Design and Implementation of a MIPS-Inspired Dual-Core Processor System:
Hardware Architecture and FreeRTOS Port

Team: Custom Processor Development (PESHWAS Pool)

September 2, 2025

Abstract

This report documents the design, implementation, and verification of a complete MIPS-inspired dual-core processor system.

The first part details the hardware architecture, focusing on the memory arbitration layer, a shared L1 cache with a write-through policy, and a single-write-port RAM interface. The architecture supports dual-port reads and serialised writes through a multiplexer controlled by a round-robin arbiter. This section includes detailed signal mappings, finite-state machine descriptions, flowcharts, timing diagrams, pseudocode suitable for direct porting into Simulink / MATLAB environments, and an analysis of the Simulink implementation.

The second part describes the software implementation, specifically the porting of the FreeRTOS real-time operating system to the custom dual-core hardware. It covers crucial components such as the scheduler, context switcher, and the Board Support Package (BSP). It also details the verification strategy, which involved integrating and testing the FreeRTOS port on an x86 (Windows) simulation environment to validate functionality and performance before deployment on the target MIPS-like hardware.

Contents

Part I

Hardware Design and Implementation

Chapter 1

Introduction

1.1 Motivation

Modern multi-core processors must coordinate access to shared memory resources to guarantee correctness and performance. In many embedded and educational implementations, a shared L1 cache with a controlled interface to main memory is used to balance throughput and complexity. This project builds a MIPS-inspired dual-core processor in a Simulink environment with the following design goals:

- Allow simultaneous read accesses to the shared L1 cache (dual-port reads).
- Serialize write accesses using a single write port (single write address and data line).
- Implement a round-robin arbiter that grants access fairly and handles cache misses by coordinating cache-to-RAM transfers.
- Use a write-through cache policy to maintain consistency between cache and RAM.
- Provide clear, testable signals for use with pipeline stall logic inside each core.

1.2 Scope of this document

This document describes the architecture, signal-level connections, arbiter FSM, cache behaviour, RAM model, and pipeline integration for a MIPS-inspired instruction set (R, I and J types). It also analyses the Simulink screenshots provided by the implementers and supplies detailed pseudocode and diagrams to make this design replicable and auditable.

Chapter 2

Background: MIPS-Inspired Processor Fundamentals

2.1 Overview of the MIPS-Inspired ISA

The implemented system follows a classic R/I/J instruction grouping (the processor is MIPS-inspired). Briefly:

- **R-type** (register): three-register format (opcode=0, func field determines operation). Operands: `rs`, `rt` read from register file; result written to `rd`.
- **I-type** (immediate): one register and an immediate, used for loads/stores/branches/ALU immediate instructions. Operands: `rs` and immediate; result often written to `rt`.
- **J-type** (jump): target address for unconditional jump.

2.2 Register File and PC semantics in this design

- The register file is a synchronous write, asynchronous read (or implemented as two read ports and one write port). Ports are labeled `rs`, `rt` (read ports) and `rd/rt` (write port depending on instruction).
- Program Counter (PC) is updated each instruction: typically `PC := PC + 4` except when a taken branch or jump modifies it. Pipeline hazards are handled by stall/insertion logic using `grant` signals from the arbiter to freeze IF/ID/ID/EX as required.
- In the implemented cores, the memory (load/store) stage drives `mem_read` and `mem_write` signals to request data access; these are in turn mediated by the arbiter and cache.

2.3 Pipeline and dataflow (short)

Each core implements a simple 5-stage pipeline (IF, ID, EX, MEM, WB) with registers between stages. The MEM stage uses the arbiter's grant; when `grant=0` for that core, the MEM stage is stalled (pipeline registers not advanced) until grant asserted.

Chapter 3

Top-Level System Architecture

3.1 Top-level block diagram

Figure **??** shows the canonical top-level architecture implemented in Simulink: two cores (Core0, Core1), the central Arbiter, Shared L1 Cache, and RAM system. The cache holds two read ports and a single write port. Writes are funnelled using an arbiter-controlled MUX.

Chapter 4

Signal Interface and Pin Mapping

This section provides an explicit, exhaustive signal map to be used when wiring Simulink blocks or converting to HDL.

4.1 Core $\leftrightarrow$ Arbiter signals

Signal	Semantics (direction)
`req_i`	Request enable: asserted by core when MEM stage needs memory access (Core$\rightarrow$Arbiter)
`rw_i`	Read/Write indicator: 0 = read, 1 = write (Core$\rightarrow$Arbiter)
`addr_i`	Address bus (32-bit) (Core$\rightarrow$Arbiter)
`wdata_i`	Write data bus (32-bit) (Core$\rightarrow$Arbiter)
`grant_i`	Grant / stall signal (Arbiter$\rightarrow$Core): 1 = core allowed to proceed, 0 = stall
`rdata_i`	Returned read data (32-bit) (Arbiter$\rightarrow$Core)
`stall_i`	Optional explicit stall (derived as `g̃rant_i`) (Arbiter$\rightarrow$Core)

4.2 Arbiter $\leftrightarrow$ Cache signals

Signal	Semantics (direction)
`read_addr_0/1`	Read addresses forwarded to cache (Arbiter$\rightarrow$Cache)
`read_enable_0/1`	Read enables for cache ports (Arbiter$\rightarrow$Cache)
`cache_rdata_0/1`	Read data returned from cache (Cache$\rightarrow$Arbiter)
`cache_hit_0/1`	Cache hit indicators (Cache$\rightarrow$Arbiter)
`write_addr`	Single write address after MUX selection (Arbiter/MUX$\rightarrow$Cache)
`write_data`	Single write data after MUX selection (Arbiter/MUX$\rightarrow$Cache)
`cache_write_enable`	When 1, perform a write into cache and trigger write-through to RAM (Arbiter$\rightarrow$Cache)
`cache_write_ack`	Cache acknowledges it stored the data (Cache$\rightarrow$Arbiter)

