

## 实验二： Chisel 编译流程介绍

### 1. 实验目的：

- 1.1. 掌握 Chisel 基本语法
- 1.2. 掌握 Chisel 从编译到生成 verilog 的过程
- 1.3. 能够使用 Chisel 编写简单的电路
- 1.4. 掌握 Chiseltest 的使用

### 2. 实验内容：

- 2.1. 以 3-8 译码器为例，学习 Chisel 语法
- 2.2. 学习实验一中 3-8 译码器的编译和仿真流程
- 2.3. 学习序列检测电路的设计，并尝试修改

### 3. 实验步骤：

#### 3.1. Chisel 基本语法

图 2.1 所示为 decoder 项目目录结构，主要有 4 个文件，之后会分别介绍其作用。

在 Linux 中想要查看文件时，可以使用 Linux 自带的编辑器 gedit 查看和编辑，在命令行中输入“gedit 文件名”即可打开 gedit 编辑器，如想查看 build.sc 文件，在命令行中输入“gedit build.sc”。

（在 Linux 中大部分时间使用的编辑器并非 gedit，而是另一个终端中的编辑器 vim，但 vim 的操作可能需要花比较多的时间学习）

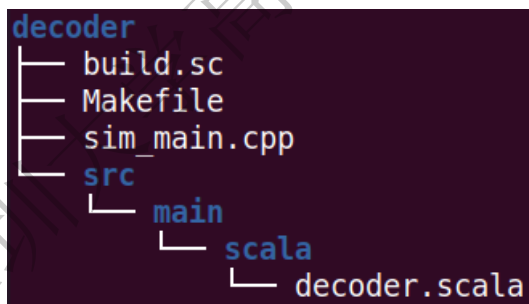


图 2.1 decoder 项目目录结构

#### 3.1.1. decoder.scala 文件

以下是 decoder 项目中最关键的 Chisel 代码，在 decoder 项目目录下的 src/main/scala/decoder.scala 中。

在学习 Chisel 基本语法前，需要先补充一些面向对象编程的基础知识。以学过的 C 语言为例，在编写 C 语言时，通常有一个主函数，从输入到输出，整个程序分为很多个步骤完成，其中主要使用的都是一些变量赋值和函数调用，这称为**面向过程编程**，这种编程思维在实现一些简单的问题时思路清晰，易于理解，但当要解决现实生活中具有更复杂的逻辑的问题时会显得混乱且繁琐，因此有了**面向对象编程**，在面向对象编程过程中有 2 个重要的概念，类(class)与对象(object)。类代表一种事物的抽象，而对象代表一个具体的事物。例如汽车是一个抽象的概念，而停在车库中的这辆车牌号为 12345 的车，单指某个

特定的事物，这就是一个对象，对象是一个类的具体化，而类是无数对象的抽象。在程序中也是类似的概念，我们可以把一个 int 整数类型看作一个类，这个类有一系列属性，例如所有的整数都可以做加减乘除计算，而定义一个 int 型的变量 a，则可以说 a 是 int 类的一个对象。在面向对象过程中，会定义许多的类，以及它们具有的属性 and 功能，通常用变量和函数来表示，而给某个类定义一个具体的变量，则生成了一个该类的对象，这个过程又叫做实例化。在 Chisel 代码中会大量使用到类和对象的定义。

```
decoder.scala
1 package decoder
2
3 import chisel3._
4 import chisel3.util._
5
6 // RawModule 与 Module 不同，它不会生成隐式的时钟，由于我们是组
合电路，因此现在还没有涉及到时钟的概念，只用 RawModule 即可
7 class Decoder extends RawModule {
8     val io = IO(new Bundle{
9         val in = Input(UInt(3.W)) // 输入的 0-7 信号，二进制表示
只需要 3bits 宽即可
10         val out = Output(UInt(8.W)) // 输出的译码后信号，8bits
宽
11     })
12
13     // 在 Chisel 中，可以对同一个变量多次赋值，以最后一次赋的值
为最终结果
14     // 因此如果下面的 switch 语句都没有执行，这一行能给 io.out 一个
默认值
15     io.out := "b00000000".U
16
17     switch (io.in) {
18         is (0.U) { io.out := "b00000001".U }
19         is (1.U) { io.out := "b00000010".U }
20         is (2.U) { io.out := "b00000100".U }
21         is (3.U) { io.out := "b00001000".U }
22         is (4.U) { io.out := "b00010000".U }
23         is (5.U) { io.out := "b00100000".U }
24         is (6.U) { io.out := "b01000000".U }
25         is (7.U) { io.out := "b10000000".U }
26     }
27 }
28
29 object testMain extends App {
30     // 实例化 Decoder 模块
```

```

31     Driver.execute(args, () => new Decoder)
32 }

```

在这个文件中包含了 Chisel 基本的模块定义的方式，接口的定义，以及赋值语句和 switch 语句的语法。在第 1 行，为这个文件定义了它属于哪个包 (package) 中，这里是 decoder，在多文件编程中，包可以用来划分不同模块的文件，这样可以确保在文件之间互相引用时能够准确找到需要引用的数据结构。第 2-3 行引入了一些 chisel 的基本语法和函数。由于 Chisel 是实现在 Scala 语言中的一门语言，因此整个源文件的后缀名是 .scala，也因此需要引用 Chisel 相关的包才能使用 Chisel 的语法。

