
   pragma PACK(WORD_LOGICOS);
   -- represents a word as an array of boolean

   type BITS_LOGICOS is array( 1 .. 8 ) of BOOLEAN;
   pragma PACK(BITS_LOGICOS);
   -- represents a byte as an array of boolean

   function WORD_TO_BOOLEAN is new
       UNCHECKED_CONVERSION(WORD, WORD_LOGICOS);
   -- converts a word to an array of boolean

   function BOOLEAN_TO_WORD is new
      UNCHECKED_CONVERSION(WORD_LOGICOS, WORD);
   -- converts an array of boolean to word type

   function BYTE_TO_BOOLEAN is new
       UNCHECKED_CONVERSION(BYTE, BITS_LOGICOS);
   -- converts a byte to an array of boolean

   function BOOLEAN_TO_BYTE is new
      UNCHECKED_CONVERSION(BITS_LOGICOS, BYTE);
   -- converts an array of boolean to byte

   type CHECKSUM is
      record
         INDATA4: BYTE;
         INDATA3: BYTE;
         INDATA2: BYTE;
         INDATA1: BYTE;
         -- Most significant bytes
      end record;

   for CHECKSUM use
      record
         INDATA4 at 0 range  0 ..  7;
         INDATA3 at 0 range  8 .. 15;
         INDATA2 at 0 range 16 .. 23;
         INDATA1 at 0 range 24 .. 31;
      end record;

   function CHECKSUM_TO_CHECKSUM is new
      UNCHECKED_CONVERSION ( WORD, CHECKSUM );

   type SPLIT_U16 is
```

```
    record
       LOW, HIGH : BYTE;
    end record;

for SPLIT_U16 use
    record
       LOW  at 0 range 0 .. 7;
       HIGH at 0 range 8 .. 15;
    end record;

function U16_TO_BYTES is new
    UNCHECKED_CONVERSION ( T_UNSIGNED_16, SPLIT_U16 );

MAX_NUMBER_OF_INPUT_BYTE:  constant POSITIVE := 4;
-- number of input bytes for the checksum

end TYPES;
```

10.1.2 Ada Package Specification "MEM_CRC"

```
with TYPES;
use  TYPES;

package MEM_CRC is
-- package specification of the function that calculates the
-- CRC-CCITT checksum

    function GET_CRC (DATA     : in BYTE;
                      SYNDROME : in WORD) return WORD;

end MEM_CRC;
```

10.1.3 Ada Package Body "MEM_CRC"

```
package body MEM_CRC is

    function GET_CRC(DATA     : in BYTE;
                     SYNDROME : in WORD) return WORD is

        AUX_SYNDROME : WORD;
        AUX_DATA     : WORD;
        DATA_EXP2    : WORD:= 16#0002#;
        DATA_EXP8    : WORD:= 16#0100#;
        DATA_NULL    : WORD_LOGICOS:= (others => FALSE);

        OCHENTA : WORD := 16#0080#;
        OCHOMIL : WORD := 16#8000#;
        ONCEMIL : WORD := 16#11021#;
        EFFFFES : WORD := 16#FFFF#;

        BOOL : BOOLEAN;

    begin

        AUX_SYNDROME:= SYNDROME;
        AUX_DATA:= WORD(DATA);

        for INDEX in 1 .. 8 loop

            BOOL :=
              ( ( WORD_TO_BOOLEAN( AUX_DATA ) and
                    WORD_TO_BOOLEAN( OCHENTA  )
                )
                xor
                ( WORD_TO_BOOLEAN
                      ( BOOLEAN_TO_WORD
                            ( WORD_TO_BOOLEAN ( AUX_SYNDROME ) and
                              WORD_TO_BOOLEAN ( OCHOMIL )
```

```
                              )
                            / DATA_EXP8
                        )
                    )
                ) /= DATA_NULL ;

        if BOOL then

            AUX_SYNDROME :=
                BOOLEAN_TO_WORD
                    ( (WORD_TO_BOOLEAN (AUX_SYNDROME * DATA_EXP2) xor
                       WORD_TO_BOOLEAN( ONCEMIL )
                      )
                      and WORD_TO_BOOLEAN( EFFFFES )
                    );
        else
            AUX_SYNDROME :=
                BOOLEAN_TO_WORD
                    ( WORD_TO_BOOLEAN(AUX_SYNDROME * DATA_EXP2) and
                      WORD_TO_BOOLEAN( EFFFFES )
                    );

        end if;

        AUX_DATA:= AUX_DATA * DATA_EXP2;

    end loop;

    return AUX_SYNDROME;

   end GET_CRC;

end MEM_CRC;
```

10.1.4 Ada main entry point procedure "MAIN"

```
with TYPES;
use TYPES;
with MEM_CRC;

procedure MAIN is

    CHK : WORD;
    -- checksum of the input data

    type BYTE_ARRAY is array
        (1.. MAX_NUMBER_OF_INPUT_BYTE + 2) of BYTE;


    INDATA : BYTE_ARRAY := (16#AB#, 16#CD#, 16#EF# ,16#01#, 16#00#, 16#00#);
    -- The input data stream for which the CRC is to be computed.

begin

    CHK:= 16#0000FFFF#;

        for I in 1 ..  MAX_NUMBER_OF_INPUT_BYTE loop
        -- the input data checksum is obtained

            CHK := MEM_CRC.GET_CRC(INDATA(I), CHK);

        end loop;

        INDATA(MAX_NUMBER_OF_INPUT_BYTE + 1)
            := CHECKSUM_TO_CHECKSUM(CHK).INDATA2;

        INDATA(MAX_NUMBER_OF_INPUT_BYTE + 2)
            := CHECKSUM_TO_CHECKSUM(CHK).INDATA1;

        -- Most significant bytes of the checksum is appended to the
```

```
            -- bytes array


        CHK := 16#0000FFFF#;

        for I in 1 ..  MAX_NUMBER_OF_INPUT_BYTE + 2 loop
        -- the input data plus the checksum is evaluated, the
        -- resultant checksum must be 0