Signal	Semantics (direction)
`cache_fill_done`	Cache finished filling a miss entry from RAM (Cache→Arbiter)

4.3 Cache ↔ RAM signals

Signal	Semantics (direction)
`ram_read_addr`	Address to read from RAM (Cache→RAM)
`ram_read_enable`	When 1 start RAM read operation (Cache→RAM)
`ram_read_data`	Data read from RAM (RAM→Cache)
`ram_read_done`	RAM indicates read data valid (RAM→Cache)
`ram_write_addr`	Address to write into RAM (Cache→RAM)
`ram_write_data`	Write data for RAM (Cache→RAM)
`ram_write_enable`	Single write enable for RAM (Cache→RAM)
`ram_write_ack`	RAM acknowledges write done (RAM→Cache)

Chapter 5

Arbiter Design and FSM

5.1 Design goals and constraints

Key requirements:

- Fairness via round-robin: avoid starvation.
- Allow concurrent reads (both cores read simultaneously).
- Serialize writes (only one core at a time may write).
- On cache miss, stall requesting core, trigger cache-to-RAM read, and un-stall when fill done.
- For writes, use a single write address/data via a MUX controlled by arbiter grants and write enables.

5.2 State machine

Define the arbiter as a small FSM. Minimal states:

- **IDLE**: no pending requests; monitor requests.
- **SERVE_READS**: both read requests may be served; hits handled immediately; misses route to WAIT_RAM_READ.
- **WAIT_RAM_READ**: waiting for RAM to return data for a cache miss.
- **SERVE_WRITE**: grant a single core write (via MUX), assert cache_write_enable.
- **WAIT_RAM_WRITE_ACK**: waiting for RAM write ack for write-through.

5.3 Detailed Arbiter pseudocode (Matlab-like)

Listing 5.1: Matlab-like arbiter skeleton (drop-in for MATLAB Function block).

```
function [grant_0, grant_1, rdata_0, rdata_1, read_addr_0, read_addr_1, ...
          read_enable_0, read_enable_1, write_addr, write_data,
             cache_write_enable, write_select] ...
          = arbiter_tick(req_0, req_1, rw_0, rw_1, addr_0, addr_1,
             wdata_0, wdata_1, ...
                          cache_hit_0, cache_hit_1, cache_rdata_0,
                             cache_rdata_1, ...
                          cache_fill_done, cache_write_ack)
% persistent state
persistent state last_served
if isempty(state); state = 0; last_served = 0; end
IDLE=0; SERVE_READS=1; WAIT_RAM_READ=2; SERVE_WRITE=3; WAIT_RAM_WRITE
   =4;
```

```

% defaults
grant_0=0; grant_1=0;
rdata_0=0; rdata_1=0;
read_enable_0=0; read_enable_1=0;
read_addr_0=0; read_addr_1=0;
write_addr=0; write_data=0; cache_write_enable=0; write_select=0;

switch state
  case IDLE
    if req_0 && ~req_1
      if rw_0==0, state=SERVE_READS; else state=SERVE_WRITE; end
    elseif req_1 && ~req_0
      if rw_1==0, state=SERVE_READS; else state=SERVE_WRITE; end
    elseif req_0 && req_1
      % concurrent requests: prefer last_served^1
      if last_served==0, % serve core1 first
        if rw_1==0, state=SERVE_READS; else state=SERVE_WRITE; end
      else
        if rw_0==0, state=SERVE_READS; else state=SERVE_WRITE; end
      end
    end

  case SERVE_READS
    read_enable_0 = req_0 && (rw_0==0);
    read_enable_1 = req_1 && (rw_1==0);
    read_addr_0 = addr_0; read_addr_1 = addr_1;
    % observe cache hits
    if read_enable_0 && cache_hit_0
      grant_0 = 1; rdata_0 = cache_rdata_0;
    end
    if read_enable_1 && cache_hit_1
      grant_1 = 1; rdata_1 = cache_rdata_1;
    end
    % handle misses
    if (read_enable_0 && ~cache_hit_0) || (read_enable_1 && ~
       cache_hit_1)
      % choose which miss to service first (round-robin)
      if read_enable_0 && ~cache_hit_0 && read_enable_1 && ~cache_hit_1
        if last_served==0, miss_core=1; else miss_core=0; end
      elseif read_enable_0 && ~cache_hit_0, miss_core=0;
      else miss_core=1;
      end
      % stall that core and trigger cache->RAM read via cache interface
      if miss_core==0, % service core0
        grant_0 = 0; grant_1 = cache_hit_1;
        % instruct cache to fetch (cache will assert ram_read_enable)
        state = WAIT_RAM_READ;
      else
        grant_1 = 0; grant_0 = cache_hit_0;
        state = WAIT_RAM_READ;
      end
    else
      % all hits covered; clear state & update last_served
      state = IDLE;
      if grant_0, last_served = 0; end
      if grant_1, last_served = 1; end
    end

  case WAIT_RAM_READ
    % wait for cache_fill_done to be asserted by cache after RAM
       completes read
    if cache_fill_done
      % assume cache has updated cache_rdata_* accordingly
      if req_0 && ~cache_hit_0, grant_0 = 1; rdata_0 = cache_rdata_0;
          last_served=0; end
      if req_1 && ~cache_hit_1, grant_1 = 1; rdata_1 = cache_rdata_1;
          last_served=1; end
      state = IDLE;
    end

  case SERVE_WRITE
    % choose which writer to grant using round-robin and requests
    if req_0 && ~req_1
      write_select = 0;
    elseif req_1 && ~req_0
      write_select = 1;
    else % both
      write_select = last_served; % give other core priority
    end
    if write_select==0
      write_addr = addr_0; write_data = wdata_0;
      grant_0 = 1; grant_1 = 0;
    else
      write_addr = addr_1; write_data = wdata_1;
      grant_1 = 1; grant_0 = 0;
    end
    cache_write_enable = 1;
    state = WAIT_RAM_WRITE;

  case WAIT_RAM_WRITE
    % wait for cache_write_ack and ram_write_ack (cache will handle RAM
        handshake)
    if cache_write_ack
      cache_write_enable = 0;
      % acknowledge to core and update last_served
      if grant_0, last_served = 0; end
      if grant_1, last_served = 1; end
      state = IDLE;
    end
end
end
```

