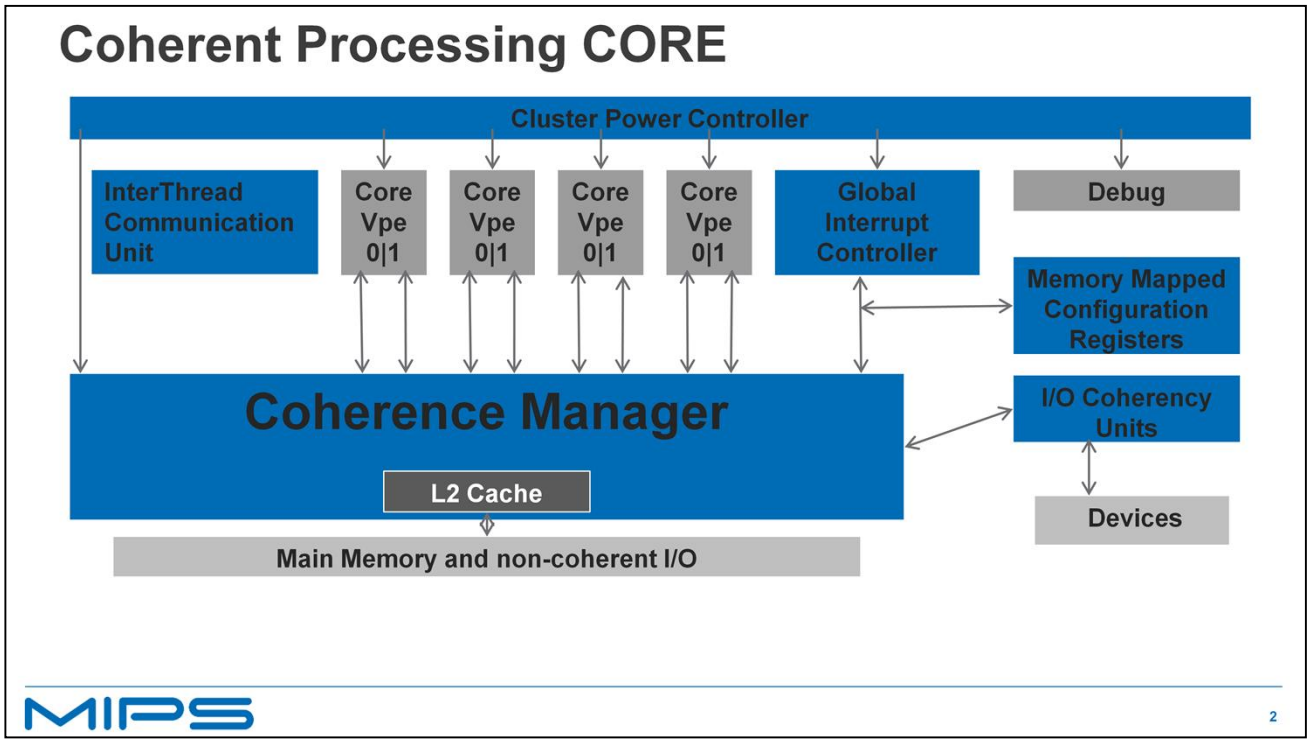
This section describes the Boot-MIPS code and walks you through the booting of a coherent processing system.



The Boot-MIPS code was designed to boot any MIPS Core. It is arranged so that you can compile the code to only use the subset that is needed for a particular core. We are going to look at the code needed for a Coherent Processing System or CPS. It is intended to aid you in becoming familiar with the initialization of a MIPS Coherent Processing System.

Some of this code should look familiar it is similar to code shown in the Programming a MIPS core class, so some of the steps I will go through quickly.

Here is a diagram of a Coherent processing core that the Boot-MIPS code will initialize.

+ A coherent processing core is made up of 1 to 4 single cores with an L2 cache, Debug control block and optional Inter Thread Communications unit.

+ A Coherency Manager or a CM is the Cache coherence domain control logic

+ A Global Interrupt Controller or GIC is the System interrupt control logic

+ The Cluster Power Controller or CPC is the Power domain control logic.

+ The Memory mapped configuration registers configure and control the Global Interrupt Controller, Coherency Manager and Cluster Power Controller.

+ And the I/O Coherency Unit or IOCU controls data coherency between external devices and the L1 and L2 caches through the coherency manager.

+ This class section will step you through the Boot-MIPS example code.

Boot-MIPS Overview

- **Register Use in Boot-MIPS**
  - Boot-MIPS does not follow an ABI (application binary interface) when executing the code in start.S.
  - Instead, some of the general-purpose registers are given a fixed duty, which I will cover when they are first used.
  - See common/boot.h and cps/cps.h for defines used in this code.
- **Exception Vectors while executing the Boot Code**
  - Exception Vectors during the boot process are handled with the software breakpoint instruction “sdbbp”, resulting in debug exceptions if encountered.
  - If you have an EJTAG debug probe attached, it will handle any debug exceptions and leave you “halted” and in control of any CPU taking an unexpected exception.

MIPS

In the code some of the General Purpose Resisters are assigned specific values. These registers have been assigned to a alias using #define so the code will be more readable. Note this does not follow the ABI while the boot code is executing which should be fine since we don’t call C functions until the very end of the code.

+ We don’t expect this boot code to cause any exceptions, but just in case, the exception vectors are programmed to cause a debug exception using the sdbbp instruction.

The target resident debug exception handler will update the display with the value in CP0 DEPC register to aid in the debug of unexpected exceptions.

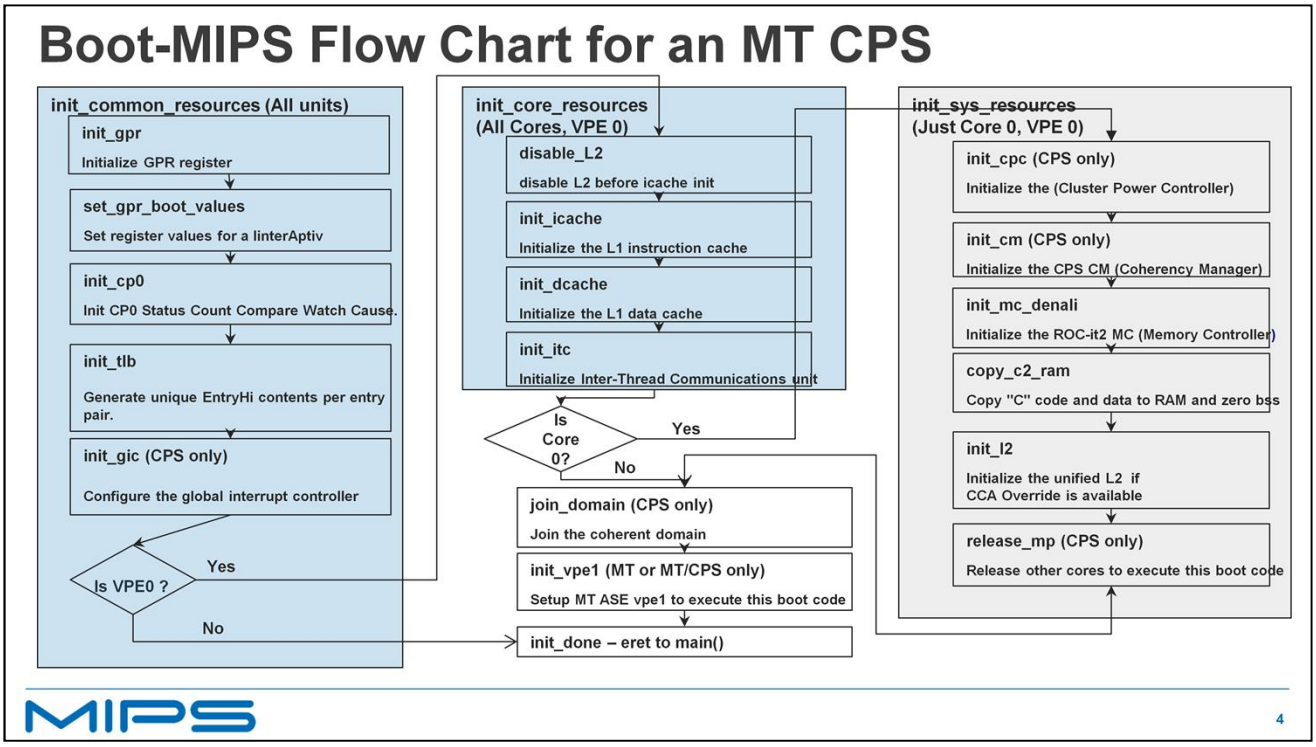
If you have an EJTAG probe attached, control will be given to the probe, which will then be in control of the CPU that took the exception.

I will use the following terminology throughout this course:

A VPE stands for virtual processing element. This is used in reference to a Core that implements the Multi-threaded ASE. Each VPE contains the functionality necessary to support threads executing privileged code such as boot code, exception handlers, interrupts, and execution of an OS. In the context of this course a VPE is synonymous with a processor unless otherwise called out.

The term CORE refers to each core in the CPS a core can be made up of 1 or more VPEs in a Multi threaded system.

A TC stands for thread context. This is used in reference to a Core that implements the Multi-threaded ASE Each TC has the functionality necessary to support a thread of execution.



All of the processors will execute the boot code in the start.S assemble file. The file is divided into sections. The execution of each section will depend on which processor is executing the code. Here is a flow chart of the boot code.

The init_common_resources section is executed by all processors in the system. This code initializes resources that are specific to each processor. These resources include the GPRs, processor specific CP0 registers like the Status, Count, watch and cause, the TLB, and the GIC.

+ The init_core_resources section is executed only once for each Core in the CPS. On a MT system this means that only VPE 0 of each Core will execute this code. It disables the L2 caches at this point. once the L1 instruction cache has been initialized the code can be run out of cache which will make the execution much faster. This section then initializes the L1 caches and the Inter-thread communications unit.

+ The init_sys_resources section is executed only by core 0. This section initializes the coherent processing elements such as the Cluster Power Controller, Coherency Manager, Memory Controller and L2 caches. It also copies the “C” code to ram and clears the bss section. Once this section has been completed, all processors in the system are released from reset and can begin their boot process by using this same code.

+Each Core then joins the coherent domain and for MT cores sets up the second VPE so that VPE can also execute this boot code.

