PROJECT

“DUAL-CORE MIPS32 PROCESSOR WITH MEMORY ARBITER”

THIS PROJECT IS AN EXTENSION OF THE SINGLE CYCLE MIPS32 PROCESSOR

https://github.com/JayaKushal24/RTL_Single-Cycle-MIPS32-Processor

BY
YANDARAPU JAYA KUSHAL
https://github.com/JayaKushal24

Design Steps and Module-Wise Modifications

1. Processor Core Modules (MIPS32_core0 and MIPS32_core1)

- I started with a single-cycle MIPS32 core and modified it to support stalling.
- The **stall logic** ensures that if a core requests access to shared memory and the arbiter does not grant it, the core pauses its program counter and disables register and memory writes.
- Once this logic was tested, I duplicated the modified core to create two identical cores, allowing each to run independently.

2. Instruction Memory Modules
(instruction_memory_Core0 and instruction_memory_Core1)

- Each core is connected to its own instruction memory (ROM), so both can execute different programs simultaneously.
- I initialized each ROM with a unique instruction sequence tailored to test both independent and shared memory operations.

3. Memory Arbiter Module (memory_arbiter)

- I designed and implemented a memory arbiter module to manage access to the shared data memory.
- The arbiter receives memory requests from both cores and grants access to only one at a time, using a round-robin policy to ensure fairness.
- It multiplexes address, data, and control signals from the granted core to the RAM and routes read data back to the correct core.

4. Top-Level Integration (Dual_core_top)

- In the top-level module, I instantiated both core modules, both instruction ROMs, the arbiter, and a single shared data memory.
- All data memory access signals from both cores are routed through the arbiter, ensuring safe and exclusive access.
- The top-level wiring connects all handshake, address, data, and control signals between the cores, arbiter, and RAM, enabling the system to run both cores in parallel.

5. Testbench

- I wrote a testbench that instantiates the top-level module, generates a clock and reset, and allows both cores to execute their programs.
- The testbench is used to verify correct operation, including arbitration, stalling, and memory sharing.

Key Features and Outcomes

- Parallel Execution: Both cores can execute their own programs independently.
- Shared Data Memory: Both cores access a single shared RAM, with safe arbitration.
- Stall Handling: Cores pause automatically if memory access is not granted, ensuring correctness.
- Fair Arbitration: The arbiter uses a round-robin scheme so both cores get fair access to memory.
- Scalability: The design can be extended to more cores or more complex arbitration policies.

Summary Table

Change	Single-Cycle MIPS32	Dual-Core MIPS32
Number of cores	1	2 (core0 and core1)
Instruction memory	1	2 (one per core)
Data memory access	Direct	Through memory_arbiter
Arbiter signals	Not present	MemRequest, MemGrant per core
Stall logic	Not present	stall signal in each core
PC gating	Not needed	Gated by stall
Top-level connections	Simple	Cores, arbiter, and shared RAM interconnected
Testbench	One core	Dual core, runs both in parallel

STALL LOGIC

Stalling is a standard and necessary technique in real-world multiprocessor and pipelined CPU systems

Modern CPUs use a combination of stalling, data forwarding, branch prediction, and out-of-order execution to minimize the performance impact of hazards

What is Stall

When multiple processors (or cores) share a single memory or bus, only one can access the memory at a time. If two or more cores try to access the memory simultaneously, an arbiter decides which one gets access. The others must **wait**—this is called a **stall**.

- **Stall** means the processor temporarily pauses its progress until it is granted access to memory
- Without stalling, a processor might proceed with invalid or incomplete data, causing errors and unpredictable behavior

Importance of Stall

- Correctness: Prevents a core from executing instructions that depend on memory access before the data is available
- Data Integrity: Ensures only one core writes or reads from memory at a time, avoiding data corruption
- Orderly Sharing: Allows fair and predictable sharing of memory resources between multiple cores or devices

The changes I made in Code:

1. Stall Signal Generation

stall goes high when the core wants to access memory but does not have permission from the arbiter

```
wire is_mem_instr = MemRead_wire | MemWrite_wire;
 wire stall = is_mem_instr & ~MemGrant;
```

2. Program Counter Stall Support

modified the program counter instantiation so that it only updates if not stalled:

```
program_counter PC (
 .clk(clk), .reset(rst), .stall(stall), .pc(pc_wire), .next_pc(pc_out_wire)
 );
```

When stall is high, the PC holds its value, so the same instruction is retried

Modified the PC code as well

3. Register File Write Gating

gated the register file write enable with ~stall:

```
registerts Reg_mem(
.clk(clk),
.reg_write( RegWrite & ~stall), // Reg write disabled if stalled
... );
```

Prevents the register file from being written during a stall, avoiding incorrect updates.

4. Memory Write Gating

gated the memory write signal with ~stall:

```
// Mem write only if not stalled..specifically for dual core
assign MemWrite = MemWrite_wire & ~stall;
```

Ensures the core only writes to memory when it has the grant.