MD00152 Boot-MIPS

Example Boot Code for MIPS I7200 Cores

Filename	:	
Version	:	1.70
Issue Date	:	15 Dec 2017
Author	:	MIPS Tech

Contents

1. Introduction

Boot-MIPS is example code for nanoMIPS32® I7200 Cores. It is intended to aid you in becoming familiar with the initialization of a MIPS Core.

Building Boot-MIPS results in executables suitable for programming the flash on a BOSTON software development board, or to download to an IASim simulator. In addition to runtime initialization, Boot-MIPS executables include 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.

- I7200: 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).
- CPC (Cluster Power Controller): Power domain control logic for a CPS.
- CCA (cache coherency attribute): describes the cacheability of an address segment.
- TLB (Translation Lookaside Buffer) Translates Virtual (program) address to Physical address using a cache of translation mappings.
- FMT (Fixed Mapped Translations) Translates Virtual (program) address to Physical address using static fixed mappings.
- MPU (Memory Protection Unit): alternative to traditional TLB and FMT program address equal physical address. Memory is segmented into fixed segments that can be configured with CCA and permission attributes. Parts of a fixed segment can be overlaid with smaller segments the can have overriding attributes to the fixed segment.
- EVA: Enhanced Virtual Addressing scheme enabling software-programmable memory segments.
- 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.
- VPE Virtual Processor Element
- TC Thread Context

1.2 Tools

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

- Boston Software Development Boards with Concord FPGA daughter cards
- MIPS32 bit files programmed into BOSTON core card
- Codescape SDK
- Codescape Debugger or Codescape for Eclipse
- SP55 EJTAG debug Probe
- IASim
- Host PC (Windows 10)

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 8.5 or higher. It is assumed that this has been installed.

Boston development board

The Boston development board is used for this core.

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 nanomips-img-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

/opt/imgtec/Toolchains/nanomips-img-elf/<version>

Windows default toolchain path

C:\Program Files\Imagination Technologies\Toolchains\nanoMIPS-img-elf\<version>

Installing the full SDK will automatically set these paths, using this script: imgtec.sh.

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. Where to get the Source 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

2.3. Building the Boot MIPS Project

The Boot-MIPS project is built using a Makefile

2.3.1. Target Selection for Make

- I7200_SIM_RAM – IASim Simulator Build with RAM copy

- I7200_SIM_SPRAM - Simulator Build with Scratchpad RAM
- I7200_SIM_RAM_MPU – IASim Build for simulator configured for MPU
- I7200_BOSTON_RAM - Boston Build with RAM copy
- I7200_BOSTON _RAM_MPU – Malta Build for bit files built for MPU
- I7200_BOSTON_SPRAM – Malta Build with Scratchpad RAM copy for interAptiv Core Family

There is also a clean target:

- clean_I7200

2.3.2. Make command line

- First cd to Boot-MIPS root directory.
- The use the make command <target> for example make I7200_BOSTON _RAM_MPU or make clean_I7200

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 that pertain to the I7200:

- I7200 – files specific to the nanoMIPS32 I7200 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

3.2. Core Directory Files

The I7200 directory may contains the following files. 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. 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_mpu.ld - This is the linker script that will link the code for a MIPS Boston Evaluation Board using a MPU 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_mpu.ld - This is the linker script that will link the code for the IASim simulator which is configured for a MPU that copies the main.c code from ROM to RAM.

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 appropriate file should be used that matches the core architecture implementation.

- boot.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_nanoMIPS.S the code 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’.
- init_caches.S, init_caches_64.S – initializes the L1 instruction and data caches and the L2 cache if present. See ‘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 v0_all_ones      v0   /* to simplify bit insertion of 1's. */
#define a0_temp_data     a0   /* a0 data to be moved */
#define a1_temp_addr     a1   /* from address */
#define a2_temp_dest     a2   /* to address */
#define a3_temp_mark     a3   /* ending address */
```

3.3.1. Copy to Instruction Scratch Pad

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    v0,C0_CONFIG
        ext     v1, v0, 24, 1
        blez    v1, copy_c2_ram_done // no ISPRAM just exit
        nop
```

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 a1, then moves it to the C0_TAGLO register.

```
        la      a1_temp_addr, _ISPram
        mtc0    a1_temp_addr, 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    s1, s0_save_C0_ERRCTL  // make copy so we can restore       C0_ERRCTL
        ins     s1, v0_all_ones, 28, 1
        mtc0    s1, 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($0)
```

Next the code sets up the register to hold the virtual ROM address to copy from. First it loads a1_temp_addr 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      a1_temp_addr, _zap1                   // starting ROM address
        ins     a1_temp_addr, v0_all_ones, 29, 1      // convert to uncached
```

The code sets up a2 register to hold the virtual memory address to copy to.

```
        la      a2_temp_dest, _ftext_ram // starting ram address to copy to
```

Then the code sets up the a3_temp_mark register to mark the end address of the copy.

```
        la      a3_temp_mark, _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     a2_temp_dest, a3_temp_mark, zero_bss // equal 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 uses zero to load the initial value into s1, which will be used as the first index to be written to.

```
        add     s1, zero, zero
```

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 v1, where it previously stored the CP0 Config register. It extracts the BE (bit 15) from v1. If this bit is set, the core is big endian; if not, it’s little endian.

```
        ext     v1, v1, 15, 1
        blez    v1, next_Iram_wordLE
        nop
```

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 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 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 a1_temp_addr into a0_temp_data, and then moves the a0_temp_data value to the C0_DATALO register.

```
next_Iram_wordBE:
        lw      a0_temp_data, 0(a1_temp_addr)
        mtc0    a0_temp_data,C0_DATAHI
        addiu   a1_temp_addr, 4
        lw      a0_temp_data, 0(a1_temp_addr)
        mtc0    a0_temp_data, 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($11)
```

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

```
        addiu s1, 8
        addiu a2_temp_dest, 8
```

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    a2_temp_dest, a3_temp_mark, next_Iram_wordBE
```

The “from” address is incremented by an instruction in the branch delay slot (always executed with the branch).

