Тема 16. Архитектуры вычислительных машин и языки ассемблера, часть 5

16.1 Плавающие числа в MIPS32

В MIPS используется стандартизованное (IEEE 754) представление числе с плавающей точкой, как для 32-, так и для 64 битной версий. Далее мы описываем только 32-битные инструкции, хотя 64-битные оказываются похожими.

«Плавающие» команды на MIPS изначально выполнялись на сопроцессоре под названием FPA (*Floating Point Accelerator*). В современных MIPS-системах доступны команды, выполняемые на центральном процессорном ядре. Однако, эти команды иногда работают так, как если бы они были на отдельном сопроцессоре.

В MIPS 32 регистра для чисел с одинарной точностью (32-битные bit). При этом:

- Их имена попадают в диапазон $*f*0 - $*f*31.

- Регистр $*f*0 не является специализированным, т.е. в нем можно хранить любые битовые последовательности, а не только нуль.

- Команды загрузки, сохранения, арифметики и др. работают с этими регистрами и не используют регистры общего назначения.

Также имеются возможности для операций над числами двойной точности (64 бита). Для этого используются **пары** из **регистров** одинарной точности. Эти пары нумеруются понятным образом: $*f*0, $*f*2, ... , $*f*30. Как видно, пара адресуется регистром с четным номером. Подразумевается, что нечетные регистры включаются в пары автоматически.

Отметим, что некоторые MIPS-системы при выполнении «плавающих» команд одинарной точности допускают только регистры с четными номерами. Однако, симулятор SPIM в таких командах разрешает пользоваться всеми 32 регистрами, поэтому данном соглашению мы и следуем.

На реальной аппаратуре наблюдается задержка между загрузкой команды и моментом, когда данные оказываются в регистре. Электроника основной памяти управляет всеми битовыми последовательностями одинаковым образом, поэтому такая пауза будет независимо от того, что эти последовательности будут содержать.

Симулятор SPIM допускает отключение задержки во время загрузки, чем мы и можем воспользоваться, т. к. плавающие числа и так по своей природе непросты для понимания.

16.2 Операции загрузки и сохранения

Команда **загрузки** значения одинарной точности выполняется следующей **псевдоинструкцией**:

```
l.s   fd, addr      # load register fd from addr
                    # (pseudoinstruction)
```

Эта команда загрузит 32-битные данные, находящиеся по адресу *addr* в регистр $*fd*, где $fd — это любой регистры из списка $*f*0, $*f*1, ... $*f*31. Что бы не находилось в *addr*, оно копируется в *fd*. Если данные не имеют смысла с точки зрения «плавающих» операций, для данной инструкции это все равно. Позднее ошибка будет обнаружена при попытке выполнения другой инструкции.

Иногда «плавающие» регистры используются в качестве временного хранилища

целочисленных данных. Например, вместо сохранения временного значения в ячейку памяти, его можно скопировать в неиспользованные «плавающий» регистр. Это нормально до тех пор, пока над данными не выполняется плавающая арифметика.

Команда **сохранения** значения одинарной точности выполняется следующей **псевдоинструкцией**:

```
s.s   fd, addr      # store register fd to addr
                    # (pseudoinstruction)
```

Любое 32-битное значение из *fd* будет скопировано в *addr*.

В обеих описанных псевдоинструкциях *addr* — символьная метка или индексированный адрес.

Имеется псевдоинструкция для **загрузки непосредственного** плавающего **значения** одинарной точности. Таким образом можно загрузить константное значение, заданное в качестве аргумента команды.

```
li.s   fd,val      # load register $fd with val
                   # (pseudoinstruction)
```

Следующий фрагмент кода иллюстрирует использование псевдоинструкции:

```
li.s    $f1, 1.0             # $f1 = constant 1.0
li.s    $f2, 2.0             # $f2 = constant 2.0
li.s    $f10, 1.0e-5         # $f10 = 0.00001
```

Далее приведен код программы, которая обменивает значения переменных *valA* и *valB*. Следует обратить внимание на то, как записаны два плавающих значения. Первое в привычной нотации, второе — в научной.

```
## swap1.s
##
## Exchange the values in valA and valB

        .text
        .globl  main

main:
        l.s     $f0, valA        # $f0  <-- valA
        l.s     $f1, valB        # $f1  <-- valB
        s.s     $f0, valB        # $f0  --> valB
        s.s     $f1, valA        # $f1  --> valA

        li      $v0, 10          # code 10 == exit
        syscall                  # Return to OS.

        .data
valA:   .float  8.32             # 32 bit floating point value
valB:   .float  -0.6234e4        # 32 bit floating point value
                                 # small 'e' only
```

Для выполнения арифметических операций плавающие числа должны быть загружены в специальные регистры.

