Introduction to Computer Science
Constructor University
Dr. Jürgen Schönwälder

Module: CH-232
Date: 2023-11-17
Due: 2023-11-24

Problem Sheet #11

Problem 11.1: *integer multiplication in risc-v rv32i assembler* (2+1+1 = 4 points)

The 32-bit RISC-V base integer instruction set (rv32i) does not support multiplication and division operations. To deal with this, a compiler may call a function when a multiplication is needed. For example, `gcc` expects that a function `__mulsi3(unsigned int a, unsigned int b)` is provided to multiply two integers.

A multiplication can be carried out by repeated additions and shifts as shown in algorithm 1:

Algorithm 1 Integer multiplication using additions and shifts

Require: $a, b \in \mathbb{N}$
$r \leftarrow 0$ ▷ initialize the result r to be zero
while $a \neq 0$ **do**
  if a is odd **then** ▷ the lowest order bit of a is 1
    $r \leftarrow r + b$
  end if
  $a \leftarrow a \gg 1$ ▷ right bit shift a by 1 position
  $b \leftarrow b \ll 1$ ▷ left bit shift b by 1 position
end while
Ensure: $r = a \cdot b$

a) Translate algorithm 1 into RISC-V rv32i assembler code. Comment the assembler code to explain how the calculation proceeds. Note that the arguments are passed via the registers a0 (x10) and a1 (x11) and that the result is returned in a0 (x10).

b) Explain the concept of a function prologue and a function epilogue in your own words.

c) Does the function `__mulsi3(unsigned int a, unsigned int b)` need a function prologue and a function epilogue? Explain why or why not.

You are welcome to use emulsiV to develop and test your assembler code.

Problem 11.2: *integer expression rendering (haskell)* (2 points)

We can represent integer expressions in Haskell using the following data type:

```haskell
module IntExp where

data Exp = C Int        -- a constant integer
         | V String     -- a variable with a name
         | S Exp Exp    -- a sum of two expressions
         | P Exp Exp    -- a product of two expressions
         deriving (Show, Eq)
```

Constants and variables are primitive expressions. More complex expressions can be constructed by forming the sum of two expressions or the product of two expressions.

Implement a function `render :: Exp -> Spring` rendering an expression into a textual (infix) notation.

Submit your Haskell code plus an explanation (in Haskell comments) as a plain text file. Below is a collection of test cases.

```haskell
module IntExpRenderTest (main) where

import Test.HUnit
import IntExp
import IntExpRender

tests = TestList
  [ render (C 42) ~?= "42"
  , render (V "foo") ~?= "foo"
  , render (S (C 0) (C 1)) ~?= "(0 + 1)"
  , render (P (C 0) (C 1)) ~?= "(0 * 1)"
  , render (P (S (C 0) (C 1)) (S (C 42) (V "foo"))) ~?= "((0 + 1) * (42 + foo))"
  ]

main :: IO Counts
main = runTestTT tests
```

Problem 11.3: *integer expression simplification (haskell)* (2+2 = 4 points)

We can represent integer expressions in Haskell using the following data type:

```haskell
module IntExp where

data Exp = C Int          -- a constant integer
         | V String       -- a variable with a name
         | S Exp Exp      -- a sum of two expressions
         | P Exp Exp      -- a product of two expressions
         deriving (Show, Eq)
```

Constants and variables are primitive expressions. More complex expressions can be constructed by forming the sum of two expressions or the product of two expressions.

Our goal is to simplify expressions whenever possible. For example, we want to write a function that can simplify `(2 * y) * (3 + (2 * 2))` to `14 * y`. We use the following rules to simplify expressions:

- S1 Adding two constants a and b yields a constant, which has the value $a + b$, e.g., $3 + 5 = 8$.
- S2 Adding 0 to a variable yields the variable, i.e., $0 + x = x$ and $x + 0 = x$.
- S3 Adding a constant a to a sum consisting of a constant b and a variable yields the sum of the $a + b$ and the variable, e.g., $3 + (5 + y) = 8 + y$.
- P1 Multiplying two constants a and b yields a constant, which as the value $a \cdot b$, e.g., $3 \cdot 5 = 15$.
- P2 Multiplying a variable with 1 yields the variable, i.e., $1 \cdot y = y$ and $y \cdot 1 = y$.
- P3 Multiplying a variable with 0 yields the constant 0, i.e., $0 \cdot y = 0$ and $y \cdot 0 = 0$.
- P4 Multiplying a constant a with a product consisting of a constant b and a variable yields the product of $a \cdot b$ and the variable, e.g., $3 \cdot (2 \cdot y) = 6 \cdot y$.

The usual associativity rules apply. Note that we have left out distributivity rules.

a) Implement a function `simplify :: Exp -> Exp` simplifying an expression that does not contain variables. In other words, `simplify` returns the (constant) value of an expression that does not contain any variables.

b) Extend the function `simplify` to handle variables as described above.