```
        addiu a1_temp_addr, 4
```

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

```
        b      set_dspram
```

Skipping the little endian copy …….

3.3.2. 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    v1,C0_CONFIG
        ext     v1, v1, 23, 1
        blez    v1, copy_c2_ram_done //no DSPRAM just exit
 nop
```

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 a1_temp_addr to C0_DTAGLO.

```
        la      a1_temp_addr, _DSPram
        // set the enable bit
        ins   a1_temp_addr, v0_all_ones, 7, 1
        // move it to the tag register
        mtc0    a1_temp_addr, 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 a1_temp_addr with 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      a1_temp_addr, _zap2                 // starting ROM address
        ins     a1_temp_addr, v0_all_ones, 29, 1     // uncached address
```

The code sets up a2 to hold the virtual memory address to copy to.

```
        la      a2_temp_dest, _fdata_ram      // starting ram address to copy to
```

Then the code set up the a3_temp_mark register to mark the end address of the copy.

```
        la      a3_temp_mark, _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     a2_temp_dest, a3_temp_mark, zero_bss // if = nothing to do
        nop
```

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 $11. Then the index is incremented to the net word to be written in the DSPRAM.

```
next_Dram_word:
        lw      a0_temp_data, 0(a1_temp_addr)
        mtc0    a0_temp_data, $28, 3
        cache   0xd,0(s1)
        addiu   s1, 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 a3_temp_mark which is the end address and the current destination address to see if they are equal.

```
        addiu   a2_temp_dest, 4
        bne     a2_temp_dest, a3_temp_mark, next_Dram_word
        addiu   a1_temp_addr, 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. 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      a1_temp_addr, _fbss
        ins     a1_temp_addr, r1_all_ones, 29, 1
        la      a3_temp_mark, _end
        ins     a3_temp_mark, r1_all_ones, 29, 1
        beq     a1_temp_addr, a3_temp_mark, copy_c2_ram_done
        nop
```

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 a1_temp_addr. 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 a3_temp_mark, and loops back if it is not.

```
next_bss_word:
        cache   0xd,0(s1)              // write DDATA LO to DSPRAM
        addiu   a1_temp_addr, 4
        addiu   s1, 4                  // add 4 to DSPRAM index
        bne     a1_temp_addr, a3_temp_mark, 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 a1_temp_addr, 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      a1_temp_addr, _ISPram
        // set the enable bit
        ins   a1_temp_addr, v0_all_ones, 7, 1
        // move it to the tag register
        mtc0    a1_temp_addr, 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
        nop
   END(copy_c2_ram)
```

- common/init_caches.S'.
- init_cp0.S initializes Coprocessor 0 Registers.
- init_gpr.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 initializes L2 cache for the coherency manager.

3.4. cps Directory Files

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

- cps.h - #defines for CM, CPC, and GIC for the CM
- cm2.h - #defines for CM, CPC, and GIC for the CM2
- init_cm.S initializes the Coherence Manager.
- init_cpc.S – initializes the Cluster Power Controller.
- init_gicCM26N.S – initializes the Global Interrupt Controller
- join_domain.S - joins a processing element to a Coherence Domain.
- release_mp.S – releases a core for multi-processing.

3.5. 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.
- Boston_lcd_ab.S – This file contains code need to display messages on the lcd display. This is code is a duplicate of Boston_lcd.S but it resides in RAM and not flash.
- init_FPGA_mem.S – This file waits for the memory controller to initialize.

3.6. mt Directory Files

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

- init_vpe_s.S– initializes the Virtual processors for an I7200. This differs from the init_vpe1.S in that it will initialize more than 2 VPEs. See 'mt/init_vpe1.S - init_vpe_s()'

3.7. 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 5 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. I7200 Core

The I7200 is a multi-core multi-threaded/mulyi-core processor that supports up to 4 Cores 3 VPEs and 9 threads. The boot begins by executing the start.S. Refer to ‘start.S ’ on page 17 for the start of a code walk through.

4.2. common/boot.h

This file contains defines that are common to many cores and are used by the Boot MIPS code.

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.

For Traditional MIPS with TLB:

Define used in main.c for writes of inter processor interrupts:

```
#define GIC_SH_WEDGE_ADDR      *((volatile unsigned int*) (0xbbdc0280))

#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

With MPU instead of TLB:
#define GIC_SH_WEDGE_ADDR   *((volatile unsigned int*) (0x1bdc0280))
#define GCR_CONFIG_ADDR     0x1fbf8000  // address of the GCR registers
#define GCR_CONFIG_ADDR_PB  0x1fbf8000  // Post Boot address of the GCR registers
#define GIC_P_BASE_ADDR     0x1bdc0000  // physical address of the GIC
#define GIC_BASE_ADDR       0x1bdc0000  // address of the GIC
#define GIC_BASE_ADDR_PB    0x1bdc0000  // Post Boot address of the GIC
#define CPC_P_BASE_ADDR     0x1bde0001  // physical address of the CPC
#define CPC_BASE_ADDR       0x1bde0000  // address of the CPC
#define CPC_BASE_ADDR_PB    0x1bde0000  // Post Boot address of the CPC

#define STACK_BASE_ADDR     0x02000000  // Change: Base on memory size

#define CDMM_P_BASE_ADDR      0x1fc10000 // address of the CDMM Regsiter
#define MPU_CDMM_OFFSET     (64*3)
#define MPU_ACSR            0
#define MPU_Config          0x8
#define MPU_SegmentControl0 0x10
#define MPU_SegmentControl1 0x14
#define MPU_SegmentControl2 0x18

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
```

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: (Note p32 ABI)

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

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

// $12 - $15 (t4 - t7 o32 or t0 - t3 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 malta word  $24  /* t8 Uncached (kseg1) base address of Malta ascii display.
GLOBAL! */
#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
```

4.3. 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.

The code begins by setting options for the assembler:

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

4.3.1. 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. This jump will go around a Malta board’s ID register that holds the revision number, because the boot code will not fit in the space allotted for the boot exception vector. The jump also 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, boot_code
    mtc0  zero, C0_COUNT     // Clear cp0 Count (Used to measure boot time.)
    jr a2
END(__reset_vector)
```

4.3.2. 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 0200.

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.

That is what happens with the first 4 exception vectors:

```
.org 0x180
#ifdef MPU
    j   General_Exception // general exception address for MPU redirect to general handler
#else
    sdbbp                 // if probe attached break to debugger
#endif
.org 0x200 /* TLB refill, 32 bit task. */
#ifdef MPU
    j   General_Exception  // Interrupt for MPU redirect to general handler
#else
    sdbbp                 // if probe attached break to debugger
#endif
```

4.3.3. General exception

The only exception the boot code is expecting is a inter-processor interrupt. The boot code uses the inter-processor interrupt to synchronize all VPEs at the end of the boot so they will all execute whatever code comes after the boot at the same time.

```
.org 0x380 /* General exception. */
General_Exception:
        // expecting an inter-processor interrupt -
// read the C0_CAUSE register and check to see if this is an interrupt
// and it is coming from vector 0. If not it will branch to other_code.