Наша последняя программа просто обменивала битовые последовательности в ячейках памяти. Эта задача может быть решена с использованием регистров общего назначения, поскольку никаких арифметических операций не было, что и иллюстрирует следующая программа:

```
## swap2.s
##
## Exchange the values in valA and valB
```

```
        .text
        .globl  main

main:
        lw      $t0, valA       # $t0  <-- valA
        lw      $t1, valB       # $t1  <-- valB
        sw      $t0, valB       # $t0  --> valB
        sw      $t1, valA       # $t1  --> valA

        li      $v0, 10         # code 10 == exit
        syscall                 # Return to OS.

        .data
valA:   .float  8.32            # 32 bit floating point value
valB:   .float  -0.6234e4       # 32 bit floating point value
                                # small 'e' only
```

Для инструкций загрузки и сохранения адреса ячеек памяти должны быть выровнены по границе машинного слова. Иначе говоря, адрес должен быть степенью четверки. Обычно, это не проблема, все на себя берет ассемблер.

Для вывода плавающего значения в симуляторе SPIM используются системные вызовы 2 для одинарной точности и 3 — двойной. Для ввода плавающего значения используются системные вызовы 6 для одинарной точности и 7 — двойной.

Сервис	Операция	Код (в регистре $v0)	Аргументы	Результаты
print_float	Вывести 32-битное «плавающее»	2	В $f12 число с плавающей точкой	Нет
print_double	Вывести 64-битное «плавающее»	3	В ($f12, $f13) число с плавающей точкой	Нет
read_float	Получить от пользователя 32-битное число с плавающей точкой	6	Нет	Число в регистре $f0
read_double	Получить от пользователя 64-битное число с плавающей точкой	7	Нет	В ($f12, $f13) число

В зависимости от системного вызова может потребоваться размещение аргументов в других регистрах. В следующем примере кода выводится число с плавающей точкой. Сначала все выполняется корректно с использованием вызова 2. Затем программист ошибочно указал системный вызов 1 (он предназначен для целого числа). Разумеется, 32-битное плавающее число можно интерпретировать как целое, поэтому-то системный вызов 2 делает все так, как требуется.

```
## print.s
##
## Print out a 32 bit pattern, first as a float,
## then as an integer.

        .text
        .globl  main

main:
        l.s     $f12, val       # use the float as an argument
        li      $v0, 2          # code 2 == print float
        syscall                 # (correct)

        li      $v0, 4          # print
        la      $a0, lfeed      # line separator
        syscall
```

```
        lw      $a0, val        # use the float as a int
        li      $v0, 1          # code 2 == print int
        syscall                 # (mistake)

        li      $v0,10          # code 10 == exit
        syscall                 # Return to OS.

        .data
val :   .float  -8.32           # floating point data
lfeed:  .asciiz "\n"
```

Такой класс ошибок знаком тем, кто обладает опытом программирования на языке C, который является языком с нестрогим контролем типов. По понятным причинам ассемблер является языком без контроля типов.

16.3 Арифметические операции

Далее приведено описание нескольких операций одинарной точности. Каждая из них соответствует одной машинной команде. Доступны версии этих операций для чисел с двойной точностью. У них суффиксы *s* заменены на *d*. Так *add.s* становится *add.d* и соответствует сложению чисел двойной точности.

Первая инструкция в таблице вычисляет модуль числа, расположенного в регистре $*fs*.

Если данные в операнде неверные или выполнена некорректная операция (деление на 0), то возникает исключительная ситуация. Стандарт IEEE 754 описывает, что должно быть сделано в подобных ситуациях.

```
Инструкция              Семантика
abs.s fd, fs            $fd = |$fs|
add.s fd, fs, ft        $fd = $fs + $ft
sub.s fd, fs, ft        $fd = $fs - $ft
mul.s fd, fs, ft        $fd = $fs * $ft
div.s fd, fs, ft        $fd = $fs / $ft
neg.s fd, fs            $fd = - $fs
```

16.4 Копирование

```
Инструкция              Семантика               Комментарии
mov.s   fd, fs          $fd = $fs
mtc1    rs, fd          $fd = $rs               Копирование в сопроцессор
                                                Внимание: обратный порядок!
mfc1    rd, fs          $rd = $fs               Копирование из сопроцессора
```

