In today's lecture we will look at two "co-processors", namely the floating point processor (called CP1 or the FPU) and the kernel processor (called CP0 or the 'system control' processor). We will look at the MIPS assembly language instructions for this processor.

This is the last lecture above MIPS programming. After this, we will go back to the circuits and connect the general ideas about circuits to the particular instructions we have seen in MIPS, mostly CPU instructions but occasionally CP0 too.

Integer multiplication and division in MIPS

[ASIDE: The slides also start out with this mini-topic, but at the beginning of the lecture, I decided to skip over this topic. I then returned to the topic (slides) a bit later.]

In Assignment 1, you built a simple but slow circuit for multiplying two unsigned integers and, in lecture 7, I discussed more complicated circuits for how you could perform fast multiplication. You should appreciate that there are many ways this could be done. Those details should now be put "under the hood". Let's just look at multiplication from the MIPS programmer's perspective.

In MIPS assembly language, there is a multiplication instruction for signed integers, `mult`, and for unsigned integers `multu`. Since multiplication takes two 32 bit numbers and returns a 64 bit number, special treatment must be given to the result. The 64 bit product is located in a "product" register. You access the contents of this register using two separate instructions.

```
mult  $s0, $s1   #  Multiply the numbers stored in these registers.
                 #  This yields a 64 bit number, which is stored in two
                 #  32 bits parts:  "hi" and "lo"
mfhi  $t0        # loads the upper 32 bits from the product register
mflo  $t1        # loads the lower 32 bits from the product register
```

You can only read from the product register. You cannot manipulate it directly. In MARS, the product register is shown as two 32 bit registers, HI and LO. If the HI register has all 0's, then the product that is computed can be represented by 32 bits (what's in the LO register). Otherwise, we have a number that is bigger than the maximum `int` and it would need to be treated separately. Details are omitted here.

What about division? To understand division, we need to recall some terminology. If we divide one positive integer by another, say 78/21, or more generally "dividend/divisor" then we get a quotient and a remainder, i.e.

$$\text{dividend} \;=\; \text{quotient} \;*\; \text{divisor} \;+\; \text{remainder}$$

$$e.g. \quad 78 = 3 * 21 + 15$$

In MIPS, the divide instruction also uses the HI and LO registers, as follows:

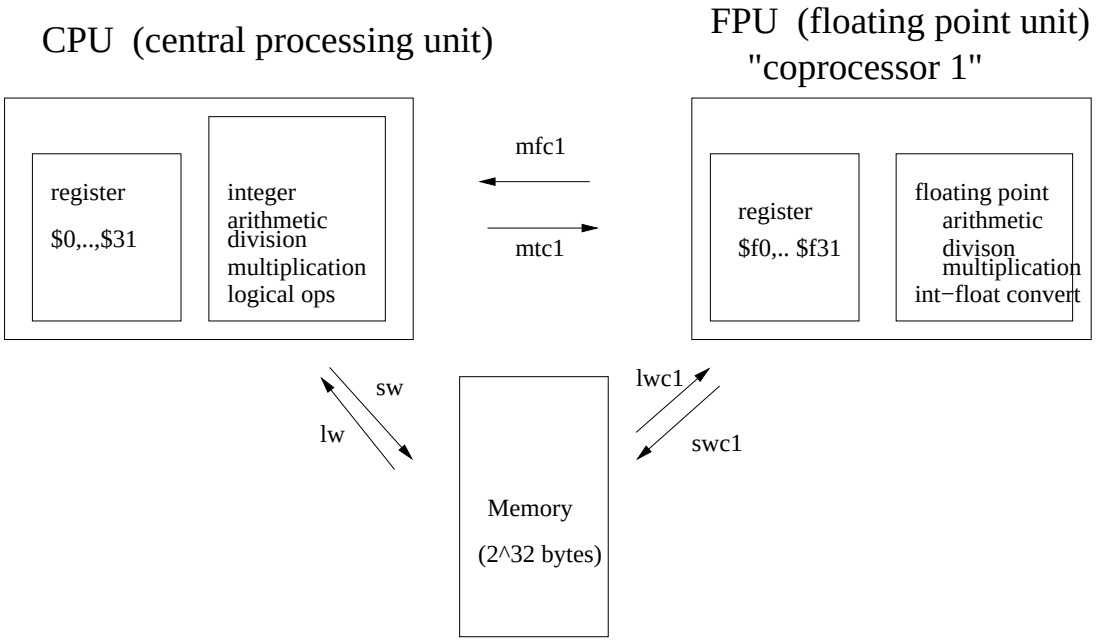
```
div   $s0, $s1   #  Hi contains the remainder,  Lo contains quotient
mfhi  $t0        #  remainder moved into $t0
mflo  $t1        #  quotient moved into $t1
```

The `mult, div, mfhi, mflo` are all R format instructions.

floating point in MIPS

As I also 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`.



