The CPU Project
Part 3: Implementing Instruction Fetch and Decode

Due Friday, May 10 at 11:55pm

The third (and last) part of the CPU project is to implement instruction fetch (loading the instruction from memory) and decode (figuring out what the instruction does based on the opcode) in the simulated MIPS processor.

You will be putting the code for this part of the project in cpu.c. Don't forget to "#include" the necessary libraries (e.g. stdio.h, stdint.h) as well as the header files containing the definitions that your code in cpu.c will need (cpu.h, r_instructions.h, etc.).

Here are the steps you should take:

Step 1

In cpu.c, you should define the global variables that are declared as `extern` in cpu.h. These are:

- `registers`. This is an array of size `NUM_REGISTERS`, where the element type is `uint64_t` (and `NUM_REGISTERS` is defined in cpu.h).
- `pc` and `npc`. Each of these is of type `uint64_t`.
- `memory`. This is an array of bytes (i.e the element type is `uint8_t`). The size of memory is determined by `MEMORY_SIZE_IN_BYTES`, which is defined in cpu.h.

Step 2

First, carefully read the discussion in the CPU project description, cpu_project_description.pdf, provided previously about using switch statements or dispatch tables to call the functions representing R-instructions, I-instructions, and J-instructions based on the opcode field and (for R-instructions) the funct field of an instruction. Then, in cpu.c, define the procedure `cpu_initialize()`, which is declared in cpu.h by

```
void cpu_initialize();
```

`cpu_initialize()` should do the following:

- Initialize both `pc` and `npc` to `CODE_STARTING_ADDRESS`.
- Initialize register 29 (the stack pointer) to `STACK_STARTING_ADDRESS`, which is defined in cpu.h.
- Set register 0 to zero (since register 0 should always contain zero).
- If you are using dispatch tables, rather than a switch statement, to call the functions representing instructions, then populate the dispatch table as described in the project description.

Step 3

In cpu.c, define the procedure `cpu_execute()`, which is declared in cpu.h by

```
void  cpu_execute();
```

`cpu_execute()` is responsible for repeatedly executing the instructions in a MIPS machine code program. `cpu_execute()` should have a loop that – until `cpu_exit()`, below, causes execution to terminate – repeatedly performs the following actions:

- set `pc` = `npc`
- increment `npc` to point to the next instruction (i.e. increment `npc` by 8)
- fetch the instruction from memory that `pc` points to. Remember to fetch a 64-bit (8-byte) instruction, even though memory is an array of bytes.
- extract the opcode from the instruction (you'll want to #define a macro for this).
- use the opcode to index into the instruction dispatch table, or as a case in a switch statement, to call the appropriate procedure for simulating the instruction.

Step 4

In cpu.c, define the procedure `cpu_exit()`, which is declared in cpu.h by

```
void cpu_exit(int errorcode);
```

If `errorcode` is 0, then `cpu_exit()` should print out a message that the program terminated normally. Otherwise, if `errorcode` is not 0, `cpu_exit()` should print out a message indicating that the program terminated with an error (and print out the value of `errorcode` as

well). Then, `cpu_exit` should cause the CPU execution (i.e. the loop in `cpu_execute`) to terminate.

Step 5

Upload your cpu.c file to Brightspace. Congratulations, you have finished the CPU project!