Следующий пример иллюстрирует использование этих инструкций.

```
 between the coprocessor and the CPU

        .text
        .globl main

main:
        li      $t0, 1          # $t0 <-- 1
                                # (move to the coprocessor)
        mtc1    $t0, $f0        # $f0 <-- $t0

        li.s    $f1, 1.0        # $f1 <-- 1.0
                                # (move from the coprocessor)
        mfc1    $t1, $f1        # $t1 <-- $f1

        li      $v0, 10         # exit
        syscall
```

Содержимое регистров в SPIM может выглядеть следующим образом.

```
Single Precision
FG0  = 1
FG1  = 3f800000
FG2  = 0
FG3  = 0
FG4  = 0
FG5  = 0
FG6  = 0
FG7  = 0
FG8  = 0
```

```
R0  [r0] = 0
R1  [at] = 3f800000
R2  [v0] = a
R3  [v1] = 0
R4  [a0] = 1
R5  [a1] = 7ffff958
R6  [a2] = 7ffff960
R7  [a3] = 0
R8  [t0] = 1
R9  [t1] = 3f800000
R10 [t2] = 0
```

Битовая последовательность 00000001 — это дополнительный код для единицы, которое расположено в регистрах $*t*0 and $*f*0.

Битовая последовательность 3f800000 — это IEEE-представление числа 1.0, расположенное в регистрах $*f*1 and $*t*1. Также оно хранится в регистре $*at*, которые используется в псевдоинструкции *li.s*.

Пример кода, вычисляющего значение ax^2+bx+c, приведен далее:

```
.text
        .globl main

main:   # read input
        la      $a0, prompt         # prompt user for x
        li      $v0, 4              # print string
        syscall

        li      $v0, 6              # read single
        syscall                     # $f0 <-- x

        # evaluate the quadratic
        . . . . .

        .data
        . . . . .

prompt: .asciiz "Enter x: "
```

Как видно, программа стартует с запроса на пользовательский ввод значения *x*. После выполнения инструкции *syscall* значение, полученное от пользователя, будет находится в $*f*0.

Следующая часть кода вычисляет значение полинома.

```
# Register Use Chart
        # $f0 -- x
        # $f2 -- sum of terms
        . . . . .

        # evaluate the quadratic
        l.s     $f2, a              # sum = a
        mul.s   $f2, $f2, $f0       # sum = ax

        l.s     $f4, bb             # get b
        add.s   $f2, $f2, $f4       # sum = ax + b
        mul.s   $f2, $f2, $f0       # sum = (ax+b)x = ax^2 + bx

        l.s     $f4, c              # get c
        add.s   $f2, $f2, $f4       # sum = ax^2 + bx + c
        . . . . . .
        .data
a:      .float  1.0
bb:     .float  1.0
c:      .float  1.0
```

Последняя часть кода.

```
. . . . .

        # print the result
        mov.s   $f12, $f2           # $f12 = argument
        li      $v0, 2              # print single
        syscall

        la      $a0, newl           # new line
        li      $v0, 4              # print string
        syscall

        li      $v0, 10             # code 10 == exit
        syscall                     # Return to OS.

        .data
        . . . . .

blank:  .asciiz " "
newl:   .asciiz "\n"
        . . . . .
```

16.5 Сравнения и переходы

В MIPS так называемый Floating Point Accelerator содержит бит условия, который устанавливается в 0 или 1, чтобы отразить факт ложности или истинности условия. Несколько FPA -инструкций изменяют этот бит, а также несколько инструкций процессора проверяют этот бит.

Проверка двух чисел с плавающей точкой на точное равенство в большинстве случаев не очень хорошая идея. Как известно, вычисления с плавающей точкой не являются точными, то есть значения в регистрах не равны, даже если математически они должны быть равными. В связи с этим рекомендуется проверять на «меньше, чем» или «меньше или равно, чем» вместо проверки на точную эквивалентность. Следующая таблица содержит описание некоторых сравнений в MIPS применительно к плавающим числам.

```
Инструкция              Семантика
c.eq.s fs, ft           if $fs == $ft
                          condition bit = 1
                        else
                          condition bit = 0
c.lt.s fs, ft           if $fs < $ft
                          condition bit = 1
                        else
                          condition bit = 0
c.le.s fs, ft           if $fs <= $ft
                          condition bit = 1
                        else
                          condition bit = 0
```

Эти инструкции изменяют значение **условного бита**, являющегося частью слова состояния процессора. Если условие истинно, то условный бит устанавливается в 1. В противном случае он сбрасывается (значение 0). В списке приведен не полный перечень инструкций сравнения. Остальные инструкции изучают результаты возникновения исключительных ситуаций.

