**Electronic System Level Design Methodology – Term Project**

**RISC-V Dual core system with DMA.**

N26100595 李秉軒

**Part I**

In this part, I explore multiple examples in OVP platform, and find 5 examples that help me design the system in part II.

**HelloWorld**

There are multiple ways to construct the platform in OVP, including using OP, iGen, ISS, and SystemC. Here we introduce the one using SystemC since we would use SystemC in part II.

There are two folders in this example, *application* and *platform*. In *application*, *application.c* is the *Hello world* program. In *platform*, *platform.cpp* is the top module of the platform.

In this platform, a RISC-V CPU, a RAM and a bus module is added. Note that we should do the connection through the interface provided by OVP, i.e., the *connect()* function.

Following the simulation result.

一張含有 文字 的圖片

自動產生的描述

**Creating DMAC**

In this example, a DMA is added to the system as a peripheral. Even though the behavior of DMA is written using *pse*. This example still helps me understand how to control a peripheral and handle interrupt in software side.

There are multiple parts in this example. The first part is adding the control registers of DMA, which requires DMA to connect to bus with a slave port. The second part is waiting for the DMA to complete through busy waiting.

The third part is the same, but the DMA can work now. This requires DMA to connect to bus through a Master port.

The last one is adding Interrupt. First, we need to control CSR. Second, we need to add interrupt handler. Last, we can check interrupt using *wfi* instruction instead of busy waiting.

Here is the result of running the example in *4.interrupt*.

一張含有 文字 的圖片

自動產生的描述

As we can see, the program will wait for interrupt after sending the DMA request. After receiving interrupt, the program counter will jump to interrupt handler and clean the interrupt, then send another request.

**Single-Core RV32G ISS**

In this example, we use an RV32G ISS to simulate the program. An ISS has built-in memory so there is no need to add a memory. Honestly, this example isn’t that useful for Part II. However, it’s quite interesting for me.

Here is the result of running *PeakSpeed2* benchmark on the ISS.

一張含有 文字 的圖片

自動產生的描述

Before the example, I assume that using software to simulate hardware will always be slower. However, using ISS can speed up the simulation by 94.71 times faster than real hardware.

I also run the Fibonacci benchmark, which is much slower. I think that’s because this benchmark has a lot of recursive function call, which requires to copy register file into memory and cannot apply some optimizations, like loop unrolling.

**SystemC TLM**

In this example, an ***uart*** module is added as a peripheral using SystemC\_TLM. This example shows us how to do add a peripheral into our system using SystemC. Also, how to bind the interrupt from peripheral to CPU is what we need.

Combining this example with the DMAC one, we can know how to add a peripheral and how to control it.

一張含有 文字 的圖片

自動產生的描述

However, in this example, something bothers me a lot. In the DMAC example, we need to explicitly tell CPU the address of interrupt handler. I cannot find where we give CPU that information in this example.