第 7 行定义了一个 Decoder 类，它继承自一个 RawModule 类，也可以理解成 Decoder 属于一个 RawModule 类，RawModule 是 Chisel 自己定义的一个模块类，代表 Decoder 是一个硬件模块，RawModule 是模块的一种类型，它所有的接口都需要自定义，在本实验的后半部分会介绍另一种 Module 类，相比于 RawModule，它默认自带了 clock 和 reset 接口。

8-11 行是对这个模块接口的定义，第 8 行定义了一个常量叫做 io，注意这里使用的是定义常量的关键字 val，而不是我们常用的变量。这里的常量代表的并不是指模块的输入输出信号是固定值，而是指这个模块的输入输出接口的数据类型和位宽是固定的，这是硬件编程与软件编程不同的概念。因为在硬件设计中，芯片设计完成后，其中所有的元器件都是固定的，不可能在运行过程中动态的改变元器件的排列和连线。因此在 Chisel 中，大部分的寄存器和连线，都使用 val 关键字定义。

等号右边的 IO 是 Chisel 中的一个类，这个类需要接收一个 Bundle 作为参数，而这个 Bundle 中需要给出模块定义的接口信号，用于输入输出。8-11 行代码是为模块定义接口的一种标准语法格式。

在 Decoder 模块的 io 中，包含 2 个信号，分别是 in 信号，它是一个 3 位宽的无符号整数类型，是 Input 信号，从外部接收进模块内部，而 out 是一个 8 位宽的无符号整型，是 Output 信号，从模块内部传往外部。和 C 一样，在 Chisel 中也有一些基本数据类型，包括有符号整数，无符号整数，布尔值，区别在于，这些数据类型在定义时，需要显式地指定数据位宽，即需要使用几个 bits 能够表示。在软件编程中，int 类型默认都是一个 32bits 的数，但是在硬件设计中，接口位宽会直接影响模块间的连线数量，进而影响整个芯片的面积和功耗，因此不能够像软件那样直接使用相同的 32bits 定义所有的数据，因此需要显示的指定，例如 3-8 译码器的输入只有 3 位，对应的 Decoder 模块的 in 接口就是一个只有 3bits 的无符号整数，同样的 out 接口是一个只有 8bits 的无符号整数，也可以理解成 8 个布尔值。

表 2.1 是 Chisel 中常用的一些数据类型。其实在硬件中，所有的数据类型归根结底就是一串 bits 信号，不同之处只是在于如何看待它。无符号整型即把所有的 bits 都看作一个二进制整数，有符号整型则是将最高位看作符号位，其他位当作补码表示的二进制整数。布尔类型是 1 个 bit 的信号，高电平 1 代表 true，低电平 0 代表 false。

表 2.1 Chisel 基本数据类型

UInt(n, W)	n 位宽的无符号整型
------------	------------

SInt (n. W)	n 位宽的有符号整型，最高位作为符号位
Bool ()	布尔类型，只有 2 个值，true 或者 false

在 Chisel 中这些数据的写法也与 C 有所不同，布尔类型只有 true 和 false，在 Chisel 中分别写作 true.B 和 false.B，而有符号整型可以写作符号+字面值.S，例如-8.S，就代表值为-8 的有符号整数。无符号整数类似，只是不能为负数，例如 8.U，就代表值为 8 的无符号整数。除了十进制外，还可以使用其它进制表示，如表 2.2，分别列出了用十进制、二进制、八进制、十六进制表示值为 156 的无符号整数

表 2.2 值为 156 的无符号整型

十进制	156.U
二进制	"b10011100".U
八进制	"o234".U
十六进制	"h9c".U

在给出一个无符号整型或者有符号整型的具体值时，Chisel 会自动计算需要几个 bits 才能够保证完整表示这个数，例如 156 最少需要 8bits 才能完整表示，但是有时想要表示一个 8bits 宽，值为 4 的无符号整数时，需要显示指定位宽，例如 4.U(8.W)，否则 Chisel 会推断只需要 3 个 bits 即可表示 4 这个值。

第 15 行，为 io.out 接口赋了一个默认值，可以看出，out 的输出默认是全为 0。注意这里的赋值使用的是冒号等于号:=，而不是单独的等于号，这是由于在 Chisel 中，等于号是用来定义硬件单位的，例如之前的接口定义，使用的就是等于号，而冒号等于号才是用来给硬件单元赋值的语法。

17-25 行，则是译码器的核心逻辑了，这是一个 switch 语句，用法类似于 C 中的 switch 语句，不同之处在于它的一些书写格式，Chisel 中的 switch 书写格式如下：

```
switch (需要检测的信号 s) {
  is (state1) {
    // 当 s === state1 时执行代码
  }
  is (state2) {
    //当 s === state2 时执行代码
  }
}
```