Инструкции перехода изучают условный бит. Команда *bc1t* осуществляет переход, если бит установлен. Команда *bc1f* осуществляет переход, если бит сброшен.

```
Инструкция              Семантика
bc1t label              branch to label
                        if the coprocessor 1 condition bit is true
```

```
bc1f label              branch to label
                        if the coprocessor 1 condition bit is false
```

Аппаратура требует после каждого перехода выполнить одну инструкцию NOP.

Следующий пример исследует два числа с плавающей точкой A и B, и выводит сообщение с меньшим из них. Сперва программа загружает два числа в регистры.

```
main:   # get the values into registers
        l.s     $f0, A
        l.s     $f2, B
        . . . .

A:      .float  4.830
B:      .float  1.012
        . . . .
```

После этого программа сравнивает, $A < B$ или $B < A$. Если ни то, ни другое, тогда, очевидно, $B == A$.

```
        . . . .
        c.lt.s  $f0, $f2          # is A < B?
        bc1t    printA            # yes:  print A

        c.lt.s  $f2, $f0          # is B < A?
        bc1t    printB            # yes:  print B

        la      $a0, EQmsg        # otherwise
        li      $v0, 4            # they are equal
        . . . .
```

Код полностью:

```
## min.s --- determine the min of two floats
##
        .text
        .globl main

main:   # get the values into registers
        l.s     $f0, A
        l.s     $f2, B

        c.lt.s  $f0, $f2          # is A < B?
        bc1t    printA            # yes -- print A
        c.lt.s  $f2, $f0          # is B < A?
        bc1t    printB            # yes -- print B

        la      $a0, EQmsg        # otherwise
        li      $v0, 4            # they are equal
        syscall
        mov.s   $f12, $f0         # print one of them
        b       prtnum

printA: la      $a0, Amsg         # message for A
        li      $v0, 4
        syscall
        mov.s   $f12, $f0         # print A
        b       prtnum

printB: la      $a0, Bmsg         # message for B
        li      $v0, 4
        syscall
        mov.s   $f12, $f2         # print B

prtnum: li      $v0, 2            # print single precision
                                  # value in $f12
        syscall
        la      $a0, newl
```

```
        li      $v0, 4            # print new line
        syscall
        jr      $ra               # return to OS

        .data

A:      .float  4.830
B:      .float  1.012
Amsg:   .asciiz "A is smallest: "
Bmsg:   .asciiz "B is smallest: "
EQmsg:  .asciiz "They are equal: "
newl:   .asciiz "\n"
```

Еще один пример иллюстрирует реализацию метода Ньютона для приближенного вычисления квадратного корня числа.

```
## newton.asm -- compute sqrt(n)

## given an approximation x to sqrt(n),
## an improved approximation is:

## x' = (1/2)(x + n/x)

## $f0  ---  n
## $f1  ---  1.0
## $f2  ---  2.0
## $f3  ---  x  : current approx.
## $f4  ---  x' : next approx.
## $f8  ---  temp

        .text
        .globl main

main:

        l.s     $f0, n                # get n
        li.s    $f1, 1.0              # constant 1.0
        li.s    $f2, 2.0              # constant 2.0
        li.s    $f3, 1.0              # x == first approx.
        li.s    $f10, 1.0e-5          # five figure accuracy

loop:
        mov.s   $f4, $f0              # x' = n
        div.s   $f4, $f4, $f3         # x' = n/x
        add.s   $f4, $f3, $f4         # x' = x + n/x
        div.s   $f3, $f4, $f2         # x    = (1/2)(x + n/x)

        mul.s   $f8, $f3, $f3         # x^2
        div.s   $f8, $f0, $f8         # n/x^2
        sub.s   $f8, $f8, $f1         # n/x^2 - 1.0
        abs.s   $f8, $f8             # |n/x^2 - 1.0|
        c.lt.s  $f8, $f10            # |x^2 - n| < small ?
        bc1t    done                 # yes: done

        j       loop                # next approximation

done:
        mov.s   $f12, $f3            # print the result
        li      $v0, 2
        syscall

        jr      $ra                  # return to OS

##
```

```
##  Data Segment
##
        .data
n:      .float  3.0
```

Литература и дополнительные источники к Теме 16

1. MIPS32 Architecture - https://imgtec.com/mips/architectures/mips32/
2. http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html
3. http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm
4. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html