            CHK := MEM_CRC.GET_CRC(INDATA(I), CHK);
        end loop;

end MAIN;
```

10.2 RAM SCRUBBING

10.2.1 Ada Package Body "MEMORY_CHECKSUM_REPORT"

The following Ada package generates an "On-Request" Telemetry Source Packet containing a CRC calcuated over a region of RAM described by an Address State number, a logical start address, a length (in words) and a flag indicating whether it is logical operand or instruction memory.

The operation has to be capable of computing the CRC for code ($ISECT) control sections as well as operand ($DATA and $CONS) control sections. However, since the programme executing in the 1750 cannot access physical memory directly, but has always to go via the MMU, all accesses to operands, e.g. to get the value of the next word in the "buffer" to be CRCed, are made through logical operand pages. The problem comes when the memory to be CRCed contains code. In that case, the "buffer" (which as far as the programme and the microprocessor are concerned is a set of operands) has to be made to "overlay" executable code. The way that this is done in MEMORY_CHECKSUM_-REPORT as follows:

If ISECT is TRUE, i.e. the caller is saying that the parameter START_ADDRESS is the logical address in a page of *instructions*, then before the call to the CRC computation algorithm (CRC.GET_CRC), the operation reads the content of the corresponding MMU instruction register whose logical page number is the same as the required instruction page, and copies it into the MMU operand page register whose page number is equal to that of the instruction page. Thus, the operand page points to the physical page containing the code control section to be CRCed. That allows the CRC algorithm to read that physical page as if it were a page of operands. After computing the CRC, the content of the operand page is restored so that it once again maps to operands.

This illustrates how what appear to be relatively simple jobs, such as reading areas of memory, become very tricky to implement when the memory is mapped via the MMU. In this case it is especially complicated because the software wants to do something which, in principal, the HW is designed to prevent, i.e. treat a code section as if it were an array.

```
--*******************************************************************
--   COPYRIGHT      :   GMV S.A., TRES CANTOS, MADRID
--   PROJECT        :   INTEGRAL DPE CSSW
--   FILE NAME      :   memory_checksum_report.ada
--   TYPE           :   PACKAGE BODY
--   UNIT NAME      :   MEMORY_CHECKSUM_REPORT
--
--   CI.-NO.        :   I323220
--
```

```
--  COMPILER       :  TLDADA v5.7.5
--  PROCESSOR      :  1750A
--
--  FUNCTION       :  This object represents the Memory Checksum
--                 :  Report TM(6,3).
--
--_**********************************************************************
--
--  SCCS VERSION NUMBER & DATE :  2.1 01/13/98
--
--_**********************************************************************

with RTX;
with CRC;
with ICB;
with SW_DRIVERS;
with CSSW_COMMON_TYPES;
with MACHINE_CODE;

with UNCHECKED_CONVERSION;
with SYSTEM;

package body MEMORY_CHECKSUM_REPORT is

   package CIT renames CSSW_IF_TYPES;
   package CCT renames CSSW_COMMON_TYPES;

   REPORT : ICB.T_ON_REQUEST_DATA_FIELD;
   -- The data field of the report TM. N.B. This variable is assumed to be
   -- visible in ALL address states, i.e. the package body of MEMORY_-
   -- CHECKSUM_REPORT must be put into a node which is seen across all AS.


   procedure GENERATE
               ( ADDRESS_STATE : in CSSW_IF_TYPES.T_ADDRESS_STATE;
                 START_ADDRESS : in CSSW_IF_TYPES.T_LOGICAL_ADDRESS;
                 LEN           : in INTEGER;
                 ISECT         : in BOOLEAN := FALSE) is
   --_****************************************************************
   -- PURPOSE : Generate a Memory Checksum Report TM (6,3) containing
   --         : the CRC computed for the specified range of logical memory.
   --_****************************************************************
      function "="(LEFT, RIGHT : CCT.T_UNSIGNED_4) return BOOLEAN
         renames CCT."=";
      -- Makes the equality operator for CIT.T_ADDRESS_STATE
      -- visible.

      function CVT is new UNCHECKED_CONVERSION (CIT.T_LOGICAL_ADDRESS,
                                                INTEGER);

      START_ADDR : INTEGER := CVT(START_ADDRESS);

      function CVT is new UNCHECKED_CONVERSION (CCT.T_UNSIGNED_16,
                                                INTEGER);

      type T_MID is
         record
            SPARE : CCT.T_UNSIGNED_12;
            AS    : CIT.T_ADDRESS_STATE;
         end record;
      for T_MID use
         record
            SPARE at 0 range 0..11;
            AS    at 0 range 12..15;
         end record;
      -- The definition of the MID (Memory ID) parameter in the
      -- TC(6,3) Calculate Memory Checksum.

      function CVT is new UNCHECKED_CONVERSION (T_MID, INTEGER);

      TIME : SW_DRIVERS.T_OBT;
      -- Used to store the time read from the Freeze 2 register.

      type T_PR_CMD is (WOPR, RIPR, ROPR);
```

```
        for T_PR_CMD use (WOPR => 16#52#,
                          RIPR => 16#D1#,
                          ROPR => 16#D2#);
        -- The values defined in MIL-STD-1750A, pages 31 and 33, for the
        -- MSbyte of the 8-bit command field of the XIO instructions used
        -- by this procedure.

        type T_PR_XIO_CMD is
            record
                CMD : T_PR_CMD;
                GRP : CCT.T_UNSIGNED_4;
                PAG : CCT.T_UNSIGNED_4;
            end record;
        for T_PR_XIO_CMD use
            record
                CMD at 0 range 0..7;
                GRP at 0 range 8..11;
                PAG at 0 range 12..15;
            end record;
        -- The composition of an XIO instruction used to read from/write to
        -- MMU page registers.

        type T_LOG_ADDR is
            record
                LPA    : CCT.T_UNSIGNED_4;
                OFFSET : CCT.T_UNSIGNED_12;
            end record;
        for T_LOG_ADDR use
            record
                LPA    at 0 range 0..3;
                OFFSET at 0 range 4..15;
            end record;
        -- The composition of a 16-bit logical address, i.e. split into
        -- 4-bit Logical Page Address (LPA) and 12-bit "address of word
        -- within 4K page".

        function CVT is new UNCHECKED_CONVERSION (INTEGER,
                                                  T_LOG_ADDR);

        LOG_ADDR : T_LOG_ADDR := CVT(START_ADDR);
        -- The start address, split into 4-bit Logical Page Address (LPA)
        -- and 12-bit "address of word within 4K page".

        SAVED_OPR : SYSTEM.UNSIGNED;
        -- The recorded contents of an MMU Operand Page Register.

        SAVED_R14 : SYSTEM.UNSIGNED;
        -- Save R14

        XIO_OPERATION : T_PR_XIO_CMD;
        -- The input/output operation to be specified in an XIO instruction.

    begin -- procedure GENERATE

        RTX.ENTER_CRITICAL_REGION;

        REPORT(REPORT'FIRST)   := CVT((SPARE => 0,
                                       AS    => ADDRESS_STATE));
        REPORT(REPORT'FIRST+1) := START_ADDR;
        REPORT(REPORT'FIRST+2) := BOOLEAN'POS(ISECT);
        REPORT(REPORT'FIRST+3) := LEN;

        SW_DRIVERS.READ_LAST_FROZEN_TIME (FROZEN_OBT => TIME);
        -- Get the time at which the acquisition of the source data, i.e.
        -- the CRC generation, began.

        if ADDRESS_STATE /= 0 then

            -- The compiler has already done a "L 2,ADDRESS_STATE,15", i.e
            -- R2 is equal to the input parameter ADDRESS_STATE.