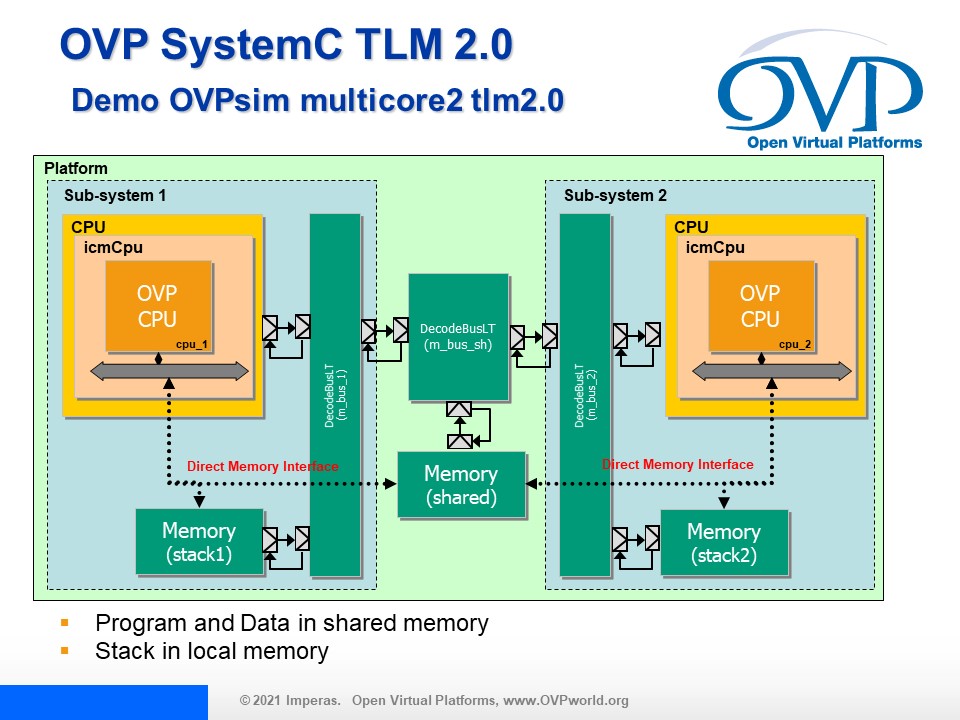
**multicore2\_mips32\_tlm2**

This example shows how to add two CPU cores and make them communicate through shared memory. Each of CPU has a local bus connected to it. They can access instruction/ data memory through it.

Both local buses are connected to system bus. And the system bus has a shared memory connected to it. Therefore, both CPU can access this shared memory.

In this example, only one CPU can access the shared memory at a time. They use a flag to acquire the permission and release it, which is very much like mutex in OS.

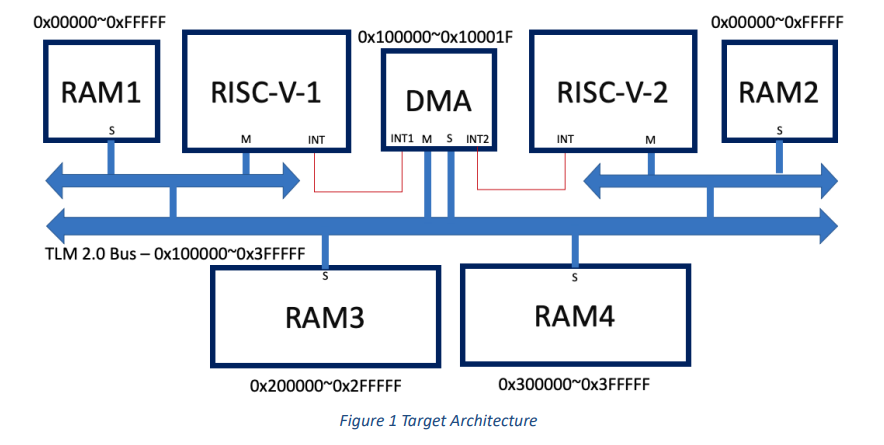
The system architecture is as follow.



This is very much like our target system in Part II. Now we have everything we need. We can hop into Part II now!

**Part II**

In this part, we are required to implement a dual-core system on OVP, the architecture of the system is as follow.



