**CPU Project Description**

This project involves building a high-level simulator for the simple 32-bit MIPS CPU discussed in class. The instruction set architecture (ISA, defining the registers, instructions, etc.) of this CPU is similar to an early version of the MIPS processor, 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 32. Register 0 always contains the value 0 and cannot be modified. Other of these registers are used for special purposes (stack pointer, frame 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\textsuperscript{th} register as R[i].

The 32 integer registers also have nicknames that assembly programmers can use, as shown at [http://en.wikipedia.org/wiki/MIPS_architecture#Compiler_register_usage](http://en.wikipedia.org/wiki/MIPS_architecture#Compiler_register_usage). This is not a concern of the CPU hardware itself.

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.

**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)

<table>
<thead>
<tr>
<th>Instr</th>
<th>Description</th>
<th>Funct</th>
<th>Operation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>signed addition</td>
<td>0x20</td>
<td>R[rd] ← R[rs] + R[rt]</td>
<td></td>
</tr>
<tr>
<td>addu</td>
<td>unsigned addition</td>
<td>0x21</td>
<td>R[rd] ← R[rs] + R[rt]</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>signed subtraction</td>
<td>0x22</td>
<td>R[rd] ← R[rs] - R[rt]</td>
<td></td>
</tr>
<tr>
<td>subu</td>
<td>unsigned subtraction</td>
<td>0x23</td>
<td>R[rd] ← R[rs] - R[rt]</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>signed multiplication</td>
<td>0x18</td>
<td>HI:LO ← R[rs]*R[rt]</td>
<td>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.</td>
</tr>
<tr>
<td>multu</td>
<td>unsigned multiplication</td>
<td>0x19</td>
<td>HI:LO ← R[rs]*R[rt]</td>
<td>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.</td>
</tr>
<tr>
<td>div</td>
<td>signed division</td>
<td>0x1a</td>
<td>LO ← R[rs] / R[rt]; HI ← R[rs] % R[rt]</td>
<td></td>
</tr>
<tr>
<td>divu</td>
<td>unsigned division</td>
<td>0x1b</td>
<td>LO ← R[rs] / R[rt]; HI ← R[rs] % R[rt]</td>
<td></td>
</tr>
<tr>
<td>mfhi</td>
<td>move from HI</td>
<td>0x10</td>
<td>R[rd] ← HI</td>
<td></td>
</tr>
<tr>
<td>mflo</td>
<td>move from LO</td>
<td>0x12</td>
<td>R[rd] ← LO</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>bitwise AND</td>
<td>0x24</td>
<td>R[rd] ← R[rs] &amp; R[rt]</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>bitwise OR</td>
<td>0x25</td>
<td>R[rd] ← R[rs]</td>
<td>R[rt]</td>
</tr>
<tr>
<td>xor</td>
<td>bitwise XOR</td>
<td>0x26</td>
<td>R[rd] ← R[rs] ^ R[rt]</td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td>bitwise NOR</td>
<td>0x27</td>
<td>R[rd] ← ~R[rs]</td>
<td>R[rt])</td>
</tr>
</tbody>
</table>
| 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                |                                                                      |
| sltv    | 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. |
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.

<table>
<thead>
<tr>
<th>Instr</th>
<th>Description</th>
<th>opcode</th>
<th>Operation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>add immediate</td>
<td>0x08</td>
<td>R[rt] ← R[rs] + sign_extend(imm)</td>
<td>See discussion on sign extension below</td>
</tr>
<tr>
<td>addiu</td>
<td>add immediate unsigned</td>
<td>0x09</td>
<td>R[rt] ← R[rs] + sign_extend(imm)</td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>load word</td>
<td>0x23</td>
<td>R[rt] ← M[R[rs] + sign_extend(imm)]</td>
<td>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.</td>
</tr>
<tr>
<td>lh</td>
<td>load half-word</td>
<td>0x21</td>
<td>R[rt] ← sign_extend(M[R[rs] + sign_extend(imm))]</td>
<td></td>
</tr>
<tr>
<td>lhu</td>
<td>load half-word unsigned</td>
<td>0x25</td>
<td>R[rt] ← M[R[rs] + sign_extend(imm)]</td>
<td>Two bytes are loaded from memory into the lower half of R[rt]. The upper half of R[rt] contains only zeros.</td>
</tr>
</tbody>
</table>

syscall

System call for OS services

0x0c

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].

