Address Translation & IRAM Cache

Members:
ori.yeffet@mail.huji.ac.il – Ori Yefet
omrids@gmail.com – Omri Dassa
Group 327

Advisor:
nadav.rotter@intel.com – Nadav Rotter

The Problem?

NIC – Hardware component used by computers/servers for communication on a network system.

This project address 2 main components of an Ethernet NIC (Network Interface controller):

- A typical NIC has several CPU controllers which are processing packets of data and rout them according to standard protocols. Those controllers follow a set of commands which reside in the IRAM (Instruction RAM). In order to change or add functionality to the controller's by demand, one can load a different/upgraded set of commands to the IRAM allowing more/other features.
- basic requirement of today NICs is to support several software entities using shared resource. The NIC contains a single public memory space that every software entity[1] need to access (by memory address) while considering itself the sole owner of the memory with no dependency of other software entities.

The 2 problem that we will face in our project are:

- The IRAM is fixed and confined, thus can only support limited number of features.
- By using a system that support several software entities with a public memory there is a need for efficient and fast Address Translation[2] unit which will translate virtual addresses into unique physical addresses.

The Key Challenges Involved?

Our solutions to both of the problems must hold the followings:

- Must not exceed a bounded area on the NIC.
- Must meet performance specifications
- Must be Versatile and parametric in order to be reused in future projects.

[1] Software entity – is a software process that controls a CPU controller which is hardware device. Every Software entity that tries to communicate with a CPU controller doesn't know that there are another Software entities that also communicate with a CPU controller.

[2] Address translation unit translate a request from Software entity into a hardware shared resource.

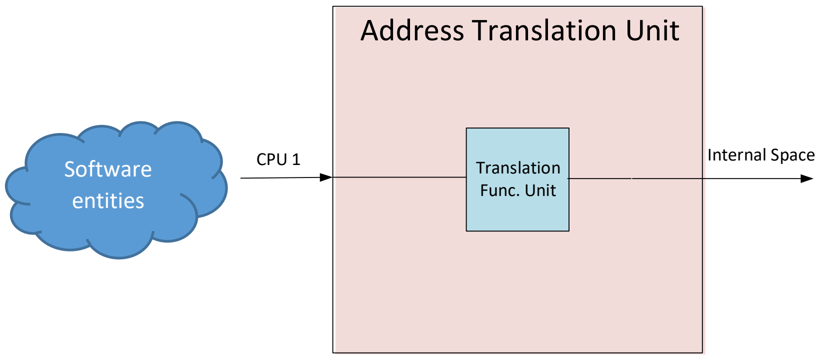
Strategy and Solution Approach?

Our approach is divided to several stages and we will elaborate on each of the components in separate. First, we will investigate older projects and study the CPU controller to IRAM flow as well as the address translation units of non-parallel compute systems. The second stage of our approach will be to stich a solution to similar but easier problems. The last stage will include a complete solution for both of the problems.

Address Translation:

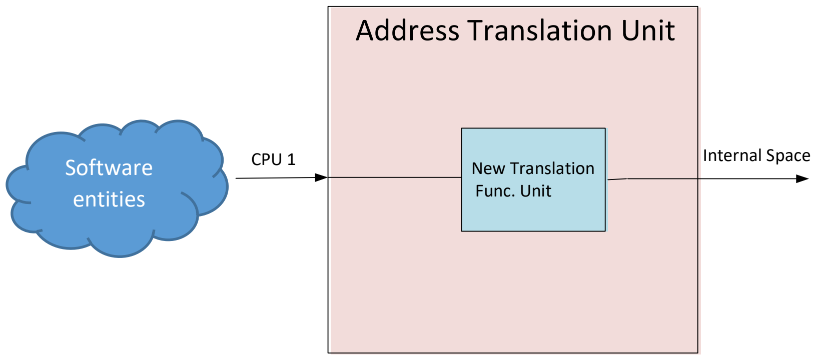
Investigation:

When a single software entity uses the shared resources it can consider itself the sole owner of those resources, as it is indeed the only user of the resources, thus an injective (according to the number of software entities) function can be used in order to translate virtual addresses (from the software entity) to physical addresses (resource).



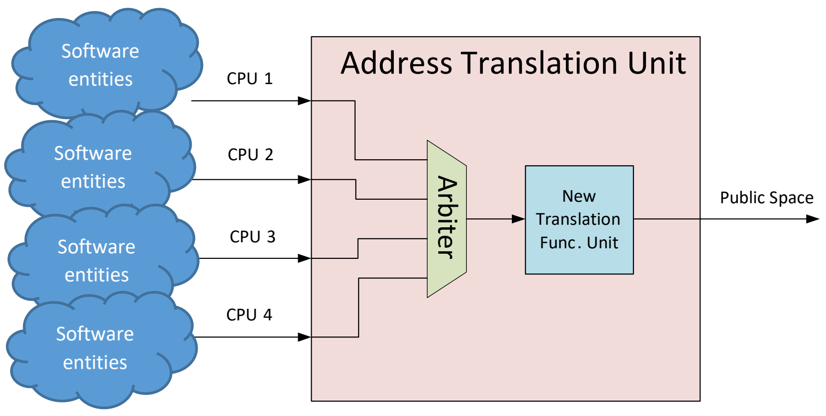
Partial Solution:

As the solution for the Address Translation consist of both translation function and arbitration between several CPU controllers, we will first solve the new translation function which should be able to translate from several CPUs to a single shared resource but will test it only with a single CPU.



