Bluespec-5
Programming Examples

Arvind
Laboratory for Computer Science
M.I.T.

Lecture 21

Quiz

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

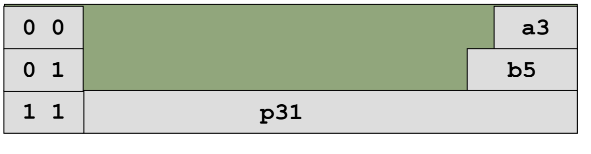
Outline

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

"deriving (Bits)" for algebraic types

```
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:



"33 bit" encoding !

Explicit pack & unpack

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

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

```
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]
             elseif
```

Instruction Encoding: *MIPS*

	6	5	5	5	5	6
Reg-Reg	Op	Rs1	Rs2	Rd	Const	Opx

Reg-Imm	Op	Rs1	Rd	Const
Branch	Op	Opx	Rs1	Const

Jump/Call	Op	Const

MIPS Instruction Type

```
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

```
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)
```

```
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
        | COP0 | COP1 | COP2 | OP19
        | BEQL | BNEL | BLEZL | BGTZL
        | DADDIe | DADDIUe | LDLe | LDRe
        | OP28 | OP29 | OP30 | OP31
        | LB | LH | LWL | LW | LBU | LHU | LWR | LWUe
        | SB | SH | SWL | SW | SDLe | SDRe | SWR | CACHEd
        | LL | LWC1 | LWC2 | OP51 | LLDe | LDC1 | LDC2 | LDe
        | SC | SWC1 | SWC2 | OP59 | SCDe | 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
             | DSLLVe | F15 | DSRLVe | DSRAVe
             | MULT | MULTU | DIV | DIVU
             | DMULTe | DMULTUe | DDIVe | DDIVUe
             | ADD | ADDU | SUB | SUBU
             | AND | OR   | XOR | NOR
             | F40 | F41  | SLT | SLTU
             | DADDe | DADDUe | DSUBe | DSUBUe
             | TGE | TGEU | TLT | TLTU
             | TEQ | F53  | TNE | F55
             | DSLLe | F57 | DSRLe | DSRAe
             | DSLL32e | F61 | DSRL32e | DSRA32e
           deriving (Bits,Eq)
```

Funct Type: MIPS Instructions

```
data REGIMM = BLTZ | BGEZ | BLTZL | BGEZL
            | R4 | R5 | R6 | R7
            | TGEI | TGEIU | TLTI | TLTIU
            | TEQI | R13 | TNEI | R15
            | BLTZAL | BGEZAL | BLTZALL | BGEZALL
            | R20 | R21 | R22 | R23
            | R24 | R25 | R26 | R27
            | R28 | R29 | R30 | R31
        deriving (Bits,Eq)
```


Instruction Decode- Pack

```
instance Bits Instruction 32 where
      pack :: Instruction -> Bit 32
      pack (Immediate op rs rt imm)  =


      pack (Register rs rt rd sa funct)  =



      pack (RegImm rs op imm)  =



      pack (Jump op target)  =

      pack (Nop)  = 0
```

Instruction Decode - Unpack

```
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

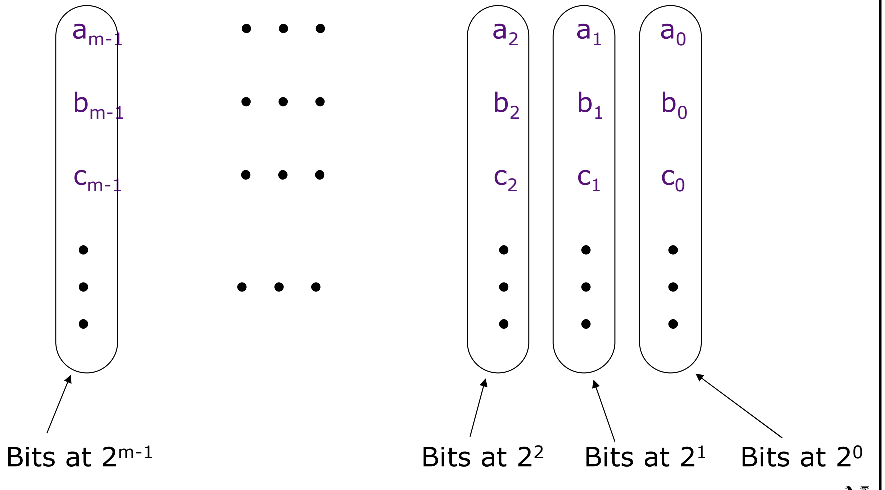
```
isImmInstr :: Bit (SizeOf Instruction) -> Bool
isImmInstr bs = not (isSpecialInstr bs || isREGIMMInstr bs
                     || isJumpInstr bs )

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

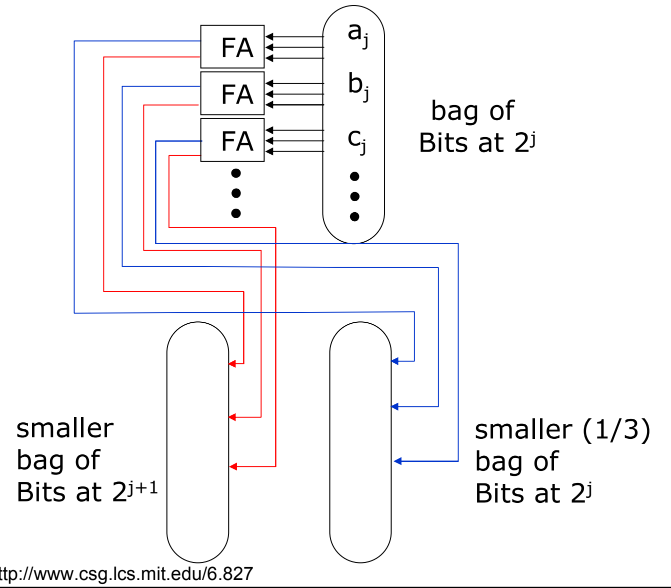
- Lennart’s problem √
- Instruction Encoding √
  - Pack and Unpack
- Wallace Tree Addition ⇐
- Solution to Lennart’s problem

