Chisel: Diplomacy and TileLink

Farhan Aslam

Associate Design Engineer, Lampro Mellon

Contents

Lazy val

- The `val`
  - is evaluated only once at the time of definition
  - once evaluated, the same value is used for all future references (without reevaluation)
- The `def`
  - is used to define functions or methods
  - is not evaluated at the time of definition
  - evaluated when called and is evaluated every time whenever called

Lazy val Cont'd

- The `lazy val`
  - is not evaluated at the time of definition
  - evaluation is delayed till its use for the first time
  - once evaluated, the same value is used for all future references (without reevaluation)

Lazy val Cont'd

```
// val evaluation illustration
object valApp extends App{
  val x = { println("x is initialized,"); 99 }
  println("before we access 'x'")
  println(x + 1)
}

// Output at the terminal
x is initialized,
before we access 'x'
100
```

Listing: The `val` evaluation.

Lazy val Cont'd

The `lazy val` evaluation

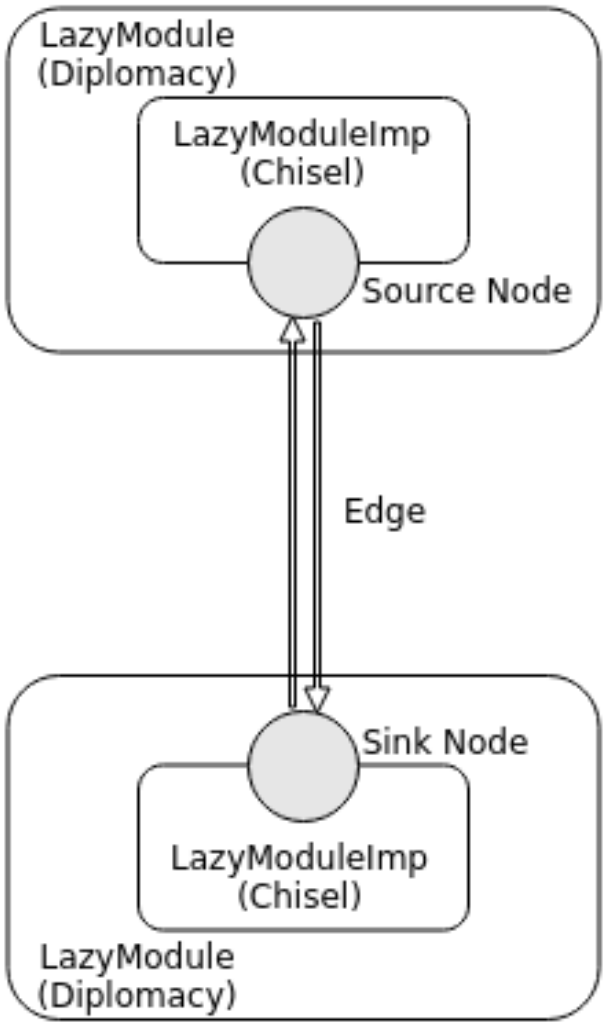
```
// lazy val illustration
object lazyvalApp extends App{
  lazy val x = { println("x is NOT initialized."); 99 }
  println("Unless we access 'x',")
  println(x + 1)
  println(x + 2)
}
```

```
// Output at the terminal
Unless we access 'x',
x is NOT initialized.
100
101
```

Diplomacy

- Diplomacy is a parameter negotiation framework for generating parameterized protocol implementations.
- Graph of logical interconnectivity
  - Directed Acyclic Graph (DAG) of nodes and edges.
  - Node:A point where parameterized hardware is going to be generated.
  - Edge:A directed connection between one masterand slave node.
  - LazyModule:Diplomatic Chisel modules that may have many nodes.
- Parameters are transmitted and derived among Lazy Modules through edges.

Diplomacy


LazyModule
(Diplomacy)
LazyModuleImp
(Chisel)
Source Node
Edge
Sink Node
LazyModuleImp
(Chisel)
LazyModule
(Diplomacy)

ClientNode

```
class MyClient(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeClientNode(TLClientParameters())
  lazy val module = new LazyModuleImp(this) {
    val (tl, edge) = node.out(0)
    // Rest of code here
  }
}
```

```
case class TLClientParameters(
  name:                String,
  sourceId:            IdRange          = IdRange(0,1),
  nodePath:            Seq[BaseNode]    = Seq(),
  requestFifo:         Boolean          = false,
  visibility:          Seq[AddressSet] = Seq(AddressSet(0,
      ~0)), // everything
  supportsProbe:       TransferSizes    = TransferSizes.none,
  supportsArithmetic:  TransferSizes    = TransferSizes.none,
  supportsLogical:     TransferSizes    = TransferSizes.none,
  supportsGet:         TransferSizes    = TransferSizes.none,
  supportsPutFull:     TransferSizes    = TransferSizes.none,
  supportsPutPartial:  TransferSizes    = TransferSizes.none,
  supportsHint:        TransferSizes    = TransferSizes.none)
      {...}
```

ClientNode

- `name:` It identifies the node in the Diplomacy graph. It is the only required argument for TLClientParameters.
- `sourceId:` It specifies the range of source identifiers that this client will use.
- `requestFifo:` It is a boolean option which defaults to false. If it is set to true, the client will request that downstream managers that support it send responses in FIFO order (that is, in the same order the corresponding requests were sent).
- `visibility:` argument specifies the address ranges that the client will access. By default it is set to include all addresses.

The edge object represents the edge of the Diplomacy graph. It contains some useful helper functions.

