MD0901 Boot-MIPS

Example Boot Code for MIPS Cores

Revision 1.20
APR 19 2018
Public

Contents

List of Figures

Document History

Issue	Date	Changes/Comments
1.0.178	25 Feb 2015	First Codescape-era publication.
1.0.179	25 Feb 2015	External Issue
1.0.4	01 Jun 2016	External Issue
1.0.16	02 Jun 2016	External Issue
1.5.18	03 Jun 2016	Bumping version number after revision for I6400 by CR
1.5.19	03 Jun 2016	External Issue
1.5.8	25 Jul 2017	I6500 info added by CR
1.5.9	25 Jul 2017	External Issue

1. Introduction

Boot-MIPS is example code for MIPS32® R2-5 or R6 Cores. It is intended to aid you in becoming familiar with the initialization of a MIPS Core. NOTE: There is a separate application note for the I7200,https://training.mips.com/cps_mips/Examples/MD00152-Boot-MIPS-I7200.pdf

Building Boot-MIPS results in an executable suitable for programming the flash on a Boston software development board, a SEAD development board or to download to an IASim simulator. In addition to runtime initialization, the Boot-MIPS executable includes some simple C example code that is copied from the Flash area to a RAM or Scratchpad and then executed at the end of the boot process. Only one executable is used in any particular system; where applicable, all code and non-stack C data are shared between all processing elements.

This document contains hyperlinks in blue that provide links to additional information in this document.

1.1 Terminology

An effort has been made to use terminology consistent with other MIPS documentation. Below is an explanation of terms used throughout this application note.

- M5100 and M5150: MIPS32 Single-core/single thread processors.
- TC (Thread Context): Hardware resource to support non-privileged threads of execution (interAptiv or interAptivUP only).
- VPE (Virtual Processing Element): One or more TCs bound together to work as if they were a single processor. For example, an MT Core can contain two VPEs, each with multiple TCs bound to it. Each VPE has enough independent architectural state to appear as a single processor, making each VPE capable of running a separate OS.
- interAptivUP Core: An MT Core that implements one or two VPEs.
- CPS – Coherent Processing System 1 or more Cores linked together in a unified coherent system all sharing the same L2 cache and memory bus.
- interAptivUP MIPS32 Core: Processor IP block consisting of one or more VPEs using the EVA memory architecture and an optional fixed page size TLB.
- interAptiv: MIPS32 CPS made up of one to six MT ASE cores, a L2 cache, a GIC (Global Interrupt Controller), CM (Coherence Manager), and optional IOCU (IO Coherence Unit).
- I6500: MIPS64 CPS consisting of up to 6 single core each having 1- 4 Virtual Processor contexts with support for Multiple coherent clusters of CPUs, Groups of coherent IO or co-processing functions or clusters. (NOTE: also supports the I6400 single cluster version)
- P5600: MIPS32 CPS consisting of one to six single cores, an I2, a GIC, CM, and optional IOCU.
- P6600: MIPS64 CPS consisting of one to six single cores, an I2, a GIC, CM, and optional IOCU.
- CM2/3: Coherence Manager with L2 cache. The Coherency Manager manages the system-wide coherency between the processing Cores, L2 cache and IOCU.

- CPC (Cluster Power Controller): Power domain control logic for a CPS.
- EVA: Enhanced Virtual Addressing scheme enabling software-programmable memory segments.
- FTLB: Fixed page size TLB.
- IOCU: (I/O Coherence Unit): Interface between CM and coherent I/O devices.
- I$, D$, and L2$: Primary instruction and data caches and the unified Level 2 cache.

1.2 Tools

Boot-MIPS was developed using the following hardware and software tools:

- BOSTON / Concord Software Development Boards
- CoreFPGA6A/6B™ Core daughter Cards (MD00898)
- SEAD™-3 Software Development boards (MD00737)
- Boston Software Development Boards with Concord FPGA daughter cards
- MIPS32 bit files programmed into CoreFPGA6A/6B core cards or SEAD™-3
- Codescape SDK
- Codescape Debugger or Codescape for Eclipse
- SP55 EJTAG debug adaptors
- Host PC (Windows 7) and Perl installed

This application note assumes that you are familiar with the listed tools, and that you have a functional working environment where you are able to build executables. You can use Codescape Debugger or Codescape for Eclipse to create, browse, make and debug your project. For additional information, refer to the documentation for each tool. You should also have an understanding of the MIPS architecture, MIPS assembly coding, Makefiles, linker scripts, and the C programming language.

Note: Information in this application note and the accompanying files may require modification when used with other processors, boards, or tools. Device and tool behaviour may also change as new versions are added, or features are enhanced.

1.3 Assumptions

Operating System

The instructions in this document assume the use of a Windows (7 or higher) PC. Codescape SDK and the examples can also be used on Linux PCs.

Note: Codescape SDK is for 64-bit versions of Windows. 32 and 64-bit Linux is supported.

Codescape SDK

The examples and instructions make use of the toolchain, debugger and utilities provided by Codescape SDK 8.2 or higher. It is assumed that this has been installed.

Parallel cable

A parallel cable is required to download a file to the flash memory on a BOSTON board.

If your PC lacks a parallel port, a USB – DB25 cable and DB25 gender changer can be used.

For example:

- C2G / Cables to Go 16899 USB To DB25 IEEE-1284 Parallel Printer Adapter Cable, 6 Feet
- C2G / Cables to Go 02776 DB25 Male/Male Mini Gender Changer(Coupler)

BOSTON development board

Debugging and development work can be done on simulated targets (supplied in the Codescape SDK) however these instructions assume the use of a BOSTON development board. For newer MIPS cores such as the P6600 and the I6400/I6500, the BOSTON development board has been replaced with the Boston board.

Boston development board

The Boston development board is used for the P6600 and the I6400/I6500 cores.

Perl

The makefiles supplied with Boot MIPS make use of Perl to create a file suitable for loading directly into flash memory.

2. Installation

This chapter covers the installation of the source code so you can browse the source files as you go through this document. The Codescape SDK can be used with the Codescape for Eclipse framework or Codescape Debugger. The following sections show how to setup the Boot-MIPS code for each one.

2.1. Toolchain selection

The default option in the makefiles is to compile, link etc using mips-mti-elf-xxx (where xxx is gcc, ld, objdump, objcopy or size).

The location of the toolchain is assumed to be in your PATH.

The default (as created by the Codescape SDK) toolchain paths are below

Linux default toolchain path

<install_dir>/Toolchains/mips-mti-elf/<version>

Windows default toolchain path

<install_dir>\Toolchains\mips-mti-elf\<version>

Installing the full SDK will automatically set these paths .

This script will be run automatically when logging in. If you have just installed the SDK you should logout and login again so the path will be in your environment or run the script manually to set the PATH.

Check that the correct toolchain location is on your PATH before using the supplied makefiles.

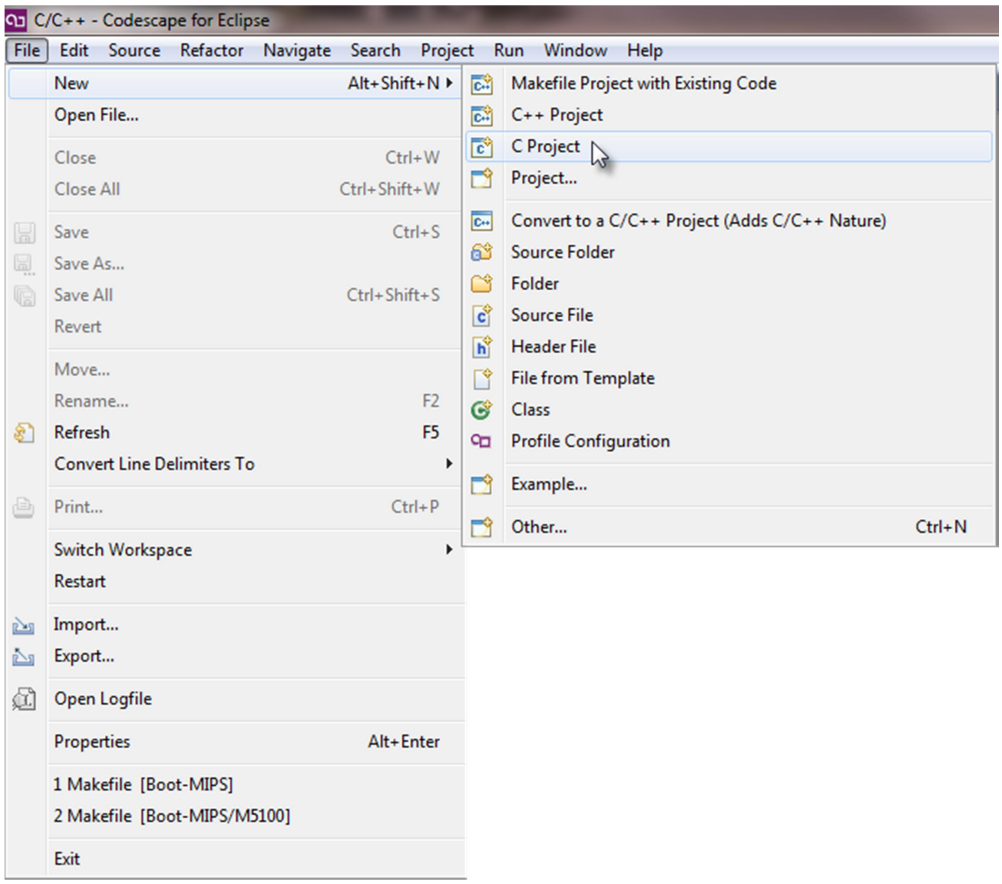
2.2. Installing and building as an Eclipse Project

The most current archive of the code and scripts referenced in this application note can be downloaded using this link:

http://training.mips.com/cps_mips/Examples/Boot-MIPS.zip

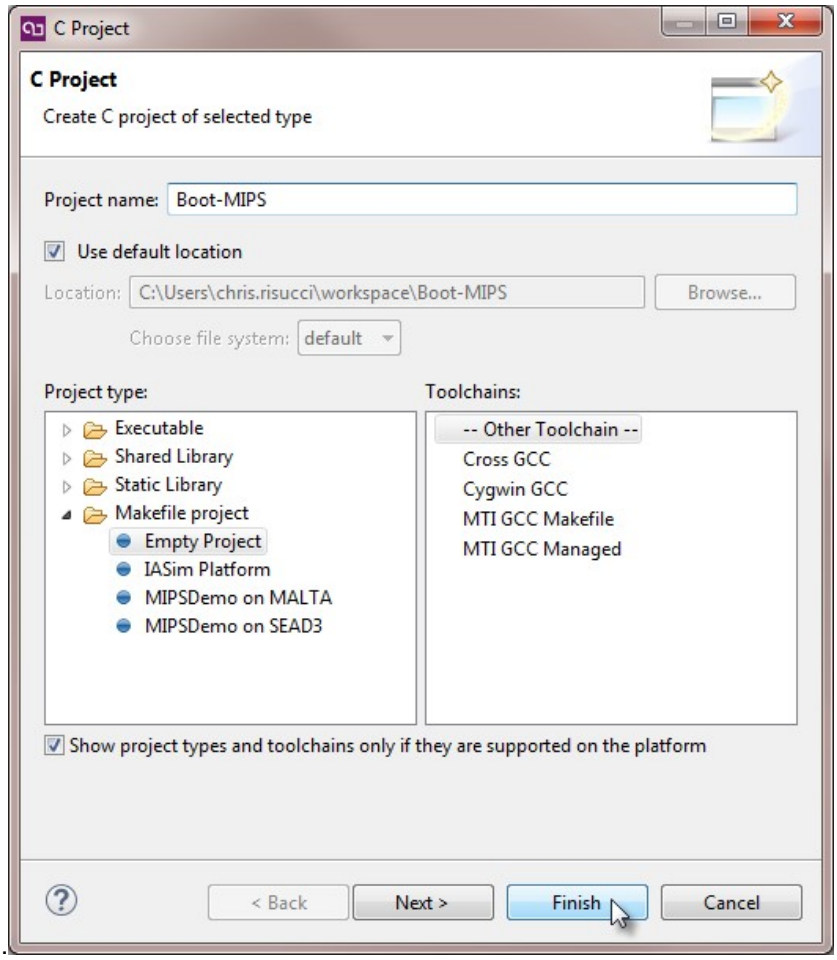
1. Download the zip file and start Codescape for Eclipse.
2. Create an empty Boot-MIPS project by selecting “New” from the “File” menu and then “C Project”
3.

4.

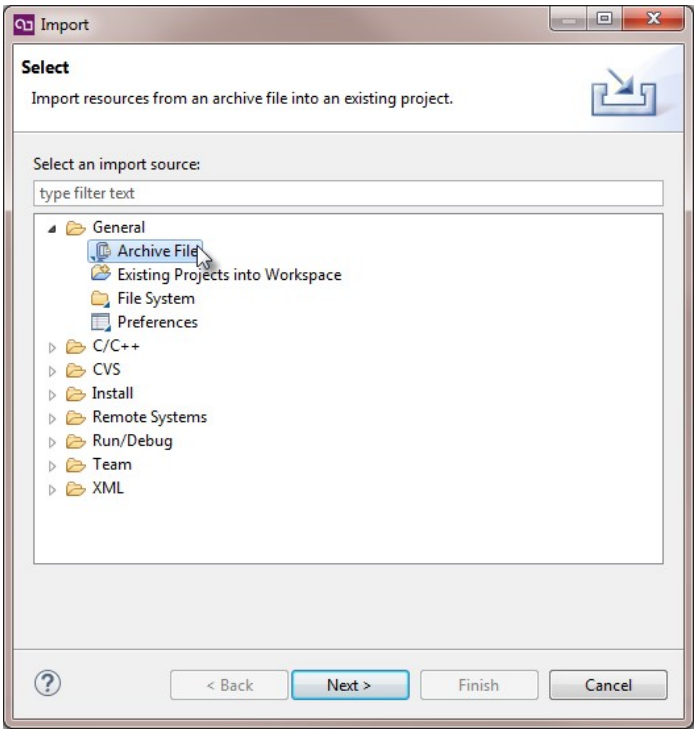

C/C++ - Codescape for Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Open Logfile
Properties Alt+Enter
1 Makefile [Boot-MIPS]
2 Makefile [Boot-MIPS/M5100]
Exit
Makefile Project with Existing Code
C++ Project
C Project
Project...
Convert to a C/C++ Project (Adds C/C++ Nature)
Source Folder
Folder
Source File
Header File
File from Template
Class
Profile Configuration
Example...
Other... Ctrl+N

5. Next Enter the “Project name”, select “Makefile project” and then select “Empty Project”. Click “Finish”.

6. .

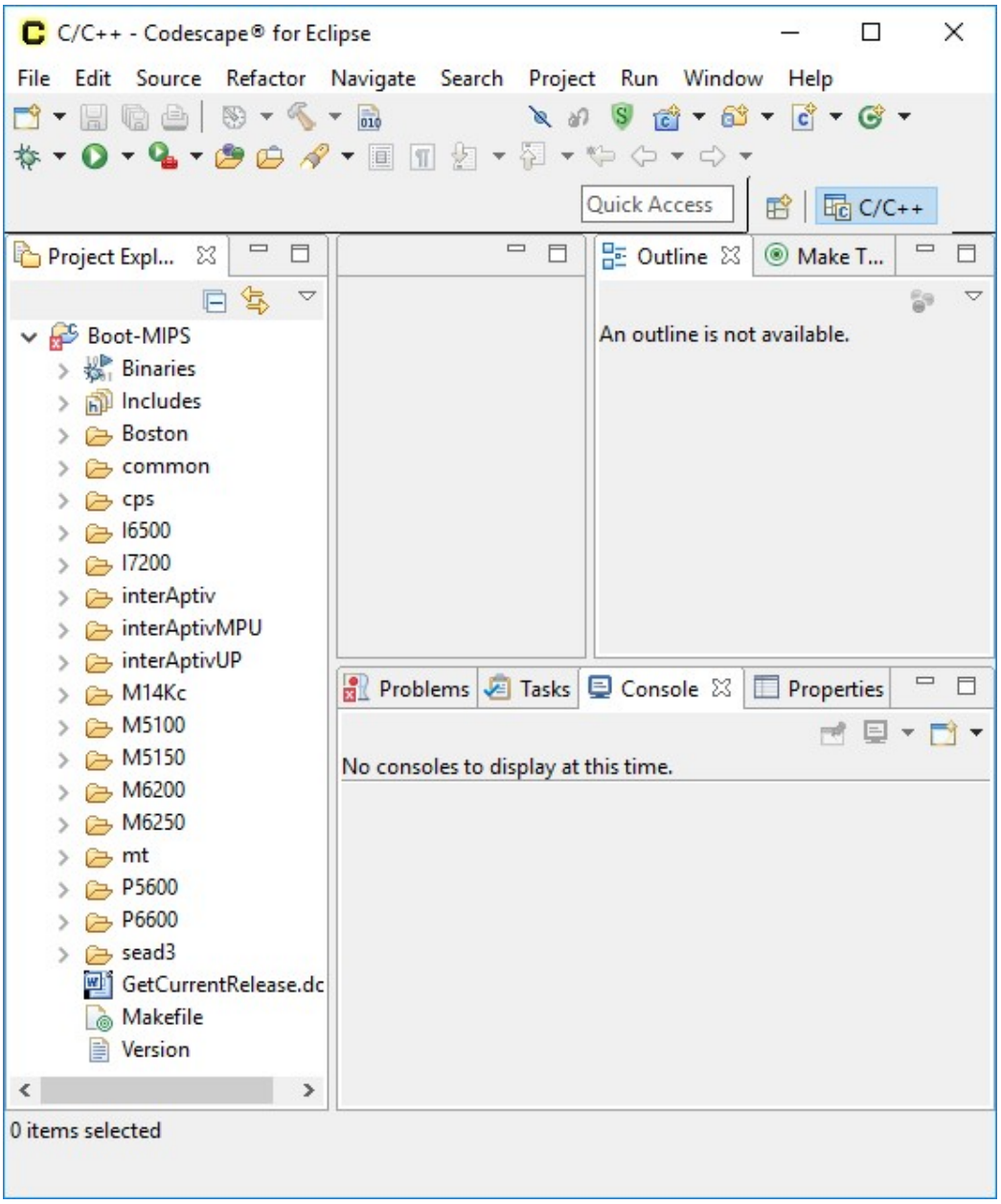


7. Then select “Archive File”.



8. Click “Next”, then Browse to the Boot_MIPS archive file on your system and select “Finish”.

The Boot-MIPS project will be imported into your workspace.



2.3. Building the Eclipse Project

The Boot-MIPS project is built using a Makefile (the project does not use the “Generate Makefile automatically” feature).

2.3.1. Target Selection

To build the boot code you will need to set the proper target. The targets are:

- interAptiv_SIM_RAM – Simulator Build with RAM copy for interAptiv Core Family

- interAptiv _SIM_SPRAM - Simulator Build with Scratchpad RAM copy for interAptiv Core Family
- interAptiv_SIM_RAM_EVA – IASim Build for simulator configured for EVA boot
- interAptiv _BOSTON_RAM - BOSTON Build with RAM copy for interAptiv Core Family
- interAptiv_BOSTON_RAM_EVA – BOSTON Build for bit files built for EVA boot
- interAptiv _BOSTON_SPRAM – BOSTON Build with Scratchpad RAM copy for interAptiv Core Family
- interAptivUP_SIM_RAM – Simulator Build with RAM copy for interAptivUP Core Family
- interAptivUP _SIM_SPRAM - Simulator Build with Scratchpad RAM copy for interAptivUP Core Family
- interAptivUP _BOSTON_RAM - BOSTON Build with RAM copy for interAptivUP Core Family
- interAptivUP _BOSTON_SPRAM – BOSTON Build with Scratchpad RAM copy for interAptivUP Core Family
- P5600_SIM_RAM – Simulator Build with RAM copy for P5600 Core Family
- P5600 _BOSTON_RAM - BOSTON Build with RAM copy for P5600 Core Family
- I6500_SIM_RAM – Simulator Build with RAM copy for I6500 and I6400 Core Family
- I6500 _BOSTON_RAM - BOSTON Build with RAM copy for I6500 and I6400 Core Family
- M5100_SIM_RAM – Simulator Build with RAM copy for M5100 Core Family
- M5100_SIM_SPRAM - Simulator Build with Scratchpad RAM copy for M5100 Core Family
- M5100_SEAD_RAM – SEAD Build with RAM copy for M5100 Core Family
- M5100_SEAD_SPRAM – SEAD Build with Scratchpad RAM copy for M5100 Core Family
- M5150_SIM_RAM – Simulator Build with RAM copy for M5150 Core Family
- M5150_SIM_SPRAM - Simulator Build with Scratchpad RAM copy for M5150 Core Family
- M5150_SEAD_RAM - SEAD Build with RAM copy for M5150 Core Family
- M5150_SEAD_SPRAM – SEAD Build with Scratchpad RAM copy for M5150 Core Family

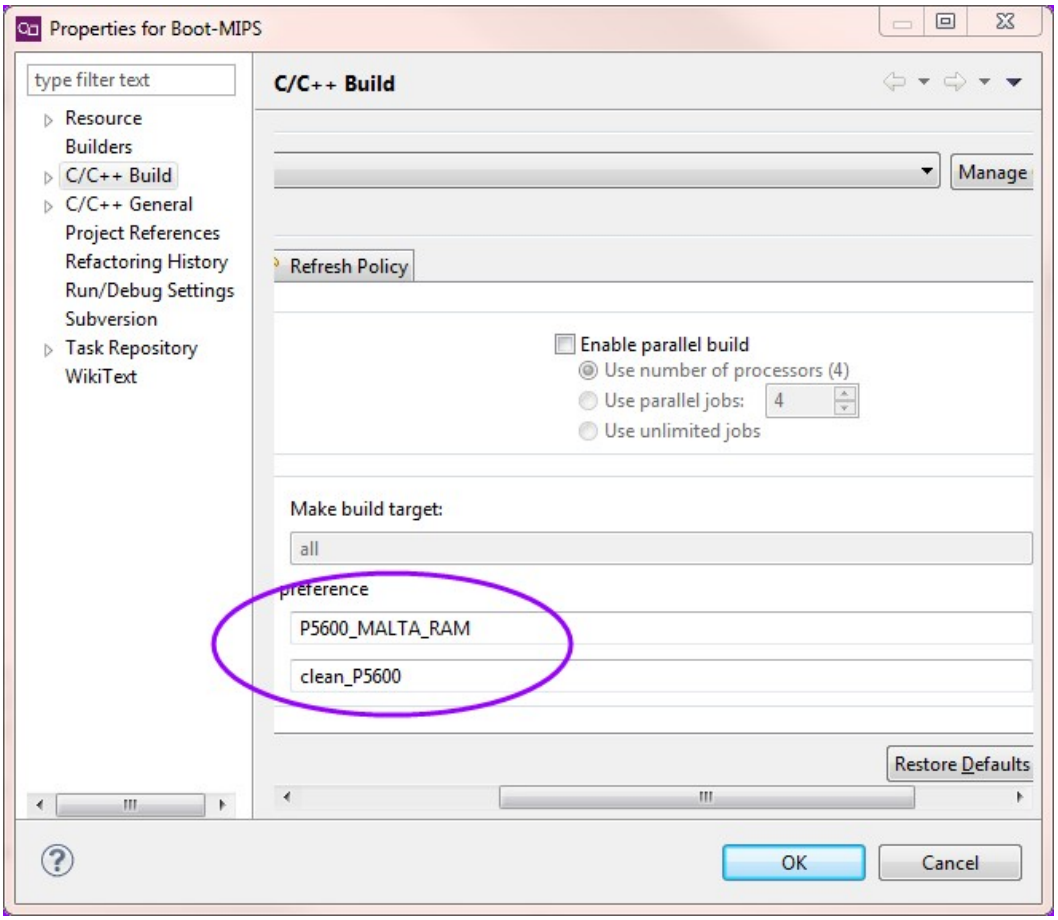
There are also clean targets for each core family:

- clean_interAptiv
- clean_interAptivUP
- clean_P5600

- clean_P6600
- clean_I6500
- clean_M5100
- clean_M5150

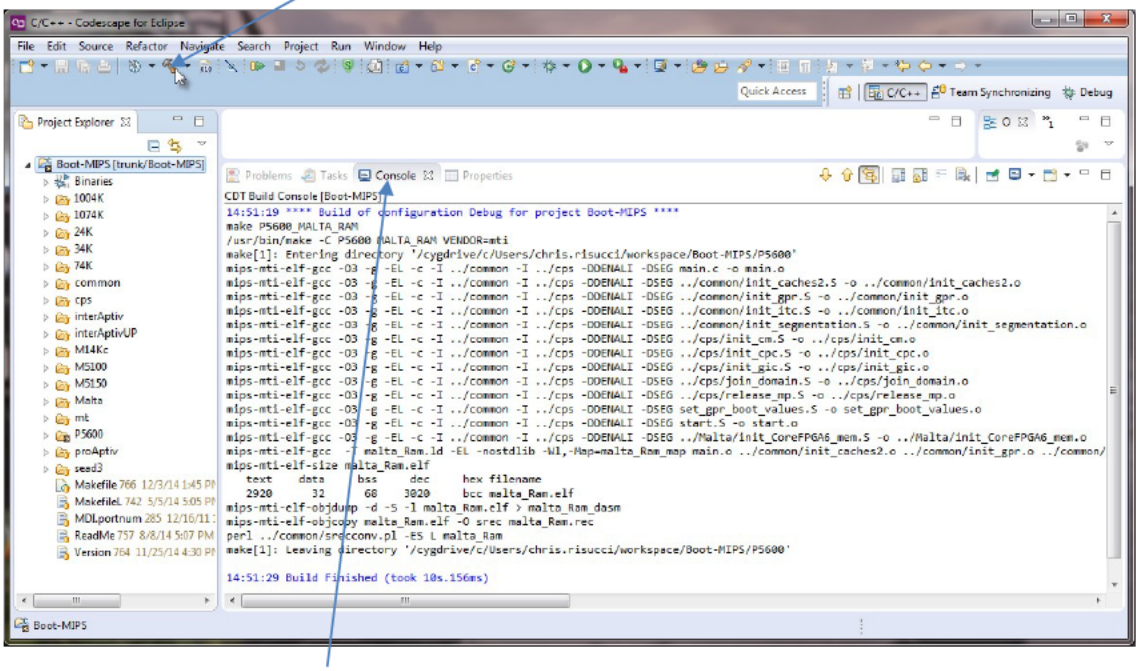
To set these targets, select the “Boot-MIPS” Project in the “Project Explorer” and press Alt + Enter to bring up the project properties. Then select “C/C++ Build” and click on “Behavior”. In the Behavior Dialog, change the “Build (Incremental build)” to the desired build. Also change the “Clean” field to the corresponding clean command.

Here is an example of setting a P5600_BOSTON_RAM target:



2.3.2. Build the selected Target

Next select the Boot-MIPS project and click on the build icon,  .



You can click on the “Console” tab to see the results of the build:

2.4. Installing and building as a Codescape Debugger project

2.4.1. Installing the code

The most current archive of the code and scripts referenced in this application note can be downloaded using this link:

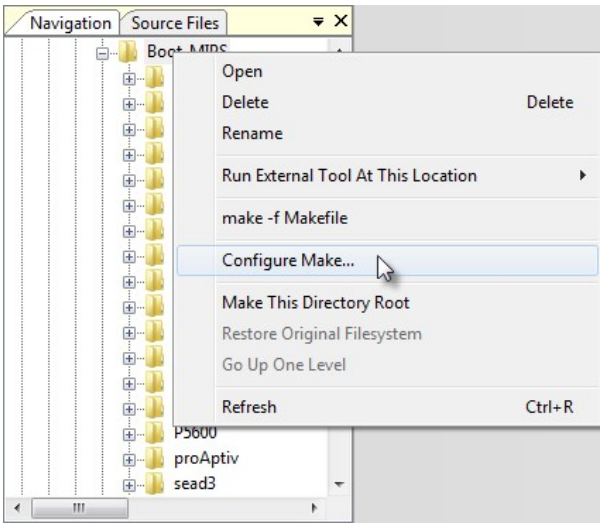
http://training.mips.com/cps_mips/Examples/Boot-MIPS.zip.

9. Download the zip file and unzip the file to your Boot-MIPS working directory. (Note that you may need to create the Boot-MIPS directory first.)

2.4.2. Building the Project

10. Start Codescape Debugger

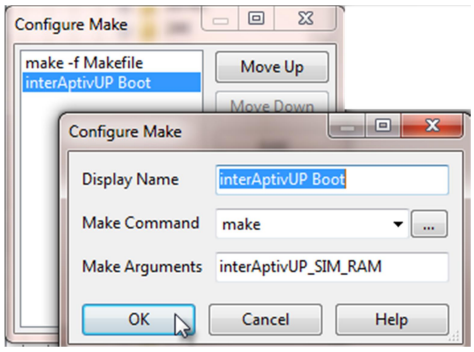
11. Navigate to the Boot-MIPS working directory. Then right click on the Boot-MIPS directory and select “Configure Make”

12. 

13. Click on “Add”. Fill in the “Display Name” with what you want to call this build, the “Make Command” with “make” and the “Make Arguments” with the make target you want to use. See the next section for an explanation on the make target.

14. Click OK then OK again to set the new make configuration.

15.

16. 
Configure Make
make -f Makefile
interAptivUP Boot
Move Up
Move Down
Configure Make
Display Name
interAptivUP Boot
Make Command
make
...
Make Arguments
interAptivUP_SIM_RAM
OK
Cancel
Help

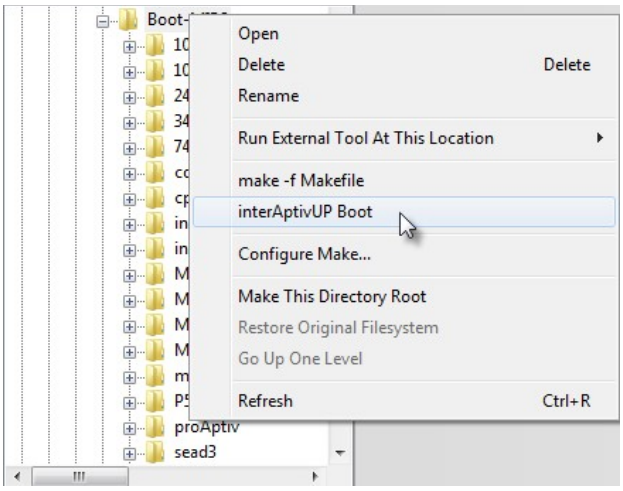
2.4.3. Target Selection

To build the boot code you will need to set the proper target. The targets are the same as for the ECLIPSE Project, See 'Target Selection' on page 15 for possible target settings.

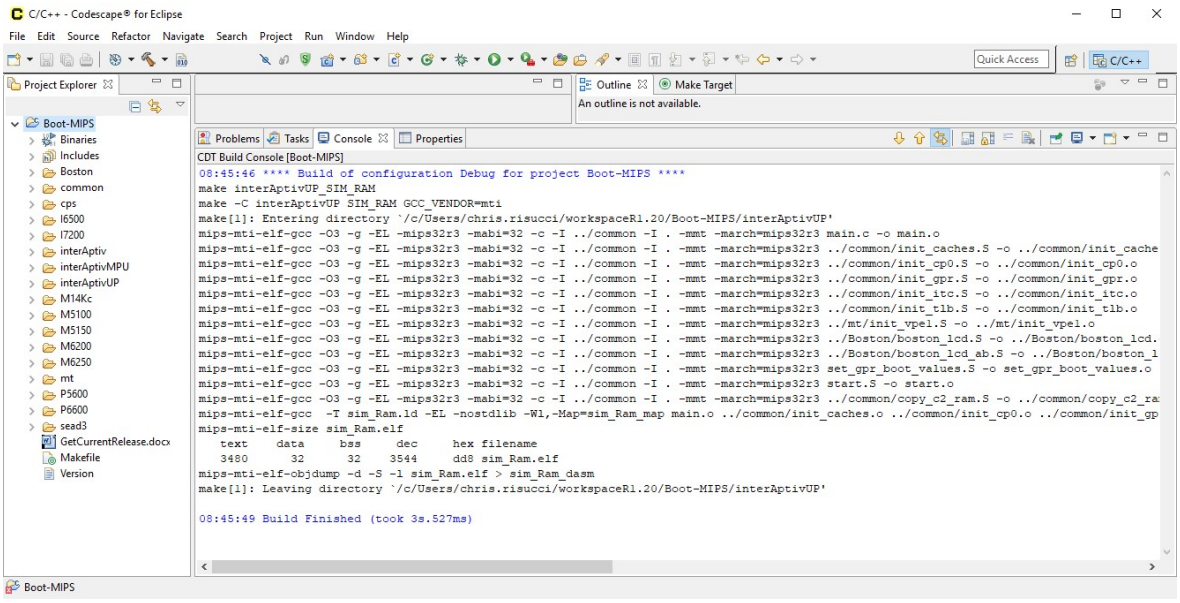
•

2.4.4. Starting the build

To start the build right click on the Boot-MIPS directory and select the make configuration you just created.



The build log will appear in the Make output region:



3. Boot-MIPS Package

3.1. Directories

The Boot MIPS project is divided into directories that are specific to each MIPS core, directories that contain common elements, and directories that are specific to a particular MIPS ASE.

The Boot-MIPS package contains the following directories:

- interAptiv - files specific to the MIPS32 interAptiv core
- interAptivUP – files specific to the MIPS32 interAptivUP family of cores
- M5100 – files specific to the MIPS32 M5100 family of cores
- M5150– files specific to the MIPS32 M5150 family of cores
- P5600 – files specific to the MIPS32 P5600 family of cores
- P6600 – files specific to the MIPS64 P6600 family of cores
- I6500 – files specific to the MIPS64 I6500 and I6400 family of cores
- common – files common to MIPS32/64 ISA and to more than one core
- cps – files common to a Coherent Processing System core
- Boston – files specific to the MIPS Boston Evaluation Board
- mt - files specific to the MT ASE cores
- sead3 – files specific to the SEAD-3 Evaluation Board

3.2. Core Directory Files

Each core directory may contain the following files depending on the features supported for the particular core. These files will be described in detail in the Code section of this document:

- core_config.h – contains #defines specific to target Core.
- main.c – A simple C file that is copied from ROM to RAM and later executed.
- set_gpr_boot_values.S – The code in this file initializes specific General Purpose Registers for later use.
- start.S – This is the start of the boot code which will be loaded at the boot exception vector (0xBFC0 0000). This code is similar for each core; however, it is trimmed to include only what is needed for that particular core.
- Makefile – This is the Makefile for the core. It will be called by the main Makefile in the top-level directory. The Makefile contains rules to build four types of executables. These object executables are described in the Build section of this document.

- BOSTON_Ram.ld - This is the linker script that will link the code for a MIPS BOSTON Evaluation Board that copies the main.c code from ROM to RAM.

- BOSTON_SPRam.ld - This is the linker script that will link the code for a MIPS BOSTON Evaluation Board that copies the main.c code from ROM to Scratchpad RAM. This can be used for systems that don't have RAM or caches.

- boston_Ram.ld - This is the linker script that will link the code for a MIPS Boston Evaluation Board that copies the main.c code from ROM to RAM.

- sim_Ram.ld - This is the linker script that will link the code for the IASim simulator that copies the main.c code from ROM to RAM.

- sim_SPRam.ld - This is the linker script that will link the code for the IASim simulator that copies the main.c code from ROM to Scratchpad RAM. This can be used to simulate systems that don't have RAM or caches.

- sim_Ram_eva.ld - This is the linker script that will link the code for the IASim simulator which is configured for a EVA boot that copies the main.c code from ROM to RAM.

- sead_Ram.ld - This is the linker script that will link the code for a MIPS sead Evaluation Board that copies the main.c code from ROM to RAM.

- sead_SPRam.ld - This is the linker script that will link the code for a MIPS sead Evaluation Board that copies the main.c code from ROM to Scratchpad RAM. This can be used for systems that don't have RAM or caches.

3.3. Common Directory Files

The common directory contains files that are common to the MIPS32 ISA or utilities that are common to all cores. These files are described in detail in the Code section of this document.

Note: In some cases there will be more than one file listed; that is because of slight differences needed for the specific architecture versions. The appreciate file should be used that matches the core architecture implementation.

- boot.h, boot_64.h - this is where you can find general #defines and the naming of some of the General Purpose Registers that make the code easier to follow.

- copy_c2_ram.S, copy_c2_ram_R6, copy c2_ram_R6_64 – the code in these files is used to copy the C code in main.c from ROM to RAM.

- copy_c2_SPram.S – the code in this file is used to set up Scratchpad RAM and copy the C code in main.c from ROM to SPRAM. See 'common/copy_c2_SPram.S' on page 67.

- copy_c2_Spram_MM.S – the code in this file is used to set up Scratchpad RAM and copy the C code in main.c from ROM to SPRAM for M5100 and M5150 cores. See 'common/copy_c2_Spram_MM.S (M5100 and M5150 cores)' on page 73.