            MACHINE_CODE.ANDM (RA   => MACHINE_CODE.R2,
                               DATA => 16#0F#);
            -- Mask out all bits except the 4 LSB which contain the
```

```
       -- address state.

       MACHINE_CODE.XIO (RA  => MACHINE_CODE.R2,
                         CMD => MACHINE_CODE.WSW);
       -- Write to the Processor Status Word to change the AS field
       -- to the new AS.
   end if;

   if ISECT then

       -- If the CRC is to be generated for instruction memory, the
       -- contents of the logical instruction page will have to be
       -- copied,temporarily, into the same number logical operand page,
       -- so that the operand page points to the physical page of
       -- instructions,i.e. we have to trick the processor so that it is
       -- able to do operand fetches to read instructions as if they were
       -- data.

       MACHINE_CODE.ST (RA   => MACHINE_CODE.R14,
                        ADDR => SAVED_R14'ADDRESS,
                        RX   => MACHINE_CODE.SP);
       -- Save R14 (which will be used as a work register).

       XIO_OPERATION := (CMD => ROPR,
                         GRP => ADDRESS_STATE,
                         PAG => LOG_ADDR.LPA);
       -- N.B. This changes R2,R3 and R4

       MACHINE_CODE.L (RA   => MACHINE_CODE.R14,
                       ADDR => XIO_OPERATION'ADDRESS,
                       RX   => MACHINE_CODE.SP);
       -- Load the input/output operation to be performed into R14.

       XIO_OPERATION := (CMD => RIPR,
                         GRP => ADDRESS_STATE,
                         PAG => LOG_ADDR.LPA);
       -- Prepare the next XIO operation. Do this before using R2 again.

       MACHINE_CODE.XIO (RA  => MACHINE_CODE.R2,
                         CMD => 0,
                         RX  => MACHINE_CODE.R14);
       -- Load R2 with the contents of the Operand page register which
       -- will be overwritten.

       MACHINE_CODE.ST (RA   => MACHINE_CODE.R2,
                        ADDR => SAVED_OPR'ADDRESS,
                        RX   => MACHINE_CODE.SP);
       -- Save the old OPR contents.

       MACHINE_CODE.L (RA   => MACHINE_CODE.R14,
                       ADDR => XIO_OPERATION'ADDRESS,
                       RX   => MACHINE_CODE.SP);
       -- Load the input/output operation to be performed into R14.

       XIO_OPERATION := (CMD => WOPR,
                         GRP => ADDRESS_STATE,
                         PAG => LOG_ADDR.LPA);
       -- Prepare the next XIO operation. Do this before using R2 again.

       MACHINE_CODE.XIO (RA  => MACHINE_CODE.R2,
                         CMD => 0,
                         RX  => MACHINE_CODE.R14);
       -- Load R2 with the contents of the Instruction page register
       -- which will be copied into the Operand page register.

       MACHINE_CODE.L (RA   => MACHINE_CODE.R14,
                       ADDR => XIO_OPERATION'ADDRESS,
                       RX   => MACHINE_CODE.SP);
       -- Load the input/output operation to be performed into R14.

       MACHINE_CODE.XIO (RA  => MACHINE_CODE.R2,
                         CMD => 0,
                         RX  => MACHINE_CODE.R14);
       -- Copy the Instruction page register contents into the
```

```
               -- Operand page register for the same logical page in the same
               -- address state.

               MACHINE_CODE.L (RA   => MACHINE_CODE.R14,
                               ADDR => SAVED_R14'ADDRESS,
                               RX   => MACHINE_CODE.SP);
               -- Restore R14.

         end if;

         REPORT(REPORT'FIRST+4) := CVT(CRC.GET_CRC (ADDRESS => START_ADDRESS,
                                                    LENGTH  => LEN));
         -- N.B. The call to CRC.GET_CRC must NOT use a transit routine,
         -- otherwise the AS will be switched again (by the transit routine).
         -- Also, the stack of the calling process must be visible at the
         -- same logical address in all AS.

         if ISECT then
            MACHINE_CODE.ST (RA   => MACHINE_CODE.R14,
                             ADDR => SAVED_R14'ADDRESS,
                             RX   => MACHINE_CODE.SP);
            -- Save R14 (which will be used as a work register).

            XIO_OPERATION := (CMD => WOPR,
                              GRP => ADDRESS_STATE,
                              PAG => LOG_ADDR.LPA);
            -- Prepare the next XIO operation. Do this before using R2 again.

            MACHINE_CODE.L (RA   => MACHINE_CODE.R2,
                            ADDR => SAVED_OPR'ADDRESS,
                            RX   => MACHINE_CODE.SP);
            -- Load the saved OPR contents into R2.

            MACHINE_CODE.L (RA   => MACHINE_CODE.R14,
                            ADDR => XIO_OPERATION'ADDRESS,
                            RX   => MACHINE_CODE.SP);
            -- Load the input/output operation to be performed into R14.

            MACHINE_CODE.XIO (RA  => MACHINE_CODE.R2,
                              CMD => 0,
                              RX  => MACHINE_CODE.R14);
            -- Restore the old contents of the OPR.

            MACHINE_CODE.L (RA   => MACHINE_CODE.R14,
                            ADDR => SAVED_R14'ADDRESS,
                            RX   => MACHINE_CODE.SP);
            -- Restore R14.
         end if;

         if ADDRESS_STATE /= 0 then
            MACHINE_CODE.XORR (RA => MACHINE_CODE.R2,
                               RB => MACHINE_CODE.R2);
            -- R2 := 0

            MACHINE_CODE.XIO (RA  => MACHINE_CODE.R2,
                              CMD => MACHINE_CODE.WSW);
            -- Write to the Processor Status Word to change the AS field to
            -- the AS 0.
         end if;

         ICB.PUT_ON_REQUEST_TM (TMSP_TYPE       => 6,
                                TMSP_SUBTYPE    => 3,
                                GENERATION_TIME => TIME,
                                REQUESTED_DATA  => REPORT);

         RTX.LEAVE_CRITICAL_REGION;

      end GENERATE;

begin
   for I in REPORT'FIRST..REPORT'LAST loop
      REPORT(I) := 0;
   end loop;
   -- Initialise the entire source data with zeros.
```

```
end MEMORY_CHECKSUM_REPORT;
```

10.2.2 Ada Package "RAM_SCRUB" and main procedure "IDLE_PROCESS"

```
package RAM_SCRUB is

   type T_PHYSICAL_PAGE is range 00..16#1F#;
   -- For our purpose of measuring the time needed to scrub
   -- 256 kbytes of RAM, i.e. 64 physical pages.

   procedure MAP_PAGE (TO : in T_PHYSICAL_PAGE);

   procedure SCRUB_PAGE;

end RAM_SCRUB;

with MACHINE_CODE;
use MACHINE_CODE;
with SYSTEM;

package body RAM_SCRUB is

   PAGE_D : array (0..16#FFF#) of INTEGER;
   for PAGE_D use at 16#D000#;
   -- An array that fills all of logical operand page D, reserved for
   -- RAM scrubbing.