5.4 Discussion of corner cases

- **Simultaneous write attempts**: write-select MUX picks one based on round-robin; the other core sees `grant=0` and must retry the request.
- **Read while write in progress to same address**: arbiter must conservatively stall read if write is pending for same address or if cache coherence risk exists. For simplicity, the implemented arbiter stalls the read (deny grant) until write ack completes if addresses match.
- **Cache fill ordering**: If two misses occur to the same address, ensure single RAM fetch and update both cores appropriately.

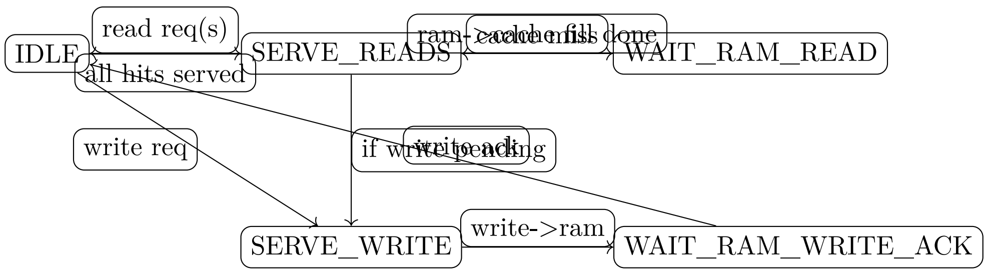

Figure 5.1: Arbiter FSM (high-level).

Chapter 6

Cache Design

6.1 Properties

- Shared L1 cache with dual-read ports and single write port.
- Write-through policy: writes are written to cache and simultaneously propagated to RAM.
- On a read miss, cache issues a RAM read and returns data on fill completion.

6.2 Cache interfaces (explicit)

Cache internal ports are consistent with the earlier signal table. Key behaviour:

- On `read_enable_i`, if tag matches, assert `cache_hit_i` and present `cache_rdata_i`.
- If miss: stall core (via arbiter) until `ram_read_done` returns and cache fills the line, then assert `cache_fill_done`.
- On `cache_write_enable`, write data to cache; forward same write to RAM by asserting `ram_write_enable` and returning `cache_write_ack` after both cache store and RAM acknowledge (or after cache store and RAM queued write).

6.3 Cache pseudocode (internal logic)

Listing 6.1: Cache pseudo-behaviour (high-level)

```
% Inputs: read_addr_0/1, read_enable_0/1, write_addr, write_data,
    write_enable
% RAM interface: ram_read_addr, ram_read_enable, ram_read_data,
    ram_read_done,
%                ram_write_addr, ram_write_data, ram_write_enable,
    ram_write_ack
% Outputs: cache_rdata_0/1, cache_hit_0/1, cache_fill_done,
    cache_write_ack
for each cycle:
  % Read ports
  if read_enable_0
    if lookup_tag(read_addr_0) == HIT
      cache_rdata_0 = read_cache_line(read_addr_0)
      cache_hit_0 = 1
    else
      cache_hit_0 = 0
      % start RAM read if not already queued
      ram_read_addr = read_addr_0; ram_read_enable = 1
    end
  end
  % Similar for port 1
  % Write-through handling
  if write_enable
    write_cache_line(write_addr, write_data)
    % forward to RAM
    ram_write_addr = write_addr; ram_write_data = write_data
    ram_write_enable = 1
  end
  % RAM response handling
  if ram_read_done
    write_cache_line(ram_read_addr, ram_read_data)
    cache_fill_done = 1
  end
  if ram_write_ack
    cache_write_ack = 1
  end
end
```

Chapter 7

RAM Model

7.1 Properties and constraints

- The RAM model supports dual independent read ports (or a multi-port read emulation) and a single write port (serialized).
- Read operations take several cycles (simulated latency); the RAM asserts a handshake signal (`ram_read_done`) when data is valid.
- Write operations also take multiple cycles and return an acknowledge (`ram_write_ack`).

7.2 Timing constraints

Define constants:

- `RAM_READ_LATENCY = N cycles`
- `RAM_WRITE_LATENCY = M cycles`

These constants determine how long arbiter waits in WAIT_RAM_READ/WAIT_RAM_WRITE states.

Chapter 8

Integration with Core Pipeline (detailed)

8.1 How cores use grant signals

- The MEM stage asserts `req` and `rw`; it then waits for `grant` to proceed.
- If `grant==1` and read, core receives `rdata` from arbiter to forward to WB stage.
- If `grant==0`, the core stalls (pipeline registers freeze), effectively pausing the whole pipeline behind MEM stage.

8.2 Register file reads/writes

For R-type:

1. Decode stage: identify rs, rt, rd.
2. IF/ID → ID/EX: register file read occurs in ID stage (rs and rt read) and values propagated to EX.
3. EX stage: ALU uses rs and rt as specified by instruction.
4. WB stage: result from ALU is written back to `rd`.

For I-type (load/store/addi,...):

- Load: EX computes effective address using rs + sign-extended immediate. MEM stage requests a read through arbiter; upon completion, the data is written back to rt.
- Store: EX computes address; MEM asserts write request and writes data (rt) through arbiter/cache.

