Bluespec-5
Type Classes, Bits and Instruction Encoding

Arvind
Laboratory for Computer Science
M.I.T.

January 15, 2003

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

Outline

- Type classes ⇐
  - Class Eq
  - Type Bit and Class Bits
  - Type Integer and Class Literal
  - Type classes for numeric types
- Instruction Encoding

Overloading and Type classes

- Overloading: using a common name for *similar*, but conceptually *distinct* operations
  - Example:
    - `n1 < n2` where n1 and n2 are integers
    - `s1 < s2` where s1 and s2 are strings
  - *Distinct*: These orderings may have nothing to do with each other -- their implementations are likely to be totally different
  - *Similar*: integer "<" and string "<" may share some common properties, such as
    - transitivity: $(a < b)$ and $(b < c) \Rightarrow (a < c)$
    - irreflexivity: $(a < b) \Rightarrow \sim(b < a)$
- *Type classes* may be seen as a systematic mechanism for overloading

Type classes

- A type class is a *collection of types*, all of which share a common set of operations with similar type signatures
- Examples:
  - All types t in the "Eq" class have equality and inequality operations:

```
class Eq t where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool
```

  - All types t and n in the "Bits" class have operations to convert objects of type t into bit vectors of size n and back:

```
class Bits t n where
  pack   :: t -> Bit n
  unpack :: Bit n -> t
```

How does a type become a member of a class?

- Membership is not automatic: a type has to be declared to be an *instance* of a class, and implementations of the corresponding operations must be supplied
  - Until t is a member of Eq, you cannot use the "==" operation on values of type t
  - Until t is a member of Bits, you cannot store them in hardware state elements like registers, memories and FIFOs
- The general way to do this is with an "**`instance`**" declaration
- A frequent shortcut is to use a "**`deriving`**" clause when declaring a type

Class “Bits”

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
         deriving (Bits)
```

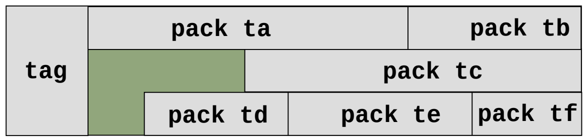
The "deriving" clause
- Declares type Day to be an instance of the Bits class
- Defines the two associated functions

```
pack   :: Day -> Bit 3
unpack :: Bit 3 -> Day
```

"deriving (Bits)" for algebraic types

```
data T = C0 ta tb | C1 tc | C2 td te tf
       deriving (Bits)
```

The canonical "pack" function created by "deriving (Bits)" produces the following packings:

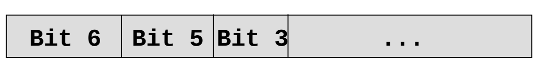


where "tag" is 0 for C0, 1 for C1, and 2 for C2.

"deriving (Bits)" for structs

- The canonical "pack" function simply bit-concatenates the packed versions of the fields:

```
struct PktHdr =
    node    :: Bit 6          -- NodeID
    port    :: Bit 5          -- PortID
    cos     :: Bit 3          -- CoS
    dp      :: Bit 2          -- DropPrecedence
    ecn     :: Bool
    res     :: Reserved 1
    length  :: Bit 14         -- PacketLength
    crc     :: Bit 32
  deriving (Bits)
```



Explicit pack & unpack

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

- Explicit "instance" decls. may permit more efficient packing than 33 bits

"32 bit" encoding !


```
instance Bits T 32 where
  pack (A a3)    = 0b00 ++ (zeroExtend a3)
  pack (B b5)    = 0b01 ++ (zeroExtend b5)
  pack (Ptr p31) = 0b1  ++ p31                  ?

  unpack x = if     x[31:30] == 0b00 then A x[2:0]
             elseif x[31:30] == 0b01 then B x[4:0]
             elseif x[31:31] == 0b1  then Ptr x[30:0]  ?
```


Class "Eq"

```
class Eq t where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool
```

- "deriving (Eq)" will generate the natural versions of these operators automatically
  - Are the tags equal?
  - And, if so, are the corresponding fields equal?
- An "instance" declaration may be used for other meanings of equality, e.g.,
  - "two pointers are equal if their bottom 20 bits are equal"
  - "two values are equal if they hash to the same address"

Type "Integer" and class "Literal"

- The type "Integer" refers to pure, unbounded, mathematical integers
  - and, hence, Integer is *not* in class Bits, which can only represent bounded quantities
  - Integers are used only as compile time entities

- The class "Literal" contains a function:

```
class Literal t where
    fromInteger :: Integer -> t
```


Class "Literal"

- Types such as (Bit n), (Int n), (Uint n) are all members of class Literal
  - Thus,

    ```
    (fromInteger 523) :: Bit 13
    ```

    represents the integer 523 as a 13-bit quantity
  - while

    ```
    (fromInteger 523) :: Int 13
    ```

    represents the integer 523 as a 13-bit Int type

- This is how all literal numbers in the program text, such as "0" or "1", or "23", or "523" are treated, i.e., they use the systematic overloading mechanism to convert them to the desired type

Type classes for numeric types

- More generally, type classes can be seen as *constraints* on types
- Examples:
  - For all numeric types t1, t2, t3 in the "Add" class, the value of t3 is the sum of the values of t1 and t2.
  - For all numeric types t1, t2 in the "Log" class, the value of t2 is large enough that a (Bit t2) value can represent values in the range 0 to valueOf t1-1
- These classes are used to represent/derive relationships between various "sizes" in a piece of hardware

Type classes for numeric types

- Suppose we have an array of *n* locations. An index into the array needs $k=\log_2(n)$ bits to represent values in the range 0 to n-1

```
mkTable :: (Bits t ts, Log n k) => Table n t
mkTable =
    module
        a :: Array (Bit k) t
        a <- mkArrayFull

        index :: Reg (Bit k)
        index <- mkRegU
        ...
```

Outline

- Type classes √
  - Class Eq
  - Type Bit and Class Bits
  - Type Integer and Class literal
  - Type classes for numeric types
- Instruction Encoding ⇐

MIPS Instruction Formats

from the MIPS manual

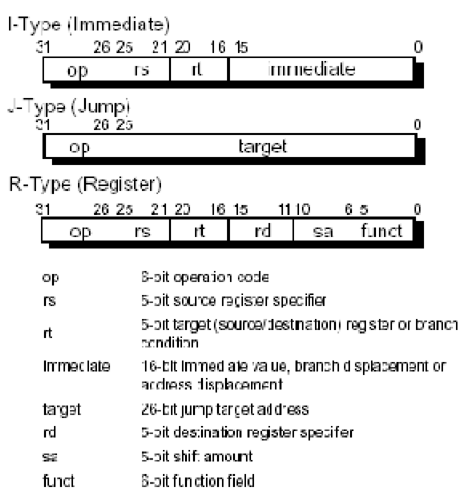


Figure 2-1 CPU Instruction Formats

MIPS Instruction Formats: Load/Store

Load/Store Instructions

100000	base	dest	signed offset	LB rt, offset(rs)
100001	base	dest	signed offset	LH rt, offset(rs)
100011	base	dest	signed offset	LW rt, offset(rs)
100100	base	dest	signed offset	LBU rt, offset(rs)
100101	base	dest	signed offset	LHU rt, offset(rs)
101000	base	dest	signed offset	SB rt, offset(rs)
101001	base	dest	signed offset	SH rt, offset(rs)
101011	base	dest	signed offset	SW rt, offset(rs)

MIPS Instruction Formats: Arithmetic/logic

Immediate Instructions

001000	src	dest	signed immediate	ADDI rt, rs, signed imm.
001001	src	dest	signed immediate	ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate	SLTI rt, rs, signed imm.
001011	src	dest	signed immediate	SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext immediate	ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext immediate	ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext immediate	XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext immediate	LUI rt, zero-ext-imm.

Register Instructions

000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rs/amt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rs/amt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rs/amt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100000	ADD rd, rs, rt
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100010	SUB rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt

L5-19
Arvind

MIPS Instruction Formats:
Jumps/Branches

000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
010100	src1	src2	signed offset			BEQL rs, rt, offset
010101	src1	src2	signed offset			BNEL rs, rt, offset
010110	src	00000	signed offset			BLEZL rs, offset
010111	src	00000	signed offset			BGTZL rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset
000001	src	00010	signed offset			BLTZL rs, offset
000001	src	00011	signed offset			BGEZL rs, offset
000001	src	10000	signed offset			BLTZAL rs, offset
000001	src	10001	signed offset			BGEZAL rs, offset
000001	src	10010	signed offset			BLTZALL rs, offset
000001	src	10011	signed offset			BGEZALL rs, offset

January 15, 2003 http://www.csg.lcs.mit.edu/IAPBlue

L5-20
Arvind

Decoding MIPS Instructions

- The input instruction formats for decoding are fixed but we choose the output instruction formats depending upon our need.
  - Dan' s decoded format
  - Jacob's decoded format

- Instruction decoding can be expressed as pack/unpack of these defined types into 32-bit values that correspond to MIPS instructions

January 15, 2003 http://www.csg.lcs.mit.edu/IAPBlue

Naming the fields in MIPS instructions

```
opcode   = b32[31:26]
rs        = b32[25:21]
rt        = b32[20:16]
rd        = b32[15:11]
shiftamt  = b32[10:6]
funct     = b32[5:0]
imm       = b32[15:0]
itarget   = b32[25:0]
zr        = 0b000000
brt       = b32[20:16]
```

As a convenience we can define new data types to refer to the contents some fields

```
opcode  :: OpcodeT
rs       :: RegT
rt       :: RegT
rd       :: RegT
funct   :: FunctionT
```


OpcodeT

```
data OpcodeT =
       SPECIAL  |
       REGIMM   |
       J        |
       JAL      |
       BEQ      |
       BNE      |
       BLEZ     |
       BGTZ     |
       ADDI     |
       ADDIU    |
       SLTI     |
       SLTIU    |
       ANDI     |
       ORI      |
       XORI     |
       LUI      |
       LW       |
       SW
```