- eva.h - #defines for segmentation and EVA support

- init_caches.S, init_caches_64.S – initializes the L1 instruction and data caches and the L2 cache if present. See 'common/init_caches.S' on page 79.

- init_cp0.S, init_cp0_64.S – initializes Coprocessor 0 Registers.
- init_gpr.S, init_gpr_shadow.S, init_gpr_64.S – initializes General Purpose Registers 1 – 31 and additional shadow set registers where applicable.
- init_itc.S – initializes Inter-Thread Communications Storage if present.
- init_L2_CM2, init_L2_CM3 – initializes L2 cache for the coherency manager.

3.4. init_tlb.S, init_tlb_FTLB.S, init_tlb_R6.S, init_ftlb_R6.S

Initializes the Translation Look-aside Buffers if present. See 'common/init_gpr_64.S - init_gpr for 64 bit cores' on page 88.

- srecconv.pl – this is a Perl script that is used to convert a file in Srec format to one that can be “Flashed” onto a BOSTON board.

3.5. cps Directory Files

The files in the cps directory pertain to a Coherent Processing Core such as the interAptiv and P5600.

- cps.h, cps_CM3.h - #defines for CM, CPC, and GIC for the CM
- cps_CM3.h - #defines for CM, CPC, and GIC for the CM3
- init_cm.S, init_CM3_64 - initializes the Coherence Manager.
- init_cpc.S, init_cpc_CM2_64, init_cpc_CM3 – initializes the Cluster Power Controller.
- init_gic.S, init_gic_CM2_64.S, init_cpc_CM3_64.S – initializes the Global Interrupt Controller
- join_domain.S - joins a processing element to a Coherence Domain.
- release_mp.S, power_up_cores_CM3_64.S – releases a core for multi-processing.

3.6. Boston Directory files

- These files are specific to the MIPS Boston Evaluation Board.
- boston_lcd.S – This file contains code need to display messages on the lcd display.
- init_FPGA_mem.S – This file waits for the memory controller to initialize.

3.7. mt Directory Files

The files in this directory are specific to the MT ASE.

- init_vpe1.S– initializes the second Virtual processor for an MT Core such as the interAptiv or interAptivUP if present. See ‘mt/init_vpe1.S - init_vpe1 (interAptivUP or interAptiv only)’ on page 115.

- init_vpe_s.S - – initializes all additional Virtual processor for an MT Core such as the I7200 if more are present.

3.8. sead3 Directory Files

- lcd.h - This file contains the defines to control the LDC device on the SEAD evaluation board
- lcd.S – This file contains code need to display messages on the sead/lcd display.
- sead.h - This file contains the defines to enable the use of the LCD display on the SEAD evaluation board
- init_MC_sead.S initializes the sead3 memory controller.

3.9. Other files in the top directory

- Makefile – this is the top-level Makefile that can be used to build a specific type for a specific core. See ‘Target Selection’ on page 15 for a list of available Targets.

4. Code Details

This section will walk through the code contained in the Boot-MIPS project. The Boot code is really exception code. The linker using the linker file script will link this code for the BEV (Boot Exception Vector). The boot code will be loaded into flash memory at the Boot Exception Vector (BEV) and will be the first instructions execution by the processor.

The basic function of this boot code is to initialize the Processor resources and get to a point where an OS such as ThreadX or Linux can be loaded to initialize the rest of the system.

4.1. M5100 Core:

The M5100 is the simplest Core to initialize because it has no caches or a TLB to initialize. The file start.S is the starting point for the boot initialization.

4.1.1. start.s for M5100 Core

The first function, __reset_vector, just loads the address of the __cpu_init function and jumps to it. This will do a mode switch to enable microMIPS mode.

```
LEAF(__reset_vector)
   la a2,__cpu_init
   jr a2
   mtc0  $0, C0_COUNT          # Clear cp0 Count (Used to measure boot time.)

# Note: For the SEAD board Address 0x1FC0.0010 is “special”, in the sense that
# it is overridden
# and does not decode to an address in the SW-EPROM, but rather to memory
# mapped register which holds the board REVISION number. This makes the
# following nops necessary  to insure the next code segment does not fall
# into the Revision address range

   nop
   nop
   nop
   nop

END(__reset_vector)

.set micromips

LEAF(__cpu_init)

# Verify the code is here due to a reset and not NMI. If this is an NMI then
# triggers a debug breakpoint using an sdbp instruction (software debug break
point).
```

```
    mfc0    s1, C0_STATUS        # Read CP0 Status
    ext     s1, s1, 19, 1        # extract NMI
    beqz    s1, init_resources   # Branch if this is NOT an NMI exception.
    sdbbp                        # Failed assertion: NMI.

init_resources:

# Set the global pointer register address to _gp and
# the stack pointer register to
# STACK_BASE_ADDR these values are assigned in the linker file

    la      gp, _gp                    # All shared globals.
    li      sp, STACK_BASE_ADDR        # Set up stack base.

 # Initialize Status to clear all errors and interrupts and keep the CPU in
boot mode

   li    v1, 0x00400004               # (M_StatusERL | M_StatusBEV)
   mtc0  v1, C0_STATUS                # write C0_Status

 # Clear all cause bits so random interrupts will not
 # occur when interrupts are
 # enabled.

   mtc0  zero, C0_CAUSE          # write C0_Cause

# Clear the timer interrupts by clearing the CP0 Compare register.
# (Count was cleared at the reset vector to allow timing of the boot process.)
   mtc0  zero, C0_COMPARE        # write C0_Compare

NOTE: At this point you should initialize you memory controller. Here is an
example that initializes the SEAD board memory controller:

   la    a2,     init_mc_sead          # initialize the sead memory controller
   jalr  a2

# Call the function to copy the main.c code from the flash memory into to ram.

    la   a2,   copy_c2_ram             # Copy "C" code and data to RAM and
zero bss
    jalr a2
# Prepare for error return, eret, to main
```

```
    la      ra, all_done          # If main returns then go to all_done:.
    la      v0, main              # load the address of the main function
    mtc0    v0, C0_ERRPC          # Write ErrorEPC with the address of main
    ehb                           # clear hazards
    # Return from exception will now execute code in main
    eret    # Exit this reset exception handler and start
            # execution of main().
all_done:
    # If main returns it will return to this point.  Just spin here.
    b       all_done
END(__cpu_init)

# Inline the code for the LCD display and Copy to C function to
# fill the rest of
# space between here and the next exception vector address.

#include <lcd.S>
#include <copy_c2_ram.S>
#include <init_MC_sead.S>
/*****************************************************************************
*****
    B O O T   E X C E P T I O N   H A N D L E R S (CP0 Status[BEV] = 1)
*****************************************************************************
*****/
/* NOTE: the linker script must insure that this code starts at */
/* BEV (boot exception vector) + 0x300 so the exception */
/* vectors will be addressed properly. All .org assume this! */
.set nomicromips      # This causes the code below to be compiled as MIPS32

.org 0x300            #SRAM Parity Error exception. start + 0x300 */
start_exceptions:
    sdbbp
    nop

.org 0x380            # General exception. start + 0x380*/
   DISP_STR(msg_gexcpt);
wait_here_GE:
    b          wait_here_GE
    nop

# If you want the above code to fit into 1k flash you will need
# to leave out the
# code below. This is the code that covers the debug exception
```

```
# which you normally will not get

.org 0x480     # EJTAG Debug (with ProbEn = 0 in the EJTAG Control Register)
wait_here_ED:
    b          wait_here_ED
    nop

.text
# Constant: message to be displayed when a general exception occurs
MSG( msg_gexcpt,          "G_EXCEPT" )
```

4.2. M5150 Core

The M5150 core has Watch registers, Instruction and Data cache and TLB resources that should be initialized in the boot process.

4.2.1. start.s for M5150 Core

The first function, __reset_vector, just loads the address of the __cpu_init function and jumps to it. This does 2 things it will jump to a KSEG0 address which is a mirror of the BEV's KSEG1 address but cacheable (configured as uncached for the time being). Second it will do a mode switch to enable microMIPS mode.

```
LEAF(__reset_vector)
   la a2,__cpu_init
   jr a2
   mtc0  $0, C0_COUNT          # Clear cp0 Count (Used to measure boot time.)

   # Note: Address 0x1FC0.0010 is "special",
   # in the sense that it is overridden
   # and does not decode to an address in the SW-EPROM, but rather to memory
   # mapped register which holds the board REVISION number. This makes the
   # following nops necessary to insure the next code segment does not fall
   # into the Revision address range

   nop
   nop
   nop
   nop

END(__reset_vector)

.set micromips       # This causes the code below to be compiled as micromips
                     # This will reduce the code size and help the code
                     # fit into 1K of memory

LEAF(__cpu_init)

# Verify the code is here due to a reset and not NMI.
# If this is an NMI then trigger
# a debugger breakpoint using a sdbp instruction.

    mfc0    s1, C0_STATUS        # Read CP0 Status
    ext     s1, s1, 19, 1        # extract NMI
```

```
    beqz    s1, init_resources    # Branch if this is NOT an NMI exception.
    nop
    sdbbp                         # Failed assertion: NMI.

init_resources:                   # initializes resources for "cpu".

# Set the global pointer register address to _gp
# and the stack pointer register to
# STACK_BASE_ADDR these values are assigned in the linker file

    la      gp, _gp               # All shared globals.
    li      sp, STACK_BASE_ADDR   # Set up stack base.

    # Initialize CP0 registers

    la a2,     init_cp0
    jalr a2
    nop

# Initialize the TLB

    la    a2, init_tlb
    jalr a2
    nop

#Initialize the Instruction cache

    la    a2, init_icache
    jalr a2
    nop

# The changing of Kernel mode cacheability must be done from KSEG1
# Since the code is executing from KSEG0 It needs to do a
# jump to KSEG1 change K0 and jump back to KSEG0

    la    a2,   change_k0_cca
    li          a1, 0xf
    ins         a2, a1, 29, 1     # changed to KSEG1 address by setting bit 29
    jalr a2
    nop

   # Initialize the Data cache
```

```
    la   a2, init_dcache    # Initialize the L1 data cache
    jalr a2
    nop

   # Copy "C" code (main.c) and data to RAM and zero bss

    la   a2, copy_c2_ram
    jalr a2
    nop

    # Prepare for eret to main (sp and gp set in set_gpr_boot_values).

    la      ra, all_done         # If main returns then go to all_done:.
    la      v0, main             # load the address of the main function
    mtc0    v0, $30              # Write ErrorEPC with the address of main
    ehb                          # clear hazards

    # Return from exception will now execute code in main

    eret        # Exit reset exception handler and start execution of main().


/*****************************************************************************
*******
*****************************************************************************
******/
all_done:
    # If main returns it will return to this point.  Just spin here.
    b       all_done
    nop

END(__cpu_init)

# Inline the code fill the rest of
# space between here and the next exception vector address.

#include <init_caches.S>
#include <copy_c2_ram.S>

/*****************************************************************************
*******
```

```
    B O O T   E X C E P T I O N   H A N D L E R S (CP0 Status[BEV] = 1)
*******************************************************************************
******/
/* NOTE: the linker script must insure that this code starts at */
/* BEV (boot exception vector) + 0x300 so the exception */
/* vectors will be addressed properly. All .org assume this! */

.org 0x200                  # TLB refill, 32 bit task.
    sdbbp                   # This has the effect of starting the debugger
    nop

# fill the empty space until the next exception vector with needed code
#include <lcd.S>

.org 0x280                  # XTLB refill, 64 bit task. start + 0x280
    sdbbp                   # This has the effect of starting the debugger
    nop

#include <init_cp0.S>

.org 0x300                  # Cache error exception. start + 0x300
    sdbbp                   # This has the effect of starting the debugger
    nop

#include <init_tlb.S>
   nop
.org 0x380
.set at                     # General exception. start + 0x380
   DISP_STR(msg_gexcpt);
.set noat
wait_here:
    b          wait_here
    nop

# If you want the above code to fit into 1k flash you
# will need to leave out the
# code below. This is the code that covers the debug
# exception which you normally will not get.

.org 0x480                  # EJTAG Debug (with ProbEn = 0 in the EJTAG Control
Register)
1:  b16       1b  /* Stay here */
    nop
```

```
.set nomicromips

.text

MSG( msg_gexcpt, "G_EXCEPT" )
```

4.3. interAptivUP Core

The interAptivUP is a single core multi-threaded processor that supports 1 or 2 VPEs and up to 9 threads distributed across VPEs. Below is the boot flow. Refer to start.S for the start of a code walk through.

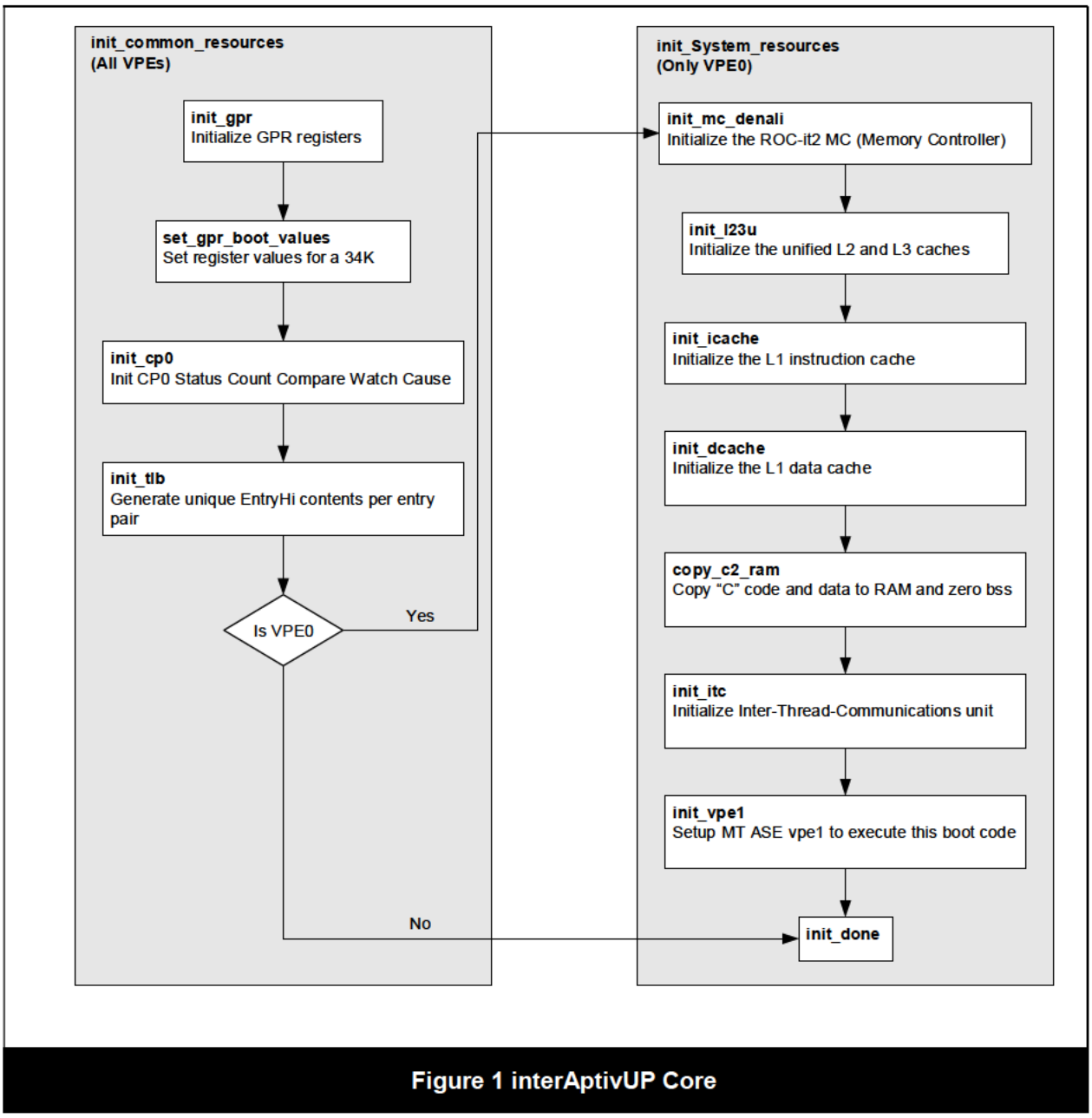


Figure 1 interAptivUP Core

4.4. interAptiv Core

The interAptiv is a multi-core multi-threaded processor that supports 1 or 2 VPEs and up to 9 threads distributed across VPEs on each core. The boot begins by executing the start.S. Below is the boot flow of the code in start.S. Refer to 'start.S ' on page 44 for the start of a code walk through.

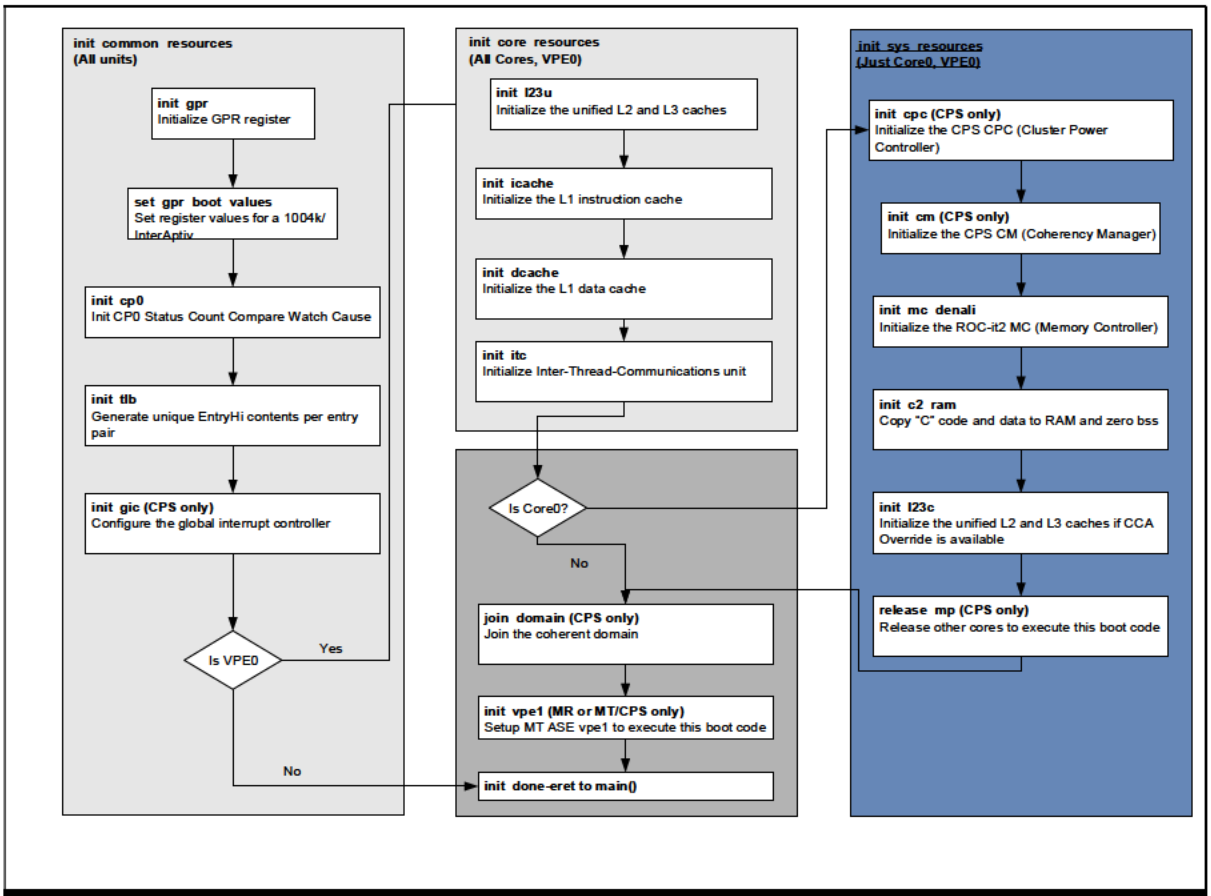


Figure 2 interAptiv Core

4.5. P5600/P6600 Cores

The P5600/P6600 can have up to 6 single threaded processors. Below is the boot flow.

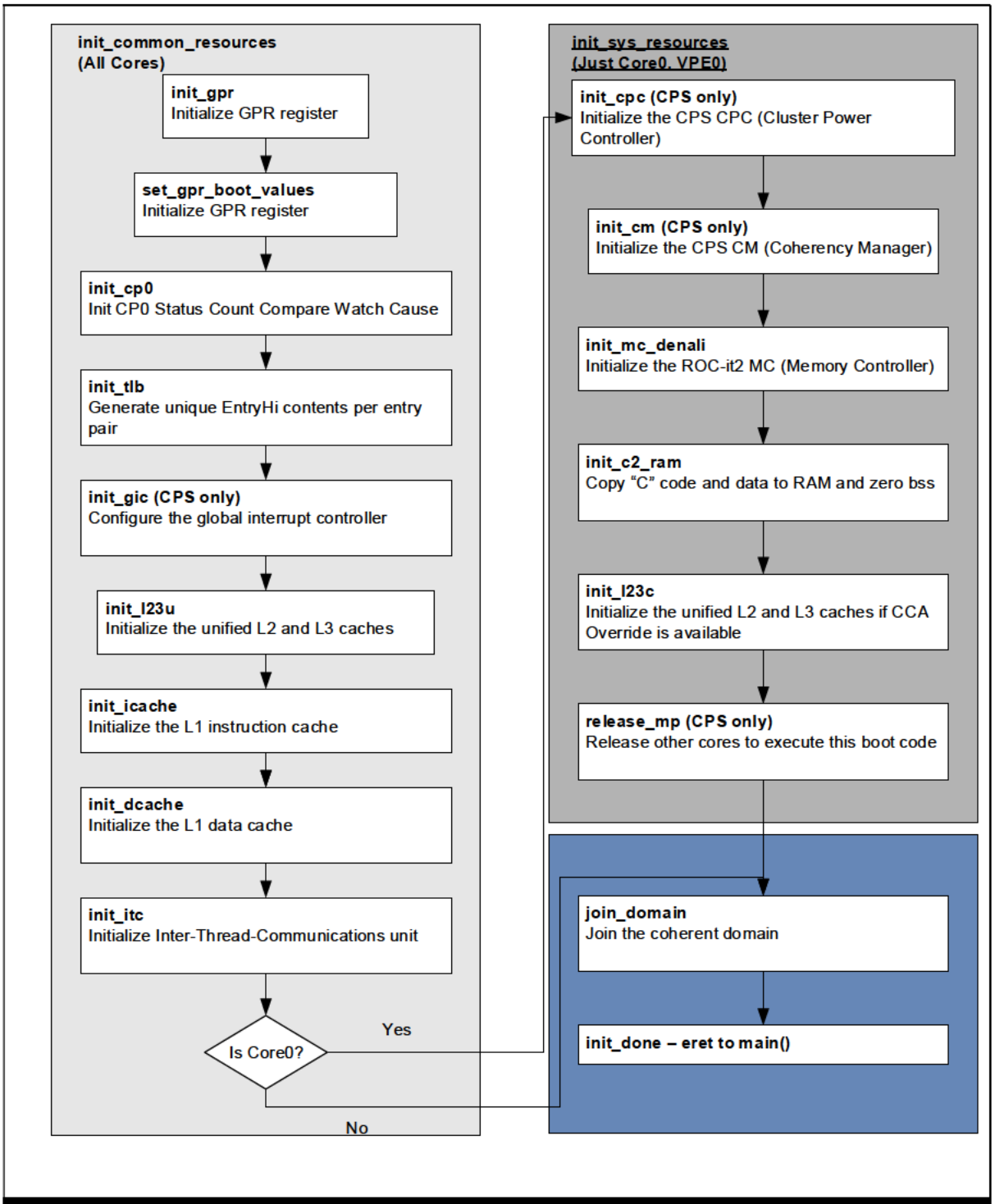


Figure 3 P5600 Core

4.6. I6400 Multi Core

The I6400 can have up to 4 Virtual processors per core and up to 6 cores threaded processors for a total of 24 processors per Multi Core Cluster. Below is the boot flow.

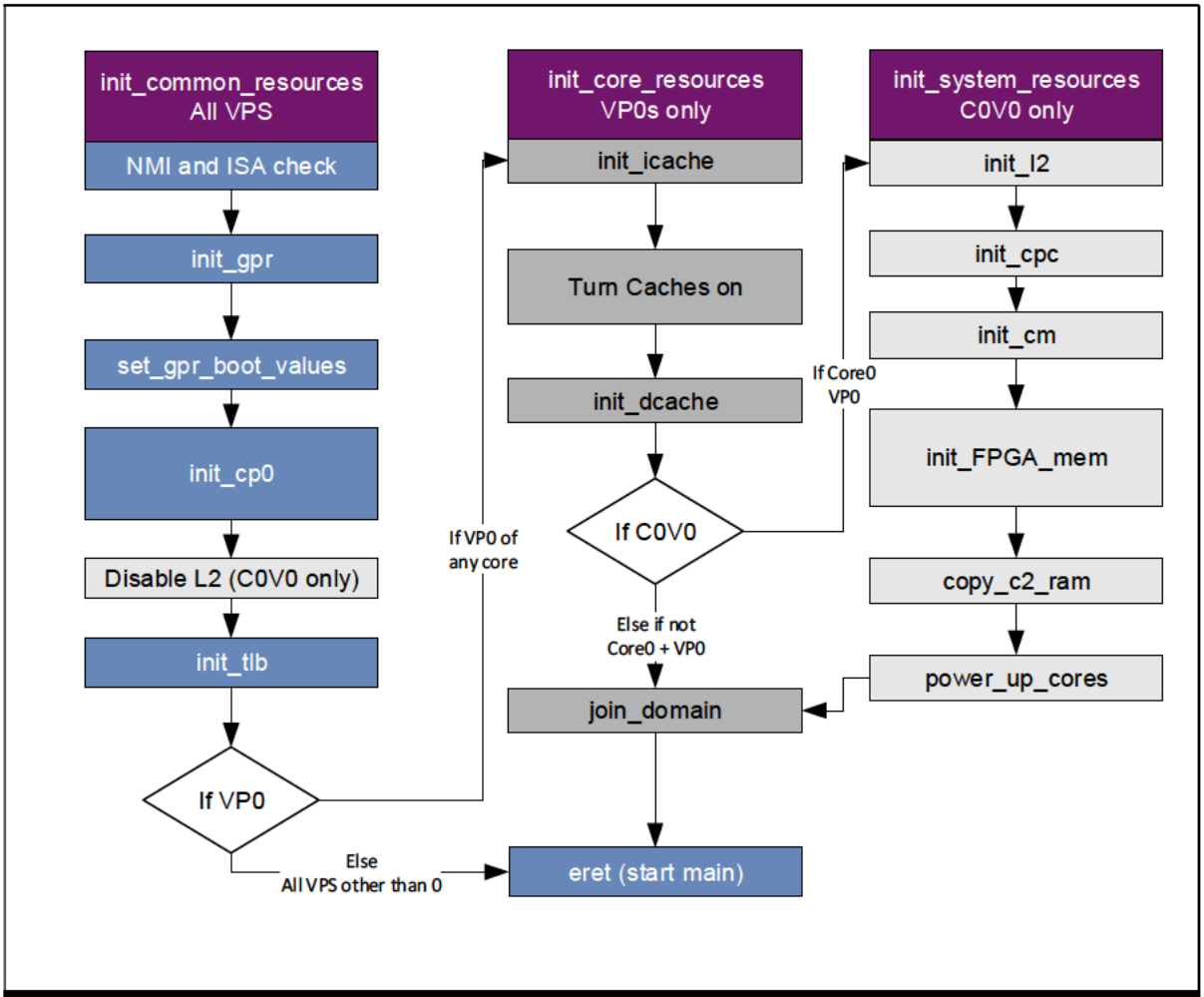


Figure 4 I6400 Boot Flow

4.7. I6500 Multi Core Multi Cluster

Each Cluster in the I6500 boots the same as the I6400 shown in 4.6.

4.8. common/boot.h/boot_64.h

This file contains defines that are common to many cores and are used by the Boot MIPS code. boot.h should be used for MIPS32 cores and boot_64.h should be used for MIPS64 cores.

The LEAF macro is used in this example for assembly functions the make no calls to other functions and are not passed any arguments (so they don't require a stack). The return address in the ra register should not be changed. Using the LEAF and END macros together help a debugger in finding the source code of the executable being debugged and helps in determining the size of the function.

```
#define LEAF(name)\
   .##text;\
   .##globl    name;\
   .##ent  name;\
name:

#define END(name)\
   .##size name,.-name;\
 .##end  name
```

There are #defines that control register locations and values for some of the memory-mapped registers of a Coherent Processing System. These #defines will only be needed for use with a CPS. The values for these #defines are dependent on how the CPS was configured. The values here correspond to the bit file that was used by us for testing. These are the default values if you have received your bit file from MIPS.

NOTE the boot_64.h has MIPS64 values instead of the MIPS32 values.

```
#define GCR_CONFIG_ADDR     0xbfbf8000  // KSEG0 address of the GCR registers
#define GCR_CONFIG_ADDR_PB  0xbfbf8000  // Post Boot address of the GCR
registers
#define GIC_P_BASE_ADDR     0x1bdc0000  // physical address of the GIC
#define GIC_BASE_ADDR       0xbbdc0000  // KSEG0 address of the GIC
#define GIC_BASE_ADDR_PB    0xbbdc0000 // Post Boot address of the GIC
#define CPC_P_BASE_ADDR     0x1bde0001  // physical address of the CPC
#define CPC_BASE_ADDR       0xbbde0000  // KSEG0 address of the CPC
#define CPC_BASE_ADDR_PB    0xbbde0000  // Post Boot address of the CPC
```

The base address and size increments of the stacks are defined. These can be changed to suit the specific memory requirements of the system. The base should be placed in KSEG0 where there is no code or data.

```
#define STACK_BASE_ADDR     0x82000000  /* Change: Based on memory size. */
```

```
#define STACK_SIZE_LOG2     22                      /* 4Mbytes each */
```

Note: Some of the Cores support Extended Virtual Addressing, EVA. These cores can be configured at boot time to boot in EVA mode. Booting in EVA mode changes the MIPS virtual memory map. The address above will be different when a Core is configured to boot EVA mode. The boot.h file contains a #ifdef EVA to changes these defines for a BOSTON bit file that has been configured to boot EVA mode. For the MIP64 cores there should be no need to use EVA so the boot_64.h doesn't contain any defines for EVA.

To improve the readability of the assembly code, the following names have been #defined for some of the general-purpose registers. These are present as applicable depending on the target core. The comments tell their intended purpose:

```
// o32 $2 - $7 (v0, v1, a0 - a3) reserved for program use
// p32 $2 - $7 (t4, t5, a0 - a3) reserved for program use

/* o32 t0 Core number. Only core 0 is active after reset. */
/* p32 a4 Core number. Only core 0 is active after reset. */
#define r8_core_num    $8
/* o32 t1 MT ASE VPE number that this TC is bound to (0 if non-MT.) */
/* p32 a5 MT ASE VPE number that this TC is bound to (0 if non-MT.) */
#define r9_vpe_num     $9
/* o32 t2 Core implements the MT ASE. */
/* p32 a6 Core implements the MT ASE. */
#define r10_has_mt_ase  $10
/* o32 t3 Core is part of a Coherent Processing System. */
/* p32 a7 Core is part of a Coherent Processing System. */
#define r11_is_cps      $11

// $12 - $15 (t4 - t7 o32 or t0 - t3 n32/64/p32)  are free to use
// $16, $17 (s0 and s1) are free to use

#define r18_tc_num      $18  /* s2 MT ASE TC number (0 if non-MT.) */
#define r19_more_cores  $19  /* s3 Number of cores in CPS in addition to core
0. GLOBAL! */
#define r20_more_vpes   $20  /* s4 Number of vpes in this core in addition to
vpe 0. */
#define r21_more_tcs    $21  /* s5 Number of tcs in vpe in addition to the
first. */
#define r22_gcr_addr    $22  /* s6 Uncached (kseg1) base address of the Global
Config Registers. */
#define r23_cpu_num     $23  /* s7 Unique per vpe "cpu" identifier (CP0
EBase[CPUNUM]). */
#define r24_boston_byte $24  /* t8 flash address of boston_lcd_display_byte
function */
#define r25_coreid      $25  /* t9 Copy of cp0 PRiD GLOBAL! */
```

```
//                                $26  /* k0 Interrupt handler scratch */
//                               $27  /* k1 Interrupt handler scratch */
// $28 gp and $29 sp
#define r30_cpc_addr    $30  /* s8 Address of CPC register block after
cpc_init. 0 indicates no CPC. */
// $31 ra
```

4.9. start.S

As the name implies, the code in start.S is the start of the Boot Code. This assembly source file contains the exception vectors and control code for the boot process and calls other assembly functions as needed to perform initialization of the sub-components of the core. Each core family has a start.S that is tailored for it, so there is a start.S file located in each core family directory. There are three main differences between the individual start.S files:

- The simplest is the start.S file for single cores such as the M5100 and M5150. This is the smallest subset of functions that are required to initialize a single core.
- Additional functionality is added to initialize a multi-threaded system such as the interAptivUP core.
- Additional functionality is added to initialize a multi-core Coherent Processing System, such as the interAptiv and P5600.

The code begins by setting options for the assembler:

```
#include <boot.h>
#include <mips/m32c0.h>
#include <mips/regdef.h>
#include <cps.h>


   .set  noat      # Don't use r1 (at)
```

The AT register ($1) is used by the assembler for synthetic instructions. This boot code is using AT for a specific purpose and does not use any synthetic instructions, so it uses the “noat” option. NOTE: If a program uses a synthetic instruction with the “noat” option set, the assembly will stop with an error.

4.9.1. Boot Exception Vector

When a MIPS Core is powered up or is reset, it is in exception mode, so the first instruction is fetched from the Boot Exception Vector. The boot code loads the address of the first code to call and then jumps to that address. The jump serves to jump to where the code was linked for. This makes it possible for the debugger to find the correct code to display in the source code window.

```
LEAF(__reset_vector)
    la a2,     check_nmi
    jr a2
```

```
    mtc0  zero, C0_COUNT        // Clear cp0 Count (Used to measure boot time.)
```

NOTE: For MIPS64 cores a dla macro instruction is needed instead of the la used above. The dla macro results in more instructions so there is a branch around the ID register.

4.9.2. Other Exceptions

The next section in start.S covers the other exception vectors. The code uses the .org directive to communicate to the linker where the code should be placed in memory. The value supplied with .org is the offset from the starting base address of the code. If the code was started at the default boot exception vector address of 0xBFC0 0000, then .org 0x200 would put the code at 0xBFC0 2000.

Any exception signal (with the exception of an interrupt) during this boot indicates a serious error in the code or the hardware, so here we are not concerned with elaborate exception handlers. For the most part, the code uses the debug breakpoint instruction “sdbbp” which will halt execution and transfer control to the debugger, if one is attached. If not the code goes into an infinite loop.

On some cores the Boot-MIPS code is used as part of a test suite which runs within a simulator. In that case the simulator is designed to look for the sltiu instruction with register 0 as the register arguments. When the simulator finds this in the instruction stream it halts and outputs the immediate value to the trace output. The boot code will follow this code path if RTEST is defined.

That is what happens with the first three exception vectors:

```
.org 0x200 // TLB refill, 32-bit task.
.org 0x280 // XTLB refill, 64 bit task.
.org 0x300 // Cache error exception.
```

Here is the code:

```
#ifdef RTEST
    sltiu $0, $0, 0xabc1    # simulator fail condition
    wait
#else
    sdbbp
    nop
1:  b       1b  // Stay here
    Nop
#endif
```

4.9.3. Multi-core inter-processor interrupts processing

The General Exception vector for Multi-core cores is used to send inter-processor interrupts. On other cores the code halts like described above.

For Multi-core cores the cause register is checked to make sure the exception is for an interrupt otherwise it will branch ahead (other_code) and halt like the exceptions above:

```
.org 0x380 // General exception.
   // expecting an inter-processor interrupt
   mfc0  k1, C0_CAUSE
   ext   k0, k1, 2, 5            // extract exit code
   bne   k0, zero, other_code    // should be 0 indicating a interrupt
   ext   k0, k1, 10,             // extract interrupt
   li    k1, 1
   bne   k0, k1, other_code      // should be interrupt 0
   nop
```

For multi-processor cores the Boot-MIPS code uses inter-processor interrupts to synchronize all processors to the same point in the main function before continuing. Processor 0 is used to send inter-processor interrupts to the remaining processors. The remaining processors use the wait instruction to wait for this interrupt.