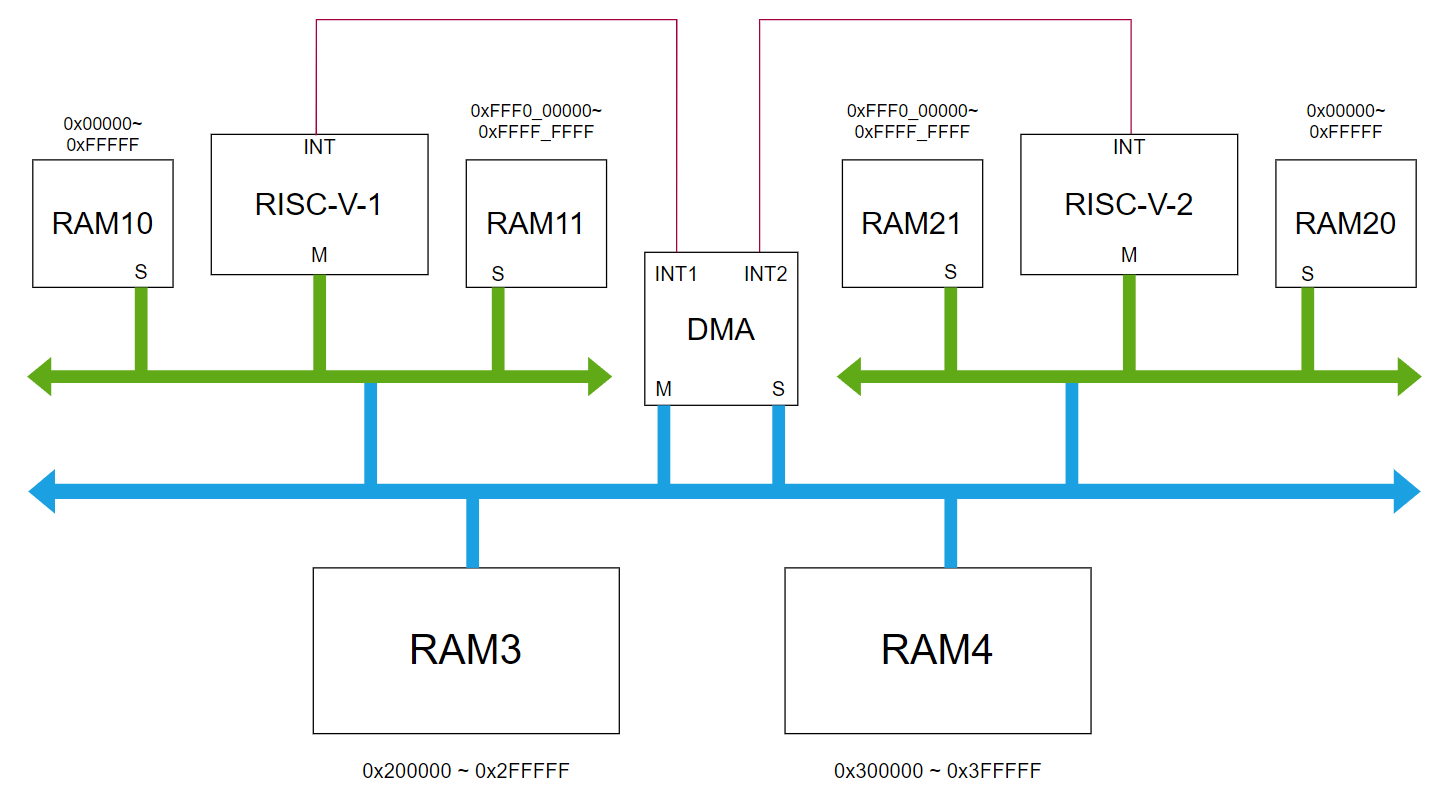
**Modifications**

However, the system requires some modifications. First, the compiler will assume the address of data memory to start from the upper bound, i.e.,0xffff\_ffff, putting the data memory at 0x0\_0000~0xffff\_f will make the bus unable to find the data memory.

Secondly, the main system bus is registered to the local bus at 0x10\_0000. When we access the shared memory through the local bus, a bus offset will be included, making the address passed to system bus to be wrong. For example, we want to access RAM3, so we send a request to the local bus at 0x20\_0000. However, the local bus will reduce the bus offset, making the address passed to system bus to be 0x10\_0000.

Therefore, we need to add a bus offset when passing request to the local bus.

The modified architecture is as follow.



**Interrupt mechanism**

The interrupt mechanism is hugely inspired by the creating DMAC example, we will briefly introduce how it works in this section.

First, let’s see how the interrupt works in RISC-V CPU design.

***CSR***

A RISC-V CPU must have a set of registers called *Control Status Register (CSR)* and supports a set of instructions that can access CSR. At the beginning, we need to use CSR instructions to set CSR so that CPU can run the ISR correctly.

The attributes of CSR that matters in this project is *MIE*, *mstatus*, and *mtvec*. **MIE** (Machine Interrupt Enable) defines what kind of interrupt is enabled. We need to enable *MEIE* *(Machine External Interrupt Enable)* at the beginning so that CPU can receive interrupt from DMA.

