Архитектура вычислительных систем

Лекция 8.

Яревский Е.А.

Кафедра вычислительной физики

Ассемблер MIPS

Эмулятор:
SPIM: A MIPS32 Simulator (QtSpim)
http://spimsimulator.sourceforge.net/

Информация:
1) Помощь в QtSpim.
2) Д.М. Хэррис, С.Л. Хэррис. Цифровая схемотехника и архитектура компьютера // 2-е издание. Morgan Kaufman (English Edition), 2013. 1627 с.
3) Д. Паттерсон, Дж. Хеннеси. Архитектура компьютера и проектирование компьютерных систем // 4-е издание. СПб., Питер, 2012. 784 с.

Архитектурное состояние

Архитектурное состояние (*architectural state*) микропроцессора содержит состояние программы.
Архитектурное состояние процессоров MIPS включает в себя содержимое регистрового файла и счётчика команд.
Если операционная система в какой-либо момент выполнения программы сохранит архитектурное состояние, то сможет эту программу прервать, сделать что-то ещё, а потом восстановить архитектурное состояние, после чего прерванная программа продолжит выполняться, как будто прерывания не было.

Когда одна функция вызывает другую, необходимо соглашение о том, где размещать аргументы и возвращаемое значение.

В архитектуре MIPS, вызывающая функция размещает до четырёх аргументов в регистры ***$a0–$a3*** перед тем, как произвести вызов, а вызываемая функция помещает возвращаемое значение в регистры ***$v0–$v1*** перед тем, как завершить работу.

Вызываемая функция должна знать, куда передать управление после завершения работы, и она не должна портить значения любых регистров или памяти, которые нужны вызывающей функции.
Вызывающая функция сохраняет адрес возврата (*return address*) в регистре адреса возврата (***$ra***) в тот момент, когда она передаёт управление вызываемой функции путем выполнения инструкции безусловного перехода с возвратом (***jal***).
Вызываемая функция не должна изменять архитектурное состояние и содержимое памяти, от которых зависит вызывающая функция.
В частности, вызываемая функция должна оставить неизменным содержимое сохраняемых регистров ***$s0–$s7***, регистра ***$ra*** и ***стека*** – участка памяти, используемого для хранения временных переменных.

Архитектура MIPS использует инструкцию безусловного перехода с возвратом (***jal***) для вызова функции и инструкцию безусловного перехода по регистру (***jr***) для возврата из функции.

Код на C

```
int main() {
        simple();
…
}
// void means the function returns no value
void simple() {
        return;
}
```

Код на языке ассемблера MIPS

```
0x00400200  main:    jal simple      # call function
0x00400204           …
0x00401020  simple:  jr $ra          # return
```

Инструкция ***jal*** выполняет две операции:
сохраняет адрес **следующей** за ней инструкции в регистре адреса возврата (***$ra***) и выполняет переход по адресу первой инструкции вызываемой функции.

Функция simple немедленно завершается, выполняя инструкцию ***jr $ra***, то есть осуществляет переход к инструкции по адресу, находящемуся в регистре ***$ra***.
После этого функция *main* продолжает выполняться с этого адреса (0x00400204).

Код на C

```
int main()      #
{
int y;
…   y = diffofsums(2, 3, 4, 5);   …
}
int diffofsums(int f, int g, int h, int i)
{
      int result;
      result = (f + g) − (h + i);
      return result;
}
```

