CPPU

We have come up with a special-purpose architecture that is geared completely towards optimizing competitive programming workloads. Competitive programming involves lots of similar kinds of tasks and requires fast execution of algorithms. Competitive programming problems are popular among programmers for being very difficult to solve within the given constraints.

Generally, competitive programming tasks involve the following workloads. Other types of tasks can also appear but they are few and far between

1. I/O: Almost every single CP problem requires the programmer to take a large input (approximately the size of 10^5 integers or the equivalent) and some require a similar sized output
2. Loops: The algorithms generally require multiple loops over the input data
3. Math operations: Lots of add, subtract, multiply, and divide operations. Another very common math operation in CP is the modulo operation (%)
4. Recursion: recursion is used in common algorithms like DFS, brute force algorithms, etc.
5. String operations: Many CP problems involve string I/O, manipulation, comparison, etc.
6. Bitwise operations: Bit problems are a popular type of CP problem, generally involving bitwise operations like AND, OR, XOR, etc., usually being run a large number of times. There are also some common subtasks that the programmer has to implement themselves like get the number of 1s in the binary representation of a number, set a bit in a number to 1 or 0, etc.

Accordingly, we have made optimizations. First, we will outline the structure of our architecture.

- We will be using a 64-bit instruction set
- Word size will be 64 bits
- Data Types
    - 64 bit integers
    - 32 bit floating points (IEEE 754 Single precision floating point format)
    - We have different registers for each
    - Chars, bool, etc. are stored as integers
- Registers:
    - Program counter (4 bytes)
    - Instruction register (8 bytes)
    - Stack base pointer and stack pointer (4 bytes each)
    - Return address register for function calls (4 bytes)
    - General purpose registers R0 to R7 (8 bytes each)
    - Floating point registers F0 to F7 (4 bytes each)

- Array pointer registers A0 and A1 (8 bytes)
- Status register (1 byte)

Input buffer ready	Output buffer ready	Zero flag	Signed Flag				

The optimizations we made for this particular architecture are given below

- I/O:
    - We will implement buffered I/O for performance
    - We will implement a DMA I/O system where devices (in this case, just the GUI) can read and write to input and output buffers present in memory
    - Once buffers are full, a status bit is set saying the buffer is full. The data inside the buffer is then processed. There will be two types of bits, one for the input and one for the output buffer
    - This DMA I/O can run on a separate thread in the simulator
    - The **DMA controller** is initialized to read or write data from/to the buffer in memory.
    - Data is transferred from the I/O device to the buffer (or vice versa), using DMA
    - Once the data transfer is complete, the buffer status register is set
    - The CPU then processes the data in the buffer, writing it to memory
    - Interrupts via input will be handled by the CPU
- Cache:
    - We will have two caches for different purposes
    - For strings, we will use a sliding window approach to load substrings into the cache because in cp, large strings are usually read and processed sequentially
    - General cache with LRU eviction strategy
    - Different instructions will use different caches. For example, SLOAD instruction will read from the string cache whereas ILOAD and FLOAD will read from the general purpose cache
- ALU will have an additional modulo operation for getting remainders
- Apart from the regular bitwise operations like AND, OR, XOR, etc., we will add a __builtin_popcount, set_bit/unset_bit, and get_bit operations which are further described in the instruction set below.

Instruction set:

I/O:

IN

Opcode (6 bits)	Operand 1 (32 bits)	Operand 2 (22 bits)
00000	Memory address	Length (number of bytes to

		read)

- Will take input from the simulator GUI via a buffer as mentioned above
- Once buffer is written, status register bit 1 will be set to indicate that the data can be written
- This is done via the DMA controller

OUT

Opcode (6 bits)	Operand 1 (32 bits)	Operand 2 (22 bits)
000001	Memory address	Length (num bytes to write)

- Will write to buffer
- Once buffer is written, wil set status register bit 2 to indicate that data can be read
- DMA will read the data and write it to the output device (simulator GUI)

Load/Store:

ILOAD

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (32 bits)	Operand 3 (8 bits)
000010	Register	Memory address	Length

- Loads integers using the general purpose cache

FLOAD

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (32 bits)	Operand 3 (8 bits)
000011	Register	Memory address	Length

- Loads floats using the general purpose cache

SLOAD

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (32 bits)	Operand 3 (8 bits)

000100	Register	Memory address	Length

- Loads strings using the special purpose string cache

ISTORE

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (32 bits)	Operand 3 (8 bits)
000101	Register	Memory address	Length