The init_done section is executed by all VPEs. It sets the exception return address register to the address of the main “C” function and then calls the eret instruction which ends the boot process and will start executing the main function.

start.S - init_common_resources for CPS

- init_common_resources section code:

```
init_common_resources:  // initializes resources for virtual or physical "cpu".
    la      a2,init_gpr/     / Fill register file with set value then boot info.
    jalr    a2
    nop
```

MIPS

This slide and the ones that follow will go through the code execution in the start.S file. The call from the start.S file will be shown and then I'll walk you through the function that is called. To start you see the init_common_resources label and the first call to the init_gpr function. The init_gpr function is in the common/init_gpr.S file. To resolve the address of the function the code loads the address of the init_gpr label into a register and then uses the Jump and link register instruction to jump to the address and place the address of the instruction following the nop into the return address register, register 31.

init_gpr.S - init_gpr

- **This function initializes all the GPR sets. Each register receives an initial value of the hex value deadbeef.**
  - Function is located in common/init_gpr.S

```
li          $1, 0xdeadbeef     # (0xdeadbeef stands out, kseg2 mapped, odd.)
# Determine how many shadow sets are implemented (in addition to the base register set.)
mfc0        $29, $12, 2 # C0_SRSCtl
ext         $30, $29, 26, 4          # S_SRSCtlHSS, W_SRSCtlHSS
next_shadow_set:
# set PSS to shadow set to be initialized
ins         $29, $30, 6, 4           # S_SRSCtlPSS, W_SRSCtlPSS
mtc0        $29, $12, 2 # C0_SRSCtl
wrpgpr      $1, $1
wrpgpr      $2, $1
wrpgpr      $3, $1   ..... And so on
```

MIPS

The init_gpr assembler function does as its name suggests and initializes the General Purpose Register sets. It first sets all registers in each register set to the hex value of deadbeef. This can be helpful when debugging the boot code, because if you are reading a register that has the unlikely value of deadbeef, you know you haven't set that register in your code and probably should not be reading it. The code also loops through all shadow register sets, if any, and initializes them. Shadow register set will only exist for non multi threaded processors. Multi threaded processors such as the 34K, 1004K and interAptiv do not have shadow registers instead they can designate the register set of a TC to be used as a shadow set. Each TC will go through this code and initialize it's own registers.

start.S - init_common_resources for CPS

- init_common_resources section code:

```
la      a2,set_gpr_boot_values  // Fill register file boot info.
jalr    a2
nop
```

MIPS

Next in start.S is the call to set_gpr_boot_values. The code is located in set_gpr_boot_values.S. The code in the file varies from core to core so it is not part of the common code.

set_gpr_boot_values.S - set_gpr_boot_values

- **Setup aliased GPRs to their static values**
  - Function is located in set_gpt_boot_values.S

```
LEAF(set_gpr_boot_values)

    li      r1_all_ones, 0xffffffff             // Simplify code and improve clarity
    mfc0    a0, C0_EBASE                        // Read CP0 Ebase
    ext     r23_cpu_num, a0, 0, 4               // Extract CPUNum
    li      r24_malta_word, MALTA_DISP_ADDR     // For reporting failed assertions.
    la      gp, _gp                             // All vpe share globals.
    li      sp, STACK_BASE_ADDR                 // Each vpe gets it's own stack.
    ins     sp, r23_cpu_num, STACK_SIZE_LOG2, 3
```

MIPS

8

The code sets up some of the aliased registers. These registers are defined with #define statements in common/boot.h. The first part of each aliased name is the actual register number, so when your debugging this code it should be easy to follow which actual registers are set.

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

+ next the code reads the Ebase register CP0 register 15 select 1.

+ The Ebase value is used to set the CPU number in GPR 23. The CPU Number is stored in bits 0 through 9 in the Ebase register. In the case of MIPS cores we only need the first 4 bits to determine the CPU number since that covers the maximum number of CPUs in our multi core designs. The code uses the extract instruction to extract 4 bits stating at bit 0. Note: The CPU number being extracted is set using the hardware interface SI_CPUNum pins. For an MT core, there is a different CPU number for each VPE.

+ To get some feed back during the boot process it is helpful to have some kind of visual display. On our Malta evaluation board there is a alphanumeric display and the code sets GPR 24 to the address of that display. Characters written to this address will be seen on the display. Of course you will need to change this for your system.

+ 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 use to reference shared global variables. The MIPS API designates that GPR 28 be used to hold the global pointer address so the code sets it here.

+ Part of each processing elements 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 call STACK_BASE_ADDR has been set by the programmer to a point in memory designated for use for the processor stacks. The MIPS API designates that GPR 29 be used to hold the stack pointer. The code first writes the STACK_BASE_ADDR to GPR 29 then manipulates it using the CPU number so each processing element will have its own stack.

set_gpr_boot_values.S - set_gpr_boot_values

- Determine if on an MT processor

```
check_mt_ase:
mfc0   a0, C0_CONFIG1      // C0_Config1
bgez   a0, no_mt_ase       // bit 31 sign bit set?
mfc0   a0, C0_CONFIG2      // C0_Config2
bgez   a0, no_mt_ase       // bit 31 sign bit set?
mfc0   a0, C0_CONFIG3      // C0_Config3
and    a0, (1 << 2)         // M_Config3MT
beqz   a0, no_mt_ase
 li    r10_has_mt_ase, 0
```

Note the M bit (more) is bit 31 so if set the value of the register would appear negative (less than 0) so successful branch using bgez tells us the bit is not set.

MIPS

9

The Boot-MIPS code can be used for either a Multi-processor system made up of Multi-threaded Cores or only single threaded cores. An MT core has the MT bit 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 will be no 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 needs to 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 signed 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 see if the register value is greater than 0, using the branch greater than zero instruction.

+ The code then looks at the Config2 register and its M bit in the same manner.

+ The code reads the config3 register, isolates the MT bit, bit 2 tests it and branches to the no mt ase function if it is not set.

set_gpr_boot_values.S - set_gpr_boot_values

- **MT specific resources**

```
has_mt_ase:
    li      r10_has_mt_ase, 1
    // 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
    mfc0    a0, C0_MVPCONF0                // C0_MVPConf0
    ext     r21_more_tcs, a0, 0, 8         // Extract PTC
    b       check_cps
    ext     r20_more_vpes, a0, 10, 4       // Extract VPE
no_mt_ase: // This processor does not implement the MIPS32 MT ASE!
    sdbbp                                  // Failed assertion: not mt.
```

MIPS

If the code has determined that it is executing on an MT processor it will set GPR 10 to a one. 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.

+ It will read the CP0 TCBind register and save the number of the VPE context it is executing in right now into GPR 9. It will save the TC it is executing into GPR 18.

+ next it will read the CP0 MVPConf0 and set GPR 21 to the number of TC within the Core and set GPR 20 to the number of VPE contexts in the Core. Then the code will branch to check if this is a coherent processing system.

In previous code there was a check to see if the core was a MT core it jumps to no_mt_ase if it is not. Since it is expecting to be executing on a MT core it will exist to the debugger here.

set_gpr_boot_values.S - set_gpr_boot_values

- Check for Coherency Manager

```
check_cps: // Determine if there is a coherency manager present. (Implementation Dependent.)

    mfc0   r25_coreid, C0_PRID          // CP0 PRId.
    ext    a0, r25_coreid, 8, 16        // Extract Manuf and Core.
    li     a3, 0x01A1                   // interAptiv Multi core
    beq    a3, a0, is_cps
    nop

is_not_interAptiv: // This processor is not a InterAptive Core so exit!
    sdbbp                               // Failed assertion: not interAptiv.
```

MIPS

Now the code needs to determine if it is running on a Multi-Core system.

+ It does this by reading the CP0 Processor ID register into GPR 25.

+ The code extracts the Core ID and Implementation bits

+ Then compares them with the values for the specific core it was compiled for. In this case it was compiled for a interAptiv core. If the test passes it branches to setting up the Coherency manager registers.

If it doesn't pass it exits to the debugger.

set_gpr_boot_values.S - set_gpr_boot_values

For a Coherent Processor:

```
is_cps:
li      r11_is_cps, 1

// Verify that we can find the GCRs.
la      a1, GCR_CONFIG_ADDR         // KSEG1(GCRBASE)
ins     a1, $0, 29, 3               // Convert KSEG1 to physical address.
lw      a0, GCR_BASE(a1)            // read GCR_BASE
ins     a0, $0, 0, 15               // Isolate physical base address of GCR
beq     a1, a0, gcr_found
nop
sdbbp   // Can't find GCR. RTL config override of MIPS default?
```

MIPS

If the code determined that it is executing on a Coherent Processor then it sets GPR 11 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. The Coherent processing system contains a structure that determines the configuration of the system. This structure is called the Global Control Block. This structure contains registers that can be read to determine the configuration of elements within the CPS. These registers are called the Global Control Registers or GCRs. Many of the registers can also be written changing 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 one that is stored within the block itself located in the GCR Base register. The given address is set by a #define. 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.

+ The value in the GCR Base register is a physical address so before the code compares the given value, it needs to convert it to a physical address. That's done by simply clearing the top 3 bits using the insert instruction and GPR 0. This line of code takes the first 3 bits of GPR 0, which is always 0, and inserts them starting at bit 29 into GPR register a1.

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

+ The starting address of the GCRs is located in the memory map on a 32K Byte boundary so the lower 15 bits of it 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 GPR a0.

+ The code checks to make sure the 2 gprs are equal and branches to the gcr_found function if they are or issues a debug break instruction to stop execution.

set_gpr_boot_values.S - set_gpr_boot_values

For a Coherent Processor (continued):

```
gcr_found:
   // Every vpe will set up the following to simplify resource initialization.
    li     r22_gcr_addr, GCR_CONFIG_ADDR
    lw    r8_core_num, (CORE_LOCAL_CONTROL_BLOCK + GCR_CL_ID) (r22_gcr_addr)
    lw   a0, GCR_CONFIG(r22_gcr_addr)                  // Load GCR_CONFIG
   ext   r19_more_cores, a0, PCORES, PCORES_S     // Extract PCORES

done_init_gpr:
    jr     ra
    nop
END(set_gpr_boot_values)
```

MIPS

13

Now that the code has determined it has valid GCRs it will save the address in GPR 22.

+ The code stores the GCR_CL_ID in GPR 8. 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 Core-Local Control Block. The Core-Local Control Block is at offset 2000 hex from the GCR Base address and the GCR_CL_ID is located at offset 28 hex within the Block. Putting those together you get 2028 hex offset from the GCR Bass address.

+ The code now will save the total number of 1004K 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 GPR 19.

+ We are now done with the init_gpr function and the code returns

initc_cp0.S - init_cp0

- The code continues and calls the init_cp0

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

  - init_cp0 is Located in common/init_cp0.S

```
LEAF(init_cp0)

    //# Initialize Status
    li      v1, 0x00400404          // M_StatusIM | M_StatusERL | M_StatusBEV)
    mtc0    v1, C0_STATUS           / / write C0_Status
```


The code calls the init_cp0 function.

The init_cp0 function is much the same as it is for a single core. So I won't cover it in detail.

