Bluespec-4
Modules and Type Classes

Arvind
Laboratory for Computer Science
M.I.T.

Lecture 20

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

Outline

- Phase 1 compilation: Flattening the modules
- Type classes
  - Class Eq
  - Type Bit and Class Bits
  - Type Integer and Class literal
- ListN: Lists of fixed size
LPM: Straight Pipeline Solution

Bluespec code: Straight pipeline

```plaintext
mkLPM :: AsyncROM lat LuAddr LuData -> Module LPM
mkLPM rom =
  module
    (rom0, rom1, rom2) <- mk3ROMports rom
    fifo0 :: FIFO Mid <- mkFIFO
    fifo1 :: FIFO Mid <- mkFIFO
    fifo2 :: FIFO Mid <- mkFIFO
    ofifo :: FIFO LuResult <- mkFIFO

  rules
    ... for Stages 1, 2 and Completion ...

  interface
    -- Stage 0
    luReq ipa = action rom0.read (zeroExtend ipa[31:16])
    fifo0.enq (Lookup (ipa << 16))
    luResp = ofifo.first
    luRespAck = ofifo.deq
```
Straight pipeline *cont.*

```haskell
data Mid = Lookup IPaddr | Done LuResult
mkLPM rom =
module
  ... state is rom0, rom1, rom2, fifo0, fifo1, fifo2, ofifo
rules
  -- Stage 1: lookup, leaf
  when Lookup ipa <- fifo0.first,
    Leaf res <- rom0.result
  ==> action fifo0.deq
  rom0.ack
  fifo1.enq (Done res)

  -- Stage 1: lookup, node
  when Lookup ipa <- fifo0.first,
    Node res <- rom0.result
  ==> action fifo0.deq
  rom0.ack
  rom1.read (addr + (zeroExt ipa[31:24]))
  fifo1.enq (Lookup (ipa << 8))
```

LPM code structure

```haskell
mkLPM rom =
module
  (rom0, rom1, rom2) <- mk3ROMports rom
  fifo0 <- mkFIFO
  fifo1 <- mkFIFO
  fifo2 <- mkFIFO
  ofifo <- mkFIFO
rules
  RuleStage1Leaf(fifo0, fifo1, rom0)
  RuleStage1Node(fifo0, fifo1, rom0, rom1)
  RuleStage2Noop(fifo1, fifo2)
  RuleStage2Leaf(fifo1, fifo2, rom1)
  RuleStage2Node(fifo1, fifo2, rom1, rom2)
  RuleCompletionNoop(fifo2, ofifo)
  RuleCompletionLeaf(fifo2, ofifo, rom2)
  RuleCompletionNode(fifo2, ofifo, rom2)
interface
  luReq = EluReq(fifo0, rom0)
  luResp = EluResp(ofifo)
  luRespAck = EluRespAck(ofifo)
```

Free variables of the rule
Port replicator code structure

```haskell
mk3ROMports rom =
    module
    tags <- mkSizedFIFO
    let
        mkPort i =
            module
                out <- mkSizedFIFO
                cnt <- mkCounter
                rules
                    RuleTags(i, rom, tags, out)
                interface
                    read = Eread(i, rom, tags, cnt)
                    result = Eresult(out)
                    ack = Eack(out, cnt)
        port0 <- mkPort 0
        port1 <- mkPort 1
        port2 <- mkPort 2
        interface (port0, port1, port2)
```

Port replicator – after step 1

```haskell
mk3ROMports rom =
    module
    tags <- mkSizedFIFO
    port0 <-
        module
            out <- mkSizedFIFO
            cnt <- mkCounter
            rules
                RuleTags(0, rom, tags, out)
            interface
                read = Eread(0, rom, tags, cnt)
                result = Eresult(out)
                ack = Eack(out, cnt)
    port1 <- ...
similarly...
    port2 <- ...
similarly...
    interface (port0, port1, port2)
```

Step 2: Flatten the module renaming bound variables
Port replicator – after step 2

```plaintext
mk3ROMports rom =
module
tags <- mkSizedFIFO
out0 <- mkSizedFIFO
cnt0 <- mkCounter
rules
  RuleTags(0, rom, tags, out0)
let port0 = interface
  read = Eread(0, rom, tags, cnt0)
  result = Eresult(out0)
  ack = Eack(out0, cnt0)

port1 <- ...similarly...
port2 <- ...similarly...
interface (port0, port1, port2)
```

