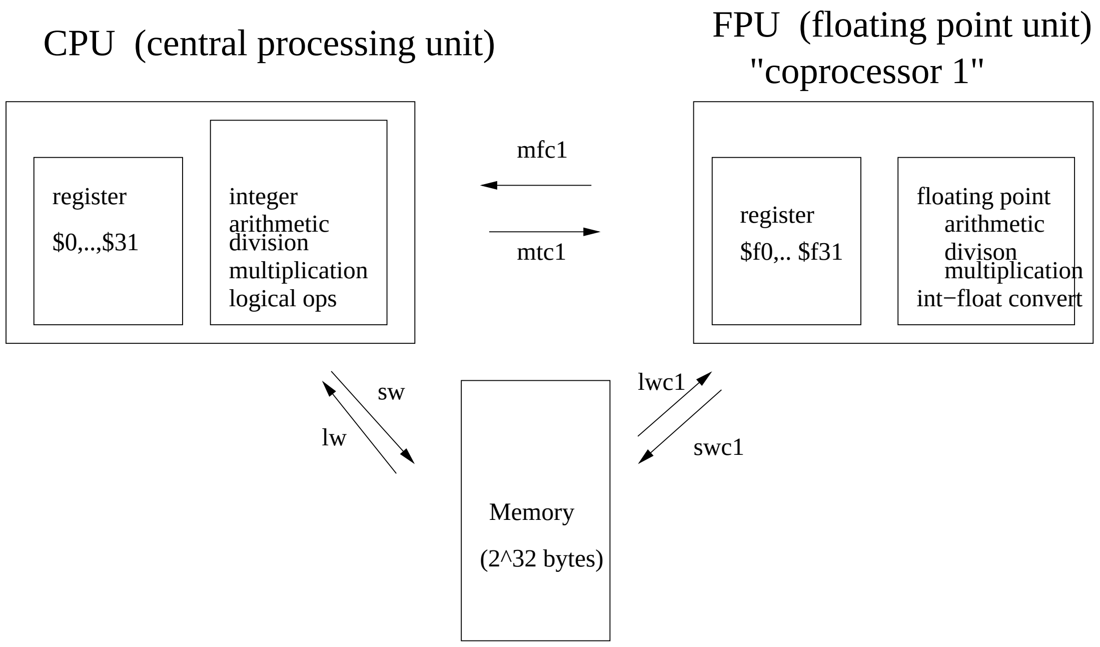
floating point registers in MIPS

As I mentioned in lecture 7, special circuits and registers are needed for floating point operations. The simple version of MIPS that we are using (called the R2000) was created back in the mid-1980s. At that time, it was not possible to fit the floating point circuits and registers on the same physical chip [1] as the chip that contained the CPU (including registers \$0-\$31, ALU, integer multiplication and division). Instead, the floating point operations were carried out on a physically separate chip called the *floating point coprocessor* or *floating point unit* (FPU) which in MIPS is called *coprocessor 1.* The FPU for MIPS has a special set of 32 registers for floating point operations, named `$f0, $f1, ... $f31.`

Recall that double precision floats require two words. In MIPS, double precision numbers require two registers. These are always consecutive registers, beginning with an even number register (`$f0, $f2,` etc). Thus, there is no need to reference both registers in the instruction. For example, a double precision number referenced by `$f2` in fact uses `$f2` and `$f3`.



floating point operations in MIPS

In MIPS, the instructions for adding and subtracting floating point numbers are of the form:

```
add.s   $f1, $f0, $f1     # single precision add
sub.s   $f0, $f0, $f2     # single precision sub
add.d   $f2, $f4, $f6     # double precision add
sub.d   $f2, $f4, $f6     # double precision sub
```

[1]by "chip", I mean the silicon-based electronics which contains the combinational and sequential circuits

Having a separate FPU takes some getting used to. For example, the following instructions have incorrect syntax and are not allowed.

```
 add.s   $s0, $s0,  $s1  #  NOT ALLOWED  (add.s expects FPU registers)
 add     $f0, $f2,  $f2  #  NOT ALLOWED  (add expects CPU registers)
```

Multiplication and division are done as follows. (Recall that multiplying or dividing floats in complicated: it requires special treatment of the significands and exponents, and hence a special set of circuits.) The instructions for multiplying or dividing two floats in MIPS are straightforward:

```
mul.s  $f0, $f1, $f2
div.s  $f0, $f1, $f2
```

similarly for double precision, except now we must use an even number register for the argument, e.g.

```
mul.d  $f0, $f0, $f2
div.d  $f0, $f0, $f2
```

There is no `Hi` and `Lo` register for floats, as there was for integers.