8.3 Control Decode block (in-depth)

The control decode block maps instruction opcodes and function fields into control signals that power datapath multiplexers and functional units. Typical control outputs:

- `alu_op`, `alu_src_b_imm`, `reg_write`, `mem_read`, `mem_write`, `mem_to_reg`, `branch`, `jump`, `link`

Detailed responsibilities:

1. Decode opcode and func to determine ALU operation (e.g., add, sub, and, or, slt).
2. Generate `reg_write` for register writes in the WB stage.
3. Generate `mem_read` and `mem_write` signals used by the MEM stage and forwarded to arbiter.
4. Provide control to multiplexers for write-back (ALU result vs memory vs PC+4).
5. Provide branch type signals and compute branch decision combining ALU zero and branch type.

Chapter 9

Flowcharts and Timing Diagrams

9.1 Arbiter decision flowchart

(Flowchart rendered in TikZ below.)

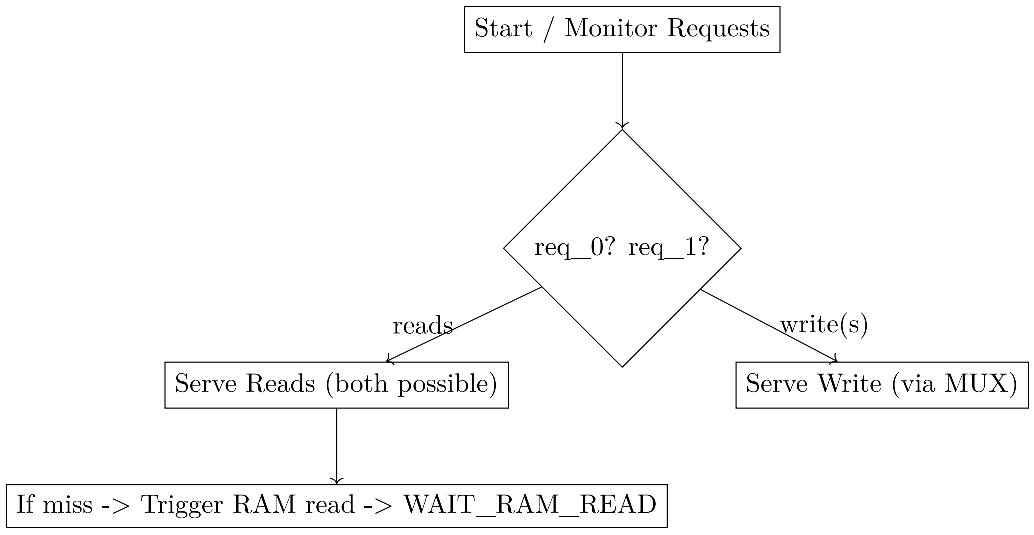
Start / Monitor Requests
req_0? req_1?
reads
write(s)
Serve Reads (both possible)
Serve Write (via MUX)
If miss -> Trigger RAM read -> WAIT_RAM_READ

Figure 9.1: Decision flow for read/write selection.

Chapter 10

Detailed wiring and Simulink mapping

This section repeats the canonical wiring list in a copy-paste friendly table (use these names in Simulink block ports precisely or map aliases):

Signal	Direction	From → To (canonical)
req_0	Core→Arbiter	Core0.req_i → Arbiter.req_0
rw_0	Core→Arbiter	Core0.rw_i → Arbiter.rw_0
addr_0	Core→Arbiter	Core0.addr_i → Arbiter.addr_0
wdata_0	Core→Arbiter	Core0.wdata_i → Arbiter.wdata_0
grant_0	Arbiter→Core	Arbiter.grant_0 → Core0.grant
rdata_0	Arbiter→Core	Arbiter.rdata_0 → Core0.rdata
read_addr_0	Arbiter→Cache	Arbiter.read_addr_0 → Cache.C1_addr
read_enable_0	Arbiter→Cache	Arbiter.read_enable_0 → Cache.C1_ren
cache_hit_0	Cache→Arbiter	Cache.C1_hit → Arbiter.cache_hit_0
cache_rdata_0	Cache→Arbiter	Cache.C1_out → Arbiter.cache_rdata_0
write_addr	MUX/Arbiter→Cache	MUX.out → Cache.write_addr
write_data	MUX/Arbiter→Cache	MUX.out_data → Cache.write_data
cache_write_enable	Arbiter→Cache	Arbiter.cache_write_enable → Cache.C_wen
ram_read_addr	Cache→RAM	Cache.ram_read_addr → RAM.read_addr
ram_read_enable	Cache→RAM	Cache.ram_read_enable → RAM.read_enable
ram_read_data	RAM→Cache	RAM.read_data → Cache.ram_read_data
ram_read_done	RAM→Cache	RAM.read_done → Cache.ram_read_done
ram_write_addr	Cache→RAM	Cache.ram_write_addr → RAM.write_addr
ram_write_data	Cache→RAM	Cache.ram_write_data → RAM.write_data
ram_write_enable	Cache→RAM	Cache.ram_write_enable → RAM.write_enable
ram_write_ack	RAM→Cache	RAM.write_ack → Cache.ram_write_ack

Table 10.1: Canonical signal mapping table.

Chapter 11

Verification and Testing Strategy

11.1 Unit tests

- **Arbiter fairness**: apply long sequences of write requests from both cores; verify round-robin arbitration and absence of starvation.
- **Concurrent reads**: assert both cores read same/different addresses with cache hits; ensure both grants asserted and correct data returned in same cycle.
- **Write-read conflict**: Core0 writes while Core1 reads same address; ensure write serialized and read obtains updated value (write-through ensures RAM and cache updated as required).
- **Cache miss handling**: generate reads to uncached addresses and verify RAM read initiated, cache fill, and core un-stalled on fill.

11.2 Integration tests

- Run small compiled programs (assembly) that perform interleaved reads and writes across both cores, verify program outcome matches single-core reference.
- Pipeline hazard injection: create sequences that cause data hazards and verify pipeline stall behaviour when grant==0.

