Tai Lung RV32I Project

HAS v1.0

(High-level Architecture Specification)

Authors:

Christian Shakkour, chrisshakkour@gmail.com

Shahar Dror, shdrth1@gmail.com

Git repository:

https://github.com/ChrisShakkour/RV32I-MAF-project

Release Information

DATE	AUTHOR	DESCRIPTION
02-FEB-2022	Chris	Created v1.0
06-FEB-2022	Chris	Table of contents created
04-MAR-2022	Shahar	Adds Memory content
01-APR-2022	Chris	Adds Core contents

Acknowledgements

Special thanks to our supervisor Amihai Ben David for contributing knowledge and experience throughout the development of this project. Many thanks to the VLSI Lab at the Technion - Israel Institute of technology for supporting and providing tools and resources that were an integral part of the development process.

About this document

In this document you will find information about the high-level Core Architecture, features, memory and much more. See MAS document for deeper dive into the Microarchitecture.

Table of Contents

Introduction

This project features a basic RISC-V CPU called Tai-Lung. The name Tai-Lung was inherited from a childhood movie called Kung Fu Panda where Tai-Lung was the evil Leopard, however the ISA RISC-V is an open-source Instruction set architecture from Berkeley university.

Tai-Lung

Tai-Lung project is a single Core RV32I CPU based on the RISC-V ISA. The CPU is Designed in synthesizable System Verilog and features an in-order execution 5 stage pipeline designed for low power microcontrollers for edge devices.

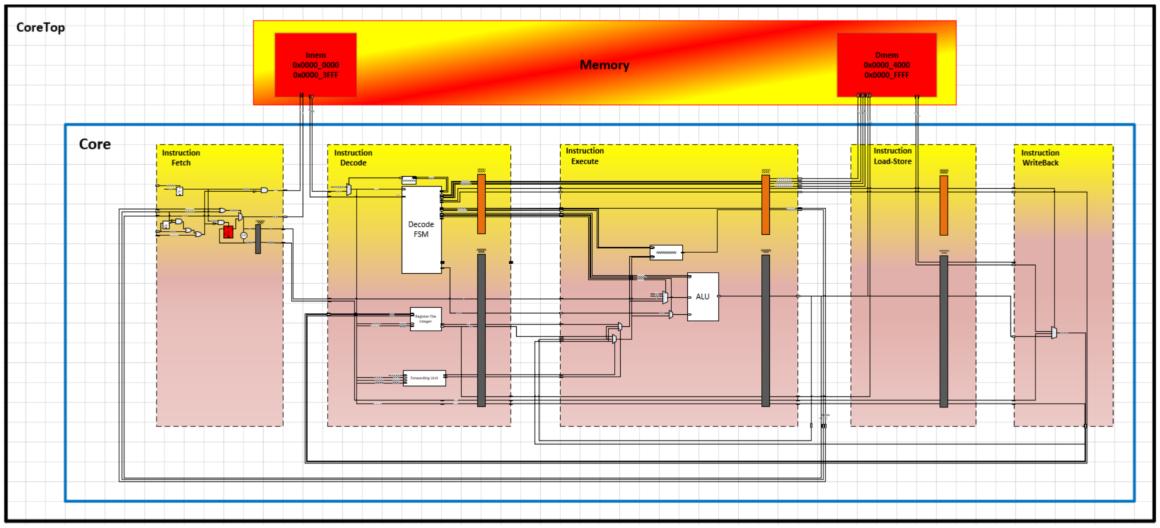
Specs

Tai Lung supports the main RV32I instruction set, in particular 37 instructions out of 47, the Excluded instructions are the ECALL(1), EBREAK(1), FENCE/I(2), and CSR instructions(6).

ISA	**RV32I**
Pipe stages	5
Forwarding	Supported
Branch prediction	Not supported
Frequency	TBD…
Power	TBD…
Die area	TBD…

Architecture

The architecture of Tai-Lung is simple, the memory and Core units are separated. Both units communicate via direct read/write access no AHB/APB agent in-between, the memory belongs only to this CPU and cannot be shared with other CPU’s.