   VALUE : INTEGER;
   -- A value to be read from each word of logical operand page D.

   procedure MAP_PAGE (TO : in T_PHYSICAL_PAGE) is
   -- *****************************************************
   -- PURPOSE: Modifies the reserved operand page register in
   --          the MMU group for AS0 so that it maps to the
   --          specified physical page.
   -- *****************************************************
   begin
      PSHM    ( R14, R14 );
      -- Save R14 on the stack.

      LIM ( R14, SYSTEM.ADDRESS'(16#520D#) );
      -- Load R14 with the WOPR instruction that will modify the logical
      -- operand page D in AS0, reserved for RAM scrubbing, such that it
      -- will map to the physical page where the word to be read resides.

      XIO ( R3, 0, R14 ); -- WOPR, Operand=TO=R3
      -- Connect the specified physical page to the reserved logical page
      -- 0.Do

      POPM ( R14, R14 );
      -- Restore R14 from the stack.

   end MAP_PAGE;

   procedure SCRUB_PAGE is
   -- *****************************************************
   -- PURPOSE: Reads every word of the logical operand page D.
   -- *****************************************************
   begin
      for OFFSET in PAGE_D'RANGE loop
         VALUE := PAGE_D (OFFSET);
      end loop;
   end SCRUB_PAGE;

end RAM_SCRUB;

with RAM_SCRUB;

procedure IDLE_PROCESS is
```

```
--  ********************************************************
-- PURPOSE: Scrubs 128Kwords of physical memory.
--  ********************************************************
begin

    for PHYS_PAGE_TO_SCRUB in RAM_SCRUB.T_PHYSICAL_PAGE loop

        RAM_SCRUB.MAP_PAGE (TO => PHYS_PAGE_TO_SCRUB);

        RAM_SCRUB.SCRUB_PAGE;

    end loop;

end IDLE_PROCESS;
```

11. ERC32 SOURCE CODE

11.1 CRC-CCITT CYCLIC REDUNDANCY CHECK

11.1.1 Ada package specification "TYPES"

```
with UNCHECKED_CONVERSION;

package TYPES is

   type BYTE                 is range 0 .. 2**8 - 1;
   -- byte type of 8 bits

   for BYTE'SIZE use 8;

   type WORD                 is range 0 .. 2**31 - 1;
   -- byte type of 32 bits

   for WORD'SIZE use 32;

   type WORD_LOGICOS is array(0..31) of BOOLEAN;
   -- representation of the word type as a boolean array

   type BITS_LOGICOS is array(0..7) of BOOLEAN;
   -- representation of the byte type as a boolean array

   pragma PACK(WORD_LOGICOS);
   for WORD_LOGICOS'SIZE use 32;

   pragma PACK(BITS_LOGICOS);
   for BITS_LOGICOS'SIZE use 8;

   function WORD_TO_BOOLEAN is new
       UNCHECKED_CONVERSION(WORD, WORD_LOGICOS);
   -- converts an word to a boolean array

   function BOOLEAN_TO_WORD is new
      UNCHECKED_CONVERSION(WORD_LOGICOS, WORD);
   -- converts a boolean array to word type

   function BYTE_TO_BOOLEAN is new
       UNCHECKED_CONVERSION(BYTE, BITS_LOGICOS);
   -- converts an byte to a boolean array

   function BOOLEAN_TO_BYTE is new
      UNCHECKED_CONVERSION(BITS_LOGICOS, BYTE);
   -- converts a boolean array to byte type

   type CHECKSUM is
      record
        INDATA4: TYPES.BYTE;
        INDATA3: TYPES.BYTE;
        INDATA2: TYPES.BYTE;
        INDATA1: TYPES.BYTE;
        -- Most significant bytes
      end record;

   for CHECKSUM use
      record
        INDATA4 at 0 range 0 .. 7;
        INDATA3 at 0 range 8 .. 15;
        INDATA2 at 0 range 16 .. 23;
        INDATA1 at 0 range 24 .. 31;
     end record;

  function CHECKSUM_TO_CHECKSUM is new
     UNCHECKED_CONVERSION (TYPES.WORD, CHECKSUM);
```

```
  MAX_NUMBER_OF_INPUT_BYTE:  constant POSITIVE := 4;
  -- number of input byte

end TYPES;
```

11.1.2 Ada package body "MEM_CRC"

```
with TYPES;

package MEM_CRC is
-- package specification of the function that calculates the
-- CRC-CCITT cheksum

   function GET_CRC (DATA     : in TYPES.BYTE;
                     SYNDROME : in TYPES.WORD)return TYPES.WORD;
end MEM_CRC;

package body MEM_CRC is

   function GET_CRC(DATA     : in TYPES.BYTE;
                    SYNDROME : in TYPES.WORD)
   return TYPES.WORD is

      AUX_SYNDROME: TYPES.WORD;
      AUX_DATA: TYPES.WORD;
      DATA_EXP22: TYPES.BYTE:= 16#02#;
      DATA_EXP2: TYPES.WORD:= 16#0002#;
      DATA_EXP8: TYPES.WORD:= 16#0100#;
      DATA_NULL: TYPES.WORD_LOGICOS:= (others => true);

      begin

         AUX_SYNDROME:= SYNDROME;
         AUX_DATA:= WORD(DATA);

         for INDEX in 1 ..8 loop

            if BOOLEAN_TO_WORD
                  (
                   (WORD_TO_BOOLEAN(AUX_DATA) and
                    WORD_TO_BOOLEAN(16#0080#)
                   )
                   xor

                   (WORD_TO_BOOLEAN
                      (BOOLEAN_TO_WORD
                         (WORD_TO_BOOLEAN(AUX_SYNDROME) and
                          WORD_TO_BOOLEAN(16#8000#)
                         )
                         / DATA_EXP8
                      )
                   )
                  ) /= WORD(0) then

               AUX_SYNDROME :=
                   BOOLEAN_TO_WORD
                     ( (WORD_TO_BOOLEAN (AUX_SYNDROME * DATA_EXP2) xor
                        WORD_TO_BOOLEAN (16#11021#)
                       ) and WORD_TO_BOOLEAN(16#FFFF#)
                     );
            else
               AUX_SYNDROME :=
                   BOOLEAN_TO_WORD
                     (WORD_TO_BOOLEAN(AUX_SYNDROME * DATA_EXP2) and
                      WORD_TO_BOOLEAN(16#FFFF#)
                     );

            end if;