Port replicator – final step

```plaintext
mk3ROMports rom =
module
tags <- mkSizedFIFO
out0 <- mkSizedFIFO ; cnt0 <- mkCounter
out1 <- mkSizedFIFO ; cnt1 <- mkCounter
out2 <- mkSizedFIFO ; cnt2 <- mkCounter
rules
  RuleTags(0, rom, tags, out0)
  RuleTags(1, rom, tags, out1)
  RuleTags(2, rom, tags, out2)
let port0 = interface
  read = Eread(0, rom, tags, cnt0)
  result = Eresult(out0)
  ack = Eack(out0, cnt0)

Next step:
substitute
mk3ROMports
into mkLPM

port1 = interface
  read = Eread(1, rom, tags, cnt1)
  ...

port2 = interface ...
interface (port0, port1, port2)
```
Port replicator call

\[(\text{rom}0, \text{rom}1, \text{rom}2) \leftarrow \text{mk3ROMports}\, \text{rom}\]

tags \leftarrow \text{mkSizedFIFO}
out0 \leftarrow \text{mkSizedFIFO};\, \text{cnt}0 \leftarrow \text{mkCounter}
out1 \leftarrow \text{mkSizedFIFO};\, \text{cnt}1 \leftarrow \text{mkCounter}
out2 \leftarrow \text{mkSizedFIFO};\, \text{cnt}2 \leftarrow \text{mkCounter}

rules
RuleTags(0, \text{rom}, \text{tags}, \text{out}0)
RuleTags(1, \text{rom}, \text{tags}, \text{out}1)
RuleTags(2, \text{rom}, \text{tags}, \text{out}2)

let port0 = interface
\quad read = \text{Eread}(0, \text{rom}, \text{tags}, \text{cnt}0)
\quad result = \text{Eresult}(\text{out}0)
\quad ack = \text{Eack}(\text{out}0, \text{cnt}0)
\quad port1 = interface \ldots
\quad port2 = interface \ldots

(\text{rom}0, \text{rom}1, \text{rom}2) = (\text{port}0, \text{port}1, \text{port}2)

After Port replicator call substitution

\[(\text{rom}0, \text{rom}1, \text{rom}2) \leftarrow \text{mk3ROMports}\, \text{rom}\]

tags \leftarrow \text{mkSizedFIFO}
out0 \leftarrow \text{mkSizedFIFO};\, \text{cnt}0 \leftarrow \text{mkCounter}
out1 \leftarrow \text{mkSizedFIFO};\, \text{cnt}1 \leftarrow \text{mkCounter}
out2 \leftarrow \text{mkSizedFIFO};\, \text{cnt}2 \leftarrow \text{mkCounter}

rules
RuleTags(0, \text{rom}, \text{tags}, \text{out}0)
RuleTags(1, \text{rom}, \text{tags}, \text{out}1)
RuleTags(2, \text{rom}, \text{tags}, \text{out}2)

let rom0 = interface
\quad read = \text{Eread}(0, \text{rom}, \text{tags}, \text{cnt}0)
\quad result = \text{Eresult}(\text{out}0)
\quad ack = \text{Eack}(\text{out}0, \text{cnt}0)
\quad rom1 = interface \ldots
\quad rom2 = interface \ldots
LPM code after flattening

```haskell
mkLPM rom =
    module
        tags <- mkSizedFIFO;
    out0 <- mkSizedFIFO; cnt0 <- mkCounter;
    out1 <- mkSizedFIFO; cnt1 <- mkCounter;
    out2 <- mkSizedFIFO; cnt2 <- mkCounter;
    fifo0 <- mkFIFO; fifo1 <- mkFIFO; fifo2 <- mkFIFO;
    ofifo <- mkFIFO;
    rules
        RuleTags(0, rom, tags, out0)...
    let rom0 = interface
        read = Eread(0, rom, tags, cnt0)
        result = Eresult(out0)
        ack = Eack(out0, cnt0)
    rom1 = interface ... ; rom2 = interface ...
    RuleStage1Leaf(fifo0, fifo1, rom0)...
    interface
        luReq = EluReq(fifo0, rom0)
        luResp = EluResp(ofifo)
        luRespAck = EluRespAck(ofifo)
```

Outline

- Phase 1 compilation: Flattening the modules ✓
- Type classes ⇐
  - Class Eq
  - Type Bit and Class Bits
  - Type Integer and Class literal
- ListN: Lists of fixed size
Type classes

- Type classes may be seen as a systematic mechanism for overloading
  - Overloading: using a common name for similar, but conceptually distinct operations
  - Example:
    - \( n_1 < n_2 \) where \( n_1 \) and \( n_2 \) are integers
    - \( s_1 < s_2 \) where \( s_1 \) and \( s_2 \) are strings
  - Distinct: integer "\(<" and string "\(<" (using, say, lexicographic ordering) may not have anything to do with each other. In particular, 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 (\( a < b \) \( \Rightarrow \) not \( b < a \))