    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, 5                       // extract interrupt
    li    k1, 1
    bne   k0, k1, other_code          // should be interrupt 0


    // process the inter processor interrupt
    // get address of write edge register
    li    k0, (GIC SH WEDGE | GIC BASE ADDR PB)

    mfc0  k1, C0_EBASE                        // Get cp0 EBase
    ext   k1, k1, 0, 10                       // Extract CPUNum
    addiu k1, 0x20                    // Offset to base of IPI interrupts
    sw    k1, 0(k0)                   // Clear this IPI by writing to the write edge register

    // Write a 1 to the element in the start test array for this processor
    // once the code returns to normal processing (in main()) it will check the
    //  start[CPUNum]
    // to see if it should break out of the idle loop and continue with its test code
    la  k0, start_test                // load the address of the start_test array
    mfc0  k1, C0_EBASE                // Get cp0 EBase
    ext   k1, k1, 0, 10                       // Extract CPUNum
    sll   k1, k1, 2                   // x 4 for integer element
    addu  k0, k0, k1                  // index into start_test array
    li    k1, 1
    sw    k1, 0(k0)                   // Release "cpu"/vpe to execute "C" test code
    eret                              // Return to code in main

other_code:
    // unexpected exception read the C0_CAUSE and C0_EPC so it will appear in a trace file
    // Just to make it easier to see what's in the registers
    mfc0  k1, C0_CAUSE
    mfc0  k0, C0_EPC

    sltiu zero, zero, 0xbc1   // if using simulator stop with simulator fail condition
                                      // otherwise this just looks like a nop
    sdbbp                             // if probe attached break to debugger
```

4.3.4. Ejtag Exception:

For an EJTAG exception (which you should only get on live hardware that includes EJTAG), the code goes into an infinite loop and there is no prob connected to the EJTAG port

```
.org 0x480 /* debug exception (EJTAG Control Register [ProbTrap] == 0.) */
    // Write to Boston display indicating a exception
    addu a0,  zero, r23_cpu_num
    li a1, 0x58 // X - Show  exception
    jalr r24_boston_byte

    // if using simulator stop with simulator fail condition
    // if not using a simulator this just looks like a nop
    sltiu      zero, zero, 0xbc1
1:  b  1b      /* Stay here */
```

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

Recall that the code at the boot exception vector just branches to boot_code.

```
.org 0x500 /* Resume code past the boot exception vectors. */
boot_code:
#ifdef MPU
// For a I7200 that is using the MPU instead of a TLB the MPU will need to
// be initialized Noted : The processor boots in FMT mode this boot code is
// loaded into a kuseg address which is mapped VA=PA for kuseg only.
// After enable MPU then MPU ALL of memory is mapped VA=PA

    // Only VPE0 can initialize the MPU
    mfc0  a0, C0_EBASE
    andi  a0, 1
    bnez  a0, MPU_EnableMPU_Done  // not VPE 0

MPU_EnableMPU:

    // load CDMM register base address
    li    a0, (CDMM_P_BASE_ADDR>>4)|(1<<10)
    mtc0  a0, C0_CDMMBASE
    ehb                           // ehb remove cp0 hazard
    li      a0, CDMM_P_BASE_ADDR      // load a0 with address for CDMM
    lw      a1, MPU_CDMM_OFFSET+MPU_Config(a0)
    li      a2, (1<<31)                       // MPU_Config bit 31 = enable
    or      a1, a1, a2
    sw      a1, MPU_CDMM_OFFSET+MPU_Config(a0)  // Enable MPU
    sync

MPU_EnableMPU_Done:
    // MPU is enabled so all memory is mapped 1to1,  VA/PA
#endif
// Check NMI bit
    mfc0    a0, C0_STATUS      // Read CP0 Status
    bbeqzc  a0, 19, verify_isa // Branch if this is NOT an NMI exception

    // NMI was set so stop, exit to debugger or process debug interrupt
    // (otherwise processor will go into an endless boot loop)
    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 (MIPS32R2 or later.)
    #ifdef BOSTON_RAM  // make sure ram is up before checking array
         dla   a2, init_FPGA_mem      // Initialize the ROC-it2 MC (Memory Controller.)
         jalr  a2
    #endif
    mfc0  a0, C0_CONFIG               // Read CP0 Config
    srl   a0, 10                      // Shift [AT AR] into LSBs.
    andi  a3, a0, 0x18                // Inspect CP0 Config[AT]
    beqz  a3, is_mips32               // Branch if executing on MIPS32 ISA

    sdbbp                             // Failed assertion: MIPS32R2
```

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_mips32:
    andi  a3, a0, 0x07                // Inspect CP0 Config[AR]
    bnez  a3, init_common_resources   // Continue if ISA is MIPS32R2 or later

    sdbbp                             // Failed assertion MIPS32R2
```

4.3.6. Initializing Common Resources

The next section of start.S initializes resources that are common to every processing element in the core(s). For an I7200, this section will be executed by each VPE on 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 after the jump delay slot 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
```

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
```

```
// 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
```

If this is not a MPU test for a TLB.

```
#ifndef MPU
// 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  a2, C0_CONFIG       // read C0_Config
    ext   a2, a2, 7, 3        // extract MT field
    li    a3, 0x1             // load a 1 to check against
    bne   a2, a3, done_tlb    // no tlb?

    la    a2,  init_tlb
    jalr  a2
done_tlb:
#endif
```

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.*

Next call the global interrupt controller initialization

```
    la    a2,  init_gic       // Initialize Global Interrupt Controller
    jalr  a2
```

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
```

4.3.7. Initializing Core Resources

```
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
```

The next two calls, init_icache and init_dcache, found in common/init_caches.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
```

If this is not an EVA or MPU 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.

```
#if defined (EVA) || defined (MPU)
    // Only for cores built to boot in EVA OR MPU 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

#endif
```

```
#ifdef MPU
    // Since the icache has been initialized change the MPU CCA setting to
    // cacheable so the remainder of the
    // code will run cached making the execution much faster
    li    a0, CDMM_P_BASE_ADDR       // a0 have kseg1 address for CDMM

    // Segment 0 set to CWB (Cacheable, coherent, write-back, write-allocate,
    // read misses request Shared)
    li    a1, (0x2)<<24|(0x2)<<16|(0x2)<<8|(0x5)
    // Segment 0 address 0~0x0FFFFFFF to cacheable
    sw    a1, MPU_CDMM_OFFSET+MPU_SegmentControl0(a0)
    sync
#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
```

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

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

4.3.8. 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

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

    // init_cm is only present for multi-processor systems.
    // It initializes the Coherence Manager.
    la a2, init_cm  // Initialize Coherence Manager
    jalr a2
```

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
```

