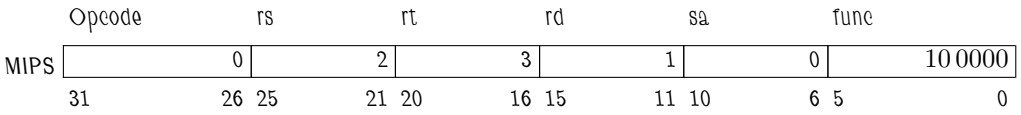
LSU EE 4720 **Homework 5** Solution **Due: 21 April 2023**

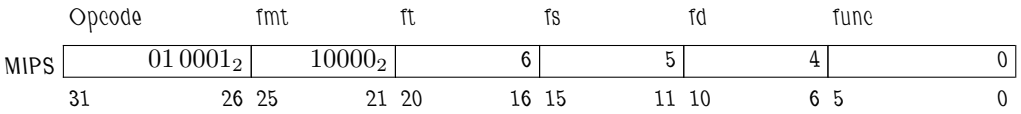
Problem 1: In this problem consider the encoding of integer and floating-point addition instructions in MIPS and RISC-V. Descriptions of MIPS and RISC-V are linked to the course references page, `https://www.ece.lsu.edu/ee4720/reference.html`.

(*a*) Show the encoding of MIPS instructions `add r1, r2, r3` and `add.s f4, f5, f6`.

The encoding of `add r1, r2, r3` is:

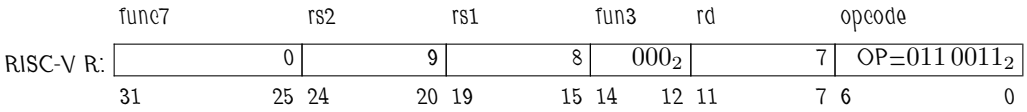


Instruction `add.s f4, f5, f6` is floating-point and in MIPS its format differs significantly from the integer instructions. (In many ISAs FP and integer instructions are encoded differently.) The register number fields are in different positions, and there is a format field, `fmt`, for specifying whether the operands are single- or double-precision. (The values for the `fmt` field can be found in Volume I Appendix A in Table A-11, in which the `fmt` field is confusingly called the `rs` field.)

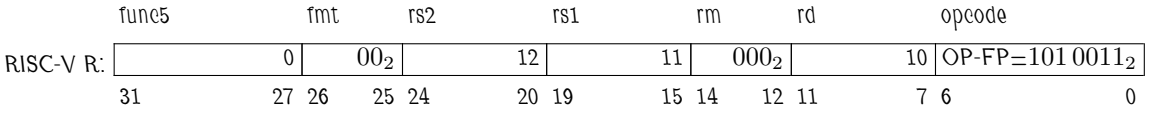


(*b*) Show the encoding of RISC-V RV32IF instructions `add x7, x8, x9` and `fadd f10, f11, f12`.

The encoding for `add x7, x8, x9` appears below. The encoding is most conveniently found in Chapter 24, RV32/64G Instruction Set Listing.



The encoding for `fadd f10, f11, f12` appears below. The encoding is most conveniently found in Chapter 24, RV32/64G Instruction Set Listing, in the table for RV32F. The problem did not specify which rounding mode (`rm`)to use, the encoding below uses *round-to-nearest, ties to even*.



(*c*) Notice that the register fields in the integer and floating-point RISC-V RV32IF instructions are the same, while the register fields in the two MIPS instructions are different. One possible reason for RISC-V's matching fields was to simplify implementations of the Zfinx variant. (Web search for it.) How do the matching fields reduce the cost of implementations of the RISC-V Zfinx variant?

In the Zfinx variant there is no separate floating-point register file. Instead, both integer and floating-point instructions' operands are from the same register file, the integer register file. Because integer and FP instructions operand fields are in the same place the connections from needed to retrieve integer instruction operands will also work for floating-point instruction operands.

ARM A64 Background

The following background will help in solving the next problem. MIPS and many other ISAs have a set of integer registers and a set of floating-point registers. Many newer ISAs, including ARM A64, have a set of *vector registers* in lieu of floating-point registers. Extensions of legacy ISAs, such as Intel 64 AVX2, have vector registers but retain floating-point registers for compatibility.

In many ISAs, including ARM A64, a vector register can be used to hold one FP value, just as a traditional FP register would, or a vector register can hold several values. *Scalar instructions* read or write one value per vector register, and *vector instructions* read and write multiple values per register.

In ARM A64 there are 32 128-bit vector registers, named `v0` to `v31`. When used in scalar instructions operating on single-precision FP values they are known by the names `s0` to `s31` and by the names `d0` to `d31` by double-precision scalar instructions. For example, the ARM A64 assembler instruction `fadd s0, s1, s2` computes `s0=s1+s2` and `fadd d0, d1, d2` computes `d0=d1+d2`. In both cases the operands were taken from vector registers `v0`, `v1`, and `v2`. The assembler name `s0` means use the low 32 bits of `v0` and interpret the value as an IEEE 754 single. The assembler name `d0` means use the low 64 bits of `v0` and interpret the value as an IEEE 754 double.

