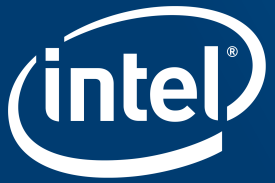


Codegen.

Горбань Игорь

20 марта 2018

g.i.b@list.ru

Agreement

Stack machine with register (accumulator) - не гарантирует эффективного кода, но достаточно проста для реализации (и более эффективна чем pure stack machine).

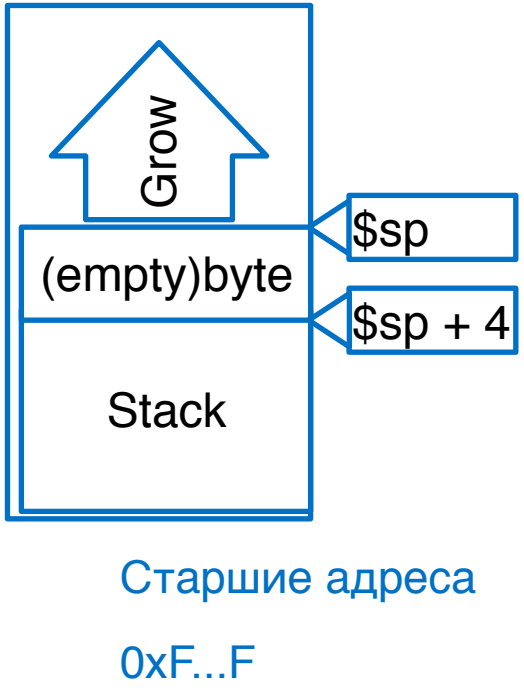
Целевая архитектура - mips. Для проверки будем использовать симулятор spim.

Accumulator на mips - будем использовать регистр $a0.

Stack растет в направлении к меньшим адресам (стандарт mips)

Адрес стека будет храниться в регистре $sp (стандарт mips).

Верхушка стека - ($sp + 4). Т.е. $sp будет указывать на место в памяти, для начала записи следующего frame (Activation record).



MIPS

1. Прототип Reduced Instruction Set Computes (RISC) - сокращенный набор инструкций.
2. Большинство инструкций используют регистры для операндов и результатов.
3. Имеет 32 регистра общего назначения по 32 бита каждый
4. Использует load/store инструкции для чтения/записи данных из памяти

В codegen используется 3 регистра:

- $sp - по назначению (stack pointer)
- $a0 - как accumulator (результат вычислений)
- $t1 - как временный регистр

Instructions #1

$lw\ reg_1\ offet(\ reg_2\ )$	Загрузить слово 32-битного размера из адреса (*reg_2 + offset*) в *reg_1*.

Offset - immediate (является константой). Lw - load word.

$sw\ reg_1\ offet(\ reg_2\ )$	Сохранить слово 32-битного размера из *reg_1* по адресу (*reg_2 + offset*).

Offset - immediate. Sw - store word.

$add\ reg_1\ reg_2\ reg_3$	Сложить *reg_2* и *reg_3*. Положить результат в *reg_1* .

Instructions #2

$addiu\ reg_1\ reg_2\ imm$	Сложить *reg_2* и *imm* и положить результат в *reg_1*

unsigned immediate (константа без знака). Переполнение не проверяется.

$li\ reg\ imm$	Копирует значение imm в reg

Load immediate

Example for stack-machine.

Stack-Machine:	MIPS:			
acc <- 7	*li*	$a0	7	
push acc	*sw*	$a0	0($sp)	
	addiu	$sp	$sp -4	<- *(0xfffffffc)*
acc <- 5	*li*	$a0	5	
acc <- acc + top_of_stack	*lw*	$t1	4($sp)	
	add	$a0	$a0 $t1	
pop	*addiu*	$sp	$sp 4	

Codegen example

Предположим, что мы разрабатываем функцию ***cgen(node)*** - которая записывает на стандартный поток вывода (stdout) ассемблер mips. При этом выполняются следующие требования:

1. Значение Expression - сохраняется в $a0

2. Сохраняется $sp и данные в стеке до и после вызова cgen(expr)

Пусть expression - immediate, то внутри cgen обработка могла быть:

```
if (node->class == int_const_class)
    std::cout << " li $a0  0x" << std::hex << node->value << std::endl;
```

Codegen example 2 : add

Пусть expression - (*e1 + e2*), причем e1 и e2 - тоже expression:

```
if (node->class == plus_class)
{
        cgen(node->e1);
        std::cout <<      "  sw          $a0  0($sp) \n"
                          "  addiu       $sp  $sp  -4 \n";
        cgen(node->e2);
        std::cout <<      "  lw          $t1  4($sp) \n"
                          "  addiu       $sp  $sp  4 \n"
                          "  add         $a0  $t1  $a0 \n";
}
```