Data transfer operations

In order to perform the above operations, the floating point registers need to be filled. And after the operations, the results need to be put somewhere. There are two ways to move data to/from floating point registers. The first is to move words to/from the CPU registers. This is done with the "move to/from coprocessor 1" instruction:

```
mtc1      $s0,  $f0     #  Note unexpected order of operands here
mfc1      $f0,  $s0     #    "
```

The second way is to load or store a word to/from Memory, we use

```
lwc1  $f1,  40( $s1 )
swc1  $f1,  40( $s1 )
```

(Note that the memory address is held in a CPU register, not in a float register.)

To load/store double precision, we could use two operations to load/store the two words. It is easier though to just use a pseudoinstruction:

```
l.d   $f0,  -10( $s1 )
s.d   $f0,   12( $s1)
```

There is a corresponding pseudoinstruction for single precision i.e. `l.s` or `s.s`.

Type conversion (casting)

If you wish to perform an operation that has two arguments (for example, addition, subtraction, multiplication, division) and if one of the arguments is an integer and the other is a float, then one of these arguments needs to be converted (or "cast") to the type of the other. The reason is that the operation is either performed on the floating point circuits or on the integer circuits, but not some combination. The conversion is done on the FPU regardless of whether you are converting from integer to floating point or floating point to integer.

Let's say we wish to add an integer (in `$s0`) and a single precision float (in `$f2`). We have two options. First, we can move the integer from the (CPU) register to a FPU (c1) register, convert the integer to floating point, and then perform the operation as floating point, the result being a floating point number.

```
mtc1      $s0,  $f0
cvt.s.w   $f0,  $f0        #  Note: "w" (not "i") to indicate integer...
add.s     $f3,  $f2, $f0   #        ...  no, I don't know why
```

Alternatively we could convert the float to an integer, move the value to the CPU, and then perform the operation on the two integer arguments, the resulting being an integer.

```
mov.s     $f4,  $f0     #  $f4 is used as a temporary register here.
cvt.w.s   $f4,  $f4
mfc1      $t0,  $f4
add       $s3,  $t0, $s0   # $s0 holds argument.  $s3 holds result.
```

Sometimes we get the same number in the the two cases (ignoring the fact the one method gives a float and the other gives an int). But typically we will not.

Example 1

```
addi     $s0, $0, -8               $s0 = 0xfffffff8
mtc1     $s0, $f0                  $f0 = 0xfffffff8
cvt.s.w  $f1, $f0                  $f1 = 0xc1000000
```

Make sure you understand why the registers have the coded values shown on the right. This goes back to lectures 1 and 2.

Example 2

```
int   i = 841242345     //  This value in binary is more than 23
                        //  bits, but less than 32 bits.

int   j = (float) i;    //  Explicitly cast i to float, and then
                        //  implicitly cast it back to int (since
                        //  we store it in j.
```

Here is the same thing in MIPS.

```
mtc1     $s0, $f0
cvt.s.w  $f1, $f0
cvt.w.s  $f1, $f1
mfc1     $f1, $s0
```

Try the above in MIPS and verify that `$s0` and `$s1` have different values.

Example 3

```
double  x,y;
x  =  -0.3;
y  =  (float) x;
```

Here it is in MIPS:

```
d1 :   .double  -0.3
        l.d       $f0, d1         #  0xbfd3333333333333
        cvt.s.d   $f2, $f0        #  0xbe9999a
        cvt.d.s   $f4, $f2        #  0xbfd3333340000000
```

Note a few things:

- You have seen -0.3 as a single precision float before, namely on the midterm exam. When you load this number as a double, the repeated pattern of 3's is longer than it was in single precision (and you should understand why).
- When you convert from double to single precision, then least significant hex digit becomes a rather than 3. This is because there is roundoff (rather than truncation) when you go to 23 bits. On the midterm, I said to truncate.
- When you convert back to double, you are not just tacking 0's onto the end of the single precision format. The reason is that the exponent needs 11 bits in double, not 8. So the locations of the significand bits in the upper word of the double do not correspond exactly to the locations of the significand bits of the single precision.

Notice that casting from float or double to int will map a fractional number to an integer. But this should not be confused with truncating or rounding a fraction number to an integer and leaving the result represented as as an IEEE float. The operations `trunc.s`, `round.s`, `ceil.s`, `floor.s` (and their double versions) all map fractional numbers to integers, but the type does not change, that is, they don't map to `int`.

Conditional branch for floats

We do a conditional branch based on a floating point comparison in two steps. First, we compare the two floats. Then, based on the result of the comparison, we branch or don't branch:

```
c.____.s     $f2,  $f4
```

Here the "_____" is any comparison operator such as: `eq, neq, lt, le, ge, gt`. These compare operations are R format instructions.