See discussion below concerning syscall.
<table>
<thead>
<tr>
<th>Instr</th>
<th>Description</th>
<th>Opcode</th>
<th>Operation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb</td>
<td>load byte</td>
<td>0x20</td>
<td>R[rt] ← sign_extend(M[R[rs]+sign_extend(Imm))]</td>
<td>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.</td>
</tr>
<tr>
<td>lbu</td>
<td>load byte unsigned</td>
<td>0x24</td>
<td>R[rt] ← M[R[rs] + sign_extend(Imm)]</td>
<td>One byte is loaded from memory into the lowest byte of R[rt]. The rest of R[rt] contains only zeros.</td>
</tr>
<tr>
<td>sw</td>
<td>store word</td>
<td>0x2b</td>
<td>M[R[rs] + sign_extend(Imm)] ← R[rt]</td>
<td></td>
</tr>
<tr>
<td>sh</td>
<td>store half-word</td>
<td>0x29</td>
<td>M[R[rs] + sign_extend(Imm)] ← R[rt]</td>
<td>The lower half of R[rt] is stored in memory.</td>
</tr>
<tr>
<td>sb</td>
<td>store byte</td>
<td>0x28</td>
<td>M[R[rs] + sign_extend(Imm)] ← R[rt]</td>
<td>The lowest byte of R[rt] is stored in memory.</td>
</tr>
<tr>
<td>lui</td>
<td>load upper immediate</td>
<td>0x0f</td>
<td>R[rt] ← (Imm &lt;&lt; 16)</td>
<td>Imm is loaded into the upper half of R[rt]. The lower half of R[rt] contains only zeros.</td>
</tr>
<tr>
<td>ori</td>
<td>bitwise or immediate</td>
<td>0x0d</td>
<td>R[rt] ← R[rs]</td>
<td>Imm</td>
</tr>
<tr>
<td>andi</td>
<td>bitwise and immediate</td>
<td>0x0c</td>
<td>R[rt] ← R[rs] &amp; Imm</td>
<td>Imm is zero extended.</td>
</tr>
<tr>
<td>xori</td>
<td>bitwise xor immediate</td>
<td>0x0e</td>
<td>R[rt] ← R[rs] ^ Imm</td>
<td>Imm is zero extended.</td>
</tr>
<tr>
<td>slti</td>
<td>set less than immediate</td>
<td>0x0a</td>
<td>if (R[rs] &lt; sign_extend(imm)) R[rt] ← 1 else R[rt] ← 0</td>
<td>The comparison is unsigned, so both operands are interpreted as non-negative.</td>
</tr>
<tr>
<td>sltiu</td>
<td>set less than immediate unsigned</td>
<td>0x0b</td>
<td>if (R[rs] &lt; sign_extend(imm)) R[rt] ← 1 else R[rt] ← 0</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>branch on equal</td>
<td>0x04</td>
<td>if(R[rs] == R[rt]) npc ← pc + 4 + (sign_extend(Immediate) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>branch on not equal</td>
<td>0x05</td>
<td>if(R[rs] != R[rt]) npc ← pc + 4 + (sign_extend(Immediate) &lt;&lt; 2)</td>
<td></td>
</tr>
</tbody>
</table>
**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:

<table>
<thead>
<tr>
<th>Instr</th>
<th>Description</th>
<th>Opcode</th>
<th>Operation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>jump</td>
<td>0x02</td>
<td>npe ← ((pc+4) &amp; 0xf0000000)</td>
<td>(address &lt;&lt; 2)</td>
</tr>
<tr>
<td>jal</td>
<td>jump and link</td>
<td>0x03</td>
<td>R[31] ← pc+4 npe ← ((pc+4) &amp; 0xf0000000)</td>
<td>(address &lt;&lt; 2)</td>
</tr>
</tbody>
</table>

**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.o: The compiled version of syscall.c, which defines the procedure (declared in r_instructions.h) representing the syscall instruction.