For MPU Cores the cache needs to be flushed.

```
    #ifdef  MPU
        // Under MPU mode, we don't have core base uncache segment
        // (system memory alias might have uncache)
        // Here copy_c2_ram done by cache address, So when copy is done the
        // code needs to flush all Dcache to memory
       // Flush dirty data from data cache using IndexWritebackInvalidate
        // CACHE instruction on all lines in the cache.
        la    a2,     flush_dcache    // Flush dirty lines out of data cache
       jalr   a2
        nop
    #endif
```

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
```

init_L2 initializes the L2 cache for multi-processor systems

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

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

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

Until this point all other cores in the CPS should have been either powered down or held in reset. Now it’s time to release them to execute this boot code.

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

init_sys_resources_done:      // All Cores (VPE0)

    // join_domain associates the CORE with a Coherence Domain.
    la    a2, join_domain     // Join the Coherence domain
    jalr  a2

    // init_vpe1 will setup the second VPE if present to run this boot code
init_vpe_next:
    la    a2, init_vpe1       // vpe1 to execute boot code
    jalr  a2
```

4.3.9. 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 MALTA 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|SEGCTL_CFG4_EU_MA
SK|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|SEG
CTL_CFG3_EU_MASK|SEGCTL_CFG2_C_UC|SEGCTL_CFG3_C_WB)
    mtc0       a0, C0_SEGCTL1
    ehb
#endif
```

```
For MPU systems the C0_EBASE register needs to be set.
```

```
#ifdef MPU
    // Because the MPU doesn't have a kesg0 or kesg1 the exception vector must
    // redirect to MPU RAM
    // Here set EBASE to the MPU RAM address and remove Status.BEV to let Ebase work
MPU_Set_Ebase:
    la    a0, __reset_vector          // EBase to reset vector
    ori   a0, (1<<11)                 // MPU Ebase need set bit 31~30 enable WG bit
    mtc0  a0, C0_EBASE
    ehb

    //Remove Status.BEV
    mfc0  a0, C0_STATUS
    li    a1, ~(1<<22)
    and   a0, a0, a1
    mtc0  a0, C0_STATUS
    ehb
#endif
```

The code will put the address of the all_done label in the Return address register (ra/$31), 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
```

For cores other than EVA or MPU the address of num_cores needs to be converted to a kseg1 uncached address:

```
    #if defined (EVA) || defined (MPU)
    #else
        li     a2, 0xFFFFFFFF
        ins    a1, a2, 29, 1                // convert it to a Uncached kseg1 address
    #endif

    add a0, r19_more_cores, 1
    sw  a0, 0(a1)
```

Before main() begins executing, the code sets up the arguments to main. These arguments correspond to the argument registers in the p32 ABI, a0 through a4.

```
    // 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 number.
    addiu a3, r20_more_vpes, 1      // main(arg3) is the number of vpe
    addiu a4, r19_more_cores, 1     // main(arg4) total cores in system
```

The boot of the VPE 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
```

4.4. 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 p32 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 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.
```

4.5. MT ASE Check

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_CONFIG3                     // read C0_Config3
    and   a0, (1 << 2)                       // MT
    li    r10_has_mt_ase, 0
    beqz  a0, no_mt_ase
```

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
    ext   r20_more_vpes, a0, 10, 4
    b     check_cps
```

4.5.1. No MT ASE

If the code is executing on a non-MT core, this is an error because the I7200 is defined to be a MT core so the code will exit to the debugger or take a debug exception.

```
no_mt_ase:     // This processor does not implement the MIPS32 MT ASE.
       sdbbp   // Failed assertion: not mt
```

4.5.2. Check for Coherent Processing System

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, 0x01B2                         // I7200 Multi core
    beq   a3, a0, is_cps

    // This processor is not a I7200 Core so exit!
    sdbbp
```

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
```

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 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
    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_ADDR
```

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
```

4.5.3. 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 21.

```
    jr    ra
```

4.6. common/copy_c2_ram_nanoMIPS.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.

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

```
#define all_ones              t2  /* at Will hold 0xffffffff (insertion of 1's) */
#define data                  t3  /* data to be moved */
#define source_addr           a1  /* from address */
#define destination_addr      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, _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, _ftext_ram
```

The _etext_ram is stored in end_addr. _etext_ram is created in the linker file and is the address of the end of the text section. The code will use this address to end the copy of the code section.

```
    LA    end_addr_a3, _etext_ram
```

If this is not an EVA or MPU 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.

NOTE: the switch to uncached is not done for EVA or MPU because of their mapping

```
#if !defined(EVA) && !defined(MPU)
    // Switch address to uncached (kseg1) so copy will go directly to memory
    ins   destination_addr_a2, all_ones_s1, 29, 1
    ins   end_addr_a3, 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, end_addr, copy_data
```

The code uses the load word multiple (lwm) and store word multiple (swm) for the copy. Their use reduces the number of loops the copy takes which will improve the performance of the code. The lwm and swm instructions will use the save registers (s0 – s7) so it save those registers to the stack before entering the copy loop.

```
    // Save registers used by lwm/swm to stack
    save       STACK_ALGN(8), s0 - s7
```

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

```
next_ram_word:
    lwm   data, 0(source_addr), 8
    swm   data, 0(destination_addr), 8
    addiu destination_addr, (4 * 8)
    addiu source_addr, (4 * 8)
    bne   end_addr, destination_addr, next_ram_word
```

Next a copy of the initialized data is done. This is very similar to the text copy. It just uses different start and end address.

```
copy_data:
// now copy the data
    LA    source_addr, _zap2
    LA    destination_addr,  _fdata_ram
    LA    end_addr, _edata_ram
    #if !defined(EVA) && !defined(MPU)  // NOTE EVA mode assumed to be uncached
        // Switch address to uncached (kseg1) so copy will go directly
        // to memory
        ins   destination_addr, all_ones, 29, 1
        ins   end_addr, all_ones, 29, 1
    #endif
    beq     destination_addr, end_addr, zero_bss  // if no data skip ahead

next_ram_dword:
    lwm   data, 0(source_addr), 8
    swm   data, 0(destination_addr), 8
    addiu destination_addr, (4 * 8)
    addiu source_addr, (4 * 8)
    bne   end_addr, destination_addr, next_ram_dword
```

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 or MPU 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, _fbss
    LA    end_addr, _end
    #if !defined(EVA) && !defined(MPU)

        // Switch address to uncached (kseg1) so copy will go directly
        // to memory
        ins   destination_addr, all_ones_s1, 29, 1
        ins   end_addr, all_ones, 29, 1
    #endif
    beq   destination_addr, end_addr, copy_c2_ram_done
```

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_a3 , and loops back if it is not.

