Bluespec-5 Programming Examples

Arvind
Laboratory for Computer Science
M.I.T.

Lecture 21

http://www.csg.lcs.mit.edu/6.827

Quiz

- Determine if a n-bit number contains exactly one “1”.
  - solution will be given at the end of the class
Outline

- Lennart’s problem √
- Instruction Encoding ⇐
  - Pack and Unpack
- Wallace Tree Addition
- Solution to Lennart’s problem

"deriving (Bits)" for algebraic types

```haskell
data T = A (Bit 3) | B (Bit 5) | Ptr (Bit 31)
deriving (Bits)
```

- the canonical "pack" function created by "deriving (Bits)" produces packings as follows:

```
    0 0   0 1   1 1
    ------- ------- -------
      |      |      |      |
      |      |      |      |
      |      |      |      |
      |      |      |      |
      |      |      |      |
      |      |      |      |
      a3     b5     p31
```

"33 bit" encoding!
Explicit pack & unpack

- Explicit "instance" decls. may permit more efficient packing

```haskell
instance Bits T 32 where
  pack (A a3) = 0b00 ++ (zeroExtend a3)
  pack (B b5) = 0b01 ++ (zeroExtend b5)
  pack (Ptr p31) =

  unpack x = if x[31:30] == 0b00 then A x[2:0]
             elseif x[31:30] == 0b01 then B x[4:0]
             else
```

```
<table>
<thead>
<tr>
<th>x[31:30]</th>
<th>A x[2:0]</th>
<th>B x[4:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Instruction Encoding: MIPS

- **Reg-Reg**
  - `Op` | `Rs1` | `Rs2` | `Rd` | `Const` | `Opx`
  - `6` | `5` | `5` | `5` | `5` | `6`

- **Reg-Imm**
  - `Op` | `Rs1` | `Rd` | `Const`

- **Branch**
  - `Op` | `Opx` | `Rs1` | `Const`

- **Jump/Call**
  - `Op` | `Const`
MIPS Instruction Type

```haskell
data Instruction =
  Immediate op :: Op
  rs :: CPUReg
  rt :: CPUReg
  imm :: UInt16
  | Register rs :: CPUReg
  rt :: CPUReg
  rd :: CPUReg
  sa :: UInt5
  funct :: Funct
  | RegImm rs :: CPUReg
  op :: REGIMM
  imm :: UInt16
  | Jump op :: Op
  target :: UInt26
  | Nop

Need to define CPUReg, UInt5, UInt16, UInt26, REGIMM, Op and Funct
```

CPUReg Type: MIPS Instructions

```haskell
data CPUReg =
  Reg0 | Reg1 | Reg2 | Reg3
  | Reg4 | Reg5 | Reg6 | Reg7
  | Reg8 | Reg9 | Reg10 | Reg11
  | Reg12 | Reg13 | Reg14 | Reg15
  | Reg16 | Reg17 | Reg18 | Reg19
  | Reg20 | Reg21 | Reg22 | Reg23
  | Reg24 | Reg25 | Reg26 | Reg27
  | Reg28 | Reg29 | Reg30 | Reg31