Submit your Haskell code plus an explanation (in Haskell comments) as a plain text file. Below is a template providing a collection of test cases.

```haskell
module IntExpSimplifyTest (main) where

import Test.HUnit
import IntExp
import IntExpSimplify

tI0 = TestList
  [ simplify (C 3) ~?= C 3                    -- 3 = 3
  , simplify (V "y") ~?= V "y"                -- y = y
  ]

tS1 = TestList
  [ simplify (S (C 3) (C 5)) ~?= C 8          -- 3 + 5 = 8
  ]

tS2 = TestList
  [ simplify (S (C 0) (V "y")) ~?= V "y"      -- 0 + y = y
  , simplify (S (V "y") (C 0)) ~?= V "y"      -- y + 0 = y
  ]

tS3 = TestList
  [ simplify (S (S (C 3) (V "y")) (C 5)) ~?= S (C 8) (V "y")    -- (3 + y) + 5) = 8 + y
  , simplify (S (S (V "y") (C 3) ) (C 5)) ~?= S (C 8) (V "y")   -- (y + 3) + 5) = 8 + y
  , simplify (S (C 3) (S (C 5) (V "y"))) ~?= S (C 8) (V "y")    -- 3 + (5 + y) = 8 + y
  , simplify (S (C 3) (S (V "y") (C 5))) ~?= S (C 8) (V "y")    -- 3 + (y + 5) = 8 + y
  ]

tS4 = TestList
  [ simplify (S (S (C 3) (C 5)) (C 8)) ~?= C 16                 -- (3 + 5) + 8 = 16
  , simplify (S (C 3) (S (C 5) (C 8))) ~?= C 16                 -- 3 + (5 + 8) = 16
  , simplify (S (C 5) (V "y")) ~?= S (C 5) (V "y")              -- 5 + y = 5 + y
  , simplify (S (V "y") (C 5)) ~?= S (V "y") (C 5)              -- y + 5 = y + 5
  , simplify (S (V "x") (V "y")) ~?= S (V "x") (V "y")          -- x + y = x + y
  ]

tP1 = TestList
  [ simplify (P (C 3) (C 5)) ~?= C 15         -- 3 * 5 = 15
  ]

tP2 = TestList
  [ simplify (P (C 1) (V "y")) ~?= V "y"      -- 1 * y = y
  , simplify (P (V "y") (C 1)) ~?= V "y"      -- y * 1 = y
  ]

tP3 = TestList
  [ simplify (P (C 0) (V "y")) ~?= C 0        -- 0 * y = 0
  , simplify (P (V "y") (C 0)) ~?= C 0        -- y * 0 = 0
  ]

tP4 = TestList
  [ simplify (P (P (C 3) (V "y")) (C 2)) ~?= P (C 6) (V "y")    -- (3 * y) * 2) = 6 * y
  , simplify (P (P (V "y") (C 3) ) (C 2)) ~?= P (C 6) (V "y")   -- (y * 3) * 2) = 6 * y
  , simplify (P (C 3) (P (C 2) (V "y"))) ~?= P (C 6) (V "y")    -- 3 * (2 * y) = 6 * y
  , simplify (P (C 3) (P (V "y") (C 2))) ~?= P (C 6) (V "y")    -- 3 * (y * 2) = 6 * y
  ]

tP5 = TestList
  [ simplify (P (P (C 3) (C 5)) (C 8)) ~?= C 120                -- (3 * 5) * 8 = 120
  , simplify (P (C 3) (P (C 5) (C 8))) ~?= C 120                -- 3 * (5 * 8) = 120
  , simplify (P (C 5) (V "y")) ~?= P (C 5) (V "y")              -- 5 * y = 5 * y
  , simplify (P (V "y") (C 5)) ~?= P (V "y") (C 5)              -- y * 5 = y * 5
  , simplify (P (V "x") (V "y")) ~?= P (V "x") (V "y")          -- x * y = x * y
  ]

tM0 = TestList [
  -- (2 * y) * (3 + (2 * 2)) = 14 * y
  simplify (P (P (C 2) (V "y")) (S (C 3) (P (C 2) (C 2)))) ~?= P (C 14) (V "y")
  -- x + (1 + -1) = x
```

```haskell
  , simplify (S (V "x") (S (C 1) (C (-1)))) ~?= V "x"
  -- (1 + -1) * x = 0
  , simplify (P (S (C 1) (C (-1))) (V "x")) ~?= C 0
  -- (2 + -1) * x = x
  , simplify (P (S (C 2) (C (-1))) (V "x")) ~?= V "x"
  -- (2 * 2) * (3 + 4) = 28
  , simplify (P (P (C 2) (C 2)) (S (C 3) (C 4))) ~?= C 28
  ]

main = runTestTT $ TestList [tI0, tS1, tS2, tS3, tS4, tP1, tP2, tP3, tP4, tP5, tM0 ]
```