Performance

The performance in ASIC's is usually measured by a few factors, CPU time, power, and Area. However, in this project Tai-Lung is just a behavioral model of a CPU hence no power and area analysis has been made. The focus will be on CPU time, this factor is measured by cycle time($\frac{1}{f}$), the instruction count(IC), and the cycles per instructions(CPI). Following equation:

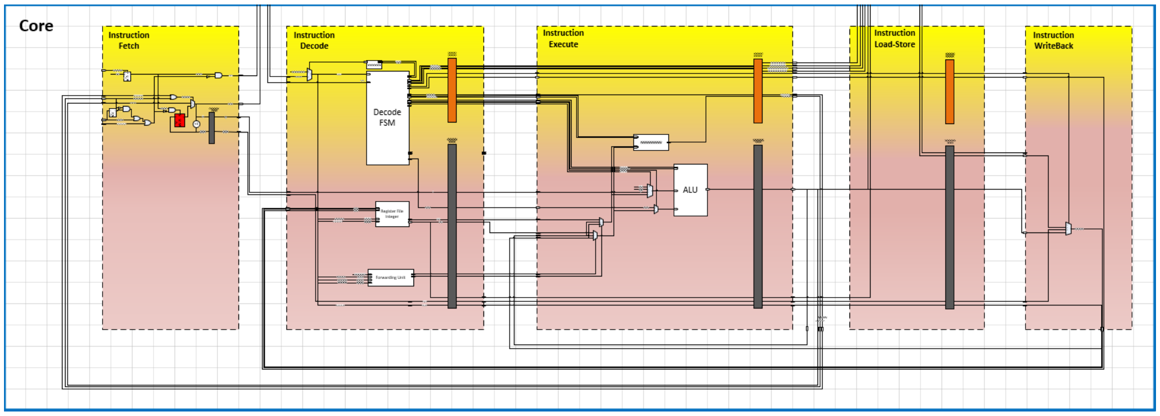
$$CPU_{time} = IC * CPI * \frac{1}{f}$$

The instruction count (IC) is application dependent, the frequency is dependent on the Design and the technology node. But the CPI is an architecture artifact that can be measured independently if we know the distribution of commands in a typical application, and the CPI of each command then the CPI can be approximated. However, here you will find a practical measurement of the IPC, where $CPI = 1/IPC$. Following a table of measured IPC from different tests.

Test	Instructions retired	Cycles	IPC	Description
Tower of Hanoi	28198	37924	0.743	N of disks = 10
Sudoku solver	3533	4909	0.720	Cube = 9x9
Fibonacci	470725	645863	0.728	Fib(20)
Binary search	652	751	0.868	N=200 elements

Core

Tai-Lung Core is an in-order 5 stage pipelined execution unit, with a forwarding unit to address the needs of data and control hazards in the figure below the yellow area is where most of the control is transferred, the other color is where the data is transferred, notice there are 2 types of pipe registers the Orange are asynchronous reset registers, and the Grey registers are Data registers with no reset or enable to reduce the area.



Pipe stages

(1) Fetch: this stage manages when and which commands are fetched from the instruction memory.
(2) Decode: when a new instruction is fetched it needs to be decoded, meaning the instructions is broken down to control bits that tell the rest of the units what needs to be done in this instruction, and data that is needed to execute the instruction.
(3) Execute: the execution unit performs the arithmetic calculations needed, weather it is adding two numbers, comparing them, or operating a bitwise XOR on them.
(4) Load-Store: for load this unit fetches Data from a specific address in the Data memory into the register file, and for store a Data is stored from the register file into the Data memory.
(5) WriteBack: in this stage the data is routed back from one of the data sources, loaded memory data, post arithmetic data, or Data stored in the register file back into the register file

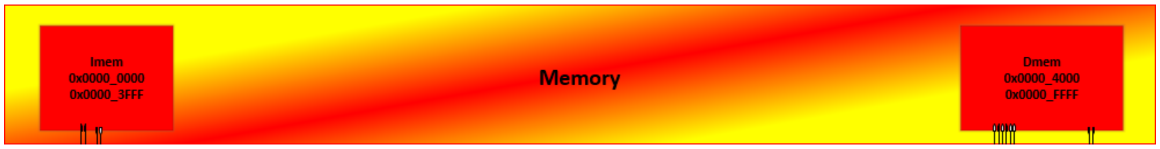
Boot flow

The boot flow is defined in the crt0 file, it prepares the CPU for execution right when the system is out of reset, the flow is defined by a set of instructions that clear and set values to registers, as well as memory regions that need to be preloaded with specific information before one can start executing the desired application.