Complete Solution

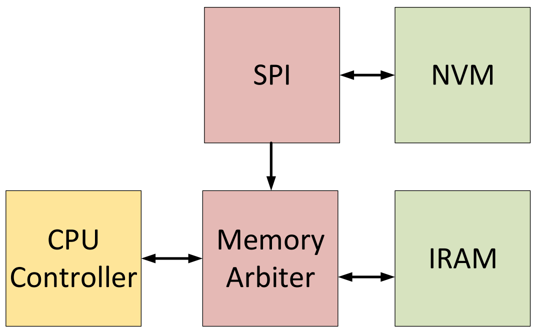
In our last stage of the solution we will add a fair arbitration between the different CPU controllers in the Address Translation unit and check the translation function for requests from several CPUs. This complete solution is offering a way for number of compute units to use one shared memory.



IRAM Cache:

<u>Investigation:</u>

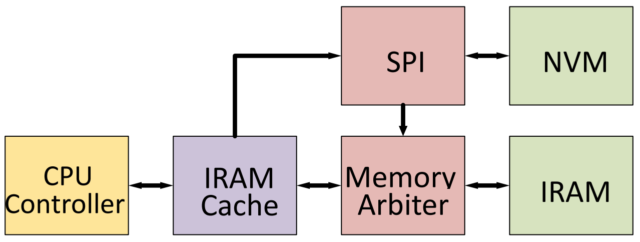
As we know a RAM type memory does not keep its values when the power is off, so in order to load instruction to the IRAM a fetching mechanism must be applied on boot sequence from a NVM (Non Violated Memory) which is a slower but larger memory. After the boot sequence CPU controller's requests are answered by a Memory Arbiter that fetches data from the IRAM.



<u>Partial Solution:</u>

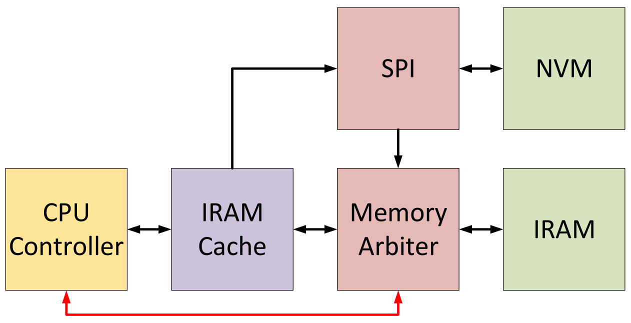
in order to expand the IRAM Memory address space we will be using a caching mechanism and make the IRAM entirely cached. Meaning that each request for instruction from the CPU controller will first be compared to identify whether the data reside in the IRAM or not. In case it does, a natural flow back to the CPU will be used, in case it doesn't, the IRAM Cache mechanism will fetch that request from the NVM.

This solution will let us use both the IRAM and NVM memory space so we can store more instruction to more features, as well as benefiting from the IRAM speed for the most used features.



Complete Solution

Regarding the IRAM Cache mechanism we would like to add a feature which will enable the user to lock a part of the IRAM memory space. By doing so ensuring the instruction data of this locked memory will not be evacuated from the IRAM and be replaced. This uncached space may be used for features that are required to meet performance.



Evaluation and Verification

The three main issues that must be addressed in order to evaluate our project are:

- Timing – both of the components described above must meet timing constraints driven from the clock frequency. Meaning the delay time of the logic in use must not exceed the clock cycle. Also, because both of the components intervene in an already existing flows our solution will be compared to the older solutions.
- Size – the components will be measured in terms of Flip-Flops, latches and combinational cells and must not exceed the size defined by the NIC architects.
- Correctness – the components obviously must do what they were designed to do.

In order to evaluate those parameters we will use 2 tools. The first will be executing a synthetizing tool which will count and measure both size and timing. The second will be to build a test-bench for the components which will check their correctness.

Schedule

Omri Dassa		Ori Yefet		Date
Priority	Task	Priority	Task	
P0	Verification course for designers + Advanced Design Concepts course	P0	Verification course for designers + Advanced Design Concepts course	August - September
P0	Asynchronous-FIFO Task + Investigating Translation & Arbitration methodologies	P0	Asynchronous-FIFO Task + Investigating Cache & Cache Eviction methodologies	October
P0	Investigating adjacent blocks + Writing MAS*	P0	Investigating adjacent blocks + Writing MAS*	November
P0	Continuing writing MAS + revising adjacent blocks	P0	Continuing writing MAS + revising adjacent blocks	December
P0	Coding Translation Function for a single CPU	P0	Coding Full-Mapping	January
P0	Debug	P0	Debug	February
P1	Coding Arbitration of several CPUs	P1	Coding Partial-Mapping	March
P1	Debug + Synthesis + Formal Verification	P1	Debug + Synthesis + Formal Verification	April
P2	Formal Verification + Debug	P2	Formal Verification + Debug	June

*MAS – Micro Architectural Specifications