Decoder 中的 switch 语法使用枚举的方式，将所有可能的 8 中输入都列举了出来，并且根据不同的输入，给 io.out 端口赋了对应的值，如此便完成了 Decoder 的核心逻辑。

最后的 29-31 行则是定义了一个名为 testMain 的 App 类型，mill 在编译时会检测到这个 App 类，并且以此为入口，执行里面的代码，31 行代码的含义就是新建了一个 Decoder 模块，作为希望 mill 编译生成 verilog 代码的模块。

掌握 Chisel 基本语法中模块和接口的定义，赋值语句之后，就可以完全理解 decoder.scala 中的代码实现逻辑了，在 decoder 项目中未使用的语法会随着之后的实验不断进行补充。

### 3.2. Chisel 编译与仿真流程

#### 3.2.1. build.sc 文件

build.sc 文件是 mill 的配置文件，主要用于配置 mill 编译项目时的依赖。

build.sc

```
1 import scalalib._
2
3 object decoder extends CrossSbtModule {
4   def crossScalaVersion = "2.12.10"
5
6   def ivyDeps = Agg(
7     ivy"edu.berkeley.cs::chisel3:3.3.2"
8   )
9 }
```

最基础的 mill 配置非常简单，第 1 行 import 代表引入 scalalib 相关的函数和类定义，第 3 行定义了一个名为 decoder 的对象，继承了 CrossSbtModule 类，这个主要是为了在编译时让 mill 知道项目名，mill 会按照规定好的路径找到需要编译的代码文件。

第 4 行和第 6、7 行都非常容易理解，分别指定了需要使用的 Scala 和 Chisel 版本。这些都是 mill 中固定的格式。

可以看出 build.sc 中并没有太多的内容，仅仅告诉 mill，有一个名为 decoder 的项目，并且指定了 Scala 和 Chisel 语言的版本。

#### 3.2.2. Makefile 文件

接下来看 Makefile 文件，这个文件是为使用 make 命令而编写的。

Makefile 是在 Linux 环境下项目开发必须要掌握的一个工程管理文件。当你使用 make 命令去编译一个工程项目时，make 工具会首先到这个项目的根目录下去寻找 Makefile 文件，然后才能根据这个文件中的代码去编译程序。

那么 Makefile 究竟做了什么呢？实际上 Makefile 指定了编译整个项目所有需要执行的命令，Makefile 主要用于简化项目的编译流程，代码如下：

（在熟悉了项目后，可以尝试在不使用 make 命令的情况下编译 decoder 项目）

Makefile

```
1 BUILD_DIR = ./build
2 TOP_V = $(BUILD_DIR)/Decoder.v
3 SCALA_FILE = $(shell find ./src -name '*.scala')
4
5 .DEFAULT_GOAL = verilog
```

```

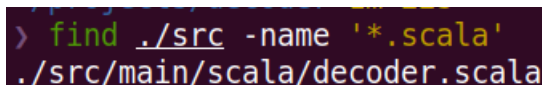
6
7 $(TOP_V): $(SCALA_FILE)
8     @mkdir -p $(@D)
9     mill decoder.run decoder.main.testMain -td $(@D) --output-
file $(@F)
10
11 verilog: $(TOP_V)
12
13 $(BUILD_DIR)/VDecoder.mk: $(TOP_V)
14     @mkdir -p $(@D)
15     verilator --cc --exe \
16         -o $(abspath $(BUILD_DIR)/emu) -Mdir $(@D)
17
18 $(BUILD_DIR)/emu: $(BUILD_DIR)/VDecoder.mk
19     $(MAKE) -C $(BUILD_DIR) -f ./VDecoder.mk
20
21 emu: $(BUILD_DIR)/emu
22
23 clean:
24     rm -rf build

```

Makefile 中 1-3 行代码定义了一些常用的变量，首先定义了 build 目录的位置，我们希望将编译生成的文件统一放在 build 目录中，避免和我们自己编写的代码文件混淆。在 Makefile 文件中，\$(变量名) 在执行时会被对应变量的值替换，例如第 2 行中的 \$(BUILD\_DIR)/Decoder.v，其实就是 ./build/Decoder.v。

第 3 行中，SCALA\_FILE 代表项目中所有的 scala 代码文件，等号右边其实也是一个变量，不过这个变量的值是运行一条命令的结果，例如这里，可以把 `find ./src -name '*.scala'` 当作终端命令输入，这条命令的含义是：查找所有在 src 目录中的后缀名为 .scala 的文件。如图 2.2 所示，可以看到它的运行结果就是 decoder.scala 文件，因此 SCALA\_FILE 就等于 decoder.scala 文件的路径。

在第 5 行定义了 Makefile 的默认目标，定义了默认的编译目标后，可以在终端直接执行 make 命令而不用加其他参数，即在 decoder 项目中，执行 make 命令和执行 make verilog 命令是相同的。



```

> find ./src -name '*.scala'
./src/main/scala/decoder.scala