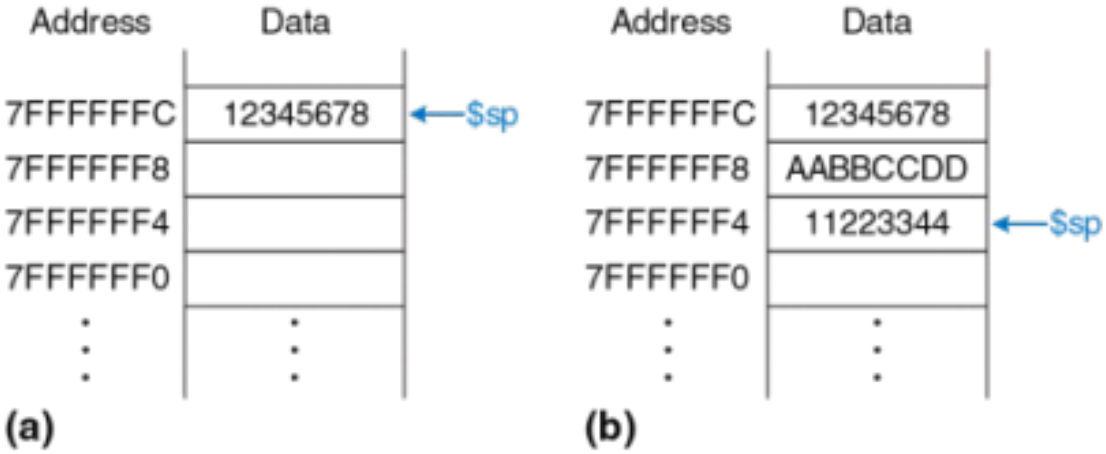
Код на языке ассемблера MIPS

```
# $s0 = y
main:
…
addi $a0, $0, 2      # argument 0 = 2        //   addi $a1, $0, 3      # argument 1 = 3
addi $a2, $0, 4      # argument 2 = 4        //   addi $a3, $0, 5      # argument 3 = 5
jal diffofsums       # call function
add $s0, $v0, $0     # y = returned value
…
# $s0 = result
diffofsums:
add $t0, $a0, $a1    # $t0 = f + g           //   add $t1, $a2, $a3    # $t1 = h + i
sub $s0, $t0, $t1    # result = (f + g) − (h + i)
add $v0, $s0, $0     # put return value in $v0
jr $ra               # return to caller
```

Стек (*stack*) – участок памяти для хранения локальных переменных функции (или других временных переменных).
Стек *расширяется* (занимает больше памяти), если нужно больше места, и *сужается* (занимает меньше памяти), если сохранённые там переменные больше не нужны.

Стек является очередью, работающей в режиме «последним пришёл – первым ушёл» (*LIFO, last-in-first-out*).
Последний элемент, помещенный (*push*) на стек, будет первым элементом, который с него снимут (извлекут, *pop*).
Каждая функция может выделить память на стеке для хранения локальных переменных, и она же должна освободить её перед возвратом.
Верхушка стека (*top of the stack*) – это память, которая была выделена последней.
Стек расширяется в сторону **младших адресов**.

Регистр указателя стека (*$sp, stack pointer*) – специальный регистр, который указывает на верхушку стека. Регистр *$sp* указывает на верхушку стека – наименьший адрес памяти, доступной на стеке.



Одно из важных применений стека – сохранение и восстановление значений регистров, используемых внутри функции.

В нашем примере регистры *$s0, $t0, $t1* меняются – *побочный эффект.*

Чтобы предотвратить его, функция сохраняет значения регистров на стеке перед тем, как изменить их, и восстанавливает их из стека перед тем, как завершиться.
Шаги:
1. Выделяет пространство на стеке для сохранения значений одного или нескольких регистров.
2. Сохраняет значения регистров на стек.
3. Выполняет функцию, используя регистры.
4. Восстанавливает исходные значения регистров из стека.
5. Освобождает пространство на стеке.

```
# $s0 = result
diffofsums:
addi $sp, $sp, −12  # make space on stack to store three registers
sw $s0, 8($sp)      # save $s0 on stack
sw $t0, 4($sp)      # save $t0 on stack
sw $t1, 0($sp)      # save $t1 on stack
add $t0, $a0, $a1   # $t0 = f + g
add $t1, $a2, $a3   # $t1 = h + i
sub $s0, $t0, $t1   # result = (f + g) − (h + i)
add $v0, $s0, $0    # put return value in $v0
lw $t1, 0($sp)      # restore $t1 from stack
lw $t0, 4($sp)      # restore $t0 from stack
lw $s0, 8($sp)      # restore $s0 from stack
addi $sp, $sp, 12   # deallocate stack space
jr $ra              # return to caller
```