```
next_bss_word:
    swm   zero, 0(destination_addr), 8
    addiu destination_addr, (4 * 8)
    bne   destination_addr, end_addr, next_bss_word

The code has finished the copy and returns.
copy_c2_ram_done:
    jalr  zero,      ra

.set at
END(copy_c2_ram)
```

4.7. 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 v0_all_ones     v0    /* to simplify bit insertion of 1's. */
#define a0_temp_data    a0    /* a0 data to be moved */
#define a1_temp_addr    a1    /* from address */
#define a2_temp_dest    a2    /* to address */
#define a3_temp_mark    a3    /* ending address */
```

4.7.1. Copy to Instruction Scratch Pad

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    v0,C0_CONFIG
        ext     v1, v0, 24, 1
        blez    v1, copy_c2_ram_done // no ISPRAM just exit
        nop
```

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 a1, then moves it to the C0_TAGLO register.

```
        la      a1_temp_addr, _ISPram
        mtc0    a1_temp_addr, 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    s1, s0_save_C0_ERRCTL  // make copy so we can restore       C0_ERRCTL
        ins     s1, v0_all_ones, 28, 1
        mtc0    s1, 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($0)
```

Next the code sets up the register to hold the virtual ROM address to copy from. First it loads a1_temp_addr 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      a1_temp_addr, _zap1                 // starting ROM address
        ins     a1_temp_addr, v0_all_ones, 29, 1     // convert to uncached
```

The code sets up a2 register to hold the virtual memory address to copy to.

```
        la      a2_temp_dest, _ftext_ram // starting ram address to copy to
```

Then the code sets up the a3_temp_mark register to mark the end address of the copy.

```
        la      a3_temp_mark, _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     a2_temp_dest, a3_temp_mark, zero_bss // equal 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 uses zero to load the initial value into s1, which will be used as the first index to be written to.

```
        add     s1, zero, zero
```

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 v1, where it previously stored the CP0 Config register. It extracts the BE (bit 15) from v1. If this bit is set, the core is big endian; if not, it's little endian.

```
        ext     v1, v1, 15, 1
        blez    v1, next_Iram_wordLE
        nop
```

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 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 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 a1_temp_addr into a0_temp_data, and then moves the a0_temp_data value to the C0_DATALO register.

```
next_Iram_wordBE:
        lw      a0_temp_data, 0(a1_temp_addr)
        mtc0    a0_temp_data,C0_DATAHI
        addiu   a1_temp_addr, 4
        lw      a0_temp_data, 0(a1_temp_addr)
        mtc0    a0_temp_data, 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($11)
```

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

```
        addiu s1, 8
        addiu a2_temp_dest, 8
```

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   a2_temp_dest, a3_temp_mark, next_Iram_wordBE
```

The "from" address is incremented by an instruction in the branch delay slot (always executed with the branch).

```
        addiu a1_temp_addr, 4
```

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

```
        b       set_dspram
```

Skipping the little endian copy …….

4.7.2. 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    v1,C0_CONFIG
        ext     v1, v1, 23, 1
        blez    v1, copy_c2_ram_done //no DSPRAM just exit
 nop
```

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 a1_temp_addr to C0_DTAGLO.

```
        la      a1_temp_addr, _DSPram
        // set the enable bit
        ins   a1_temp_addr, v0_all_ones, 7, 1
        // move it to the tag register
        mtc0    a1_temp_addr, 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 a1_temp_addr with 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      a1_temp_addr, _zap2                 // starting ROM address
        ins     a1_temp_addr, v0_all_ones, 29, 1    // uncached address
```

The code sets up a2 to hold the virtual memory address to copy to.

```
        la      a2_temp_dest, _fdata_ram      // starting ram address to copy to
```

Then the code set up the a3_temp_mark register to mark the end address of the copy.

```
        la      a3_temp_mark, _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     a2_temp_dest, a3_temp_mark, zero_bss // if = nothing to do
        nop
```

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 $11. Then the index is incremented to the net word to be written in the DSPRAM.

```
next_Dram_word:
        lw      a0_temp_data, 0(a1_temp_addr)
        mtc0    a0_temp_data, $28, 3
        cache   0xd,0(s1)
        addiu   s1, 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 a3_temp_mark which is the end address and the current destination address to see if they are equal.

```
        addiu   a2_temp_dest, 4
        bne     a2_temp_dest, a3_temp_mark, next_Dram_word
        addiu   a1_temp_addr, 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. 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      a1_temp_addr, _fbss
        ins     a1_temp_addr, r1_all_ones, 29, 1
        la      a3_temp_mark, _end
        ins     a3_temp_mark, r1_all_ones, 29, 1
        beq     a1_temp_addr, a3_temp_mark, copy_c2_ram_done
        nop
```

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 a1_temp_addr. 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 a3_temp_mark, and loops back if it is not.

```
next_bss_word:
        cache   0xd,0(s1)              // write DDATA LO to DSPRAM
        addiu   a1_temp_addr, 4
        addiu   s1, 4                  // add 4 to DSPRAM index
        bne     a1_temp_addr, a3_temp_mark, 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 a1_temp_addr, 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      a1_temp_addr, _ISPram
        // set the enable bit
        ins   a1_temp_addr, v0_all_ones, 7, 1
        // move it to the tag register
        mtc0    a1_temp_addr, 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
        nop
    END(copy_c2_ram)
```

4.8. 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**	**DA**

- 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
```

4.8.1. 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
```

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 CONFIG1_a2, starting at bit 19, and extracts 3 bits to the least-significant bits of LINE_SIZE_v1.

```
    // Isolate I$ Line Size
    ext   LINE_SIZE, CONFIG1, 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
```

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 CONFIG1_a2, using the extract instruction.

```
    ext   SET_SIZE, CONFIG1, 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_a1. The code then adds one to the value to get the actual number of ways.

```
    // Config1IA == I$ Assoc - 1
    ext   ASSOC, CONFIG1, 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_v0) .

```
    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_v0 because the code needs to use + and – register offsets.

```
    lui   CURRENT_ADDR, 0x8000
    srl   TEMP1, BYTES_PER_LOOP_v0, 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_v0, 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

4.8.2. 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.

