目录

[1 文件Base.ts 2](#_Toc185605690)

[1.1 Module类 3](#_Toc185605691)

[1.1.1 constructor 3](#_Toc185605692)

[1.1.2 基础方法 3](#_Toc185605693)

[addInterface 3](#_Toc185605694)

[**addSignal** 3](#_Toc185605695)

[**addRegister** 3](#_Toc185605696)

[**addSequentialAlways** 3](#_Toc185605697)

[**addOperation** 3](#_Toc185605698)

[**addCombAlways** 3](#_Toc185605699)

[**addAssign** 3](#_Toc185605700)

[1.1.3 方法addInterface 3](#_Toc185605701)

[1.1.4 方法addSubmodule 3](#_Toc185605702)

[1.1.5 方法addSystemVerilogSubmodule 6](#_Toc185605703)

[2 文件Registers.ts 8](#_Toc185605704)

[3 文件SRAM.ts 9](#_Toc185605705)

[3.1 SRAM类 9](#_Toc185605706)

[3.2 SRAM\_Parameters 接口 9](#_Toc185605707)

[4 专利 10](#_Toc185605708)

1. 文件Base.ts

Base.ts这个文件是一个TSSV库，提供了一组用于构建和操作SystemVerilog代码的类和函数。它是为了简化HDL代码的生成而设计的，特别是针对那些使用模块化方法来设计复杂数字电路的场景。以下是该文件的主要内容概述：

* 导入fs模块：使用Node.js的fs模块来同步读取文件。
* IntRange类型：一个泛型类型，用于指定整数范围。
* ParameterValue类型：用于参数值的联合类型，可以是多种数据类型。
* TSSVParameters接口：用于定义模块参数的对象。
* SVEnum接口：用于定义SystemVerilog枚举类型的接口。
* baseSignal接口：定义了信号的基本属性。
* PortDirection类型：信号方向的枚举类型。
* IOSignal接口：继承自baseSignal，添加了direction属性。
* IOSignals类型：信号集合的类型定义。
* Sig类：表示TSSV信号的容器类。
* Expr类：表示TSSV表达式的容器类。
* Signal接口：继承自baseSignal，添加了value属性。
* Signals类型：信号记录的类型定义。
* OperationIO接口：定义了二元操作的输入输出接口。
* BinaryOp枚举：定义了二元操作的类型。
* Interface类：用于定义信号接口的类。
* Module类：是所有TSSV模块的基类，提供了丰富的方法来添加信号、子模块、接口以及生成SystemVerilog代码。
* serialize和deserialize函数：用于序列化和反序列化对象，特别是处理BigInt类型的数据。
* default导出：包含Module、Sig、Expr类的默认导出。
* 文件中还包含了大量的注释，提供了对代码中每个部分的详细说明。

整体来看，这个库提供了一套面向对象的方法来创建和操作SystemVerilog代码，支持参数化、信号定义、模块实例化和连接等硬件设计的基本元素。

* 1. Module类
     1. constructor
     2. 基础方法

addInterface

**addSignal**

**addRegister**

**addSequentialAlways**

**addOperation**

**addCombAlways**

**addAssign**

* + 1. 方法addInterface
    2. 方法addSubmodule

函数的主要功能是将一个子模块（submodule）添加到父模块中，并绑定输入/输出信号和接口。这和 Chisel 中的Module类似，Chisel 中的模块可以通过Module()的实例化添加子模块，并使用io接口进行端口的绑定。

下面是它们在功能上的对比：

1. 1. \*\*模块实例化\*\*

TSSV:

this.addSubmodule(instanceName, submodule, bindings, autoBind);

这里通过addSubmodule函数来实例化子模块submodule`，并将端口绑定信息通过bindings进行传递。

\*\*Chisel\*\*:

```scala

val mySubmodule = Module(new Submodule)

```

在 Chisel 中，模块实例化通过Module(new Submodule)来完成。

1. 2. \*\*端口绑定\*\*

\*\*TSSV\*\*:

使用bindings字典来指定父模块和子模块之间的信号连接，并自动绑定或扩展宽度。

```TSSV

if (autoBind) {

// 自动绑定逻辑

}

```

\*\*Chisel\*\*:

在 Chisel 中，端口绑定是通过io接口进行手动或自动连接的：

```scala

mySubmodule.io.in := io.in

mySubmodule.io.out := io.out

```

1. 3. \*\*宽度扩展\*\*

\*\*TSSV\*\*:

在autoWidthExtension参数为true的情况下，代码会根据输入/输出信号的宽度自动进行符号扩展或零扩展：

```TSSV

if (autoWidthExtension && (thisPort.direction === 'input')) {

// 宽度扩展逻辑

}

```

\*\*Chisel\*\*:

在 Chisel 中，宽度扩展可以通过UInt或SInt的asUInt和asSInt方法手动完成。宽度不匹配时，可以使用.padTo或.resize进行调整：

```scala

mySubmodule.io.in := io.in.padTo(16) // 自动扩展