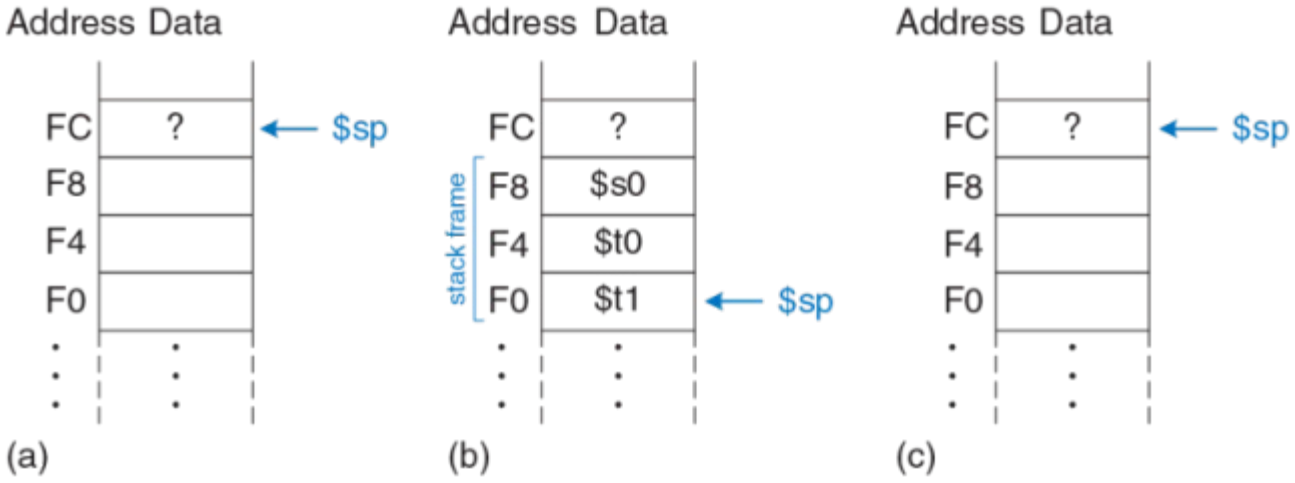


Рис. 6.25 Стек до (a), во время (b) и после (c) вызова функции `diffofsums`

В архитектуре MIPS регистры разделены на две категории:
оберегаемые (*preserved*) и
необерегаемые (*nonpreserved*).
Оберегаемые регистры – *$s0–$s7* (сохраняемые, *saved*).
Необерегаемые регистры – *$t0–$t9* (временные, *temporary*).

Функция должна сохранять и восстанавливать любые оберегаемые регистры, с которыми она собирается работать, но может свободно менять значения необерегаемых регистров.

```
# $s0 = result
diffofsums:
addi $sp, $sp, −4   # make space on stack to store one register
sw $s0, 0($sp)      # save $s0 on stack
add $t0, $a0, $a1   # $t0 = f + g
add $t1, $a2, $a3   # $t1 = h + i
sub $s0, $t0, $t1   # result = (f + g) − (h + i)
add $v0, $s0, $0    # put return value in $v0
lw $s0, 0($sp)      # restore $s0 from stack
addi $sp, $sp, 4    # deallocate stack space
jr $ra              # return to caller
```

Оберегаемые регистры называют *сохраняемыми вызываемой функцией*,
а необерегаемые регистры называют *сохраняемыми вызывающей функцией*.

Табл. 6.3 Оберегаемые и необерегаемые регистры

Оберегаемые	Необерегаемые
Сохраняемые регистры: `$s0–$s7`	Временные регистры: `$t0–$t9`
Адрес возврата: `$ra`	Регистры аргументов: `$a0–$a3`
Указатель стека: `$sp`	Возвращаемые значения: `$v0–$v1`
Содержимое стека выше указателя стека	Стек ниже указателя стека

В архитектуре MIPS регистры разделены на две категории:
оберегаемые (*preserved*) и
необерегаемые (*nonpreserved*).
Оберегаемые регистры – *$s0–$s7* (сохраняемые, *saved*).
Необерегаемые регистры – *$t0–$t9* (временные, *temporary*).

У функций могут быть более четырёх аргументов, и локальные переменные.
Стек используют для хранения этих временных значений.
MIPS: если у функции больше четырёх аргументов, то первые четыре из них, как обычно, передаются через регистры для аргументов.
Дополнительные аргументы передаются на стеке, прямо над указателем стека *$sp*.
Вызывающая функция должна расширить стек для дополнительных аргументов.