The exception code that follows is used to process the inter-processor interrupts. First the code clears the interrupt condition by writing the interrupt number to the Global Interrupt Write Edge Register. The interrupt to be cleared is the Core number plus 0x20. (There will be more about how this was set in the section that details the main.c code.) The code below shows the loading of the address for the Global Interrupt Write Edge Register (GIC_SH_WEDGE), then obtaining the Core number from the Exception Base Register (EBASE), and adding the 0x20 offset to compute the interrupt number to clear. Finally, the interrupt number is written to the Global Interrupt Write Edge Register to clear the interrupt.

```
    li    k0, (GIC_SH_WEDGE | GIC_BASE_ADDR_PB)
    mfc0  k1, C0_EBASE      // read EBASE
    ext   k1, k1, 0, 10     // Extract CPUNum
    addiu k1, 0x20          // Offset to base of IPI interrupts.
    sw    k1, 0(k0)         // Clear this IPI.
```

There is an external array declared in main.c called start_test. Each element in the array corresponds to a processing element that is waiting to continue processing after its corresponding array element has been cleared. The following code clears the element of start_test for the processor that has taken the exception. It does this by writing the address of start_test to a register, getting the CORE number from the CP0 EBASE register, multiplying it by 4 (shift left 2) to get the correct byte offset into the start_test array (declared as a integer array, so each element is 4 bytes), and then writing a 1 to that array element.

```
    la    k0, start_test    // NOTE: dla will be used for MIPS64 address
```

```
    mfc0   k1, C0_EBASE      // read EBASE
    ext    k1, k1, 0, 10     // Extract CPUNum
    sll    k1, k1, 2         // x 4 for integer element
    addu   k0, k0, k1        // index into array
    li     k1, 1
    sw     k1, 0(k0)         // set element of array
    eret
    nop
```

4.9.4. Ejtag Exception:

For an EJTAG exception (which you should only get on live hardware that includes EJTAG), the code goes into an infinite loop.

```
.org 0x480 /* debug exception (EJTAG Control Register [ProbTrap] == 0.) */
1: b     1b    /* Stay here */
   Nop
```

4.9.5. Start of normal boot processing; NMI and ISA Verification

Recall that the code at the boot exception vector just branches to check_nmi. That is because the NMI exception vector is the same as the boot exception vector.

The NMI is handled in the next Block of code. If this was an NMI exception, the NMI bit (19) in the Status Register (CP0 register 12) will be set. The code first moves the value in the status register to a temp register, then shifts it to the right to put the NMI bit in the least-significant bit. Then it checks to see if it's 0, and if so, branches ahead to the verify_isa label. If not, it executes a sdbbp that will cause a break into the debugger, if attached.

```
.org 0x500 /* Resume code past the boot exception vectors. */
check_nmi: // Verify we are here due to a reset (and not NMI.)
   mfc0   a0, C0_STATUS          // Read CP0 Status
   srl    a0, 19                 // Shift [NMI] into LSBs.
   andi   a0, a0, 1              // Inspect CP0 Status[NMI]
   beqz   a0, verify_isa         // Branch if NOT an NMI
   nop
   sdbbp
```

This boot code was designed with a specific Instruction Set Architecture, ISA in mind, so it checks the CP0 Config register (16) to make sure the core is the proper ISA.

The AT field (bits 13 and 14) are is set in the core hardware to the Architecture Type and the AR field (bits 10 through 12) is set to the Architecture Release level. Both must be correct for the

code to continue. The code below reads the CP0 Config register, then shifts it right by 10 bits, leaving the AT and AR fields in bits 0 through 4. Then it masks off the AT bits (3 and 4) and branches ahead if they are correct. If they're not, then the code issues a break instruction to stop in the debugger, if attached.

```
verify_isa: // Verify device ISA meets code requirements (MIPS64R2 or later.)
    addu a0, zero, zero
    mfc0    a0, C0_CONFIG        // Read CP0 Config
    ext   a1, a0, 13, 2          // Extract Architecture Type (AT)
    li    a3, 2                  // load expected value for MIP642R6 ISA
    beq   a3, a1, is_mips64      // Branch if executing on MIP642R6 ISA.
    ext     a3, a0, 10, 2        // Extract Architecture Release (AR)
    sdbbp                        // Failed assertion: MIPS64R6.
```

Note: Expected value should be set to the correct value for the particular core being booted. The code shown is as expected for a MIPS64 R6 ISA

The code next checks the AR bits are set, which indicate the core is at least Release 2 of the ISA. If the check fails, the code issues a break instruction to stop in the debugger, if attached. The code is shown below:

```
is_mips64:
   bnez  a3, init_common_resources
   nop
   sdbbp // Failed assertion
```

4.9.6. Initializing Common Resources

The next section of start.S initializes resources that are common to every processing element in the core(s).

- For single-core single-threaded processors like the interAptivUP single VPE or P5600 single core, the functions in this section will be called once.
- For an interAptivUP multi VPE Core, this section will be executed by each VPE.
- For an interAptiv CPS, this section will be executed by each VPE on each Core.
- For a I6400/I6500 this section will be executed by each VP.
- For a P5600 and P6600 CPS, this section will be executed by each Core.

The actual functions called by the code will be covered in a section specific to the source file that contains that code. If viewing this electronically, you can follow the links to the section that contains the function that is called.

Each function call below begins by loading the address of the function name and then jumping to that address. The Jump and Link Register (jalr) instruction jumps to the address supplied by the register and puts the address of the instruction into the Return Address ($31/ra) register. This will be used by the called function to jump back to the next code to be executed.

```
   init_common_resources:  // initializes resources
```

init_gpr function sets all of the General Purpose Registers, including shadow register sets, to a known value.

```
la    a2, init_gpr      // Fill register file with boot info
jalr  a2
nop
```

set_gpr_boot_values sets the values for the General Purpose registers that will be used by the rest of the code.

```
la    a2, set_gpr_boot_values  // Set register info
jalr  a2
nop
```

init_cp0 initializes all CP0 Watch, Cause, Compare, and Config registers.

```
la a2,       init_cp0   // Init CP0 Status, Count, Compare, Watch*,
                        // and Cause.
jalr a2
nop
```

A TLB type MMU is optional on some cores. A check is done to see if this core has a TLB and if it does not it will branch around calling tlb_init.

```
// Determine if we have a TLB
mfc0  v1, C0_CONFIG           // read C0_Config
ext   v1, v1, 7, 3            // extract MT field
li    a3, 0x1                 // load a 1 to check against
bne   v1, a3, done_tlb        // no tlb?
nop
```

init_tlb initializes the Translation Look-a-side Buffers.

```
la a2,       init_tlb         // Initialize the TLB
jalr a2
nop
```

Note: Initializing the TLB is necessary before the TLB can be used however it is not strictly necessary for it to be initialized in the boot code or in the order shown here (for some cores this is done later in the Boot-MIPS core). For example if your using the Linux OS, Linux initializes the TLB so there is no need to do it in Boot-MIPS.

init_gic is only present for multiprocessor cores. It initializes the Global Interrupt Controller

```
done_tlb:
   la a2,       init_gic   // Initialize Global Interrupt Controller.
   jalr a2
   nop
```

This next check is only present for the interAptiv cores. For these cores, the code will only continue if it is executing on VPE0, because the rest of the code only needs to be done once per processor.

```
   bnez  r9_vpe_num, init_done // If this is not a vpe0, goto done.
   nop
```

The next check is only present for a interAptiv CPS. If this is not the first core in a CPS, the code will branch around the next section of code, because it only needs to be executed once per CPS (not once for each Core).

```
   bnez  r8_core_num, init_core_resources // continue for element 0
   nop
```

4.9.7. Initializing Core Resources

This section of code will be executed on a:

- M5150, M6250
- interAptivUP single VPE Cores always
- interAptivUP multi VPE Core only by VPE0
- interAptiv only VPE0 of Core 0
- I6400 only VP0 of Core 0.
- I6500 only VP0 of Core 0 of each Cluster
- P5600/P6600 only Core 0.

init_core_resources:

If executing on Core 0 disable the L2 cache.

```
// The L2 cache needs to be disabled because it has not been initialized
// Once the Icache has been initialized below caching will be turned on for
// KSEG0.
// This makes initializing the Dcache and the Icache much faster since the
// code will be cached. It would also cause the L2 cache to be used before it
has
// been initialized if it were not disabled here.
   bnez       r8_core_num, init_L1_icache    //  Only done from core 0.
   la a2,     disable_L2                     // Disable L2 caches
   jalr a2
   nop
```

The next two calls, init_icache and init_dcache, found in common/init_caches.S or common/init_caches2.S, will initialize the Level 1 Instruction and Data caches so they can be used from this point on.

```
   la a2, init_icache // Initialize the L1 Icache
   jalr a2
   nop
```

If this is not an EVA boot then before the code calls the init_dcache function, it enables the caches by setting the Cache Coherency Attribute (CCA) in the K0 field of the CP0 Config register. The Boot MIPS code executes in KSEG0, and up to now KSEG0 has been operating in uncached mode (CCA = 2). Now that the instruction cache has been initialized, the code

changes the CCA for KSEG0 to cacheable (3 or 5). All instructions will now be cached, so the code will run faster through the processor. All loads and stores will also be cached, so it is important not to use loads or stores until the Data cache has been initialized (in the code section following this code).

The trick is, the code that changes the CCA must be executed from KSEG1 addresses (not cacheable). This is done by setting bit 29 of the register holding the change_k0_ca address jump point and then uses the JALR instruction to jump to that address.

```
#ifdef EVA
   // Only for cores built to boot in EVA mode
   // continue in uncached mode
#else
   // The changing of Kernel mode cacheability must be done from KSEG1
   // Since the code is executing from KSEG0 It needs to do a jump to KSEG1
   // change K0 and jump back to KSEG0
   la    a2,change_k0_cca
   li    a1, 0xf
   ins   a2, a1, 29, 1 // changed to KSEG1 address by setting bit 29
   jalr  a2
   nop
#endif
```

Next the code calls the init_dcache function to initialize the Data Cache.

```
   la a2, init_dcache // Initialize the L1 D cache
   jalr a2
   nop
```

init_itc will initialize the Inter-thread Communication unit, if present.

```
   la a2, init_itc    // Initialize ITC
   jalr a2
   nop
```

4.9.8. Initialize System Resources

This next section of code will be executed only once per processor. It will be executed by each single Core of a multi core system, only on VPE0/VP0 of a multithreaded Core.

```
   // Only core0/vpe0 needs to init systems resources.
   bnez    r8_core_num, init_sys_resources_done
   nop.
init_sys_resources:    // for core0/vpe0.
```

init_cpc is only present for multi-processor systems. It initializes the Cluster Power Controller.

```
   la a2,  init_cpc // Initialize Cluster Power Controller
   jalr a2
   nop
```

init_cm is only present for multi-processor systems. It initializes the Coherence Manager.

```
   la a2, init_cm  // Initialize Coherence Manager
   jalr a2
   nop
```

This next section of code is compiled only if the code is being built for a BOSTON Board. It will initialize the memory controller. For SEAD boards, if you are adapting this code for your hardware, this is where you need to put the call to initialize your memory controller.

```
#ifdef BOSTON_RAM
   dla   a2, init_FPGA_mem
   jalr  a2
#endif
```

NOTE the dla instruction is for a 64bit MIPS 32bit MIPS cores will use la instead.

copy_c2_ram will copy the C code in main.c from the ROM memory area to RAM or Scratchpad RAM, depending on the Makefile target. It also copies initialized data and clears the uninitialized variables in the bss section.

```
   la a2, copy_c2_ram // Copy code/data to RAM, zero bss
   jalr a2
   nop
```

If not executing on a Coherent Processing system (Multi-core) then skip to VPE initialization.

```
   beqz    r11_is_cps, init_vpe_next // UP Core so skip multi core stuff
   nop
```

init_L2 initializes the L2 cache for multi-processor systems

```
   bnez       r8_core_num, release_cores     //  L2 Only done from core 0.
   nop

   la a2,      init_L2      // Init the L2 cache
   jalr a2
   nop
```

```
   la          a2, enable_L2 // enable L2 cache
   jalr  a2
   nop
```

release_cores is present for multi-processor systems (interAptiv and P5600 only) only. It is used to start them processing this boot code.

```
release_cores:
   la a2, release_mp      // Release other cores to execute
   jalr a2
   nop
```

```
init_sys_resources_done:          // All Cores (VPE0)
```

For a CPS with CM version 2.5 or lower:

join_domain associates the CORE with a Coherence Domain. NOTE: This is not done for a CPS that contains a version 3 of the Coherency Manager (CM3)

```
    la a2, join_domain // Join the Coherence domain
    jalr a2
    nop
```

init_vpe1 will setup the second VPE if present to run this boot code (interAptivUP or interAptiv only).

```
init_vpe_next:
   la a2, init_vpe1   // vpe1 to execute boot code
   jalr a2
   nop
```

For a CPS with CM3:

Each core needs to be powered up. The power_up_cores function will power up the cores and the VP0 of each core will start executing.

```
    dla a2, power_up_cores // Power up other cores
    jalr a2
    nop
```

At this point each processor on each core can be started to execute the Boot-MIP code. The code below does that. Your system may want to execute other code or wait for the OS to start the other processors.

```
      dla      a2, start_vps
      jalr     a2
      nop
```

4.9.9. Initialization Complete

The initialization is now complete for the executing Core, VPE or VP, and this is the point at which any setup needed for an OS should take place, after which the OS takes control of the system. This code example however, instead of call an OS sets up arguments to main and then executes a return from exception (necessary because all of the code so far has been part of the Boot exception handler).

init_done:

The next code is included only if this is an EVA boot. It will change the memory map so that after the eret instruction is executed. This is done for a BOSTON board configuration and for the purposes of this boot example code. The changes make segments CFG4 and 5 coherent which allows global variables to be coherent across all cores. CFG1 and CFG0 segments are mapped uncached so that the Coherency Manager memory mapped registers can be access through the virtual address in those sections. Whether your code should do the same as this will depend on your memory layout.

```
#ifdef EVA
// For non-EXL Kernel mode - Segments CFG5 and CFG4 are directly mapped to the
// lower 2GB of the physical address space encompassing the 2 RAM memory
// blocks
// and the I/O space to be accessed as coherent cached exclusive on write.
// For user and supervisor modes - Segments CFG5 and CFG4 are mapped through
// the
// TLB
   li          a0,
(SEGCTL_CFG4_PA_4|SEGCTL_CFG5_PA_0|SEGCTL_CFG4_AM_MUSUK|SEGCTL_CFG5_AM_MUSUK|S
EGCTL_CFG4_EU_MASK|SEGCTL_CFG5_EU_MASK|SEGCTL_CFG4_C_CWBE|SEGCTL_CFG5_C_CWBE)
   mtc0  a0, C0_SEGCTL2
// Set CFG1 and CFG0 to be UnMapped Kernel to Physical address 0x00000000 and
// 0x20000000 respectively both uncached. This will be used to address the CM
// registers and
// can be used for I/O devices. NOTE: This is needed to access the CM
// registers
// uncached
```

```
    li          a0,
(SEGCTL_CFG2_PA_0|SEGCTL_CFG3_PA_0|SEGCTL_CFG2_AM_UK|SEGCTL_CFG3_AM_UK|SEGCTL_
CFG2_EU_MASK|SEGCTL_CFG3_EU_MASK|SEGCTL_CFG2_C_UC|SEGCTL_CFG3_C_WB)
    mtc0  a0, C0_SEGCTL1
    ehb
#endif
```

The code will put the address of the all_done label in the Return address register (ra/ra), so if main returns it will go to that code (which is just a for ever loop).

// Prepare for eret to main (sp and gp set up per vpe in init_gpr)

```
    la    ra, all_done      // main's return
```

Before the code executes an eret (exception return), it must first change the address it will return to. Normally the core uses the address of the instruction that was executing when the exception occurs, which in this case is the boot exception vector. So if that has not changed, the code will loop through the boot code forever. In this case, the code places the address of the main function into the CP0 ErrorEPC register, so that when the eret is done, that is the code that will start executing. If an OS is to be started, then use the address of the start of the OS entry point instead of the address to main.

```
    la    a1, main
    mtc0  a1, C0_ERRPC            // ErrorEPC
```

For Coherent Processing Multi-Cores, the external variable num_cores is set. num_cores is declared and used in main.c. The code here loads the address of the variable and makes it an uncached address (by setting bit 29) so that it will be globally written to memory. Then the code uses the value in r19_more_cores ($19/s3) and adds 1 to it to account for core 0 (r19_more_core was set in set_gpr_boot_values.S).

```
    // initializes global variable num_cores
    la    a1, num_cores
    #ifndef EVA  // EVA boot mode not cached or mapped the same way
          ins   a1, r1_all_ones, 29, 1 // Uncached kseg1
    #endif
    add   a0, r19_more_cores, 1
    sw    a0, 0(a1)
```

Before main() begins executing, the code sets up the arguments it will use. These arguments will vary depending on the MIPS core family being booted. The temp registers 4 through 7 correspond to the argument registers in the MIPS ABI (GPR registers 4 – 7, also known as a0 through a3).

```
    // Prepare arguments for main()
    move  a0, r23_cpu_num         // main(arg0) is the "cpu" number
    move  a1, r8_core_num         // main(arg1) is the core number.
```

```
   move  a2, r9_vpe_num          // main(arg2) is the vpe/vp number.
   addiu a3, r20_more_vpes, 1    // main(arg3) is the number of vpe/vp
```

The boot of the core or VPE/VP is now complete. Executing the eret instruction will bring the core out of exception mode and start execution at the address in ErrorEPC (which was set to the address of main above).

```
   eret  // Exit reset exception handler
```

The all_done label is used for the return address of main(). It is not expected that main will return. main would normally be the stat of the OS and OS's usually just go into a control loop that never exists. If main exited, it would return to this never ending loop.

```
all_done:
   // Looks like main returned. Just busy wait spin.
   b     all_done
   nop
```

4.10. set_gpr_boot_values.S

The boot code names General Purpose Registers and assigns them specific purposes. The boot.h section already covered the naming of the registers. The set_gpr_boot_values.S source file assigns values to many of these registers. The register assignment can be divided into two types, one that assigns registers according to the O32 API (such as the global pointer), and one that holds an attribute of the core. The API assignment is standard for every core, but since each core can have different attributes, each core's version of set_gpr_boot_values.S can differ.

It should also be noted that there is an underlying style in this boot code that you don't necessarily have to follow for your system. The boot code is divided into core sections with each only compiling in what is needed for that core. However, there is still code that makes some runtime decisions. To make this code even smaller and slightly faster, you can customize it for your specific core and remove those decision points. That work is left up to the reader.

```
LEAF (set_gpr_boot_values)
```

The first register assignment is r1_all_ones. This sets GPR 1 to all ones. It will be used many times in the code in conjunction with the insert instruction. It simplifies the code because we can use it over and over again without having to set up a register with ones each time we use the insert instruction.

```
   li    r1_all_ones, 0xffffffff // Simplify code and improve clarity
```

The code reads the EBASE register and extracts the core number into r23_cpu_num (r23/s7).

```
   mfc0  a0, C0_EBASE            // Read CP0 EBASE
   ext   r23_cpu_num, a0, 0, 4   // Extract CPUNum
```

The Global pointer is common to all processing elements. Its address is defined in the linker file and set by the linker. This address will be used to reference shared global variables. The MIPS API designates that GPR 28/gp be used to hold the global pointer address, so the code sets it here.

```
   la    gp, _gp                 // Shared globals.
```

Part of each processing element's context is its own stack. The stack is used to hold local variables while executing a function. It also holds other context such as GPR values that are saved to the stack when a function is called, and then restored when returning from a function

call. In this case, a constant named STACK_BASE_ADDR is #defined in boot.h to point to memory designated for use by processor stacks. The MIPS API designates that GPR 29/sp be used to hold the stack pointer. The code first writes the STACK_BASE_ADDR to sp, then manipulates it using the VPE or CORE number so that each processing element will have its own stack.

```
    li    sp, STACK_BASE_ADDR
    ins   sp, r23_cpu_num, STACK_SIZE_LOG2, 3 // stack.
```

4.10.1. VP Cores (I6400/I6500)

For cores with virtual processors (VP) this register contains the VP number. The way VP numbers are assigned to each Core in these is as follows; each core has 4 bits reserved for its possible 4 VPs like this:

Core	VP number range
0	0-3
1	4-7
2	8-11
3	12-15
4	15-19
5	20-23

For a core with VPs it means that the lower 2 bits will represent the VP number with in each core and the upper 8 bits are the cores number. So really there 3 different pieces of information within the VP number field, the VP number within the entire CPS, the VP number within the individual core and the individual core number. The for these virtual processor cores the Boot-MIPS code will use 2 of these, the VP number within the entire CPS and the VP number within the individual core.

```
   mfc0  a0, C0_EBASE            // Read CP0 EBase
   ext   r23_cpu_num, a0, 0, 10  // Extract core Number (VP number in CPS)
   ext   r9_vpe_num, a0,  0, 2   // Extract VP Number within the core
```

4.10.2. MT ASE Check (interAptivUP or interAptiv Only)

The next sections of code are only present for interAptivUP or interAptiv. An MT core has CP0 Registers Config 1, 2, and 3, and the MT bit will be set in the Config 3 register. But you can't just read the Config 3 register and see if the MT bit is set, because on non-MT processors, there won't be a Config 3 register, and the operation of trying to read the Config 3 register will have undetermined results (in other words, nothing good will happen).

To read Config 3 properly, the code must first read the Config 1 register and check to make sure the M bit is set. The M bit in the Config1 register indicates whether or not there is a Config2 register. The M bit is bit 31 in the Config1 register. If this register is treated as a signed integer, this bit would be the sign bit, and if the bit is set, the register value would appear as a negative number or a number less than 0. The simplest way to test the bit is to check if the register value is greater than 0, using the branch greater than or equal to zero instruction. The code then looks at the Config2 register and its M bit in the same manner. The code reads the config3 register and isolates the MT bit. Bit 2 tests it and branches to the no MT ASE function if it is not set.

```
check_mt_ase:
   mfc0  a0, C0_CONFIG, 1              // read C0_Config1
   bgez  a0, no_mt_ase                 // No Config2 register
   mfc0  a0, C0_CONFIG, 2              // read C0_Config2
   bgez  a0, no_mt_ase                 // No Config3 register
   mfc0  a0, C0_CONFIG, 3              // read C0_Config3
   and   a0, (1 << 2)                  // MT
   beqz  a0, no_mt_ase
   li    r10_has_mt_ase, 0
```

If the code has determined that it is executing on an MT processor, it will set r10_has_mt_ase to 1. It will use this register in cases where it needs to do special configuration for MT.

The rest of the code will save MT-specific data in specific registers.

```
has_mt_ase:
   li    r10_has_mt_ase, 1
```

It reads the CP0 TCBind register and saves the number of the VPE context in which it is currently executing into r9_vpe_num. It will save the number of the TC it is executing in r18_tc_num.

```
// Every vpe will set up the following to simplify resource initialization.
   mfc0  a0, C0_TCBIND           // Read CP0 TCBind
   ext   r9_vpe_num, a0,  0, 4   // Extract CurVPE
   ext   r18_tc_num, a0, 21, 8   // Extract CurTC
```

Next it will read the CP0 MVPConf0 and set r21_more_tcs to the number of TC in the Core and set r20_more_vpes to the number of VPE contexts in the Core. Then the code will branch to check if this is a coherent processing system.

```
mfc0  a0, C0_MVPCONF0          // read C0_MVPConf0
ext   r21_more_tcs, a0, 0, 8
b     check_cps
      ext   r20_more_vpes, a0, 10, 4
```

4.10.3. No MT ASE (interAptivUP single VPE /thread P5600)

If the code is executing on a non-MT core, the MT core-specific values will be set to zero.

```
no_mt_ase: // This processor does not implement the MIPS32 MT ASE.
   li    r9_vpe_num, 0
   li    r18_tc_num, 0
   li    r20_more_vpes, 0
   li    r21_more_tcs, 0
```

4.10.4. Check for Coherent Processing System (interAptiv, P5600, P6600 or I6400/I6500)

Now the code needs to determine if it is running on a coherent multi-core system. It does this by reading the CP0 Processor ID register into r25_coreid. The code extracts the Core ID and interAptiv/P5600 Implementation bits and then compares them with the values for the specific core to determine if this is a Coherent Core. If it is, it branches to setting up the Coherence Manager GPR registers.

```
check_cps: // Determine if there is a Coherence Manager present


   mfc0    r25_coreid, C0_PRID         // CP0 PRId.
   ext     a0, r25_coreid, 8, 16       // Extract Manufacture and Core.
   li      a3, 0x01A1                  // interAptiv Multi core
   beq     a3, a0, is_cps
   nop
   li      a3, 0x01A0                  // Check for interAptivUP
   beq     a3, a0, is_not_cps
   nop
```

4.10.5. Is a Coherent Processing System (interAptiv, P5600, P6600 and I6400/I6500)

If the code determined that it is executing on a Coherent Processor, it sets r11_is_cps to 1 to indicate we have a Coherent Processor. r11_is_cps will be used in several places in the code to branch to the appropriate execution path.

```
is_cps:
   li    r11_is_cps, 1
```

A Coherent Processing System contains a structure called the Global Control Block that determines the configuration of the system. This structure contains registers, the Global Control Registers or GCRs, that can be read to determine the configuration of elements within the CPS. Many of the registers can also be written to change the CPS configuration.

To verify that we have a correct Global Control Block address, the code will compare the given address of the control block with the one stored within the block itself located in the GCR Base register. The given address is set by a #define in boot.h. Consult your SOC designer to determine the value of this “#define”. If the given address is not the same as the address in the

GCR Base register, something is wrong, and this system should not be treated as a Coherent system. If it is equal, the code loads the given address of the GCR Configuration Block into a1.

```
// Verify that we can find the GCRs.
la    a1, GCR_CONFIG_ADDR       // KSEG1(GCRBASE)
```

Then it loads the GCR Base register that is located at byte offset 8 into a0.

```
lw    a0, GCR_BASE(a1)              // read GCR_BASE
```

The GCRs are located in the memory map on a 32K-byte boundary so the lower 15 bits of the address will always be 0. The GCR Base register uses these lower bits to store additional information. Therefore to get the correct physical address the code needs to clear these bits that are now stored in a0.

```
ins   a0, zero, 0, 15         // Isolate address of GCR.
```

The value in the GCR Base register is a physical address, so before the code compares the given value, it must convert it to a physical address. That's done by simply clearing the top 3 bits using the insert instruction and zero. (Note that ERL is set while executing this boot code, so this step turns the address into a direct mapped address, where virtual equals physical address.) This line of code takes the first 3 bits of zero, which is always 0, and inserts them starting at bit 29 into a1.

```
ins   a1, zero, 29, 3         //Convert KSEG1 to physical address.
```

The code checks to make sure the two GPRs are equal and branches to the gcr_found function if they are, or issues a debug break instruction to stop execution.

```
beq   a1, a0, gcr_found
nop
sdbbp   // Can't find GCR RTL config override of MIPS default
```

Now that the code has determined it has valid GCRs, it will save their address in r22_gcr_addr.

gcr_found:

```
li    r22_gcr_addr, GCR_CONFIG
```

The code stores the GCR_CL_ID in r8_core_num. The GCR_CL_ID is the number of the core that is executing this code within the Coherent Processing system. The GCR_CL_ID is located within the Core-Local Control Block. The Core-Local Control Block is located at offset 2000 hex from the GCR Base address, and the GCR_CL_ID is located at offset 28 hex within the Block. Putting these together results in offset 2028 hex from the GCR Bass address.

```
lw r8_core_num, (CORE_LOCAL_CONTROL_BLOCK + GCR_CL_ID)(r22_gcr_addr)
```

The code now saves the total number of Cores in the system. This information is stored in the GCR_CONFIG register located at offset 0 from the GCR Base. Bits 0 through 7 contain the value, so these bits are extracted from the register value and stored in r19_more_cores.

```
lw    a0, GCR_CONFIG (r22_gcr_addr)             // Load GCR_CONFIG
ext   r19_more_cores, a0, PCORES, PCORES_S      // Extract PCORES
b     done_init_gpr
nop
```

4.10.6. Not a Coherent Processing System (interAptivUP)

For non-CPS systems, the code clears the GPR registers that are assigned to deal with a Coherent Processor.

```
is_not_cps: // This processor is not part of a Coherent Processing System. Set
up valid defaults.
   Li    r11_is_cps, 0
   li    r8_core_num, 0
   li    r19_more_cores, 0
```

4.10.7. Done with set_gpr_boot_values

We are now done with the init_gpr function and the code returns to the calling function, init_common_resources located in start.S in ‘Initializing Common Resources’ on page 48.

```
done_init_gpr:
   jr    ra
   nop
```

4.11. common/copy_c2_ram.S

This example boot code shows how to place C code in ROM that will later be copied to RAM or SPRAM. How to place the code in ROM is covered in the linker file section. This section covers the copying of the C code from ROM to RAM. NOTE: There are variants of this code copy_c2_ram_R6.S which is the same basic code but uses MIPS32 release 6 compact branch instructions and copy_c2_ram_R6_64.S which in addition adds the use of double word load and store to take advantage of the 64 bit registers.

There are a few defines to make the code easier to read.

```
#define all_ones_s1         s1   /* to simplify bit insertion of 1's. */
#define data_a0             a0   /* a0 data to be moved */
#define source_addr_a1       a1   /* from address */
#define destination_addr_a2 a2   /* to address */
#define END_ADDR            a3   /* ending address */
```

The copy_c2_ram function starts by putting the first address of the “C” code’s text section into source_addr_a1. Then `_start_rom_text`, created in the Linker script, locates the area right after all the init code in the flash memory. The `_start_rom_text` address is the start of the “C” code that will be copied to RAM. In other words it is the copy from address.

```
LEAF(copy_c2_ram)
   li    all_ones_s1, 0xffffffff
   // Copy code and read-only/initialized data from FLASH to (uncached) RAM.
    LA      source_addr_a1, _start_rom_text
```

Next the code stores the _ftext_ram value into destination_addr_a2. _ftext_ram is also created in the linker file. It is the start of the “C” code section that will be copied to. In other words, it is the copy to address

```
   LA      destination_addr_a2, _ftext_ram
```

The _edata_ram is stored in END_ADDR. _edata_ram is created in the linker file and is the address of the end of the initialized data section. The code will use this address to end the copy of the code and initialized data sections.

```
   LA      END_ADDR, _edata_ram
```

If this is not an EVA boot, the `_start_rom_text` address is a cached address in KSEG0. Since we haven’t yet initialized the caches, we don’t want to use this cached address. As you should know, in the MIPS architecture KSEG0 and KSEG1 are two virtual address sections that access the same physical addresses. Accesses to KSEG0 are first looked for in the cache, whereas addresses in KSEG1 go directly to memory and never access the cache. KSEG0 and KSEG1 addresses differ only in their three most-significant bits;- the rest of the address bits are the same. KSEG0 addresses have the top three bits set to 100, and KSEG1 addresses have the top three bits set to 101. For example, the KSGE0 cacheable address 0x8001 0000 and the KSEG1 uncached address 0xA001 0000 access the same physical memory location. What this code does is convert the KSEG0 address into a KSEG1 address by inserting a 1 into bit 29, changing the top byte from an 8 to an A.

```
    #ifndef EVA  // NOTE EVA mode assumed to be uncached
    // Switch address to uncached (kseg1) so copy will go directly to memory
         ins     destination_addr_a2, all_ones_s1, 29, 1
```

```
        ins     END_ADDR, all_ones_s1, 29, 1
    #endif
```

The code checks to make sure we have anything to copy by comparing the start of the code and data address with the end address. If there is nothing to copy, the code will skip around the copy and proceed to the clearing the uninitialized variable section (For this example, there should always be something to copy).

```
    beq     destination_addr_a2, END_ADDR, copy_c2_ram_done
    nop
```

The copy is simply reading from the location where the “C” code and data is stored in flash (source_addr_a1) and writing it to its destination address (destination_addr_a2) in RAM.

```
next_ram_word:
    lw      data_a0, 0(source_addr_a1)
    sw      data_a0, 0(destination_addr_a2)
```

The source and destination addresses are incremented by 4, the number of bytes in a word, and the code checks to see if it still has more to copy by using END_ADDR which is the end address and the current destination address.

```
    addiu   destination_addr_a2, 4
    bne     END_ADDR, destination_addr_a2, next_ram_word
    addiu   source_addr_a1, 4
```

Now the code turns its attention to the uninitialized variable section (also known as the bss section, which strangely enough stands for Block Started by Symbol). It is mandated by the C specification that the bss section be initialized to 0 before a program starts. This clearing of the bss section usually is done by the program loader. It is the responsibility of the boot loader to clear the first bss section before calling the main “C” function.

This code is similar to the code we just went through for the copy. It uses two values created in the linker script. _fbss is the first address of the bss section and _end is the end address of the bss section. If not a EVA boot, it converts both those addresses to uncached KSEG1 addresses. Then the code checks to see if there is bss to clear by seeing if they are equal.

```
zero_bss:
      LA      destination_addr_a2, _fbss
      LA      END_ADDR, _end
      #ifndef EVA  // NOTE EVA mode assumed to be uncached
            // Switch address to uncached (kseg1) so copy will go directly
            // to memory
            ins     destination_addr_a2, all_ones_s1, 29, 1
            ins     END_ADDR, all_ones_s1, 29, 1
      #endif
      beq     destination_addr_a2, END_ADDR, copy_c2_ram_done
      nop
```

The label next_bss_word will be used as a loop point. The code stores a zero using the zero register to the destination address in destination_addr_a2. It then adds 4 bytes to the destination address, checks to see if it is at the end of the copy by comparing it to the end address stored in END_ADDR , and loops back if it is not.

```
next_bss_word:
   sw      zero, 0(destination_addr_a2)
   addiu   destination_addr_a2, 4
   bne     destination_addr_a2, END_ADDR, next_bss_word
   nop
```

The code has finished the copy and returns.

```
copy_c2_ram_done:
   jalr  zero,      ra
   nop
END(copy_c2_ram)
```

4.12. common/copy_c2_SPram.S

You may have a system that uses Scratchpad RAM instead of regular RAM, or uses both, and you want to copy the main code to the Scratchpad RAM. The copy_c2_Spram.S should be used in place of the copy_c2_ram.S. The copy to Scratchpad RAM requires the memory controller to setup the SRAM for the copy. Also, there is a difference in the layout requirements for SRAM, namely that there has to be one Scratchpad RAM for instructions and one for data. This means that the code must be split to copy the instructions to the Instruction Scratchpad RAM using cache instructions, and the data to the data Scratchpad RAM using regular loads and store instructions.

Here are some #defines to make the code easier to read:

```
#define s0_save_C0_ERRCTL  s0 /* use s0 only to save C0_ERRCTL */
#define all_ones_t4        t4   /* at Will hold 0xffffffff to simplify bit
insertion of 1's. */
#define data_a0            a0   /* a0 data to be moved */
#define source_addr_a1     a1   /* from address */
#define destination_addr_a2 a2   /* to address */
#define END_ADDR     a3   /* ending address */
#define TEMP_t5            t5
#define CODE_INDEX         s1   /* code index */

   .set  noat               # Don't allow the assembler to use r1(at)
                            # for synthetic instr.
```

4.12.1. Copy to Instruction Scratch Pad

The next few lines of code set the starting address of the ISPRAM in the ISPRAM controller. To clarify further, while the physical address of the ISPRAM can be set at core build time, it can be changed by software to place it anywhere in physical memory. The code here is changing the physical address of the ISPRAM to match the address where the main.c code was linked. The code assumes that the system is not using a TLB but instead uses Fixed Mapping Translation (FMT). With FMT, KUSEG starts at virtual address 0 and maps to Physical address 0x4000 0000. In this example, the main.c code is linked to virtual address 0x1000 0000, so the ISPRAM is placed at physical address 0x5000 0000 (_ISPram = 0x5000 0000).

The “cache” instruction is used to program the ISPRAM physical address and fill it with instructions. The “cache” instruction does this by writing the tag registers to the Scratchpad controller. There are two tag registers for each Scratchpad RAM, one set for the ISPRAM and one set for the DSPRAM. Tag 0 is located at offset 0 and tag 1 is located at byte offset 8 into the Scratchpad controller. Here is a table that shows what bit and tags contain information.

I or D Tags					
tag	31 20	19 12	….	7	6 0
0	Physical Base Address			E	

1		Size	0

As shown in the table, the physical address is located in tag 0, bits 12 through 31 (4K boundary), and the Enable bit is located in tag 0 at bit 7. Both of these bits are read/write. The size in 4K sections is located in tag 1, bits 12 through 19.

The following code will place the physical address of the ISPRAM into the CP0 C0_TAGLO register. The code puts _ISPram in source_addr_a1 then moves it to the C0_TAGLO register.

```
LA    source_addr_a1, _ISPram
mtc0  source_addr_a1,C0_TAGLO
```

The "cache" instruction will then be used to program the instruction Scratchpad controller with the value stored in the C0_TAGLO register. By default, the cache instruction directs all of it operations to the cache controller. The code needs to change this, so that the cache operations are directed to the Scratchpad controller. It does this by setting the SPR bit (28) in the CP0 Error Control register (26, 0).