When out of reset the control registers of the pipeline are set to Zero (the pipe is stalled), then the register file is cleared by setting all registers to Zero because the registers in the register file don’t have a reset signal so when introduced to power there will be garbage values that need to be cleared. One register which is the stack pointer register has a special value that it must be initialized to, and it is the address of the top of the stack frame. Then the arguments Argc, and Argv are set to Zero to prepare for calling the main program. After this is done the CPU can start executing commands from the main function.

Memory

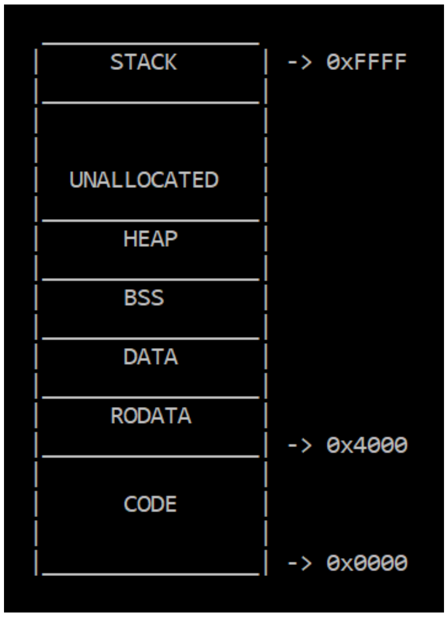
Tai Lung memory model is 64Kbytes in size and is divided into two independent memory blocks, one block called the instruction memory holds the application code only and the other called the data memory occupies the application data, heap, and stack sections.



MEMORY BLOCK	SIZE	ADDRESS RANGE	ACCESS	PORT	ENDIANNESS
I-MEM	16Kbytes	0x0000-0x3FFF	Read only - executable	Single	Little
D-MEM	48Kbytes	0x4000-0xFFFF	Read/Write – non executable	Single	Little

Address mapping

The following figure shows the different memory sections. Code section is held in the I-mem and the rest is occupied by the D-mem, on the right the addresses of those sections can be seen. note that the sizes of the D-mem sections are determined according to the application, the number of variables, the recursion calls, and so forth.



Memory Sections

1. CODE:
   The code segment, also known as text segment, contains executable code, it is fixed size, and read only, linker will dump all (.text) sections there.

2. RODATA:
   read only data section linker will dump the following sections (.rodata, .srodata, .rdata)

3. DATA:
   The data segment contains initialized static variables, global variables and local static variables which have a defined value and can be modified. Linker will dump the following sections (.data, .sdata)

4. BSS:
   The BSS segment contains uninitialized static data, both variables and constants, global variables and local static variables that are initialized to zero or do not have explicit initialization in source code. Linker will dump the following sections (.bss, .sbss)

5. HEAP:
   The heap segment contains dynamically allocated memory, commonly begins at the end of the BSS segment and grows to larger addresses from there(upwards). It is managed by malloc, calloc, realloc, and free functions. no linkage needed dynamically adjusted.

6. UNALLOCATED:
   occupies dynamic HEAP expansion upwards, and dynamic STACK expansion downwards.

7. STACK:
   initialized to 0xFFFF within the boot flow. The stack segment contains the call stack, a LIFO structure, typically located in the higher parts of memory. A "stack pointer" a register that tracks the top/bottom of the stack according to the ABI, in RISC-V it hold the Bottom address and expands downwards. It is changed each time a value is pushed onto the stack. The set of values pushed for one function call is termed a "stack frame". A stack frame consists at minimum of a return address. Automatic variables are also allocated on the stack when no more argument registers are available.

References