Локальные переменные функции хранятся в регистрах *$s0–$s7*.
Если их много, или есть массивы, их можно хранить в кадре стека функции.

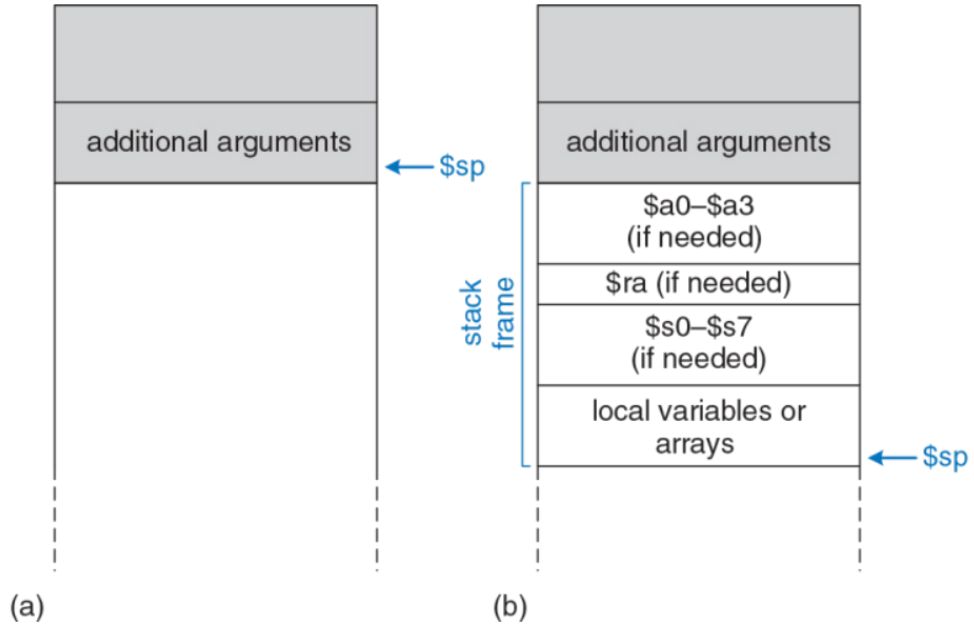


Рис. 6.27 Использование стека перед вызовом (a) и после вызова (b)

1) Регистровая адресация (*register-only*).
Регистры используются для всех операндов и результата.
Инструкции типа **R**.
2) Непосредственная адресация (*immediate*).
Операнды – регистры и 16-битные константы.
3) Базовая адресация (*base*).
Эффективный адрес операнда в памяти вычисляется путем сложения базового адреса в регистре и 16-битного смещения с расширенным знаком, являющегося непосредственным операндом.
Инструкции для доступа в память (*lw, sw*).
4) Адресация относительно счетчика команд (*PC-relative*).
Используется инструкциями условного перехода, или ветвления. Смещение со знаком прибавляется к счетчику команд (PC) для определения нового значения PC.

Целевой адрес ветвления (*branch target address, BTA*) – адрес инструкции, которая будет выполнена следующей в том случае, если случится ветвление.
16-битный непосредственный операнд задаёт число инструкций между целевым адресом ветвления и инструкцией, находящейся *сразу после инструкции перехода* (т.е. инструкции по адресу PC + 4).
Процессор вычисляет целевой адрес ветвления, выполняя знаковое расширение 16-битного непосредственного операнда до 32 бит и умножая полученное значение на 4 (чтобы преобразовать слова в байты), после чего добавляя это произведение к значению PC + 4.

5) Псевдопрямая адресация.
При прямой адресации адрес перехода задаётся внутри инструкции.
Инструкции безусловного перехода *j* и *jal* могли бы использовать прямую адресацию для определения 32-битного целевого адреса перехода (jump target address, JTA).
Но бит не хватает – всего 26 вместо 32.

Таким образом:
JTA (1:0) = 0
JTA (27:2) – берутся из поля адреса инструкции
JTA(31:28) – четыре старших бита адреса перехода берутся из четырёх старших бит значения PC+4.