```

图 2.2 在终端里查找后缀为 .scala 的文件

7-9 行就是编译 Chisel, 生成 Verilog 的关键命令, 首先看第 7 行, 以冒号分隔为左右两个部分, 左边是 TOP\_V 变量, 也就是 ./build/Decoder.v, 这是这段命令的目标文件, Decoder.v 是编译 Chisel 生成的 Verilog 文件, .v 是 verilog 文件的后缀名, 而右边是 SCALA\_FILE 变量, 也就是 decoder.scala 文件, 这一行的含义是: 这段代码用于生成 Decoder.v 文件, 生成这个文件需要依赖于 decoder.scala 文件, 因此在执行这段代码之前, 会去检测 SCALA\_FILE 是否已经生成。这里 SCALA\_FILE 是之前就定义好的变量, 因此接下来会执行 8-9 行的命令。第 8 行中有一个奇怪的变量 @D, 这个变量的含义是目标文件所在的目录, 也就是 ./build/Decoder.v 的目录 ./build, 另外还有 @F, 这个变量的含义是目标文件取出路径后的文件名, 也就是 Decoder.v。第 8 行命令在当前目录下新建一个 build 目录, 用于之后存放生成的文件。在 mkdir 命令前有一个单独的 @ 符号, 这种在命令前的 @ 符号的作用是在执行 make 命令编译时, 前面带 @ 号的命令不会显示在终端中。可以尝试去掉这个 @ 符号再编译, 观察有什么不同。

第 9 行调用了 mill 命令, 主要功能就是在 src 目录下所有的 scala 文件中找到主函数, 然后将其中的模块编译生成 verilog 文件, -td 参数指定了想要存放生成的 verilog 文件的目录, --output-file 参数指定了生成 verilog 文件的名称。

之后如果想生成 Decoder.v 文件, 就可以在终端中输入 make ./build/Decoder.v, 第 7-9 行的命令就会被执行, 之后的第 11 行代码, 相当于为 ./build/Decoder.v 定义了一个别名, 在输入 make verilog 命令时, Makefile 检测到需要 ./build/Decoder.v 的依赖, 然后再去调用 7-9 行的命令, 这样我们只要输入 make verilog 或 make 命令即可, 相比于原来的命令更加方便输入。

在 13-16 行代码, 定义了根据 Decoder.v 生成仿真文件的命令。首先与第 7 行类似, 13 行代码定义了一个目标文件 VDecoder.mk, 这其实也是一个 Makefile 文件, 不过是由 Verilator 自动生成的, 可以不用关心其中的内容, 只要知道运行它会编译生成一个用于仿真的可执行文件即可。而生成这个文件前, 需要先生成 Decoder.v 文件, 因此在冒号右边将 Decoder.v 文件作为依赖。14 行同第 8 行一样, 都是确保 build 目录存在。15-16 行代码就是使用 Verilator 来生成仿真文件了, 其中 -cc 参数代表将指定的 verilog 文件转为 C++ 代码输出, -exe 代表希望生成可执行的文件, -o 指定了最终生成的可执行文件的名字, 注意这里需要使用绝对路径。

什么是绝对路径呢? 这就涉及到在电脑中如何确定一个文件的位置, 在 decoder 项目目录下输入 pwd 命令, 可以看到当前目录的绝对路径: 例如 /home/lc3/projects/decoder, 它是从 Linux 文件系统的最顶层目录, 详细描述了如何一步步找到 decoder 目录的, 当我们在 decoder 目录下时, 可以不同输入完整的路径, 而直接用 “.” 代替 /home/lc3/projects/decoder, 而想要表达在 decoder 目录下的 build 目录中的 Decoder.v 文件时, 就有两种表达方式, 一种是 /home/lc3/projects/decoder/build/Decoder.v, 这就是绝对路径, 另一种是 ./build/Decoder.v, 这就是相对路径。其中 “.” 代表当前终端所在的目录位置, 我们可以使用 cd 命令切换当前终端所在的目录, 因此, 绝对路径无论终端在哪个目录下, 都可以准确的定位到某个文件, 而相对路径只有在某个确定的路径下, 才能够准确的定位到某个文件。



-Mdir 代表指定保存输出文件的路径，这里指定 build 目录，之后就是给出需要仿真的 verilog 文件，然后给出 Verilator 仿真的顶层文件 sim\_main.cpp，这个文件的内容接下来会作介绍。

这部分命令执行完之后，可以看到在 build 目录中会生成许多 C++ 文件，包括 .cpp、.h 头文件等，还有后缀为 .mk 的 Makefile 文件，其中 VDecoder.mk 文件是最重要的，运行它，即可生成可执行的文件。

在 18-19 行即是运行 VDecoder.mk 文件生成可执行文件。18 行中，目标文件是 ./build/emu，依赖于 13-16 行生成的 VDecoder.mk 文件，19 行中的 \$(MAKE) 是一个默认的变量，其实就是 make 命令，在 Makefile 中通常都是使用这个变量，而不直接使用 make 命令。-C 参数代表需要执行的 Makefile 文件所在的目录，这里是 ./build 目录，-f 指定需要使用的 Makefile 文件，也就是 VDecoder.mk 文件。