            AUX_DATA:= AUX_DATA * DATA_EXP2;
```

```
        end loop;

        return AUX_SYNDROME;

    end GET_CRC;

end MEM_CRC;
```

11.1.3 Ada main entry point procedure "MAIN"

```
with TYPES;
use TYPES;
with MEM_CRC;

procedure MAIN is

   CHK : TYPES.WORD;
   -- checksum of the input data

   INDATA: array (1.. MAX_NUMBER_OF_INPUT_BYTE + 2) of TYPES.BYTE:= (
           16#AB#, 16#CD#,16#EF# ,16#01#, 16#00#, 16#00#) ;
   -- input bytes to be tested

   begin

      CHK:= 16#0000FFFF#;

      for I in 1 ..  MAX_NUMBER_OF_INPUT_BYTE loop
      -- the input data checksum is obtained

         CHK:= MEM_CRC.GET_CRC(INDATA(I), CHK);
      end loop;

      INDATA(MAX_NUMBER_OF_INPUT_BYTE + 1)
         := CHECKSUM_TO_CHECKSUM(CHK).INDATA2;


      INDATA(MAX_NUMBER_OF_INPUT_BYTE + 2)
         := CHECKSUM_TO_CHECKSUM(CHK).INDATA1;
      -- Most significant bytes of the checksum is appended to the bytes
      -- array

      CHK:= 16#0000FFFF#;

      for I in 1 ..  MAX_NUMBER_OF_INPUT_BYTE + 2 loop
      -- the input data plus the checksum is evaluated, the resultant
      -- checksum must be 0

         CHK:= MEM_CRC.GET_CRC(INDATA(I), CHK);
      end loop;

end MAIN;
```

11.2 RAM SCRUBBING

```
with SYSTEM;
with UNCHECKED_CONVERSION;

procedure RAM_SCRUB is

    --*****************************************************************
    --  PURPOSE : Ram scrubbing
    --*****************************************************************

    RAM_START_ADD: constant SYSTEM.ADDRESS
       := SYSTEM.VALUE("16#02000000#");

    type T_INTEGER_32 is range -2**31.. 2**31 -1;
    for T_INTEGER_32'size use 32;

    type T_RAM_POOL is array(integer range 1..2**16) of T_INTEGER_32;

    type T_RAM_POOL_PTR is access T_RAM_POOL;

    function CVT is new UNCHECKED_CONVERSION (SYSTEM.ADDRESS,
    T_RAM_POOL_PTR);

    function "=" (LEFT, RIGHT : SYSTEM.ADDRESS) return BOOLEAN
       renames SYSTEM."=";

    RAM_PTR:T_RAM_POOL_PTR:= CVT(RAM_START_ADD);

    MEMORY_WORD: T_INTEGER_32;

begin

  for I in T_RAM_POOL'RANGE loop

     MEMORY_WORD:=  RAM_PTR(I);
  end loop;

end RAM_SCRUB;
```

12. - TLD/1750 PERFORMANCES RESULTS

12.1 PERFORMANCE RESULTS OF THE CRC-CCITT ALGORITHM

```
Log file started

TLD Ada Symbolic Debugger  TLDdbg/SOLARIS  V-5.4.0  21-JAN-1998  15:44:45
(c) TLD Systems, Ltd., 1997


1=>
1=> set mode line
1=> -- NO window oriented output.
1=>
1=> attach tldenv
TLD Environment Controller  tldenv/SOLARIS  V-3.7.0  (c) TLD Systems, Ltd., 1997
Processor MAIN_CONTROLLER (2) successfully attached  (Cluster TLDENV)
Processor UNIT1 (3) successfully attached  (Cluster TLDENV)
Processor MS1750_CONTROLLER (4) successfully attached  (Cluster TLDENV)
Processor CPU1 (5) successfully attached  (Cluster TLDENV)
Processor TLDENV (1) successfully attached  (Cluster TLDENV)
1=>
1=> load ../main.ldm  -- Our loadable code.
MAIN_CONTROLLER (2):
------------------------------------------------------------
            ************* HW SIMULATOR *************

           GMV -- DPE HW Simulation Environment 1.1
     Copyright 1997, GMV S.A., Tres Cantos, Madrid, Spain.

           TLD Environment Interface V-3.5.3
------------------------------------------------------------
MS1750_CONTROLLER  (4): TLD MS1750 Simulator   ms1750/SOLARIS  V-3.7.0   (c)  TLD
Systems, Ltd., 1997
UNIT1 (3): External register overlay has been established ...
MAIN_CONTROLLER (2): Processor MAIN_CONTROLLER (2) successfully configured
MAIN_CONTROLLER (2): Processor UNIT1 (3) successfully configured
Loading "/home/users/jjqg/trabajo/erc32/crc/main.ldm" to CPU1 (5)
UNIT1 (3): ---> RTC Initialised
UNIT1 (3): ---> WATCHDOG Initialised
UNIT1 (3): ---> OBT Initialised
UNIT1 (3): ---> CDMU initialised
UNIT1 (3): ---> RELAYS Initialised
UNIT1 (3): ---> HSSL Initialised
UNIT1 (3): ---> ANALOG LINES Initialised
UNIT1 (3): ---> LSSL Initialised
UNIT1 (3): ---> EDAC Initialised
UNIT1 (3): ---> INPUT initialised
CPU1        (5):         Instruction     description       file        used:
/home/users/jjqg/trabajo/erc32/crc/Sim/ms1750a.idf
CPU1 (5): INTERRUPTS have been overlayed to 16#10000000#
MS1750_CONTROLLER (4): Processor MS1750_CONTROLLER (4) successfully configured
MS1750_CONTROLLER (4): Processor CPU1 (5) successfully configured
Loading "/home/users/jjqg/trabajo/erc32/crc/main.trb"
Loading "/home/users/jjqg/trabajo/erc32/crc/main.dbg"

---- SOURCE05 --  /export/home/tldacs/97Sa025/rtx/kernel/rtx_start.mac ---------
    66:     |
    67:     | RTXSTART        ENTER   []
->  68: 410|                  XIO     R1,DSBL               ; Make sure interru
    69: 411|                  XIO     R1,TAH                ; Halt timer A
    70: 412|                  XIO     R1,TBH                ; Halt timer B
    71: 413|                  XIO     R1,RCFR               ; Clear the fault r
    72: 414|                  SR      R1,R1                 ; Need a zero value
    73: 415|                  XIO     R1,SMK                ; Clear interrupt m
--------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count: 0               Clock: 0.000_000_000
R0: 0000  0000  0000  0000  0000  0000  0000  0000  IC: 0000 MK: 0000 SW: 0000
R8: 0000  0000  0000  0000  0000  0000  0000  0000  PI: 0000 TA: 0000 TB: 0000
CPU1 (5): Start at RTX_START.RTXSTART._410: XIO        R1,DSBL
1=>
1=> set trace/exception/continue/calls
```