Chapter 12

Analysis of Provided Simulink Screenshots

12.1 Top-level screenshot (Figure ??)

- The screenshot shows Core1/ Core2 blocks, an Arbiter in center, and a Shared_Cache block with clearly labelled ports: `C1_addr`, `C2_addr`, `C1_data`, `C2_data`, `C_wen`, and RAM ports.
- Observations: the arbiter already exports `grant1` and `grant2` and a `ramload` signal—consistent with the described FSM (ramload used to kick off RAM reads).

12.2 Core datapath screenshots (Figures ?? and ??)

- The core datapath contains IF ID EX MEM WB registers and a RegFile block with `rs` and `rt` read ports and `wdata` write port. The mem control block exports memR/memW signals that connect to `mem_arb` (arbiter interface).
- Observations: the internal control decode block exists and produces `memR` and `memW`. The mem stage already connects to `mem_arb` with signal names consistent with this report.

12.3 Cache screenshot (Figure ??)

- The shared cache block has output signals `c1_out`, `c2_out`, `stall1`, `stall2`, `ram_addr`, `ram_data`, `ram_wen`.
- This confirms the chosen interface mapping and shows the cache is implemented as a module that controls RAM handshake.

Chapter 13

Hardware Conclusion and Future Work

13.1 Summary

This document compiled a complete specification for a MIPS-inspired dual-core processor memory subsystem with a round-robin arbiter, a shared dual-port-read single-write L1 cache (write-through), and a single-write-port RAM. The report contains:

- Signal-level mapping and canonical port names.
- Arbiter FSM, pseudocode, and flowcharts.
- Cache internal behaviour, cacheRAM handshake, and timing diagrams.
- Analysis of Simulink screenshots and mapping to the documented interface.

13.2 Immediate next steps

1. Implement the arbiter FSM in a Stateflow chart or MATLAB Function block using the provided pseudocode.
2. Ensure `grant_i` is wired to core pipeline register enables to effect stalling.
3. Add explicit `cache_hit` outputs and `cache_fill_done` / `cache_write_ack` to the cache block in Simulink.
4. Build test benches for the scenarios in the verification section.

13.3 Future improvements

- Extend to MESI-like coherence if multiple caches later used.
- Support multi-write queuing with a small write buffer to reduce write stall time.
- Add profiling counters and performance counters to measure arbitration latency.

Part II

Software Implementation: FreeRTOS Port

Chapter 14

Introduction to the FreeRTOS Port

This part provides a detailed description of the implementation of FreeRTOS for a dual-core MIPS like architecture. We explain the crucial components such as the scheduler, context switcher, and Board Support Package (BSP) with complete code examples. In addition, we include our attempt to integrate FreeRTOS with the x86 architecture (Windows) to simulate and test the operating system's functionality and correctness before deploying it on the actual MIPS like hardware. We also present an example task scenario used for benchmarking the OS and analyze its performance.

Chapter 15

System Overview from a Software Perspective

15.1 Dual-Core MIPS like Architecture

Our system consists of a dual-core MIPS like processor, which is responsible for executing tasks. Each core operates independently, but they share memory and other system resources. The FreeRTOS-based kernel manages the execution of tasks by distributing them across both cores. Special care is taken to manage synchronization and context switching, ensuring that the two cores operate seamlessly in parallel without conflicts. The scheduler must ensure tasks are correctly distributed across the cores and that high-priority tasks on either core get CPU time as needed.

15.2 FreeRTOS on MIPS like

FreeRTOS was initially designed for single-core systems. For a dual-core system, we made modifications to the FreeRTOS kernel to ensure proper task management on both cores. Key changes were made to:

- **Scheduler Behavior**: Modified to account for multiple cores and allow simultaneous task execution on two processors.
- **Context Switching**: Extended to save and restore context on a per-core basis, so that tasks on different cores can be preempted and resumed independently.
- **Synchronization Mechanisms**: Enhanced (using mutexes and semaphores) to allow safe communication between tasks possibly running on different cores, preventing race conditions.

Chapter 16

Implementation of Key Software Components

16.1 Scheduler

The scheduler is responsible for managing the execution of tasks on both cores. Each task is assigned a priority, and the scheduler ensures that the highest-priority ready task is executed first on an available core. For our dual-core MIPS like architecture, the scheduler was modified to handle tasks across two cores while ensuring fair and efficient distribution.

When initializing the scheduler, we set up the data structures needed for two cores (such as ready queues for each core or a combined priority list with core affinity information). An idle task is created for each core to ensure the core is never idle without a task. The task creation function is also adapted for the dual-core context; tasks can be created and will be managed by the scheduler which may assign them to a core (either statically or dynamically based on load).

Below is the code for initializing the scheduler and creating tasks in our dual-core system:

```c
// Initialize the FreeRTOS scheduler for dual-core MIPS like system
void scheduler_init(void) {
    // Initialize task lists or structures for two cores
    initCoreStructures(); // e.g., separate ready lists for each core
    // Create idle task for each core to run when no other task is available
    createIdleTask(0);
    createIdleTask(1);
    // (Additional initialization as needed)
}

// Task creation API for both cores
// In this simplified model, tasks are created without binding to a specific core.
// The scheduler will assign tasks to cores dynamically based on availability and priority.
void create_task(void (*task_function)(void *), const char *name, uint32_t priority) {
    // Use FreeRTOS API to create a task. The new task is added to the ready list.
    BaseType_t result = xTaskCreate(task_function, name, configMINIMAL_STACK_SIZE, NULL, priority, NULL);
    if (result != pdPASS) {
        // Handle task creation failure (e.g., out of memory)
    }
}

```

```c
// Start FreeRTOS scheduler for dual-core system
void start_scheduler(void) {
    // Start the scheduler which will begin tick interrupts on both
       cores
    // and allow tasks to start executing.
    vTaskStartScheduler();
    // vTaskStartScheduler will not return unless there is insufficient
        heap.
}
```

In our implementation, the FreeRTOS kernel was extended so that when the scheduler runs, it can choose a task for each core. For example, if two high-priority tasks are ready, the scheduler will run one on core 0 and the other on core 1 simultaneously. If a single highest-priority task is ready, one core will run that task while the other core might run a lower-priority task or its idle task. This dual-core scheduling ensures the CPU resources are utilized effectively.