Длина такого перехода ограничена.
Все инструкции типа **J** (*j* и *jal*) используют псевдопрямую адресацию.

Инструкции безусловного перехода по регистру (*jr* и *jalr*) **НЕ** являются инструкциями типа **J**.
Они являются инструкциями типа **R** и выполняют переход по 32-битному адресу.

Карта памяти:

1) Сегмент кода (text segment) содержит машинные команды исполняемой программы. До 256 Мбайт кода. Четыре старших бита адреса в сегменте кода всегда равны нулю, что позволяет использовать инструкцию *j* для перехода по любому адресу в программе.

2) Сегмент глобальных данных (global data segment) содержит глобальные переменные, которые находятся в области видимости всех функций программы. До 64 Кбайт. Глобальные переменные инициализируются при загрузке программы, до начала ее выполнения.

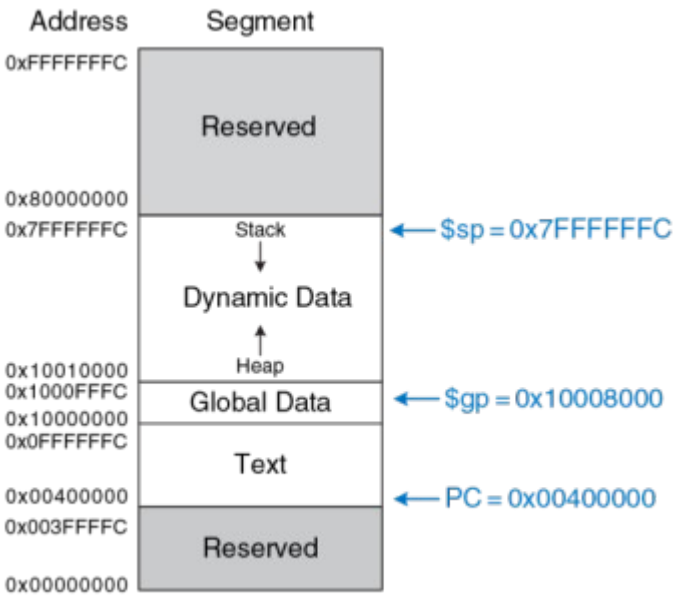


Рис. 6.31 Карта памяти MIPS

Доступ к глобальным переменным осуществляется при помощи глобального указателя (*$gp*), который инициализируется значением *0x100080000*.
$gp не меняется во время выполнения программы.
Любая глобальная переменная доступна при помощи 16-битного положительного или отрицательного смещения относительно *$gp*.

Карта памяти:

3) Сегмент динамических данных (dynamic data segment) содержит стек и кучу. В момент запуска программы этот сегмент не содержит данных, они динамически выделяются и освобождаются в процессе выполнения программы. До 2 Гбайт.

Зарезервированный сегмент используется операционной системой и не может использоваться программой. Часть зарезервированной памяти используется для прерываний и для отображения устройств ввода-вывода.