The code reads the C0_ERRCTL register, makes a copy so that later it can be restored to its current state, sets the SPR bit, and writes the value back to the C0_ERRCTL register.

```
mfc0 s0_save_C0_ERRCTL, C0_ERRCTL
move TEMP_t5, s0_save_C0_ERRCTL    # make copy so we can restore C0_ERRCTL
ins  TEMP_t5, all_ones_t4, 28, 1
mtc0 TEMP_t5,C0_ERRCTL
```

Now the code can use the cache instruction to write the Instruction Scratchpad tag.

Here is the instruction format of the cache instruction:

```
cache op, offset(base)
```

The "*op*" is encoded with two pieces of information: bits zero and one tell the cache instruction which Scratchpad block the operation will be performed on:

- 00 sets it for the Instruction Scratchpad
- 01 sets it for the Data Scratchpad

Bits two, three, and four of the "*op*" tell the instruction which operation to perform:

- 001 will load a tag
- 010 will store a tag
- 011 will store data into the Scratchpad blocks memory

The offset and base register control which of the two possible tags the operation will be performed on, or which address within the Scratchpad block the data will be stored to.

The code will use 8 (010 00) as the *op,* bits 0, 1 = 00 = Instruction Scratchpad bits 2, 3, and 4 = 010 = store a tag. Since tag 0 is being written the offset is 0 and the Base address is 0, it uses GPR 0 (which is always 0).

```
cache 0x8,0(zero)
```

First check to see if there is an Instruction Scratchpad RAM by reading the CP0 Config register. If there is, the ISP bit in the Config register will be set. So the code extracts the ISP bit (bit 24) and checks to see if it's 0. If it is, it assumes there is no Scratchpad RAM and branches to the end of the function. If it is set, the code falls through to the next instruction.

```
    mfc0 TEMP_t5,C0_CONFIG
    ext  TEMP_t5, TEMP_t5, 24, 1
    blez TEMP_t5, set_dspram #no ISPram
```

Next the code sets up the register to hold the virtual ROM address to copy from. First it loads source_addr_a1 with the address to copy from using the value _zap1, which is declared in the linker and set by the linker at link time. This address is a cached address, and because we might not have a cache, the code converts the address to an uncached address by setting bit 29.

```
    LA      source_addr_a1, _zap1 # starting ROM address
    ins     source_addr_a1, all_ones_t4, 29, 1 # convert to uncached address
```

The code sets up destination_addr_a2 register to hold the virtual memory address to copy to.

```
    LA      destination_addr_a2, _ftext_ram # starting ram address to copy to
```

Then the code sets up the END_ADDR register to mark the end address of the copy.

```
    LA      END_ADDR, _etext_ram # ending ram address
```

Now it compares the starting address with the end address and will jump ahead if there is nothing to copy.

```
   beq destination_addr_a2, END_ADDR, zero_bss #if = there is nothing to do
```

The Instruction Scratchpad memory cannot use the simple approach of using stores to write to it, because it is not attached to the load store unit of the core, only to the fetch unit, so the "cache" instruction must be used to fill the Instruction Scratchpad memory array. Therefore it doesn't actually use the destination addresses. Instead, the instruction Scratchpad is treated as an array of words (4 bytes each). The code uses a register to store the base array element within the Instruction Scratchpad array where the code will be loaded, which will be used by the "cache" instruction. According to the way the linker script has laid out the code and the way the code has used values set in the linker script, the first instruction should be loaded into location 0

The code loads the initial value into CODE_INDEX, which will be used as the first index to be written to.

```
   # Need to set the position within the ISPram to write first
   # instructions to
   # _code_index is defined in the linker file
   LA          CODE_INDEX, _code_index
```

The code needs to take into account the endianness of the core because it fetches instructions two at a time. The endianness will affect the order in which the instructions are stored in the Instruction Scratchpad array. To determine the core endianness, the code uses the value stored in TEMP_t5, where it previously stored the CP0 Config register. It extracts the BE (bit 15) from TEMP_t5. If this bit is set, the core is big endian; if not, it's little endian.

```
    ext  TEMP_t5, TEMP_t5, 15, 1
    blez TEMP_t5, next_Iram_wordLE
```

The code for big and little endian is the same except for the order in which instructions are stored in the Instruction Scratchpad array, so just the big endian version will be described.

Instructions are loaded into the Instruction Scratchpad array by the “cache” instruction, two at a time, by loading the two instructions into CP0 register C0_DATAHI and C0_DATALO before the cache instruction is executed.

Recall that source_addr_a1holds the current copy-from address, a2 holds the current copy-to address, and a3_temp_mark holds the ending address (in RAM).

The code loads the data from a1_temp_addr into a0_temp_data and then moves a0_temp_data value to the C0_DATAHI register. Then it increments the address by one word (4 bytes), loads the data from source_addr_a1into data_a0, and then moves the data_a0 value to the C0_DATALO register.

```
next_Iram_wordBE:
    lw          data_a0, 0(source_addr_a1)
    mtc0        data_a0,C0_DATAHI
    addiu       source_addr_a1, 4
    lw          data_a0, 0(source_addr_a1)
    mtc0        data_a0, C0_DATALO
```

The “op” will use C *(*011 00) as the *op,* bits 0, 1 = 00 = Instruction Scratchpad bits 2, 3 , and 4 = 011 = load data into the Scratchpad blocks memory. The base address in the array is stored in GPR 11 and the offset from the base address is 0.

```
   cache 0xc,0(CODE_INDEX)
```

The Base and the destination addresses are then incremented by two instructions (8 bytes).

```
   addiu   CODE_INDEX, 8
   addiu   destination_addr_a2, 8
```

The “from” address is incremented.

```
   addiu   source_addr_a1, 4
```

The current destination address is compared to the ending address and branches to the top of the copy loop if they are not equal.

```
    Bne  destination_addr_a2, END_ADDR, next_Iram_wordBE
```

The code branches around the little endian copy when the loop falls through.

```
   b     set_dspram
```

Skipping the little endian copy …….

4.12.2. Copy to Data Scratch Pad

The next step is to copy the initialized data. Copying to the Data Scratchpad is similar to the Instruction Scratchpad Copy, but only uses the DDATALO register to load the DSPRAM. Since there is only one word written at a time, there is no need for a Big/Little version of the code.

First the code reads the CP0 Config register (CPO Register 16,0) and extracts the DSP bit. If it is set, the code continues setting up the Data Scratchpad.

```
set_dspram:
```

```
    mfc0 TEMP_t5,C0_CONFIG
    ext  TEMP_t5, TEMP_t5, 23, 1
    blez TEMP_t5, copy_c2_ram_done #no DSPram just exit
```

The code sets the physical address of the Data Scratchpad by moving the DSPRAM value (defined in the linker script) into a register and then setting the enable bit (7). Then it moves the source_addr_a1 to C0_DTAGLO.

```
   LA    source_addr_a1, _DSPram
   # set the enable bit
   Ins   source_addr_a1, all_ones_t4, 7, 1
   # move it to the  tag register
   mtc0  source_addr_a1, C0_DTAGLO
   # write data tag 0 using the cache data controller
```

The “op” for the cache instruction will use 9 (010 01) , bits 0, 1 = 01 = Data Scratchpad bits 2, 3 and 4 = 010 = store a tag. Since tag 0 is being written, the offset is 0 and the Base address is 0 so it uses zero (which is always 0)

```
   // write data tag 0 using the cache instruction
   cache 0x9,0(zero)
```

Next the code sets up the register to hold the virtual ROM address to copy from. First it loads source_addr_a1with the address to copy from. _zap2 is declared in the linker and set by the linker at link time. This address is a cached address. Since we may not have a cache the code converts the address to a cached address by setting bit 29.

```
    LA      source_addr_a1, _zap2 # starting ROM address
    ins     source_addr_a1, all_ones_t4, 29, 1 # convert to uncached address
```

The code sets up destination_addr_a2 to hold the virtual memory address to copy to.

```
    LA      destination_addr_a2, _fdata_ram # starting ram address to copy to
```

Then the code set up the END_ADDR register to mark the end address of the copy.

```
    LA      END_ADDR, _edata_ram # ending ram address
```

Now it compares the starting address with the ending address and will jump ahead if there is nothing to copy.

```
    beq     destination_addr_a2, END_ADDR, zero_bss // if = nothing to do
```

The copy is simply reading from the location where the “C” code is stored in flash (a1_temp_addr), moving that value to the DDataLo register and issuing the cache instruction to write the DSPRAM at the index stored in CODE_INDEX. Then the index is incremented to the net word to be written in the DSPRAM.

```
next_Dram_word:
    lw          data_a0, 0(source_addr_a1)
    mtc0        data_a0,C0_DDATALO
    cache       0xd,0(CODE_INDEX)
    addiu       CODE_INDEX, 4
```

The source and destination addresses are incremented by 4, the number of bytes in a word and the code checks to see if it still has more to copy by checking END_ADDR which is the end address and the current destination address to see if they are equal.

```
    addiu   destination_addr_a2, 4
    addiu   source_addr_a1, 4
    bne     destination_addr_a2, END_ADDR, next_Dram_word
```

Now the code turns its attention to the uninitialized variable section also known as the bss section which strangely enough stands for Block Started by Symbol. It is mandated by the C specification that the bss section be initialized to 0 before a program starts. This clearing of the bss section usually is done by the program loader. It is the responsibility of the boot loader to clear the first bss section before calling the main “C” function.

This code is similar to the code we just went through for the copy. It uses two values created in the linker script. _fbss is the first address of the bss section and _end is the end address of the bss section. It converts both those addresses to uncached KSGE1 addresses. Then it checks to see if there is anything to copy by seeing if they are equal.

```
zero_bss:
    LA      source_addr_a1, _fbss       // start address
    LA      END_ADDR, _end              // end address
    beq     source_addr_a1, END_ADDR, copy_c2_ram_done
```

The code moves a 0 to the DDataLo register so that the entries it writes to the DSPRAM will be initialized to 0.

```
   // assume bss follows the initialized data
   // write a 0 to the DDataLo register
   mtc0  zero, C0_DDATALO // set to 0
```

The label next_bss_word will be used as a loop point. The code stores a zero using the zero register to the destination address in source_addr_a1. It then adds 4 bytes to the destination address, checks to see if it is at the end of the copy by comparing it to the end address stored in END_ADDR, and loops back if it is not.

```
next_bss_word:
    cache       0xd,0(CODE_INDEX)       // write DDATA LO to DSPRam
    addiu       source_addr_a1, 4
    addiu       CODE_INDEX, 4           // add 4 to DSPRam index
    bne         source_addr_a1, END_ADDR, next_bss_word
```

The copy is now done, but there is still some cleanup to do. The code needs to enable the Instruction Scratchpad RAM so that instructions can be fetched from it. The code loads the address _ISPram into source_addr_a1, sets the enable bit, bit 7, and moves that value to the C0_TAGLO register. Then it executes an “ehb” instruction to ensure any hazard barrier is cleared before it issues the cache instruction.

```
copy_c2_ram_done:
   LA    source_addr_a1, _ISPram
   # set the enable bit
   Ins   source_addr_a1, all_ones_t4, 7, 1
   # move it to the tag register
    mtc0 source_addr_a1, C0_TAGLO
```

```
    ehb
```

The code then executes the cache instruction with the same op it used to write the Instruction Scratchpad address.

```
   // write instruction tag lo using the cache instruction
   cache 0x8,0(zero)
```

Finally, to insure that the cache instruction will be directed to the caches instead of the Scratchpads, the code restores the C0_ERRCTL using the saved value in s0_save_C0_ERRCTL, then returns to the start code.

```
   // restore C0_ERRCTL
   mtc0   s0_save_C0_ERRCTL,C0_ERRCTL
   jr     ra
END(copy_c2_ram)
```

4.13. common/copy_c2_Spram_MM.S (M5100 and M5150 cores)

You may have a system that uses Scratchpad RAM instead of regular RAM or one that uses both, and you want to copy the main code to the Scratchpad RAM. The copy_c2_Spram.S should be used in place of the copy_c2_ram.S. For the copy to RAM, the memory controller was setup before the copy was done. The copy to Scratchpad RAM needs to set up the Scratchpad RAM using the cache controller before it can perform the copy, so there is an additional step that needs to be done here to do that setup.

There is also another difference in the Scratchpad memory layout: there must be one Scratchpad RAM for instructions and another for data. This means that the code needs to be split to copy the instructions to the Instruction Scratchpad RAM using cache instructions and the data to the data Scratchpad RAM using regular loads and store instructions.

Here are some #defines to make the code easier to read:

```
#define s0_save_C0_ERRCTL        s0 /* use s0 only to save C0_ERRCTL */
#define all_ones_v0              v0   /* to simplify bit insertion of 1's. */
#define data_a0                  a0   /* a0 data to be moved */
#define source_addr_a1           a1   /* from address */
#define destination_addr_a2      a2   /* to address */
#define END_ADDR           a3   /* ending address */
#define TEMP_v1                  v1
```

4.13.1. Copy to Instruction Scratch Pad

Check to see if there is an Instruction Scratchpad RAM. To do this the code reads the CP0 Config register. If there is an Instruction Scratchpad RAM the ISP bit in the Config register will be set so the code extracts the ISP bit, bit 24 and then checks to see if it's 0. If it is it assumes that there is no Scratchpad RAM at all and branches to the end of the function. If it is set then the code will fall through to the next instruction.

```
    mfc0 TEMP_v1,    C0_CONFIG
    ext  TEMP_v1, TEMP_v1, 24, 1
    blez TEMP_v1, set_dspram //no ISPram
```

The next few lines of code set the starting address of the ISPRAM in the ISPRAM controller. To clarify further, while the physical address of the ISPRAM can be set at core build time, it can also be changed by software to place it anywhere in physical memory. The code here is changing the physical address of the ISPRAM to match the address where the main.c code was linked. The code assumes that the system is not using a TLB but instead uses Fixed Mapping Translation (FMT). With FMT, KUSEG starts at virtual address 0 and maps to Physical address 0x4000 0000. In this example, the main.c code is linked to virtual address 0x1000 0000, so the ISPRAM is placed at physical address 0x5000 0000 (_ISPram = 0x5000 0000).

The “cache” instruction is used to program the Scratchpad memory physical address and fill the instruction Scratchpad. The “cache” instruction does this by writing the tag registers to the Scratchpad controller. There are two tag registers for each Scratchpad RAM, one set for the ISPRAM and one set for the DSPRAM. Tag 0 is located at offset 0 and tag 1 is located at byte offset 8 into the Scratchpad controller. Here is a table that shows what bits and tags contain information.

I or D Tags							
tag	31	20	19	12	….	7	6 0
0	Physical Base Address					E	
1			Size		0		

As shown in the table, the physical address is located in tag 0 bits 12 through 31 (4K boundary), and the Enable bit is located in tag 0 at bit 7. Both of these bits are read/write. The size in 4K sections is located in tag 1 bits 12 through 19.

The following code will place the physical address of the ISPRAM into the CP0 C0_TAGLO register.

The code puts the _ISPram value into a1 then moves it to the C0_TAGLO register.

```
LA    source_addr_a1, _ISPram
mtc0  source_addr_a1, C0_TAGLO
```

The “cache” instruction will then be used to program the instruction Scratchpad controller with the value stored in the C0_TAGLO register. By default, the cache instruction directs all of it operations to the cache controller. The code needs to change that, so that the cache operations are directed to the Scratchpad controller. It does this by setting the SPR bit (28) in the CP0 Error Control register (26, 0).

To do this, the code reads the C0_ERRCTL register, makes a copy so it can later restore it to its current state, sets the SPR bit, and writes the value back to the C0_ERRCTL register.

```
mfc0 s0_save_C0_ERRCTL, C0_ERRCTL
move    TEMP_v1, s0_save_C0_ERRCTL // copy so we can restore C0_ERRCTL
ins     TEMP_v1, all_ones_v0, 28, 1// set SR bit (28)
mtc0 TEMP_v1,    C0_ERRCTL
```

Now the code can use the cache instruction to write the Instruction Scratchpad tag.

Here is the instruction format of the cache instruction:

```
cache op, offset(base)
```

The "*op"* is encoded with 2 pieces of information; bits zero and one tell the cache instruction which Scratchpad block the operation will be performed on:

- 00 sets it for the Instruction Scratchpad
- 01 sets it for the Data Scratchpad

Bits two, three and four of the "*op"* tell the instruction which operation to perform

- 001 will load a tag
- 010 will store a tag
- 011 will store data into the Scratchpad blocks memory

The offset and base register control which of the 2 possible tags the operation will be performed on or which address within the Scratchpad block data will be stored to.

The code will use 8 *(*010 00) as the *op,* bits 0, 1 = 00 = Instruction Scratchpad bits 2, 3 and 4 = 010 = store a tag. Since tag 0 is being written the offset is 0 and the Base address is 0 so it uses GPR 0 (which is always 0)

```
cache 0x8,0(zero)
```

Next the code sets up the register to hold the virtual ROM address to copy from. First it loads source_addr_a1 with the address to copy from. _zap1 is declared in the linker and set by the linker at link time. This address is a cached address. Since we may not have a cache the code converts the address to an uncached address by setting bit 29.

```
LA      source_addr_a1, _zap1               // starting ROM address
ins     source_addr_a1, all_ones_v0, 29, 1 // convert to uncached address
```

The code sets up destination_addr_a2 register to hold the virtual memory address to copy to.

```
 LA      destination_addr_a2, _ftext_ram  // starting ram address
```

Then the code sets up the END_ADDR register to mark the ending address of the copy.

```
 LA      END_ADDR, _etext_ram             // ending ram address
```

Now it compares the starting address with the ending address and will jump ahead if there is nothing to copy.

```
beq     destination_addr_a2, END_ADDR, zero_bss // if = nothing to do
```

The Instruction Scratchpad memory cannot use the simple approach of using stores to write to it because it is not attached to the load store unit of the core, just the fetch unit. The "cache" instruction must be used to fill the Instruction Scratchpad memory array. Therefore it doesn't actually use the destination addresses. Instead the instruction Scratchpad is treated as an array of words (4 bytes each). The code will need a register for the "cache" instruction to store the base array element within the Instruction Scratchpad array where the code will be loaded into. The way the linker script has laid out the code and the code has used values set in the linker script the first instruction should be loaded into location 0 for the Instruction Scratchpad memory array.

The code just uses GPR zero to load the initial value into s1 which will be used as the first index to be written to.

```
   add   TEMP_v1, zero, zero
```

Instructions will be loaded into the Instruction Scratchpad array by the “cache” instruction, 1 at a time. One instruction is loaded into CP0 register C0_DATALO before the cache instruction is executed.

Recall that a1_temp_addr holds the current copy from address, a2 holds the current copy to address and a3_temp_mark holds the ending address (in RAM).

The code loads the data from source_addr_a1into data_a0 and then moves data_a0’s value to the C0_DATALO register.

```
next_Iram_word:
    lw      data_a0, 0(source_addr_a1)
    mtc0 data_a0, C0_DATALO
```

The “op” will use C *(*011 00) as the *op,* bits 0, 1 = 00 = Instruction Scratchpad bits 2, 3 and 4 = 011 = will load data into the Scratchpad blocks memory. The base address in the array is stored in TEMP_v1 and the offset from the base address is 0.

```
   cache 0xc,0(TEMP_v1)
```

The Base and the destination addresses are then incremented by 1 instruction (4 bytes).

```
    addiuTEMP_v1, 4
    addiu   destination_addr_a2, 4
```

The “from” address is incremented

```
    addiu   source_addr_a1, 4
```

The current destination address is compared to the ending address and branches to the top of the copy loop if they are not equal.

```
    Bne  destination_addr_a2, END_ADDR, next_Iram_word
```

4.13.2. Copy to Data Scratch Pad

The next step is to copy the initialized data. Copying to the Data Scratchpad is similar to the Instruction Scratchpad Copy, but only uses the DDATALO register to load the DSPRAM. Since there is only one word written at a time, there is no need for a Big/Little version of the code.

First the code reads the CP0 Config register (CPO Register 16,0) and extracts the DSP bit. If it is set, the code continues setting up the Data Scratch Pad.

```
set_dspram:
    mfc0 TEMP_v1,C0_CONFIG
    ext  TEMP_v1, TEMP_v1, 23, 1
    blez TEMP_v1, copy_c2_ram_done //no DSPram just exit
```

The code sets the physical address of the Data Scratchpad by moving the DSPRAM value (defined in the linker script) into source_addr_a1and then setting the enable bit (7). Then it moves the source_addr_a1 to the C0_DTAGLO register.

```
   LA    source_addr_a1, _DSPram
   // set the enable bit
   ins   source_addr_a1, all_ones_v0, 7, 1
```

```
    // move it to the  tag register
    mtc0 source_addr_a1, C0_DTAGLO
```

The “op” for the cache instruction will use 9 (010 01) , bits 0, 1 = 01 = Data Scratchpad bits 2, 3 and 4 = 010 = store a tag. Since tag 0 is being written the offset is 0 and the Base address is 0 so it uses zero (which is always 0)

```
   // write data tag 0 using the cache instruction
   cache 0x9,0(zero)
```

Next the code sets up the register to hold the virtual ROM address to copy from First it loads source_addr_a1 with the address to copy from using the value _zap1, which is declared in the linker and set by the linker at link time. This address is a cached address, and because we might not have a cache, the code converts the address to an uncached address by setting bit 29.

```
    LA      source_addr_a1, _zap2              // starting ROM address
    ins     source_addr_a1, all_ones_v0, 29, 1 // convert to uncached address
```

The code sets up the destination_addr_a2 register to hold the virtual memory address to copy to.

```
    LA      destination_addr_a2, _fdata_ram  // starting ram address to copy
```

Then the code set up the END_ADDR register to mark the ending address of the copy.

```
    LA      END_ADDR, _edata_ram        // ending ram addressss
```

Now it compares the starting address with the ending address and will jump ahead if there is nothing to copy.

```
    beq     destination_addr_a2, END_ADDR, zero_bss // if = nothing to do
```

The copy is simply reading from the location where the “C” code is stored in flash (source_addr_a1), moving that value to the DDataLo register and issuing the cache instruction to write the DSPRAM at the index stored in $11. Then the index is incremented to the next word to be written in the DSPRAM.

```
next_Dram_word:
    lw      data_a0, 0(source_addr_a1)
    mtc0    data_a0,C0_DDATALO
    cache   0xd,0(TEMP_v1)
    addiu   TEMP_v1, 4
```

The source and destination addresses are incremented by 4 (the number of bytes in a word). The code checks to see if it still has more to copy by checking END_ADDR, which is the end address, and the current destination address to see if they are equal.

```
   addiu   destination_addr_a2, 4
   addiu   source_addr_a1, 4
   bne     destination_addr_a2, END_ADDR, next_Dram_word
```

Now the code turns its attention to the uninitialized variable section (also known as the bss section). The C specification requires that the bss section be initialized to 0 before a program starts. This clearing of the bss section is usually done by the program loader. It is the responsibility of the boot loader to clear the first bss section before calling the main “C” function.

This code is similar to the code we just used for the copy. It uses two values created in the linker script: _fbss is the first address of the bss section, and _end is the end address of the bss section. Then it checks to see if there is anything to copy by determining if they are equal.

```
zero_bss:
    LA      source_addr_a1, _fbss      // start address
    LA      END_ADDR, _end       // end address
    beq     source_addr_a1, END_ADDR, copy_c2_ram_done
```

The code moves a 0 to the DDataLo register to initialize the DSPRAM to 0.

```
   // assume bss follows the initialized data
   // write a 0 to the DDataLo register
   mtc0  zero, C0_DDATALO // set to 0
```

The label next_bss_word will be used as a loop point. The code stores a zero using the zero register to the destination address in a1_temp_addr. It then adds 4 bytes to the destination address.

Then it checks to see if it is at the end of the copy by comparing it to the end address stored in END_ADDR and loops back if it is not.

```
next_bss_word:
    cache   0xd,0(TEMP_v1)                    // write DDATA LO to DSPRam
    addiu   source_addr_a1, 4
    addiu   TEMP_v1, 4                        // add 4 to DSPRam index
    bne     source_addr_a1, END_ADDR, next_bss_word
```

The copy is now done, but there is still some cleanup to do. The code needs to enable the Instruction Scratchpad RAM so that instructions can be fetched from it. The code loads the address _ISPram into source_addr_a1, sets the enable bit, bit 7, and moves that value to the C0_TAGLO register. Then it executes an “ehb” instruction to ensure any hazard barrier is cleared before it issues the cache instruction.

```
copy_c2_ram_done:
   // Enable ISPRAM
   LA    source_addr_a1, _ISPram
   // set the enable bit
   ins   source_addr_a1, all_ones_v0, 7, 1
   // move it to the tag register
   mtc0  source_addr_a1,C0_TAGLO
   ehb
```

The code then executes the cache instruction with the same op it used to write the Instruction Scratchpad address.

```
   // write instruction tag lo using the cache instruction
   cache 0x8,0(zero)
```

Finally, to ensure that cache instructions will once again be directed to the caches instead of the Scratchpads, the code restores the C0_ERRCTL using the saved value in s0_save_C0_ERRCTL, and then returns to the start code.

```
    // restore C0_ERRCTL
    mtc0    s0_save_C0_ERRCTL,C0_ERRCTL
    jr      ra
END(copy_c2_ram)
```

4.14. common/init_caches.S

Before use, the cache must be initialized to a known state; that is, all cache entries must be invalidated. This code example initializes the cache, determines the total number of cache sets, then loops through the cache sets using the cache instruction to invalidate each set.

The CP0 Config1 register has fields containing information about the cache, as shown in the figure below.

Config1 Register					
24 22	21 19	18 16	15 13	12 10	9 7
IS	IL	IA	DS	DL	D A

- IS : I-cache sets per way (cache lines) 0 = 64, 1 = 128, 2 = 256, 3 = 512, 4 = 1024, 5 = 2048, 6 = 4096
- IL: I-cache line size 0 = No I-Cache present; 4 = 32 bytes
- IA: always 4-way
- DS : D-cache sets per way (cache lines) 0 = 64, 1 = 128, 2 = 256, 3 = 512, 4 = 1024, 5 = 2048, 6 = 4096
- DL: D-cache line size 0 = No I-Cache present; 4 = 32 bytes
- DA: always 4 way

- To make the code easier to follow there are #defines at the top of init_caches.S to associate general purpose registers with a more function label:

```
#define LINE_SIZE               $3
#define BYTES_PER_LOOP          $2
#define SET_SIZE                $4
#define ASSOC                   $5
#define CONFIG_1                $6
#define END_ADDR                $7
#define TOTAL_BYTES             $12
#define CURRENT_ADDR            $13
#define TEMP1                   $14
#define TEMP2                   $15
```

- There is also a #define for the number of cache lines in the main initialization which will be loaded into a register for the total bytes per loop calculation. NOTE: This should not be changed!

```
#define LINES_PER_ITER 8 // number of cache instructions per loop
```

4.14.1. init_icache

The init_icache function must first compute the number of sets or cache lines it has to invalidate. The total number of lines in the cache is equal to the number of ways times the number of sets per way.

The code starts by checking the hardware cache initialization bit (HCI). If this bit is set it means that the hardware has built in initialization so the software does not need to do it and the code will branch to the end of the cache initialization. (It is common for simulators to have this feature because software initialization of caches would be very slow.)

```
LEAF(init_icache)
   // Can be skipped if Config7[HCI] set
   mfc0  TEMP1, C0_CONFIG, 7           // Read CP0 Config7
   ext   TEMP1, TEMP1, HCI, 1          // extract HCI
   bne   TEMP1, zero, done_icache
   nop
```

Next get the contents of the CP0 Config1 register to obtain the cache information.

```
   // determine how big the I$ is
   mfc0  CONFIG_1, C0_CONFIG1          // read C0_Config1
```

Next it determines the line size of the I-cache, using the extract instruction is to extract the line size. It uses the Config1 register value that was saved in CONFIG_1, starting at bit 19, and extracts 3 bits to the least-significant bits of LINE_SIZE.

```
   // Isolate I$ Line Size
   ext   LINE_SIZE, CONFIG_1, CFG1_ILSHIFT, 3
```

The extracted value is tested to see if it is 0; if so, there is no Instruction cache, so it branches ahead without initializing the cache.

```
   // Skip ahead if No I$
   beq   LINE_SIZE, zero, done_icache
   nop
```

Now the code decodes the line size to get the actual number of bytes in a line. It does this by shifting 2 to the left by the encoded line-size value.

```
   li    TEMP1, 2
   sllv  LINE_SIZE, TEMP1, LINE_SIZE   // line size in bytes
```

Now the code extracts the number of sets per way from the value read from the Config1 register that was stored in CONFIG_1, using the extract instruction.

```
   ext   SET_SIZE, CONFIG_1, CFG1_ISSHIFT, 3 // extract IS
```

The extracted value is converted to the actual number of sets per way by shifting 64 left by the extracted value.

```
li    TEMP1, 64
sllv  SET_SIZE, TEMP1, SET_SIZE     // I$ Sets per way
```

The number of ways is extracted using the extract instruction starting at bit 16 and extracting 3 bits to the least-significant bits of ASSOC. The code then adds one to the value to get the actual number of ways.

```
// Config1IA == I$ Assoc - 1
ext   ASSOC, CONFIG_1, CFG1_IASHIFT, 3    // extract IA
addiu ASSOC, ASSOC, 1
```

The code computes the Total number of bytes in the cache (`TOTAL_BYTES`) and the total number of bytes initialized with each iteration of the invalidation loop(BYTES_PER_LOOP) . NOTE: release 6 of the architecture uses a slightly different instruction so there is a compile time choice made.

```
mul   SET_SIZE, SET_SIZE, ASSOC
mul   TOTAL_BYTES, SET_SIZE, LINE_SIZE
mul   BYTES_PER_LOOP, LINE_SIZE, TEMP1
```

CURRENT_ADDR will be used as an index into the cache. It will be set to a virtual address, and then translated to a physical address. Since the address 0x8000 0000 is in KSEG0, the CPU will ignore the top bit, so virtual 0x8000 0000 will become physical address 0x0000 0000. This has the effect of the way and index the first time through the loop so that the cache instruction will write the tag to way 0, index line 0.

The starting CURRENT_ADDR also need to be adjusted to a starting address that is in the middle of the `BYTES_PER_LOOP` because the code needs to use + and – register offsets.

```
li    CURRENT_ADDR, 0x0000000080000000
srl   TEMP1, BYTES_PER_LOOP, 1
addu  CURRENT_ADDR, TEMP1, CURRENT_ADDR
```

Now compute the loop ending address:

```
addu  END_ADDR, CURRENT_ADDR, TOTAL_BYTES
subu  END_ADDR, END_ADDR, BYTES_PER_LOOP  // -1
```

Clearing the tag registers does two important things: it sets the Physical Tag address called PTagLo to 0, which ensures the upper physical address bits are zeroed out, and it also clears the valid bit for the set, which ensures that the set is free and may be filled as needed.

The code uses the Move to Coprocessor zero instruction to move the general purpose register zero, which always contains a zero, to the tag registers.

```
// Clear TagLo/TagHi registers
mtc0    zero, C0_TAGLO // write C0_ITagLo
mtc0    zero, C0_TAGHI // write C0_ITagHi
```

The cache instruction will be using the Index Store tag operation on the Level 1 instruction cache so the op field is coded with 8. The first two bits are 00 for the level one instruction cache, and the operation code for Index Store tag is encoded as 010 in bits two, three and four.

The index type of operation can be used to address a byte in the cache in a specific way of the cache. This is done by dividing the virtual address argument stored in the base register of the cache instruction into several fields.

Unused	Way	Index	Byte Index
	13 12	11 5	4 0

The size of the index field will vary according to the size of a cache way. The larger the way, the larger the index needs to be. In the table above is an example of how the indexes are broken down assuming the way size is 4K. Because each way of the cache is 4K, the combined byte and page index is 12 bits, The way number is always the next two bits following the index.

Therefore the code below does not explicitly set the way bits. Instead it just increments the virtual address by the BYTES_PER_LOOP, so that the next time through the loop, the cache instructions will initialize the next sets in the cache.

Eventually this increment has the effect of setting the cache to index 0 of the next way in the cache, because it overflows into the way bits.

The loop does 8 cache lines at a time to cut down on loop overhead and make the boot faster. Here is the loop:

```
next_icache_tag:
      // Index Store Tag Cache Op
      // Will invalidate the tag entry, clear the lock bit, and clear the LRF bit
    cache 0x8, (ILINE_SIZE*-2)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*-1)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*0)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*1)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*-4)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*-3)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*2)(CURRENT_ADDR)
    cache 0x8, (ILINE_SIZE*3)(CURRENT_ADDR)

    addu  CURRENT_ADDR, BYTES_PER_LOOP // Get next starting line address
    bge   END_ADDR, CURRENT_ADDR,  next_icache_tag    // Done yet?

done_icache:
    jalr  zero,      ra
END(init_icache)
```

The function is complete and returns to start.S

4.14.2. init_dcache

The init_dcache code is very similar to the init_icache code. The main difference is the cache instruction. The cache instruction will be using the Index Store tag operation on the Level 1 data cache, so the op field is coded with a 9. The first two bits are 01 for the Level 1 data cache, and the operation code for Index Store tag is encoded as 010 in bits two, three, and four.

The rest of the code is the same as the init_icache and will not be described again.

4.14.3. change_k0_cca

This function will change the Cache Coherency Attribute (CCA) of KSEG0 when in Kernel mode. It will turn caching on. (NOTE: This will have no effect on a Core that boots directly into EVA mode because the K0 has been disabled and there is no overlap of memory segments).

For coherent processor cores the CCA will be set to 5, which means that the KSEG0 address space will be set to cached write back with write allocate, coherent, and when a read misses in the cache, the line will become shared.

For non-coherent cores (interAptivUP), the CCA will be set to 3, which means that the KSEG0 address space will be set to cached write back with write allocate.

The code will read the CP0 Config register. Then it will test to see if it is executing on a coherent core. The non-coherent CCA of 3 is set, so that if it isn't a coherent core, the code will branch around the next instruction at sets the coherent CCA of 5 so if it is not a coherent core the code will fall through and set the CCA to 3. Next it will insert the CCA into the register that holds the Config Register value that was just read and then write it back to the CP0 Config Register.

The non-coherent CCA of 3 is set, so that if it isn't a coherent core, the code will branch around the next instruction that sets the coherent CCA of 5, so if it is not a coherent core, the code will fall through and set the CCA to 3. Next it will insert the CCA into the register that holds the Config Register value that was just read and then write it back to the CP0 Config Register.

```
LEAF(change_k0_cca)
   // NOTE! This code must be executed in KSEG1 (not KSGE0 uncached)
   // Set CCA for kseg0 to cacheable
   mfc0  TEMP1, C0_CONFIG  // read C0_Config
   li    TEMP2, 3    // CCA for all others
   beqz  r11_is_cps, set_kseg0_cca
   li    TEMP2, 5    // CCA for coherent cores (fall through)

set_kseg0_cca:
   ins   TEMP1, TEMP2, 0, 3      // insert K0
   mtc0  TEMP1, C0_CONFIG  // write C0_Config
   jalr.hb   zero, ra

END(change_k0_cca)
```

4.14.4. disable/enable L2 CM revision 2 - init_L2_CM2.S (interAptiv or proAptiv)

First a little background on the larger picture:

- L2 cache is a system resource that is used by all cores in a CPS. Initialization of the L2 cache is done only by Core 0 in a CPS, because it only needs to be done once.

- The initialization of the L2 cache can be very time consuming because of their size. The initialization code will execute a lot faster if it is being run out of the instruction cache, so it should be done after the instruction cache has been initialized.

- The instruction cache is a Core resource and not initialized in the System initialization section of the code. Therefore, to be efficient and run the L2 cache initialization out of the I-cache, the boot code tries to delay cache initialization until Core resources are initialized and it has checked to make sure it is only initialized by Core 0. This can only be done if the L2 cache can be disabled before other cores are released to run this boot code. Otherwise there is a danger that other cores will use the L2 cache before Core 0 has initialized them.

- The CCA override feature controls the cache attributes for the L2 cache. It allows disabling the L2 cache by enabling the CCA override and setting the CCA to uncached.

- The CCA override works along with the L2 cache implementation. If your CPS uses the current MIPS implementation for the L2 cache, the CCA override feature is supported. However, your system can implement its own version of the L2 that does not support this feature. If that is the case, the L2 cache must be initialized as a system resource before other cores are released to run.

- MIPS does not have a reference implementation of an L3 cache. If your CPS has an L3 cache and it can be disabled, you will need to add code here to disable it. If it can't be disabled, it will need to be initialized at this point in the code.

