Use Firrtl Transform to Control the Effective Range of ‘printf’ in Large Scale Circuits

Lin Jiawei

Institute of Computing Technology, Chinese Academy of Sciences

2021/6/26

source code

Use Firrtl Transform to Control the Effective Range of ‘printf’ in Large Scale Circuits

- ‘printf’ is a useful Chisel tool to debug circuits
  - ‘printf’ can be translated to ‘fwrite’ in Verilog
- But it will slow down the simulation
  - I/O overhead
  - We need ‘args’, which may affect dead code elimination (dce)

```
case class Print(info: Info, string: StringLit, args: Seq[Expression], clk: Expression, en: Expression)
```

Use Firrtl Transform to Control the Effective Range of ‘printf’ in Large Scale Circuits

- There may be many ‘printf’ in a large chisel design
- X GB/min logs may be generated by ‘printf’ statements in a large design
  - Hard to save
  - Hard to analysis
- We usually only need some small parts of all ‘printf’ statements
  - Example: Only BPU’s ‘printf’ in a large out-of-order processor when debugging branch predictors

Use Firrtl Transform to Control the Effective Range of ‘printf’ in Large Scale Circuits

- Can we disable/enable ‘printf’ in a specific module and its submodule by some parameters?
  - Firrtl transform can help us
  - Pass module names by annotations, then we can process them in a custom transform
  - No need to modify Chisel source code!

Implementation

- Define annotations
- Add command line options
- Write a custom Firrtl transform

Implementation

- We defined 4 types of annotations
  - DisablePrintfAnnotation
  - EnablePrintfAnnotation
  - DisableAllPrintAnnotation
  - RemoveAssertAnnotation
- Each annotation has a corresponding command line option

```
25  case class DisablePrintfAnnotation(m: String) extends NoTargetAnnotation
26  object DisablePrintfAnnotation extends HasShellOptions{
27
28    val options = Seq(
29      new ShellOption[String](
30        longOption = "disable-module-print",
31        toAnnotationSeq = s => Seq(DisablePrintfAnnotation(s)),
32        helpText =
33          "The verilog 'printf' in the <module> and it's submodules will be removed\n",
34        shortOption = Some("dm"),
35        helpValueName = Some("<module>")
36      )
37    )
38
39  }
```

```
41  case class EnablePrintfAnnotation(m: String) extends NoTargetAnnotation
42  object EnablePrintfAnnotation extends HasShellOptions {
43    val options = Seq(
44      new ShellOption[String](
45        longOption = "enable-module-print",
46        toAnnotationSeq = s => Seq(EnablePrintfAnnotation(s)),
47        helpText =
48          "The verilog 'printf' except the <module> and it's submodules will be removed\n",
49        shortOption = Some("em"),
50        helpValueName = Some("<module>")
51      )
52    )
53
54  }
```

Implementation

- We defined 4 types of annotations
  - DisablePrintfAnnotation
  - EnablePrintfAnnotation
  - DisableAllPrintAnnotation
  - RemoveAssertAnnotation
- Each annotation has a corresponding command line option

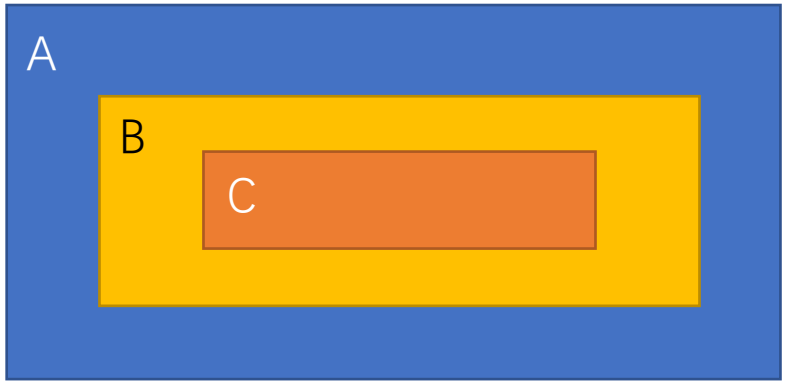
```
56  case class DisableAllPrintAnnotation() extends NoTargetAnnotation
57  object DisableAllPrintAnnotation extends HasShellOptions {
58    val options = Seq(
59      new ShellOption[Unit](
60        longOption = "disable-all",
61        toAnnotationSeq = _ => Seq(DisableAllPrintAnnotation()),
62        helpText =
63          "All the verilog 'printf' will be removed\n",
64        shortOption = Some("dall")
65      )
66    )
67  }
68
69  case class RemoveAssertAnnotation() extends NoTargetAnnotation
70  object RemoveAssertAnnotation extends HasShellOptions{
71    val options = Seq(
72      new ShellOption[Unit](
73        longOption = "remove-assert",
74        toAnnotationSeq = _ => Seq(RemoveAssertAnnotation()),
75        helpText = "All the 'assert' will be removed\n",
76        shortOption = None
77      )
78    )
79  }
```