Оптимизация: сохранить значение cgen(e1) - во временной переменной $t1

кладем в стек

берем из стека

вот оно - сложение

Instructions #3

- Код для "+" - шаблон с дырами для заполнения e1 и e2
- Код для stack-machine - генерится рекурентно

=> Генерация кода может быть описана как рекурсивный обход AST (как минимум для expressions)

$sub\ reg_1\ reg_2\ reg_3$	Вычесть *reg_3* из *reg_2* и положить результат в *reg_1*
$beq\ reg_1\ reg_2\ label$	Прыжок на метку *label* если *reg_1 == reg_2*
$b\ \ label$	Прыжок на метку *label*

Codegen example 3 : cond #1

Пусть expression - (*if e1 == e2 then e3 else e4*), e* - expression:

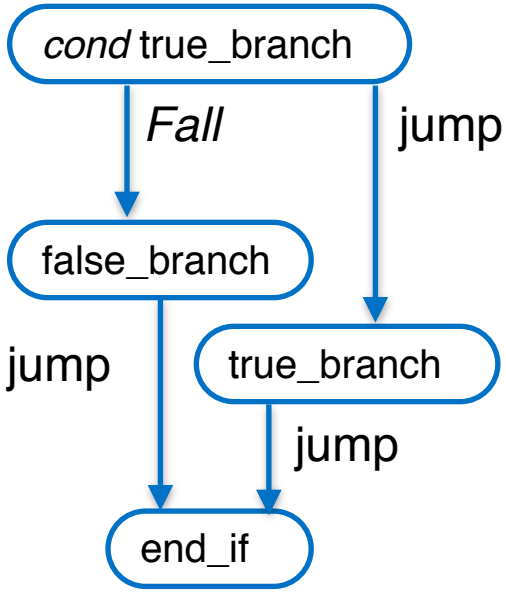
```
if (node->class == cond_class)
{
        cgen(node->e1);
        std::cout <<      "  sw          $a0  0($sp) \n"
                          "  addiu       $sp  $sp  -4 \n";
        cgen(node->e2);
        std::cout <<      "  lw          $t1  4($sp) \n"
                          "  addiu       $sp  $sp  4 \n"
                          "  beq         $a0  $t1  true_branch" << node->get_id() << "\n";
...
```

абсолютно идентично со сложением

Codegen example 3 : cond #2

```
    std::cout <<      "false_branch" <<  node->get_id() << ": \n";
    cgen(node->e4);
    std::cout <<      "  b end_if" << node->get_id() << "\n"
                      "true_branch" << node->get_id() << ":\n";
    cgen(node->e3);
    std::cout <<      "end_if" << node->get_id() << ": \n";
}
```

"Пустая" метка. Переход произойдет на следующую инструкцию за ней.



Стоит заметить, что условия выполнены - результат будет записан в $a0

Function calls + function definitions

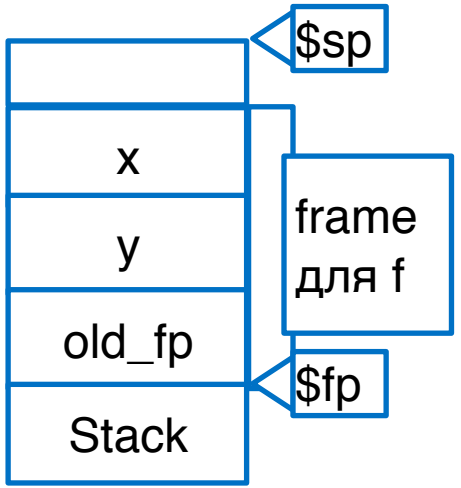
Их внешний вид сильно зависит от структуры frames (Activation record)

- Frame хранит параметры для f(x1, ... , xn) на стеке
- Нет необходимости хранить результат - он записывается в accumulator

Предположим, соглашение гарантирует, что при выходе из функции $sp - такой же, что и перед входом - тогда нет необходимости в указателе на предыдущий фрейм.

Однако при этом нужно сохранять адрес возврата. Тогда, давайте хранить указатель на текущий frame(AR) в регистре ***$fp***.