```
EXCEPTION Trace:
  All UnHandled:  CONTINUE execution, SHOW CALLS
  All   Handled:  CONTINUE execution, SHOW CALLS
1=>
1=>
1=> set proc 5;
Current Processor is CPU1 (5)
1=>
1=> b main
Breakpoint  1  at  MAIN._4  ( physical 16#01C7# )
       Target Id:  98
        Action:  HALT execution
1=> -- Main program
1=>
1=> b main._10 do ( g/w )
Breakpoint  2  at  MAIN._10  ( physical 16#01D7# )
       Target Id:  97
        Action:  HALT execution
      Do List:  g/w
1=> b main._13 do ( g/w )
Breakpoint  3  at  MAIN._13  ( physical 16#01F2# )
       Target Id:  96
        Action:  HALT execution
      Do List:  g/w
1=>
1=> b main._16 do ( g/w )
Breakpoint  4  at  MAIN._16  ( physical 16#01FD# )
       Target Id:  95
        Action:  HALT execution
      Do List:  g/w
1=> b main._19 do ( g/w )
Breakpoint  5  at  MAIN._19  ( physical 16#0213# )
       Target Id:  94
        Action:  HALT execution
      Do List:  g/w
1=>
1=> set proc 1;
Current Processor is TLDENV (1)
1=>
1=>
1=> g/w

---- SOURCE05 --  /home/users/jjqg/trabajo/erc32/crc/main.ada -----------------
    19:     | with MEM_CRC;
    20:     | -- with TEXT_IO;
->  21:    4| procedure MAIN is
    22:     |
    23:     |     CHK : TYPES.WORD;
    24:     |     -- checksum of the input data
    25:     |
    26:     |     type BYTE_ARRAY is array (1.. MAX_NUMBER_OF_INPUT_BYTE + 2) of
------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count:*664            Clock:*0.000_585_700
R0:*5840 *1E01 *1F3D  0000 *1F94 *1612  0000  0000  IC:*01C7 MK:*5840 SW:*4000
R8: 0000  0000  0000  0000 *1E00 *1F3D  0000 *FEFE  PI: 0000 TA: 0000 TB:*0002
CPU1 (5): Breakpoint (1) hit at MAIN._4: LIM       R2,16#FFDB#
1=> -- Stopped at procedure main
1=>
1=> g/w

---- SOURCE05 --  /home/users/jjqg/trabajo/erc32/crc/main.ada -----------------
    39:    9|     CHK:= 16#0000FFFF#;
    40:     |
->  41:   10|     for I in 1 ..  MAX_NUMBER_OF_INPUT_BYTE loop
    42:     |     -- the input data checksum is obtained
    43:     |
    44:   11|         CHK := MEM_CRC.GET_CRC(INDATA(I), CHK);
    45:     |
    46:   12|     end loop;
------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count:*689            Clock:*0.000_623_000
R0:*0000 *FFFF *FEFB  0000  1F94  1612  0000  0000  IC:*01D7 MK: 5840 SW: 4000
R8: 0000  0000  0000  0000 *1BA2  1F3D  0000 *FED9  PI: 0000 TA: 0000 TB:*0003
CPU1 (5): Breakpoint (2) hit at MAIN._10: STC       16#1#,16#22#,R15
```

```
---- SOURCE05 --   /home/users/jjqg/trabajo/erc32/crc/main.ada ------------------
      50:      |  --      TEXT_IO.NEW_LINE;
      51:      |
->    52:   13 |      INDATA(MAX_NUMBER_OF_INPUT_BYTE + 1):= CHECKSUM_TO_CHECKSUM(
      53:      |
      54:      |      --BYTE_IO.PUT(INDATA(MAX_NUMBER_OF_INPUT_BYTE + 1), BYTE'WID
      55:      |      --TEXT_IO.NEW_LINE;
      56:      |
      57:   14 |      INDATA(MAX_NUMBER_OF_INPUT_BYTE + 2):= CHECKSUM_TO_CHECKSUM(
---------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count:*11322             Clock:*0.009_810_300
R0: 0000 *A07E *001E *0001 *0000 *B457  0000 *0008  IC:*01F2 MK: 5840 SW:*2000
R8: 0000  0000  0000  0000 *0000  1F3D  0000  FED9  PI: 0000 TA: 0000 TB:*005E
CPU1 (5): Breakpoint (3) hit at MAIN._13: XORR      R2,R2

---- SOURCE05 --   /home/users/jjqg/trabajo/erc32/crc/main.ada ------------------
      63:   15 |      CHK:= 16#0000FFFF#;
      64:      |
->    65:   16 |      for I in 1 ..  MAX_NUMBER_OF_INPUT_BYTE + 2 loop
      66:      |      -- the input data plus the checksum is evaluated, the result
      67:      |
      68:   17 |        CHK:= MEM_CRC.GET_CRC(INDATA(I), CHK);
      69:   18 |      end loop;
      70:      |
---------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count:*11330             Clock:*0.009_816_100
R0: 0000 *FFFF *007E  0001  0000  B457  0000  0008  IC:*01FD MK: 5840 SW:*4000
R8: 0000  0000  0000  0000  0000  1F3D  0000  FED9  PI: 0000 TA: 0000 TB:*005F
CPU1 (5): Breakpoint (4) hit at MAIN._16: STC       16#1#,16#22#,R15

---- SOURCE05 --   /home/users/jjqg/trabajo/erc32/crc/main.ada ------------------
      71:      |  --      TEXT_IO.PUT("CHECKSUM VALUE OF THE INPUT BYTES PLUS THE CHE
      72:      |  --      WORD_IO.PUT(CHK, TYPES.WORD'WIDTH, 16);
      73:      |  --      TEXT_IO.NEW_LINE;
      74:      |
->    75:   19 | end MAIN;
      76:      |
      77:      |
      78:      |
---------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count:*22541             Clock:*0.019_515_600
R0: 0000 *0000 *0020 *007E  0000 *7E00  0000  0008  IC:*0213 MK: 5840 SW:*2000
R8: 0000  0000  0000  0000  0000  1F3D  0000  FED9  PI: 0000 TA: 0000 TB:*00C0
CPU1 (5): Breakpoint (5) hit at MAIN._19: LIM       R15,16#25#,R15

