COL 216 Homework 1

1) Please take a look at the attached program matmul.c that initializes two matrices, multiplies them, and prints the sum of the elements.

Part I

2) You need to implement the C code in SimpleRisc assembly with the following modifications.

1. Make the parameter N (dimension of the matrix) variable. It will be supplied by the user. You can make the assumption that N is a power of 2, and the matrix is a square matrix. N can vary from 4 to 128 (powers of 2).
2. We will use the command line interpreter written by Ritesh Noothigattu for evaluating the assignment. (see http://www.cse.iitd.ac.in/~srsarangi/archbooksoft.html)
3. You need to make some changes to the C emulator:
   1. It should accept the parameter N via the command line and save it in r0.
   2. We will invoke the emulator as follows: a.out <name of assembly file> <value of N>
   3. Use the .print macro to print the sum of the elements in the default format (decimal)
   4. Read the instruction manual of the emulator.
4. For the purposes of debugging, you can use the GUI based emulator (available at the site). However, we will not use it for testing.
5. There are some restrictions on the size of the stack, and the immediates that can be used with ld/st instructions. Hence, do not save your large matrices on the stack. Save them somewhere else.
6. Submit a single assembly file. (**Verify** that it works with the command line emulator).

Part II

3) Now, try to speed up your assembly code as much as possible. Here are some optimizations that you can do. (There are many more also, find out ...)

1. Loop unrolling (Google for it, and find out what it is)
2. Constant folding and strength reduction (again consult Google)
3. Submit an assembly file with these optimizations. It should lead to a faster implementation.
4. Try with even larger values of N.
5. HINT: Increase temporal and spatial locality (ASK me in class about these concepts)

Final Submission:

1) Two assembly files + 1 emulator file (written in C)
2) Deadline: Feb 28th, 11:59 PM
3) Ensure that all of you have accounts on CSE Moodle.

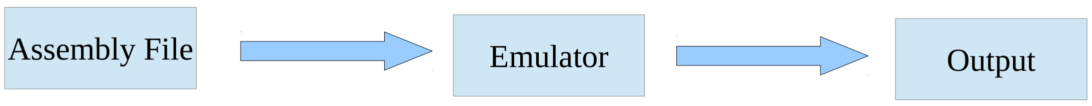
Evaluation Criteria

Co-ordinating TA: Ishani Mahajan

1. Part I has 6 marks. We will have 6 test cases (1 mark per test case).
2. Part II has 4 marks. We will have 4 test cases (0.5 marks per test case).
3. 2 marks are for the speed of the optimized code (Part II) (the thresholds will be decided after we evaluate the assignments. Ask me for the details of the thresholds).

Working with Memory

Your task is to write a SimpleRisc assembly file. This assembly file will be run by a SimpleRisc *emulator*. The job of the emulator is to run the file and produce a valid output.



The emulator is written in a high level language (in this case C). Its job is to read the assembly file (line by line), interpret the assembly statements, and execute them (like a native machine). The emulator essentially simulates a native machine. We can think of it as a *virtual machine.* The emulator has two main components. The first component reads the assembly file, interprets the text, checks for syntax errors, and understands the contents of the instruction. The second component simulates the instruction. For example, if we have an instruction of the form: *add r1, r2, r3* then the emulator will read the values of the registers *r2* and *r3*, add them and save the result in register *r1*.

In particular, the emulator needs to simulate the processor with a program counter and register file, and the memory. Simulating the processor and the register file is easy. However, simulating the memory is difficult. The reason for this is that we have a 32 bit memory system. It can theoretically save 4 billion bytes. We cannot have an array of a billion integers to save the entire memory. This is too big and too slow. Fortunately, programs use a very small portion of the memory space. They just need space to save the stack, read-only constants, and global variables. Each category of data (stack data or global data) is assigned to a distinct region in memory. Each such *distinct region* is called a *segment*. It is the job of the compiler and the assembler to assign the location of the segments in memory and manage them. For example, the compiler can say that the stack segment will start from the memory address 0xFF00 and extend till 0x0000. It can save the global segment in the memory region (0xFF 00 00 ↔ 0x10 00 00). This is a software convention similar to the register usage convention and the hardware is oblivious to these details. In this case, we do not have the luxury of a compiler. Hence, we need to manage the memory ourselves.

We need two distinct segments. We need the stack to manage local variables and function calls. Additionally, we need a global segment (region of memory) to save the matrices (A, B and C). We need to assign each segment a starting address and a range in the assembly file. While you are writing the assembly code you need to be explicitly aware of the locations of each segment such that you can read and write the correct data. For example, if you are accessing the array element A[3][10], you need to know where that specific array element is stored in memory.

The issue is that by default the emulator supports only the stack, and it places a strict limit on its size (4096 bytes). This is not enough for our purpose. Hence, we need to **modify** the emulator's code. We need support for global data as well. One solution is as follows. We can add a directive in the assembly file to explicitly create segments. Recall that a directive is an assembly statement, which does not correspond to a machine instruction. We are treating it as a directive (order) to the emulator. The directive will look like this:

.alloc <start of segment> <size>

example: .alloc 0x0000 0xFFFF

This directive will tell the emulator that the assembly programmer wishes to use memory in the range (0x0000 ↔ 0xFFFF). The emulator (which is mimicking the hardware) does not care about the purpose that this memory region is being used for. This memory region can be used to save the stack, can be used to save global data, or can be used to save read-only constants.

After this directive, the assembly programmer can start using memory in this range. For example, if we want to use this region as the stack, we can initialize the stack pointer to 0xFFFC (multiple of 4 bytes). The stack can then grow in a downward fashion.

We can additionally allocate another memory region and use it for saving our global variables, which in this case are large matrices. In any case, the emulator is oblivious to the way that the memory region is being used.

What is the emulator supposed to do?

The emulator can record the set of regions that the assembly programmer intends to use. For each region of memory, the emulator needs to create a data structure that will save the memory values. Depending on the size of the region, it can choose an appropriate data structure. For example, if a memory region is small, it can use a regular array. If it is large it can use a combination of an array and a hash table. Designing an appropriate data structure is up to the student.

Whenever, the programmer accesses a memory address, the emulator needs to find the region that this address belongs to. If it belongs to the stack, then it needs to access the appropriate data structure, and implement the memory access (load or store). Let us consider an example. Let us assume that the assembly programmer wishes to use the region 0x0000 ↔ 0xFFFF as the stack. This has been conveyed to the emulator using an assembly directive. The emulator has then created an internal data structure to save the memory values in this range of memory addresses.

Subsequently, let us assume that the programmer issues a load to the address 0x1124. The emulator can immediately determine that this address belongs to the stack segment. It will then access the data structure corresponding to this memory region (the stack), and read 4 bytes starting from the address 0x1124. These 4 bytes will then be used to populate the destination register of the load instruction.

Relevance to native software?

When we are running a program on a real machine, there is obviously no emulator present. However, the mechanism for allocating, and managing memory is very similar to what we have just described. This mechanism is called ``paging'' and requires both hardware and operating system support. We will discuss paging in detail after minor 2. What we are doing in this assignment is implementing a very primitive version of paging. Nevertheless, the basic idea is exactly the same.