CPU Project Description

This project involves building a high-level simulator for a simple 32-bit RISC CPU in C. The instruction set architecture (ISA, defining the registers, instructions, etc.) of this CPU is similar to an early version of the MIPS processor that we are discussing in class, except that only integer registers and instructions will be simulated and various other details (floating point registers and instructions, overflow, etc.) will be left out.

The MIPS Registers

There are 32 integer registers, numbered from 0 to 31. Register 0 always contains the value 0 and cannot be modified. Other of these registers are used for special purposes (stack pointer, base pointer, etc.), but that is by convention and does not affect the implementation of your CPU. In this document, we will refer to the i^{th} register as R[i].

There are also two other registers, HI and LO, that are used for multiplication and division, as explained below.

Finally, there is a program counter register, PC, which is not explicitly referenced in any instruction (we have referred to the program counter in class as an "instruction pointer" register).

The MIPS Instructions

All instructions are exactly 32 bits wide.

There are three categories of instructions, R-instructions, I-instructions, and J-instructions. Within each category, every instruction has the same format, as described below.

R-Instructions

The R-instructions are used to perform operations where the operands are registers.

R-instructions have the following format, from left to right (most significant bit to least significant bit):

- opcode field: 6 bits (NOTE: for all R-instructions, opcode = 0)
- RS field: 5 bits (specifies one of the registers)
- RT field: 5 bits (specifies one of the registers)
- RD field: 5 bits (specifies one of the registers)
- shamt field: 5 bits (specifies a shift amount in a shift instruction)
- funct field: 6 bits (specifies which R-instruction to execute)

The list of R-instructions that you will need to implement, and their descriptions, is as follows.

(Remember, the opcode for R-instructions is always 0)

Instr	**Description**	**Funct**	**Operation**	**Note**
add	signed addition	0x20	R[rd] ← R[rs] + R[rt]	
addu	unsigned addition	0x21	R[rd] ← R[rs] + R[rt]	
sub	signed subtraction	0x22	R[rd] ← R[rs] - R[rt]	
subu	unsigned subtraction	0x23	R[rd] ← R[rs] - R[rt]	
mult	signed multiplication	0x18	HI:LO ← R[rs]*R[rt]	The top 32 bits of the 64-bit result of the multiplication are put into HI, the bottom 32 bits are put into LO. See discussion below about performing signed 64-bit multiplication in C.
multu	unsigned multiplication	0x19	HI:LO ← R[rs]*R[rt]	The top 32 bits of the 64-bit result of the unsigned multiplication are put into HI, the bottom 32 bits are put into LO. See discussion below about performing unsigned 64-bit multiplication in C.

div	signed division	0x1a	LO ← R[rs] / R[rt]
			HI ← R[rs] % R[rt]	

divu	unsigned division	0x1b	LO ← R[rs] / R[rt];
			HI ← R[rs] % R[rt]	

mfhi	move from HI	0x10	R[rd] ← HI	
mflo	move from LO	0x12	R[rd] ← LO	
and	bitwise AND	0x24	R[rd] ← R[rs] & R[rt]	
or	bitwise OR	0x25	R[rd] ← R[rs] | R[rt]	
xor	bitwise XOR	0x26	R[rd] ← R[rs] ^ R[rt]	
nor	bitwise NOR	0x27	R[rd] ← ~(R[rs] | R[rt])	

slt	set less than (signed)	0x2a	If (R[rs] < R[rt])
			R[rd] ← 1
			else
			R[rd] ← 0
				The comparison treats R[rs] and R[rt] as signed numbers.

sltu	set less than (unsigned)	0x2b	If (R[rs] < R[rt])
			R[rd] ← 1
			else
			R[rd] ← 0
				The comparison treats R[rs] and R[rt] as unsigned numbers.

sll	shift left logical	0x00	R[rd] ← R[rt] << shamt	
sllv	shift left logical variable	0x04	R[rd] ← R[rs] << R[rt]	
srl	shift right logical	0x02	R[rd] ← R[rt] >> shamt	
srlv	shift right logical variable	0x06	R[rd] ← R[rs] >> R[rt]	
sra	shift right arithmetic	0x03	R[rd] ← R[rt] >> shamt	See discussion below about arithmetic shift.
srav	shift right arithmetic variable	0x07	R[rd] ← R[rs] >> R[rt]	See discussion below about arithmetic shift.

jr	jump register	0x08	npc ← R[rs]	The target of the jump is specified in R[rs]. See discussion below concerning npc and pc.

jalr	jump and link register	0x09	r[31] ← pc + 4
			npc ← r[rs]
				For procedure calls. The return address (pc+4) is stored in R[31]. The target of the jump is specified in R[rs]. The target of the jump is specified in R[rs].

syscall	System call for OS services	0x0c		See discussion below concerning syscall.