• Makefile: Allows for the re-compilation of the .c files in the project, as necessary, by simply typing “make” in the shell.

• 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.

• 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.

The programming project comprises three parts:

Part 1: Implement your own versions of r_instructions.c and syscall.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 C stdint library

Because we want to be able to declare signed and unsigned integer values that are exactly 8 bits, 16 bits, 32 bits, and 64 bits wide, we use the C stdint library. At the top of each .c file that you write, you should put

#include <stdint.h>

The integer types defined in this library, and which you should use in your code, are:

unsigned: uint8_t, uint16_t, uint32_t, and uint64_t

signed: int8_t, int16_t, int32_t, and int64_t

Notice, for example, that the type of the registers declared in cpu.h is uint32_t.

The simulated integer registers

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

uint32_t registers[];
uint32_t HI;
uint32_t 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 i\textsuperscript{th} 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:

```c
uint8_t 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 (Hint: use a pointer of type `uint32_t *`) 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.

```c
uint32_t pc;
uint32_t 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,

<table>
<thead>
<tr>
<th>16-bit value</th>
<th>sign-extended 32-bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 1101</td>
<td>0000 0000 0000 0000 0000 0000 0000 1101 (this is 13 in decimal)</td>
</tr>
<tr>
<td>1111 1111 1111 1101</td>
<td>1111 1111 1111 1111 1111 1111 1111 1101 (this is -3 in decimal)</td>
</tr>
</tbody>
</table>

The same holds for extending an 8-bit number to a 16 bit number, extending a 32-bit number to a 64-bit number, etc.
**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 bit(s) that opened up as a result of the shift. For example,

**Shifting right by 3**

<table>
<thead>
<tr>
<th>Before shift</th>
<th>After shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011 0001 0010 ...</td>
<td>0000 0110 0010 010 ...</td>
</tr>
<tr>
<td>1011 0001 0010 ...</td>
<td>1111 0110 0010 010 ...</td>
</tr>
</tbody>
</table>

**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:

```c
uint64_t res = (uint64_t) A * (uint64_t) B;  //unsigned multiplication
```

If you want to produce a signed 64-bit result from the same two unsigned operands, you probably need to cast A and B to signed 32-bit integers, then cast them to 64-bit integers, then perform the multiplication (I don’t know why, but I figured this out through experimentation):

```c
int64_t res = (int64_t)((int32_t) A) * (int64_t)((int32_t) 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. For the purposes of this project, you will need to simulate the syscall instruction by defining the procedure syscall() in a file named syscall.c. If you look in the r_instructions.h file, you’ll see that the syscall procedure is declared by

```c
void syscall(uint32_t instruction);
```

The code that you write to define syscall() in syscall.c should operate as follows. It examines the value stored in R[2] (which is the register whose nickname is $V0). Depending on the value of R[2], your syscall() should perform the following actions:

- **R[2] == 1:** (“print_int”) print the integer value stored in R[4] (also known as $A0)
- **R[2] == 2:** (“print_float”) not supported, print an error message and call cpu_exit(1)
- **R[2] == 3:** (“print_double”) not supported, print an error message and call cpu_exit(1)
- **R[2] == 4:** (“print_string”) print the string that starts at Mem[R4].
- **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, print an error message and call cpu_exit(1)
R[2] == 7: (“read_double”) not supported, print an error message and call cpu_exit(1)
the size of the string buffer, so make sure the string is no longer than R[5] bytes.
R[2] == 9: (“sbrk”) not supported, print an error message and call cpu_exit(1)
R[2] == 10: (“exit”) call cpu_exit(0)
Otherwise: unsupported operation, print an error message and call cpu_exit(1)

Email me if you have any questions. Good luck!