CprE 381 Homework 1

[Note: This homework covers material on the construction and definition of ISAs and begins to get you introduced to MIPS. It also has you start to run MIPS programs through a simulator (SPIM or MARS) that you will use going forward both in homework AND to help test your term project. A little extra time spent working with SPIM now will pay off later.]

1. ISAs
    a. Identify the (concrete) benefits and drawbacks (at least one of each and three total) of establishing a hardware abstraction that uses simple, primitive instructions rather than implementing a high-level programming language directly.
    To appreciate the context, a hardware abstraction enables software designers (i.e., those writing assembly/compilers/system software and above) and hardware designers to focus on the design and optimization of their own components. Reducing the scope of designers allows them to be more productive. In addition, it enables hardware to generalized and reused for applications that do not even exist when a particular hardware design is fabricated (a key difference from very early computing devices that were fixed-function). These are large benefits, but also come with potential drawbacks: a specific general-purpose design may be less efficient (i.e., less optimized for performance, power, energy, area) than a design where the hardware and software are co-designed (custom application specific integrated circuits – ASICs – or application specific instruction set processors – ASIPs – are common).

    With respect to choosing how primitive vs application-specific/complex the hardware abstraction (i.e., ISA) is, there are both hardware and software tradeoffs. Benefits of a more primitive set of operations include simpler hardware design that can often be easier to optimize for performance/power/area and easier to write optimizing compilers (i.e., fewer human hours to develop software that convert high-level code to performant machine code. Drawbacks include the loss of efficiency that an ASIC or ASIP could provide and the potential for larger code size.

    b. Identify if the following items ***impact*** the ISA, ABI (but not ISA), or micro-architecture (i.e., a specific implementation of the ISA including which functional units are used and how they are interconnected). Note that some of these may have multiple answers. Provide a brief justification (1 sentence is sufficient).
        i. Number of named registers (ISA – the number of general-purpose registers that can be addressed/named in an instruction is something both the hardware and software need to know and agree on (i.e., the contract between HW/SW). Therefore, it is an important part of the ISA.

Of course, once chosen, the uArch must implement the appropriate register file and the ABI may choose to reserved registers for a specific purpose.)

ii. Number of cycles an instruction takes to execute (uArch – generally an ISA makes no specification as to how long an instruction takes to execute, just that it appears to complete before the next instruction begins.)
iii. Whether or not immediate operands can be used directly in arithmetic instructions (ISA – the types of operands that can be used is critical to the interface between hardware and software. This also impacts the uArch since it must implement immediate addressing.)
iv. Which register number is the stack pointer (ABI, ISA – in MIPS this is completely an ABI question since the stack is entirely managed by software and, therefore, the ISA requires no knowledge of which register holds the address of the top of the stack. In other ISAs, there is a hardware-implemented stack pointer and corresponding instructions to manipulate it. In this case, the uArch would also be impacted.)
v. Which, if any, registers numbers produce constants (e.g., 0 or -1) (ISA – HW-SW contract. The uArch would also have to support. Note that if you were to try to do this with the ABI, one could change the value/meaning of 0. o_O)
vi. Which register numbers pass arguments to function calls (ABI – in MIPS argument registers have identical hardware functionality to other general-purpose registers.)
vii. Which register numbers are temporary or saved (ABI – temporary vs saved registers have identical hardware functionality in MIPS. Indeed, only pieces of software which are part of the same call stack need to know whether or not to save a particular register.)
viii. Addressable address range for memory operations (ISA – the ISA will define the hardware bus widths to memory along with how addresses are constructed. For the version of MIPS we use, the total address is 32 bits constructed from the read of a 32-bit general-purpose register and a 16-bit immediate that is sign extended to 32 bits. Software must use this information when constructing memory addresses.)
ix. Type of adder used in the ALU (uArch – adders such as ripple-carry, carry-skip, carry-lookahead, etc all implement the identical function. An ISA would only specify the needed function such as 32-bit add.)
x. Which instructions update the PC (ISA – HW/SW contract.)
xi. Which bits of an instruction correspond to the opcode or an operand (ISA – HW/SW contract.)
xii. Number of functional units in the processor (uArch – the uArch needs to determine how many functional units, such as adders, it needs to achieve its desired performance. There is also some implicit need for additional functional units to support new instructions from an ISA perspective.)