The first thing it does is set the status register to a known value for this board.

initc_cp0.S - init_cp0

```
// Initialize Watch registers if implemented.
mfc0    v0, C0_CONFIG1      // read C0_Config1
ext     v1, v0, 3, 1        // extract bit 3 WR (Watch registers implemented)
beq     v1, zero, done_wr
li      v1, 0x7             // (M_WatchHiI | M_WatchHiR | M_WatchHiW)
// Clear Watch Status bits and disable watch exceptions
mtc0    v1, C0_WATCHHI      // write C0_WatchHi0
mfc0    v0, C0_WATCHHI      // read C0_WatchHi0
bgez    v0, done_wr         // Check for bit 31 (sign bit) for more Watch registers
mtc0    zero, C0_WATCHLO    // write C0_WatchLo0
mtc0    v1,  C0_WATCHHI, 1  // write C0_WatchHi1
mfc0    v0, C0_WATCHHI, 1   // read C0_WatchHi1
bgez    v0, done_wr         // Check for bit 31 (sign bit) for more Watch registers
mtc0    zero, C0_WATCHLO,1 // write C0_WatchLo1
```

MIPS

15

Next the code checks for watch registers. If the WR bit, bit 3 in the CP0 Config1 register, is set to 1, then the core has at least 1 watch register.

Set GCR v1 to the initialization value for the watch registers. This will clear all watch conditions.

+ Use GCR v1 to clear all watch conditions.

+ read the watch hi register and check to see if the M bit, bit 31 is set indicating there is at least another watch register and branch to done wr if not set.

+ In the branch delay slot clear the watch lo register

+ Continue until there are no more watch registers

initc_cp0.S - init_cp0

- Clear Cause register

```
//# Clear WP bit to avoid watch exception upon user code entry, IV, and software interrupts.
   mtc0   zero, C0_CAUSE      # C0_Cause: Init AFTER init of CP0 WatchHi/Lo registers.
```

- Clear Compare register

```
//# Clear timer interrupt.
   mtc0   zero, C0_COMPARE// write C0_Compare

jr     ra
nop
END(init_cp0)
```

MIPS

16

Clearing the CP0 Cause register to avoid spurious interrupts once interrupts are enabled.

+ Clear the CP0 Compare register to clear timer interrupts.

+ Then the code returns to start.S

start.s – Call init_tlb?

```
// Determine if we have a TLB
mfc0    v1, C0_CONFIG       // read C0_Config
ext     v1, v1, 7, 3        // extract MT field
li      a3, 0x1             // load a 1 to check against
bne     v1, a3, done_tlb    // no tlb?
nop
// determined if this is VPE 0 so skip shared TLB check
beqz    r9_vpe_num, do_tlb  // VPE 0?
nop
```

NOTE: TLB init is covered in the Basic MIPS TLB section and will not be repeated here

MIPS

17

Back in start.S

This code Reads the config register and checks the TBL-MMU field by extracting it to v1then 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.

+ Next the code checks the VPE number that is executing the code and if it is VPE 0 it branched to do_tlb because if there is a TLB VPE 0 will always have a TLB to initialize.

init_tlb.S - init_tlb

- Initialize the TLB

```
    mfc0   v0, C0_CONFIG1                  // read C0_Config1
    nop

  start_init_tlb:
    // Config1MMUSize == Number of TLB entries - 1
    ext    v1, v0, CFG1_MMUSSHIFT, 6       // extract MMU Size
    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
    li     a0, 0x80000000
```

MIPS

18

Read the Config 1 register that contains the TLB information

+ Extract the MMU size field which gives us the highest number TLB entry into v1. This will be used as the first index into the TLB.

+ Then to clear all entry registers so we will initialize their TLB entire fields to 0

To do this we use the same move to Coprocessor zero instruction using the general purpose register 0 which always contains a 0 value and move to the corresponding Coprocessor 0 register

+ Load a0 with the address to be placed in the entry. Note it will be invalid but will insure that the TLB does not have duplicate entries.

init_tlb.S - init_tlb

- Loop through the TLB

```
next_tlb_entry_pair:
   mtc0   v1, C0_INDEX          // write C0_Index
   mtc0   a0, C0_ENTRYHI        // write C0_EntryHi
   ehb
   tlbwi
   // Add 8K to the address to avoid TLB conflict with previous entry
   add     a0, (2<<13)
   bne     v1, zero, next_tlb_entry_pair
   add     v1, -1
done_init_tlb:
    jr       ra
    nop
END(init_tlb)
```

MIPS

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.

+ Remember we stored the highest numbered TLB entry to v1. We will use it here to decrement through the TLB entries from the highest to the lowest and use it to program the TLB entry index.

The code uses the move to Coprocessor 0 instruction to copy the contents of v1 to Co Processor 0 register which is the index register. The index is the TLB entry to be written.

+ The ehb instruction is used to make sure all the writes to coprocessor 0 have been completed before writing the TLB entry.

+ Now the TLB Write Index Instruction is used to write the TLB entry

+ the address written to the TLB entry is advanced so each will be different.

+ Next is the branch instruction. Here the code compares the TLB index value that is in v1 with 0 and if they are not equal the code branches back to next_tlb_entry_pair

+ The last instruction is in the branch delay slot and will always be executed It use the add instruction to increment the index value in v1 by a -1

+ Once the loop is complete, the code returns to start.S.

This section describes the Boot-MIPS code and walks you through the booting of a coherent processing system.

init_gic.S - init_gic

- GIC Address Map

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

MIPS

21

The init_gic code pokes around getting information and setting bits in the Global Interrupt Controller or GIC for short.

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

The GIC is divided into sections:

The first section 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 local section which starts at the Base address plus 8 thousand hex. This is the section in which interrupts local to a VPE are registered, masked, and assigned to a particular interrupt pin.

The "local" CPU can access the registers of another CPU core by using the Core-Other address spaces. Software must write the VPE-Other Addressing Register before accessing these address spaces. The value of this register is used by hardware to index the appropriate subset of the control registers for the other core.

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.

init_gic.S - init_gic

- Global Interrupt Controller Initialization

```
done_tlb:
    la      a2,    init_gic    // Configure the global interrupt controller.
    jalr    a2
    nop
```

- Located in cps/init_gic.S

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

MIPS

22

Back in start.S the code calls init_gic.

+ Then in the init_gic function the code checks to see if it needs to initialize the global interrupt controller. It checks GPR 11 and if it is not set then this is not a Coherent Processing system so the code will skip the GIC initialization.

init_gic.S - init_gic

GCR GIC Status Register

Register Fields		Description	Read/Write	Reset State
Name	Bits			
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0	R	0
GIC_EX	0	If this bit is set, the GIC is connected to the CM.	R	1

```
la      a1, GCR_GIC_STATUS + GCR_CONFIG_ADDR  // Read GCR_GIC_STATUS
lw      a0, 0(a1)
ext     a0, a0, GIC_EX, GIC_EX_S              // Isolate  GCR_GIC_STATUS[GIC_EX].
beqz    a0, done_gic                          // If no gic then skip.
nop
bnez    r23_cpu_num, init_vpe_gic             // Only core0 vpe0 inits shared portion.
nop
```

MIPS

23

To check to be sure there is a GIC the code reads the Global Control Blocks GIC Status register

+ 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.

+ There are 2 parts of the GIC that need to be initialized, a Shared Part that needs only to be initialized by CPU 0 and a local part the needs to be initialized by each processor. This code will do the shared part only if this is CPU 0 so it checks GPR 23 the CPU number and will skip the shared section if it is not 0.

init_gic.S - init_gic

- Enable the GIC

```
li      a1, GCR_CONFIG_ADDR + GCR_GIC_BASE      // Locate GIC Config Register
li      a0, GIC_P_BASE_ADDR | 1                 // Physical address + enable bit
sw      a0, 0(a1)
```

GCR_GIC_BASE Offset 0x0080				
Register Fields		Description	Read/Write	Reset State
Name	Bits			
GIC_BaseAddress	31 - 7	The base address of the 128KB Global Interrupt Controller block	R/W	Undefined
GIC_EN	0	Setting to 1 enables GIC	R/W	0

MIPS

24

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

As you can see from the table the address is on a 128K boundary so the lower 7 bits will always be 0. This leaves space for additional information in the register. The GIC_EN field controls the enabling of the GIC.

+ Code loads a0 with the address of GIC: Note this is a Physical address ored with 1 to enable the GIC.

+ Next stores the value to the GCR_GIC_BASE register.

init_gic.S - init_gic

Register Fields		Description GIC Configuration Register	Read/Write	Reset State
Name	Bits			
COUNTSTOP	28		R/W	0
COUNTBITS	27:24		R	0x8
NUMINTERRUPTS	23:16	Value is fixed by customer at IP configuration time.	R	IP Config

```
// 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
    ext     a0, NUMINTERRUPTS, NUMINTERRUPTS_S  // NUMINTERRUPTS (actually slices - 1)
    li      a3, 4
    beq     a0, a3, configure_slices
    nop
    sdbbp                                   // Failed assertion that gic implements 40 external interrupts.
```

MIPS

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.

+ To do this the code loads the define GIC_BASE_ADDR which is the starting address of the Shared section of the GIC into a1.

+ The Shared Configuration register shown above is located at offset 0. The code loads its value into a0.

+ Then the code extracts the number of interrupt groups.

+ The code loads the expected value into a3. So it 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.

init_gic.S - init_gic

- **GIC Shared Section Global Interrupt Reset Mask**
  - Reset and Disable interrupt

```
configure_slices:
    li      a0, 0xff000000 // Disable Interrupts 24 – 31 (top 8 bits)
    sw      a0, GIC_SH_RMASK31_0(a1)
```

Register Offset	Interrupt Sources in Register	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 are 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	

MIPS

26

There is a convention in MIPS Linux to use the last 16 interrupt sources for inter processor interrupts. In the system we are configuring those are interrupts 24 through 39. The system could contain up to 8 processors.

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 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 2 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 interrupt sources 24 – 31 by writing a 1 to bits 24 – 31 in the first Global interrupt Reset Mask Register.

init_gic.S - init_gic

- **Global Interrupt Trigger Type Registers**

sw a0, GIC_SH_TRIG31_0(a1) // edge sensitive 24..31 (for Interprocessor Interrupts)

The code uses a0 to write 1's to bits 24 through 31 of the first interrupt Trigger Register setting those interrupts to be edge sensitive

Register Offset	Interrupt Sources in Register	Description
0x0180	0 - 31	Edge or Level triggered 0x0 - Level 0x1 - Edge 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.
0x0184	32 - 63	
0x0188	64 - 95	
0x018c	96 - 127	
0x0190	128 -159	
0x0194	160 - 191	
0x0198	192 - 223	
0x019c	224 - 255	

MIPS

27

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 signaling. If the bit is cleared the interrupt is level signaling. The offset of the Global Interrupt Trigger Type Registers in the GIC Section is hex 180.

+ 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 which is needed to support inter-processor interrupts.

init_gic.S - init_gic

- **Global Interrupt Polarity Registers**

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

The code a0 to write 1's to bits 24 through 31 of the first interrupt Polarity Register making interrupts 24-31 rising edge sensitive for Interprocessor Interrupts.