**Mstatus** stands for *Machine Status*, we need to set it to enable interrupt, so that we can enter ISR properly. Noted that CPU will set it to disable interrupt during ISR.

**Mtvec** stands for *Machine trap vector*. It stores the address of ISR. We need to set it to our interrupt handler.

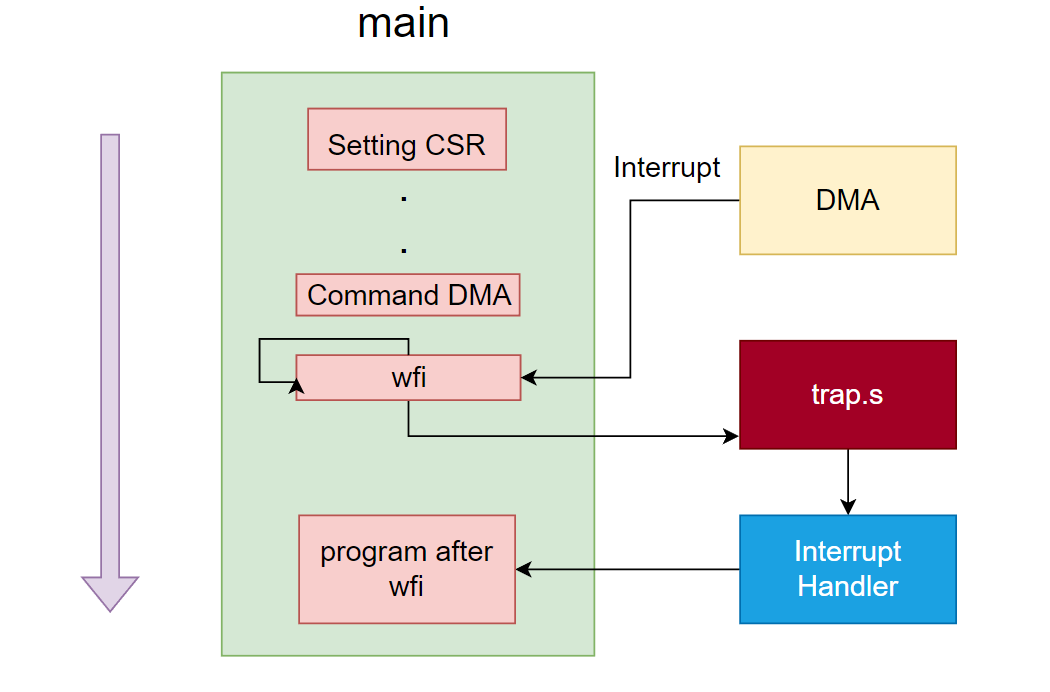
***ISR***

We need to write an assembly code as our ISR, i.e., **trap.s**. In ISR, all data in register file will be stored into memory. After that, ISR will jump to our interrupt handler written in C. In this interrupt handler, the Start-Clear register of DMA will be cleared.

After that, retrieve all data from memory and use **mret** instruction to return to main program

After we have finished *trap.s* and an interrupt handler. We can start adding interrupt in our main program. At the beginning, we need to use CSR instruction to enable MEIE in MIE and set Mstatus to enable interrupt.