The programmer doesn't have access to the result of the comparison. It is stored in a special D flip-flop, which is written to by the comparison instruction, and read from by the following conditional branch instruction:

```
bc1t  Label1   #  if the condition is true, branch to Label1
bc1f  Label1   #  if the condiiton if false, branch to Label1
```

The same same one bit register is used when comparisons are made using double precision numbers.

```
c._____.d $f2, $f4
```

System call

You can print and read `float` and `double` from the console, using `syscall`.

```
                     $v0     from/to (hardwired i.e. no options)
                     ---     -------
print float           2        $f12
print double          3        $f12
read  float           6        $f0
read  double          7        $f0
```

MIPS Coprocessor 0: the System Control Coprocessor

There is another coprocessor, called coprocessor 0, which is used for "exception handling". (We will discuss exceptions in detail later in the course. For now, think of an exception as an event that requires your program to stop and the operating system to have to do something. A `syscall` is an example.) Coprocessor 0 has special registers which are used only in kernel mode. When you run MARS, you will see four of these registers displayed, namely:

- *EPC: ($14)* The exception program counter contains a return address in the program. It is the address in the program that follows the instruction that caused the exception.
- *Cause: ($13)* contains a code for what kinds of exception occured. (invalid operation, division by zero, overflow)
- *BadVaddr: ($8)* holds then address that led to an exception if the user program tried to access an illegal address e.g. above 0x80000000.
- *Status: ($12)*: says whether the processor is running in kernel mode or user mode. Says whether the processor can be interrupted by various possible interrupts (We will discuss interrupts in a few weeks).

Example 1: overflow error

Here is an example in which I cause an overflow error. First, I put 2^31 into `$s0`. Notice that an *unsigned* representation is used. Then I add this value to itself which gives 2^{32} which is too big. [2]

```
0x0040070              addiu   $s0,  $0, 1     #  $s0 = 1
0x0040074              sll     $s0,  $s0, 31   #  $s0 = 2^31
0x0040078              addu    $t0,  $s0, $s0  #  $t0 = 2^32 -> overflow
```

If you step through this code, you'll see that the c0 registers change their values when the program crashes. The code what caused the crash and they code where the crash occurred (`EPC = 00400078`).

Example 2: division by zero

```
addiu   $s0,  $0, 5     #  $s0 = 5
sll     $s1,  $0, 0     #  $s0 = 0
div     $t0,  $s0, $s1  #  $t0 = 5/0 -> division by zero
```

MARS produces the same error as in the previous example.

Example 3: illegal address

```
str:    .asciiz  "hello"
        la  $s0,  str
        j   str
```

The MARS assembler gives an error on this one. It knows you are trying to jump to the data segment, which makes no sense. Instructions are not data.

Example 3: illegal address

```
inst:    la  $s0,  inst
#        la  $s0,  str            This line commented out!
         lw  $t0,  0($s0)
```

This is the opposite problem. Now you are trying to load from the text (instruction) segment. The MARS assembler cannot detect this problem, since the problem only occurs when the two instructions are combined. For example, if you were to uncomment the middle line, then it would run fine, namely it would load the 4 bytes starting at data address `str`.

[2]There was a typo in the original slides. The second `addu` was written `add`, which is not what I wanted.

Floating point exceptions?

In the last part of the lecture, I gave some examples of similar operations in floating point. These illustrate how certain events such as division by zero behave differently with floats than with ints, namely some operations that produce exceptions with ints do *not* produce exceptions with floating point.

Example 4: Overflow

```
  addi     $s0, $0, 1
  sll      $s0, $s0, 30   #  $s0 = 2^30
  mtc1     $s0, $f0
  cvt.s.w  $f0, $f0       #  $f0 = 2^30 = 1.00000000000000000000000 x 2^30
  mul.s    $f0, $f0, $f0  #  $f0 = 2^60
  mul.s    $f0, $f0, $f0  #  $f0 = 2^120
  mul.s    $f0, $f0, $f0  #  $f0 = 2^240  ->  overflow
```

Interesting, no exception occurs here. Instead, if you look at the result that ends up in `$f0`, you will find that it represents $+\infty$.

Example 5: Division by zero

```
float1:    .float  13.4
           l.s    $f0,  float1
           mtc1   $0,   $f1        #  Note that no cast is needed since
                                   #  0x00000000 also is 0 in float.  Wow!
           div.s  $f2,  $f0,  $f1
```

Again, no exception occurs. The result is $+\infty$.

Example 6: 0/0

```
         mtc1   $0,   $f1
         div.s  $f2,  $f1, $f1     #   0/0
```

The result is `NaN`, but no exception occurs.