In the next problem, the one with `sum_thing_unusual`, the ARM code contains only scalar floating point instructions and base (integer register) instructions. To solve the next problem one needs to look up instructions in the ARM Architecture Reference Manual. Instructions that operate on vector registers, including `fadd` can be found in the *Advanced SIMD and Floating Point* section, C7.2 for the list of instructions. Other instructions can be found in the *A64 Base Instruction* section, C6.2 for the list of instructions.

Vector instructions are not needed in this assignment, but they will be briefly described anyway. For vector instructions the vector register name indicates how many elements in the vector to use, and what their format is. For example, `v0.4s`, means to use vector register `v0` and to split its 128 bits into 4 32-bit *lanes*, with each lane holding one float (the `s`). The names can be used in instructions such as `fadd v0.4s, v0.4s, v1.4s`. This instruction performs four additions, one on each lane of the vector register.

Problem 2: Appearing below is a C++ procedure with a `for` loop that computes the sum of elements in an array. This would be a totally ordinary loop were it not for the fact that the iteration variable, `i`, and the increment, `delta`, are both `floats`. Since `i` is a float the number of iterations, depending on `delta`, can be less than 1024 (say, if `delta=2.3`) or more than 1024 (say, if `delta=0.25`). Below the C code are MIPS-I and ARM A64 assembler versions of the loop. *Yes, that means you don't have to write them!* (The MIPS-I code was hand written, and the A64 was based on code generated by a compiler.) Notice that the ARM code is shorter than the MIPS code. That's because some of the ARM instructions do the equivalent of several MIPS instructions.

☑ Next to each ARM instruction indicate the MIPS instruction(s) from the MIPS code that it corresponds to.

☑ When an ARM instruction corresponds to more than one MIPS instruction explain what the ARM instruction is doing.

A short reference for MIPS floating-point instructions is the course lfp.s notes. This should be sufficient for all but the MIPS-II `trunc` instruction. For the `trunc` instruction see the MIPS documentation (linked to the course ISA page).

```
float sum_thing_unusual( float *a, float delta ) {
  float sum = 0;
  for ( float i = 0; i < 1024; i += delta ) sum += a[int(i)];
  return sum;
}
```

```
        # MIPS Code for sum_thing_unusual.
        #
        # $a0: The address of array a.
        # $f0: i.   At this point it contains a zero.
        # $f4: delta.
        # $f5: The constant 1024, in FP format.
        # $f8: sum. At this point it contains a zero.
LOOP:
        trunc.w.s $f6, $f0  # Note: This is a MIPS-II instruction.
        mfc1 $t1, $f6
        sll $t1, $t1, 2
        add $t2, $t1, $a0
        lwc1 $f7, 0($t2)
        add.s $f0, $f0, $f4
        c.lt.s $f0, $f5
        bc1f LOOP
        add.s $f8, $f8, $f7
```

Solution on next page.

The ARM instruction use is described below. Unlike MIPS, the ARM64 conversion instruction converts from FP to integer format, and writes the result to the integer register file. The ARM memory instructions, including `ldr`, can do more to compute an address than MIPS load instructions. They not only can add two registers, but also shift one of the registers, reducing the amount of code needed for array access.

```
.arch arm
        @ ARM A64 Code for sum_thing_unusual.
        @
        @ x0-x31: Integer registers. x31 is sometimes the zero register.
        @ s0-s31: Scalar single-precision floating-point registers.
        @
        @ x0: The address of array a.
        @ s0: sum. At this point it contains a zero.
        @ s1: i.   At this point it contains a zero.
        @ s3: delta.
        @ s4: The contains 1024, in FP format.
        @@ SOLUTION
LOOP:

        fcvtzs  x1, s1
        @ MIPS trunc. and mfc1.
        @ Convert FP in s1 to an integer, with truncation and write
        @ result in integer register x1.

        fadd    s1, s1, s3              @ MIPS add.s f0

        fcmpe   s1, s4
        @ The closest equivalent instruction is MIPS c.lt.s.
        @ But this instruction sets the comparison flags, NZCV, based
        @ on the comparison. (The names N, Z, C, V are for the result
        @ of an integer operation, but ARM uses the for FP comparisons too.)

        ldr     s2, [x0, x1, lsl 2]
        @ MIPS sll, add, lwc1
        @ ldr first computes an address by shifting x1 left two bits,
        @ then adds the result to x0. The memory at that address (and
        @ the three following) is read and the value placed in
        @ register s2.  Put another way:
        @ s2 = Mem[ x0 + x1*4 ];

        fadd    s0, s0, s2              @ MIPS add.s f8
        bmi     LOOP                    @ MIPS bc1t and partly c.lt.s
```