deriving (Bits, Eq, Bounded)
```

```haskell
type UInt32 = Bit 32
type UInt26 = Bit 26
type UInt16 = Bit 16
type UInt5  = Bit  5
```
Op Type: MIPS Instructions

data Op = SPECIAL | REGIMM
   | J  | JAL | BEQ | BNE | BLEZ | BGTZ
   | ADDI | ADDIU | SLTI | SLTIU | ANDI | ORI | XORI | LUI
   | C0F0 | C0P1 | C0P2 | C0P9
   | BEQL | BNEI | BLEIL | BGTZL
   | DADD1e | DADDIUe | LDLe | LDRe
   | OP28 | OP29 | OP30 | OP31
   | LB | LH | LWL | LW | LBH | LHU | LWR | LWUe
   | SB | SH | SWL | SW | SDLle | SDRle | SWR | CACHED
   | LL | LWC1 | LWC2 | OP51 | LLDe | LDC1 | LDC2 | LDe
   | SC | SWC1 | SWC2 | OP59 | SCDle | SDC1 | SDC2 | SDe

deriving (Eq, Bits)

Funct Type: MIPS Instructions

data Funct = SLL | F1 | SRL | SRA
   | SLLV | F5 | SRLV | SRAV
   | JR | JALR | F10 | F11
   | SYSCALL | BREAK | F14 | SYNC
   | MFHI | MTHI | MFLO | MTLO
   | DSLLe | F15 | DSLRe | DSRAle
   | MULT | MULTU | DIV | DIVU
   | DMULTe | DMULTUe | DDIVE | DDIVUe
   | ADD | ADDU | SUB | SUBU
   | AND | OR | XOR | NOR
   | F40 | F41 | SLT | SLTU
   | ADDDe | DADDUe | DSWBe | DSWBe
   | TGE | TGEU | TLT | TLTU
   | TEO | F53 | TNE | F55
   | DSLLe | F57 | DSRLe | DSRAle
   | DSSL32e | F61 | DSR32e | DSRA32e

deriving (Bits, Eq)
Funct Type: MIPS Instructions

\[
data \ \text{REGIMM} = \text{BLTZ} \mid \text{BGEZ} \mid \text{BLTZL} \mid \text{BGEZL} \\
| \text{R4} \mid \text{R5} \mid \text{R6} \mid \text{R7} \\
| \text{TGEI} \mid \text{TGEIU} \mid \text{TLTI} \mid \text{TLTIU} \\
| \text{TEQI} \mid \text{R13} \mid \text{TNEI} \mid \text{R15} \\
| \text{BLTZAL} \mid \text{BGEZAL} \mid \text{BLTZALL} \mid \text{BGEZALL} \\
| \text{R20} \mid \text{R21} \mid \text{R22} \mid \text{R23} \\
| \text{R24} \mid \text{R25} \mid \text{R26} \mid \text{R27} \\
| \text{R28} \mid \text{R29} \mid \text{R30} \mid \text{R31} \\
\]

deriving \ (\text{Bits,Eq})

Instruction Decode- Pack

\[
\text{instance Bits Instruction 32 where} \\
\text{pack :: Instruction } \rightarrow \text{Bit 32} \\
\text{pack (Immediate op rs rt imm)} = \\
\text{pack (Register rs rt rd sa funct)} = \\
\text{pack (RegImm rs op imm)} = \\
\text{pack (Jump op target)} = \\
\text{pack (Nop)} = 0
\]

http://www.csg.lcs.mit.edu/6.827
Instruction Decode - Unpack

```haskell
instance Bits Instruction 32 where
    unpack :: Bit 32 -> Instruction
    unpack bs when isImmInstr bs = Immediate {
        op = unpack bs[31:26];
        rs = unpack bs[25:21];
        rt = unpack bs[20:16];
        imm = unpack bs[15:0];
    }
    unpack bs when isREGIMMInstr bs = RegImm {
        rs = unpack bs[25:21];
        op = unpack bs[20:16];
        imm = unpack bs[15:0];
    }
    unpack bs when isJumpInstr bs = Jump {
        op = unpack bs[31:26];
        target = unpack bs[25:0];
    }
```

Decoding Functions

```haskell
isImmInstr :: Bit (SizeOf Instruction) -> Bool
isImmInstr bs = not (isSpecialInstr bs || isREGIMMInstr bs || isJumpInstr bs)

isREGIMMInstr :: Bit (SizeOf Instruction) -> Bool
isREGIMMInstr bs = bs[31:26] == (1::Bit 6)

isJumpInstr :: Bit (SizeOf Instruction) -> Bool
isJumpInstr bs = isJumpOp (unpack bs[31:26])

isSpecialInstr :: Bit (SizeOf Instruction) -> Bool
isSpecialInstr bs = bs[31:26] == (0::Bit 6)
```

Outline

• Lennart’s problem √

• Instruction Encoding √
  – Pack and Unpack

• Wallace Tree Addition ⇐

• Solution to Lennart’s problem

Wallace addition

Add several m-bit numbers

Add several m-bit numbers
Basic step: idea

Step, across all the bags of bits
Putting it all together

Given a list of numbers $x_0, x_1, \ldots, x_{k-1}$,
- unpack each number into $m$ bits $b_0, b_1, \ldots, b_{m-1}$ (thus the first element of list will contain the least significant bit of $x$
- transpose the list of bitbags such that the $i^{th}$ element of the list contains the $i^{th}$ bit of each of the $k$ numbers
- pad the list with sufficient Nil's (empty bitbags) so that its length is equal to $n$, the desired number of bits in the answer
- apply the Wallace algorithm
- extract the bit from each of the $n$ bitbags
- pack the $n$ bits to form the answer

$\text{wallaceAdder} = \text{pack} \cdot (\text{map head}) \cdot \text{wallace} \cdot \text{padWithNil} \cdot \text{transpose} \cdot (\text{map unpack})$
Basic step: Full adders on a list of bits

```haskell
type BitBag = List (Bit 1)
step :: (BitBag, BitBag) -> BitBag -> (BitBag, BitBag)
step (cs,ss) Nil = (cs,ss)
step (cs,ss) (Cons x Nil) = (cs,(Cons x ss))
step (cs,ss) (Cons x (Cons y Nil)) =
  let (c,s) = halfAdd x y
  in  ((Cons c cs),(Cons s ss))
step (cs,ss) (Cons x (Cons y (Cons z bs))) =
  let (c,s) = fullAdd x y z
  in  step ((Cons c cs),(Cons s ss)) bs
```

Apply `step` to `bitbags`, i.e. to `bag_0`, `bag_1`, ..., `bag_{n-1}`

Combine: carry-bitbag\_i and sum-bitbag\_i+1