ManagerNode

```
class MyManager(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeManagerNode(beatBytes,
      TLManagerParameters())
  lazy val module = new LazyModuleImp(this) {
    val (tl, edge) = node.in(0)
 //Reset of the code goes here
  }
}
```

ManagerNode

```
case class TLManagerParameters(
  address:            Seq[AddressSet],
  resources:          Seq[Resource] = Seq(),
  regionType:         RegionType.T  = RegionType.GET_EFFECTS,
  executable:         Boolean       = false,
  nodePath:           Seq[BaseNode] = Seq(),
  supportsAcquireT:   TransferSizes = TransferSizes.none,
  supportsAcquireB:   TransferSizes = TransferSizes.none,
  supportsArithmetic:TransferSizes = TransferSizes.none,
  supportsLogical:    TransferSizes = TransferSizes.none,
  supportsGet:        TransferSizes = TransferSizes.none,
  supportsPutFull:    TransferSizes = TransferSizes.none,
  supportsPutPartial:TransferSizes = TransferSizes.none,
  supportsHint:       TransferSizes = TransferSizes.none,
  mayDenyGet:         Boolean = false, // applies to:
      AccessAckData, GrantData
  mayDenyPut:         Boolean = false, // applies to:
      AccessAck,     Grant,    HintAck
  alwaysGrantsT:      Boolean = false, // typically only true
       for CacheCork'd read-write devices
  fifoId:             Option[Int] = None,
  device: Option[Device] = None){...}
```

ManagerNode

- `address:` The only required argument for TLManagerParameters is the address, which is the set of address ranges that this manager will serve.
- `resources:` is usually retrieved from a Device object. This argument is necessary if you want to add an entry to the DeviceTree in the BootROM so that it can be read by a Linux driver. The two arguments to SimpleDevice are the name and compatibility list for the device tree entry.
- `regionType:` It gives some information about the caching behavior of the manager. CACHED, TRACKED, UNCACHED, VOLATILE.

Get Method

- It is a constructor for a TLBundleA encoding a Get message, which requests data from memory.
- Arguments:
  - `fromSource:` UInt - Source ID for this transaction
  - `toAddress:` UInt - The address to read from
  - `lgSize:` UInt - Base two logarithm of the number of bytes to be read
- Returns:
  - A (Bool, TLBundleA) tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this edge. The second is the A channel bundle.

From: [ChipYard, 2019]

Put Method

- It is a constructor for a TLBundleA encoding a `PutFull` or `PutPartial` message, which write data to memory. It will be a `PutPartial` if the `mask` is specified and a `PutFull` if it is omitted.
- Arguments:
  - `fromSource:` UInt - Source ID for this transaction.
  - `toAddress:` UInt - The address to write to.
  - `lgSize:` UInt - Base two logarithm of the number of bytes to be written.
  - `data:` UInt - The data to write on this beat.
  - `mask:` UInt - (optional) The write mask for this beat.
- Returns:
  - A (Bool, TLBundleA) tuple. The first item in the pair is a boolean indicating whether or not the operation is legal for this

AccessAck Method

- It is constructor for a TLBundleD encoding an `AccessAck` or `AccessAckData` message. If the optional `data` field is supplied, it will be an `AccessAckData`. Otherwise, it will be an `AccessAck`.
- Arguments:
  - `a:` TLBundleA - The A channel message to acknowledge
    `data:` UInt - (optional) The data to send back.
- Returns:
  - The TLBundleD for the D channel message

All different methods related to TileLink is define in Edge.scala file of RocketChip.

Other Methods

- `Arthimatic:` Constructor for a TLBundleA encoding an Arithmetic message, which is an atomic operation.
- `Logical:` Constructor for a TLBundleA encoding a Logical message, an atomic operation
- `AcquireBlock:` Constructor for a TLBundleA encoding an AcquireBlock message, which request data from memory.
- `AcquirePerm:` Constructor for a TLBundleA encoding an AcquirePerm message, which request data permission from memory.
- `Release:` Constructor for a TLBundleA encoding a Hint message, which is used to send prefetch hints to caches.

Other Methods

- `ProbeAck:` Constructor for a TLBundleD encoding a HintAck message.
- `Hint:` Constructor for a TLBundleA encoding a Hint message, which is used to send prefetch hints to caches.
- `HintAck:` Constructor for a TLBundleD encoding a HintAck message.
- `first:` Determines whether the current beat is the first beat in the transaction.
- `last:` Determines whether the current beat is the last beat in the transaction.
- `count:` Determines the count of the current beat in the transaction.
- `hasdata:` Determines whether the TileLink message contains data or not.

Diplomatic Connections

- `:=` This is the basic connection operator. This operator connects Diplomacy nodes and creates a single edge between the two nodes.
- `:=*` This is a "query" type connection operator. It can create multiple edges between nodes, with the number of edges determined by the client node.
- `:*=` This is a "star" type connection operator. It also creates multiple edges, but the number of edges is determined by the manager.
- `:*=*` This is a "flex" connection operator. It creates multiple edges based on whichever side of the operator has a known number of edges. This can be used in generators where the type of node on either side isn't known until runtime.

Diplomatic Widgets

- `TLXbar`
- `TLBuffer`
- `TLROM`
- `TLRAM`
- `TLWidthWidget`
- `TLFragmenter`
- `TLFIFOFixer`

Case Study

- TileLink in RISCV Mini.
- How to generate own library based on diplomacy "Diplomatic Adder"

References

ChipYard (2019).

Chipyard.

https://chipyard.readthedocs.io/en/latest/TileLink-Diplomacy-Reference/index.html.