Для такой модели frame(AR) при вызове f(x, y) будет выглядеть так :



Codegen example 4 : dispatch

Рассмотрим cgen(node), если node - вызов функции (*f(e1, ... , en)*):

```
if (node->class == dispatch) {
        std::cout <<      "  sw          $fp   0($sp) \n"
                          "  addiu       $sp   $sp  -4 \n";
        cgen(node->en);
        std::cout <<      "  sw          $a0   0($sp) \n"
                          "  addiu       $sp   $sp  -4 \n";
...
        cgen(node->e1);
        std::cout <<      "  sw          $a0   0($sp) \n"
                          "  addiu       $sp   $sp  -4 \n"
                          "  jal       " << node->f().name <<  "_entry \n";
};
```

old_fp

en

...

e1

jump

Сохранение адреса старого фрейма

4 байта

Сохранение аргументов в обратном порядке

(4 * n) байтов

Адрес возврата из ***f*** будет находится в регистре $ra

Instructions #4

$jal \;\; label$	Прыгает на метку *label*, сохраняя адрес текущей инструкции в ***$ra*** .

В других архитектурах подобные инструкции сохраняют адрес возврата на stack и зовется ***call.***

$jr \;\; reg$	Прыгает по адресу, записанному в регистр ***reg***

$move \;\; reg_1 \;\; reg_2$	Копирует значение регистра ***reg2*** в ***reg1***

Codegen example 4 : method

Рассмотрим cgen(node), если node - функция (*def f(x1, ... , xn) = body*) :

```
if (node->class == method_class) {
    std::cout <<    node->f().name  << "_entry:  \n"
                    "  move         $fp   $sp \n"
                    "  sw           $ra   0($sp) \n"
                    "  addiu        $sp   $sp  -4 \n";
    cgen(node->body);
    std::cout <<    "  lw           $ra   4($sp) \n"
                    "  addiu        $sp   $sp  " << node->stack_size() << " \n"
                    "  lw           $fp   0($sp)  -4 \n"
                    "  jr           $ra  \n";
```

save ret_addr

load ret_addr

pop stack

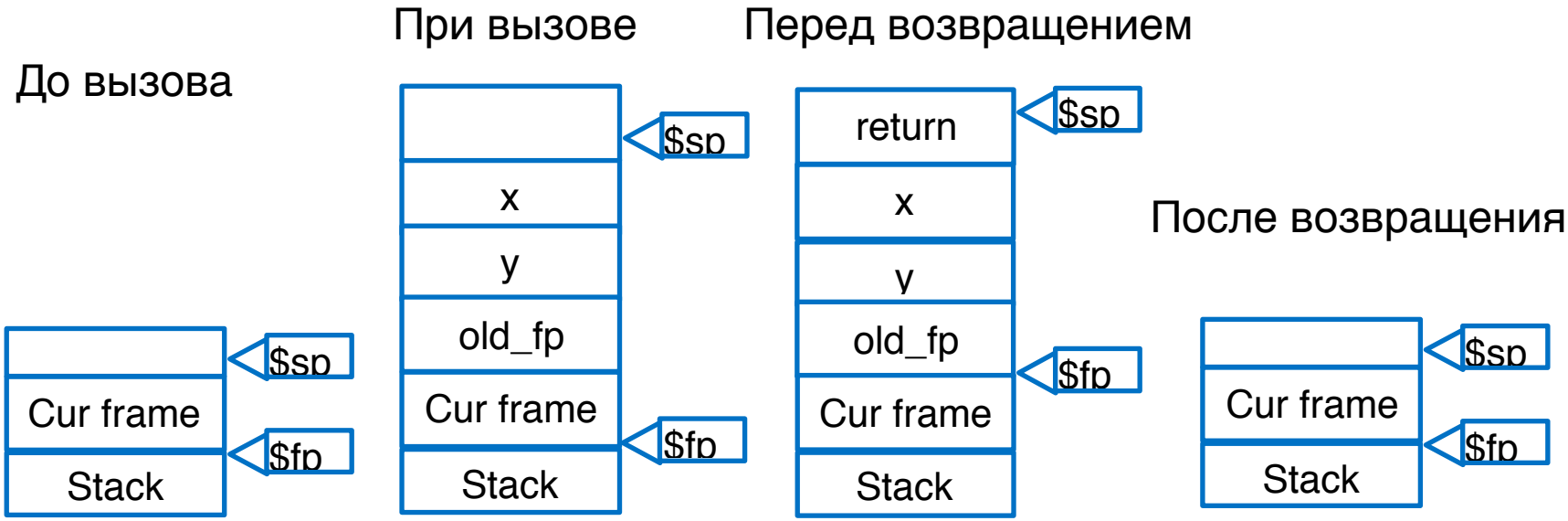
4 * n	+ 4	+ 4
args	+ ret_addr	+ old_fp

jump back

Call stack

$fp - указывает на начало frame.

Перед вызовом сохраняется frame pointer и аргументы функции.



Task

```
li $a0 5
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 4
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 3
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

5 + (4 - 3)

5 - (4 + 3)

(5 + 4) - 3

(5 - 4) + 3