Collect Annotations

```
86    override protected def execute(state: CircuitState): CircuitState = {
87
88      val disableList = state.annotations.collect {
89        case DisablePrintfAnnotation(m) => m
90      }
91      val enableList = state.annotations.collect {
92        case EnablePrintfAnnotation(m) => m
93      }
94      val disableAll = state.annotations.collectFirst {
95        case DisableAllPrintAnnotation() => true
96      }.nonEmpty
97      val removeAssert = state.annotations.collectFirst{
98        case RemoveAssertAnnotation() => true
99      }.nonEmpty
100
101     assert(!(enableList.nonEmpty && (disableAll || disableList.nonEmpty)))
```

Analyze Circuit

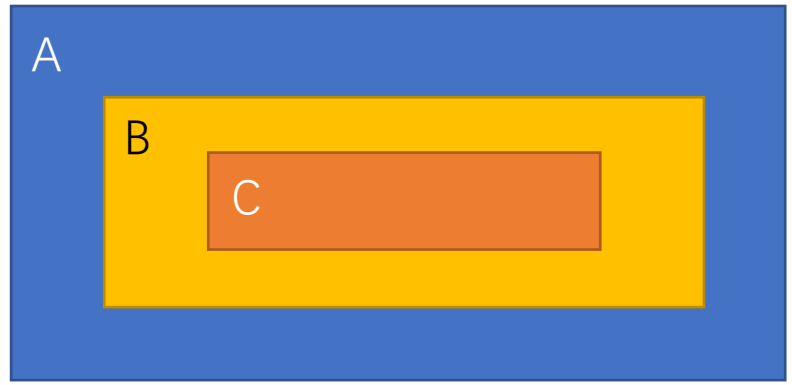
- ancestor(C) = {A, B}




```
val c = state.circuit

val top = c.main
val queue = new mutable.Queue[String]()
val ancestors = new mutable.HashMap[String, mutable.LinkedHashSet[String]]()

queue += top
ancestors(top) = mutable.LinkedHashSet.empty

while (queue.nonEmpty) {
  val curr = queue.dequeue()
  c.modules.find(m => m.name==curr).foreach(m => {
    def viewStmt(s: Statement): Statement = s match {
      case DefInstance(_, _, module, _) =>
        ancestors(module) = ancestors(curr) + m.name
        queue += module
        s
      case other =>
        other.mapStmt(viewStmt)
    }
    m.foreachStmt(viewStmt)
  })
}
```

Analyze Circuit

- **firrtl.analyses.CircuitGraph** is a better implementation
- Why not CircuitGraph?
  - CircuiteGraph works with MidForm
  - We want to run our transform with HighForm
  - We haven’t found a solution