Wallace addition

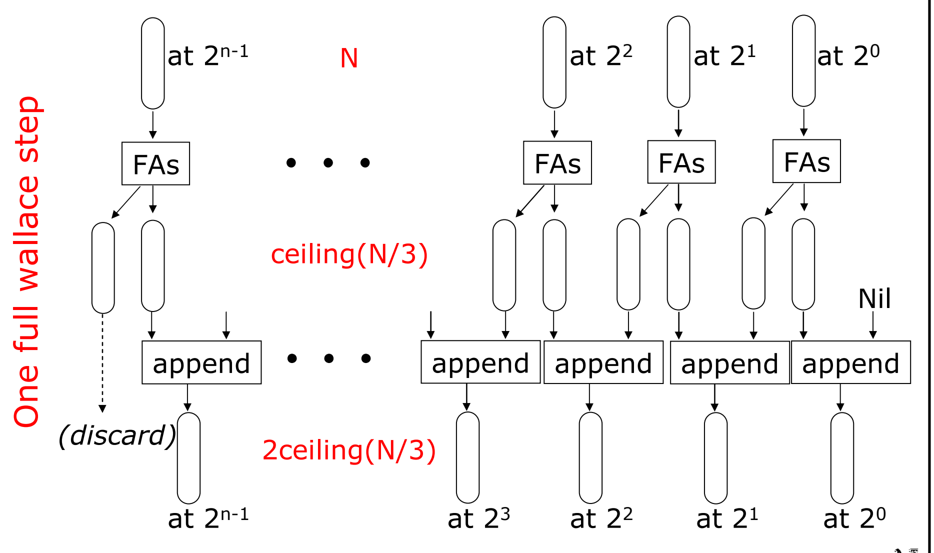
Add several m-bit numbers



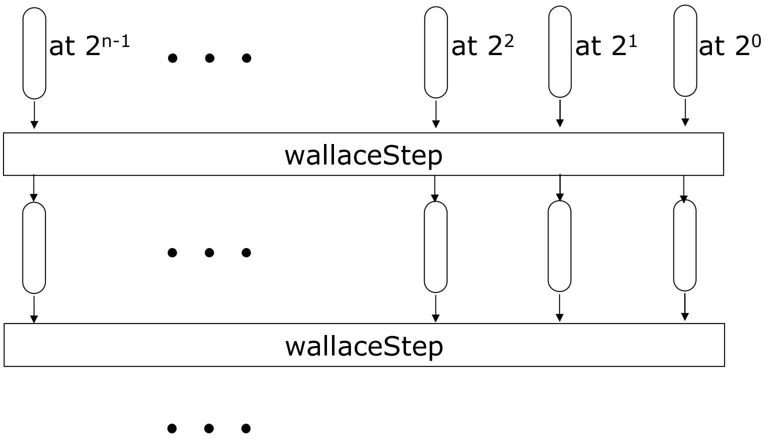
Basic step: *idea*



Step, across all the bags of bits



Putting it all together



until every bag has 2 bits in it,
at which point we can use normal adder

Putting it all together

Given a list of numbers x_0, x_1, ..., x_{k-1},

- unpack each number into m bits b_0, b_1, ..., 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

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

Basic step: *Full adders on a list of bits*

```
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$_{i+1}$

```
combine :: List (BitBag, BitBag) -> List BitBag
```

carry sum

```
combine csbags =
      zipWith append
```

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

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 toListN outBitbags)
       rules
          when True ==> regOut := pack outBitbagsN
       interface
          regOut.read
```

Pipelined Wallace

```
while :: (t->Bool) -> (t->t) -> t -> t
while p f x = if p x then (while p f (f x)) else x
```

```
whileM :: (t->Bool) -> (t->(Module t)) -> t -> (Module t)
whileM p f x = if p x then do
                              x' <- f x
                              (whileM p f x')
                          else do
                              return x
```

```
wallaceM :: (Bit n*k) -> Module (Bit n*2)
wallaceM = whileM isAnyLengthGT2 wallaceStepM
```

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

```
wallaceStepM :: (Bit n*k) -> Module (Bit n*k')


wallaceM :: (Bit n*k) -> Module (Bit n*2)
wallaceM x =
      do
             x' :: (Bit n*k') - k' is 2 * ceiling (k/3)
             x'  <- wallaceStepM x
             x'' :: (Bit n*k'') - k'' is 2 * ceiling (k'/3)
             x'' <- wallaceStepM x'
                ...
             return
               xfinal
```


Lennart’s Borneo Numbers

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

```
data Borneo = Zero | One | Many

toB :: Bit 1 -> Borneo
toB 0 = Zero
toB 1 = One

isMany :: Borneo -> Bool
isMany Many = True
isMany _    = False

addB :: Borneo -> Borneo -> Borneo
addB Zero n    = n
addB One Zero = One
addB  _   _    = Many
```