Register Offset	Interrupt Sources in Register	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 are 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	

MIPS

28

Similar to the Reset Mask Registers there is a set of registers that configures the polarity of the interrupt. The polarity determines how the interrupt is signaled to the core. Interrupts can be level or edge sensitive . If level sensitive setting the interrupts 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 to support inter-processor interrupts.

init_gic.S - init_gic

- **Global Interrupt Set Mask Registers**
  - Enables interrupts

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

The code uses a0 to write 1's to bits 24 through 31 of the first interrupt mask register

Register Offset	Interrupt Source in Register	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	

MIPS

29

There is a set of registers that correspond 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.

+ The code sets the same bits still in a0 as it did for the Polarity registers to set the enable bits for interrupts 24 through 31.

init_gic.S - init_gic

- **Configuring interrupts 32 through 39**

  Then the code disables the interrupts, Sets the **Polarity Registers ,** Sets the **Trigger Register** And last enables the interrupts for interrupt pins 32 through 39.

```
// Then interrupts 32... 39 (lower 8 bits of the registers)
    li      a0, 0xff
    sw      a0, GIC_SH_RMASK63_32(a1)    // (disable     32..63)
    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)
```

MIPS

30

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 32 through 39 bits.

+ Then the code disables the interrupts,

Sets the Polarity Registers

Sets the Trigger Register

And last enables the interrupts

init_gic.S - init_gic

- Global Interrupt Map to VPE Register

Register Offset	Interrupt Map Src0 to VPE Register numbers	Description
0x2000	Interrupt Source 0, processors 0 - 31	Assigns this interrupt source to a particular VPE. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources and the number of VPEs. These are read/write bits.
0x2004	Interrupt Source 0, processors 32 - 63	
0x2020	Interrupt Source 1, processors 0 - 31	
0x2024	Interrupt Source 1, processors 32 - 63	
0x2040	Interrupt Source 2, processors 0 - 31	
0x2044	Interrupt Source 2, processors 32 - 63	
……		
0x3fe0	Interrupt Source 255, processors 0 - 31	
0x3fe4	Interrupt Source 255, processors 32 - 63	

MIPS

31

Next the code will configure which Processor a particular interrupt will be assigned to. To do this the GIC has registers for each interrupt source. Each bit in those registers corresponds to a processor in the multi core system.

For example for interrupt source 1, bit 0 would assign the interrupt to processor 0. The current schema supports up to 64 different processors so there are 2, 32 bit registers for each interrupt. To allow for future expansion the registers are spaced 32 bytes apart.

Lets look at the table. The Interrupt Map to VPE Registers are in the GIC shared section and start at offset 2000 hex. 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 20 hex so its registers are at 2020 and 2024 hex and so on.

init_gic.S - init_gic

- **Global Interrupt Map to VPE Register Setting**
  - This code assigns 2 interrupt sources to each processing unit in the system. It does this using a0 which is set up with a processor unit number.

```
li      a0, 1                   // set bit 0 for processor 0 or for MT vpe0
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 24) (a1) // Source 24 map to 31
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 32) (a1) // Source 32 map to 39
```

- **Calculating GIC_MAP_TO_VPE register address**
  - A1 – GIC_BASE_ADDR
  - #defines found in cps/cps.h

```
#define GIC_SH_MAP0_VPE31_0 0x2000 // offset from GIC BASE
#define GIC_SH_MAP_SPACER 0x20     // space between registers
```

  - GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 24) (a1)

MIPS 32

+ This code assigns 2 interrupt sources to each processor in the system.

+ The corresponding bit number for a processor is stored in a0. a0 is primed with a 1 for the first processor, processor 0.

+ Then the code stores the value in a0 to the appropriate MAP registers. The interrupts sources are divided into 2 groups, one group from 24 to 31 and the other from 32 through 39. The code will take one interrupt source from each group and program it to a processor.

+ Lets go over how the address for the GIC_MAP_TO_VPE register is calculated.

+ Previously the GIC BASE ADDR was stored into a1 remember this is a #define in boot.h.

+ The code also uses #defines which come from cps/cps.h

+ The GIC_SH_MAP0_VPE31_0 is the base offset of the GIC_MAP_TO_VPE registers

+ The GIC_SH_MAP_SPACER is the size or space between each interrupts GIC_MAP_TO_VPE registers.

+ The calculation is the base address of the GIC registers offset by the base offset of the GIC_MAP_TO_VPE registers plus the space between the interrupt registers times the interrupt number.

init_gic.S - init_gic

- **Global Interrupt Map to VPE Register Setting continued:**

```
sll     a0, a0, 1       // set bit 1 for processor 1 or for MT vpe1
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 25) (a1) // Source 25 map to 1
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 33) (a1) // Source 33 map to 1
sll     a0, a0, 1       // set bit 2 for processor 2 or for MT vpe2
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 26) (a1) // Source 26 map to 2
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 34) (a1) // Source 34 map to 2
sll     a0, a0, 1       // set bit 3 for processor 3 or for MT vpe3
.
.       and so on ……
.
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 31) (a1) // Source 31 map to 7
sw      a0, GIC_SH_MAP0_VPE31_0 + (GIC_SH_MAP_SPACER * 39) (a1) // Source 39 map to 7
```

MIPS

33

The code continues to program each processing unit in turn.

+ To increment the processor number stored in a0 to the correct bit position it will shift a0 to the left one bit.

+ and program the appropriate map registers for the next interrupt pair

init_gic.S - init_gic

- **Per Processor Element initialization**

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

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 CPU's COP0 IntCtlIPFDCI register field.	IP config value
SWINT_ROUTABLE	3	If this bit is set, the CPU SW Interrupts are routable within the GIC. If this bit is clear it is routed back to the CPU directly.	IP config value
PERFCOUNT_ROUTABLE	2	If this bit is set, the CPU Performance Counter Interrupt is routable within the GIC. If this bit is clear, it is hardwired to one of SI_Int pins as described by the CPU's COP0 IntCtlIPPCI register field.	IP config value
TIMER_ROUTABLE	1	If this bit is set, the CPU 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 CPU'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

MIPS

34

This next section of the code will initialize the per-processor elements of the GIC.

+ The section of registers being initialize is called VPE-Local and is located at GIC offset 8000 hex.

+ The code reads the Local Interrupt Control Register into a3. The code will be using some of the values from this register.

init_gic.S - init_gic

- **Check the Timer Interrupt Source**

```
map_timer_int:
    ext     a0, a3, TIMER_ROUTABLE, TIMER_ROUTABLE_S
    beqz    a0, map_perfcount_int
    nop
```

Register Fields		Description of the Local Interrupt Control Register (GIC_VPEi_CTL) 0x8000	Reset State
Name	Bits		
TIMER_ROUTABLE	1	If this bit is set, the CPU 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 CPU's COP0 IntCtlIPTI register field.	IP config value

MIPS

35

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 had previously read into a3.

+ Then it checks to see if it's set. If it is not set the timer interrupt is not routable and the code will branch around routing it.

init_gic.S - init_gic

- **Map the Timer Interrupt Source**

```
li      a0, 0x80000005                  // map to pin 5 for timer routing
sw      a0, GIC_COREL_TIMER_MAP(a1)
```

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	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

MIPS

36

The code sets up a0 with the en-coding that is used to route the local CPU timer interrupt to the desired processor pin. a0 is written with bit 31 set and a 5 in the Map field. This will map the local Core's timer interrupt to the current Processor's units interrupt pin 5.

+ This value is stored to the GIC Local CPU Timer Map-to-Pin Register.

Boot-MIPS init_gic

- **Check The Performance Counter Interrupt Source**

```
map_perfcount_int:
    ext         a0, a3, PERFCOUNT_ROUTABLE, PERFCOUNT_ROUTABLE_S
    beqz  a0, done_gic
    nop
```

Register Fields		Description of the Local Interrupt Control Register (GIC_VPEi_CTL) 0x8000	Reset State
Name	Bits		
PERFCOUNT_ROUTABLE	2	If this bit is set, the CPU Performance Counter Interrupt is routable within the GIC. If this bit is clear, it is hardwired to one of SI_Int pins as described by the CPU's COP0 IntCtlIPPCI register field.	IP config value

MIPS

37

As it did with the Timer interrupt, 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 is not set the performance counter interrupt is not routable and the code will branch around routing it.

Boot-MIPS init_gic

- **Map The Performance Counter Interrupt Source**

```
    li      a0, 0x80000004                  // map to pin 4 for performance routing
    sw      a0, GIC_COREL_PERFCTR_MAP(a1)
done_gic:
    jr      ra
    nop
```

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	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

MIPS

38

The code sets up a0 with the in-coding that will map the performance counter interrupt. a0 is written with bit 31 set and a 4 in the Map field. This will map the local Core's Performance Counter interrupt to the current Processor's interrupt pin 4.

+ This value is stored to the GIC Local CPU Performance counter Map-to-Pin Register.

+ This ends the GIC configuration.

start.S - init_common_resources

- **For MT Cores VPE 0 Check**
  - Only need to continue if VPE 0 is executing

```
bnez    r9_vpe_num, init_done # If we are not a vpe0 then we are done.
nop
```

- **Disable L2/L3 caches**

```
init_core_resources:    // We are a vpe0
  la      a2,    disable_L23   // Disable L2/L3 caches
  jalr    a2
  nop
```

MIPS

39

Back in start.S code is done initializing each Processer. It continues checking to see if it is VPE 0 or processor 0. If it is not then it is done with the initialization and will branch to init done.

+ The code is about to initialize the icache and then enable caching. It initializes the Dcache and the L2/3 caches after enabling the caches because the code will run faster for those functions once the Icache is enabled. For all this to work correctly the L2/3 caches must be disabled until they are initialized so at this point the disable_L23 function is called to do that.

init_L2_CM2.S - disable_L2

Register Fields		GCR Base Register (GCR_BASE)	Reset State
Name	Bits		
CCA_DEFAULT_ OVERRIDE_ENABLE	4	If CCA_DEFAULT_OVERRIDE_ENABLE is set to 1 and CM_DEFAULT_TARGET is set to Memory, then transactions with addresses do not map to any region will have a CCA value set to CCA_DEFAULT_ OVERRIDE_VALUE when driven to L2/Memory	0

```
LEAF(disable_L2)
    bnez    r8_core_num, done_disable_L2    // Only done from core 0.
    lw      a0, 0x0008(r22_gcr_addr)        // Read GCR_BASE
    li      a3, 0x50                        // Enable CCA and set to uncached
    ins     a0, a3, 0, 8                    // Insert bits
    sw      a0, 0x0008(r22_gcr_addr)        // Write GCR_BASE
done_disable_L2:
      jr    ra
      nop
END(disable_L2)
```

MIPS

40

To disable the L2 cache the CCA Override Enable bit will be set. This bit is bit 4 of the GCR Base Register

+ The code checks to see if it is executing on a coherent processing system and if it isn't it will branch around the next piece of code and assume there is no L2 cache.