---- SOURCE05 --   /export/home/tldacs/97Sa025/rtx/common/rtxsim.mac ------------
     220:      |               EXPORT   A_TARGET_HALT
     221:      | A_TARGET_HALT  ENTER    []
->   222:  915 |               BPT                         ; We are dead
     223:  916 |               BR       A_TARGET_HALT       ; Make sure we are
     224:  918 |               RETURN   A_TARGET_HALT
     225:      |
     226:      |   END  ; RTX_HALT
     227:      |
---------------------------------------------------------------------------------
Proc   5  HALTED      Inst Count:*22622             Clock:*0.019_577_300
R0: 0000  0000 *0022 *1D52 *0010 *0000 *FEF4  0008  IC:*024F MK: 5840 SW:*1000
R8: 0000  0000 *1EED  0000 *1EEC  1F3D  0000 *FEF3  PI: 0000 TA: 0000 TB: 00C0
CPU1 (5): Program MAIN (Id=1) terminated at RTX_HALT.A_TARGET_HALT._915: BPT
1=>
1=> exit
=> q
CPU1 (5): EXECUTION SUMMARY of CPU1 (5):
CPU1 (5):   - Host CPU Execution Time: 1.150_000_000 seconds
CPU1 (5):   - Target Simulation Time : 0.019_577_300
CPU1 (5):   - Target Operation Count : 22622
***** Session Time of processor CPU1 (5) is 17.0 seconds *****
***** Session Time of processor MS1750_CONTROLLER (4) is 17.0 seconds *****
UNIT1 (3): EXECUTION SUMMARY of UNIT1 (3):
UNIT1 (3):   - Host CPU Execution Time: 0.000_000_000 seconds
UNIT1 (3):   - Target Simulation Time : 0.019_531_250
***** Session Time of processor UNIT1 (3) is 17.0 seconds *****
***** Session Time of processor MAIN_CONTROLLER (2) is 17.0 seconds *****
```

```
***** Session Time of processor TLDENV (1) is 22.0 seconds *****

Exiting Debugger.

Total Debug Session Time is 51.0 seconds
```

12.2 PERFORMANCE RESULTS FOR RAM SCRUBBING 128KWORDS

```
Log file started

TLD Ada Symbolic Debugger  TLDdbg/SOLARIS  V-5.4.2  26-JAN-1998  13:45:23
(c) TLD Systems, Ltd., 1997

=> attach tldenv
TLD Environment Controller  tldenv/SOLARIS  V-3.7.0  (c) TLD Systems, Ltd., 1997
Processor TLDENV (1) successfully attached  (Cluster TLDENV)
MAIN_CONTROLLER (2):
-------------------------------------------------------------
           ************* HW SIMULATOR *************

            GMV -- DPE HW Simulation Environment 1.2
      Copyright 1998, GMV S.A., Tres Cantos, Madrid, Spain.

            TLD Environment Interface V-3.5.3
-------------------------------------------------------------
MS1750_CONTROLLER (4): TLD MS1750 Simulator   ms1750/SOLARIS   V-3.7.0   (c) TLD
Systems, Ltd., 1997
UNIT1 (3): External register overlay has been established ...
MAIN_CONTROLLER (2): Processor MAIN_CONTROLLER (USER_DEFINED, 2) successfully
configured
UNIT1 (3): Processor UNIT1 (USER_DEFINED, 3) successfully configured
UNIT1 (3): ---> RTC Initialised
UNIT1 (3): ---> WATCHDOG Initialised
UNIT1 (3): ---> OBT Initialised
UNIT1 (3): ---> CDMU initialised
UNIT1 (3): ---> RELAYS Initialised
UNIT1 (3): ---> HSSL Initialised
UNIT1 (3): ---> ANALOG LINES Initialised
UNIT1 (3): ---> LSSL Initialised
UNIT1 (3): ---> EDAC Initialised
UNIT1 (3): ---> INPUT initialised
CPU1           (5):           Instruction           description           file           used:
/home/users/dpe/work/code/Simulator/exec/ms1750a.idf
CPU1 (5): INTERRUPTS have been overlayed to 16#10000000#
MS1750_CONTROLLER (4): Processor MS1750_CONTROLLER (MS1750, 4) successfully
configured
CPU1 (5): Processor CPU1 (MS1750, 5) successfully configured
=> l "/home/users/mmrr/proposals/erc32eval/ramscrub/For1750/idle_process"
Loading "/home/users/mmrr/proposals/erc32eval/ramscrub/For1750/idle_process.ldm" to
CPU1 (5)
Loading "/home/users/mmrr/proposals/erc32eval/ramscrub/For1750/idle_process.trb"
Loading "/home/users/mmrr/proposals/erc32eval/ramscrub/For1750/idle_process.dbg"

---- SOURCE05 --  ...s/dpe/work/code/rtx_min/min_kernel/rtx_start.mac -------
        | 62|
        | 63| RTXSTART           ENTER    []
-> 393  | 64|                    XIO    R1,DSBL                ; Disable interr
        | 65|                                                  ; N.B. Interrupt
        | 66|                                                  ; remain enabled
    396 | 67|                    XIO    R1,TAH                 ; Halt timer A
    397 | 68|                    XIO    R1,TBH                 ; Halt timer B
    398 | 69|                    XIO    R1,RCFR                ; Clear the faul
-------------------------------------------------------------------------
Proc   5  HALTED     Inst Count: 0                   Clock:*0.000_000_000
R0: 0000  0000  0000  0000  0000  0000  0000  0000  IC: 0000 MK: 0000 SW: 000
R8: 0000  0000  0000  0000  0000  0000  0000  0000  PI: 0000 TA: 0000 TB: 000
CPU1 (5): Start at RTX_START.RTXSTART._393: XIO        R1,DSBL
=> set proc 5
```

```
Current Processor is CPU1 (5)
=> b idle_process
Breakpoint  1  at  IDLE_PROCESS._30  ( physical 16#015C# )
      Target Id:  100
      Action:  HALT execution
=> set proc 1
Current Processor is TLDENV (1)
=> go/w

---- SOURCE05 --  ...sals/erc32eval/ramscrub/For1750/idle_process.ada -------
          64| with RAM_SCRUB;
          65|
->  30    66| procedure IDLE_PROCESS is
          67| -- ******************************************************
          68| -- PURPOSE: Scrubs 128Kwords of physical memory.
          69| -- ******************************************************
          70| begin
          71|
------------------------------------------------------------------------------
Proc   5  HALTED       Inst Count:*120            Clock:*0.000_101_200
R0: 0000  0000 *02BE *5840  0000  0000  0000  0000  IC:*015C MK:*5840 SW:*100
R8: 0000  0000  0000 *0415 *0285 *02F8  0000 *FEFD  PI: 0000 TA: 0000 TB: 000
CPU1 (5): Breakpoint (1) hit at IDLE_PROCESS._30: LIM       R15,16#FFFF#,R15
=> set proc 5
Current Processor is CPU1 (5)
=> s