```haskell
combine :: List (BitBag, BitBag) -> List BitBag
   carry sum
combine csbags =
  zipWith append

wallaceStep :: List BitBag -> List BitBag
wallaceStep bitbags =
  combine (map bitbags)
```

http://www.csg.lcs.mit.edu/6.827
Wallace algorithm

```
while p f x = if p x then (while p f (f x))
else x
isLengthGT2 x = (length x) > 2
isAnyLengthGT2 xs = foldr (or) False (map isLengthGT2 xs)
```

```
wallace :: List BitBag -> List BitBag
wallace bitbags =
  let twoNumbers =
    while isAnyLengthGT2 wallaceStep bitbags
    in  fastAdd2 twoNumbers

wallaceAdder = pack · (map head) · wallace ·
                padWithNil · transpose · (map unpack)
```

Stateful Wallace Step using `wallaceStep`

```
wallaceStepM :: (Bit n*k) -> Module (Bit n*k')
wallaceStepM inReg =
  Module
    regOut :: (Register (Bit n*k')) <- mkReg _
    inBitbagsN :: ListN n (ListN k (Bit 1))
    inBitbagsN = unpack inReg
    inBitbags :: List (List (Bit 1))
    inBitbags = toList (map toList inBitbagsN)
    outBitbags :: List (List (Bit 1))
    outBitbags = wallaceStep inBitbags
    outBitbagsN :: ListN n (ListN k' (Bit 1))
    outBitbagsN = toListN (map toList outBitbags)

rules
  when True ==> regOut := pack outBitbagsN

interface
  regOut.read
```

http://www.csg.lcs.mit.edu/6.827
Pipelined Wallace

\[
\text{while} :: (t \rightarrow \text{Bool}) \rightarrow (t \rightarrow t) \rightarrow t \rightarrow t \\
\text{while} \ p \ f \ x = \text{if } p \ x \ \text{then} \ \text{while} \ p \ f \ (f \ x) \ \text{else} \ x
\]

\[
\text{whileM} :: (t \rightarrow \text{Bool}) \rightarrow (t \rightarrow (\text{Module } t)) \rightarrow t \rightarrow (\text{Module } t) \\
\text{whileM} \ p \ f \ x = \text{if } p \ x \ \text{then do} \\
\quad x' \leftarrow f \ x \\
\quad \text{whileM} \ p \ f \ x' \\
\text{else do} \\
\quad \text{return } x
\]

\[
\text{wallaceM} :: \text{(Bit } n^k) \rightarrow \text{Module } (\text{Bit } n^2) \\
\text{wallaceM} = \text{whileM isAnyLengthGT2 wallaceStepM}
\]

\text{wallaceM does not work because of types!}

Alternatives

- Write a less parameterized solution.
  - Given a \( k \) we can figure out how many wallace iterations are needed and do all the unfolding manually
- Keep the register size the same after every iteration
  - need to pack the bits in some suitable order
  - extra hardware and may be messy coding
  - different termination condition
- Fix the language!
  - discussions underway
Manual unrolling

\[
\text{wallaceStepM : : (Bit n \times k) -> Module (Bit n \times k')} \]

\[
\text{wallaceM : : (Bit n \times k) -> Module (Bit n \times 2)} \]
\[
\text{wallaceM x = do}
\]
\[
x' : : (\text{Bit n} \times k') - k' is 2 \times \text{ceiling}(k/3)
\]
\[
x' \leftarrow \text{wallaceStepM x}
\]
\[
x'' : : (\text{Bit n} \times k'') - k'' is 2 \times \text{ceiling}(k'/3)
\]
\[
x'' \leftarrow \text{wallaceStepM x'}
\]
\[
\ldots
\]
\[
\text{return}
\]
\[
\text{xfinal}
\]

Lennart’s Borneo Numbers

Determine if a n-bit number contains exactly one “1”.

\[
\text{data Borneo = Zero | One | Many}
\]
\[
\text{toB : : Bit 1 -> Borneo}
\]
\[
\text{toB 0 = Zero}
\]
\[
\text{toB 1 = One}
\]
\[
\text{isMany : : Borneo -> Bool}
\]
\[
\text{isMany Many = True}
\]
\[
\text{isMany _ = False}
\]
\[
\text{addB : : Borneo -> Borneo -> Borneo}
\]
\[
\text{addB Zero n = n}
\]
\[
\text{addB One Zero = One}
\]
\[
\text{addB _ _ = Many}
\]