```
instance Bits OpcodeT 6 where
       pack SPECIAL   = 0b000000
       pack REGIMM    = 0b000001
       pack J         = 0b000010
       pack BEQ       = 0b000100
       pack ADDI      = 0b001000
       pack LW        = 0b100011
       pack SW        = 0b101011
        ...
       unpack 0b000000 = SPECIAL
       unpack 0b000001 = REGIMM
       unpack 0b000010 = J
       unpack 0b000100 = BEQ
       unpack 0b001000 = ADDI
       unpack 0b100011 = LW
       unpack 0b101011 = SW
        ...
```

Dan's format

```
data MIPSInstructionT =
       LW_T {base::RegT; dest::RegT; soff::ImmT;} |
       SW_T {base::RegT; dest::RegT; soff::ImmT;} |
              ...
       ADDI_T  {src::RegT; dest::RegT; simm::ImmT;} |
       ADDIU_T {src::RegT; dest::RegT; simm::ImmT;} |
              ...
       ADD_T  {src1::RegT; src2::RegT; dest::RegT;} |
       ADDU_T {src1::RegT; src2::RegT; dest::RegT;} |
              ...
       J_T   {target::JumpTargetT;} |
       JAL_T {target::JumpTargetT;} |
              ...
       BEQ_T   {src1::RegT; src2::RegT; simm::ImmT;} |
       BNE_T   {src1::RegT; src2::RegT; simm::ImmT;} |
              ...
       JR_T   {src::RegT;}             |
       JALR_T {src::RegT; dest::RegT;} |
              ...
```

```
type RegT = Bit 5
type ImmT = Bit 16
type JumpTargetT = Bit 26
```


Dan's pack

```
instance Bits MIPSInstructionT 32 where

   pack (LW_T {base; dest; soff;}) =
        (pack LW) ++ (pack base) ++ (pack dest) ++ (pack soff)
   pack (SW_T {base; dest; soff;}) =
       (pack SW) ++ (pack base) ++ (pack dest) ++ (pack soff)
         ...
```

L5-25
Arvind

Dan’s unpack

```
instance Bits MIPSInstructionT 32 where
   unpack b32 =
        let
            opcode :: OpcodeT   = unpack b32[31:26]
            rs     :: RegT      = unpack b32[25:21]
            rt     :: RegT      = unpack b32[20:16]
            rd     :: RegT      = unpack b32[15:11]
            shiftamt            = unpack b32[10:6]
            funct  :: FunctionT = unpack b32[5:0]
            imm    :: ImmT      = unpack b32[15:0]
            itarget             = unpack b32[25:0]
            zr                  = unpack 0b000000
            brt    :: RegT      = unpack b32[20:16]
        in
            case opcode of ...
```

unpack is like decoding the instruction

L5-26
Arvind

Dan’s unpack *continued*

```
case opcode of
        LW      -> LW_T {base = rs; dest = rt; soff = imm;}
        SW      -> SW_T {base = rs; dest = rt; soff = imm;}
        ADDI    -> ADDI_T  {src = rs; dest = rt; simm = imm;}
        ADDIU   -> ADDIU_T {src = rs; dest = rt; simm = imm;}
        SPECIAL ->
           case funct of
                ADD     -> ADD_T   {src1 = rs; src2 = rt; dest = rd;}
                ADDU    -> ADDU_T  {src1 = rs; src2 = rt; dest = rd;}

                JR      -> JR_T    {src = rs;}
                JALR    -> JALR_T  {src = rs; dest = rd;}
        J       -> J_T    {target = itarget;}
        JAL     -> JAL_T  {target = itarget;}
        BEQ     -> BEQ_T  {src1 = rs; src2 = rt; simm = imm;}
        REGIMM  ->
            case brt of
                BLTZ    -> BLTZ_T   {src = rs; soff = imm;}
                BGEZ    -> BGEZ_T   {src = rs; soff = imm;}
                BLTZAL  -> BLTZAL_T {src = rs; soff = imm;}
                BGEZAL  -> BGEZAL_T {src = rs; soff = imm;}
        ...
```

Jacob's format

```
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 op) ++ (pack rs) ++ (pack rt) ++ (pack imm)
      pack (Register rs rt rd sa funct) =
          (pack SPECIAL) ++ (pack rs) ++ (pack rt) ++
                   (pack rd) ++ (pack sa) ++ (pack funct)
      pack (RegImm rs op imm) =
           (pack REGIMM) ++ (pack rs) ++ (pack op) ++
                 (pack imm)
      pack (Jump op target) =  (pack op) ++ (pack target)
      pack (Nop) = 0
```

```
Immediate     op     :: OPReg
              rs     :: CPUReg
              rt     :: CPUReg
              imm    :: UInt16
```

```
Register      rs     :: CPUReg
              rt     :: CPUReg
              rd     :: CPUReg
              sa     :: UInt5
              funct  :: Funct
```

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

```
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)

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