- Stores integers using the general purpose cache

FSTORE

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (32 bits)	Operand 3 (8 bits)
000110	Register	Memory address	Length

- Stores floats using the general purpose cache

SSTORE

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (32 bits)	Operand 3 (8 bits)
000111	Register	Memory address	Length

- Stores strings using the special purpose string cache

Control:

CALL

Opcode (6 bits)	Operand 1 (32 bits)	Unused (28 bits)
001000	Function address	N/A

- Save the return address to the return address register and then jump to function address

RET

Opcode (6 bits)	Unused (58 bits)
001001	N/A

- Returns to address in return address register

CMP

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (3 bits)	Unused (52 bits)
001010	Register 1	Register 2	N/A

- Compares 2 registers and sets Zero and Signed flags in the status register

JMP

Opcode (6 bits)	Operand 1 (32 bits)	Unused (22 bits)
001011	Target address	N/A

JE (jump if equal)

Opcode (6 bits)	Operand 1 (32 bits)	Unused (22 bits)
001100	Target address	N/A

JNE (jump if not equal)

Opcode (6 bits)	Operand 1 (32 bits)	Unused (22 bits)
001101	Target address	N/A

JG (jump if greater)

Opcode (6 bits)	Operand 1 (32 bits)	Unused (22 bits)
001110	Target address	N/A

JL (jump if less than)

Opcode (6 bits)	Operand 1 (32 bits)	Unused (22 bits)

001111	Target address	N/A

JZ (jump if zero)

Opcode (6 bits)	Operand 1 (32 bits)	Unused (22 bits)
010000	Target address	N/A

- All of the above jump instructions use the Zero flag and Signed flags

Arithmetic:

ADD/FADD, SUB/FSUB, MUL/FMUL, DIV/FDIV

Opcodes (6 bits)	Operand 1 (3 bits)	Operand 2 (3 bits)	Unused (52 bits)
010001-011000	Register 1	Register 2	N/A

- All the above arithmetic operations store the final result in R1
- Same format for ints and float registers
- Type checking will be done on the registers by the assembler to make sure the correct type of registers are being used

MOD

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (3 bits)	Unused (52 bits)
011001	Register 1	Register 2	N/A

- Extra operation supported by our hypothetical hardware
- Useful in many competitive programming problems
- As with the other arithmetic operations, will be stored in R1

LSHIFT/RSHIFT

Opcodes (6 bits)	Operand 1 (3 bits)	Operand 2 (8 bits)	Unused (47 bits)
011010-011011	Register	Number of shifts	N/A

- Bitwise shift operator

- Number of shifts can be at most 64 bits (integer size) so only 8 bits are necessary for the operand

SETBIT/UNSETBIT

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (8 bits)	Unused (47 bits)
011100-011101	Register	Bit to set/unset	N/A

- Sets/unsets a specified bit in an integer register

GETBIT

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (8 bits)	Operand 3 (3 bits)	Unused (44 bits)
011110	Register 1	Bit to set/unset	Register 2	N/A

- Gets a specified bit in an integer register
- Stores value in register 2

POPCOUNT

Opcode (6 bits)	Operand 1 (3 bits)	Operand 2 (3 bits)	Unused (52 bits)
011111	Register 1	Register 2	N/A

- Similar to __builtin_popcount() in c++
- Gets the number of set bits in the binary representation of an integer
- Only works on integer registers
- Stores value in Register 2

The GUI will mostly be a simple view into the different registers and what they contain. We will also have a section to view the different caches. The GUI will also support input and output for ASCII characters. There will be support to step through the program one step at a time. It will show the the pipeline state.

We will use C++ for our project with the QT framework. We will use Github for versioning and team collaboration. We will communicate via text and will meet regularly during the week to update each other on the progress and get help if necessary.

Initially, everyone will be responsible for the detailed design of the API, code structure, etc. Once that is finalized, Pranav will work on the implementation of the memory layout. Prathik will begin to build the pipeline and Sagnik will work on the UI. We will switch to Pranav working on the UI, and while Prathik finishes up the pipeline, Sagnik will start working on the optimizations like I/O. Once the pipeline implementation is finished, Prathik will take up any remaining work on the UI. Pranav and Sagnik will work on the final and more complicated optimizations like the caches. Finally, we all will be responsible for writing a benchmark for the baseline as well as each optimization. We will also all be responsible for the report