+ The code reads the GCR Base register.

+ The next 3 lines of code are used to enable CCA Override and set the L2 cache CCA to non cached..

start.S - init_core_resources: L1 caches

- Initialize L1 Icache

```
la      a2,    init_icache // Initialize the L1 instruction cache. (Executing using I$ on return.)
jalr    a2
nop
```

- Turn caching on
  - The function that changes the CCA of KSEG 0 must be done from KSEG1 the uncached segment.

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

NOTE: Code assumes legacy boot mode (not EVA boot)!

MIPS

41

Next The init_icache function is called. I'm not going to go over the I cache initialization code in this class because it is already covered in the basic software training class.

Now that the Icache has been initialized we can take advantage of it and change the CCA of KSEG0 to be cached. One important thing to note is the changing of the CCA of KSEG0 must be done from KSEG1 which is a never cache segment.

+ The code loads the address of the change_k0_cca function

+ and then changes it to a address in the KSEG1 segment by inserting a 1 to bit 29.

+ When the jalr is executes it will jump to this uncached segment change the CCA of KSEG0 then jump back to this code an start executing from the Icache.

init_caches.S – change_k0_cca

```
LEAF(change_k0_cca)
   // NOTE! This code must be executed in KSEG1 (not KSGE0 uncached)
    // Set CCA for kseg0 to cacheable
   mfc0   v0, C0_CONFIG                  // read C0_Config
   beqz   r11_is_cps, set_kseg0_cca
   li     v1, 3                          // CCA for coherent cores
   li     v1, 5                          // CCA for all others

set_kseg0_cca:
    ins    v0, v1, 0, 3                  // instert K0
   mtc0   v0, C0_CONFIG                  // write C0_Config
   jr.hb  ra
   nop
END(change_k0_cca)
```

MIPS

42

The change K0 CCA function is a function common to all MIPS cores so there is a little run time decision to be made to set the correct CCA. The code uses a cache writeback CCA of 3 for non coherent cores and a cached coherent writeback CCA of 5 for coherent cores.

+ To do this is simply checks to see if the r11_is_cps register is not set.

+ If it is not set it stores the non coherent CCA to v1 in the branch delay slot and branches to the setting of the config register code.

+ If it is set the code falls through the branch and stores the CCA value for coherent cores to v1 and continues to the setting of the config register.

+ The value of v1 is inserted into bits 0 through 3 of the stored value of the config register and then written to the the register.

+ The code uses a jr.hb instruction the will clear the hazard barrier to make sure the config registers write has completed and then jump back to the calling function.

start.S

- Initialize the Dcache

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

- Initialize Inter-Thread Communications

```
la      a2,    init_itc    // Initialize Inter-Thread Communications unit
jalr    a2
nop
```

- If not core 0 branch to done

```
bnez    r8_core_num, init_sys_resources_done
nop
```

MIPS

43

Next in statrt.S is a call to initialize the Dcache that is also covered in the programming a MIPS Core Software course and I'm not going to cover it here.

+ The init_itc is a place holder for you to put any code you need to it initialize the ITC for your system.

+ At this point the code will continue initializing system wide shared resources. Since this only needs to be done by only one processor the code tests to see if it is executing on core 0 if it is not it will jump to done. If it is it will continue the boot and initialize system resources.

Cluster Power Controller Initialization

- start.S call to initialize the Cluster Power controller

```
init_sys_resources:     // We are core0 vpe0.
    la      a2,     init_cpc    // Initialize the CPS CPC (Cluster Power Controller.)
    jalr    a2
    nop
```

- init_cpc is located in cps/init_cpc.S

```
LEAF(init_cpc)
    beqz  r11_is_cps, done_init_cpc                 // Skip if non-CPS.
    nop
    lw    a0, GCR_CPC_STATUS(r22_gcr_addr)          // Read GCR_CPC_STATUS
    andi  a0, 1                                     // CPC_EX is bit 0
    beqz  a0, done_init_cpc     // Skip if CPC is not implemented (CPC_EX not set)
    move  r30_cpc_addr, zero
```

MIPS

Back in start.S, in the init_sys_resources section, the init_cpc is called to initialize the cluster power controller.

+ In the init cpc function the code first checks to see if this is a coherent processing system by checking r11 is cps. If it's not set then it will not have a CPC so it will skip to the end and return.

If this is a CPS then the code checks for a Cluster Power Controller. It does this by checking the Cluster Power Controller Status Register. This register is located within the Global Configuration Registers at offset f0 hex.

+ The code uses the previously stored address of the GCR base and the define GCR CPC STSTUS as an offset to read 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.

+ The code singles out bit 0 of the GCR_CPC_STATUS value

+ If it's 0 then there is no CPC and it branches around this code and returns to the initialization function.

+ In the branch delay slot we insure GPR 30 is clear to indicate we don't have a CPC.

init_cpc.S - init_cpc

Register Fields		GCR - Cluster Power Controller Base Address Register (offset 0x88)	Reset State
Name	Bits		
CPC_Base Address	31-15	This field sets the base address of the 32K Cluster Power Controller.	Undefined
CPC_EN	0	If this bit is set, the address region for the CPC is enabled. This bit can not be set if 1 CPC_EX = 0, indicating that a CPC is not attached to the CM.	0

- **Write Address Of CPC into the GCR**

```
  li      a0, CPC_P_BASE_ADDR                 // Locate CPC (configuration dependent value)
  sw      a0, GCR_CPC_BASE(r22_gcr_addr)      // Write CPC_BASE address to GCR
  li      r30_cpc_addr, CPC_BASE_ADDR         // Maintain address of CPC register block.
done_init_cpc:
  jr      ra
  nop
  END(init_cpc)
```

MIPS

45

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 Global Configurations Registers.

+ The code uses the known value of where the CPC is within the system and writes that to the Cluster Power Controller Base Address Register. Note: This is a physical address.

+ Then the code stores this address for later use in GPR 30 using the KSEG1 equivalent address and it is done setting up the CPC.

init_cm.S init_cm

- start.S call to init_cm

```
la      a2,    init_cm    // Initialize the CPS CM (Coherence Manager.)
jalr    a2
nop
```

- Initialize the Coherency Manager
  - Located in cps/init_cm.S

```
LEAF(init_cm)
   beqz   r11_is_cps, done_cm_init
   nop
```

MIPS

The code now calls init cm to initialize the coherency 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.

init_cm.S - init_cm

Register Fields		Cluster Power Controller Global CSR Access Privilege Register	Reset State
Name	Bits		
CM_ACCESS_EN	7-0	Each bit in this field represents a power domain CPU. If the bit is set, that requester is able to write to the CPC registers (this includes all registers within the Global, Core-Local and Core-Other blocks. If the bit is clear, any write request from that requestor to the CPC registers (Global, Core-Local, Core-Other) will be dropped. They are read/write bits.	0xff

- **Allow each core access to the CM registers**

```
li      a0, 2                          // Start building mask for cores in this cps.
sll     a0, a0, r19_more_cores
addiu   a0, -1                         // Complete mask.
sw      a0, GCR_ACCESS(r22_gcr_addr)// GCR_ACCESS
```

MIPS

47

The lower 8 bits of the Global CSR Access Privilege Register controls the write access to the GCR registers by a processor. If a bit is set then the processor unit can change the GCR.

The code here is rather clever

+ first it loads a 2 into a0

+ then the code shifts the 2 to the left by the number of processors minus one. For example is there were 4 processor system then GCR 19 would contain a 3. 2 shifted left by 4 is 16 or 10 hex.

+ now the code subtracts 1 so again assuming 4 processors 16 – 1 is 15 or F hex so now we have all four lower bits set

+ The value is written to the Global CSR Access Privilege Register which will now allow all 4 processor units to change the GCR

init_cm.S - init_cm

Register Fields		Global Config Register	Reset State
Name	Bits		
NUMIOCU	11-8	Total number of IOCUs in the system. Note: only 0 or 1 IOCU is currently supported.	IP Config value

Encoding	Meaning
0x0	No! IOCU
0x1	1 IOCU
…	Number… IOCU

- **Check For IOCU (if no IOCU we don't have to init regions)**

```
lw      a0, GCR_CONFIG(r22_gcr_addr)      // Load GCR_CONFIG
ext     a0, a0, NUMIOCU, NUMIOCU_S        // Extract NUMIOCU.
beqz    a0, done_cm_init
```

MIPS

48

This code checks to see if there is an IOCU.

+ It does this by loading the GCR configuration register into a0

+ extracting the NUMIOCU field as shown in the table above

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

init_cm.S - init_cm

Register Fields		CM Region[0 - 5] Address Mask Register (GCR_REGn_MASK)	Reset State
Name	Bits		
CM_REGION_ TARGET	1-0	Maps this region to the specified device	0

- **Disable the CM regions**

```
lui   a0, 0xffff
sw    a0, GCR_REG0_BASE(r22_gcr_addr)// GCR_REG0_BASE
sw    a0, GCR_REG0_MASK(r22_gcr_addr)// GCR_REG0_MASK
sw    a0, GCR_REG1_BASE(r22_gcr_addr)// GCR_REG1_BASE
sw    a0, GCR_REG1_MASK(r22_gcr_addr)// GCR_REG1_MASK
sw    a0, GCR_REG2_BASE(r22_gcr_addr)// GCR_REG2_BASE
sw    a0, GCR_REG2_MASK(r22_gcr_addr)// GCR_REG2_MASK
sw    a0, GCR_REG3_BASE(r22_gcr_addr)// GCR_REG3_BASE
sw    a0, GCR_REG3_MASK(r22_gcr_addr)// GCR_REG3_MASK
```

Encoding	Meaning
0x0	Disabled
0x1	Memory
0x2	IOCU

MIPS

If there is an IOCU then the code will disable the IOCU regions.