The MIPS 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
```

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)
```

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

Multiplication and division are done as follows.

```
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,

```
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 must 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 order of operands here
mfc1      $s0,  $f0     #    "
```

The second is to load/store a word to/from Memory:

```
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. The conversion is done on the FPU though, regardless of whether you are converting from integer to float or float to integer.

The MIPS instruction for cast is `cvt._._` where the underline is filled with a symbol `f` (for single precision float), `d` (for double precision float), `w` (for integer). I guess MIPS avoids using `i` for integer because it often indicates immediate.

Example 1

Let's say we have an integer, say -8 in `$s0`, and we want to convert it to a single precision float and put the result in `$f0`. First, we move the integer from the CPU to FPU (c1). Then we convert.

```
addi     $s0, $0, -8     #  $s0 = 0xfffffff8
mtc1     $s0, $f0        #  from $s0 to $f0  ->  $f0 = 0xfffffff8
cvt.s.w  $f0, $f0        #  from w to s      ->  $f0 = 0xc1000000
```

As an exercise, recall from the first few lectures of the course why the registers have the coded values shown on the right.

Similarly, suppose we have a single precision float (in `$f0`) and we want to convert it to an integer and put it into `$s0`. Here is how we could do it.

```
cvt.w.s   $f2,  $f0        #  we use $f2 as a temp here
mfc1      $s0,  $f2
```

Example 2

```
int   i = 841242345    //  This value in binary would be more than 23
                       //  bits, but less than 32 bits.  (Specifically
                       //  we cannot  write it exactly as a float.)

int   j = (float) i;   //  Explicitly cast i to float, and then
                       //  implicitly cast it back to int (implicit
                       //  since we store the result as an int j).
```

Here is the same thing in MIPS.

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

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

Example 3

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

Here it is in MIPS:

```
        .data
 d1 :   .double  -0.3

        .text
         l.d       $f0, d1      #  $f0 = 0xbfd3333333333333 <-  these two...
         cvt.s.d   $f2, $f0     #  $f2 = 0xbe9999a
         cvt.d.s   $f4, $f2     #  $f4 = 0xbfd3333340000000 <- ... are different
```

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 condition if false, branch to Label1
```

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

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

System call

I did not mention it in the slides, but you also can print and read `float` and `double` from the console, using `syscall`. Rather than printing from or reading to an argument register, it uses a particular coprocesor 1 register shown in table below.

	$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. Coprocessor 0 has special registers which are used only in kernel mode. One of the uses of this coprocessor is "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.) *Exceptions are errors that happen at runtime. They cannot be detected by the assembler i.e. before the program runs.*

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 the 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 which produces an overflow error. First, I put 1 into `$s0` and then multiply it by 2^{30}. Then I added this value to itself which gives 2^{31} which is one bigger than the largest signed 32 bit integer, i.e. overflow.

```
addi    $s0,  $0, 1     #  $s0 = 1
sll     $s0,  $s0, 30   #  $s0 = 2^30
add     $t0,  $s0, $s0  #  $t0 = 2^31 -> overflow (signed)
```

If you step through this code, you'll find that it crashes and that the c0 registers change when this happens. The registers encode what caused the crash and they encode the instruction address where the crash occurred.

Note that if I insert an instruction which subtracts 1, then I don't get overflow since the sum is $2^{31} - 2$ which is less than the largest signed int.

```
add     $s0,  $0, 1     #  $s0 = 1
sll     $s0,  $s0, 30   #  $s0 = 2^30
addi    $s0,  $s0, -1   #  $s0 = 2^30 - 1  <--- inserted
add     $t0,  $s0, $s0  #  $t0 = 2^31 - 2
```

Example 2: illegal address

Here is a different type of exception.

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

The problem is that the program is jumping (from the text segment) to the data segment, which makes no sense. Instructions are not data.

Example 3: illegal address

```
inst:    la  $s0,  inst
         lw  $t0,  0($s0)
```

This is the opposite problem. Now I am trying to load from the text (instruction) segment. Note that the MARS assembler cannot detect this problem, since the problem only occurs at runtime, once it is determined that `$s0` contains an address in the text segment, not the data segment.

Floating point exceptions?

Here are some examples of similar operations in floating point. These illustrate how certain events 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`, namely `0x7f80 0000`, you will find that it represents $+\infty$, namely 0 for the sign bit, 11111111 for the exponent, and then 23 0's for the significand.

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 again $+\infty$.

Example 6: 0/0

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

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