4.8.3. 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 in the branch delay slot, 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 in the branch delay slot, 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
    beqz  r11_is_cps, set_kseg0_cca
    li    TEMP2, 3                    // CCA for all others
    li    TEMP2, 5            // CCA for coherent cores

set_kseg0_cca:
    ins   TEMP1, TEMP2, 0, 3  // insert K0 field
    mtc0  TEMP1, C0_CONFIG    // write C0_Config
    jr    ra

END(change_k0_cca)
```

4.9. init_L2_CM26.S - init_L2 for CM revision 2.6

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.

4.10. 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,

4.10.1. Initialize the CP0 Status register

Status Register, CP0 12, 0																
31	27	26	25	24	22	21	20	19	18	17	15 8	7 5	4 3	2	1	0
CU	RP	FR	RE	MX	BEV	TS	SR	NMI	0	CEE	IM(7:0)	0	KSU	ERL	EXL	IE

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
```

4.10.2. 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, a0, 3, 1                // bit 3 (WR) Watch registers implemented
    beq   a1, zero, done_wr
```

The code sets up the initialization value for the Watch Registers in the branch delay slot. 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  a0, C0_WATCHHI              // read C0_WatchHi0
    bne   zero, a0, done_wr
```

The code uses the branch delay slot to clear 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
```

4.10.3. 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)
```

4.11. common/init_gpr.S - init_gpr

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
    nop
END(init_gpr)
```

4.12. common/init_tlb.S

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 v1, 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
```

In the branch delay slot, 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 a0. 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 a0 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 a3. 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 v1 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
```

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 a0 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)
```

4.13. cps/init_cm.S - init_cm Coherence manager

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)
```

4.14. cps/init_cpc.S - init_ cpc Cluster Power Controller

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. In the branch delay slot 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)
```

4.15. cps/init_gic26N.S - init_gic Global Interrupt Controller

init_gic26N.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
```

4.15.1. 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.
```

4.15.2. 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..39) for IPI.
    li    a0, 0xff000000
    sw    a0, GIC_SH_RMASK31_0 (a1) // (disable 0..31)
```

4.15.3. 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.

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)
```

4.15.4. 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.

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 Trigger Register. This configures interrupt sources 24 through 31 to be edge sensitive.

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

4.15.5. Interrupt Dual Edge Registers

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

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	

4.15.6. 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)
```

4.15.7. 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.

The example code will use interrupt sources 24 through 40 for inter-processor interrupts. Although the I7200 only could contain up to 12 VPEs, the interrupt source are allocated in blocks of 4. So if Core 0 had 2 VPEs they would use source 24 and 25. The next Core would use source 28 and 29 and so on.

The following example loops through each Core and with a inter loop looping through each VPE on the Core to set the Interrupt Map Source to VPE register for each VPE.

```
// Initialize configuration of shared interrupts

#define MAP_BIT_SET a0        // VPE bit for MAP Source to VPE register
#define CORE a1               // core within cluster
#define VPE a2                // VPE within core
#define SOURCE a3             // interrupt source number
#define SOURCE_BASE t3        // interrupt source number base
#define SLOTS  t0             // number of Source slots per Core
#define SPACER t1             // Space between Map source to VPE registers
#define BASE_OFFSET  t2       // offset index from the start of the GIC address region
#define ONE    s0             // constant for VPE bit setting in MAP source to VPE register

// initialize register starting values
       li      CORE, 0
       li      SLOTS,  4
       li      SOURCE, 24
       li      SOURCE_BASE, 24
       li      SPACER, GIC_SH_MAP_SPACER
       li      ONE, 1

CORE_LOOP:
       li      VPE,    0
VPE_LOOP:
// MAP_BIT_SET = (CORE * SLOTS) + (VPE)
       mul     MAP_BIT_SET, SLOTS, CORE // compute the base bit index for the Core
       add     MAP_BIT_SET, MAP_BIT_SET, VPE // add VPE number to base bit index

// GIC Interrupt Map source to VPE
// build the Base offset from the GIC_BASE address for the Source register
       // compute the offset index for the particular Source register
       mul     BASE_OFFSET, SPACER, SOURCE
       // add base offset of the GIC Map register to get offset from
       // the GIC_BASE address region
       addi    BASE_OFFSET, BASE_OFFSET, GIC_SH_MAP0_VPE31_0
       // shift the bit for the particular VPE using the VPE bit index
       sllv    MAP_BIT_SET, ONE, MAP_BIT_SET
       // write the VPE bit to the MAP Source to VPE register
       swx     MAP_BIT_SET, BASE_OFFSET(GIC_BASE)


// increment the VPE and Source for next loop
       addi    VPE, VPE, 1
       add     SOURCE, SOURCE, VPE
// check to see if the number of VPEs on core is exceeded if not goto the top of loop
// and program next source
       ble     VPE, r20_more_vpes, VPE_LOOP

// Once the number of VPE on core is exceeded the increment to the next Core
       addi    CORE, CORE, 1

// increment to next Source index
       mul     SOURCE, SLOTS, CORE           // starting slot index number for core
       add     SOURCE, SOURCE, SOURCE_BASE   // add the Source Base index

// check to see if the number of COREs on Cluster is exceeded if not goto the top
// of loop and program next source
       ble     CORE, r19_more_cores, CORE_LOOP

// done programming Source to VPE for interprocessor interrupts
```

4.15.8. 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 Control Register 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
```

4.15.9. 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
    nop
END(init_gic)
```

4.16. 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 increments the other core count and checks to see if there are more cores to wait for and if there are, it branches back to the next coherent core label.

```
    addiu a3, 1
    bge   r19_more_cores, a3, next_coherent_core
```

When all the cores have been waited for, the code returns to start.

```
done_join_domain:
    jr    ra
END(join_domain)
```

4.17. cps/release_mp.S - release_mp

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
                                                      // else use CPC Power Up command.
```

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** / **Meaning** 4’d3 / **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

Code	**Meaning**
4’d3	**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*.

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)
```

The processor count is incremented. 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. If there are other cores, the code loops back to power up the next processor.

```
    addiu a3, a3, 1
    bge   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)
```

4.18. 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.

4.19. mt/init_vpe_s.S - init_vpe1

This code initializes VPEs/TCs of an I7200 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 target_VPE t0                   // will hold the current VPE
```

It first checks to see if there is at least 1 additional TC. If there is then initialize the starting TC number.

```
LEAF(init_vpe1)
   // each vpe will need to set up additional TC bound to it
   // If there are more TCs than VPEs the remaining TCs will be bound to
   // the highest numbered VPE

    beqz r21_more_tcs, done_init_vpe   // return if there is no more TCs
    li   target_TC, 1
```

Now the code figures out what the starting VPE should be. The code presents the starting VPE to 0 so if there are no more VPEs then the remaining TCs will be bound to VPE0. Then it checks to see if there are more VPEs and if there are I changes the starting VPE number to 1.

```
    li   target_VPE, 0              // preset the target VPE to 0
    beqz r20_more_vpes, just_1_VPE // If there is no other vpes then leave target_VPE set to 0
    li   target_VPE, 1              // Since there are more than 1 VPEs start with VPE1
```

To setup the remaining VPEs 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.

```
    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
```

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, 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 a0 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 a0 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 a3_target_TC'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
```

NOTE: The code above will generate the following error:

../mt/init_vpe_s.S: Assembler messages:

../mt/init_vpe_s.S:107: Warning: used $at without ".set noat"

This is because of the "mttgpr $2, $1", $1 is the assembler temporary register that the assembler will use when it needs to expand an instruction. The assembler can't tell that the code is writing to another TCs registers so it gives this warning. You can safely ignore this warning.

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.