I-instructions

I-instructions are used to perform operations using an immediate value (i.e. a constant). These include some arithmetic instructions, load and store instructions, and conditional branch instructions.

I-instructions have the following format, from left to right (most significant bit to least significant bit):

- opcode field: 6 bits (specifies which instruction to execute)
- RS field: 5 bits (specifies one of the registers)
- RT field: 5 bits (specifies one of the registers)
- immediate field: 16 bits (specifies a constant value to be used)

The list of I-instructions that you will need to implement, and their descriptions, is as follows.

Instr	Description	opcode	Operation	Note
addi	add immediate	0x08	R[rt] ←R[rs] + sign_extend(imm)	See discussion on sign extension below
addiu	add immediate unsigned	0x09	R[rt] ← R[rs] + sign_extend(imm)	
lw	load word	0x23	R[rt]← M[R[rs]+ sign_extend(imm)]	
lh	load half-word	0x21	R[rt] ← sign_extend(M[R[rs]+sign_extend(Imm)])	Two bytes are loaded from memory into the lower half of R[rt]. The upper half of R[rt] results from sign-extending the loaded value.

Instr	Description	opcode	Operation	Note
lhu	load half-word unsigned	0x25	R[rt] ← M[R[rs] + sign_extend(Imm)]	Two bytes are loaded from memory into the lower half of R[rt]. The upper half of R[rt] contains only zeros.
lb	load byte	0x20	R[rt] ← sign_extend(M[R[rs]+sign_extend(Imm)])	One byte is loaded from memory into the lowest byte of R[rt]. The rest of R[rt] results from sign-extending the loaded value.
lbu	load byte unsigned	0x24	R[rt] ← M[R[rs] + sign_extend(Imm)]	One byte is loaded from memory into the lowest byte of R[rt]. The rest of R[rt] contains only zeros.
sw	store word	0x2b	M[R[rs] + sign_extend(Imm)] ← R[rt]	
sh	store half-word	0x29	M[R[rs] + sign_extend(Imm)] ← R[rt]	The lower half of R[rt] is stored in memory.
sb	store byte	0x28	M[R[rs] + sign_extend(Imm)] ← R[rt]	The lowest byte of R[rt] is stored in memory.
lui	load upper immediate	0x0f	R[rt] ← (Imm << 16)	Imm is loaded into the upper half of R[rt]. The lower half of R[t] contains only zeros.
ori	bitwise or immediate	0x0d	R[rt]←R[rs] | Imm	Imm is zero extended (i.e. the upper 16 bits are zero).
andi	bitwise and immediate	0x0c	R[rt]←R[rs] & Imm	Imm is zero extended.
xori	bitwise xor immediate	0x0e	R[rt]←R[rs] ^ Imm	Imm is zero extended.

slti	set less than immdiate	0x0a	if (R[rs] < sign_extend(imm))
			R[rt] ← 1 else R[rt] ← 0	

sltiu	set less than immdiate unsigned	0x0b	if (R[rs] < sign_extend(imm))
			R[rt] ← 1 else R[rt] ← 0
				The comparison is unsigned, so both operands are interpreted as non-negative.

beq	branch on equal	0x04	if(R[rs] == R[rt])
			npc ← pc + 4 + (sign_extend(Immediate) << 2)	

Instr	Description	opcode	Operation	Note

bne	branch on not equal	0x05	if(R[rs] != R[rt])
			npc ← pc + 4 + (sign_extend(Immediate) << 2)	

J-instructions

J-instructions are jump instructions.

J-instructions have the following format.

- opcode field: 6 bits (specifies which instruction to execute)
- address field: 26 bits (used in specifying the target address of the jump)

The list of J-instructions that you will need to implement, and their descriptions, is as follows:

Instr	Description	Opcode	Operation	Note
j	jump	0x02	npc ← ((pc+4) & 0xf0000000) | (address << 2)	The target of jump comes from the top 4 bits of PC+4 followed by the 28 bits resulting from shifting address left by 2.

jal	jump and link	0x03	R[31] ← pc+4
			npc ← ((pc+4) & 0xf0000000)
			| (address << 2)
				For procedure calls. The return address (pc+4) is stored in R[31]. The target of jump comes from the top 4 bits of PC+4 followed by the 28 bits resulting from shifting address left by 2.

