Weekly assignment 4

Troels Thomsen - qvw203

15. December 2013

1 Week assignment 3

1.1 Exercise 1

1.1.1 1.a

In order to translate $gcd(x + y, y + 1) * 2$ into intermediate code, we need to first translate the inner expressions.
We need to place the final result in t0 and assume that

$$vtable = [x- > v, y- > w]$$
$$ftable = [gcd- > \_GCD\_]$$

In the following intermediate code, t1 corresponds to the constant 2, t2 to the constant 1, t3 to x, t4 to y, t5 to x+y and t6 to y+1. Finally t7 will contain the call to gcd(t5, t6), and t0 will contain t6*2.

$$\begin{aligned} t1 &:= 2 \\ t2 &:= 1 \\ t3 &:= v \\ t4 &:= w \\ t5 &:= v + w \\ t6 &:= w + t2 \\ t7 &:= CALL\ \_GCD\_(t5, t6) \\ t0 &:= t7 * t1 \end{aligned}$$

1.1.2 1.b

First we translate the following into intermediate code, then into MIPS assembler code.

```
while (b != 0) and (a/b != 0) {
    if b < a then {
        a := a - b
    }
    else {
        b := b - a
    }
}
```

The intermediate code:

```
t1 := 0
t2 := v
t3 := w
LABEL loop
    t4 := t3 != t1
    t5 := t2 / t3
    t6 := t5 != t1
    IF t4 AND t6 THEN 1t ELSE 1f
LABEL 1t
    IF t3 < t2 THEN 2t ELSE 2f
LABEL 2t
    t2 := t2 - t3
    GOTO loop
LABEL 2f
    t3 := t3 - t2
    GOTO loop
LABEL 1f
```

Then the MIPS code:

```
# we put our vtable values into registers.
add $t2, v, $zero
add $t3, w, $zero
# the loop label calculates the while condition.
loop:
    nand $t4, $t3, $zero
    div $t5, $t2, $t3
    nand $t6, $t5, $zero
    andi $t0, $t4, 1
    andi $t1, $t6, 1
    beq $t0, $t1, 1t
# 1f jumps to end
1f:
    j end
# 1t calculates the if-statement
# and jumps to 2t if it evaluates to true.
1t:
    slt $t7, $t3, $t2
    bne $t7, $zero, t2
# 2f is the body of the else
2f:
    sub $t3, $t3, $t2
    j loop
# 2t is the body of the if
2t:
    sub $t2, $t2, $t3
    j loop
# end terminates the program or continues to next code.
end:
```

1.1.3 1.c

Both of these instructions can be seen as if-then-else blocks.
We can write the pattern replacement for (i) and (ii) as follows:

(i)	`IF rs = rt THEN labelt ELSE labelf`
	`LABEL labelf`
		`beq rs,rt labelt`
		`labelf:`
		`  addi rd, $zero, 0`
		`  j end`
		`labelt:`
		`  addi rd, $zero, 1`
		`end:`

(ii)	`IF r = 0 THEN labelt ELSE labelf`
	`LABEL labelf`
		`beq rd,$zero labelt`
		`labelf:`
		`  addi rd, $zero, 0`
		`  j end`
		`labelt`
		`  addi rd, $zero, 1`
		`end:`

1.2 Excercise 2

1.2.1 2.a

Since we only need to define map for a two-dimentional array in this example, and we do not know the syntax to give map a function as an argument, map can be defined as follows:

```
function f (x : int) : char
begin
    return chr(x);
end;

function map(x : array of array of int) : array of array of
   char
var
    y : array of array of char;
    i : int;
    ii : int;
    lenD1 : int;
    lenD2 : int;
begin
    lenD1 := len(0,x);
    lenD2 := len(1,x);
    y := new(lenD1, lenD2);
    i := 0;
    ii := 0;
    while (i < lenD1) do
        begin
            while (ii < lenD2) do
                begin
                    y[i, ii] := f(x[i, ii]);
                    ii := ii+1;
                end;
            i := i+1;
        end;
    return y;
end;
```

1.2.2 2.b

```
.data
    intArray: .word 1, 2, 3, 4
    charArray: .word 0, 0, 0, 0 # no data yet
    len: 4 # length of array
.text

main:
    la $s0, intArray # address of array
    la $s1, charArray # address of array
    la $v1, len
    lw $v1, 0($v1)
    addi $v0, $zero, $zero

map:
    slt $t0, $v0, $v1
    beq $t0, 1, loop
    j end

loop: # loop body
    lw $v2, 0($s0) # load current array index
    addi $s0, $s0, 4 # shift the address to next index
       address
    CALL(f, $v2) # store the char representation in $v2
    sw $v2, 0($s1)
    addi $s1, $s1, 4 # shift the address to next index
       address
    addi $v0, $v0, 1 # increase the iteration counter
    j map

end:
    # end program
```

1.2.3 2.c

The most efficient of these two MIPS translations, will be the one created in section 2.b. The paladim compiler generates almost 900 lines of MIPS assembler code, while we in our own implementation is only 29 lines of code. But the most important factor, is that our own implementation run in linear time complexity, since we only iterate the length of the array. The paladim compiler will not do this optimization, and will just create two loops.