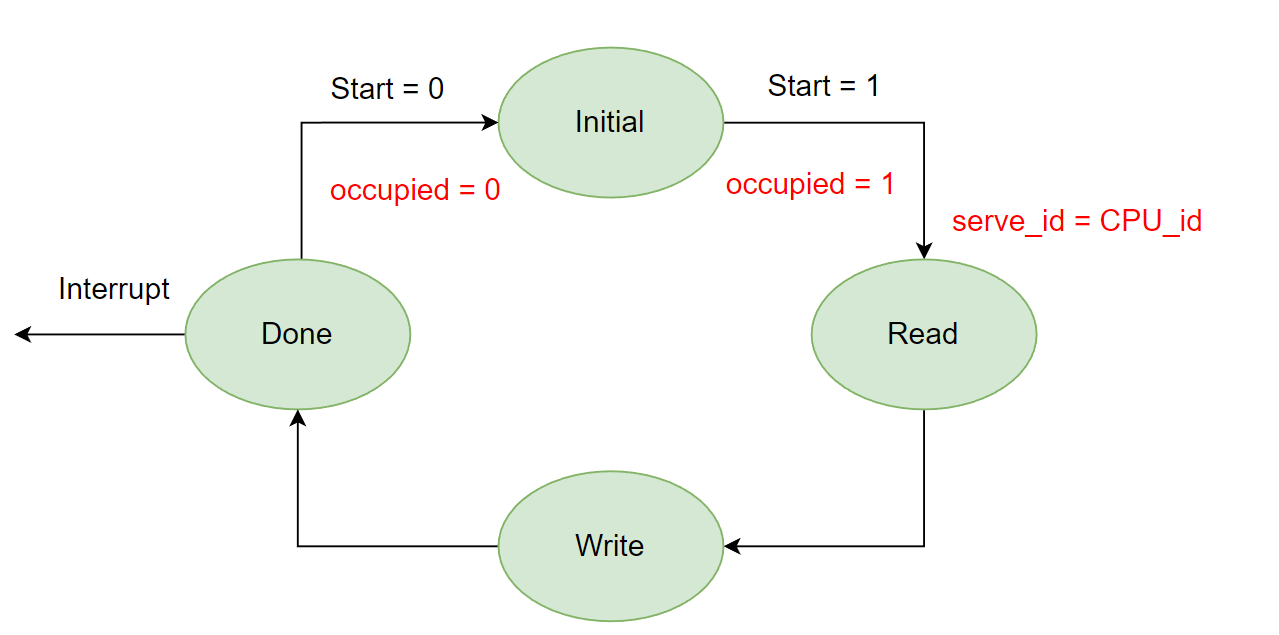
After sending commands to DMA, use *wfi()* instruction to wait for interrupt from DMA.



**DMA design**

In this project, our DMA requires 2 sets of control registers, controlled by 2 CPU respectively. Since 2 CPUs might try to command CPU in the same time(although it won’t happen.), a First-Come-First-Serve policy is adopted.

We use a state machine to control DMA. The state graph is as follow.



Initially, DMA will wait to be accessed. When a *b*\_*transport* modify the control register, DMA will set the ***occupied*** variable to 1 and the ***serve*\_*id*** variable to the index of CPU. The *occupied* variable will block all the request from another CPU. After CPU set the *Start* attribute to 1, DMA moves to next state.

DMA will first read data from *Source* address, then write to *Target* address. In this project, we finish the data transfer in a single beat, i.e., we transfer all data in a single action. After that, DMA moves to *Done* state.

In *Done* state, DMA raises interrupt. DMA will be stuck in this state until the *Start* attribute to be set to 0 by the CPU. Then DMA will drop interrupt, set *occupied* to 0, and wait for next request.

**Simulation result**

Here is the simulation result:

Initially, both CPU will write data to RAM3 and RAM4 respectively. Then, since monitor address 0x200800 is reset to 0, RISC-V-1 will command DMA first.

一張含有 文字 的圖片

自動產生的描述

After receiving Interrupt, RISC-V-1 will dump the content in RAM4 and set the monitor flag to 1.

After that, CPU2 will start commanding DMA. After receiving Interrupt, RISC-V-2 will dump the content in RAM3 and set the monitor flag to 0.

一張含有 文字 的圖片

自動產生的描述

Following is the memory dump after first iterations.

一張含有 文字, 電腦, 檔案 的圖片

自動產生的描述 一張含有 桌 的圖片

自動產生的描述

After 3 iterations, both programs will stop. Then the simulation has completed. Following is the simulation detail of both CPUs.

一張含有 文字 的圖片

自動產生的描述

**Conclusion**

In this project, I built a dual-core RISC-V system with DMA on Open Virtual Platform. In part I, I introduce multiple examples that help me design and implement this system.

In part II, I introduce the system architecture after modifications. I detailly introduce the Interrupt mechanism and the design of my DMA. Simulation result shows that our dual-core system can work with DMA properly.

Huge thanks to Dr. Su for this ESL course. Through this course, I learn the concept of ESL and knows how to use it to help me build my system. I wish your start-up will have a very promising future. Good luck!