Implementing the MIPS Simulator

I have provided you with a number of files, as follows:

- cpu.h: Declares the global variables representing the registers and memory, defines related constants, and declares the procedures needed to initialize the CPU, execute the CPU, and stop the CPU.
- cpu.o: The compiled version of cpu.c, which defines the variables and procedures declared in cpu.h as well as implementing the core CPU operation, as described below.
- r_instructions.h: Declares the procedures that implement the R-instructions (other than syscall), one procedure per instruction.
- r_instructions.o: The compiled version of r_instructions.c, which defines the procedures declared in r_instructions.h.
- i_instructions.h: Declares the procedures that implement the I-instructions, one procedure per instruction.
- i_instructions.o: The compiled version of i_instructions.c, which defines the procedures declared in i_instructions.h.

- j_instructions.h: Declares the procedures that implement the J-instructions, one procedure per instruction.

- j_instructions.o: The compiled version of j_instructions.c, which defines the procedures declared in j_instructions.h.

- syscall.c: This defines the procedure representing the syscall instruction.

- Makefile: Allows for the compilation of the .c files in the project, as necessary, by simply typing “make” in the shell. It generates an executable file name “cpu” (or cpu.exe on Windows), which you can run by typing “./cpu”.

- test_programs.c: Contains the main() procedure as well as procedures for testing the CPU on MIPS machine code programs. You should probably not modify this code unless you really understand what’s going on inside it. The `main()` procedure causes instructions to be loaded into memory starting at address `CODE_STARTING_ADDRESS` (see cpu.h).

- utilities.h, utilities.c: Contains procedures for printing out the bits of a 32-bit or 64-bit integer. You may find this useful for debugging.

- register_names.h, programming.h: Code I wrote to aid in testing the CPU (used in test_programs.c). You can take a look at these files, but you won’t need to use them.

- ben_cpu: This is a working version of the MIPS simulator that you can compare the output of your version to. To run it, simply type “./ben_cpu”.

The programming project comprises three parts:

Part 1: Implement your own version of r_instructions.c.

Part 2: Implement your own versions of i_instructions.c and j_instructions.c

Part 3: Implement your own version of cpu.c.

The simulated integer registers

As you can see in cpu.h, the integer registers are declared as follows:

```
unsigned int registers[];
unsigned int HI;
unsigned int LO;
```

The procedures that you write to implement the various MIPS instructions should use these registers as appropriate. Thus, even though this document refers to the ith register as R[i], your code would refer to it as `registers[i]`.

The simulated memory

Memory is declared as an array of bytes as follows:

```
unsigned char memory[];
```

Any instruction that you implement that accesses memory (e.g. lw, lh, lb, sw, sh, sb, etc.) will need to access this array. Be sure that your code is loading or storing the right number of bytes. For example, lw should load a 32-bit quantity from memory into a register. Referencing `memory[...]` in your code will just load a single byte, so to load a 4-byte quantity from memory, use a pointer of type `(unsigned int *)` to point to the area of `memory` that you want to load from.

The program counter (pc and npc)

For convenience in the code, there are two variables that implement the program counter.

```
unsigned int pc;
unsigned int npc;
```

The variable `pc` always points to the current instruction being executed and `npc` is assigned to the *next* instruction to be executed. That is, the main loop of the CPU (as implemented by `cpu_execute()` in cpu.c) is:

```
loop while not done:
    pc ← npc
    npc ← pc + 4
    execute instruction at memory[pc]
end loop
```

This way, if the executed instruction does not modify `npc`, the next instruction in memory will be executed in the next iteration of the loop. If the instruction being executed is a jump or branch instruction, it should set `npc` to the address to jump to. **IMPORTANT**: instructions should not modify `pc`, they should only read the value of `pc`. Conversely, `npc` may be written to (but only by jump or branch instructions) and should not be read. No other instructions should write to `npc`.