+ The code loads an upper immediate value into a0 this value sets bits 16 through 31 and clears bits 0 through 15. As you can see from the table 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.

init_cm.S - init_cm

- **All Done With init_cm**

```
done_cm_init:
    jr     ra   # return to init_sys_resources
    nop
```

- **Initialize The Memory Controller**
  - start.s:

```
#ifdef DENALI
    la   a2,    init_mc_denali// Initialize the ROC-it2 MC (Memory Controller.)
    jalr a2
    nop
#endif
```

  - Implementation dependent

MIPS

50

This completes the CM initialization and the code returns to init_sys_resources

+ In start.S a call is made to initialize the memory controller. You would need to place your own code in the init_mc function to initialize the specific controller for your system.

copy_c2_ram.S – common/copy_c2_ram

- start.S call to copy_c2_ram

```
la      a2,    copy_c2_ram // Copy "C" code and data to RAM and zero bss (uncached.)
jalr    a2
nop
```

- (Located in common/copy_c2_ram)

```
LEAF(copy_c2_ram)
 //# Copy code and read-only/initialized data from FLASH to (uncached) RAM.
   la      a1_temp_addr, _zap1                  # start of code in flash (from)
   ins     a1_temp_addr, r1_all_ones, 29, 1     # convert to uncached address
   la      a2_temp_dest, _ftext_ram             # start of code area in ram (to)
   ins     a2_temp_dest, r1_all_ones, 29, 1     # convert to uncached address
   la      a3_temp_mark, _edata_ram             # ending data address in ram
   ins     a3_temp_mark, r1_all_ones, 29, 1     # convert to uncached address
   beq     a2_temp_dest, r7_temp_mark, zero_bss # anything to copy?
   nop
```

MIPS

51

The code now calls the copy C to ram function. This will copy the C code part of the application and zero out the uninitialized global variable section, bss.

+ The copy C 2 ram function starts by putting the first address of the “C” code’s text section into a1_temp_addr. _zap1 is created in the Linker script malta_Ram.ld. This address is the start of the “C” code in flash that will be copied to RAM.

+ The _zap1 address is a cached address in the KSEG0 region. We want to make sure the code gets copied to main memory. What this code does is to convert the KSEG0 address into a KSEG1 uncached address by inserting a 1 into bit 29. This turns the top byte from an 8 to an A.

+ Next the code stores the _ftest_ram value into a2_temp_addr. _ftest_ram is also created in the linker file. It is the start of where the “C” code section will be copied to in main memory. It is also converted to a KSGE1 address by inserting a 1 into bit 29.

+ The _edata_ram is stored into a3_temp_addr. _edata_ram is created in the linker file and is the address in main memory where the initialized data section ends. The code will use this address to end the copy of the code and initialized data sections.

+ 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.

copy_c2_ram.S - copy_c2_ram

```
next_ram_word:
    lw     data_a0, 0(source_addr_a1)
    sw     data_a0, 0(destination_addr_a2)
    addiu  destination_addr_a2, 4
    bne    end_addr_a3, destination_addr_a2, next_ram_word
    addiu  source_addr_a1, 4

zero_bss:
    LA     destination_addr_a2, _fbss
    ins    destination_addr_a2, all_ones_s1, 29, 1
    LA     end_addr_a3, _end
    ins    end_addr_a3, all_ones_s1, 29, 1
    beq    destination_addr_a2, end_addr_a3, copy_c2_ram_done
    nop
```

MIPS

52

The label next_ram_word will be used as a loop back point for the copy loop that follows.

+ The copy is simply reading from the location where the “C” code and data is stored in flash

+ and writing it to it’s destination address in RAM.

+ The source and destination addresses are incremented by 4, the number of bytes in a word

+ and the code checks to see if it still has more to copy by using a3 which is the end address and the current destination address.

+ Now the code turns its attention to the uninitialized variable section also known as the bss section. 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 uncashed KSGE1 addresses.

+ then checks to see if there is anything to clear by seeing if they are equal.

copy_c2_ram.S - copy_c2_ram

```
next_bss_word:
   sw     zero, 0(destination_addr_a2)
   addiu  destination_addr_a2, 4
   bne    destination_addr_a2, end_addr_a3, next_bss_word
   nop

copy_c2_ram_done:
   jr     ra
   nop
```

MIPS

53

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 main memory.

+ it then adds 4 bytes to the destination address

+ checks to see if it is at the end of the clear by comparing it to the end address stored in a1 and loops back if it is not.

+ the code is done with the copy and returns.

This section describes the Boot-MIPS code and walks you through the booting of a coherent processing system.

release_mp.S - release_mp

- Release the core for Multi processing
- start.S call to release_mp

```
la      a2, release_mp          // Release other cores to execute this boot code.
jalr    a2
nop
```

- Located in cps/release_mp.S

```
LEAF(release_mp)
    blez    r19_more_cores, done_release_mp     // If no more cores then we are done.
    li      a3, 1
    beqz    r30_cpc_addr, release_next_core     // If no CPC then use GCR_CO_RESET_RELEASE
    nop                                         // else use CPC Power Up command.
```

MIPS

55

The code is almost done with the system initialization. The last thing it needs to do is to release any other cores in the system from reset. Once they are released they will all start executing this boot code. The code calls release_mp.

+ The code checks to see if there are more cores in the system and if not it will branch to the end of this section and return.

+ the code uses a3 as a counter to decide if it has released all the remaining cores

+ The code checks for a cluster power controller by seeing 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.

release_mp.S - release_mp

Register Fields		Core-Other Addressing Register (CPC_OTHER_REG)	Reset State
Name	Bits		
CORENUM	23:16	CoreNum of the register set to be accessed in the Core-Other address space.	0x0

Defines in cps.h:

```
#define CPS_CORE_LOCAL_CONTROL_BLOCK 0x2000
#define CPC_OTHERL_REG 0x010

powerup_next_core:
   // Send PwrUp command to next core causing execution at their reset exception vector.
   move  a0, a3
   sll     a0, 16
   sw     a0, (CPS_CORE_LOCAL_CONTROL_BLOCK | CPC_OTHERL_REG)(r30_cpc_addr)
```

MIPS

56

Remember this code is only executed by core 0. The code will send the power up signal to each other core in the system.

The code will use the Cluster Power controller to do this. The code needs to place the number of what ever other core it wants to power up in Core-Other Addressing Register.

+ To do this the code moves the number of the core it wants to power up into a0.

+ 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 by computing the offset into the CPC registers and using r30 that contains the Base address of the CPC registers.

release_mp.S - release_mp

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.	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 to SI_PwrUp hardware signal

```
li      a0, PWR_UP// "PwrUp" power domain command.
sw      a0, (CPS_CORE_OTHER_CONTROL_BLOCK | CPC_CMDO_REG)(r30_cpc_addr)
bne     r19_more_cores, a3, powerup_next_core  // If more cores loop back
addiu   a3, a3, 1
jalr    zero,   ra
nop
```

MIPS

57

The first register in Local Control Block is the CPC Local Command Register. This register is used to power up or down the core. It has a command field called CMD which is the first 4 bits of the register.

+ Right now we are interested in powering the core up. The Code loads the command into a0 and stores that register to command register of the CPC. This will power on the core which will begin executing this boot code from the beginning.

+ Next the code checks to see if there are any other cores in the system by comparing the current core number with the highest core number. If there are other cores, the code loops back to power up the next core. The core number is incremented for the next iteration of the loop in the branch delay slot.

+ when the code has looped through all remaining cores it returns.

Join the Coherent Domain

- start.S call to join_domain

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

    la      a2,    join_domain // Join the Coherence  domain. (OK to use D$ on return.)
    jalr    a2
    nop
```

- Located in cps/join_domain.S

```
LEAF(join_domain)
 # If this is not a CPS then we are done.
    beqz    r11_is_cps, done_join_domain
    nop
```

MIPS

59

Next the code calls the init function that joins this core to the Coherent Domain.

+ the code first checks to see if this is a Coherent Processing System. If it not then it will branch to the end of this function and return.

join_domain.S - join_domain

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

```
// Enable coherence and allow interventions from all other cores.
// (Write access enabled via GCR_ACCESS by core 0.)
li      a0, 0x0f   // Set Coherent domain enable for 4 cores
sw      a0, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_COHERENCE)(r22_gcr_addr)
ehb
```

MIPS

60

The Core Local Coherence Control Register controls the entry and exit of a core into the coherent Domain. Bits 0 through 7 represent a coherent requestor within the system.

+ The code sets to 1 the first 4 bits of a0. Then it stores it to the Core Local Coherence Control Register. This enables the four cores possible in this system to communicate via interventions to this core.

+ Notice the EHB instruction. This is needed to clear an instruction hazard barrier and make sure the Core Local Coherence Control Registers write has taken effect before the code continues.

join_domain.S - join_domain

```
    move  a3, zero  // set initial value of the core number being waited for
next_coherent_core:
    sll     a0, a3, 16
    sw      a0, (CORE_LOCAL_CONTROL_BLOCK | GCR_CL_OTHER)(r22_gcr_addr)
busy_wait_coherent_core: // Wait for each core in turn to join the domain
    lw      a0, (CORE_OTHER_CONTROL_BLOCK | GCR_CO_COHERENCE)(r22_gcr_addr)
    beqz    a0, busy_wait_coherent_core   // Busy wait on core joining.
    nop
    bne     a3, r19_more_cores, next_coherent_core
    addiu   a3, 1       // increment core number
done_join_domain:
    jr      ra
    nop
END(join_domain)
```

MIPS

61

+ The code initializes a3 which it will use as a loop counter to 0.

+ “next coherent core” is the label which the code will loop back to and join the next core domain.

+ The code sets up the Core-Other Addressing Register, with the core number it wants to join with. 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.

The code now reads the Core Local Coherence Control Register of the other core. Remember how the code set this cores 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.

+ Once the other core has joined, the code checks to see if there are more cores to wait for and if there is it branches back to the next coherent core label. It also increments the other core count.

+ when all the cores have been waited for the code returns to start.S.

Start VPE1

- start.S call to init_vpe1

```
la      a2,    init_vpe1   // call to set up MT ASE vpe1 to execute this boot code also.
jalr    a2
nop
```

- mt/init_vpe1.S

```
LEAF(init_vpe1)

    beqz    r21_more_tcs, done_init_vpe1
    nop
    beqz    r20_more_vpes, done_init_vpe1
    nop
```

MIPS

63

The init_vpe1 is called if the core is a MT core. This code initializes the second VPE of an MT system.

+ It first checks to see if there is an additional TC to bind to a second VPE

+ and then if there is a second VPE. If neither is true, no action is required so the code will jump to the done point.

init_vpe1.S

- Enable Virtual Processor Configuration

```
// This is executing on TC0 bound to VPE0.
// Therefore VPEConf0.MVP is set to enter config mode
mfc0   v0, C0_MVPCTL                 // C0_MVPCtl
or     v0, (1 << 1)                  // M_MVPCtlVPC (VPE Configuration State enable)
mtc0   v0,  C0_MVPCTL                // C0_MVPCtl
ehb
```

MIPS 64

The setup for the second VPE will require access to some registers that are usually non-writable. To write to these registers, the code needs to enable Virtual Processor Configuration. To do this the code reads the MVPControl Register.

+ Then it sets the VPC bit, writes it back to CP0, and executes an ehb to ensure the write has been completed before it continues.

init_vpe1.S (Continued)

- **Set up defines and variables**