```
    mftc0 a1, C0_TCBIND       // Read C0_TCBind
    ins   a1, target_VPE, 0, 4// insert vpe into CurVPE field
    mttc0 a1, C0_TCBIND       // write C0_TCBind
```

Each VPE will have 1 TC executable. The code checks here to see if this TC is an additional TC and if so will branch around making this TC active.

```
    // only what one TC per VPE to execute. Since each TC will be assigned
    // to a VPE in order then when the TC number exceeds the VPE number
    // Don't continue to set it up to execute the boot code
    bgt  target_TC, target_VPE, check_for_more
```

Next set this TC to be the only TC runnable on the VPE. This effectively sets TC1 to run exclusively on VPE1 and TC2 to run exclusively on VPE 2. 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
    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.

```
    // 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).

```
    // Load the C0_TCRESTART with the reset vector so each VPE will execute
    // the boot code
    la   a1, __reset_vector // load boot code starting address
    // If this is not a EVA or MPU systm then the __reset_vector address to
    // cached address vpe other than 0 of each core can execute cached as
    // it's L1 I$ has already been initialized and the L2$ has been initialized or "disabled".
    #if !defined(EVA) && !defined(MPU)
        ins a1, zero, 29, 1  // Convert to cached kseg0 address in case we linked to kseg1.
    #endif
    mttc0 a1,  C0_TCRESTART // write C0_TCRestart so TC 1 on VPE 1
                       // will execute this boot code once EVP is set in C0_MVPCtl below
```

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
```

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
```

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
```

Now the code checks for additional VPEs

```
check_for_more:       // Done initializing VPE
    // If incrementing the target_VPE would be greater than or equal to hightest vpe number
    // then all VPEs have a TC bound to them to execute. If there are any remaining
    // TCs bind them to the last VPE
    addu  a0, target_VPE, 1 // Set up a0 for the check
    beq   a0, r20_more_vpes, more_vpe
    bgt   a0, r20_more_vpes, more_tc
    // NOTE Never can be less than!
more_vpe:
    move  target_VPE, a0 // else advance target_VPE and fall through to check for more TC's

more_tc:
    addu target_TC, 1  // advance TC number
    ble  target_TC, r21_more_tcs, nexttc // if no more TCs then fall through

        // No more TC's to bind
```

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_vpe:
    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 target_VPE
```

4.20. main.c

The main.c code that is located in the I7200 directory.

For all cores, the boot_count array is intended to hold the cycle time it took to boot each VPE.

(NOTE: volatile int boot_count[16];//global variable zeroed in start.S when bss is zeroed.)

Core 0/VPE0 will wait until the remaining VPEs (if present) have gone through this boot code and are executing in main and have set their ready flag. Core 0/VPE0 will then send an interprocessor interrupt to each VPE and wait until each VPE has cleared their ready flag then continue.

All VPEs except Core/VPE0 will set a ready flag and then wait for an interprocessor interrupt from Core0/VPE0. When the interrupt is received exception interrupt routine will set the start_test flag for the VPE. The VPE will clear its ready flag and continue.

All VPEs will then display their Core's number in the segment on the Boston display or in the case of a simulator will stop execution.

4.20.1. set_ipi()

The set_ipi function will send an interrupt through the Global Interrupt Controller to a specific VPE number using a combination of the Core number and VPE number (this also corresponds to the CP0 EBase register CPUNum field.). The init_gic.S code sets up the GIC to wire an interrupt to a specific VPE.

IPIs are separated into blocks of 4 per Core starting with Core0/VPE0-2 corresponding to interrupts 24-26. Core1/VPE0-2 corresponding to interrupts 28-30.

The #define GIC_SH_WEDGE _ADDR (common/boot.h) 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 (main.c) 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_ADDR       *((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 _ADDR = 0x80000000 + FIRST_IPI + cpu_num ; }
```

4.20.2. Main()

```
//
// 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], $9": [temp] "=r"(temp) :) ;
boot_count[cpu_num] = temp ;
```

Each Core writes the number of VPE on it to its element in the vpe_on_core array.

```

        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 section.

```
        // 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.
                ready[0] = 1;  // don't want to wait for itself;
                for(core = 0; core < num_cores_in_system; core++)
                   for(vpe = 0; vpe < num_vpe_this_core; vpe++) {
                        display_char = (char)(cpu_num + 0x30);   // ASCII char for cpu number.
                                boston_lcd_display_byte_ab(cpu_num, 'W');
                                while(!ready[(core * 4)+ vpe]);
                        }
```

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 VPEs to run their tasks.
                for(core = 0; core < num_cores_in_system; core++)
                        for(vpe = 0; vpe < num_vpe_this_core; vpe++) {
                                i = ((core * 4)+ vpe);
                                if(i != 0)              // don't send interrupt to itself
                                        set_ipi(i) ;                // Send the ipi
                        }
```

Core 0 will wait for all other processors to acknowledge they have received the interrupt. Once a core receives the interrupt it will acknowledge it by clearing its element in the ready array.

```
                // Wait for other VPEs to clear ready to indicate they got the IPI.
                ready[0] = 0;  // don't want to wait for itself;
                for(core = 0; core < num_cores_in_system; core++)
                        for(vpe = 0; vpe < num_vpe_this_core; vpe++)
                                while(ready[(core * 4)+ vpe]);
```

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_ADDR = 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...
                display_char = (char)(cpu_num + 0x30);
                boston_lcd_display_byte_ab(cpu_num, 'r');
```

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") ;
```

Set the element for this processor in the ready array. This will indicate to Core 0 the this processor is ready for its inter-processor interrupt.