16.2 Context Switcher

The context switcher is a core part of FreeRTOS. It is responsible for saving the state (CPU registers, stack pointer, etc.) of a running task before switching to another task, and restoring the state of the task being resumed. In a dual-core system, context switching must occur independently on each core. Each core can perform a context switch when its scheduler decides to run a different task (for example, during a tick interrupt or when a high-priority task becomes ready).

Below is the code (simplified) for the context switching mechanism in our dual-core MIPS like FreeRTOS port. We use macros `portSAVE_CONTEXT()` and `portRESTORE_CONTEXT()` (which are implemented in assembly for MIPS like) to save and load task contexts. The variable `pxCurrentTCB` holds a pointer to the current task's control block (which includes the task's stack pointer), and `pxNextTCB` is a pointer to the next task chosen to run by the scheduler.

```c
void vTaskSwitchContext(void) {
    // Select the highest priority task that is ready to run
    TCB_t *pxNextTCB = getHighestPriorityTask();

    // If the next task is different from the current task, perform a
       switch
    if (pxCurrentTCB != pxNextTCB) {
        // Save context of current task (push registers onto its stack,
            save stack pointer in TCB)
        portSAVE_CONTEXT();

        // Update the current task pointer to the new task
        pxCurrentTCB = pxNextTCB;

        // Restore the context of the new task (pop registers from its
           stack, set up stack pointer, etc.)
        portRESTORE_CONTEXT();
        // After this macro, the CPU registers have been loaded for the
            new task, and it will resume execution.
    }
}
```

Each core will perform this context switch routine independently. For instance, when core 0's tick interrupt fires, it may switch tasks on core 0 if needed, while core 1 can continue running its task (or also perform a switch if a tick on core 1 causes a higher priority task to be ready on that core).

16.3 Board Support Package (BSP)

The BSP is responsible for low-level hardware initialization, such as setting up interrupt vectors, timers, clocks, and memory, as well as booting secondary cores in a multi-core system. For the dual-core MIPS like architecture, we extended the BSP to initialize both cores and ensure the system is set up properly before the scheduler starts.

Key responsibilities of the BSP initialization include:

- Setting the CPU clock frequency and configuring the system timer (SysTick) for periodic tick interrupts on each core.
- Initializing the interrupt controller and defining the interrupt service routines (ISRs) for timer interrupts and software interrupts (used for yields or inter-core interrupts).
- Releasing the second core from reset and starting its execution. Often, the primary core (core 0) brings up core 1 by setting its start address or using a special inter-processor interrupt.
- Initializing hardware peripherals (like UART for debugging, GPIOs, etc.) and setting up any required memory regions or cache configuration.

Below is a simplified code snippet for the BSP initialization on the dual-core MIPS like system:

```c
void bsp_init(void) {
    // Initialize system clock and timer
    initSystemClock();
    initSysTickTimer(); // Configure SysTick timer for periodic interrupts

    // Set up the interrupt vector table and handlers for both cores
    initInterruptController();
    registerInterruptHandler(SYS_TICK_INT, SysTick_Handler);
    registerInterruptHandler(SOFTWARE_INT, Yield_Handler);

    // Boot up the second core (core 1) if not already running
    startSecondaryCore();  // e.g., release core 1 from reset and set its start address

    // Initialize memory management unit (if applicable) and caches
    initMemorySubsystem();

    // Initialize other board peripherals (UART, GPIO, etc.)
    initUART();
    initGPIO();
}
```

In the above code, `SysTick_Handler` would be the ISR that triggers on each timer tick to increment the OS tick count and possibly request a context switch. The `Yield_Handler` might be a software interrupt used to handle explicit yield calls (for example, from `portYIELD()` in the context switcher or when a task yields). The function `startSecondaryCore()` is hardware-specific; on our platform, it writes the starting program counter for core 1 and then releases core 1 from reset, causing it to begin execution (where it will then call `bsp_init()` for core 1 or a simplified startup routine that joins the scheduler).

By the end of `bsp_init()`, both cores are up and running, the system timer and interrupts are configured, and the FreeRTOS scheduler can be started to begin multitasking on both cores.

16.4 Synchronization Mechanisms

In a dual-core system, tasks may need to communicate or share resources (like memory, peripherals, etc.), so synchronization is crucial. We utilize FreeRTOS mechanisms like semaphores and mutexes to prevent race conditions and ensure that tasks on different cores do not interfere with each other. FreeRTOS provides thread-safe synchronization primitives which we use without modification, as they are inherently safe to use across multiple cores as long as the kernel is properly managing critical sections and interrupts.

For example, if two tasks (possibly running on different cores) need to access a shared resource, we protect that access with a mutex. If one task holds the mutex, another task (even on the other core) that tries to take the mutex will block until the mutex is released. The FreeRTOS kernel handles the blocking and waking of tasks under the hood, even in our dual-core context.

Below is a simple example of a task function using a mutex for synchronization:

```
SemaphoreHandle_t xMutex; // global mutex handle

void task_function(void *pvParameters) {
    // Task code that needs to access a shared resource
    for(;;) {
        xSemaphoreTake(xMutex, portMAX_DELAY); // Acquire the mutex (
           block indefinitely until available)
        // --- Begin Critical Section: access shared resource here ---
        sharedVariable++;
        // (perform other operations on shared resource)
        // --- End Critical Section ---
        xSemaphoreGive(xMutex); // Release the mutex

        // Continue with non-critical work or yield
        vTaskDelay(10);
    }
}
```

In this example, `xMutex` is a binary mutex (created elsewhere using `xSemaphoreCreateMutex()`) that protects access to `sharedVariable`. Even if two instances of `task_function` run on different cores, the mutex ensures that only one at a time enters the critical section. FreeRTOS handles the blocking of the second task and the resumption once the mutex is free. This prevents race conditions and ensures data integrity.