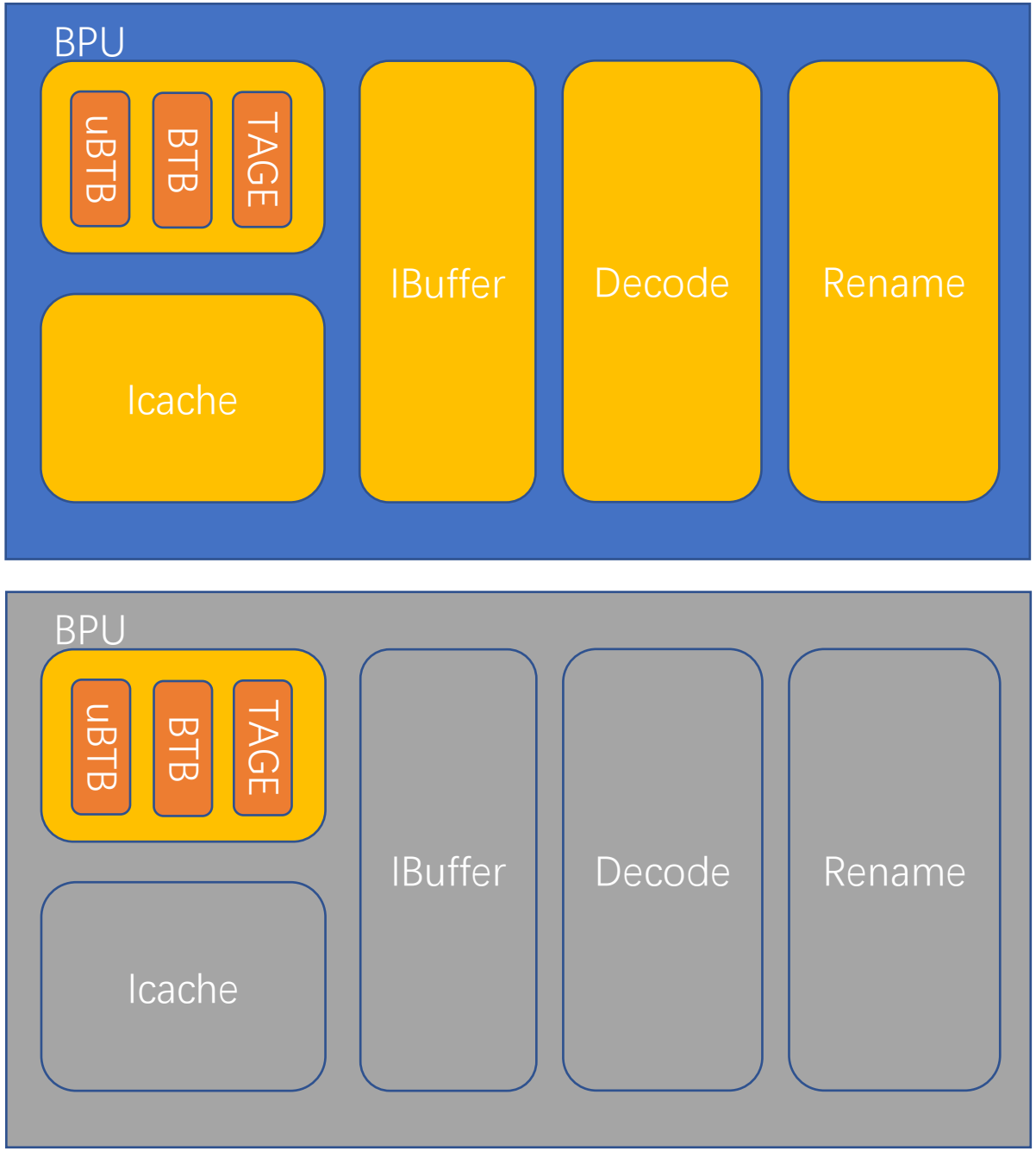
Modify Circuit

- For each module m
  - If m is in the range that needs to be disabled, change Print statements to Empty statements

```
def onModule(m: DefModule): DefModule = m match {
  case _: ExtModule => m
  case _: Module =>
    def inRange(seq: Seq[String]): Boolean = {
      seq.nonEmpty && (seq.contains(m.name) || seq.map(elm => {
        ancestors(m.name).contains(elm)
      }).reduce(_||_))
    }
    val enable = enableList.isEmpty || inRange(enableList)
    val disable = disableAll || inRange(disableList) || !enable
    def onStmt(s: Statement): Statement = s match {
      case _: Print if disable =>
        EmptyStmt
      case _: Stop if removeAssert => EmptyStmt
      case other => other.mapStmt(onStmt)
    }
    m.mapStmt(onStmt)
}
state.copy(circuit = c.mapModule(onModule))
```

Example usage

- Command line args: -em BPU
  - ‘em’ means **e**nable **m**odule print
- Only print statements in BPU and its submodule will be emitted to Verilog



Thank You!