---- SOURCE05 --  ...sals/erc32eval/ramscrub/For1750/idle_process.ada -------
          70| begin
          71|
->  32    72|    for PHYS_PAGE_TO_SCRUB in RAM_SCRUB.T_PHYSICAL_PAGE loop
          73|
    33    74|        RAM_SCRUB.MAP_PAGE (TO => PHYS_PAGE_TO_SCRUB);
          75|
    34    76|        RAM_SCRUB.SCRUB_PAGE;
          77|
------------------------------------------------------------------------------
Proc   5  HALTED       Inst Count:*121            Clock:*0.000_101_800
R0: 0000  0000  02BE  5840  0000  0000  0000  0000  IC:*015E MK: 5840 SW: 100
R8: 0000  0000  0000  0415  0285  02F8  0000 *FEFC  PI: 0000 TA: 0000 TB: 000
CPU1 (5): Stepped to IDLE_PROCESS._32: STC       16#0#,16#0#,R15
=> b _36
Breakpoint  2  at  IDLE_PROCESS._36  ( physical 16#016D# )
      Target Id:  99
      Action:  HALT execution
=> set proc 1
Current Processor is TLDENV (1)
=> go/w
---- SOURCE05 --  ...sals/erc32eval/ramscrub/For1750/idle_process.ada -------
          75|
    34    76|        RAM_SCRUB.SCRUB_PAGE;
          77|
    35    78|    end loop;
          79|
->  36    80| end IDLE_PROCESS;
          81|
          82|
------------------------------------------------------------------------------
Proc   5  HALTED       Inst Count:*1442328        Clock:*0.970_413_900
R0: 0000 *DFFF *001F *0FFF  0000  0000  0000  0000  IC:*016D MK: 5840 SW:*200
R8: 0000  0000  0000 *0000 *0000 *0FFF  0000  FEFC  PI:*0020 TA: 0000 TB:*25E
CPU1 (5): Breakpoint (2) hit at IDLE_PROCESS._36: LIM       R15,16#1#,R15
=> exit
Current Processor is TLDENV (1)
UNIT1 (3): EXECUTION SUMMARY of UNIT1 (3):
UNIT1 (3):   - Host CPU Usage Time    : 0.030_000_000 seconds
UNIT1 (3):   - Host CPU Simulation Time: 0.000_000_000 seconds
UNIT1 (3):   - Target Simulation Time  : 0.968_750_000 seconds
Processor UNIT1 (3) has been detached
***** Session Time of processor UNIT1 (3) is 4:04.0 seconds *****
Processor MAIN_CONTROLLER (2) has been detached
***** Session Time of processor MAIN_CONTROLLER (2) is 4:04.0 seconds *****
CPU1 (5): EXECUTION SUMMARY of CPU1 (5):
CPU1 (5):   - Host CPU Usage Time    : 44.720_246_000 seconds
```

```
CPU1 (5):   - Host CPU Simulation Time: 40.-1826_676_704 seconds
CPU1 (5):   - Target Simulation Time  : 0.970_413_900 seconds
CPU1 (5):   - Target Operation Count  : 1442328 instructions
Processor CPU1 (5) has been detached
***** Session Time of processor CPU1 (5) is 4:05.0 seconds *****
Unloading                      symbols                       of
"/home/users/mmrr/proposals/erc32eval/ramscrub/For1750/idle_process.trb"
Unloading                      symbols                       of
"/home/users/mmrr/proposals/erc32eval/ramscrub/For1750/idle_process.dbg"
Processor MS1750_CONTROLLER (4) has been detached
***** Session Time of processor MS1750_CONTROLLER (4) is 4:05.0 seconds *****
TLDENV (1): EXECUTION SUMMARY of TLDENV (1):
TLDENV (1):   - Host CPU Usage Time      : 0.040_000_000 seconds
TLDENV (1):   - Host CPU Simulation Time: 0.040_000_000 seconds
***** Session Time of processor TLDENV (1) is 4:08.0 seconds *****

Exiting Debugger.
```

13. ERC32 PERFORMANCES RESULTS

The performance results described in this section have been obtained using the SIS - SPARC instruction simulator 2.7.4 and the ERC32 Target Simulator from Spacebel.

13.1 PERFORMANCE RESULTS OF THE CRC-CCITT ALGORITHM

13.1.1 SIS

```
143: sis -freq 10 main.x

 SIS - SPARC intruction simulator 2.7.4,  copyright Jiri Gaisler 1995
 Bug-reports to jgais@wd.estec.esa.nl

loading main.x:
section .sec1 at 0x02020000 (57168 bytes)
section .sec2 at 0x0202e000 (4096 bytes)
serial port A on stdin/stdout
sis> go
resuming at 0x02020000
 Stopped at time 43639
sis> perf

 Cycles       :      43639
 Instructions :      28432
 Overall CPI  :       1.53

 ERC32 performance (10.0 MHz):  6.52 MOPS ( 6.51 MIPS,  0.01 MFLOPS)
 Simulated ERC32 time         :   4.36 ms
 Processor utilisation        : 100.00 %
 Real-time / simulator-time   : 1/0.00
 Simulator performance        : 28 KIPS
 Used time (sys + user)       :    0 s
```

13.1.2 Spacebel TS

The statistics associated with executing the programmes using the Spacebel TS were obtained using a Tcl/Tk script furnished by T. Vardenega (ESTEC) that processes the data generated by the simulator.

```
_____ Execution Statistics _____
cycle  count= 74524
time elapsed= 7.4524 ms @ 10.0 MHz
executed instructions= 50605
             integer= 99.1799 %
                load= 17.2924 %
               store= 6.70276 %
               float= 0.0612576 %
     raw performance= 6.79043 MOPS (CPI : 1.47263)
```

13.2 PERFORMANCE RESULTS FOR RAM SCRUBBING FOR 256 KBYTES

13.2.1 SIS

```
142: sis -freq 10 ram_scrub.x

 SIS - SPARC intruction simulator 2.7.4,  copyright Jiri Gaisler 1995
 Bug-reports to jgais@wd.estec.esa.nl

loading ram_scrub.x:
section .sec1 at 0x02020000 (58152 bytes)
section .sec2 at 0x0202f000 (4096 bytes)
serial port A on stdin/stdout
sis> go
resuming at 0x02020000
 Stopped at time 2432449
sis> perf

 Cycles       :    2432449
 Instructions :    1446600
 Overall CPI  :       1.68

 ERC32 performance (10.0 MHz):  5.95 MOPS ( 5.95 MIPS,  0.00 MFLOPS)
 Simulated ERC32 time        : 243.24 ms
 Processor utilisation       : 100.00 %
 Real-time / simulator-time  : 1/16.44
 Simulator performance       : 361 KIPS
 Used time (sys + user)      :    4 s
```

13.2.2 Spacebel TS

```
_____ Execution Statistics _____
cycle  count= 2463007
time elapsed= 246.301 ms @ 10.0 MHz
executed instructions= 1468422
             integer= 99.9919 %
                load= 27.13 %
               store= 13.426 %
               float= 0.00211111 %
     raw performance= 5.9619 MOPS (CPI : 1.67732)
```