```
                ready[cpu_num] = 1 ;
```

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") ;

                }
                // Clear ready so CPU0 will know
                // this CPU has cleared the wait state.
                ready[cpu_num] = 0 ;
        }
```

All CPUs will now display a “t’. This code is just a place holder, and it’s a good place to put your test code.

```
// Put test code here:
```

If the code is executing in the IASim simulator the next line will triger the simulator to stop and print a Pass message. NOTE: once the simulator get this instruction it will halt all VPEs on the Core. In other words the first VPE that executes this instruction will stop all other VPEs where ever they are at in their code.

```
asm volatile ("sltiu $r0, $r0, 0xbc2");
```

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 go back to the top of the while loop. The effect will be a blinking 0 on the Boston display.

display_char = (char)(cpu_num + 0x30);   // ASCII char for cpu number.

blank = (char)0x20;

```
 while (1) {
     boston_lcd_display_byte_ab(cpu_num, display_char);
     for (j = 0 ; j < ((cpu_num+1) * 1000000) ; j++)  {
        asm volatile ("nop     ": : :) ;
     }
     boston_lcd_display_byte_ab(cpu_num, blank); // blank out display position for this "cpu".
     for (j = 0 ; j < ((cpu_num+1) * 1000000) ; j++) {
        asm volatile ("nop     ": : :) ;
     }
 }
 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.

The targets are:

I7200_SIM_RAM_MPU

Builds a simulator version with the main function copied to normal RAM with a MPU

```
I7200_SIM_RAM_MPU:
        ${MAKE} -C I7200 SIM_RAM_MPU GCC_VENDOR=$(GCC_VENDOR)
```

I7200_SIM_RAM

Builds a simulator version with the main function copied to normal RAM.

```
I7200_SIM_RAM:
        ${MAKE} -C I7200 SIM_RAM
```

I7200_SIM_SPRAM

Builds a simulator version with the main function copied to Scratchpad RAM.

```
I7200_SIM_SPRAM:
        ${MAKE} -C I7200 SIM_SPRAM
```

I7200_BOSTON_RAM_MPU

Builds a Boston Board version with the main function copied to normal RAM with a MPU

```
I7200_MALTA_RAM_MPU:
        ${MAKE} -C I7200 BOSTON_RAM_MPU
```

I7200_BOSTON_RAM

Builds a Boston Board version with the main function copied to normal RAM.

```
I7200_MALTA_RAM:
        ${MAKE} -C I7200 BOSTON_RAM
```

I7200_BOSTON_SPRAM

Builds a Boston Board version with the main function copied to Scratchpad RAM.

```
I7200_MALTA_SPRAM:
        ${MAKE} -C I7200 BOSTON _SPRAM
```

clean_interAptivUP

Cleans all object files in common areas and all built files

```
clean_ I7200:
        ${MAKE} clean -C I7200
```

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 14. 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 Malta 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.

```
CC=nanomips-$(GCC_VENDOR)-elf-gcc
LD=nanomips-$(GCC_VENDOR)-elf-ld
OD=nanomips-$(GCC_VENDOR)-elf-objdump
OC=nanomips-$(GCC_VENDOR)-elf-objcopy
SZ=nanomips-$(GCC_VENDOR)-elf-size
SREC=../../bin/srec_cat.exe
```

$(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. MALTA is the path to the files that are specific to the Malta 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
CPS=$(BASE)cps
MT=$(BASE)mt
```

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) -I $(CPS) -I . -mmt

  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.

- 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_SIM_MPU_RAM=-T sim_Ram_mpu.ld -EL -nostdlib -Wl,-Map=sim_Ram_mpu_map
- LDFLAGS_BOSTON_RAM_MPU=-msoft-float -T Boston_Ram_mpu.ld -EL -nostdlib -Wl,-Map=boston_Ram_mpu_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
- 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

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
```

For MPU Builds and TLB builds if it is a MPU build then –DMPU is added the CFLAGS to provide a define for the source code. If it not a MPU build then init_tlb.S is added to the assembler sources so it will be compiled in.

```
        ifeq ($(findstring BOSTON_RAM_MPU, $(MAKECMDGOALS)), BOSTON_RAM_MPU)
        CFLAGS += -DMPU
        else
        ASOURCES += $(COMMON)/init_tlb.S
        endif

        ifeq ($(findstring SIM_RAM_MPU, $(MAKECMDGOALS)), SIM_RAM_MPU)
        CFLAGS += -DMPU
        else
        ASOURCES += $(COMMON)/init_tlb.S
        endif
```

5.2.7. Make Targets

As discussed previously, there are several make targets for each core.

BOSTON_SPRAM

This target builds for a Boston Board using Scratchpad RAM:

```
BOSTON_SPRAM : $(COBJECTS) $(AOBJECTS_SP) $(BOSTONOBJECTS)
```

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 boston_MIPS_BOOT.mcs -intel boston_Ram.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 boston_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 (boston_MIPS_BOOT.mcs).

```
BOSTON_RAM : $(COBJECTS) $(AOBJECTS_RAM) $(BOSTONOBJECTS)
        $(CC)  $(LDFLAGS_BOSTON_RAM) $(COBJECTS) $(AOBJECTS_RAM) \
        $(BOSTONOBJECTS) -o boston_Ram.elf
        $(OD) -d -S -l boston_Ram.elf > boston_Ram_dasm
        $(OC) boston_Ram.elf -O srec boston_Ram.rec
        $(SREC) -output boston_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 (boston_MIPS_BOOT.mcs).

```
BOSTON_RAM_EVA : $(COBJECTS) $(AOBJECTS_RAM) $(BOSTONOBJECTS)
        $(CC) $(LDFLAGS_BOSTON_EVA_RAM) $(COBJECTS) $(AOBJECTS_RAM)   \
        $(BOSTONOBJECTS) -o boston_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 malta_Ram_eva.rec
        $(SREC) -output boston_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 = 0x1fc00000 ;
         .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 Malta 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 Malta 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 I7200 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)
    ../cps/init_gic_CM26N.o(.text)
    ../cps/init_cpc.o(.text)
    ../cps/init_cm.o(.text)
    ../Boston/init_FPGA_mem.o(.text)
    ../common/init_caches.o(.text)
    ../common/init_L2_CM26.o(.text)
    ../common/copy_c2_ram_nanoMIPS.o(.text)
    ../cps/release_mp.o(.text)
    ../common/init_itc.o(.text)
    ../cps/join_domain.o(.text)
    ../mt/init_vpe_s.o(.text)
    ../Boston/boston_lcd.o(.text)
    . = ALIGN(0x100);
    _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(0x100);
       _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(0x100); aligns the end of the section to an 64-byte boundary (align for the use of the lwm/swm instructions).

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(0x100);
      _gp = ABSOLUTE(. + 0x7ff0); /* Base of small data                 */
      *(.lit8)
      *(.lit4)
      *(.sdata)
      . = ALIGN(0x100);
      _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_nanoMIPS.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.