The disable_L2 function enables the CCA override and sets the L2 cache to uncached in the GCR_BASE register, thus disabling it. On older system prior to the interAptiv and proAptiv cores that do not support CCA override, writes to the CCA override field have no effect, and reading back the GCR_BASE register will not show the CCA override being set.

```
LEAF(disable_L2)
```

First the code checks to see if it is executing on a core 0, and if it isn't, it will branch to the end of the function.

```
   bnez    r8_core_num, done_disable_L2      # Only done from core 0.
    // Use CCA Override disable the L2 cache
    // NOTE: If you have a L3 cache you must add code here
    // to disable it or initialize it if it can't be disabled.
    // Disable the L2 cache using CCA override by writing a 0x50 to
    // the GCR Base register. 0x50 enables the CCA override bit and sets
    // the CCA to uncached.
    lw   GCR_BASE, 0x0008(r22_gcr_addr)// Read GCR_BASE
    li   TEMP, 0x50                    // Enable CCA and set to uncached
    ins  GCR_BASE, TEMP, 0, 8          // Insert bits
    sw   GCR_BASE, 0x0008(r22_gcr_addr)      // Write GCR_BASE
done_disable_L2:
    jr      ra
END(disable_L2)

LEAF(enable_L2)
    bnez    r8_core_num, done_enable_L2# Only done from core 0.
    // Use CCA Override disable the L2 cache
    // NOTE: If you have a L3 cache you must add code here
    // to enable it or initialize it if it can't be enabled.
```

```
    lw   GCR_BASE, 0x0008(r22_gcr_addr)      // Read GCR_BASE
    ins  GCR_BASE, zero, 0, 8                // CCA Override disabled
    sw   GCR_BASE, 0x0008(r22_gcr_addr)      // Write GCR_BASE
done_enable_L2:
    jr      ra
END(enable_L2)
```

4.15. init_L2_CM2.S - init_L2 for CM revision 2 (interAptiv or P5600)

The init_L2 code is very similar to the init_icache code. The main difference is the cache instruction. The cache instruction will be using the Index Store tag operation on the L2 cache, so the op field is coded with a B. Also information for the L2 cache is in the CP0 CONFIG2 register. The rest of the code is the same as the init_icache and will not be described again.

4.16. init_L2_CM3_64.S - init_L2 for CM revision 3 (I6400/I6500)

For the core using revision 3 of the C M the L2 cache is part of the CM and is disabled on cold reset so it does not need to be disabled as does the L2 cache for revision 2 of the CM. The init_L2 code is very similar to the init_icache code. The main difference is the cache instruction. The cache instruction will be using the Index Store tag operation on the L2 cache, so the op field is coded with a B. Also the configuration information for the L2 is also in a different place for revision 3. The information is located within the CM registers. The rest of the code is the same as the init_icache and will not be described again.

4.17. common/init_cp0.S - init_cp0 for 32 bit cores

The init_cp0 code will initialize the Status Register, Watch registers, clear watch exceptions, clear timer exceptions, and set the Cache Coherence Attributes for KSEG0,

4.17.1. Initialize the CP0 Status register

At this point in the boot, the status register should be set as follows:

- ERL set - the processor is running in kernel mode, Interrupts are disabled, the ERET instruction will use the return address held in *ErrorEPC* instead of *EPC,* and the lower 2 bytes of KUSEG are treated as an unmapped and uncached region.

- BEV set – bootstrap exception mode.

```
LEAF(init_cp0)
    // Initialize Status
    li   a1, 0x00400404 // (IM|ERL|BEV)
    mtc0 a1, $12         // write C0_Status
```

4.17.2. Initialize the Watch Registers

Next the code will initialize the Watch registers. The Watch registers are undefined following reset and need to be initialized to avoid getting spurious exceptions after the ERL bit is cleared. The code reads the CP0 Config1 register and extracts the WR field, bit 3. If this bit is not set, there are no Watch registers, so the code checks to see if it is equal to 0, and if it is, the code branches forward around the Watch register initialization.

```
   // Initialize Watch registers if implemented.
   mfc0  a0, C0_CONFIG1          // read C0_Config1
   ext   a1, v0, 3, 1// bit 3 (WR) Watch registers implemented
   beq   a1, $0, done_wr
```

The code sets up the initialization value for the Watch Registers. This value effectively clears all watch conditions.

```
   li    a1, 0x7                 // Clear I, R and W conditions
```

There are up to 8 Watch Registers. The next 8 segments are all very similar and will initialize up to 8 Watch Registers. Each one begins by writing the initialization value in a1 to the Watch Hi register.

```
   // Clear Watch Status bits and disable watch exceptions
   mtc0  a1, C0_WATCHHI          // write C0_WatchHi0
```

Each Watch Hi register contains an M bit (bit 31) to indicate that there are “more” Watch registers if it is set. Bit 31 is the sign bit for each word, and if set would indicate that the word is a negative value (less than 0). The code reads the Watch Hi register and checks to see if it is greater than or equal to zero (no M bit set) and if it is, the code branches ahead around the remaining Watch register initializations.

```
   mfc0  v0, C0_WATCHHI          // read C0_WatchHi0
   bne zero, v0, done_wr
```

The code clears the Watch Lo register to clear the Watch for address.

```
   mtc0   zero, C0_WATCHLO         // write C0_WatchLo0
```

The remaining 7 segments are very similar to the one just described, so they will be skipped.

After the Watch registers are initialized, the code clears the CP0 Cause register. This register indicates the cause of the most recent exception and comes up in an undefined state. In the case of the Watch Register, if the WP bit in the Cause register is set, then after the ERL bit is cleared, it would cause a spurious Watch exception.

```
done_wr:
   // Clear WP bit to avoid watch exception upon user code entry,
   // IV, and software interrupts.
   mtc0   zero, C0_CAUSE  // write C0_Cause: Init AFTER init of WatchHi/Lo
```

4.17.3. Clear the Compare Register

The code clears the CP0 Compare register so that the core will not get a Timer interrupt after the ERL bit is cleared.

```
   // Clear timer interrupt.
   mtc0   zero, C0_COMPARE         // write C0_Compare
   return;
   jr      ra
END(init_cp0)
```

4.18. common/init_cp0_64.S - init_cp0 for 64 bit release 6 cores (I6400/I6500)

The code for a 64 bit release 6 core differs only in a small change to the instructions used. Otherwise the logic is the same as for the 32 bit version and will not be described again.

4.19. common/init_gpr.S - init_gpr for 32 bit cores

init_gpr.S will initialize all of the GPR register set in the processor/VP/VPE.. The initializing of the GPR registers is not strictly necessary, but may help debug improperly written code where a value is read without being written.

The code starts out by setting the default value that it will write to each register.

```
LEAF(init_gpr)
   li    $1, 0xdeadbeef    // (0xdeadbeef stands out)
```

The move instruction is used to initialize the registers.

```
   move  $1, $1
   move  $2, $1
   move  $3, $1
```

```
   …..

   move  $30, $1
```

Note: When the code reaches 31, it doesn't want to wipe out the return value held in ra

The GPR initialization is complete and the code returns to start.

```
done_init_gpr:
   jr      ra
END(init_gpr)
```

4.20. common/init_gpr_64.S - init_gpr for 64 bit cores

init_gpr_64.S is the same logic as init_gpr for a 32 bit core. There are small changes for release 6 of the architecture.

4.21. common/init_tlb.S (non P5600 and I6400/I6500 cores only)

init_tlb.S will initialize the Translation Look aside Buffer (TLB) if present. The TLB needs to be initialized so that there are no random translations in it.

The code first checks to see if the core has a TLB. It reads the CP0 Config Register (16) and checks the MT (MMU Type) field (3 bits starting at bit 7) by extracting it to a0, then sees if it is set to 1. If it's not, the core doesn't have a TLB, so the code will go to the end of the function.

```
LEAF(init_tlb)
check_for_tlb:
   // Determine if we have a TLB
   mfc0    a0, C0_CONFIG   // read C0_Config
   ext     a0, a0, 7, 3    // check MT field
   li      a2, 0x1         // load a 1 to check against
   bne     a0, a2, done_init_tlb
```

The code reads the MMU Size field in the CP0 Config1 register for later use.

```
   mfc0    a1, C0_CONFIG1  // read C0_Config1
```

Now the code will use the CP0 Config1 value stored earlier in v0 and extract the MMU size field (6 bits starting at bit 25) into v1. This is used as the highest TLB entry and will be used as the first index into the TLB.

```
start_init_tlb:
   // Config1MMUSize == Number of TLB entries - 1
   ext     a0, a1, CFG1_MMUSSHIFT, 6   // extract MMU Size
```

Now to clear all entry registers, we initialize all fields in the TLB to 0. To do this we use the same move to Coprocessor zero instruction using general purpose register 0, which always contains the value 0, and move its contents to the corresponding Coprocessor 0 register.

```
   mtc0    zero, C0_ENTRYLO0           // write C0_EntryLo0
   mtc0    zero, C0_ENTRYLO1           // write C0_EntryLo1
   mtc0    zero, C0_PAGEMASK           // write C0_PageMask
```

```
    mtc0    zero, C0_WIRED              // write C0_Wired
```

Now we load a3 with the address to be placed in the entry. Note that it will be marked invalid but will ensure that the TLB does not have duplicate entries.

```
    li    a3, 0x80000000
```

We will now use a loop to initialize each TLB entry.

The next_tlb_entry_pair label in the left column is the label of the start of the loop and the point we will loop back to. To make sure the address that will be written to the TLB entry is unique, the VPE or core number is inserted into it.

```
next_tlb_entry_pair:
```

Previously the code stored the highest numbered TLB entry in general purpose register v1. Here it is used to program the TLB entry index. The code uses the move to Coprocessor 0 instruction to copy the contents of general purpose register a0 to Coprocessor 0 register, which is the index register. The index will indicate the TLB entry to be written. The address to be initialized is written to the CP0 EntryHi register.

```
    mtc0  a0, C0_INDEX                  // write C0_Index
    mtc0  a3, C0_ENTRYHI                // write C0_EntryHi
```

The code needs to make sure all the writes to CP0 been completed before writing the TLB entry. It does this by using the ehb instruction.

```
    ehb
```

Now the TLB Write Indexed Instruction is used to write the TLB entry.

```
    tlbwi
```

The address is incremented by 8K so there are no duplicates.

```
    add   a3, 0x2000              // Add 8K to the address
```

Use the add instruction to decrement the index value in a0 by adding a -1.

```
    add   a0, -1
```

The branch instruction is used to see if the code has written the last TLB entry (entry 0). Here the code compares the TLB index value that is in v1 with 0, and if they are not equal, the code branches back to the top of the loop.

```
    bne   a0, zero, next_tlb_entry_pair
```

When the loop is finished, the function returns to start.

```
done_init_tlb:
    jr      ra
  END(init_tlb)
```

4.22. common/init_tlb_FTLB.S - init_tlb (P5600 only)

init_tlb initializes the VTLB and FTLB Translation Look-aside Buffers. This code example is specific to processors with the VTLB/FTLB feature. In addition, it uses the fact that these cores

will always have a least a 64K VTLB, and if they have the optional FTLB, it will always be 512 entries. The TLB needs to be initialized so that there are no random translations in it.

```
LEAF(init_tlb)
compute_TLB_size:
```

The MT field in the CP0 Config register (16, 0) is used to decode what type of TLBs are in the Core. If CP0 Config MT = 1, there is only a VTLB and it may be greater than 64K. If CP0 Config MT = 4, there is also an FTLB and the VTLB is 64K.

The code reads the CP0 Config register and extracts the MT field.

```
   / Determine if we have an FTLB or just a VTLB and set size accordingly
   mfc0  v1, C0_CONFIG     // read C0_Config
   ext   v1, v1, 7, 3      // check MT field
```

A 1 is loaded into a3 to test against. The code checks to see if MT is equal to 1, indicating there is only a VTLB, and sets the VTLB size to 64. If it was not equal to one, the only other value it could be is 4, indicating it has an FTLB, so the code jumps ahead to set the total TLB size with FTLB.

```
   li    a3, 0x1           // load a 1 to check against
   bne   v1, a3, ftlb      // if not only VTLB branch
   li    v1, 64            // set VTLB entries to 64
```

If the branch falls through, it means that we don't have an FTLB, but the VTLB may have an extended size (beyond the default 64 entries). The MMUSizeExt field in CP0 config 4 (16, 4) contains 7 bits of extended entries. These bits are the upper 7 bit of the number of total entries in the VTLB. The code reads Config 4 and extracts the MMUSizeExt field. It then shifts this value 7 to the left to put the bits in the correct position. Then it adds the default value of 64 to get the total number of TLB entries. Once the TLB size is determined, the code loads it into v1 that will be used as a loop counter to integrate through and initialize TLB entries and branches to the start of the TLB initialization.

```
   mfc0  $15, C0_CONFIG, 4 // Read Config 4
   ext   a3, a3, 0, 7      // get MMUSizeExt
   sll   a3, a3, 7         // shift to upper bits
   add   v1, v1, a3        // add to the 64
   b     start_init_tlb
```

The code reaches the ftlb label if the MT field was not 1. This means that there is an FTLB. The FTLB is a set size of 512 entries, so the total number of entries is the addition of the 64 VTLB entries plus the 512 FTLB entries or 576. The code loads 576 into v1 that will be used as a loop counter to integrate through and initialize TLB entries.

```
ftlb:
   li    v1, 576           // set value of 64 + 512 if using FTLB
```

Now to clear all entry registers, we will initialize all fields in the TLB to 0. To do this we use the same move to Coprocessor zero instruction using general purpose register 0, which always contains the value 0, and move its contents to the corresponding Coprocessor 0 register.

```
start_init_tlb:
   mtc0    zero, C0_ENTRYLO0     // write C0_EntryLo0
```

```
   mtc0    zero, C0_ENTRYLO1     // write C0_EntryLo1
   mtc0    zero, C0_PAGEMASK     // write C0_PageMask
   mtc0    zero, C0_WIRED        // write C0_Wired
```

Now we load $12 with the address to be placed in the entry. Note that it will be marked invalid but will ensure that the TLB does not have duplicate entries.

```
   li    a0, 0x80000000
```

We will now use a loop to initialize each TLB entry.

The next_tlb_entry_pair label in the left column is the label of the start of the loop and the point we will loop back to. To make sure the address that will be written to the TLB entry is unique, the VPE or core number is inserted into it.

```
next_tlb_entry_pair:
```

Use the add instruction to decrement the index value in the general purpose register 11 by adding a -1.

```
   add   v1, -1
   ins     a0, r23_cpu_num, 20, 4  // add "Core" number
```

Previously the code stored the highest numbered TLB entry in general purpose register 11. Here it is used to program the TLB entry index. The code uses the move to Coprocessor 0 instruction to copy the contents of general purpose register 11 to Coprocessor 0 register, which is the index register. The index will indicate the TLB entry to be written. The address to be initialized is written to the CP0 EntryHi register.

```
   mtc0  v1, C0_INDEX            // write C0_Index
   mtc0  $12, C0_ENTRYHI         // write C0_EntryHi
```

The code needs to make sure all the writes to CP0 been completed before writing the TLB entry. It does this by using the ehb instruction.

```
   ehb
```

Now the TLB Write Indexed Instruction is used to write the TLB entry.

```
   tlbwi
```

The address is incremented by 16K so there are no duplicates.

```
   add   a0, (2<<13)
```

The branch instruction is used to see if the code has written the last TLB entry (entry 0). Here the code compares the TLB index value that is in general purpose register 11 with 0, and if they are not equal, the code branches back to the top of the loop.

```
   bne   v1, zero, next_tlb_entry_pair
```

When the loop is finished, the function returns to start.

```
done_init_tlb:
   jr      ra
END(init_tlb)
```

4.23. common/ init_ftlb_R6.S- init_tlb (P6600 and I6400/I6500 only)

These cores implement a feature that initializes the VTLB with one instruction:

```
   tlbinvf     // invalidate all VTLB entries
```

The init_ftlb_R6.S initializes only the FTLB and uses some R6 only instructions.

4.24. cps/init_cm.S - init_cm Coherence manager (interAptiv or P5600)

The code in init_cm.S initializes the Coherence Manager.

First the code checks to see if it is booting a coherent processing system, and if not, will branch to the end of this function.

```
LEAF(init_cm)
   beqz    r11_is_cps, done_cm_init     // skip if not a CPS
```

Register Fields		Global CSR Access Privilege Register (GCR_ACCESS Offset 0x0020)	Reset State
Name	Bits		
CM_ACCESS_EN	7-0	Each bit in this field represents a coherent requester. If the bit is set, that requester is able to write to the GCR registers (this includes all registers within the Global, Core-Local, Core-Other, and Global Debug control blocks). The GIC is always writable by all requestors. If the bit is clear, any write request from that requestor to the GCR registers (Global, Core-Local, Core-Other, or Global Debug control blocks) will be dropped.	0xff

The lower 8 bits of the Global CSR Access Privilege Register controls the write access to the GCR registers by a processor unit (VPE or single core). If a bit is set, the processor unit can change the GCR.

Load a 2 into a0.

```
   li    a0, 2       // mask for cores in this cps.
```

Then the code shifts the 2 to the left by the number of processor units previously stored in r19_more_cores and then subtracts 1. For example, if there were 4 processor units. then r19_more_cores would contain a 3. Then 2 shifted left by 3 is 16 or 10 hex.

```
   sll  a0, a0, r19_more_cores
```

Now the code subtracts 1. 16 – 1 is 15 or F hex, so now we have all four lower bits set.

```
   addiu   a0, -1    // Complete mask.
```

These are written to the Global CSR Access Privilege Register, which will now allow all 4 processor units to change the GCR.

```
   sw    a0, GCR_ACCESS(r22_gcr_addr)  // write GCR_ACCESS
```

The code then checks to see if there is an IOCU. It does this by loading the GCR configuration register into a0 and extracting the NUMIOCU field.

```
   // Check to see if this CPS implements an IOCU.
   lw    a0, GCR_CONFIG(r22_gcr_addr)  // read GCR_CONFIG
   ext   a0, a0, NUMIOCU, NUMIOCU_S    // extract NUMIOCU
```

It then jumps around the next section of code to the end of the init_cm function if there are no IOCUs in the system.

```
   beqz    a0, done_cm_init
```

If there is an IOCU, then the code will make sure that the CM regions are disabled. The code loads an upper immediate value into a0, which sets bits 16 through 31 and clears bits 0 through 15. The lowest bit, bit 0, set to 0 will disable the CM region. The code uses a0 to store the value to all CM regions and thus disables them.

```
   lui   a0, 0xffff
    // Disable the CM regions if there is an IOCU.
   sw  a0, GCR_REG0_BASE(r22_gcr_addr) // write GCR_REG0_BASE
   sw  a0, GCR_REG0_MASK(r22_gcr_addr) // write GCR_REG0_MASK
   sw  a0, GCR_REG1_BASE(r22_gcr_addr) // write GCR_REG1_BASE
   sw  a0, GCR_REG1_MASK(r22_gcr_addr) // write GCR_REG1_MASK
   sw  a0, GCR_REG2_BASE(r22_gcr_addr) // write GCR_REG2_BASE
   sw  a0, GCR_REG2_MASK(r22_gcr_addr) // write GCR_REG2_MASK
   sw  a0, GCR_REG3_BASE(r22_gcr_addr) // write GCR_REG3_BASE
   sw  a0, GCR_REG3_MASK(r22_gcr_addr) // write GCR_REG3_MASK
```

This completes the CM initialization and the code returns to start.

```
done_cm_init:
   jr    ra
END(init_cm)
```

4.25. init_CM3_64.S - init_cm for CM version 3 (I6400/I6500 only)

The init_cm for the CM3 just needs to configure the CM to be accesses by all cores. The access mask is build the same as the code above. The difference is the GCR Access register is 64 bit so it needs to be accessed using double word load and stores.

```
LEAF(init_cm)
   // skip if not a CPS or CM register verification failed.
   beqz    r11_is_cps, done_cm_init
   // Allow each core access to the CM registers
```

```
   dli   a0, 2       // Start building mask for cores in this cps.
         dsll  a0, a0, r19_more_cores
         daddiu a0, -1     // Complete mask.
   sd     a0, GCR_ACCESS(r22_gcr_addr) // GCR_ACCESS
done_cm_init:
   jalr  zero,      ra
END(init_cm)
```

4.26. cps/init_cpc.S - init_ cpc Cluster Power Controller (interAptiv or P5600)

The init_cpc function sets the location of the Cluster Power Controller in the GCR CPC Base Register and stores the address for further use.

First the code checks to see if this is a coherent processing system by checking r11_is_cps that was set in the beginning. If it's not, then it will not have a CPC and will skip to the end and return.

```
LEAF(init_cpc)
   beqz    r11_is_cps, done_init_cpc   // Skip if non-CPS.
```

If there is a CPS, the code checks for a Cluster Power Controller by checking the Cluster Power Controller Status Register. This register is located within the Global Configuration Registers at offset 0xf0. The code uses the previously stored address of the GCR base and the 0xf0 offset to load the value of the Cluster Power Controller Status Register into a0.There is only one field in the Cluster Power Controller Status Register, called CPC EX, and if that bit is set, then the CPC is connected into the CPS. So all the code needs to do is test it for 0. If it is 0, there's no CPC and it branches around this code and returns to the initialization function. We ensure r30_cpc_addr is clear to indicate we don't have a CPC.

```
   lw      a0, GCR_CPC_STATUS(r22_gcr_addr)  // GCR_CPC_STATUS
   andi    a0, 1
   move    r30_cpc_addr, zero
   beqz    a0, done_init_cpc                  // Skip if no CPC
```

If there is a CPC, the code will set the address of the CPC in the Cluster Power Controller Base Address Register. The address of the Cluster Power Controller Base Address Register is at offset 88 hex of the GCR.

The code uses the known value of the location of CPC within the system and writes that to the Cluster Power Controller Base Address Register. This is a physical address. Also, bit 0 is set, to enable the address region for the CPC.

```
   li      a0, CPC_P_BASE_ADDR               // Locate CPC
   sw      a0, GCR_CPC_BASE (r22_gcr_addr)   // GCR_CPC_BASE
```

Then the code stores this address for later use in r30_cpc_addr using the KSEG1 equivalent address, and is now done setting up the CPC.

```
   li      r30_cpc_addr, CPC_BASE_ADDR       // copy to register
```

This completes the CPS initialization and the code returns to start.

```
done_init_cpc:
```

```
    jr      ra
END(init_cpc)
```

4.27. cps/init_cpc_CM2_64.S - init_ cpc Cluster Power Controller (P6600)

The P6600 uses CM2.5 which has been adapted from CM2 to work with 64bit values so it needs to use 2 32bit registers instead of 1 32bit register is some cases. The code follows the same logic as the code above but does 2 stores to 2 connective 32bit registers to construct a 64bit value.

4.28. cps/init_gic.S - init_gic Global Interrupt Controller

init_gic.S initializes the Global Interrupt controller for this example boot code.

The GIC address space is accessed with uncached load and store commands. For each load or store command, the hardware supplies the physical address and the Processor/VPE Number of the requester. The processor/VPE Number is used as an index to reference the appropriate subset of the instantiated control registers. By using the processor/VPE Number information, the hardware writes or reads the correct subset of the control registers pertaining to the “local” Core. Software does not need to explicitly calculate the register index for the “local” Core; it is done entirely by hardware.

The GIC is divided into segments:

Segment	Base Offset	Addressing Method	Size
Shared Section	0x0000	Offset relative to *GCR_GIC_Base*	32K
VPE-Local Section	0x8000	Offset relative to *GCR GIC Base + using* VPE Number as Index	16K
VPE-Other Section	0xc000	Offset relative to GCR_GIC_Base + using VPE-Other Addressing Register as Index	16K
User-Mode Visible Section	0x10000	Offset relative to *GCR_GIC_Base*	64K

The Shared segment starts at the Base address of the GIC. This shared section is where the external interrupt sources are registered, masked, and assigned to a particular processing element and interrupt pin. This section is used by all processing elements.

Next is the VPE-local section which starts at the Base address plus 0x8000. This is the section in which interrupts local to a VPE are registered, masked, and assigned to a particular interrupt pin.

Using the VPE-other segment, the “local” CORE can access the registers of another Core by using the Core-Other address spaces. Software must write the VPE-Other Addressing Register before accessing these spaces. The value of this register is used by hardware to index the appropriate subset of the control registers for the other core(s).

An additional section called the User-Mode Visible section is used to give quick user-mode read access to specific GIC registers. The use of this section is meant to avoid the overhead of system calls to read GIC resources, such as counter registers.

First the code checks to see if it needs to initialize the global interrupt controller. It checks r11_is_cps and if it is not set, then this is not a Coherent Processing system, so the code will skip the GIC initialization.

```
LEAF(init_gic)
   beqz    r11_is_cps, done_gic            // Skip if non-CPS.
```

Even though this is a Coherent Processing System, there still may not be a GIC. To find out, the code reads the Global Control Blocks GIC Status register, offset 0xD0, extracts the GIC_EX bit, and then tests to see if it is set. If it is not set, there is no GIC, so the code will skip the GIC initialization.

```
   la      a1, GCR_GIC_STATUS + GCR_CONFIG_ADDR
   lw      a0, 0(a1)
   ext     a0, a0, GIC_EX, GIC_EX_S
   beqz    a0, done_gic                      // If no GIC then skip
```

There are two parts of the GIC that need to be initialized: a Shared Part that needs to be initialized by only one core, and a local part that needs to be initialized by each processing unit. This code will do the shared part only if this is core 0, so it checks r23_cpu_num for the processing unit number and skips the shared section if it is not 0.

```
   bnez    r23_cpu_num, init_vpe_gic           // Only core0 vpe0
```

4.28.1. Setting the GIC base address and Enable the GIC

GCR_GIC_BASE Offset 0x0080				
Register Fields		Description	Read/Write	Reset State
Name	Bits			
GIC_BaseAddress	31-17	The base address of the 128KB Global Interrupt Controller block	R/W	Undefined
GIC_EN	0	Setting to 1 enables GIC	R/W	0

As you can see from the table, the address is on a 128K boundary, so the lower 17 bits will always be 0. This leaves space for additional information in the register. The GIC_EN field controls the enabling of the GIC.

The code loads the address of the GIC Base Address Register into a1.

```
li      a1, GCR_CONFIG_ADDR + GCR_GIC_BASE
```

The code loads a0 with the address of GIC (Physical address). Then bit 0 is set, which enables the GIC. This value is stored to the GCR_GIC_BASE register.

```
li      a0, (GIC_P_BASE_ADDR | 1) // Physical address + enable
sw      a0, 0(a1)
```

Next the code will use the GIC Configuration Register to confirm how many external interrupt sources we have. To do that, the code will read the register and isolate the NUMINTERRUPTS field, bits 16 through 23. Interrupt sources are configured in the core in groups of 8. This field tells you how many groups of 8 minus 1 the core has.

The define GIC_BASE_ADDR is the address of the Shared section of the GIC which is loaded into `a1`. The Shared Configuration register is located at offset 0. The code loads the value of the register into `a0`.

```
// Verify gic is 5 "slices" of 8 interrupts giving 40 interrupts.
li      a1, GIC_BASE_ADDR           // load GIC KSEG0 Address
lw      a0, GIC_SH_CONFIG (a1)      // GIC_SH_CONFIG
```

Then the code extracts the number of interrupt groups.

```
ext a0, NUMINTERRUPTS, NUMINTERRUPTS_S //extract NUMINTERRUPTS
```

For this example, the code loads the expected value of NUMINTERRUPTS into a3. This example is expecting 40 interrupt sources (4 + 1 times 8). If the code doesn't get what it expects, it executes a debug breakpoint to stop at a point where you can use the debug probe to see what's going on.

```
li      a3, 4
```

```
   bqe     a0, a3, configure_slices
   sdbbp   // Failed assertion of 40 interrupts.
```

4.28.2. Disable interrupts

Next the code will disable interrupts for the interrupts used by the example.

Register Offset	Reset Mask Register numbers	Description
0x0300	0 - 31	Writing a 0x1 to any bit location masks off (disables) that interrupt. At IP configuration time, the appropriate number of these registers is instantiated to support the number of External Interrupt Sources. These are write-only bits.
0x0304	32 - 63	
0x0308	64 - 95	
0x030c	96 - 127	
0x0310	128 -159	
0x0314	160 - 191	
0x0318	192 - 223	
0x031c	224 - 255	

To disable interrupts, the code will use the Global interrupt Reset Mask Registers. Each interrupt source has a corresponding bit in a Reset Mask Register. Setting a bit to one resets and disables the interrupt in the GIC. The GIC can control up to 256 interrupt sources. Since all registers in the GIC are 32 bits wide, in order to have enough bits to cover all 256 sources, we will need 8 Reset Mask Registers. The first register will control interrupts 0 through 31, the second set will control 32 through 63, and so on. The system in our example has external interrupts connected to interrupt pins 24 through 39. These interrupt sources will use the first two Global interrupt Reset Mask Registers.

The code that follows configures the interrupts one section at a time. First it will configure interrupts 24 through 31 and then 32 through 39.

The code disables the first 32 interrupt sources by writing a 1 to bits 24 – 31 in the first Global interrupt Reset Mask Register. The offset of the Global interrupt Reset Mask Registers into the GIC Section is hex 300.

```
configure_slices:
   // Hardcoded to set up the last 16 of 40 external interrupts
   // (24..31) for IPI.
   li      a0, 0xff000000
   sw      a0, GIC_SH_RMASK31_0 (a1) // (disable 0..31)
```

4.28.3. Setting the Global Interrupt Polarity Registers

Similar to the Reset Mask Registers, there is a set of registers that configures the polarity of the interrupt.

Register Offset	Interrupt Polarity Register numbers	Description
0x0100	0 - 31	Polarity of the interrupt. For Level Type: 0x0 - Active Low 0x1 - Active High For Single Edge Type: 0x0 - Falling Edge used to set edge register 0x1 - Rising Edge used to set edge register **At IP configuration time**, the appropriate number of these registers is instantiated to support the number of External Interrupt Sources. These bits are read/write.
0x0104	32 - 63	
0x0108	64 - 95	
0x010c	96 - 127	
0x0110	128 -159	
0x0114	160 - 191	
0x0118	192 - 223	
0x011c	224 - 255	

The polarity determines how the interrupt is signalled to the core. Interrupts can be level or edge sensitive. If level sensitive, setting the interrupt's corresponding bit to 1 will configure it active high, and setting it to 0 will configure it active low. If the interrupt is edge sensitive, setting the corresponding bit to 1 will configure it to interrupt on the rising edge, and setting it to 0 will configure it to interrupt on the falling edge. The offset of the Global interrupt Polarity Registers in the GIC Section is hex 100.

Register Offset	Trigger Type Register numbers	Description
0x0180	0 - 31	Edge or Level triggered 0x0 - Level 0x1 - Edge **At IP configuration time**, the appropriate number of these registers is instantiated to support the number of External Interrupt Sources. These are read/write bits.
0x0184	32 - 63	
0x0188	64 - 95	
0x018c	96 - 127	
0x0190	128 -159	
0x0194	160 - 191	

0x0198	192 - 223	
0x019c	224 - 255	

The code uses a0 to write 1's to bits 24 through 31 of the first interrupt Polarity Register. This configures interrupt sources 24 through 31 to be rising-edge sensitive.

```
sw   a0, GIC_SH_POL31_0 (a1)  // (high/rise    24..31)
```

4.28.4. Configuring Interrupt Trigger Type

There is a set of registers that configures the Trigger type of the interrupt. Setting the corresponding bit causes the interrupt to be treated as Edge signalling; if the bit is cleared, the interrupt is level signalling. The offset of the Global Interrupt Trigger Type Registers in the GIC Section is 0x180.

The code uses a0 to write 1's to bits 24 through 31 of the first interrupt Trigger Register. This configures interrupt sources 24 through 31 to be edge sensitive.

```
sw      a0, GIC_SH_TRIG31_0 (a1)    // (edge 24..31)
```

4.28.5. Interrupt Dual Edge Registers

There is a set of registers that configures the Edge type if the interrupt is edge signaling.

Register Offset	Interrupt Dual Register numbers	Description
0x0200	0 - 31	Writing a 0x1 to any bit location sets the appropriate external interrupt source to be type dual-edged. **At IP configuration time**, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources. These are read/write bits.
0x0204	32 - 63	
0x0208	64 - 95	
0x020c	96 - 127	
0x0210	128 -159	
0x0214	160 - 191	
0x0218	192 - 223	
0x021c	224 - 255	

4.28.6. Interrupt Set Mask Registers

There is a set of registers that corresponds to the Global Interrupt Reset Mask registers; these are the Global Interrupt Set Mask Registers. Where the Reset Mask registers disable interrupts, the Set Mask Registers enable interrupts.

Register Offset	Interrupt Set Mask Register numbers	Description
0x0380	0 - 31	Writing a 0x1 to any bit location sets the mask (enables) for that interrupt. **At IP configuration time**, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources. These are write only bits.
0x0384	32 - 63	
0x0388	64 - 95	
0x038c	96 - 127	
0x0390	128 -159	
0x0394	160 - 191	
0x0398	192 - 223	
0x039c	224 - 255	

Here is the code that sets the interrupt mask.

```
sw      a0, GIC_SH_SMASK31_00 (a1)  // (enable 24..31)
```

This next section of code configures interrupts 32 through 39 the same way it configured interrupts 24 through 31. The configuration registers that control this range of interrupts is in the second register of each set, so you can see the code is offsetting each register by an additional 4 bytes.

Interrupts 32 through 39 are located in the lower 8 bits of the registers, so the code sets a0 to hex ff and will use this register to set interrupt bits 32 through 39, then disable the interrupts, set the Polarity Registers, set the Trigger Register, and then enable the interrupts.

```
li      a0, 0xff
sw      a0, GIC_SH_RMASK63_32(a1)     // (disable  32..39)
sw      a0, GIC_SH_POL63_32(a1)       // (high/rise 32..39)
sw      a0, GIC_SH_TRIG63_32(a1)      // (edge     32..39)
sw      a0, GIC_SH_SMASK63_32(a1)     // (enable   32..39)
```

4.28.7. Map Interrupt to Processing Unit

Next the code will configure the Processing unit to which a particular interrupt will be assigned. To do this, the GIC has registers for each interrupt source. Each bit in those registers corresponds to a processing unit in the multi-core system. Remember that a processing unit can be a VPE in a multi-threaded, multi-core system or just a single processor in a multi-core system.

For example, for interrupt source 1 registers, bit 0 would assign the interrupt to core 0 in a single core system or to VPE 0 in an MT system. The current scheme supports up to 64 different processing units, so there are 2, 32-bit registers for each interrupt. To allow for future expansion, the registers are spaced 32 bytes apart.

Let's look at the table.

Register Offset	Interrupt Map Src to VPE Register numbers	Description
0x2000	Source 0, 0 - 31	Assigns this interrupt source to a particular VPE. **At IP configuration time**, the appropriate number of these registers is instantiated to support the number of External Interrupt Sources and the number of VPEs. These are read/write bits.
0x2004	Source 0, 32 - 63	
0x2020	Source 1, 0 - 31	
0x2024	Source 1, 32 - 63	
0x2040	Source 2, 0 - 31	
0x2044	Source 2, 32 - 63	
……..		
0x3fe0	Source 255, 0 - 31	
0x3fe4	Source 255, 32 - 63	

The Interrupt Map Src to VPE Registers are in the GIC shared section and start at offset 0x2000. The first interrupt has its registers at 2000 and 2004 hex, thus giving it a 64-bit map area. The next interrupt starts at the start of the section plus 32 bytes or 0x20, so its registers are at 2020 and 2024 hex and so on.

There is a convention in MIPS Linux to use the last 16 interrupt sources for inter-processor interrupts. In the system we are configuring, those are interrupts 24 through 39. The system could contain up to 8 virtual processing units if it is made up of multi-threaded cores, or 4 physical processing units if it is made up of single-threaded cores.

This code assigns 2 interrupt sources to each processing unit in the system. It does this using a0 which is set up with a processor unit number.

So for VPE 0, a0 is programmed with a 1. Then the code stores the value in a0 in the appropriate MAP register. In this case, the code divides the interrupt sources into 2 groups, one group from 24 to 31 and the other from 32 through 39. The code will take one interrupt source from each group and program it to a processing unit .

For VPE 0, the code uses interrupt 24 and 32. The Map registers for 24 will start at GIC offset 0x2300. The number 0x2300 is obtained by multiplying the interrupt number by the spacing size which is 0x20 and adding the offset for the Global Interrupt Map to VPE Registers which is 0x2000.

