Digital Systems (Sheet #3)

Marius Gavrilescu

8.2 As suggested in the lecture notes, we will first prove `foldl step (store, stack) (compileS expr) = (store, eval store expr : stack)` using induction on the structure of expr.

We have three cases for expr:

If it is a number,

```
foldl step (store, stack) (compileS (Num x)) =
foldl step (store, stack) [PushN x] =
step (store, stack) (PushN x) =
(store, n : stack) =
(store, eval store (Num x) : stack)
```

A similar argument applies if expr is a variable.

If expr is the application of a binary operator,

```
foldl step (store, stack) (compileS (Bin l op r)) =
foldl step (store, stack) (compileS l ++ compileS r ++ [Do op]) =
foldl step (foldl step (foldl step (store, stack) l) r) [Do op] =
foldl step (foldl step (store, eval store l : stack) r) [Do op] =
foldl step (store, eval store r : eval store l : stack) [Do op] =
step (store, eval store r : eval store l : stack) (Do op) =
(store, operate (eval store l) (eval store r) : stack) =
(store, eval store (Bin l op r) : stack)
```

We can now use this hypothesis to prove correctness:

```
exec store (compileS expr) =
answer (foldl step (store, []) (compileS expr)) =
answer (store, eval store expr : []) =
eval store expr
```

8.3
```
compileS (Num n) = [PushN n]
compileS (Var v) = [PushV v]
compileS (Bin l op r)
  | (stackuse a) >= (stackuse b) = a ++ b ++ [Do op]
  | otherwise = b ++ a ++ [Swap, Do op]
  where a = compileS l
        b = compileS r

stackuse :: [Instr] -> Int
stackuse xs = maximum (scanl use 0 xs)

use :: Int -> Instr -> Int
use x Swap   = x
use x (Do _) = x - 1
use x _      = x + 1
```

A swap instruction makes the implementation simpler but less efficient, reversed versions of instructions make the implementation more complicated but also more efficient.

9.5
```
move $5, $3      is add $5, $3, $0
clear $5         is add $5, $0, $0
li $5, small     is addi $5, $0, small
li $5, big       is lui $5, big[31:16]; addi $5, $5, big[15:0]
lw $5, big($3)   is li $at, big;        add $at, $at, $3; lw $5, 0($at)
addi $5, $3, big is li $at, big;        add $5, $3, $at
beq $5, small, L is li $at, small;      beq $5, $at, L
beq $5, big, L   is li $at, big;        beq $5, $at, L
```

9.6
```
bz $5, L      is beq $5, $0, L
bnez $5, L    is bne $5, $0, L
blt $5, $3, L is sub $at, $5, $3; bltz $at, L;
ble $5, $3, L is sub $at, $5, $3; blez $at, L;
bgt $5, $3, L is sub $at, $5, $3; bgtz $at, L;
bge $5, $3, L is sub $at, $5, $3; bgez $at, L;
```

9.8
```
compileR :: Env -> [Reg] -> Expr -> [String]
compileR env (r : free) (Num n) = ["li " ++ r ++ " " ++ n]
compileR env (r : free) (Var v) = ["lw " ++ r ++ " 0(" ++ (env `at` v) ++ ")"]
compileR env (r0 : r1 : free) (Bin l op r)
                            = compileR env (r0 : r1 : free) l ++
                              compileR env (r1 : free) r ++
                              [opstr op ++ " " ++ r0 ++ " " ++ r0 ++ " " ++ r1]

opstr Add = "add"
opstr Sub = "sub"
opstr And = "and"
opstr Or  = "or"
-- etc
```

10.2
```
r := 0;
WHILE b # 0 DO
  IF b MOD 2 = 1 THEN r := r + a END;
  b := b DIV 2;
  a := a + a;
END;
```

```
mul:    clr $v0
loop:   bz $a1, end
        andi $t0, $a1, 1
        bz $t0, skip
        add $v0, $v0, $a0
skip:   srl $a1, $a1, 1
        add $a0, $a0, $a0
        b loop
end:    jr $ra
```

10.3
```
        clr $2
        clr $3
        clr $5
loop:   beq $2, $4, exit
        lw $6, a($5)
        add $3, $3, $6
        addi $2, $2, 1
        addi $5, $5, 4
        b loop
exit:
```

10.7 The subroutine for mul was already written above.

```
fact: subi $sp, $sp, 8
sw $ra, 4($sp)
sw $a0, 0($sp)
bne $a0, 0, else
then: li $v0, 1
addi $sp, $sp, 8
jr $ra
else: subi $a0, $a0, 1
jal fact
lw $a0, 0($sp)
move $a1, $v0
jal mul
lw $ra, 4($sp)
addi $sp, $sp, 8
jr $ra
```