```
#define a0_NTCS    a0
#define a2_NVPES   a2
#define a3_TC      a3

    // Get number of a3_TC's and VPE's
    mfc0   v0, C0_MVPCONF0          // read C0_MVPCONF0
    ext    a0_NTCS, v0, 0, 8        // extract PTC
    ext    a2_NVPES, v0, 10, 4      // extract PVPE
    // Initialize TC's/VPE's
    move   TC, zero
```

MIPS

65

The setup for the second VPE will require access to some registers that are usually non-writable. To write to these registers, the code needs to enable Virtual Processor Configuration. To do this the code reads the MVPControl Register.

+ Then it sets the VPC bit to enable Virtual Processor Configuration, writes it back to CP0, and executes an ehb to ensure the write has been completed before it continues.

+ Defines are created to make it easier to follow the code. NTCS is a register that will hold the number of TCs in the system, NVPES will hold the number of VPEs in the system, and TC will hold the current number of the TC being initialized.

+ The code reads the MVPConf0 register.

+ Then it extracts the highest TC number and the highest VPE number into the registers noted above. Next it initializes the TC count to 0.

init_vpe1.S

- Initialize remaining TCs

```
nexttc:
        // Select TCn
        mfc0    v0, C0_VPECTL       // read C0_VPECTL
        ins     v0, a3_TC, 0, 8     // insert TargTC being configured
        mtc0    v0, C0_VPECTL       // write C0_VPECTL
        ehb

        // Bind a3_TC to next VPE
        beqz    a3_TC, nextvpe      // Check for TC 0 so code doesn't rebind TC0
        nop

        // Halt a3_TC being configured
        li      v0, 1               // set Halt bit
        mttc    0v0, C0_TCHALT      // write C0_TCHALT
        ehb
        slt     v1, a2_NVPES, a3_TC
        bnez    v1, 2f              // Bind spare a3_TC's to VPElast (next slide)
        move    v1, a2_NVPES
```

MIPS

66

This loop will initialize the remaining TCs in the system.

+ The code sets up the target TC field in the VPEControl register by reading the register, inserting the TC number into the TargTC field and writing it back to the VPE Control register. It executes an ehb to ensure the write took effect before it continues. Doing this controls which TC the mftc0 and mttc0 instructions write to. This is how the CP0 registers of a TC other than the one executing the code are written to.

+ The code checks to see if the current TC in the loop is TC 0. If it is, it branches forward to the next section to initialize the next VPE since it doesn't want to re-initialize itself.

+ The TC should be halted before the code starts changing its configuration. To do that, a 1 is placed in v0 and then moved to the C0_TC Halt register. The code executes an ehb to ensure the move has taken effect.

+ The code tests to see if this TC is the first TC to be bound to a VPE. If not, it branches to the binding code. This is done because this example sets up only one TC to be executable on each VPE. If there are more TCs than VPEs, the branch will be taken and the TC will be bound to the last VPE in the system.

+ The branch delay slot is used to save the number of VPEs in a2_NVPES.

init_vpe1.S

```
        // Set XTC for active TC's
        mftc0   v0, C0_VPECONF0     // read C0_VPEConf0
        ins     v0, a3_TC, 21, 8    // insert TC -> XTC
        mttc0   v0, C0_VPECONF0     // write C0_VPEConf0

        move    v1, a3_TC
2:
        // Bind TC to a VPE
        mftc0   v0, C0_TCBIND       // read C0_TCBind
        ins     v0, v1, 0, 4 // insert VPE -> CurVPE
        mttc0   v0, C0_TCBIND       // write C0_TCBind
```

MIPS

67

Next set this TC to be the only TC runnable on the VPE. For current cores, this effectively sets TC 1 to run exclusively on VPE 1. To do this, the XTC field in the VPEConf0 register 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.

+ Now set v1 to the current TC so the TC will be bound to its corresponding VPE. This overwrites the value set in the branch delay slot on the last slide.

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.

NOTES:

If the TC number was equal to or greater than NVPES, then v1 = NVPES, and the TC will be bound to the last VPE in the system.

If the TC Number was less than NVPES, then v1 = TC, and the TC will be bound to the corresponding VPE.

init_vpe1.S

- Disable interrupts

```
li      v0, (1 << 10)       // IXMT bit 10 = 1
mttc0   v0, C0_TCSTATUS     // C0_TCStatus
```

- Initialize GPRs

```
// Initialize the TC's register file
mttgpr  v0, $1
mttgpr  v0, $2
mttgpr  v0, $3
mttgpr  v0, $4
.........
mttgpr  v0, $30
mttgpr  v0, $31
```

MIPS

68

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 the interrupt exempt bit, IXMT up v0 and then it move it to the TCStatus register (CP0 register 2 select 1).

+ The code then initializes the TC's GPR registers using the same method used in init_gpr.S.

init_vpe1.S

- Check for more VPEs

```
nextvpe:
    slt     v1, a2_NVPES, a3_TC         // NVPE < a3_TC?
    bnez    v1, donevpe                 // No more VPE's
    nop
```

MIPS

69

The code checks to see if there any more VPEs to initialize.

It does this by checking to see if the number of VPEs left is less than the number of TCs. If it is, all the VPEs have already been initialized, so the code branches forward to donevpe where it will check to see if there are any more TCs to initialize.

init_vpe1.S

- Initialize the Second VPE

```
// Disable multi-threading with TC's
   mftc0    v0, C0_VPECTL       // read C0_VPECtl
   ins      v0, zero, 15, 1     // clear TE (threading enable)
   mttc0    v0, C0_VPECTL       // write C0_VPECtl

   beqz     a3_TC, donevpe      // no more TCs
   nop
```

MIPS

The code will now initialize the second VPE. In fact, on current cores there are at most 2 VPEs, so this code will be executed only if this core has a second VPE.

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.

+ The code checks to see if this is TC 0. If it is, it branches around the initialization to the end of the function, because this has already been done for TC 0.

init_vpe1.S

- Set up to initialize CP0 registers of VPE1

```
// For VPE1..n
// Clear VPA and set master VPE
mftc0   v0, C0_VPECONF0    // read C0_VPEConf0
ins     v0, zero, 0, 1     // clear VPA
or      v0, (1 << 1)       // set MVP
mttc0   v0, C0_VPECONF0    // write C0_VPEConf0
```

MIPS

71

The code needs to make sure that no TC is running on the VPE it is initializing.

+ It does this by reading the CP0 VPEConf0 of the VPE it is initializing and inserting a 0 into the Virtual Processor Activated or VPA Field.

+ 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.

init_vpe1.S

- initialize CP0 registers of VPE1

```
mfc0    v0, C0_STATUS       // read C0_Status
mttc0   v0, C0_STATUS       // write C0_Status

li      v0, 0x12345678
mttc0   v0, C0_EPC          // write C0_EPC with dummy value
mttc0   zero, C0_CAUSE      // Clear C0_Cause
mfc0    v0, C0_CONFIG       // read VPE 0 C0_Config
mttc0   v0, C0_CONFIG       // write it to VPE 1 C0_Config
mftc0   v0, C0_EBASE        // read C0_EBASE
ext     v0, v0, 0, 10       // extract CPUNum
mttgpr  v0, r23_cpu_num     // write CPUNum to GPR 23 of VPE1
```

MIPS

72

The code copies the Status register of the running TC to the Status register of the TC being initialized.

+ Initialize the Error PC to a dummy value.

+ Clear the Cause register.

+ Copy the Config register of the running TC to the Config register of the TC being initialized.

+ The code initializes gpr 23 of VPE1 with the core number it is on. It does this by reading the EBASE register and extracting the Cpu_num field. Then it copies the CPUNum to the TC's GPR 23.

init_vpe1.S

- Set TC start address

```
la      a1, __reset_vector    // load boot code starting address
ins     a1, 29, 1             // Convert to cached kseg0 address
mttc0   a1, C0_TCRESTART      // write C0_TCRestart so TC 1 on VPE 1
                              // will execute this boot code once virtual processing is enabled
mftc0   v0, C0_TCSTATUS       // read TCStatus
ins     v0, zero, 10, 1       // clear IXMT to enable this TC to take interrupts
ori     v0, (1 << 13)         // set A to activate this TC
mttc0   v0, C0_TCSTATUS       // write TCStatus
mttc0   zero, C0_TCHALT       // clear H in TCHalt to unhalt this TC
mftc0   v0, C0_VPECONF0       // read VPEConf0
ori     v0, 1                 // set VPA Virtual Processor Activated
mttc0   v0, C0_VPECONF0       // write VPEConf0 to start this TC executing once EVP is set
1:
```

MIPS

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).

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.

+ 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.)

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.

init_vpe1.S

- More TCs to initialize?

```
donevpe:  // Done initializing VPE 1 if there was one
   addu    a3_TC, 1              // advance TC number
   sltu    v1, a0_NTCS, TC       // set v1 if TC number is less than total # of TC in system
   beqz    v1, nexttc            // go back and initialize another TC
   nop
                                 // Fall through once all TCs are initialized
```


Next the code will check to see if there are any more TCs in the system to initialize.

It adds 1 to the current TC being initialized, and then tests to see if it is still within the limits of the number of TCs that are in the system. If it is, it branches back to the top of the loop.

init_vpe1.S

- Enable Virtual Processing

```
// Exit config mode
mfc0    v0, C0_MVPCTL       // read MVPCtl
ori     v0, 1               // set EVP (Enable Virtual Processing) to enable execution by vpe1
ins     v0, zero, 1, 1      // Clear VPC (VPE Configuration State) bit
mtc0    v0,  C0_MVPCTL      // write MVPCtl (VPE 1 TC 1 will start executing this boot code)
ehb
```

MIPS

75

This is the path the code falls through when there are no VPEs or TCs left to initialize. Next 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.

+ And write it back to the MVPCtl register

init_vpe1.S

- Clean up and Return

```
#undef a0_NTCS
#undef a2_NVPES
#undef a3_TC

done_init_vpe1:

  jr      ra
  nop

END(init_vpe1)
```


Now for some clean-up of the code to remove the "//defines" it created in the beginning of this file.

This function is done and returns to start.S.

start.S - init_done

- **This core is done with the assemble part of the init.**
  - Set up arguments to C code and return from boot exception.