For other synchronization needs, we similarly use semaphores (for signaling events between tasks or ISRs and tasks) and queues (for message passing). These facilities are part of FreeRTOS and work in a multi-core setup with the kernel managing atomic access by briefly disabling interrupts or using atomic instructions where appropriate.

Chapter 17

FreeRTOS Port-Specific Definitions

The following definitions are crucial for the FreeRTOS implementation on the dual-core MIPS like system (these come from FreeRTOS configuration and port layer):

- `configUSE_16_BIT_TICKS`: Defines whether the system uses 16-bit or 32-bit tick counters. If set to 1, the tick count (and related timing calculations) use 16-bit integers, otherwise 32-bit (default for our 32-bit MIPS like system is 0, using 32-bit ticks).
- `portNUM_CONFIGURABLE_REGIONS`: Defines the number of memory regions that can be configured for Memory Protection Unit (MPU) support. This is relevant if using FreeRTOS MPU features. For our purposes, this may remain at the default (typically 0 or a small number) since standard FreeRTOS ports without MPU do not use configurable memory regions.
- `TickType_t`: The data type used for the OS tick count. It is an unsigned integer type. With `configUSE_16_BIT_TICKS` set to 0, `TickType_t` is typically defined as `uint32_t` (32-bit unsigned).
- `UBaseType_t`: An unsigned base type used by FreeRTOS for variables like loop counters, task priorities, and handle counts. On a 32-bit architecture like MIPS like, `UBaseType_t` is typically a `uint32_t`. (On 64-bit or 8-bit architectures, this type adjusts accordingly to the natural word size for efficiency.)
- `StackType_t`: The data type used for a task's stack entries. On a 32-bit system, this is usually defined as `uint32_t` (since the stack is an array of 32-bit words). Each task has an array of `StackType_t` allocated for its stack.

These definitions ensure that the kernel and application use consistent data types. For example, using a 32-bit tick count allows our system to run for a very long time before the tick count overflows (which is important for long uptimes), whereas a 16-bit tick count would overflow faster (after 65535 ticks). We chose to use 32-bit ticks given the capability of our 32-bit MIPS like cores and the requirement for long-running tasks.

Chapter 18

Verification via x86 Simulation

18.1 Purpose of Integration

Before deploying our FreeRTOS-based OS on the actual dual-core MIPS like hardware, we wanted to thoroughly test its functionality and correctness. For this purpose, we integrated FreeRTOS with an x86 architecture environment (specifically, using the Windows FreeRTOS simulator). This allowed us to simulate the dual-core behavior and verify the scheduler, context switching, and synchronization in a controlled setting where debugging is easier. By using a PC-based simulation, we could also gather performance metrics and identify issues early, without needing the physical hardware powered on for every test.

18.2 Changes to FreeRTOS for x86

Running FreeRTOS on a Windows (x86) environment required some adaptations since a standard PC/Windows is not a real-time system and does not provide the same hardware features as a microcontroller. We made the following changes for the simulation:

- **Timer Simulation**: Replaced the hardware SysTick timer with the Windows `SetTimer` function to generate periodic callbacks. This simulates the periodic tick interrupt by invoking a callback function at a regular interval (in our case, 1 ms for a 1 kHz tick rate).
- **Interrupt Handling**: Because we cannot directly manipulate hardware interrupts in user-mode Windows, we simulated interrupts using threads or timer callbacks. The FreeRTOS tick increment and context switch trigger are called from the `SetTimer` callback (which acts as our tick ISR). Additionally, to simulate a software interrupt or yield, we used a Windows event or a higher-priority thread to trigger context switches.
- **Task Execution as Threads**: Each FreeRTOS task is run as a separate Windows thread at a low priority. The FreeRTOS scheduler doesn't truly context-switch the CPU as it would on a microcontroller; instead, it controls the suspension and resumption of these threads to simulate task switching. This approach is used in the FreeRTOS Windows port so that tasks can be managed in a way that respects the FreeRTOS scheduling decisions.
- **Simulated Peripherals**: For example, where our MIPS like hardware might use a UART or GPIO, in the simulation we use standard I/O calls. For instance, a task toggling an LED on hardware would simply print a message or toggle a GUI element on Windows. We implemented dummy functions for GPIO and UART that print messages to the console, to simulate the hardware behavior.

These modifications allow the core logic of our OS to run unmodified, with only the hardware-specific parts abstracted. The FreeRTOS kernel itself (scheduler, queues, semaphores, etc.) remains the same; we primarily changed the *port layer* (the part of FreeRTOS that is specific to the CPU architecture).

18.3 Simulation Code

In the Windows simulation, the key part of the port layer is setting up the timer to generate tick interrupts and handling context switches. We use the Win32 API function `SetTimer` to call a function `vPortIncrementTick` at a regular interval (1 ms). This function increments the FreeRTOS tick and calls the scheduler. If a context switch is needed (i.e., a higher-priority task was woken by the tick), it triggers a yield to switch tasks.