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:
    ```haskell
    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:
    ```haskell
    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

---

The Bits class

- Example:

```hs
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

```hs
pack :: Day -> Bit 3
unpack :: Bit 3 -> Day
```
"deriving (Bits)" for algebraic types

- Given an algebraic type such as:

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

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

  where "tag" is 0 for C0, 1 for C1, and 2 for C2, and has enough bits to represent C2.

- Thus, for:

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

  the canonical "pack" function produces:

  where "tag" is 0 for Sun, 1 for Mon, ..., 6 for Sat, and is a Bit 3.
Class "(Bits)" for algebraic types

- What if we had to inter-operate with hardware that used a different representation (e.g., 0-5 for M-Sa and 6 for Su)?
  - We use an explicit "instance" decl. instead of "deriving"

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

instance Bits Day 3 where
    pack Sun = 6
    pack Mon = 0
    ...
    pack Sat = 5

unpack 0 = Mon
    ...
unpack 6 = Sun
```

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

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

instance Bits T 32 where
    pack (A a3) = (00)::(Bit 2) ++ (zeroExtend a3)
    pack (B b5) = (01)::(Bit 2) ++ (zeroExtend b5)
    pack (Ptr p31) = (1)::(Bit 1) ++ p31

... 

unpack ...
```
"deriving (Bits)" for structs

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

```haskell
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)
```

Class "Eq"

- Class "Eq" contains the equality (==) and inequality (/=) operators
- "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:

  \[ \text{fromInteger} :: \text{Integer} \rightarrow t \]

Class "Literal"

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

  \[ \text{(fromInteger 523)} :: \text{Bit 13} \]

  will represent the number 523 as a 13-bit quantity

- 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 $t_1, t_2, t_3$ in the "Add" class, the value of $t_3$ is the sum of the values of $t_1$ and $t_2$.
  – For all numeric types $t_1, t_2$ in the "Log" class, the value of $t_2$ is large enough that a $(\text{Bit } t_2)$ value can represent values in the range 0 to $\text{valueOf } t_1 - 1$

• These classes are used to represent/derive relationships between various "sizes" in a piece of hardware

Type classes for numeric types

• Example: bit concatenation:

  $(++): (\text{Add } n \hspace{0.2cm} m \hspace{0.2cm} k) \Rightarrow \text{Bit } n \Rightarrow \text{Bit } m \Rightarrow \text{Bit } k$

and its inverse:

  $\text{split}: (\text{Add } n \hspace{0.2cm} m \hspace{0.2cm} k) \Rightarrow \text{Bit } k \Rightarrow (\text{Bit } n, \text{Bit } m)$
Type classes for numeric types

• Example: a lookup table containing up to $n$ elements, each of type $t$
  – Suppose we store the elements in an array of $n$ locations. An index into the array needs $k=\log_2(n)$ bits to represent values in the range 0 to $n-1$.

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

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

  ...
```

Outline

• Phase 1 compilation: Flattening the modules √

• Type classes √
  – Class Eq
  – Type Bit and Class Bits
  – Type Integer and Class literal

• ListN: Lists of fixed size ←
The type ListN

- Unlike the type "List t", which represents a list of zero or more elements of type t, the type ListN n t represents a list of exactly n elements of type t.
- Advantage over List:
  - Can be converted into bits & wires, stored in registers and FIFOs, etc., since size is known
  - Can assert exactly how many items there are, e.g., "The arbiter module has a list of 16 interfaces"
- Disadvantage:
  - Cannot write recursive programs on ListN, if the size of the list keeps changing from call-to-call.
  - Alleviated by a rich library of functions like map, foldl, zip, ... where the size transformation is known (e.g., map preserves length)

Examples of ListN functions

- map preserves length
  \[
  \text{map} :: (a \to b) \to \text{ListN } n \ a \to \text{ListN } n \ b
  \]

- foldl's result has nothing to do with the input list's length
  \[
  \text{foldl} :: (b \to a \to b) \to b \to \text{ListN } n \ a \to b
  \]

- genList creates a list 1..n, but does not need an argument telling it about n!
  \[
  \text{genList} :: \text{ListN } n \ \text{Integer}
  \]
Examples of ListN functions cont.

- Conversion to and from ListN and List

\[
\text{toList} :: \text{ListN} \ n \ a \rightarrow \text{List} \ a \\
\text{toListN} :: \text{List} \ a \rightarrow \text{ListN} \ n \ a
\]