# Assignment 4: Computer Architecture (ECGR 4181/5181)

**Part 1: CPU and RAM joint simulation**

Consider the 4-stage pipelined CPU architecture from the previous assignment with Fetch, Decode, Execute and Store stages shown in Figure 1 – this time PIPELINED. In order to support instruction fetches in the Fetch stage and memory access in the Execute stage, the CPU maintains 2 different memory ports. The Instruction Port is a read-only port managed by the Fetch stage of the pipeline and is used for fetching new instructions from memory. The Data Port is a read/write port used by the Writeback stage for loading and storing data in memory.



**Figure 1**

In order to avoid memory contention between instruction fetch and memory access modern processors take advantage of data locality with a series of microarchitectural tricks that we will cover later in class. For the purposes of this assignment we will assume that the memory is banked and multi-ported to as shown in Figure 2 in order to support concurrent instruction fetch and memory operations.



**Figure 2**

You will simulate a PIPELINED single core system that executes the code provided in vadd.c

CPU Design Specs:

1. The CPU will implement the RISCV instructions from Assignment 3.
2. The CPU architecture is an extension of the previous assignment, without the Instruction queue – we are reading directly from RAM.
3. The CPU has 32 Integer Registers (32 bits wide) in a Bank and 32 Floating Point Registers (32 bits wide) in a separate Bank.
4. The CPU has PC (program controller) circuits (not shown) to handle regular increments (PC = PC + 4) as well as conditional branch and jump.
5. 1 CPU cycle will be equal to 10 simulation ticks
6. Instruction Latencies for Execute Stage (***stalls*** may be required between instructions):

RV32I instructions = 1 CPU cycle (10 sim ticks)

RV32F instructions = 5 CPU cycles (50 sim ticks)

RAM Design Specs:

1. Each location in RAM is 1 byte wide, and RAM holds an array of these bytes.
2. The memory map of the RAM is show in Figure 2 (not to scale)
   1. The instructions from vadd.c will be loaded into addresses 0x0 – 0x093
   2. The addresses 0x200 – 0x2FF will allocated for the stack
   3. The address ranges 0x400 – 0x7FF (ARRAY\_A) and 0x800 – 0xBFF (ARRAY\_B) will be initialized as arrays of random FP32 values.
   4. All other addresses can be left uninitialized
3. RAM Read/Write Latency = 2 CPU cycles (20 sim ticks).

Deliverables:

1. Write the assembly code for the C code in vadd.c. You are required to use Stack Pointer. [5]
2. Your PIPELINED simulator must be able to run the following RISCV instructions: RV32I Base instruction set, and the required *f* (Float) instructions from the RV32F Standard Extension set (all without OS interactions). [20]
3. Implement the RAM component of your simulator with all its associated circuitry. Explain the different signals required between the CPU and RAM. (***NOPs*** may be required while waiting for data to *load* from RAM, ***stalls*** may be required while waiting for data to *store* in RAM) [15]
4. Simulate the system with the CPU and RAM using the instructions generated from vadd.c
   1. Calculate the achieved CPI of the simulation [10]
   2. Validate the operation of your simulator by independently running the same vector addition code outside of your simulation. [10]

**Part 2: Multiprocessor Contention and Arbitration**

Now that we have working CPU and RAM simulators, we can extend our system to explore multiprocessor systems and contention over shared resources.



**Figure 3**

The example in Figure 3 shows a system with 2 CPUs that must share access to a common resource (RAM). Since the RAM only has 1 port allocated to data that must be shared across the CPUs, we must introduce a bus that can route data and arbitrate disputes over shared resources. In this example the membus can receive memory requests from the instruction and data ports of both CPUs and correctly routing those requests to the corresponding memory banks. In the event of simultaneous access of the same memory bank, the membus will allow only one request to process and automatically prioritize and retry the other request on a later simulation tick. Requests to different memory ports do not cause contention and can be handled simultaneously.

Repeat the tasks from Part 1 with the inclusion of the second CPU. The first CPU will run the code in vadd.c while the second CPU will run the code in vsub.c. The memory layout for instructions, program stacks, and data are shown in Figure 3.

Deliverables:

1. Implement the system bus and the Bus Arbiter for multiplexed access to the bus. [15]
2. Integrate the RAM and the 2 CPUs in your simulator. Generate data to calculate the CPI for each processor independently at the end of the entire simulation. [25]