The code continues until all 8 possible processing units are configured for these interrupts.

```
// Initialize configuration of shared interrupts

// Direct GIC_int24..39 to vpe 0..7
// MIPS Linux convention that last 16 interrupts implemented be set
// aside for IPI signalling.
// (The actual interrupts are tied low and software sends interrupts
```

```
// via GIC_SH_WEDGE writes.)
li a0, 1            // set bit 0 for CORE0 or for MT vpe0
// Source 24 to VPE 0
sw a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 24) (a1)
// Source 32 to VPE 0
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 32)(a1)
sll a0, a0, 1 // set bit 1 for CORE1 or for MT vpe1
// Source 25 to VPE 1
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 25)(a1)
// Source 33 to VPE 1
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 33)(a1)
sll a0, a0, 1 // set bit 2 for CORE2 or for MT vpe2
// Source 26 to VPE 2
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 26)(a1)
// Source 34 to VPE 2
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 34)(a1)
sll a0, a0, 1 // set bit 3 for CORE3 or for MT vpe3
// Source 27 to VPE 3
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 27)(a1)
// Source 35 to VPE 3
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 35)(a1)
sll a0, a0, 1 // set bit 4 for CORE4 or for MT vpe4
// Source 28 to VPE 4
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 28)(a1)
// Source 36 to VPE 4
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 36)(a1)
sll a0, a0, 1 // set bit 5 for CORE5 or for MT vpe5
// Source 29 to VPE 5
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 29)(a1)
// Source 37 to VPE 5
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 37)(a1)
sll a0, a0, 1 // set bit 6 for CORE6 or for MT vpe6

// Source 30 to VPE 6
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 30)(a1)
// Source 38 to VPE 6
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 38)(a1)
sll a0, a0, 1 // set bit 7 for CORE7 or for MT vpe7
// Source 31 to VPE 7
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 31)(a1)
// Source 39 to VPE 7
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 39)(a1)
```

4.28.8. Per-Processor initialization

At this point we have completed initializing the shared section of the GIC. This next section of the code will initialize the per-processor elements of the GIC. The section of registers being initialized is called VPE-Local and is located at GIC offset 0x8000.

Register Fields		Description of the Local Interrupt Control Register (GIC_VPEi_CTL) 0x8000	Reset State
Name	Bits		
FDC_ROUTABLE	4	If this bit is set, the CPU Fast Debug Channel Interrupt is routable within the GIC. If this bit is clear, it is hardwired to one of the *SI_Int* pins as described by the CORE's COP0 *IntCtlIPFDCI* register field.	IP config value
SWINT_ROUTABLE	3	If this bit is set, the CORE SW Interrupts are routable within the GIC. If this bit is clear, it is routed back to the CORE directly.	IP config value
PERFCOUNT_ ROUTABLE	2	If this bit is set, the CORE Performance Counter Interrupt is routable within the GIC. If this bit is clear, it is hardwired to one of the *SI Int* pins as described by the CORE's COP0 *IntCtlIPPCI register* field.	IP config value
TIMER_ROUTABLE	1	If this bit is set, the CORE Timer Interrupt is route-able within the GIC. If this bit is clear, it is hardwired to one of the *SI_Int pins, as described by the* CORE's COP0 *IntCtlIPTI register field.*	IP config value
EIC_MODE	0	Writing a 1 to this bit will set this VPE local interrupt controller to EIC (External Interrupt Controller) mode. It is a read/write bit.	0

The code reads the local interrupt controle rgister which is located at offset 0 in the Local section, so it is at GIC location 8000 hex. The code will be using some of the values from this register.

```
init_vpe_gic:
   // Initialize configuration of per vpe interrupts
   li      a1, (GIC_BASE_ADDR | GIC_CORE_LOCAL_SECTION_OFFSET)
   lw      a3, GIC_COREL_CTL (a1)     // GIC_VPEi_CTL
```

4.28.9. Map Timer interrupt Source

The code checks to see if the timer interrupt is routable. It does this by extracting the Timer routable bit from the Control Register value it just read. Then it checks to see if it's set. If it's not set, the timer interrupt is not routable and the code will branch around routing it.

```
map_timer_int:
   // extract TIMER_ROUTABLE
   ext     a0, a3, TIMER_ROUTABLE, TIMER_ROUTABLE_S
   beqz    a0, map_perfcount_int
```

The table below shows the layout of the Registers that will be programmed.

Register Fields		Description of the Local WatchDog /Compare/PerfCount/SWIntx Map to Pin Registers	Reset State
Name	Bits		
MAP_TO_PIN	31	If this bit is set, this interrupt source is mapped to a VPE interrupt pin (specified by the *MAP field below).* Only one of the *MAP_TO_PIN, MAP_TO_NMI, or MAP_TO_YQ bits can be set at any one time.* It is a read/write bit.	0x1 for Timer, Perf-Count and SWIntx; 0x0 for WatchDog
MAP_TO_NMI	30	If this bit is set, this interrupt source is mapped to NMI. Only one of the *MAP_TO_PIN, MAP_TO_NMI, or MAP_TO_YQ bits can be set at any one time.* It is a read/write bit.	0x1 for WatchDog; 0x0 for Others
MAP_TO_YQ	29	If this bit is set, this interrupt source is mapped to an MT Yield Qualifier pin (specified by the *MAP field below).* Only one of the *MAP TO PIN, MAP TO NMI, or MAP_TO_YQ bits can be set at any one time.* It is a read/write bit.	0
MAP	5:0	When the MAP_TO_PIN bit is set, this field contains the encoded value of the VPE interrupts signals Int[63:0]. The user should only use values of 0 to 5 (decimal). When MAP_TO_YP is set, this field contains the encoded signal selection of the Yield Qualifier.	0

The code sets up a0 with the encoding used to route the local CORE timer interrupt to the desired processor pin. a0 is written with bit 31 set (MAP_TO_PIN) and a 5 in the Map field to map to pin 5. This will map the local Core's timer interrupt to the current Processor's interrupt pin 5. This value is then stored to the GIC Local CORE Timer Map-to-Pin Register.

```
   li      a0, 0x80000005 // Int5 is selected for timer routing
   sw      a0, GIC_COREL_TIMER_MAP(a1)
```

The code checks to see if the Performance Counter interrupt is routable. It does this by extracting the Perfcount Routable bit from the Control Register. Then it checks to see if it's set. If it's not set, the performance counter interrupt is not routable and the code will branch around routing it.

```
map_perfcount_int:
   // extract PERFCOUNT_ROUTABLE
   ext     a0, a3,PERFCOUNT_ROUTABLE, PERFCOUNT_ROUTABLE_S
   beqz    a0, done_gic
   li      a0, 0x80000004             // Int4 is selected for
   sw      a0, GIC_COREL_PERFCTR_MAP (a1)
```

This completes the GIC initialization and the code returns to start.

```
done_gic:
   jr      ra
END(init_gic)
```

4.28.10. cps/init_gic_CM2_64.S (P6600 only)

The P6600 uses CM2.5 which has been adapted from CM2 to work with 64bit values so it needs to use 2 32bit registers instead of 1 32bit register is some cases. The code follows the same logic as the code above but does 2 stores to 2 connective 32bit registers to construct a 64bit value.

4.28.11. cps/init_gic_CM3_64.S (I6400/I6500 only)

The I6400 uses CM3 which changed the approach to working with 64bit values to using 64bit registers instead of 2 32bit registers. The code for the CM3 looks much the same as the code for the CM2 with the exception of using 64bit instructions to work with the 64bit registers. Another change is the processor mapping. The I6400 has 24 possible virtual processors. The mapping starts at interrupt source 24 and goes through interrupt source 47 with only one interrupt being assigned to each VP.

4.29. cps/power_up_cores_CM3_64.S (I6400/I6500 only)

Core 0 VP 0 powers up all other cores.

```
LEAF(power_up_cores)


    blez  r19_more_cores, done_power_up // If no more cores then we are done.
    dli   a3, 1// a3 - core number to be powered up starting with 1
    dli   t0, 1


powerup_next_core:
   // set the other core register to the core to be powered up
   daddu  a0, zero, a3     // copy a3 to a0
   dsll   a0, CPC_CORENUM  // position core number into CORENUM field
```

```
    sd      a0, (CPS_CORE_LOCAL_CONTROL_BLOCK | CPC_OTHERL_REG)(r30_cpc_addr)

    // Make sure all VPs are stopped
    // create bit mask from number of VPs on core
    daddiu  t1, r20_more_vpes, 1  // add 1 to include VP0
    dsllv   t1, t0, t1            // shift 1 by number of VPs
    daddiu  t1, t1, -1            // subtract 1 to complete mask
    sd      a0, (CPS_CORE_LOCAL_CONTROL_BLOCK |
                CPC_CO_VP_STOP_REG)(r30_cpc_addr)

    // power up the other core
    // set the Run bit for VP0 of the other core
    sd      t0, (CPS_CORE_OTHER_CONTROL_BLOCK |
                CPC_CO_VP_RUN_REG)(r30_cpc_addr)
    // Send the power up command to the CPC to power up VP0
    dli     a0, PWR_UP            // "PwrUp" power domain command.
    sd      a0, (CPS_CORE_OTHER_CONTROL_BLOCK | CPC_CMDO_REG)(r30_cpc_addr)
    daddiu  a3, a3, 1
    bne     r19_more_cores, a3, powerup_next_core

done_power_up:
    jalr zero, ra
    nop
END(power_up_cores)
```

4.30. cps/start_VPs_CM3_64.S (I6400/I6500 only)

start_vps is called from each cores VP0 to start the remaining VP within the core. This is done through the VP Run Register located in the local or other section at offset 0x28:

Register Fields		Core Local VP Run Register (CPC_CL_VP_RUN_REG/CPC_CO_VP_RUN_REG)	Reset State
Name	Bits		

VP_RUN	3:0	Each bit of this field corresponds to a VP of a given core. Bit 0 corresponds to VP0, while bit 3 corresponds to VP3. If a bit in this register is set and the corresponding bit in the CPC_CL_VP_RUNNING_REG is 0, then VP will start execution from the reset vector, which is defined as: The sign-extended virtual address programmed in GCR_BEV_BASE (If GCR_CL_RESET_BASE.SELECT_BEV for the corresponding VP is 1). The sign extended virtual address programmed in GCR_CL_RESET_BASE.RESET_BASE (if GCR_CL_RESET_BASE.SELECT_BEV for the corresponding VP is 0).	Set by hardware at reset

```
LEAF(start_vps)
      // set VP_RUN bits
      Dli   a3, 1
      // start remaining VPs of the core
      Beqz  r20_more_vpes, done_start_vps
      // create bit mask from number of VP on core
      Daddu a0, a3, r20_more_vpes   // add 1 to include VP0
      dsllv a0, a3, a0              // shift 1 by number of VPs
      daddiu a0, a0, -1       // subtract 1 to complete mask
      sd    a0, (CPS_CORE_LOCAL_CONTROL_BLOCK | CPC_CL_VP_RUN_REG)(r30_cpc_addr)
      daddu a0, zero, r20_more_vpes //    copy number of vp on this core

next_vp:
      // Set go. This is done because core may already be powered
      // up as is the case for a FPGA
      dla   k0, go_vp
      li    k1, 0xA0000000          // make uncached
      or    k0, k0, k1
      daddu k1, a0, r23_cpu_num     // CPUNum + vp(a0) = VP in system
      dsll  k1, k1, 2               // x 4 for integer element
      daddu k0, k0, k1              // index into the go_vp array
      li    k1, 1                   // load 1
      sw    k1, 0(k0)               // set core element of go_vp array
      sync
```

```
      addiu a0, a0, -1              // subtract 1 from the VP on this core
      bnezc a0, next_vp             // loop again if not equal to 0
done_start_vps:
    jalr    zero, ra
END(start_vps)
```

4.31. cps/join_domain.S - join_domain

Next the code calls the init function to join this core to the Coherent Domain and the rest of the system.

The code first checks to see if this is a Coherent Processing System. If it's not, it will branch to the end of this function and return.

```
LEAF(join_domain)
   beqz    r11_is_cps, done_join_domain    // If CPS then we are done.
```

The Core Local Coherence Control Register located at offset 8 within the Core-Local control block located at 0x2000 of the Global Control Registers controls the entry and exit of a core into the coherent Domain. Bits 0 through 7 represent a coherent requestor within the system.

Register Fields		Core Local Coherence Control Register (GCR_Cx_COHERENCE)	Reset State
Name	Bits		
COH_DOMAIN_EN	7:0	Each bit in this field represents a coherent requester within the CPS. Setting a bit within this field will enable interventions to this Core from that requester. The requestor bit which represents the local core is used to enable or disable coherence mode in the local core. Changing the coherence mode for a local core from 0x1 to 0x0 can only be done after flushing and invalidating all the cache lines in the core; otherwise, the system behavior is UNDEFINED.	0x0

The code sets the first 4 bits of a0 to 1. Then it stores it to the Core Local Coherence Control Register. This enables the other three cores possible in this system to communicate via interventions to this core.

```
   // Enable coherence and allow interventions from all other cores.
   // (Write access enabled via GCR_ACCESS by core 0.)
   Li    a0, 1
   addiu a1, r19_more_cores, 1         // add 1 to include Core 0
   sllv  a1, a0, a1                    // shift 1 by number of cores
   addiu a1, a1, -1                    // subtract 1 to complete mask
```

```
    sw    a1, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_COHERENCE)(r22_gcr_addr)
```

The code initializes a3 which it will use as a loop counter to 0.

```
    move    a3, zero
```

Next is the loop back label next_coherent_core, which the code will loop back to and join the next core domain.

next_coherent_core:

The code sets up the Core-Other Addressing Register, offset 2018 hex from the GCR register base, with the core number it wants to join with.

Register Fields		Core-Other Addressing Register (GCR_Cx_OTHER)	Reset State
Name	Bits		
CoreNum	31:16	CoreNum of the register set to be accessed in the Core-Other address space.	0x0

It first stores the value of the core into a0, shifts it into the upper 16 bits, and stores it to the Core-Other Addressing Register.

```
    sll     a0, a3, 16
    sw a0,(CORE_LOCAL_CONTROL_BLOCK|GCR_CL_OTHER) (r22_gcr_addr)
```

The code now reads the Core Local Coherence Control Register of the other core. Recall how the code set this core's Core Local Coherence Control Register to enable interventions from other cores, thus entering the domain. The code now needs to wait for the other core to do the same.

```
busy_wait_coherent_core:
    lw    a0,(CORE_OTHER_CONTROL_BLOCK|GCR_CO_COHERENCE) (r22_gcr_addr)
    beqz  a0, busy_wait_coherent_core    // Busy wait
```

Once the other core has joined, the code checks to see if there are more cores to wait for and if there are, it branches back to the next coherent core label. It also increments the other core count.

```
    addiu   a3, 1
    bne     a3, r19_more_cores, next_coherent_core
```

When all the cores have been waited for, the code returns to start.

```
done_join_domain:
    jr      ra
END(join_domain)
```

4.31.1. cps/join_domain_CM3_64.S (I6400/I6500)

The I6400/I6500 uses CM3 which changed the approach to working with 64bit values to using 64bit registers instead of 2 32bit registers. The code for the CM3 the same steps as the code for the CM2 using 64bit instructions to work with the 64bit registers. The CM3 has a different register to enable coherence on each VP. It is called the Core-Local/Core-Other Coherence

Enable Register. Each core just enables itself to be part of the coherent domain by setting bit 0 of this register. The code looks like this:

```
   // Enable coherence for this core
   // (Write access enabled via GCR_ACCESS by core 0.)
   dli       a0, 0x01            // Set Coherent enable bit  for core
   sd        a0, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_COH_EN)(r22_gcr_addr)
   daddu  a3, zero, r19_more_cores     // set core number to highest core
   beq      a3, zero, done_enable_coherence  // branch out if only core 0
```

4.32. cps/release_mp.S - release_mp (interAptiv, interAptivUP only)

After the first processor in an MP system has completed the boot code, it can release the remaining processors to execute the boot code.

The code checks to see if there are more cores in the system, and if not, it branches to the end of this section and returns.

```
LEAF(release_mp)
   blez    r19_more_cores, done_release_mp    // If no more cores done.
```

The code uses a3 as a counter to decide if it has released all the remaining cores.

```
   li      a3, 1
```

The code checks for a Cluster Power Controller by checking if the address was set for the CPC register block. If this value is 0, there is no CPC, and the code will skip ahead and just release the next core so it can begin execution.

```
   beqz    r30_cpc_addr, release_next_core    // If no CPC then use
                                              // GCR_CO_RESET_RELEASE
```

For systems that have a Cluster Power Controller, only one processor should be powered up when power is first applied. That first processor needs to power up the remaining processors in order for them to continue the boot process. To do that the code will send the power up signal to each of the other cores in the system using the Cluster Power Controller. At offset 0x2010 from the base address of the CPC is the Core-Other Addressing Register shown here. The code needs to place the number of the other core it wants to power up in this register.

To do this the code moves the number of the core it wants to power up into a0. Recall that the first time through, a 1 was stored in a3.

```
powerup_next_core:
   // Send PwrUp command to next core causing execution at
   // their reset exception vector.
   move    a0, a3
```

Next the code shifts that value to the left into the range of the Core Number field of the Core-Other Addressing Register. Then that value is stored to the Core-Other Addressing Register.

```
   sll     a0, 16
   sw      a0,(CPS_CORE_LOCAL_CONTROL_BLOCK|CPC_OTHERL_REG)(r30_cpc_addr)
```

Now the code can use the Core-Other Control Block within the CPC located at offset 0x4000 to control the core whose number it just placed in the Core-Other Addressing Register. The first

register in that block is the CPC Local Command Register. This register is used to power up or down signals for the core. It has a command field called CMD in its first 4 bits.

Register Fields		Local Command Register (CPC_CMD_REG)	Reset State
Name	Bits		
CMD	3:0	Requests a new power sequence execution for this domain. Read value is the last executed command. Code: 4'd3 — Meaning: **PwrUp -** this domain using setup values in CPC_STAT_CONF_REG. Usable only for Core-Others access. It is the software equivalent of the SI_PwrUp hardware signal.	0x0

Right now we are interested in powering up the core. That command is 3, so the Code loads a 3 into a0 and stores that register to the address of the CPC at offset 0x4000. This will power up the core and it will begin executing at the reset exception vector, which is the start of this boot code.

```
li      a0, PWR_UP          // "PwrUp" power domain command.
sw      a0,(CPS_CORE_OTHER_CONTROL_BLOCK|CPC_CMDO_REG)(r30_cpc_addr)
```

Next the code checks to see if there are any other processors in the system by comparing the current core number with the highest core number that was previously stored in r19_more_cores. The processor count is incremented. If there are other cores, the code loops back to power up the next processor.

```
addiu   a3, a3, 1
bne     r19_more_cores, a3, powerup_next_core
```

When all of the processors have been powered up, the function returns to start.

```
jalr    zero, ra
```

The following code is executed in older Coherent Processing Systems without a Cluster Power Controller. When there is no CPC, the cores other than Core0 are held in reset until Core0 releases them.

This loop will set the core other value so it can get to the other core's GCR reset release.

```
release_next_core:
   // Release next core to execute at their reset exception vector.
   move    a0, a3
   sll     a0, 16
   sw      a0,(CORE_LOCAL_CONTROL_BLOCK|GCR_CL_OTHER)  (r22_gcr_addr)
```

It then stores a zero to the GCR_CO_RESET_RELEASE register to release that core from reset.

```
   sw      zero, 0x4000(r22_gcr_addr)       // write GCR_CO_RESET_RELEASE
```

It continues looping until all cores are released.

```
    addiu   a3, a3, 1
    bge     a3, r19_more_cores, release_next_core
```

Once all the cores have been released from reset, the code returns to start.

```
done_release_mp:
   jalr    zero, ra
END(release_mp)
```

4.33. Boston/init_FPGA_mem.S

The code will initialize the Boston memory controller. It is not covered here since it is unlikely you would be using the same memory controller as a Boston Board.

4.34. mt/init_vpe1.S - init_vpe1 (interAptivUP or interAptiv only)

This code initializes the second VPE of an MT system.

Defines are created to make it easier to follow the code.

```
#define target_TC a3        // will hold the current TC being configured
#define VPE_1 1             // used to load register
```

It first checks to see if there is an additional TC to bind to a second VPE and then if there is a second VPE. If neither is true, no action is required so the code will jump to the done point.

```
LEAF(init_vpe1)
   beqz    r21_more_tcs, done_init_vpe1
   beqz    r20_more_vpes, done_init_vpe1
```

To setup the second VPE will require access to some registers that are usually non-writable. To write to these registers, the code needs to enable Virtual Processor Configuration. This is done by setting the VPC bit in the MVPControl register (CP0 register 0, select 1). The code reads the Register.

```
   // This is executing on TC0 bound to VPE0.
   // Therefore VPEConf0.MVP is set to enter config mode
   mfc0  a0, C0_MVPCONTROL // C0_MVPCtl
```

Then it sets the VPC bit (bit 1),

```
   or    a0, (1 << 1)      // M_MVPCtlVPC
```

writes it back to CP0, and executes an ehb to ensure the write has been completed before it continues.

```
   mtc0  a0,  C0_MVPCONTROL      // C0_MVPCtl
   ehb
   // Initialize target_TC, a3_target_TC will be incremented at the
```

```
   // bottom of the loop if there are more TCs
   li    target_TC, 1

nexttc:
```

Setup the Target TC

```
   // Set TargTC in the CP0 VPECONTROL register
   // TargTC  Selects the TC number of the "other thread context" for
   // any Move to Thread Context or Move from Thread Context
   // instructions (any instructions that begin with mtt or mft)

   mfc0  a0, C0_VPECONTROL       // read C0_VPECTL
   ins   a0, a3_target_TC, 0, 8        // insert TargTC
   mtc0  a0, C0_VPECONTROL       // write C0_VPECTL
   ehb
```

The TC should be halted before the code starts changing its configuration. To do that, a 1 is placed in v0 and then moved to the C0_TCHalt register (CP0 register 2 select 4). The code executes an ehb to ensure the move has taken effect.

```
   // Halt TC being configured
   li    a0, 1             // TCHalt
   mttc0 a0, C0_TCHALT     // C0_TCHalt
   ehb
```

Next the code sets this TC to prevent it from taking interrupts and clears all other status and control bits in the TCStatus register. It does this by setting up v0 to all 0s except for the IXMT bit (10). Then it writes that value to the TCStatus register (CP0 register 2 select 1).

```
   // Set up TCStatus register:
   // Disable Coprocessor Usable bits
   // Disable MDMX/DSP ASE
   // Clear Dirty a3_target_TC
   // not dynamically allocatable
   // not allocated
   // Kernel mode
   // interrupt exempt
   // ASID 0
   // NOTE: Only bit that needs to be set is IXMT
   li    a0, (1 << 10)     // IXMT bit 10 = 1
   mttc0 a0, C0_TCSTATUS   // C0_TCStatus
```

The code now initializes the TC's GPR registers using the same method used in init_gpr.S.

```
    li $2, 0xdeadbeef
   // Initialize the target_TC a3's register file
   // (NOTE: $2 is in the gpr of the tc executing  this code)
   // NOTE: Good practice but not programmatically necessary
```

```
    mttgpr      $2, $1
    mttgpr      $2, $2
    mttgpr      $2, $3
    mttgpr      $2, $4
    mttgpr      $2, $5
    mttgpr      $2, $6
    mttgpr      $2, $7
    mttgpr      $2, $8
    mttgpr      $2, $9
    mttgpr      $2, $10
    mttgpr      $2, $11
    mttgpr      $2, $12
    mttgpr      $2, $13
    mttgpr      $2, $14
    mttgpr      $2, $15
    mttgpr      $2, $14
    mttgpr      $2, $17
    mttgpr      $2, $18
    mttgpr      $2, $19
    mttgpr      $2, $20
    mttgpr      $2, $21
    mttgpr      $2, $22
    mttgpr      $2, $23
    mttgpr      $2, $24
    mttgpr      $2, $25
    mttgpr      $2, $26
    mttgpr      $2, $27
    mttgpr      $2, $28
    mttgpr      $2, $29
    mttgpr      $2, $30
    mttgpr      $2, $31
```

The code will now bind the TC to the VPE..

It does this by reading the TCBind register, inserting the VPE number into the CurVPE Field, and writing it back.

```
    // Bind TC to a VPE
    li    a0, VPE_1
    mftc0 a1, C0_TCBIND     // Read C0_TCBind
    ins   a1, a0, 0, 4      // insert vpe 1 into CurVPE field
    mttc0 a1, C0_TCBIND     // write C0_TCBind
```

To insure the code next is only done once the code checks to see if this is TC1

```
    // Must only do next part up to check_for_more_TC label only once
```

```
bne   a0, target_TC, check_for_more_TC // branch if not TC1
```

Next set this TC to be the only TC runnable on the VPE. For current cores, this effectively sets TC 1 to run exclusively on VPE 1. To do this, the XTC field (bits 21 – 28) in the VPEConf0 register (CP0 register 1 select 2) is set to the TC number. The code reads the VPEConf0 register, inserts the TC number into the XTC field, and writes the register back.

```
// Set XTC for active TC's
mftc0 a0, C0_VPECONF0   // read C0_VPEConf0
ins   a0, target_TC, 21, 8    // insert TC -> XTC
mttc0 a0, C0_VPECONF0   // write C0_VPEConf0
```

First the code makes sure multi-threading is disabled. It needs to do this because only one TC should be executing this code at a time. It does this by clearing the TE bit (15) in the VPEControl register (CP0 register 1 select 1). The code reads the register, inserts a 0 into to the bit, and writes the register back.

```
// Disable multi-threading for VPE1
mftc0 a0, C0_VPECONTROL       // read C0_VPECTL
ins   a0, zero, 15, 1         // clear TE (only tc1 can execute code)
mttc0 a0, C0_VPECONTROL       // write C0_VPECTL
```

The code needs to make sure that no TC is running on the VPE it is initializing. It does this by reading the VPEConf0 (CP0 register 1, select 2) of the VPE it is initializing and inserting a 0 into the VPA Field (Virtual Processor Activated). It also needs to ensure that it is the Master Virtual Processor by setting the MVP bit. This enables the writing of registers associated with the VPE. Then the code writes it back to the VPEConf0 register of the VPE.

```
// For VPE1..n
// Clear VPA and set master VPE
mftc0 a0, C0_VPECONF0   // read C0_VPEConf0
ins   a0, zero, 0, 1    // clear VPA
or    a0, (1 << 1)      // set MVP
mttc0 a0, C0_VPECONF0   // write C0_VPEConf0
```

Next copy the Status register of the running TC to the Status register of the TC being initialized.

```
mfc0  a0, C0_STATUS     // read C0_Status
mttc0 a0, C0_STATUS     // write C0_Status
```

Initialize the Error PC to a dummy value.

```
li    a0, 0x12345678
mttc0 a0, C0_EPC        // write C0_EPC
```

Clear the Cause register.

```
mttc0 zero, C0_CAUSE    // write C0_Cause
```

Copy the Config register of the running TC to the Status register of the TC being initialized.

```
mfc0  a0, C0_CONFIG     // read C0_Config
mttc0 a0, C0_CONFIG     // write C0_Config
```

The code now puts the Core number into r23_cpu_num of the TC being initialized. It does this by reading the EBASE register (CP0 register 15, select 1) and extracting the Cpu_num field. Then it copies the CPUNum to the TC's r23_cpu_num.

```
   mftc0 a0, C0_EBASE      // read C0_EBASE
   ext   a0, a0, 0, 10     // extract CPUNum
   mttgpr a0, r23_cpu_num
```

The code programs the TC's reset vector so that when it is set to run, it will start executing the boot code. It loads the address of the reset_vector from the label created in the linker file. It sets bit 29 to convert the address to a KSEG0 address so it will execute from a cacheable address. Then it writes this value to the TC's TCRestart register (CP0 2 select 3).

```
   // vpe1 of each core can execute cached as it's L1 I$ has
   // already been initialized.
   // and the L2$ has been initialized or "disabled" via CCA override.
   la    a1, __reset_vector
   #ifndef     EVA         // trick not need for EVA boot
         ins   a1, zero, 29, 1   // Convert to cacheable cached kseg0 address
   #endif
   mttc0 a1, C0_TCRESTART  // write C0_TCRestart
```

Now the first thread for this VPE is ready to run, so the code sets it to start running . However, it will not run until all TCs have been initialized and the code exits VPE config mode and enables virtual processing, which is does at the end of this function.

The code reads the TCStatus register and enables the TC for handling interrupts by clearing the IXMT bit (10). (This doesn't really enable interrupts; it just makes it possible for this TC to access them.) It also activates the TC by setting the A bit (13). Then it writes the value to the TC's TCStatus register.

```
   mftc0 a0, C0_TCSTATUS   // read TCStatus
   ins   a0, zero, 10, 1   // clear IXMT
   li    a1, 1
   ins   a0, a1, 13, 1     // set A to Activate this TC
   mttc0 a0, C0_TCSTATUS   // write TCStatus
```

Now the code un-halts the TC by clearing the H field in its TCHalt register. (No other bit needs to be set, so it just clears the whole register.)

```
   mttc0 zero, C0_TCHALT   // clear H in TCHalt
check_for_more_TC:         // Done initializing VPE 1 if there was one


   addu  target_TC, 1      // advance TC number
   // set a1 if TC number is less than total # of TC in system
   sltu  a1, r21_more_tcs, target_TC
   beqz  a1, nexttc        // go back and initialize another TC
```

The code then sets the Virtual Processor Activated (VPA) bit in the VPEConf0 register to activate the VPE and allow the TC it has just initialized to start running. It does this by reading the initialized VPE's VPEConf0 register and setting the VPA bit, then writing it back.

```
   mftc0 a0, C0_VPECONF0   // read VPEConf0
```

```
   ori   a0, 1             // set VPA
   mttc0 a0, C0_VPECONF0   // write VPEConf0
```

At this point, all TCs and VPEs have been initialized. The code needs to “Enable Virtual Processing” and take the processor out of “Virtual Processor Configuration” mode. The code will read the MVPCtl register (CP0 register 0 select 1), set the EVP bit (1), and clear the VPC bit.

```
   // Exit config mode
   mfc0  a0, C0_MVPCONTROL // read C0_MVPCtl
   // set EVP (Enable Virtual Processing) to enable execution by vpe1
   ori  a0, 1
   ins   a0, zero, 1, 1    // Clear VPC (VPE Configuration State) bit
   mtc0  a0, C0_MVPCONTROL // write C0_MVPCtl
   ehb
```

This function is done and returns to start.

```
done_init_vpe1:
   jr    ra
END(init_vpe1)
```

Now for some clean-up of the code to remove the “//defines” it created in the beginning of this file.

```
#undef target_TC
#undef VPE_1
```

4.35. main.c

The main.c code that is located in each core's directory differs depending on the core and the capabilities being demonstrated. This section contains subsections that describe each core's main.c. All are intended to use the BOSTON display to display VPE or CORE numbers. There is no output for simulators.

```
void boston_lcd_display_byte_ab(unsigned int, char);
```

Also for all cores, the boot_count array is intended to hold the cycle time it took to boot.

```
volatile int boot_count[8];//global variable zeroed in start.S
                           // when bss is zeroed
```

4.35.1. main.c for P5600 single core

The number of the CORE that is executing this code is passed in the cpu_num argument. In the case of the interAptivUP single VPE/thread and P5600 single core, this will always be 0.

```
// Global variables.
//
// main():
//
int main(unsigned int cpu_num) {


    int  j, temp;
```

The inline assembly code reads the Count register (CP0 register 9) into the temp variable. Then it writes it to the element of the boot_count array that corresponds to cpu_num.

```
    // End timing of boot for this "Core".
    asm volatile ("mfc0 %[temp], v1": [temp] "=r"(temp) :) ;
    boot_count[cpu_num] = temp ;
```

Next it displays the CORE number on the display by writing to it. It does this in a loop, so it will display for a time, then the next code will blank the display in a similar loop, and then go back to the top of the while loop. The effect will be a blinking 0 on the BOSTON display.

```
    while (1) {
        display_char = (char)(cpu_num + 0x30);   // ASCII char for cpu number.
        blank = (char)0x20;
        for (j = 0 ; j < (4 * 8192) ; j++) {
         boston_lcd_display_byte_ab(cpu_num, display_char);
        }
        for (j = 0 ; j < (4 * 8192) ; j++) {
         boston_lcd_display_byte_ab(cpu_num, blank); // blank out display
        }
    }
```

The code should never get to this point, because the while loop above is endless.

```
    return 0 ;
}
```

4.35.2. main.c for interAptivUP

For the interAptivUP, VPE0 will wait until VPE1 (if present) has gone through this boot code and is executing in main.c. The BOSTON display is used to display status and (in the end) the VPE number as the VPE works through the code.

```
// Global variables.
//
// main():
//
int main(unsigned int vpe_num, unsigned int num_vpe_this_core) {


    int i, j, k ;
    int num_cpus = 0 ;
    char display_char, blank;
    int temp;
```

The inline assembly code reads the Count register (CP0 register 9) into the temp variable. Then it writes it to the 0 element of the boot_count array.

```
    // End timing of boot for this "Core".
    asm volatile ("mfc0 %[temp], v1": [temp] "=r"(temp) :) ;
    boot_count[vpe_num] = temp ;
```

If the code is executing on VPE0, it will wait for any other VPEs to set their ready bit in the ready array. While it's waiting, it will display a "W" in its segment on the BOSTON display.

```
    // VPE0 synchronizes test code execution.
    if (vpe_num== 0) {
           // Wait for other VPEs to indicate they are ready.
           for (i = 1; i < num_vpe_this_core; i++) {
                    boston_lcd_display_byte_ab (vpe_num, 'W' ;
                    target_TC (!ready[i]) ;  // Busy wait for all VPEs
           }
```

If the code is executing on VPE1, then it will display an "r" (for ready) in its segment of the BOSTON display and write a 1 to its element of the ready array. The ready array is used in the code above by VPE0.

```
    } else {
         // Other VPE indicates ready and waits...
         boston_lcd_display_byte_ab (vpe_num,  'r' ; // display 'r' for ready.
         ready[vpe_num] = 1 ;
    }
```

Next it displays the VPE number on the display by writing to it. It does this in a loop, so it will display for a time, and then the next code will blank the display in a similar loop and go back to the top of the while loop. The effect will be a blinking 0 on the display.

```
    while (1) {
         for (j = 0 ; j < (4 * 1024) ; j++) {
               boston_lcd_display_byte_ab(vpe_num, display_char);.
         }
         for (j = 0 ; j < (4 * 1024) ; j++) {
               boston_lcd_display_byte_ab(vpe_num, blank);// blank position
         }
    }
```

The code should never get to this point, because the while loop above is endless.

```
    return 0 ;
}
```

4.35.3. main.c for P5600 CPS

For the P5600 CPS, Core0 will wait until the remaining CPUs (if present) have gone through this boot code and are executing in main(). All CPUs except CPU0 will wait for an Inter-processor Interrupt before they continue. CPU0 will just wait until all other CPUs have set their ready element in the ready array. Then CPU0 will continue.

All Cores will display their CORE number in their corresponding segment on the BOSTON display.

The set_ipi function will send an interrupt through the Global Interrupt Controller to a specific CORE number provided by cpu_num. The init_gic.S code sets up the GIC to wire each interrupt to a specific CORE.

```
#define FIRST_IPI           32 // GIC interrupts 32+ are used to signal
                            // interrupts between cores.
// set_ipi(): Send an inter-processor interrupt // to the specified Core.
void set_ipi(int cpu_num) {
    // Use external interrupts 32..39 for ipi
    GIC_SH_WEDGE = 0x80000000 + FIRST_IPI + cpu_num;}
```

Interrupts 23 through 39 correspond to a CORE, so CPU0 uses interrupt 32, CPU1 uses interrupt 33, and so on.

```
#define GIC_SH_WEDGE        *((volatile unsigned int*) (0xbbdc0280))
```

The #define GIC_SH_WEDGE sets up a pointer to the GIC Global Interrupt Write Edge Register. It does so by combining the address of the GIC register control block, which in this case is at 0xbbdc 0000 with the offset 0x280 of the Global Interrupt Write Edge Register.

The value written to the Global Interrupt Write Edge Register has two parts: bit 31 is set to indicate that the code is sending the interrupt signal, and bits 0 through 30 determine to which interrupt the signal will be sent. The code calculates the proper interrupt by using the #define FIRST_IPI as a base interrupt number and adding the CORE number.

Once this value is written, the corresponding CORE will receive an interrupt which will wake it up to continue executing.