```
init_done:

    // Prepare for eret to main (sp and gp set up per vpe in init_gpr).
    la      ra, all_done // If main return then go to all_done:.
    la      a1, main      // address of main to be loaded into the Error Exception Program Counter
    mtc0    a1, C0_ERRPC // ErrorEPC
```

MIPS

77

The assemble part of the initialization is complete. The code is now ready to call the main function.

+ the code sets the return address to return to the all_done label. This is done in case the main function returns. Normally the main function never returns. It usually starts up the OS and the OS handles everything from there on including halting the system. Just in case something goes wrong we don't want just any random code to execute. The all done function is just a simple loop that will cause the processor to spin forever.

+ Next the address of main is loaded into a1 and then the value of a1 is moved to the CP0 Error exception PC. This is the return from exception address. To explain what is happening, when a MIPS core boots it is in exception mode so all of the code that has executed so far is part of the boot exception processing. At the end of this boot exception processing the code will use the error return instruction to change mode to normal processing and start executing instructions at the address stored in the Error exception PC.

start.S - init_done

```
// initialize global variable num_cores.
la      a1, num_cores                   // load address of num_cores into a1
ins     a1, r1_all_ones, 29, 1          // convert it to a Uncached kseg1 address
add     a0, r19_more_cores, 1           // set a0 to the total number of cores
sw      a0, 0(a1)                       // store number of cores to address of num_cores

// Prepare arguments for main()
move    a0, r23_cpu_num                 // main(arg0) is the "cpu" number (cp0 EBase[CPUNUM].)
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 on this core.

eret                    // Exit reset exception handler for this vpe and start execution of main().
```

MIPS

78

The variable num_cores is a global variable used in the C code. This is an example on how you would initialize a global variable in assemble to be used later in C.

+ First the address of the global variable num_cores ins loaded into a1 and converted to a uncached address.

+ a0 is set to the number of cores in the system by adding 1 to the highest core number that was stored in r19.

+ a0 is then stored to the address of num_cores.

The code places the arguments to main in 4 argument registers. The use of these registers to pass arguments follow the “C” ABI calling convention. The arguments are the CPU number, the core number, the VPE number if this is an MT core and the total number of VPEs .

+ Eret is called this will clear exception mode and start the execution of the main function.

Main.c

- **Start of main function**
  - Located in main.c

```
// 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 ;
  int temp;

  // End timing of boot for this vpe/cpu.
  asm volatile ("mfc0 %[temp], $9": [temp] "=r"(temp) :) ;
  boot_count[cpu_num] = temp ;
```

MIPS 79

The code in the main function gives you an example of how to synchronize all CPUs in the CPS before continuing execution of test code or starting the OS.

+ Main starts out by declaring some local variables it will be using.

+ Next the code reads the value of the CP0 Count register. The inline assemble code moves the count register value into the local variable temp. Then the code writes that value into the boot_count array element for this particular core. The boot_count array is a global variable array with will collect the number of cycles the boot process took for each CPU. This is just an example of recording statistics from each CPU. The boot_count is not used further in this example.

Main.c

- Report the number of VPEs on the Core

```
// if there is only one core skip the sync step
if (num_cores != 1)
   {
   // Each core's vpe0 reports number of vpe in the core.
    if (vpe_num == 0) {
            // Dedicated word in global array.
           vpe_on_core[core_num] = num_vpe_this_core ;
    }
```

MIPS

80

Each Core will report how many Virtual processors are available on the core. The OS could use this to set itself up for each Core.

Main.c

- Synchronize CPUs

```
if (cpu_num == 0) {
    // Tally number of "VPEs" there are in this CPS.
    // Busy wait for core to report number of vpe.
    for (i = 0; i < num_cores; i++) {
        while (!vpe_on_core[i]) {
        }
        num_cpus += vpe_on_core[i] ;
    }
```

MIPS

81

The code uses CPU 0 do the synchronizing of the rest of the CPUs.

+ the code waits for each core to report the number of VPEs on a core. If a core is a single core processor then it will report a 1 so no distinction will be made between a single core or an MT system with only 1 VPE. The code tallies the number of VPEs each core reports which it will use a little later in this loop.

Main.c

- Synchronize CPUs (continued)

```
// Wait for other VPEs to indicate they are ready.
  for (i = 1; i < num_cpus; i++) {
      MALTA_CHAR(cpu_num) = 'W' ;    // display 'W' for Waiting.
      while (!ready[i]) ;            // Busy wait for all CPUs to be ready
  }
// Release other VPEs to run their tasks.
   for (i = 1; i < num_cpus; i++) {
      MALTA_CHAR(i) = 'i' ;          // display 'i' for interrupted (from cpu0.)
      set_ipi(i) ;                   // Send a inter-processor interrupt to all other VPEs
  }
} // end if core 0
```

MIPS

82

The code uses CPU 0 do the synchronizing of the rest of the CPUs.

+ The next loop will use the num_cpus variable to wait in turn for each CPU to report it is ready. The MIPS Malta evaluation board has a small display and the code uses it here to report which core it is Waiting for. The code writes a “W” to the display element that corresponds to the Core being Waited for.

+ When a core reports it is ready it will call the wait instruction which will stop the core until the core receives an interrupt. It is CPU 0s job to send the interrupt to wakeup the other cores so they can continue processing. It does this by using inter-processor interrupts that were set up in the boot cps 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. I’ll talk about the set_ipi function in the next slide. Once CPU 0 finishes this loop it can continue and execute test code and/or the OS.

Main.c set_ipi() inter-processor interrupt function

Register Fields		Global Interrupt Write Edge Registers (GIC_SH_WEDGE)	Reset State
Name	Bits		
RW	31	Controls whether this write is setting or clearing a bit in the Edge Detect Register. If this bit is set, the selected bit in the register is set. If this bit is cleared, the selected bit in the register is cleared.	UD
Interrupt	30:0	This field is the encoded value of the interrupt that is being cleared or set. For example, a value of 0xB means interrupt 11 (decimal).	UD

- **Send an inter-processor interrupt**

```
#define GIC_SH_WEDGE     *((volatile unsigned int*) (0xbbdc0280))  // address of write edge register
#define FIRST_IPI        32          // GIC interrupts 32+ interrupts between cores.
void set_ipi(int cpu_num) {
   // Use external interrupts 32..39 for ipi
   GIC_SH_WEDGE = 0x80000000 + FIRST_IPI + cpu_num ; // Use external interrupts 32..39 for ipi
}
```

MIPS

83

The set_ipi function will send an interrupt through the Global Interrupt Controller to a specific CPU number provided by cpu_num. The boot MIPS code sets up the GIC to wire each interrupt to a specific CPU.

Recall that interrupts 32 through 39 correspond to CPU or VPEs 0 through 8 so CPU 0 uses interrupt 32 CPU 1 interrupt 33 and so on.

+ The #define GIC_SH_WEDGE sets up a pointer to the GIC Global Interrupt Write Edge Register. It does so by combining the address of the GIC register control block, which in this case is at hex bb dc 00 00 with the offset of the Global Interrupt Write Edge Register of hex 280.

The value written to the Global Interrupt Write Edge Register has 2 parts. Bit 31 is set to indicate that the code is sending the interrupt signal. Bits 0 through 30 determine which interrupt the signal will be sent to.

+ The code calculates the proper interrupt by using the #define FIRST IPI as a base interrupt number and adding the cpu number.

Once this value is written the corresponding CPU will receive an interrupt which will wake it up to continue executing code where it left off.

Main.c - other CPUs interact with CPU 0

```
} else {
    GIC_SH_WEDGE = FIRST_IPI + cpu_num ;    // Clear this "cpu"'s ipi source
    // Enable interrupts and wait to be released via ipi from cpu0
    asm volatile ("ei") ;
    MALTA_CHAR(cpu_num) = 'r' ;     // display 'r' for ready to receive interrupt
    ready[cpu_num] = 1 ;
    asm volatile ("di") ;               // disable interrupts
    while (!start_test[cpu_num]) { // if start_test[cpu_num] is set than core has received IPI  (see next slide)
      asm volatile ("wait") ;           // Wait for interrupt (qualified with "start_test").
      // enable interrupts so interrupt routine can run and set the start_test bit
      asm volatile ("ei") ;             // enable interrupts
      asm volatile ("ehb") ;            // clear hazard barrier (interrupt should be taken at this point)
      asm volatile ("di") ;             // disable interrupts
      asm volatile ("ehb") ;            // clear hazard barrier
    }
  }
```


All processors other than processors will Stop and wait for processor 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.

+ 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.

+ Next interrupts are disabled for the VPE. This avoids any race condition between the testing of the start_test array and the wait instruction.

+ The code will loop, testing its element of the start_test array and calling wait to wait for an interrupt (sent by CPU0).

+ When any interrupt is signaled the processor will wake up and enable 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.

+ 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.

Interrupt code for IPI

```
.org 0x380 /* General exception. */
   // ..... Write ASCII char to Malta ASCII display - system specific and just for debugging
   li      k0, (GIC_SH_WEDGE | GIC_BASE_ADDR)      // Load address of Write Edge Register
   mfc0    k1, C0_EBASE                            // Get cp0 EBase
   ext     k1, k1, 0, 10                           // Extract CPUNum
   addiu   k1, 0x20                                // Offset to base of IPI interrupts (decimal 32).
   sw      k1, 0(k0)                               // Clear this IPI.
   la      k0, start_test                          // Load address of the start_test array
   mfc0    k1, C0_EBASE                            // Get cp0 EBase
   ext     k1, k1, 0, 10                           // Extract CPUNum
   sll     k1, k1, 2                               // CPUNum * 4 - to get array index
   addu    k0, k0, k1                              // index into CMP global "C" variable start_test
   li      k1, 1                                   //
   sw      k1, 0(k0)                               // set array element to 1
   eret
   nop
```

MIPS

85

This is the interrupt routine that will receive the Interprocessor interrupts. Remember a interrupt is sent by processor 0 through the use of the set_ipi code as shown in a previous slide. The interrupt code shown here is in start.S. It is loaded at 0xBFC0 0380 which is the general exception vector. Since the only interrupt generated by this code is an Interprocessor interrupt the code proceeds to clear the interrupt in the Write Edge register.

+ The code loads the address of the write edge register into K0.

+ then reads the CP0 EBASE register and extracts the CPU number

+ Again Recall that interrupts 32 through 39 correspond to processors 0 through 8 so the code uses 32 or Hex 20 as a base number for the interrupt and adds it to the CPU number to get the correct IPI number for this processor.

+ next the code writes the interrupt number to the Write Edge register to clear the interrupt.

Now that the interrupt is cleared the code will set this processors element in the start_test array.

+ The code does this by loading the address of the start_test array into K0.

+ Then the code again reads the CP0 Ebase register, extracts the CPU number multiplies it by 4 because the array is an array of integers (which are 4 bytes each).

+ next the code adds this index to the starting address of the start_test array to get the address of the array element specific to this processor.

+ The code then sets K1 to 1 and stores it to the array element.

When the code returns from the interrupt each VPE will return from the wait instruction and look at its element of the Start_test array to see if it should continue.

Main.c

- The End or the Beginning
  - Start test or OS

```
// Put test code here:
   MALTA_CHAR(cpu_num) = 't' ;        // 't' for test.
   while (1) {
      for (j = 0 ; j < (4 * 1024) ; j++) {
            MALTA_CHAR(cpu_num) = cpu_num + 0x30 ; // ASCII char for cpu number.
      }
      for (j = 0 ; j < (4 * 1024) ; j++) {
            MALTA_CHAR(cpu_num) = ' '; // blank out display position for this "cpu".
      }
   }
   return 0 ;  // should never return!
}
```

MIPS

86

At this point the boot of the CPUs are done. You can add your code here to run a test or call the OS boot up.