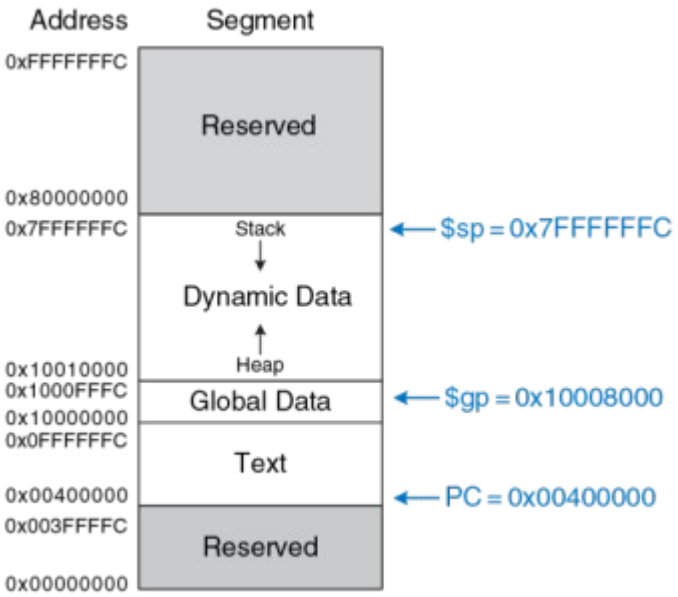
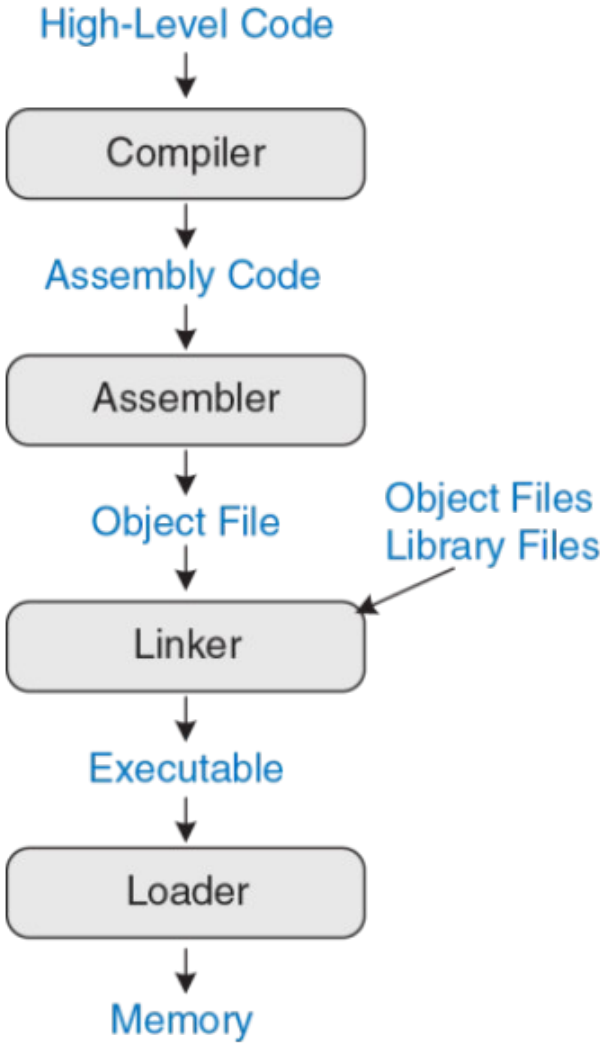


Рис. 6.31 **Карта памяти MIPS**



1) Компиляция.
2) Трансляция. 2 прохода.
3) Компоновка.

4) Загрузка:

загрузка кода программы в память

$gp = 0x10008000

$sp = 0x7FFFFFFC

jal 0x00400000

```
int f, g, y; // global variables
int main(void)
{
      f = 2;      g = 3;      y = sum(f, g);        return y;
}
int sum(int a, int b) {      return (a + b);}
```

Код на языке ассемблера MIPS

```
.data
f:     //     g:     //     y:
.text
main:
      addi $sp, $sp, -4     # make stack frame
      sw $ra, 0($sp)        # store $ra on stack
      addi $a0, $0, 2       # $a0 = 2
      sw $a0, f             # f = 2
      addi $a1, $0, 3       # $a1 = 3
      sw $a1, g             # g = 3
      jal sum               # call sum function
      sw $v0, y             # y = sum(f, g)
      lw $ra, 0($sp)        # restore $ra from stack
      addi $sp, $sp, 4      # restore stack pointer
      jr $ra                # return to operating system
sum:
      add $v0, $a0, $a1     # $v0 = a + b
      jr $ra                # return to caller
```

Первый проход – назначаются адреса командам и идентифицируются символы (в таблицу).

```
0x00400000 main:  addi $sp, $sp, –4
0x00400004        sw $ra, 0($sp)
0x00400008        addi $a0, $0, 2
0x0040000C        sw $a0, f
0x00400010        addi $a1, $0, 3
0x00400014        sw $a1, g
0x00400018        jal sum
0x0040001C        sw $v0, y
0x00400020        lw $ra, 0($sp)
0x00400024        addi $sp, $sp, 4
0x00400028        jr $ra
0x0040002C sum:   add $v0, $a0, $a1
0x00400030        jr $ra
```

Табл. 6.4 Таблица символов

Символ	Адрес
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Второй проход – машинный код.