第 21 行给 ./build/emu 定义了一个别名 emu，之后只要直接执行 make emu 命令，就可以直接一键生成可执行文件了。

23-24 行定义了一个清除命令，其实就是在执行 make clean 时删除 ./build 目录。

### 3.2.3. sim\_main.cpp 文件

sim\_main.cpp 文件虽然是一个 C++ 文件，但是如果有 C 的基础是容易理解的。sim\_main.cpp 文件是 Verilator 在进行仿真时的一个顶层文件，也就是说，Verilator 仿真其实就是编译这个文件，然后运行这个文件中的 main 函数而已。sim\_main.cpp 的代码如下所示：

```
sim_main.cpp
1 #include "VDecoder.h" // 这个头文件会根据你模块的名字不同而改变
2 #include <verilated.h>
3 #include <iostream>
4 #include <bitset> // 用于输出二进制的数
5
6 using namespace std;
7
8 int main(int argc, char **argv, char **env) {
9     Verilated::commandArgs(argc, argv);
10    VDecoder* decoder = new VDecoder; // 模块的实例
11
12    int code = 0; // 用于 decoder 模块的输入，从 0-7 遍历
13
14    while (!Verilated::gotFinish() && code < 8) {
15        decoder->io_in = code;
16
17        decoder->eval(); // 每执行一次 eval 函数，就对 decoder 模
        块执行一次仿真
18        cout<<"in: "<<code<<"\t";
19        cout<<"out: "<<bitset<8>(decoder->io_out)<<endl; // 输
        出 deocder 模块的 out 接口的信号
```



```

20
21     code++;
22 }
23
24     decoder->final();
25     delete decoder;
26     exit(0);
27 }

```

1-4 行是 C 语言引用头文件，其中 VDecoder.h 头文件就是 Verilator 根据 verilog 文件生成的，可以在 build 目录中找到这个头文件，而 verilated.h 是 Verilator 自带的一个头文件，定义了一些基本操作，iostream 中定义了 cout 函数，类似于 C 中的 printf，bitset 主要是用在 cout 中，以二进制的格式输出数据。

第 6 行指定了使用的命名空间为 std，这个是 C++ 中特有的概念，这一行不加也可以，只是所有的 cout 函数要改为 std::cout，对程序整体没什么影响。

第 8 行是 main 函数的定义，第 9 行是向 verilator 中传入一些参数，可以不用关心，10-26 行开始，才是最重要的部分，第 10 行定义了一个 VDecoder 模块的变量 decoder，这个 VDecoder 就是 Chisel 代码中定义的模块。在第 12 行定义了一个变量 code，这个是由于之后给 decoder 传入输入信号的。可以看到在 15 行，给 decoder 的 io\_in 端口赋值为 code，并且 21 行在每次循环结束 code 变量都会递增。其中 14 行是一个仿真过程中主要的循环，当 verilator 仿真没有结束，并且 code 小于 8 的时候不断循环，code 小于 8 是因为 3-8 译码器输入只有 3 位，只能表示 0-7 的数，因此在遍历完 0-7 所有情况后，就可以退出仿真了。

17 行 eval 函数就是对 decoder 模块进行仿真，每执行一次，都会去更新 decoder 中所有信号的状态。而 18-19 行分别是输出 decoder 接口的 in 信号和 out 信号。

最后 24-25 行，在仿真结束后调用 final 函数和 delete 关键字，释放 decoder 占用的资源。然后 26 行退出程序。

#### 3.2.4. 小结

decoder 项目中的 4 个文件至此已经介绍完了，由此可以总结得出 decoder 项目的整体开发到编译到仿真的流程：首先使用 build.sc 文件，指定 mill 使用的 Scala 和 Chisel 语言版本，然后用 Chisel 编写主要的 decoder.scala 代码，用 mill 编译生成对应的 verilog 代码，最后用 verilator 对 verilog 代码进行仿真，运行生成的仿真文件。为了避免每次编译都输入复杂的命令，编写了 Makefile 文件，用于简化编译的流程。

### 3.3. 序列检测电路设计

介绍完 3-8 译码器的电路设计后，再介绍另一个简单电路：序列检测电路。在介绍序列检测电路之前，首先需要学习一些相关的基础知识，时钟，以及时序电路和组合电路的区别。