Below is a snippet of the code used in the Win32 port for the tick simulation and scheduler start:

```c
// This function is called by the Windows timer every 1ms to simulate
    the SysTick.
void vPortIncrementTick(HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD
    dwTime) {
     // Increment the FreeRTOS tick count
     if (xTaskIncrementTick() != pdFALSE) {
         // If a context switch is required (i.e., a task with higher
            priority than current was unblocked)
         portYIELD(); // request a context switch yield
     }
}

// Override the FreeRTOS scheduler start function for Windows port
void vTaskStartScheduler(void) {
     // Create a Windows timer that calls vPortIncrementTick every 1 ms
     SetTimer(NULL, 0, 1, (TIMERPROC) vPortIncrementTick);
     // Start the first task manually (this sets up the initial task
        context on the current thread)
     vTaskStartFirstTask();
     // The scheduler is now running. In the Windows port, tasks are
        managed as threads.
     // vTaskStartScheduler will not return unless an error occurs.
}
```

In the above code:

- `xTaskIncrementTick()` is a FreeRTOS kernel function that increments the tick count and checks if any task's delay has expired or if any timeouts occurred. It returns `pdTRUE` if a context switch is needed at the end of the tick.
- `portYIELD()` in the Windows port will signal the FreeRTOS scheduler to switch tasks. In the MIPS like hardware, this would correspond to triggering a software interrupt or using the `Yield` assembly instruction.
- `vTaskStartFirstTask()` is a FreeRTOS internal function that starts the first task running. In the Windows port, this sets up the current thread context to the first FreeRTOS task. After this, the FreeRTOS scheduler logic (now driven by the Windows timer interrupts) will take over managing which task (thread) runs.

With this setup, our tasks can be created and run in the Windows environment as if they were running under the FreeRTOS scheduler on actual hardware. The timing won't be real-time exact (Windows is not a real-time OS, so the 1ms tick can jitter), but it's sufficient for functional testing and approximate performance measurement.

18.4 Testing and Results

To validate the system on the x86 simulation, we set up a benchmark scenario with two example tasks. The goal was to stress the scheduler and measure context switch times, CPU utilization, and general stability.

18.4.1 Benchmark Task Scenario

We created two tasks for the test:

- **TaskA**: A high-priority task that continually increments a counter and periodically yields. This task simulates a CPU-intensive high-priority workload.
- **TaskB**: A low-priority task that periodically prints the counter value. This task simulates a background task that should be preempted whenever TaskA is ready to run.

By running these two tasks, we force frequent context switches (since TaskA will run, then yield or block briefly, allowing TaskB to run, and then TaskA immediately preempts again due to its higher priority). This pattern is ideal for measuring the context switch overhead and scheduler efficiency.

Below is the code for these two tasks and the `main` function that sets up the test in the simulation environment:

```c
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

// Shared counter and mutex for thread-safe access
static int sharedCounter = 0;
static SemaphoreHandle_t xMutex;

// High-priority task: increments a counter continuously
void vHighPriorityTask(void *pvParameters) {
    for (;;) {
        // Simulate work by incrementing a shared counter
        xSemaphoreTake(xMutex, portMAX_DELAY);
        sharedCounter++;
        xSemaphoreGive(xMutex);
        // Yield to let other tasks run (simulating periodic work)
        vTaskDelay(1); // delay 1 tick (1ms in our tick rate)
    }
}

// Low-priority task: periodically prints the counter value
void vLowPriorityTask(void *pvParameters) {
    for (;;) {
        xSemaphoreTake(xMutex, portMAX_DELAY);
        int value = sharedCounter;
        xSemaphoreGive(xMutex);
        // Print the counter (simulating output, e.g., to UART)
        printf("LowPriorityTask:␣Counter␣=␣%d\n", value);
        // Delay for some ticks to simulate slower periodic activity
        vTaskDelay(5); // delay 5 ticks (5ms)
    }
}

int main(void) {
    // Initialize the mutex before creating tasks
    xMutex = xSemaphoreCreateMutex();

    // Create the high and low priority tasks
    // Higher number means higher priority in FreeRTOS (
        configMAX_PRIORITIES is configured accordingly)
    xTaskCreate(vHighPriorityTask, "HighTask", configMINIMAL_STACK_SIZE
        , NULL, 2, NULL);
    xTaskCreate(vLowPriorityTask, "LowTask",  configMINIMAL_STACK_SIZE,
        NULL, 1, NULL);

    // Start the FreeRTOS scheduler (this will also start the Windows
       timer for ticks)
    vTaskStartScheduler();

    // We should never reach here as the scheduler will be running
    // If we do, it likely means there was insufficient heap memory to
       start the scheduler.
    return 0;
}
```

In the above code, `vHighPriorityTask` and `vLowPriorityTask` operate on a shared counter protected by a mutex (`xMutex`). TaskA (`HighTask`) runs with higher priority (priority 2) and increments the counter every iteration, yielding briefly with `vTaskDelay(1)` to allow TaskB to run. TaskB (`LowTask`) runs at lower priority (priority 1) and prints the counter value every 5 ticks. In a correct implementation, TaskB will only run when TaskA is in *delay* state, and as soon as TaskA becomes ready again (after 1 tick), it will preempt TaskB.

We instrumented the code to measure performance metrics. For example, to measure context switch latency, we recorded timestamps (using high-resolution timers provided by Windows) around the `portYIELD()` call and the actual task start to calculate how long the context switch took. We also measured CPU utilization of the simulation program to estimate scheduler overhead.

Chapter 19

Software Implementation Conclusion

In this project, we extended FreeRTOS to support a dual-core MIPS like architecture by modifying the scheduler, context switching mechanism, and BSP initialization. We tried validating our implementation by integrating it with an x86 (Windows) simulation, which allowed us to run the OS in a controlled environment. The simulation was done to prove that tasks are scheduled correctly across both cores, context switching overhead is low, and synchronization between tasks works reliably.

We expect normal behavior on hardware, with potentially even better performance due to the real MIPS like cores and lack of underlying OS overhead. This implementation demonstrates that FreeRTOS can be adapted to multi-core processors. Future work could involve further optimizing the scheduler for load balancing between cores and exploring more complex inter-core communication mechanisms for enhanced performance.