Sign Extension

There are a number of instructions (e.g. most I-instructions) that require taking a 16-bit number and extending it to 32 bits. If the 16-bit number is non-negative, i.e. if bit 15 (the leftmost bit) is zero, then the leftmost 16 bits of the extended (32-bit) version of the number should be all zeros. But, if the 16-bit number is negative, i.e. if bit 15 is one, then the leftmost 16 bits of the extended version of the number should be all ones. For example,

16-bit value		sign-extended 32-bit value	
0000 0000 0000 1101	⇒	0000 0000 0000 0000 0000 0000 0000 1101	(this is 13 in decimal)
1111 1111 1111 1101	⇒	1111 1111 1111 1111 1111 1111 1111 1101	(this is -3 in decimal)

To implement this sign extension, perform the usual masking to check the value of the leftmost bit of the 16-bit value and, if that bit is not 0, then use a mask and the bitwise-OR operation to write all 1's in the upper16 bits of the result.

The same holds for extending an 8-bit number to a 32 bit number.

NOTE: You could implement the same sign-extension by relying on the C compiler, but I'd like you to use masking as described above (so you will know about sign extension if I ask it on the final exam).

Arithmetic Right Shift

In order to use right shift as division by 2 or greater powers of two, it is necessary to preserve the sign of the number being shifted. To do so, the leftmost bit of the number before shifting should be repeated in the bits that opened up as a result of the shift. For example,

Shifting right by 3

Before shift		After shift
0011 0001 0010 ...	⇒	0000 0110 0010 010 ...
1011 0001 0010 ...	⇒	1111 0110 0010 010 ...

Here are some hints for implementing an arithmetic shift to the right by *n* bits:

- Perform the usual masking to test if the leftmost bit (bit 31) of the original number is 0 or not.
- If the leftmost bit of the original number is not 0, then after you shift the number to the right by *n*, you'll need to use a mask write *n* ones into the leftmost *n* bits of the shifted number.
- To create a mask with *n* ones in the rightmost position, use `((1 << n) - 1)`. This, of course, is $2^n - 1$.
- That mask will need to be shifted to the left by (32 – *n*) to put the ones in the leftmost position.

NOTE: You could implement the same sign-extension by relying on the C compiler, but I'd like you to use masking as described above (so you will know about arithmetic shift right if I ask it on the final exam).

64-bit Multiplication in C

As described above, the MULT and MULTU instructions multiply two 32-bit numbers to produce a 64-bit result. In order to get this to work correctly in C, it's important to cast the 32-bit numbers to 64-bit numbers before performing the multiplication. For example, if you wanted to multiply two unsigned 32-bit variables A and B to produce an unsigned 64-bit result, you might write something like:

```
unsigned long res = (unsigned long) A * (unsigned long) B;
                                           //unsigned multiplication
```

If you want to produce a signed 64-bit result from the same two unsigned operands, you could write:

```
long res = (long) A * (long B);  //signed multiplication
```

Syscall

The syscall ("system call") instruction is an R-instruction, with no operands, that traps to the operating system so that the OS can perform a requested operation. You'll learn much more about system calls in your Operating Systems course. If you look in the r_instructions.h file, you'll see that the syscall procedure is declared by

```
void syscall(unsigned int instruction);
```

I have already defined the syscall() procedure for you in syscall.c, but looking at this code may be helpful to you. All your r_instruction code has to do is call syscall() when the syscall instruction is encountered.

The code I wrote for syscall() in syscall.c operates as follows. It exams the value stored in R[2] (which is the register whose nickname is $V0). Depending on the value of R[2], syscall() perform the following actions:

R[2] == 1: ("print_int") prints the integer value stored in R[4] (also known as $A0)
R[2] == 2: ("print_float") not supported, prints an error message and call cpu_exit(1)
R[2] == 3: ("print_double") not supported, prints an error message and call cpu_exit(1)
R[2] == 4: ("print_string") prints the string that starts at Mem[R[4]].
R[2] == 5: ("read_int") read in an integer and store it in R[2] (also known as $V0)
R[2] == 6: ("read_float") not supported, prints an error message and call cpu_exit(1)
R[2] == 7: ("read_double") not supported, prints an error message and call cpu_exit(1)
R[2] == 8: ("read_string") read a string into memory starting at Mem[R[4]]. R[5] contains the size of the string buffer, so make sure the string is no longer than R[5] bytes.
R[2] == 9: ("sbrk") not supported, prints an error message and call cpu_exit(1)
R[2] == 10: ("exit") call cpu_exit(0)
Otherwise: unsupported operation, prints an error message and call cpu_exit(1)

Again, you do not have to implement anything in syscall.c.