时钟，应该称为时钟脉冲，是一个按一定电压幅度，一定时间间隔连续发出



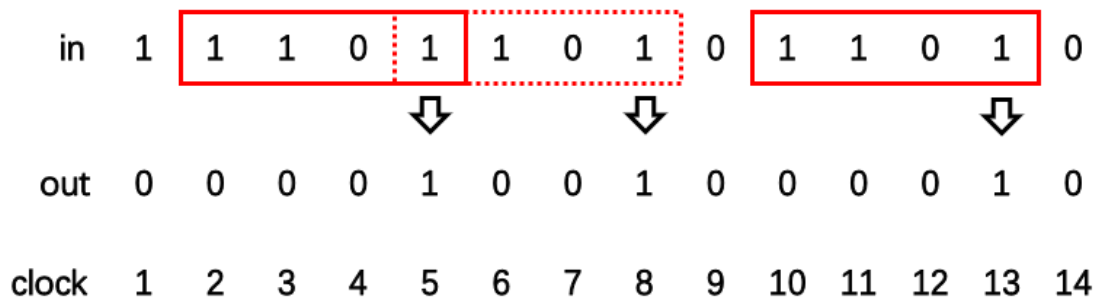


图 2.4 1101 序列检测示例

想要检测跨 4 个时钟周期的 bit 序列，必然要设法保存之前的输入，可以思考一下，如果使用 C 完成这个功能，如何实现？最简单的方法就是把过去 3 个周期的输入全部记录下来，然后和当周期的输入拼在一起，看是否需要检测的序列。然而实际上有更简洁的实现，就是使用有限状态机 (finite state machine, FSM)，即表示有限个状态以及在这些状态之间的转移和动作等行为的模型，状态机的下一个状态由当前状态和当前输入共同决定。根据功能需求，可以设置以下几个状态，并画出图 2.5 所示的状态转移图。

S0: 代表之前输入的信号为 0，当检测到输入信号为 1 时，进入 S1，否则维持 S0。

S1: 代表之前输入的信号为 1，当检测到输入信号为 1 时，进入 S2，否则回到 S0。

S2: 代表之前输入的信号为 11，当检测到输入信号为 1 时，维持 S2，否则进入 S3。

S3: 代表之前输入的信号为 110，当检测到输入信号为 1 时，检测到目标序列，输出高电平，同时进入 S1，否则回到 S0。

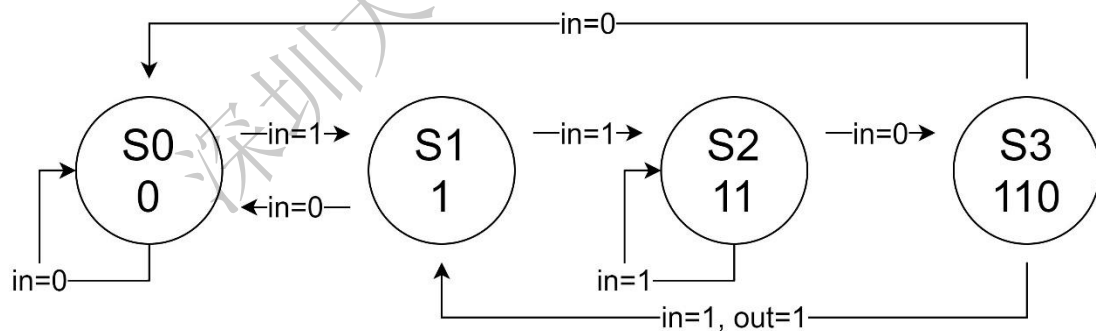


图 2.5 状态转移图

接下来学习序列检测的相关代码，进入 `detection` 项目目录下，这个项目的结构和 `decoder` 项目十分相似，但是每个文件都略有区别。

在 `detection` 项目的 `build.sc` 中，可以看到多了许多代码，其中大部分是添加了一些新的依赖，来保证 Chisel 代码的正常编译，具体原因不用深入研究。此外还添加了一个名为 `test` 的对象，这是用于之后会介绍到的单元测试所需要的相关依赖。

`detection` 项目的 `Makefile` 文件改动不大，主要是将项目名称由 `decoder` 改

为了 **detection**。

主要看 **detection** 项目下的 `src/main/scala/detection.scala` 代码，如下所示：

```
detection.scala
1 package detection
2
3 import chisel3._
4 import chisel3.util._
5
6 class Detection extends Module {
7     val io = IO(new Bundle{
8         val in = Input(Bool())
9         val out = Output(Bool())
10    })
11
12    val S0 = 0.U(2.W) // 0
13    val S1 = 1.U(2.W) // 1
14    val S2 = 2.U(2.W) // 11
15    val S3 = 3.U(2.W) // 110
16
17    val stat = RegInit(0.U(2.W))
18
19    switch(stat) {
20        is (S0) { when(io.in) {stat := S1} .otherwise {stat :=
S0} }
21        is (S1) { when(io.in) {stat := S2} .otherwise {stat :=
S0} }
22        is (S2) { when(io.in) {stat := S2} .otherwise {stat :=
S3} }
23        is (S3) { when(io.in) {stat := S1} .otherwise {stat :=
S0} }
24    }
25
26    printf(p"in = ${io.in}, out = ${io.out}\n")
27    io.out := stat === S3 && io.in
28 }
29
30 object testMain extends App {
31     Driver.execute(args, () => new Detection)
32 }
```

首先值得注意的是第 6 行，**Detection** 模块不同于 **Decoder** 模块，**Decoder** 模块是一个 **RawModule** 类型，而 **Detection** 模块是一个 **Module** 类型，**RawModule** 和 **Module** 的不同在于，**Module** 模块默认的接口中会有一个 **clock**

时钟信号和一个 reset 信号，而 RawModule 不会有，作为一个时序逻辑电路，时钟信号是必须的，因此 Detection 使用的是 Module 而不是 RawModule，时钟信号在仿真时需要模拟生成一个时钟，接入到 Detection 模块的 clock 接口上。