Main has the single argument cpu_num. This is the processor number of the P5600 Core that is executing this code.

```
//
// main(): Synchronized run of shared test code coordinated by cpu0
//
int main(unsigned int cpu_num) {


   int i, j, k ;
   int num_cpus = 0 ;
   char display_char, blank;
   int temp;
```

The inline assembly code reads the Count register (CP0 register 9) into the temp variable. Then it writes it to the 0 element of the boot_count array.

```
   // End timing of boot for this "Core"
   asm volatile ("mfc0 %[temp], v1": [temp] "=r"(temp) :) ;
   boot_count[cpu_num] = temp ;
```

If the code is executing on CPU0, it will wait for any other Cores to set their ready bit in the ready array. While it's waiting, it will display a "W" in its segment on the BOSTON display.

```
   // cpu0 synchronizes test code execution.
   if (cpu_num == 0) {
         num_cpus = num_cores;
         // Wait for other Cores to indicate they are ready.
         for (i = 1; i < num_cpus; i++) {
               boston_lcd_display_byte_ab(cpu_num, 'W');
               // Busy wait for all Core to be ready.
               while (!ready[i]) ;
          }
```

When a core reports it is ready, it will call the wait instruction which will stop the core until the core receives an interrupt. It is CORE 0's job to send the interrupt to wake up the other cores so they can continue processing. It does this by using inter-processor interrupts that were set up in the init_gic.S code. The code first changes the segment display by writing an "I" to the corresponding element for the core it is going to interrupt. Then it calls the set_ipi function to send the interrupt. Once CPU0 finishes this loop, it can continue and execute test code and/or the OS.

```
         // Release other Core to run their tasks.
         for (i = 1; i < num_cpus; i++) {
               // display 'i' for interrupted (from cpu0)
               boston_lcd_display_byte_ab(i, 'i');
               set_ipi(i) ;            // Send the ipi
         }
```

All CPUs other than CORE 0 will halt and wait for CORE 0 to synchronize them. First the code makes sure the interrupt source bit corresponding to its CORE number is cleared by writing to

the GIC_SH_WEDGE register. Notice that bit 31 is not set, so this clears any interrupt that might be pending. Then it enables interrupts.

```
} else {
      // Clear this Core's IPI source
      GIC_SH_WEDGE = FIRST_IPI + cpu_num ;
      // Enable interrupts and wait to be released
      // via ipi from cpu0
      asm volatile ("ei") ;
```

Next each CORE will write an "r" to the segment display to indicate it is ready. Then it will write to the global array to indicate to CORE 0 that it is ready.

```
      // Other Core indicates ready and waits...
      boston_lcd_display_byte_ab(cpu_num, 'r');
ready[cpu_num] = 1 ;
```

Next interrupts are disabled for the CORE. This avoids any race condition between the testing of the start_test array and the wait instruction.

```
      asm volatile ("di") ;
```

The code will loop, testing its element of the start_test array, and call wait to wait for an interrupt (send by CPU0).

```
      while (!start_test[cpu_num]) {
            // This code will only work reliably if the
            // WII bit is set in config7.
            // When this bit is set, any interrupt even
            // when they are disabled will cause
            // wait to return. This avoids a race condition.

             // Wait for interrupt (qualified with "start_test").
            asm volatile ("wait") ;
```

Once an interrupt occurs, the code enables interrupts so its interrupt service routine can run and process the interrupt. The interrupt routine will set the start_test element for this CORE.

```
             // enable interrupts so interrupt routine can run
             // and set the start_test bit
            asm volatile ("ei") ;
            asm volatile ("ehb") ;
```

By the time the code gets here, the interrupt routine will have run. The code will disable interrupts before it goes back to the top of the loop. The top of the loop is where the start_test array is checked and just as before, interrupts need to be disabled to avoid a race condition. The start_ array needs to be checked, because any interrupt could have caused termination of the wait instruction.

```
            // Disable interrupts again so there is not race
            // condition between testing the
            // start_test bit variable and going back to wait.
```

```
                // NOTE for this code we are only expecting IPI that
                // interrupt.
                asm volatile ("di") ;
                asm volatile ("ehb") ;
        }
    }
```

Next it displays the CORE number on the display by writing to it. It does this in a loop, so it will display for a time, and then the next code will blank the display in a similar loop and go back to the top of the while loop. The effect will be a blinking 0 on the display.

```
   display_char = (char)(cpu_num + 0x30);
   blank = (char)0x20;
   while (1) {
        target_TC (j = 0 ; j < (4 * 1024) ; j++) {
            boston_lcd_display_byte_ab(cpu_num, display_char);
        }
        for (j = 0 ; j < (4 * 1024) ; j++) {
            boston_lcd_display_byte_ab(cpu_num, blank);
        }
   }
   return 0 ;
}
```

4.35.4. main.c for interAptiv CPS

For interAptiv CPS, main() defines a macro that it uses to write to the BOSTON Board's address for its 8-segment display. VPE0 will wait until the remaining VPEs (if present) have gone through this boot code and are executing in main. All VPEs except VPE0 will wait for an inter-processor Interrupt before they continue. VPE0 will just wait until all other VPEs have set their ready element in the ready array. Then VPE0 will continue.

All VPEs will display their number in their segment on the BOSTON display.

Note: For a CPS, the VPE number will be a combination of the VPE number within a Core and the Core number + 1. For example, VPE0 of Core 0 is numbered 0, and VPE 0 of Core 1 is numbered 2.

The set_ipi function will send an interrupt through the Global Interrupt Controller to a specific VPE number provided by cpu_num. The init_gic.S code sets up the GIC to wire each interrupt to a specific VPE.

Interrupts 32 through 39 correspond to a VPE, so VPE0 uses interrupt 32, VPE1 interrupt 33, and so on.

The #define GIC_SH_WEDGE sets up a pointer to the GIC Global Interrupt Write Edge Register. It does so by combining the address of the GIC register control block, which in this case is at 0xbbdc 0000, with the offset 0x280 of the Global Interrupt Write Edge Register.

The value written to the Global Interrupt Write Edge Register has two parts: bit 31 is set to indicate that the code is sending the interrupt signal, and bits 0 through 30 determine which interrupt the signal will be sent to. The code calculates the proper interrupt by using the #define FIRST IPI as a base interrupt number and adding the CPU number.

After this value has been written, the corresponding VPE will receive an interrupt that will wake it up to continue executing code where it left off.

```
#define GIC_SH_WEDGE   *((volatile unsigned int*) (0xbbdc0280))
#define FIRST_IPI 32 // GIC interrupts 32+ are used to signal
                     // interrupts between cores.
//
// set_ipi(): Send an inter-processor interrupt to the specified VPE.
//
void set_ipi(int cpu_num) {
   // Use external interrupts 32..39 for ipi
   GIC_SH_WEDGE = 0x80000000 + FIRST_IPI + cpu_num ; }

//
// main(): Synchronized run of shared test code coordinated by cpu0.
//
int main(unsigned int cpu_num, unsigned int core_num, unsigned int vpe_num,
unsigned int num_vpe_this_core) {
   int i, j, k ;
   int num_cpus = 0 ;
   char display_char, blank;
   int temp;
```

The inline assembly code reads the Count register (CP0 register 9) into the temp variable. Then it writes it to the 0 element of the boot_count array.

```
// End timing of boot for this "VPE".
asm volatile ("mfc0 %[temp], v1": [temp] "=r"(temp) :) ;
boot_count[cpu_num] = temp ;
// if there is only one core skip the sync step

if (num_cores != 1)
    {

   // Each core's vpe0 reports number of vpe in the core.
   if (vpe_num == 0) {
         vpe_on_core[core_num] = num_vpe_this_core ;
   }
```

If the code is executing on CPU0 (VPE0), it waits for each core to report the number of VPEs on a core. The code tallies the number of VPEs each core reports in num_cpus, which it will use a little later in this loop.

```
// cpu0 synchronizes test code execution.
if (cpu_num == 0) {


      // Tally number of "VPEs" there are in this CPS.
      for (i = 0; i < num_cores; i++) {
            while (!vpe_on_core[i]) {
            }     // Busy wait for core to report number of vpe.
            num_cpus += vpe_on_core[i];
      }
```

It will wait for any other VPEs to set their ready bit in the ready array. While it's waiting, it will display a "W" in its segment on the BOSTON Display.

```
      // Wait for other VPEs to indicate they are ready.
      for (i = 1; i < num_cpus; i++) {
            boston_lcd_display_byte_ab(cpu_num, 'W');
            // Busy wait for all VPEs to be ready.
            while (!ready[i]) ;
      }
```

When a VPE reports it is ready, it will call the wait instruction, which will stop the VPE until the VPE receives an interrupt. It is VPE 0's job to send the interrupt to wake up the other cores so they can continue processing. It does this by using inter-processor interrupts that were set up in the init_gic.S code. The code first changes the segment display by writing an "I" to the corresponding element for the core it is going to interrupt. Then it calls the set_ipi function to send the interrupt. Once CPU0 finishes this loop, it can continue and execute test code and/or the OS.

```
      // Release other VPE to run their tasks.
      for (i = 1; i < num_cpus; i++) {
            // display 'i' for interrupted (from cpu0.)
            boston_lcd_display_byte_ab(i, 'i');
            set_ipi(i) ;            // Send the ipi
      }
```

All VPEs other than VPE 0 will Stop and wait for VPE 0 to synchronize them. First the code makes sure the interrupt source bit corresponding to its VPE number is cleared by writing to the GIC_SH_WEDGE register. Notice that bit 31 is not set, so this clears any interrupt that might be pending. Then it enables interrupts.

```
} else {
      // Clear this "VPE"'s ipi source
      GIC_SH_WEDGE = FIRST_IPI + cpu_num;
      // Enable interrupts and wait to be released
      // via ipi from cpu0
      asm volatile ("ei") ;
```

Next each VPE will write an “r” to the segment display to indicate it is ready. Then it will write to the global array to indicate to VPE 0 that it is ready.

```
        // Other VPE indicate ready and wait...
        boston_lcd_display_byte_ab(cpu_num, 'r');
        ready[cpu_num] = 1 ;
```

Next interrupts are disabled for the VPE. This avoids any race condition between the testing of the start_test array and the wait instruction.

```
        asm volatile ("di") ;
```

The code will loop, testing its element of the start_test array and calling wait to wait for an interrupt (sent by CPU0).

```
        while (!start_test[cpu_num]) {
              // This code will only work reliably if the
              // WII bit is set in config7.
              // When this bit is set any interrupt even
              // when they are disabled will cause
              // wait to return. This avoids a race condition

              // Wait for interrupt (qualified with "start_test").
              asm volatile ("wait") ;
```

When an interrupt is signalled, it enables interrupts so that its interrupt service routine can run and process the interrupt. The interrupt routine will set the start_test element for this CPU.

```
              // enable interrupts so interrupt routine can run
              // and set the start_test bit
              asm volatile ("ei") ;
              asm volatile ("ehb") ;
```

By the time the reaches this point, the interrupt routine will have run. The code will disable interrupts before it returns to the top of the loop. The top of the loop is where the start_test array is checked and just as before, interrupts need to be disabled to avoid a race condition. The start_ array needs to be checked because any interrupt could have terminated the wait instruction.

```
              // Disable interrupts again so there is no race
              // condition between testing the
              // start_test bit variable and going back to wait.
              // NOTE: for this code we are only expecting an
              // interprocessor interrupt.
              asm volatile ("di") ;
              asm volatile ("ehb") ;
        }
   }
}
```

Next it displays the VPE number on the display by writing to it. It does this in a loop, so it will display for a time, then the next code will blank the display in a similar loop and go back to the top of the while loop. The effect will be a blinking 0 on the BOSTON display.

```
    while (1) {
          for (j = 0 ; j < (4 * 1024) ; j++) {
                boston_lcd_display_byte_ab(cpu_num, display_char); // VPE
          }
          for (j = 0 ; j < (4 * 1024) ; j++) {
                boston_lcd_display_byte_ab(cpu_num, blank);// blank out display
          }
    }
    return 0 ;
}
```

5. Makefiles

There are two Makefiles used with each core's build: a top-level Makefile and a core-level Makefile.

5.1. Top Level Makefile

The top level Makefile can contain up to 5 targets for each core.

Using the interAptiv core as an example, the targets are:

interAptivUP_SIM_RAM

Builds a simulator version with the main function copied to normal RAM.

```
interAptivUP_SIM_RAM:
   ${MAKE} -C interAptivUP SIM_RAM
```

interAptivUP_SIM_SPRAM

Builds a simulator version with the main function copied to Scratchpad RAM.

```
interAptivUP_SIM_SPRAM:
   ${MAKE} -C interAptivUP SIM_SPRAM
```

interAptivUP_BOSTON_RAM

Builds a BOSTON Board version with the main function copied to normal RAM.

```
interAptivUP_BOSTON_RAM:
   ${MAKE} -C interAptivUP BOSTON_RAM
```

interAptivUP_BOSTON_SPRAM

Builds a BOSTON Board version with the main function copied to Scratchpad RAM.

```
interAptivUP_BOSTON_SPRAM:
   ${MAKE} -C interAptivUP BOSTON_SPRAM
```

clean_interAptivUP

Cleans all object files in common areas and all built files for the interAptivUP.

```
clean_interAptivUP:
   ${MAKE} clean -C interAptivUP
```

5.2. Core Level Makefile

Each Core has a Makefile customized for the code elements needed for it. The code and build details differ for each core. Each source file is covered in ‘Code Details’ on page 28. The interAptivUP core Makefile will be used as an example in the following description of the different sections of the Makefile.

5.2.1. Defines for common utilities

At the top of the Makefile are defines for common utilities. If you are using a different toolchain, you may have to change these defines to correspond to your tool chain’s names. “CC” is set to the name of the C compiler, “LD” is set to the name of the Linker, “OD” is set to the object dump utility used to produce a disassembly of the code, and “OC” is set to the name of the object copy utility, which is used for a BOSTON Board build to convert the elf file to an S-record file needed to download to the Board’s Flash memory. SZ will print out the size of the executable at the end of the make process.

```
// interAptivUP Makefile


CC=mips-$(VENDOR)-elf-gcc
LD=mips-$(VENDOR)-elf-ld
OD=mips-$(VENDOR)-elf-objdump
OC=mips-$(VENDOR)-elf-objcopy
SZ=mips-$(GCC_VENDOR)-elf-size
```

`$(VENDOR)` is set to ‘mti’ in the initial Makefile, in the root directory.

5.2.2. Defines for directory paths

The next defines are used to find directories for the source and object files. BASE is the path to the top-level directory of the project. COMMON is the path to the common source and object files. BOSTON is the path to the files that are specific to the BOSTON Board. Other make files will have additional defines for the CPS, which is the path to the source and object files specific to the Coherent Processing System, and for MT, which is the path to the Multi-threaded source and object files.

```
BASE=../
COMMON=$(BASE)common
BOSTON=$(BASE)BOSTON
```

5.2.3. Compiler and Linker arguments

Next are the defines used as arguments to the build commands:

- `CFLAGS=-O3 -g -EL -c -I $(COMMON) –mmt -march=mips32r3`

  CFLAGS are the arguments to the C command line. For this example these arguments are:

  - –O3 this is the optimization level. O3 is the highest level of optimization. This causes problems when using the source-level debugger because of the nature of the optimizations. It will cause the debugger to look like it is repeating lines of code and

jumping forward and backward in the code as you step through it. If you find this hard to follow, you can change or remove the –O argument. This will cause the compiler to optimize less or not at all, but it will make debugging easier. Once you have debugged the code, you can change it back to –O3 for the production build.

- `-g` is used to produce debugger information that is needed if you want to debug with a source-level debugger. This may be removed for the final production build.
- –EL causes the code to be built for Little Endian. If you want to build for Big Endian, then change this to –EB.
- `-I` tells the Makefile where to find the include files (other than the system include files). Here it points to the common directory that contains the boot.h file
- –mmt tells the compiler to use MT instructions. This should only be present for multi-threaded cores.
- **-march**=mips32r3 tells the compiler to compile the code as MIPS32 release 3.

- `LDFLAGS_SIM_RAM=-T sim_Ram.ld -EL -nostdlib -Wl,-Map=sim_Ram_map`
- `LDFLAGS_SIM_SPRAM=-T sim_SPRam.ld -EL -nostdlib -Wl,-Map=sim_SPRam_map`
- `LDFLAGS_BOSTON_RAM=-T BOSTON_Ram.ld -EL -nostdlib -Wl,-Map=BOSTON_Ram_map`
- `LDFLAGS_BOSTON_SPRAM=-T BOSTON_SPRam.ld -EL -nostdlib -Wl,-Map=BOSTON_SPRam_map`

Note: Only for the interaptive core: it can be compiled for EVA boot mode

- **`LDFLAGS_BOSTON_EVA_RAM=-T`** `BOSTON_Ram_eva.ld` **`-EL -nostdlib -Wl,`**`-Map=BOSTON_Ram_eva_map`

  There are several LDFLAG defines, one for each type of build:

  - –T is used to pass the name of the linker script file to the linker. There will be more information on the linker script in the next section.
  - –EL links for Little Endian. To link for Big Endian, change this to –EB.
  - –nostdlib tells the linker to not use standard libraries. The boot code does not support standard library calls (like printf). By using the –nostdlib option, the linker will report an error if a standard library call is made.
  - -Wl,-Map=<MAP file Name> this option tells the linker to produce a Map file with the given name. The map file is useful in determining to which addresses the linker has linked the code and data.

5.2.4. Source file lists

There are several source file lists that correspond to different builds:

ASOURCES

A list of the assembly files common to all targets in this Makefile. The list differs from core to core depending on the source needed to boot that particular core.

BOSTONSOURCES

A list of assembly files that are specific to a BOSTON Board build.

ASOURCES_SP

Used to combine common sources with a specific source to build for the Scratchpad RAM version.

ASOURCES_RAM

Used to combine common sources with a specific source to build for the non-Scratchpad RAM version.

CSOURCES

A list of C source files in the build.

5.2.5. Object file lists

The object file lists are built using built-in make rules that convert the source file lists into object file lists. There is a corresponding OBJECT file define for each source file list.

- `BOSTONOBJECTS=$(BOSTONSOURCES:.S=.o)`
- `COBJECTS=$(CSOURCES:.c=.o)`
- `AOBJECTS=$(ASOURCES:.S=.o)`
- `AOBJECTS_SP=$(ASOURCES_SP:.S=.o)`
- `AOBJECTS_RAM=$(ASOURCES_RAM:.S=.o)`

The object file lists will be used in the different target builds.

5.2.6. Adding to CFLAGS for BOSTON Board Builds

In order to have a more generic start.S file, the code contains a #define for the Denali memory controller present on BOSTON boards. For BOSTON Board target builds, the define “BOSTON_RAM” needs to be added to the CFLAGS.

```
ifeq ($(findstring BOSTON_, $(MAKECMDGOALS)), BOSTON_)
CFLAGS += -DBOSTON_RAM
endif
```

To do this, the Makefile built-in command “findstring” is used to search the target name passed to the make command (`MAKECMDGOALS`) for BOSTON_. If it finds it, it adds –DDENALI to the CFLAGS define.

For EVA boot mode a –DEVA is added in a similar way by:

```
ifeq ($(findstring SIM_RAM_EVA, $(MAKECMDGOALS)), SIM_RAM_EVA)
CFLAGS += -DEVA
endif
```

5.2.7. Make Targets

As discussed previously, there are four make targets for each core.

BOSTON_SPRAM

This target builds for a BOSTON Board using Scratchpad RAM:

```
BOSTON_SPRAM : $(COBJECTS) $(AOBJECTS_SP) $(BOSTONOBJECTS)


   $(OC) BOSTON_SPRam.elf -O srec BOSTON_SPRam.rec
   perl $(COMMON)/srecconv.pl -ES L BOSTON_SPRam
```

This target depends on the common C objects (COBJECTS), the Scratchpad-specific Assembly objects (AOBJECTS_SP), and the BOSTON Board-specific objects (BOSTONOBJECTS):

```
   $(CC)  $(LDFLAGS_BOSTON_SPRAM) $(COBJECTS) $(AOBJECTS_SP)\
         $(BOSTONOBJECTS) -o BOSTON_SPRam.elf
```

The CC rule line will build the BOSTON_SPRam.elf executable file using the object lists and Linker flags appropriate to the BOSTON and Scratchpad build.

```
   $(OD) -d -S -l BOSTON_SPRam.elf > BOSTON_SPRam_dasm
```

The OD rule will produce a disassembly file.

```
   $(OC) BOSTON_SPRam.elf -O srec BOSTON_SPRam.rec
   $(SREC) -output interAptiv_SP_MIPS_BOOT.mcs -intel boston_SPRam.rec
   -offset -0x18000000 -disable=exec-start-address
```

The last two lines use object copy and a perl script to convert the elf file into a flashable file called interAptiv_SP_MIPS_BOOT.mcs.

BOSTON_RAM

This rule differs from the previous one by using object lists, a linker file script, and output names to produce a BOSTON Board RAM version of the flash file (interAptiv_MIPS_BOOT.mcs).

```
BOSTON_RAM : $(COBJECTS) $(AOBJECTS_RAM) $(BOSTONOBJECTS)
   $(CC)  $(LDFLAGS_BOSTON_RAM) $(COBJECTS) $(AOBJECTS_RAM)
   $(BOSTONOBJECTS) -o boston_Ram.elf
   $(SZ) boston_Ram.elf
   $(OD) -d -S -l boston_Ram.elf > boston_Ram_dasm
   $(OC) boston_Ram.elf -O srec boston_Ram.rec
   $(SREC) -output interAptiv_MIPS_BOOT.mcs -intel boston_Ram.rec          -
offset -0x18000000 -disable=exec-start-address
```

This rule differs from the previous one by using object lists, a linker file script, and output names to produce a BOSTON Board RAM EVA version of the flash file (interAptiv_EVA_MIPS_BOOT.mcs).

```
BOSTON_RAM_EVA : $(COBJECTS) $(AOBJECTS_RAM) $(BOSTONOBJECTS)
   $(CC) $(LDFLAGS_BOSTON_EVA_RAM) $(COBJECTS) $(AOBJECTS_RAM)
   $(BOSTONOBJECTS) -o boston_Ram_eva.elf
```

```
   $(SZ) boston_Ram_eva.elf
   $(OD) -d -S -l boston_Ram_eva.elf > boston_Ram_eva_dasm
   $(OC) boston_Ram_eva.elf -O srec boston_Ram_eva.rec
   $(SREC) -output interAptiv_EVA_MIPS_BOOT.mcs -intel
   boston_Ram_eva.rec -offset -0x18000000
         -disable=exec-start-address SIM_RAM
```

This rule will produce an elf file which is suitable to be used with a simulator with normal RAM.

```
SIM_RAM : $(COBJECTS) $(AOBJECTS_RAM)
```

The rule depends on the C objects (COBJECTS) and the Assembly Objects for RAM (AOBJECTS_RAM).

```
   $(CC)  $(LDFLAGS_SIM_RAM) $(COBJECTS) $(AOBJECTS_RAM) -o sim_Ram.elf
```

The CC rule uses the linker script and object file list appropriate to build a Simulator RAM version of the elf file sim_Ram.elf.

```
   $(OD) -d -S -l sim_Ram.elf > sim_Ram_dasm
```

The OD rule produces a disassembly file from the elf file.

SIM_SPRAM

This rule differs from the SIM_RAM rule by using objects lists, a linker file script, and output name to produce a Simulator Scratchpad version of the elf file sim_SPRam.elf.

5.2.8. C and Assembly rules

These rules will build the objects file needed for the CC rule from the dependency list for the target.

```
.c.o:
   $(CC) $(CFLAGS) $< -o $@
```

The .c.o rule takes an object name from the provided object list and compiles it from the same-named file ending in .c instead of .o.

```
.S.o:
   $(CC) $(CFLAGS) $< -o $@
```

The .S.o rule takes an object name from the provided object list and assembles it from the same- named file ending in .S instead of .o.

5.2.9. Clean rule

The clean rule removes all traces of files produced from all target builds. It does this by using the shell commands listed in the clean rule.

6. Linker scripts

The linker scripts are used by the linker to locate the code properly during the link step. They also provide symbols that are used in the boot code to perform the copy from flash memory to RAM or SPRAM. There are four linker scripts per core, one for each make target.

All linker scripts use 0x9fc0 0000 (except for M5100 which uses 0xBfc0 0000 because there is not caches) as the starting address of the boot code. This is a KSEG0 address that mirrors the KSEG1 boot exception vector address, 0xbfc0 0000, in all MIPS systems, i.e., the memory location of the first instruction that will be fetched. The difference between using a KSEG0 address and a KSEG1 address is that KSEG1 is a non-cached memory region whereas KSEG0 is a cacheable region that first starts out as uncached and can then be switched to cacheable. (The switch is done after the I-cache has been initialized in start.S.)

6.1. BOSTON_Ram.ld

This linker script is used in the BOSTON_RAM target builds for a BOSTON Board with a copy of the main code from flash to normal RAM.

```
_monitor_flash = 0xbe000000 ;
    .text_init 0x9fc00000 :
AT( _monitor_flash )
```

The script tells the linker to create a symbol called _monitor_flash with a value of the address of the monitor flash on the BOSTON board, which is the starting address of the BOSTON Board's flash. The BOSTON Board is setup to alias the normal boot vector, 0xbfc0 0000, to this address.

The .text_init 0x9fc0 0000 : gives the linker the starting address for linking the object files that follow.

The "AT" command directs the linker to load the code at the _monitor_flash address. Thus the code will be linked to execute starting at the boot vector, but will be loaded into the flash at the _monitor_flash address, which on the BOSTON Board also appears to the VPE as the boot exception vector 0xbfc0 0000.

The next part of the linker script is a list of object files that will be linked into the .text_init section. Here is an example of the list for an interAptivUP core:

```
    {
    _ftext_init = ABSOLUTE(.) ;  /* Start of init code. */
    start.o(.text)              /* Reset entry point                */
    ../common/init_gpr.o(.text)
    set_gpr_boot_values.o(.text)
    ../common/init_cp0.o(.text)
    ../common/init_tlb.o(.text)
    ../Boston/init_FPGA_mem.o(.text)
    ../common/init_caches.o(.text)
    ../common/copy_c2_ram.o(.text)
    ../common/init_itc.o(.text)
    ../mt/init_vpe1.o(.text)
    ../Boston/boston_lcd.o(.text)
    . = ALIGN(8);
```

```
    _etext_init = ABSOLUTE(.);    /* End of init code. */
  } = 0
```

This list is the main reason there are different sets of linker scripts for each Core, i.e., because each Core can have a different set of object files. The list for the interAptivUP is the smallest subset of objects files. The other Cores will have a superset of object files, depending on the code that needs to be included to boot that Core.

There are two symbols, _ftext_init and _etext_init, that are set in accordance with the list of object files. These are seen in the section above in the beginning and end of the list:

```
    _ftext_init = ABSOLUTE(.) ;    /* Start of init code. */
```

_ftext_init is a symbol that is set to the current link location. In this case, that location is 0x9fc0 0000, because the symbol is at the start of the text_init section (0x9fc0 0000).

```
    _etext_init = ABSOLUTE(.);    /* End of init code. */
```

_etext_init is the symbol set to the last link location of the text_init section.

The next part of the script will be used to link and load any object files not in the above list. This happens to be the main.o file for this boot code. The code in main.o will be copied from flash to some type of RAM, so it will be loaded at a flash location but linked for a RAM location. Along the way it will create symbols that will be used by the copy code to copy the code from flash to RAM.

```
    _zap1 = _etext_init - _ftext_init + _monitor_flash;
    start_rom_text = _etext_init - _ftext_init + 0xbfc00000;
```

The `_start_rom_text` symbol will be computed at link time. This is the address where main.o code will be loaded into the flash and be used by the copy code as a starting address for the source of the copy to RAM. It is computed by taking the starting address of the BEV and adding the difference between the start of the text_init section (_ftext_init) and the ending address of the text_init section (_etext_init). In other words, the _start_rom_text symbol is computed as the starting address of the flash plus the size of the text_init section.

```
    .text_ram 0x80100000 :
    AT( _zap1 )
```

The .text_ram line tells the linker to link the text_ram section to the 0x8010 0000 RAM address. (NOTE: For a EVA boot the text_ram address will be 0x00100000 due to the EVA memory mapping.) The AT line tells the linker to load it into memory at the address in the symbol _zap1.

```
  {
   _ftext_ram = ABSOLUTE(.) ;    /* Start of code and read-only data */
   *(.text)*(.text.*)
   . = ALIGN(8);
   _etext_ram = ABSOLUTE(.);     /* End of code and read-only data   */
   } = 0
```

The above line tells the linker what goes into the text_ram section. First it sets the _ftext_ram symbol to point to the current link address, which in this case is the start of the text_ram section. The code will use this address as the starting destination address for the copy.

Next the *(.text) line tells the linker what to put into this section. The .text sections from any object files on the linker command line (in the Makefile) that are not specifically named will go into the .text_ram output section.

In the above code, the . = ALIGN(8); aligns the end of the section to an 8-byte boundary.

The _etext_ram symbol is set to the end of the text_ram section.

The next section covers the initialized data section. It contains external variables that are initialized in the code.

```
  _zap2 = _etext_ram - _ftext_ram + _zap1 ;
```

The _zap2 symbol is computed to contain the load address of the next section (data). It is computed by taking the end load address of the text_init, _zap1 and adding the difference between the first address of the text_ram section, _ftext_ram, and the ending address, _etext_ram. In other words, _zap2 equals the ending load address of the first section text_init and the size of the next section text_ram.

```
  .data _etext_ram :
  AT( _zap2 )
```

Next the .data line tells the linker where to link the .data section. Here it is set to _etext_ram, which is the ending link address of the last section, text_ram.

The AT line tells the linker where to load the data section. This is going to be an address in RAM.

The next part describes what's in the data section.

```
 {
  _fdata_ram = ABSOLUTE(.);      /* Start of initialised data        */
  *(.rodata)
  *(.rodata.*)
  *(.data)
  . = ALIGN(8);
  _gp = ABSOLUTE(. + 0x7ff0); /* Base of small data                  */
  *(.lit8)
  *(.lit4)
  *(.sdata)
  . = ALIGN(8);
  _edata_ram  = ABSOLUTE(.);     /* End of initialised data          */
 }
```

Once again the symbols for the beginning and end link points for the section are set up (_fdata_ram and _edata_ram).

In between these symbols is the list of subsections that go into the data section. It also sets up the Global Pointer symbol. This data area is 64K, with the _gp symbol pointing to the middle of it, so address offsets will be no larger than 16 bits plus or minus the Global pointer. This fits with the number of bits allowed for the offset field of instructions like sw. The _gp will be written to GPR $28, gp, before the code in main() is called.

Next are the uninitialized variable sections, sbss and bss.

```
  _fbss = .;
  .sbss :
  {
      *(.sbss)
```

```
    *(.scommon)
}
.bss :
{
    *(.bss)
    *(COMMON)
}_end = . ;
```

What's important here is that _fbss contains the starting link address (in RAM) for the bss section, and _end contains the ending address of the bss section. These will be used in the copy code to zero out the uninitialized variables to comply with the C standard.

6.2. BOSTON_SPRam.ld

The BOSTON_SPRam.ld linker script is almost the same as the BOSTON_Ram.ld described above. There are two differences:

- In the object file list for the text_init section, the copy_c2_ram.o has been swapped out for copy_c2_SPram.o.
- There are additional symbols used to set up the Scratchpad RAMs and to aid in the copy to Scratchpad RAM.

6.2.1. Linking for Scratchpad RAM

The Scratchpad RAM example boot code is an example of a system that has Instruction and Data Scratchpad RAM, no normal RAM, and uses Fixed Mapping Translation (FMT). The boot code will program the Scratchpad RAM controller with the physical address of the ISPRAM and DSPRAM memory regions. Then it will copy the code in main.c into the ISPRAM, the initialized data into the DSPRAM, and clear the uninitialized variables.

The linker script controls where in memory the Scratchpads are placed and links the code and data for those addresses. It does this by defining and computing additional symbols used in the script. Below are the additional symbols and how they are used.

```
_FMT_offset = 0x40000000 ;
```

The symbol _FMT_offset is the translation from virtual address 0 to an address in physical memory using Fixed Mapping Translation (FMT). The value, 0x4000 0000 is defined by the MIPS Architecture and cannot be changed.

```
_ISPram = 0x50000000 ;
_DSPram = 0x60000000 ;
```

The symbols _ISPram and _DSPram are the physical address where the Scratchpad RAM will be located in the system. They will be used by the code in the `common/copy_c2_SPram.S` to position the Scratchpad RAMs in the physical memory map. These are not fixed addresses. They can be changed, usually to a physical address in the KUSEG region.

```
_code_start = _ISPram - _FMT_offset ;
_data_start = _DSPram - _FMT_offset ;
```

The symbols _code_start and _data_start will be used as the starting link addresses for the code and data in main.c. These are computed by using the difference between the start of the SPRam and the _FMT_offset. The difference in these 2 physical addresses translates into a virtual address in KUSEG.

```
.text_ram _code_start :
.data _data_start :
```

The address for the SPRAM computed above is used as the link address for the text_ram and data sections.

6.3. sim_Ram.ld and sim_SPRam

The only differences in these two linker scripts from their BOSTON counterparts is the value of the _monitor_flash symbol. The BOSTON board has flash that starts at 0xbe0 0000, which is aliased to the boot exception vector at 0xbfc0 0000. There is no aliasing done in the simulators, so _monitor_flash has the value of the boot exception vector 0xbfc0 0000.

7. Downloading to the Boston Boot Flash

Our larger designs such as the P6600 and I6400/I6500 require larger FPGAs. The Boston Evaluation platform has a larger FPGA which accommodates these cores. In addition to the larger FPGA the Boston board incorporates a Linux PC to make it easier to work with. Below is a typical setup for a Boston board.

This section will demonstrate how to flash the boot code into the Boston board.

- Login as: user
- Password is: user
- cd /home/user/Boston
- Copy your mcs file from the Boot-MIPS build to an accessible directory in the files system
- Cnter the command:

```
boston flash <path to your.mcs file>
```

8. Debugging using Codescape Debugger

8.1. Debugging Multi-threaded and Multi-core systems

When debugging Multi-threaded or Multi-core systems, you need to be aware of when a thread or additional core is ready to run before you can initiate a debug session for it. Always start the first core for a multi-core system and the first core's first VPE in an MT multi-core system. Then you need to pick a spot in your code when additional cores or VPEs are at the point where they are ready to run so you can start them.

This is an example of an interAptiv 4 Core 2 VPE/core in a Concord daughter card with a BOSTON base board. The Boot-MIPS executable code has been flashed into the BOSTON boards flash.

The process explained in this section has the following steps:

1. Connect the SP55
2. Start the debugger and connect to the target
3. Set Breakpoint mechanism to Hardware
4. Restart and stop processors
5. Load debug symbols
6. Locating source code in Disassembly
7. Starting additional cores

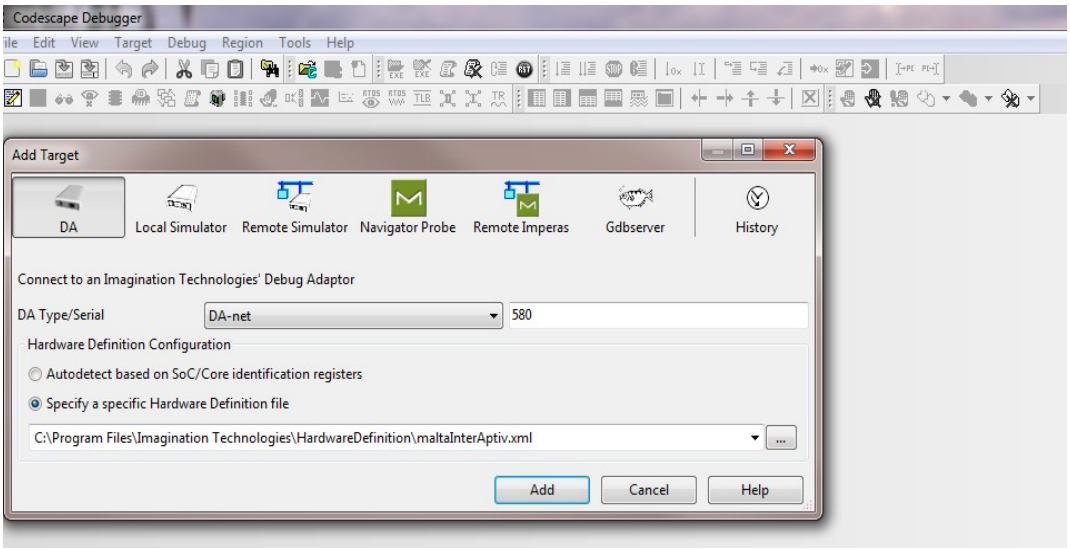
Connect the SP55 to the BOSTON board

To start this example, connect the SP55 to the BOSTON board, Ethernet and power. Then power up the BOSTON board.