Email me if you have any questions. Good luck!

****** IMPORTANT: See the next page for hints! ******

Hints

1. This assignment is not too hard, so don't stress out. It is long, though, so get started right away.

2. Use this document, not online materials, for understanding how the MIPS instructions should work.

3. The first file you need to implement is r_instructions.c. Logically, there are two parts to implementing the R-instructions:

    - Write macros to extract the various fields of an R-instruction, i.e. the opcode, RS, RT, RD, shamt, and funct fields. These macros should use shifts and masks, just like the macros you wrote to extract the sign, exponent, and fraction fields of floating point numbers in assignment 2.

    - Implement each R-instruction (add, addu, sub, etc.) as a separate, tiny function. The complete list of the functions you need to implement is found in r_instructions.h. To get you started, the `add()` function, which takes the 32-bit instruction as a parameter, should perform the following operations:

        - Extract the RS, RT, and RD fields from the instruction (the other fields aren't needed).

        - Compute the sum of `registers[RS]` and `registers[RT]` and write the result into `registers[RD]`.

    Since `registers` is an array of unsigned int, but the add instruction is signed addition, you'll need to cast `registers[RS]` and `registers[RT]` to int, perform the addition, and then cast the result back to unsigned int before writing it to `registers[RD]`. This function only needs to be a few (between one and four) lines.

4. The process for implementing i_instructions.c and j_instructions.c is much the same as for r_instructions.c, namely defining the macros (e.g. using masks) you need and then defining the very simple functions listed in i_instructions.h and j_instructions.h. Be sure to pay close attention to how each I-instruction and J-instruction is described in this document. Remember that each J-instruction (which is a jump) should write the destination address to `npc`.

5. Within cpu.c:

    - Declare the `registers` array, the registers `HI`, `LO`, `pc`, and `npc`, and the `memory` array. The types of these variables will be exactly as declared in cpu.h, but without the `extern` specifier.

    - Define a macro for extracting the opcode field of an instruction.

    - Define the functions `cpu_initialize()`, `cpu_execute()`, and `cpu_exit()`, as specified in cpu.h.

    - To implement `cpu_execute()`, you should think about how to call the appropriate function in r_instructions.c, i_instructions.c, or j_instructions.c, based on the opcode of an instruction. For example, if the opcode is 0x08, the addi() function (see r_instructions.h) should be called.

Furthermore, if the opcode is 0, a function in r_instructions.c should be called based on the funct field. For example if the opcode is 0 and the funct field is 0x18, then the mult() function (see r_instructions.h) should be called.

There are two ways that you might implement the calling of a C function (e.g. add(), lw(), jal()) based on the value of the opcode and, in the case of R-instructions, the funct field of an instruction:

- Use a switch statement for the opcode field, e.g.,

  ```
  #define ADDI_OPCODE (0x08)
  ...
  switch (opcode) {
      case 0:  handle_r_instructions(instruction);
                              break;
      case ADDI_OPCODE: addi(instruction);
                        break;
      ...
  }
  ```

  where handle_r_intructions() has a similar switch statement for calling the R-instruction function (add(), sub(), etc.) based on the funct field, or

- Use a dispatch table, which is an array of function pointers (e.g., click here). For example, you could declare the table opcode_dispatch_table as follows:

  ```
  typedef void (*OP_DISPATCH)(unsigned int);

  OP_DISPATCH opcode_dispatch_table[MAX_NUMBER_OF_OPCODES];
  ```

  You would need need to populate the dispatch table with the functions for each instruction, indexed by the opcode:

  ```
  #define ADDI_OPCODE (0x08)
  ...
  opcode_dispatch_table[0] = &dispatch_r_instruction;
  opcode_dispatch_table[ADDI_OPCODE] = &addi;
  ...
  ```

  where dispatch_r_instruction would similarly call an R-instruction function (e.g. add()) based on the funct field.

  When an instruction is to be executed,your cpu_execute() function would simply use the opcode as an index into the dispatch table, and call the function found at that index in the table.

6. As discussed above, a program to be executed is loaded into memory by the main() function already defined in test_programs.c. If your cpu executable is getting a different output than the ben_cpu executable, you can look at the instructions that are being executed in test_programs.c to figure out which instruction your code isn't executing correctly.