8-9 行，定义了 Detection 模块的接口，除了隐式定义的 clock 和 reset 信号，还有一个 in 和一个 out 信号，这两个都是 Bool 类型，因为只有 1bit，当然定义成 UInt(1.W)也是完全可以的，硬件上不会有任何区别。

12-15 行定义了 4 个状态的常量，位宽 2 位就足以表示 S0 到 S3 了。

17 行定义了一个状态寄存器，寄存器是电路设计中，用来暂存一些数据的小型存储器件。在序列检测中，我们使用一个 2bits 的寄存器来保存电路当前所在的状态。Detection 中使用的是 RegInit 函数，它会根据给出的初始值生成保存对应数据类型的寄存器，这里状态的初始值是 0，对应 S0 状态。

需要注意的是，不同于软件编程，硬件编程中，寄存器的赋值并不是立即生效，给寄存器赋值后，新的值要在下个周期才会被更新，当周期寄存器独处的数据依然是赋值之前的旧值。且每周期只能给寄存器赋一次值，如果有多条给同一个寄存器赋值的语句，则会以最后一条赋值语句为准。

19-23 行代码是 Decoder 模块中使用过的 switch 语句，根据 stat 当周期不同的状态，有着不同的状态转移逻辑。

其中的 when(条件 1) {...}.elsewhen(条件 2) {...}.otherwise {...} 语句，类似于 C 中的 if(条件 1) {...} else if(条件 2) {...} else {...} 语句。可以将 20-23 行的状态转移逻辑与图 2.5 对照，将状态转移图和 Chisel 代码结合起来理解。

26 行则是 Chisel 中自带的用于仿真调试的 printf 语句，它的使用方式与 C 中的略微不同，用 \${变量名} 来表示想要输出的变量值。

27 行则是为 io.out 赋值的语句，当状态为 S3，且输入的信号为高电平时，则表示检测到了目标序列，输出高电平，注意这里判断相等使用的不是 C 中的 == 号，而是三个连续的等于号 ===，这是 Chisel 中的固定语法，Chisel 中的不等于用 != 来表示，其他的逻辑运算符与 C 基本相同。

detection 项目的 sim\_main.cpp 文件也有改动，代码如下所示：

sim\_main.cpp

```
1 #include "VDetection.h" // 这个头文件会根据你模块的名字不同而改变
2 #include <verilated.h>
3 #include <iostream>
4 #include <bitset> // 用于输出二进制的数
5
6 using namespace std;
7
8 int main(int argc, char **argv, char **env){
9     Verilated::commandArgs(argc, argv);
10    VDetection* detection = new VDetection; // 模块的实例
11
12    int main_time = 0;
13    int seq_ptr = 0;
14    int seq[] = {1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0};
15
```

```

16     while (!Verilated::gotFinish() && seq_ptr < 14) {
17
18         if ((main_time % 10) == 1) {
19             detection->clock = 1;
20         }
21         if ((main_time % 10) == 6) {
22             detection->clock = 0;
23
24             seq_ptr = seq_ptr + 1;
25         }
26
27         detection->io_in = seq[seq_ptr];
28
29         detection->eval();
30
31         main_time++;
32     }
33
34     detection->final();
35     delete detection;
36     exit(0);
37 }

```

主要观察相比于 **decoder** 项目有改动的部分，12 行代码定义了一个变量 **main\_time**，代表仿真过程中的时间，主循环每迭代一次，**main\_time** 递增 1。

13-14 行，定义了用于送给模块输入接口的信号序列，有一个指针 **seq\_ptr**，**seq\_ptr** 每周期递增 1。

16 行中，仿真结束的条件改为了当 **seq\_ptr** 等于 14 时，代表所有的输入序列都输入完成，可以结束仿真。

18-24 行，模拟实现了一个时钟，可以看出，一个时钟周期对应 **main\_time+10**，一个周期内会对 **detection** 的 **clock** 接口赋 2 次值，0，1 交替赋值。

尝试运行 **make emu**，并且执行 **./build.emu**，能够看到如图 2.6 所示的输出。

```
> ./build/emu
in = 1, out = 0
in = 1, out = 0
in = 1, out = 0
in = 0, out = 0
in = 1, out = 1
in = 1, out = 0
in = 0, out = 0
in = 1, out = 1
in = 0, out = 0
in = 1, out = 0
in = 1, out = 0
in = 0, out = 0
in = 1, out = 1
in = 0, out = 0
```

图 2.6 detection 项目仿真输出

### 3.4. 单元测试设计

在 3-8 译码器和序列检测的模块设计中，对模块功能的测试都是通过修改 `sim_main.cpp` 的 `main` 函数实现的，在实际项目开发过程中，无法为了单一模块的测试，而去频繁地改动仿真的顶层文件，因此接下来介绍 chisel 官方的测试与验证库：`chiseltest`。可以看到在 `detection` 项目目录下，新增了一个 `src/test/scala/DetectionTest.scala` 文件，这其中就是专门为 `Detection` 模块编写的单元测试，在学习具体代码之前，可以先运行