Start Codescape Debugger and connect to the target

1. Start Codescape Debugger.

17. Right-click in the Target Pane and select ‘Add Target’.

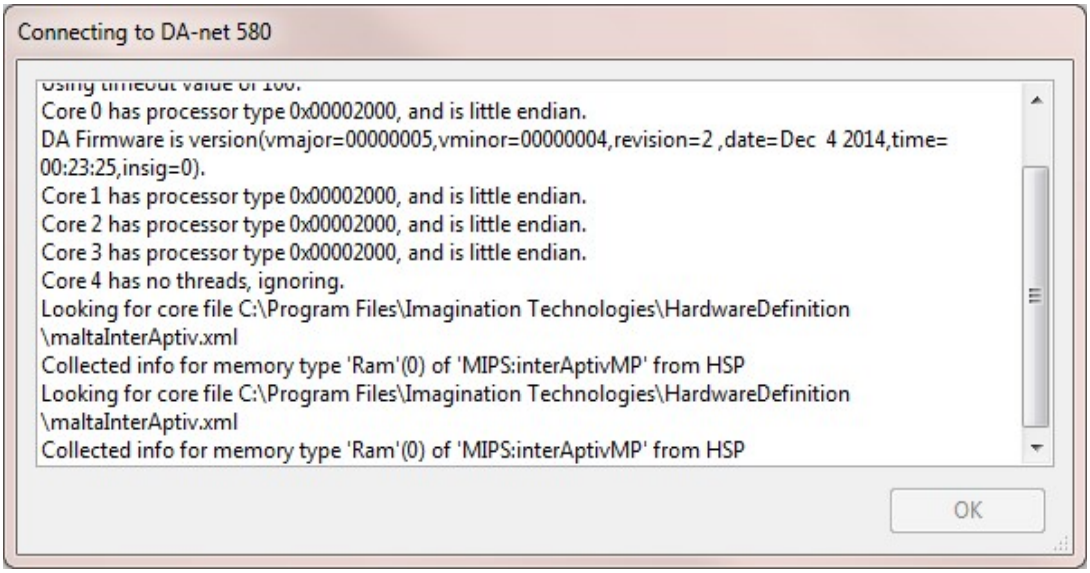


18. Select SP55 and enter the last 3 digits of the SP55 serial number.

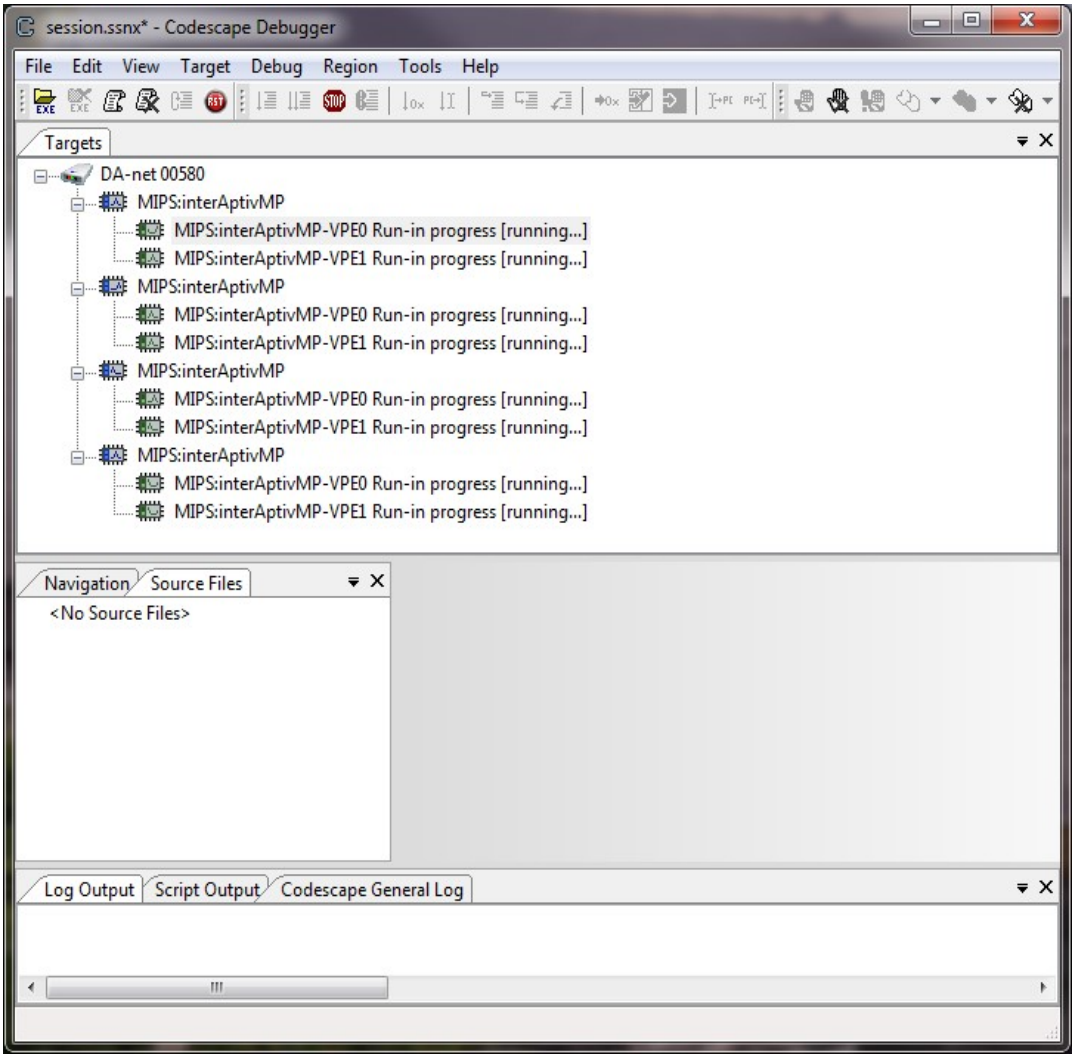
    For this example a Hardware definition was created specifically for a BOSTON board so “Specify a Specific Hardware Definition file” has been selected and the full path name to that Hardware Definition file.

    Creating a Hardware Definition File for your board makes it easier to step and set breakpoints in the code because the Debugger will know what areas of memory are in flash and read-only. The debugger will use hardware breakpoints in those sections of memory. Once everything is setup select “Add”.

    A popup will appear to show the progress of the connection:



    Then the target processors will be displayed:



As you can see each core is designated as MIPS:interAptivMP. Each Virtual Processor is shown under the tree structure for each core and shows the current status, which in this case is “running”.

Set breakpoint mechanism to Hardware only

You must use hardware breakpoints when debugging Boot MIPs. Codescape Debugger defaults to using software breakpoints where possible.

If you have created a Hardware definition file for your board and have configured the Flash or ROM area that the boot code resides in as read-only then you can just set a breakpoint and the debugger will use a hardware breakpoint. If you have not created a Hardware Definition file then you will need to change the Target Debug Options as shown below:

1. Right-click on the SP55 in the Target pane and select ‘Target Debug Options’

19. Select the following options, as shown in the example below:

For user breakpoints use

‘just the hardware breakpoints’

For multiple instances of inline code use

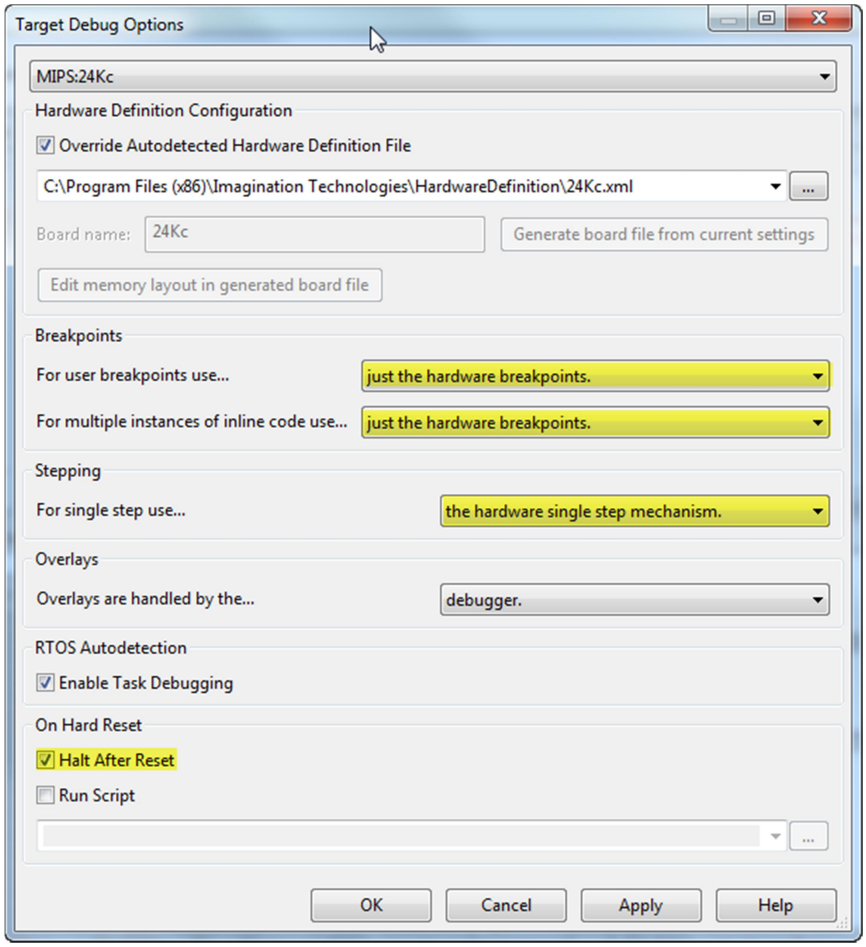
'just the hardware breakpoints'

For single step use

'the hardware breakpoint single step mechanism'

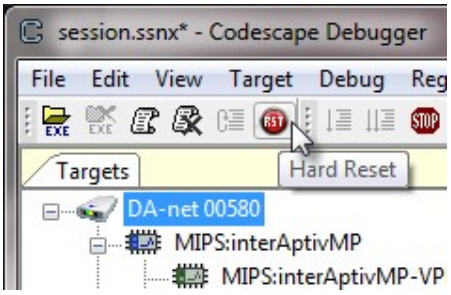
On Hard Reset

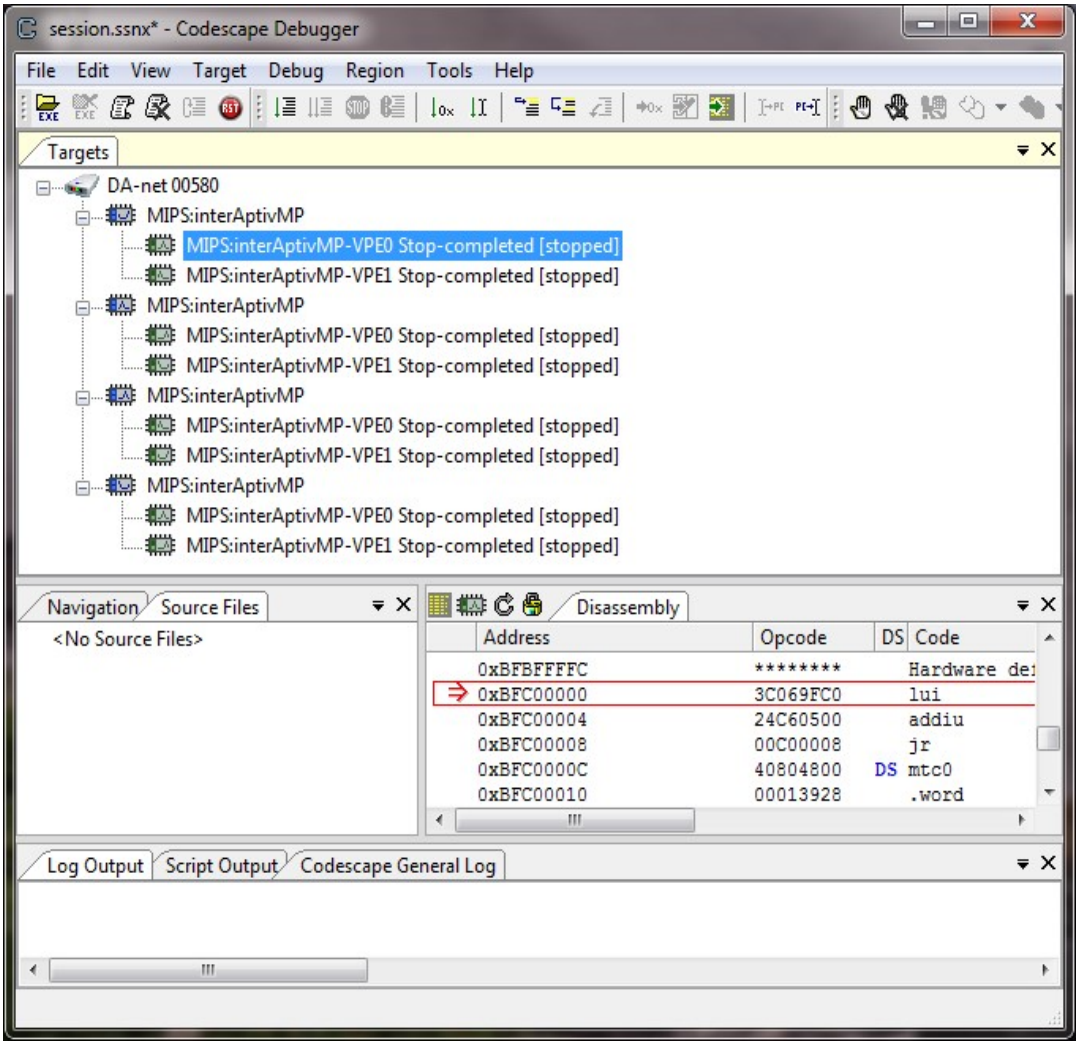
'Halt After Reset'



Restart and stop processors

- Selecting the top element of the tree, SP55, then clicking on the RST icon on the tool bar will reset and stop all processors:





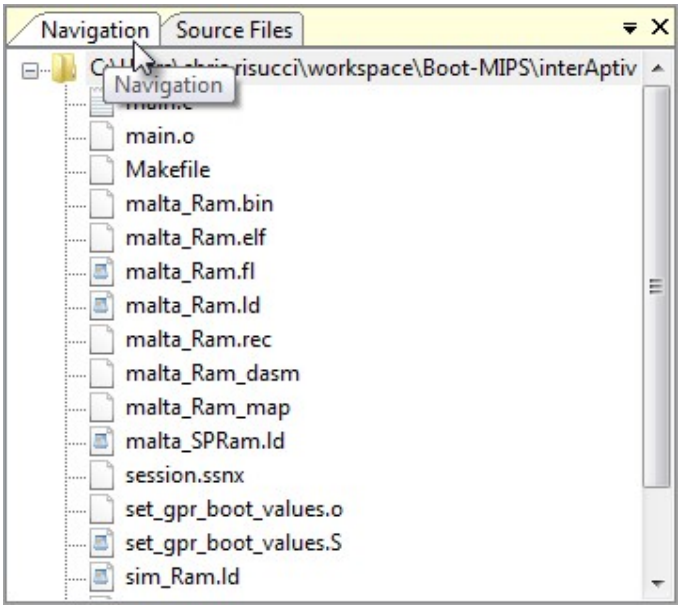
Now all processors have been stopped.

Note: This is the default action for a hard reset. It can be changed for each target from the Target Debug Options dialog (right-click on the target > Target Debug Options).

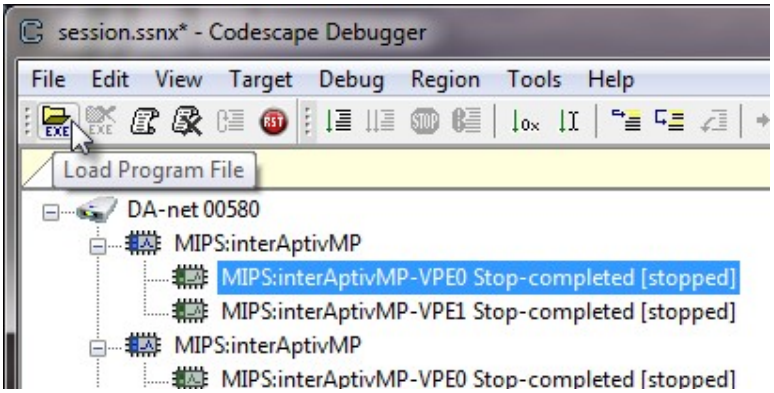
Load debug symbols

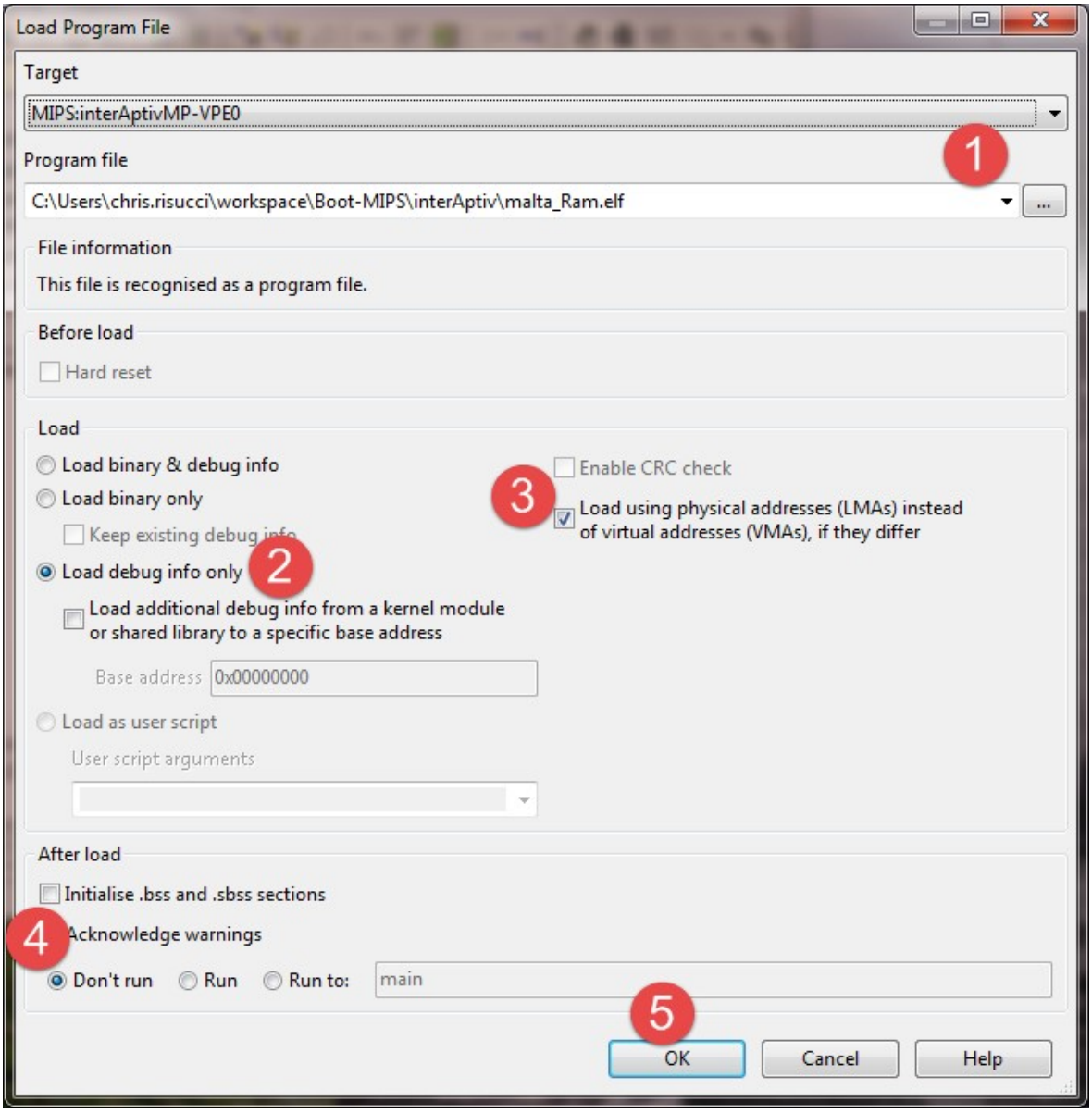
The boot flash has already been downloaded to the flash so there is no need to download the executable but the debugger still needs to know what the symbols are and what files they are located in so it can display source for you.

1. It is helpful if the Navigation pane is displaying the Boot-MIPS interAptiv directory. This is done by selecting the “Navigation” tab and navigating to the directory:



20. To load debug symbols select a processor and then click on the “EXE” icon:

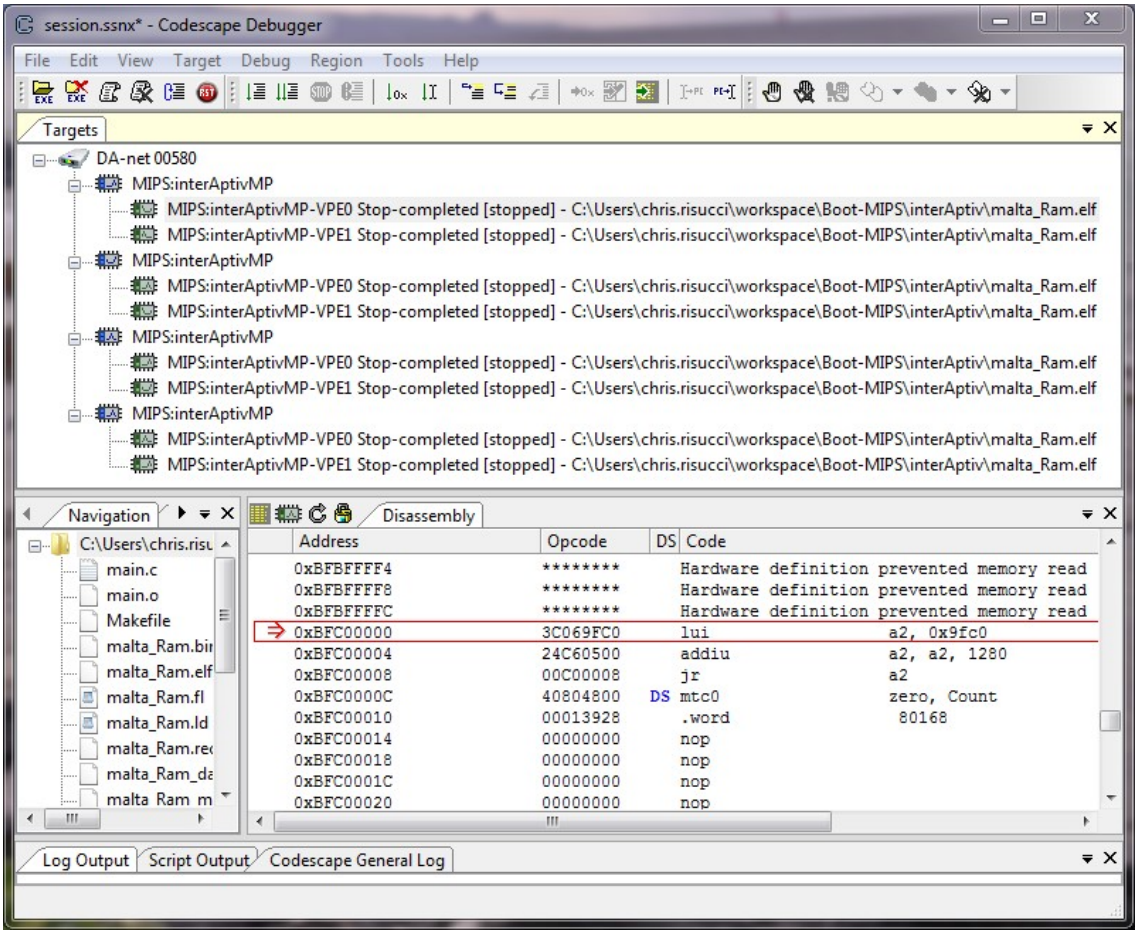




1. Set the “Program File” to the interAptiv ELF that was created in the Make step.
2. Select “Load debug info only” because we don’t need to load the executable.
3. Check “Load using physical address ….” Because the debugger needs to know we are relocating code that was in the flash to RAM.
4. Select “Don’t run” because we want to add the debug symbols for each Virtual Processor.
5. Then click on “OK”

This process needs to be repeated for each processor.

Once finished the “Targets” region should look like this:

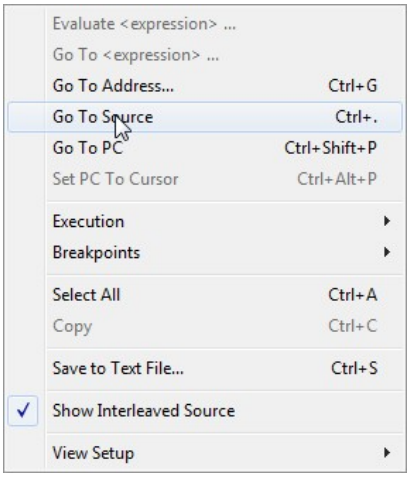


Locating source code in Disassembly

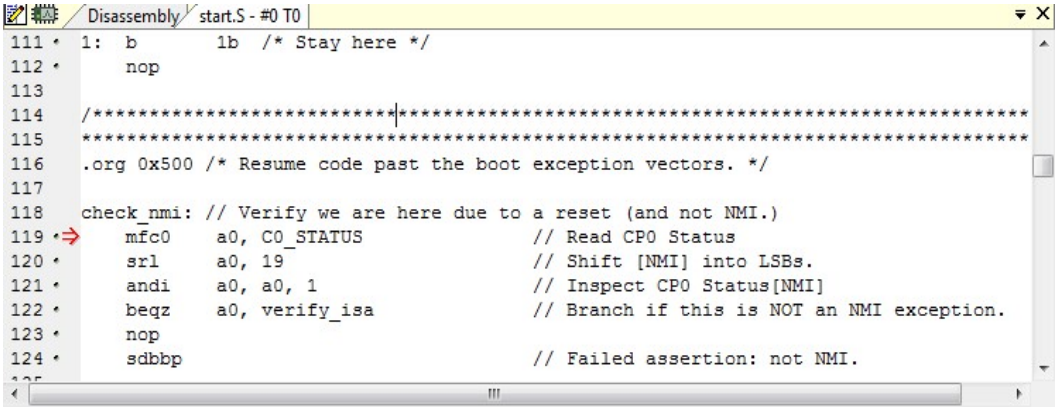
The program counter (indicated by the red arrow) is at the boot vector start address of 0xBFC00000. However the source code is not shown because it has been linked for the KSEG0 address 0x9FC00000. This address mirrors the boot vector at 0xbfc0 0000. Stepping through to the code in 0xBFC00008 will execute a jump to 0x9FC00000 and source will be displayed.

At the start of the boot code, you will only see the code in the assembler view. This is because the code has been linked for the KSEG0 address 0x9fc0 0000. This address mirrors the boot vector at 0xbfc0 0000. However, the debugger has no reference to this address to correlate with the linked source code, so you will only see dissemble in the source window. You will need to step through the next few instructions in instruction single-step mode before you will see source code in the source window.

1. Step about 3 instructions (press F7 to single-step) then right-click in the “Dissemble” region and select “Go To Source”.



You should now see the Boot-MIPS source code:

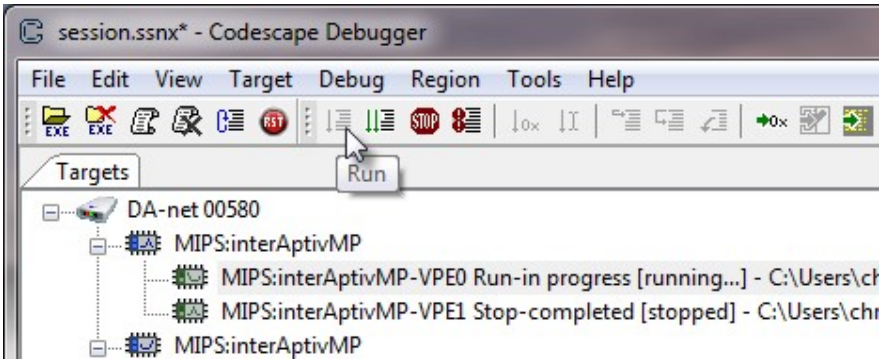


At this point you can set breakpoints or just step to the next line of the source code.

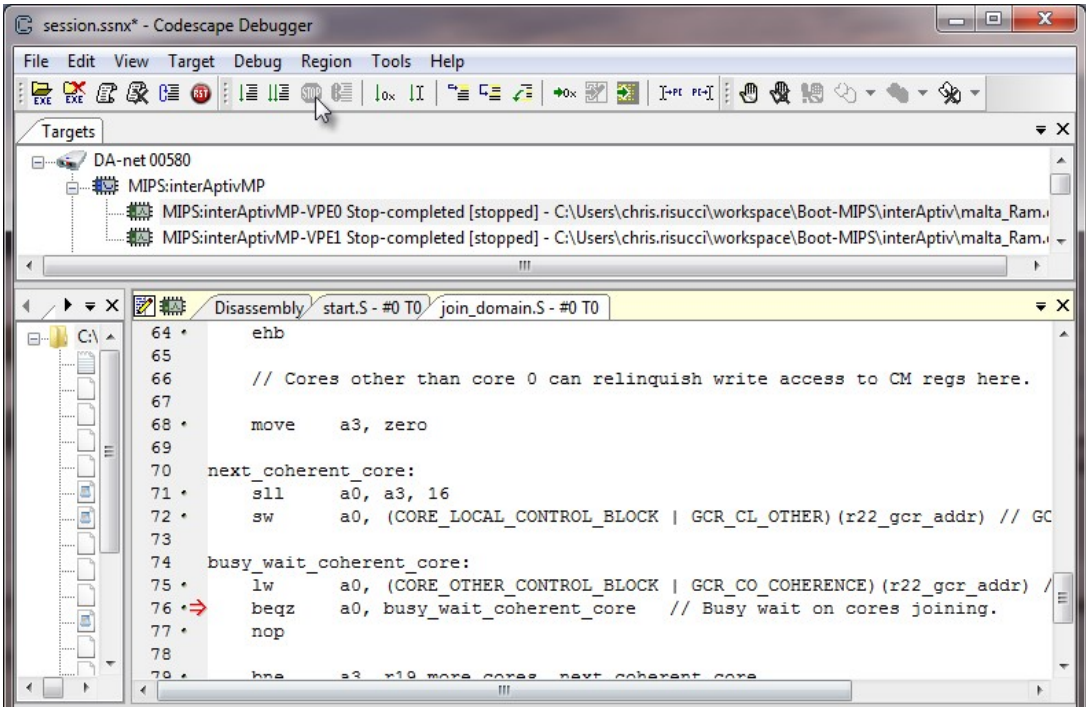
Starting additional cores

Before you can start the next core, you need to get to the point in the code where the first core is waiting for additional cores to join the CPS Domain. This point is in the join_domain function in the join_domain.S file. You can debug to this point or you can just click on the run button.

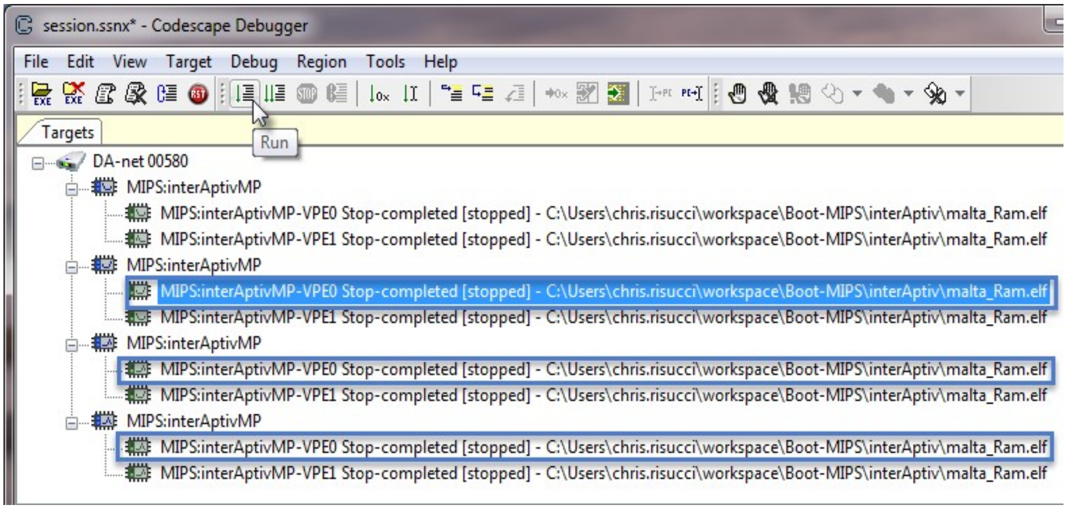
1. Run a specific core by selecting it and clicking on the Run button



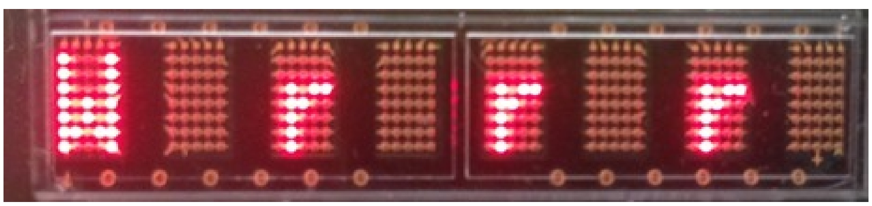
21. Then after a short time click on the stop button. You should now be in the wait loop in the join_domain function:


session.ssnx* - Codescape Debugger
File Edit View Target Debug Region Tools Help
Targets
DA-net 00580
MIPS:interAptivMP
MIPS:interAptivMP-VPE0 Stop-completed [stopped] - C:\Users\chris.risucci\workspace\Boot-MIPS\interAptiv\malta_Ram.
MIPS:interAptivMP-VPE1 Stop-completed [stopped] - C:\Users\chris.risucci\workspace\Boot-MIPS\interAptiv\malta_Ram.
Disassembly
start.S - #0 T0
join_domain.S - #0 T0
C:\
64 ehb
65
66 // Cores other than core 0 can relinquish write access to CM regs here.
67
68 move a3, zero
69
70 next_coherent_core:
71 sll a0, a3, 16
72 sw a0, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_OTHER)(r22_gcr_addr) // GC
73
74 busy_wait_coherent_core:
75 lw a0, (CORE_OTHER_CONTROL_BLOCK | GCR_CO_COHERENCE)(r22_gcr_addr) /
76 beqz a0, busy_wait_coherent_core // Busy wait on cores joining.
77 nop
78

22. Now you can start all other core’s VPE0, (not the second VPEs yet). For each core select VPE0 then click on run.



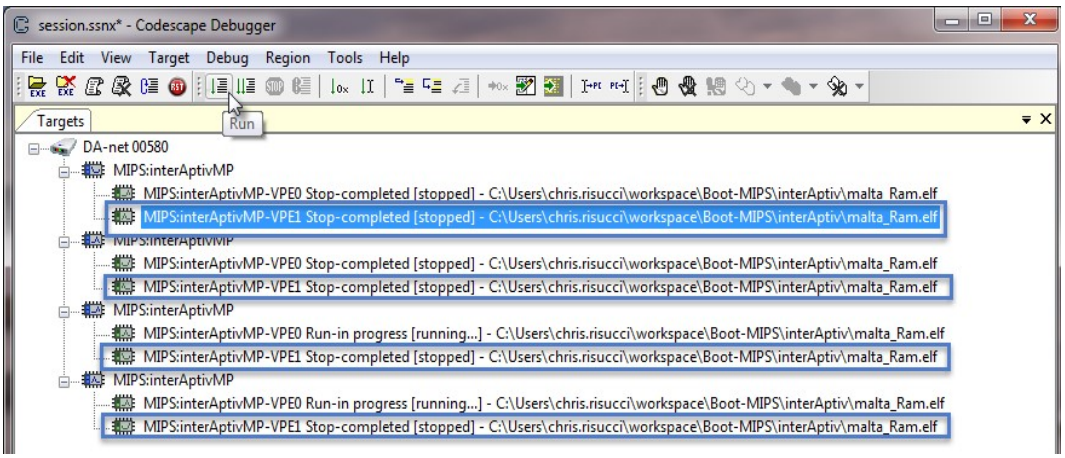
23. You can debug each core by selecting its thread in the “Debug” pane. After you have finished debugging each core, select Run All (Ctrl-F9). The display on the BOSTON Board will show Core 0 Waiting and cores 1, 2 and 3 are Ready:



24. If you stop Core 1, 2 or 3 at this point, you will see they are in main().

```
121             MALTA_CHAR(cpu_num) = 'r' ;     // display 'r' for ready.
122             ready[cpu_num] = 1 ;
123             asm volatile ("di") ;
124
125             while (!start_test[cpu_num]) {
126                 // This code will only work reliably if the WII bit is set in config7.
127                 // When this bit is set any interrupt even when they are disabled will c
128                 // wait to return. This avoids a race condition
129                 asm volatile ("wait") ;     // Wait for interrupt (qualified with "start
130                 // enable interrupts so interrupt routine can run and set the start_test
131                 asm volatile ("ei") ;
132                 asm volatile ("ehb") ;
133                 // Disable interrupts again so there is not race condition between testi
134                 // start_test bit variable and going back to wait.
135                 // NOTE for this code we are only expecting IPI that interrupt.
136                 asm volatile ("di") ;
137                 asm volatile ("ehb") ;
```

25. At this point, run all core's VPE1:



You will also notice that the BOSTON display has changed:

The display should continue to blink the processor numbers indefinitely.

9. Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

Revision	Date	Description
01.00	March 3, 2012	Initial release
01.01	July 16, 2012	Added support for interAptiv and proAptiv. Improved debug session bring up, reflecting code changes done for readability.
01.02 01.03 01.04 01.07	February 1, 2013 August 19, 2013 June 1, 2014 July 9, 2015	Minor fixes Added interAptiv UP Update for current cores Update I6400 and interAptiv EVA
01.09	April 8, 2016	added P6600 and I6400
01.15	July 2017	added I6500
01.20	April 2018	Updated code to current source