2. MIPS
   a. Identify three products not mentioned in lecture that use a MIPS-based processor.
      Solutions widely variable, but here's the class list:
      i. Play Station
      ii. Nintendo 64
      iii. SGI Indigo

   b. MIPS does not have a clear instruction (clr dst). The intent of this command is to set the "dst" register to 0. Why would such a simple, basic instruction not be included in a RISC ISA? List three arithmetic or logical instructions or instruction sequences not mentioned during lecture that produce the same effect as a clr instruction.
      The RISC philosophy is to provide simple, fundamental operations as individual instructions and to not provide multiple instructions that perform the same task. Since the clr instruction can already be performed by several arithmetic instructions with careful operand selection, a clr instruction would just clutter the ISA and needlessly add to the complexity of hardware. The following is a non-exhaustive list of possible instructions that can perform the **`clr`** functionality:

```
 i.  xor dst, dst, dst
ii.  addi dst, $0, 0
iii. slt dst, dst, dst
iv.  srl $at, dst, 31
     srl $dst, $at, 1
```

   c. What does the following program do (give a description in English sentences)?

```
xor $1, $1, $2
xor $2, $1, $2
xor $1, $1, $2
```

      This program swaps the contents of register $1 and register $2.

   d. Assume that variables `a`, `b`, `c`, and `d` are mapped to registers `$s0`, `$s1`, `$s2`, and `$s3`, respectively. Write a MIPS program that implements

$$a = (\text{floor}(b / 16) \ \& \ c) - d \ \% \ 4$$

      using only **`add`**, **`sub`**, **`addi`**, **`and`**, **`andi`**, **`or`**, **`ori`**, **`sll`**, and **`srl`** instructions. Since `b`, `c`, and `d` must not be overwritten, use as many temporary registers (`$t0-$t9`) as needed.
      Several possible solutions, including the following:

```
li $s0, 0  # initialize a
li $s1, 32 # initialize b
li $s2, 4  # initialize c
li $s3, 4  # initialize d
```

srl $t0, $s1, 4 # floor(b / 16)
and $s0, $s2, $t0 # floor(b/16) & c
andi $t2, $s3, 3 # d % 4
sub $s0, $s0, $t2 # (floor($b$ / 16) & c) – d % 4

```
# modulus can be calculated with an andi as
# a % n is the same as computing a & (n-1)
# so t1 = d % 4 is MIPS andi $t1, $s3, 3
```

3. MARS Simulation
    a. Read the introduction to MARS found at http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpIntro.html and Part 1 of the MARS tutorial found at:http://courses.missouristate.edu/KenVollmar/mars/tutorial.htm. Some of the specifics of MIPS that are referenced you may just be learning about (or haven't learned about at all in the case of syscalls) However, you should be able to answer: does MARS simulate the ABI, ISA, and/or microarchitecture of MIPS? Explain. *[We'll use MARS since a modified version of it is used for testing the MIPS processors you will design for your term project.]*
    MARS effectively simulates the MIPS ISA and some amount of the ABI (e.g., syscalls and named registers). It certainly does not simulate microarchitectural characteristics where operations may take multiple cycles, etc. Although MARS does understand the ABI, it certainly is faithful to the ISA where pseudo instructions can be broken down into their composite parts and simulated as individual steps.
    b. Download the MARS simulator from https://canvas.iastate.edu/courses/88911/files/18164292?wrap=1. Load prob3.s into your simulator. Run three times with different inputs and report the result

(and corresponding inputs). Looking at the code, what does this program do? (write some C code that *could* compile to this)

prob3.s takes two inputs. The first is multiplied by 32. The second is used as an index into an array of words. The value at this index is added to the first input multiplied by 32 and printed.

A C-ish implementation (bolded part is the intended solution):

```
int a,b;
int vals = {25, 1, 4, 10, 381, 42, 100, 60, 0, 12, 25};
printf("Please enter a number:\n");
scanf("%d",a);
printf("Please enter a number:\n");
scanf("%d",b);
printf("%d",a*32 + vals[b]);
```

c. Modify prob3.s to find the minimum of two array values and write the value back to a third array location. The minimum of two numbers, x and y, can be given by:

$$MIN(x, y) = (x + y - abs(x - y))/2$$

To find the abs in MIPS, you may use the abs $t0, $t1 pseudo-instruction (sets $t0 to abs($t1)).

All three indices should be entered by the user. For this assignment you may assume that the user inputs correct values. Provide three test cases (inputs and observed output) showing your code works and describe how you verified correctness. Submit your new program as prob3_<Net_ID>.s.

The modified prob3.s (i.e., prob3_sol.s) is posted including comments.

To test I start with MIN(vals[0],vals[1]) = 1, a common case where the second value is the min. Then I test MIN (vals[6], vals[4]) = 100, a common case where the first value is the min. Lastly, I test the edge case where the two values are equivalent. MIN(vals[0], vals[10])= 25 odd common case. I overwrite a different value in each test case. Clearly there are many more cases to test, including out-of-bounds indexes, varying storage locations, including negative numbers in the array, etc.