`mill -i detection.test.testOnly detection.DetectionTest`

命令，尝试运行这个单元测试，如果测试通过，可以在终端看到图 2.7 所示的输出。

```
DetectionTest:
DetectionTest
in = 1, out = 0
in = 1, out = 0
in = 1, out = 0
in = 0, out = 0
in = 1, out = 1
in = 1, out = 0
in = 0, out = 0
in = 1, out = 1
in = 0, out = 0
in = 1, out = 0
in = 1, out = 0
in = 0, out = 0
in = 1, out = 1
in = 0, out = 0
- should test Detection
```

图 2.7 Detection 模块单元测试运行结果



在 `build.sc` 文件中，可以看到在 `detection` 对象内，又定义了一个 `test` 对象，继承了一个 `Test` 类和一个 `ScalaTest` 类，其中给出了希望使用的 `chiseltest` 版本和 `scalatest` 版本，这样 `mill` 在编译时，就能够从代码中找到相对应的单元测试。

单元测试的主要内容在 `src/test/scala/DetectionTest.scala` 文件中，以下是 `DetectionTest.scala` 的代码：

```
DetectionTest.scala
1 package detection
2
3 import chisel3._
4 import chiseltest._
5 import org.scalatest.flatspec.AnyFlatSpec
6
7 class DetectionTest extends AnyFlatSpec with
  ChiselScalatestTester {
8   behavior of "DetectionTest"
9
10  it should "test Detection" in {
11    test(new Detection) { c =>
12      var seq_in = Array(1,1,1,0,1,1,0,1,0,1,1,0,1,0)
13
14      // 检测 1101 序列的参考输出
15      var seq_out = Array(0,0,0,0,1,0,0,1,0,0,0,0,1,0)
16
17      // 检测 1011 序列的参考输出
18      // var seq_out = Array(0,0,0,0,0,1,0,0,0,0,1,0,0,0)
19
20      for (i <- 0 until seq_in.length) {
21        c.io.in.poke(seq_in(i).B)
22        c.io.out.expect(seq_out(i).B)
23        c.clock.step() // 执行一个时钟周期的仿真
24      }
25    }
26  }
27 }
```

需要注意的是 `DetectionTest.scala` 文件中，并不是模块的具体实现，而是单元测试，因此实际上这是一个软件程序，使用的是 `Scala` 语言，某些地方与 `Chisel` 语法是不同的，例如 `Scala` 中的整数和 `Chisel` 中的无符号整型不是同一类型的数据。在介绍单元测试时，也会介绍 `Scala` 相关的基础语法。

单元测试的代码相对比较简单，1-5 行首先是定义了包名，引入了一些必要的包，第 7 行定义了单元测试模块的名称，继承了单元测试必须的两个类。

12-25 行是单元测试的核心逻辑，首先定义了一个用来保存测试输入的数组 `seq_in`，注意这里使用的是 `Scala` 语法，因此定义的是变量，`Scala` 中定义变量使用关键字 `var`，`Array` 是定义数组的关键字，括号中就是测试输入的序列。15 行

用同样的方法定义了一个数组 `seq_out`，其中保存的是测试的参考输出。

在 20-23 行中，是一个 `for` 循环，定义了一个循环变量 `i`，由 0 到 14 递增（不包括 14），14 就是 `seq_in` 数组的长度，用 `seq_in.length` 来表示。Scala 中的 `for` 循环有 2 种写法，一种是 `for (x <- a until b) {}`，另一种是 `for (x <- a to b) {}`，它们的主要区别在于，使用 `until` 关键字，循环变量不包括 `b`，使用 `to` 关键字，循环变量包括 `b`，即 20 行代码改为 `for (i <- 0 to seq_in.length-1)`，逻辑上是相同的。

21-23 行使用 `c.io.接口名.poke(传入信号)`，来指定本周期传给测试模块输入接口的信号，使用 `c.io.接口名.expect(参考信号)` 来将本周期测试模块输出接口的值与参考信号的值相比较，如果相同则继续测试，否则单元测试出错终止，并输出错误信息。在 Scala 中，需要访问一个数组中第 `n` 个元素时，使用的是小括号，例如 `seq_in(5)`，而不是 C 中的中括号。使用 `c.clock.step()` 表示令测试模块执行 1 个周期的仿真，也可以在括号中添加参数，如 `c.clock.step(n)` 来令测试模块执行 `n` 个周期的仿真。

单元测试代码写完后，在项目目录下运行 `mill -I 项目名.test.testOnly 项目名.单元测试类名`，即可运行对应的单元测试。

### 3.5. 练习

参考检测 1101 序列的 `detection` 模块，实现一个检测 1011 序列检测的模块 `detection2`，画出新的状态转移图，并在 `detection` 模块基础上对 `detection.scala` 文件中的 Chisel 代码进行修改，需要通过单元测试。单元测试代码中也给出了检测 1011 序列的参考输出，测试时注释掉 1101 序列的参考输出，使用 1011 序列的参考输出即可。

**实验二-任务一：**根据实验指导，画出 1011 序列检测电路的状态转移图，完成对序列检测电路的修改，并且能够通过单元测试