```

1. 4. \*\*接口绑定\*\*

\*\*TSSV\*\*:

支持子模块接口绑定，通过submodule.interfaces和this.interfaces进行接口的匹配和创建。

\*\*Chisel\*\*:

Chisel 通过Bundles来实现复杂的信号捆绑，并在不同模块间传递。

1. 5. \*\*错误处理\*\*

\*\*TSSV\*\*:

错误处理通过throw Error实现，比如当绑定宽度不匹配或接口未绑定时抛出异常。

\*\*Chisel\*\*:

Chisel 中没有类似的异常机制，错误通常在仿真或综合时通过工具报告。

1. 总结

TSSV 中的addSubmodule函数显式地处理模块和接口的绑定、宽度扩展和错误检测，而 Chisel 中的模块和端口连接机制更多是依赖 Scala 语言的特性，结合硬件生成器来进行更高层次的描述。

* + 1. 方法addSystemVerilogSubmodule

函数读取一个 SystemVerilog 文件，将文件中的 I/O 端口与现有的绑定信息进行匹配，然后将其封装成一个Module对象，并将其添加为子模块。Chisel 中实现类似功能的方法主要依赖于模块层次结构、参数化设计和 I/O 接口的灵活连接机制。我们可以从以下几个方面进行对比：

1. 读取和解析 SystemVerilog文件： TSSV使用正则表达式来解析SystemVerilog文件，获取模块名称和 I/O 信息。在 Chisel 中，由于它是基于 Scala 的 DSL 并且模块定义通常是代码生成的一部分，Chisel 不直接解析Verilog 文件，而是通过ChiselStage的emitVerilog方法生成Verilog。
2. \*\*参数化模块\*\*

在 TSSV 中，params用于传递模块的参数，类似于 Chisel 中的 Parameters。

在 Chisel 中，可以通过参数化类（如Bundle`）传递模块的参数。例如，在 Chisel 中定义模块时，参数化常常通过构造函数传递：

```scala

class MyModule(val width: Int) extends Module {

val io = IO(new Bundle {

val in = Input(UInt(width.W))

val out = Output(UInt(width.W))

})

}

```

与此类似，TSSV 中的params被传递给SystemVerilog模块，在 Chisel 中通过 Params来定制模块行为。

1. 3. \*\*I/O 绑定\*\*

在 TSSV 中，代码通过bindings和正则表达式检测是否为input或output`，然后将其绑定到现有信号。

在 Chisel 中，I/O 是通过IO(new Bundle)来声明，并通过 Scala 语言的赋值操作进行连接。例如：

```scala

io.out := io.in

```

Chisel 通过:=来连接不同的信号，

而在这段 TSSV 代码中，绑定是通过bindings和findSignal函数来处理的。

1. 4. \*\*自动绑定和宽度扩展\*\*

在这段代码中，`autoBind机制允许自动处理绑定信号，并进行位宽扩展或截断。在 Chisel 中，类似的功能可以通过UInt和SInt自动处理位宽不匹配：

```scala

io.out := io.in.asUInt.pad(newWidth)

```

Chisel 会根据需要自动扩展或截断位宽，通过pad或asUInt方法。

1. 5. \*\*模块实例化\*\*

TSSV 中通过this.addSubmodule将 SystemVerilog 模块实例化为子模块。

在Chisel中，模块实例化也很简单，直接在模块中声明并使用：

```scala

val submodule = Module(new MySubModule)

submodule.io.in := io.in

io.out := submodule.io.out

```

1. 总结：

TSSV 中的addSystemVerilogSubmodule方法与 Chisel 中的模块化设计理念相似，都是通过参数化、自动绑定和 I/O 连接来实现模块实例化。

不同的是，Chisel 不直接解析外部 Verilog 文件，而是通过生成 Verilog 的方式与现有设计工具链集成。

1. 文件Registers.ts

Field Definition:

Registers.ts: Allows defining fields within registers using objects:

fields: {

fieldName: {

bitRange: [31, 24],

reset: BigInt('0xFF')

}

}

Chisel: Uses bit extraction and concatenation for field manipulation:

val field = myReg(31, 24)

Registers.ts这段代码是库的一部分，用于定义和操作硬件描述语言中的Register Block。它提供了一套API来创建寄存器块，并与总线接口进行交互。以下是代码概述：

1. **RegisterBlock类**：继承自**Module**类，用于创建寄存器块。它接受参数和寄存器定义，提供了一系列方法来添加信号、接口和逻辑。
2. **导入依赖**：从tssv/lib/core/TSSV导入了Module类和相关的类型定义，从tssv/lib/interfaces/Memory导入了Memory接口。
3. **类型定义**：
   * RegisterType：定义了寄存器的类型，如只读（RO）、可读写（RW）、只写（WO）、随机存取存储器（RAM）和只读存储器（ROM）。
   * Field：定义了寄存器字段的结构，包括位范围、是否有符号等。
   * Register：定义了寄存器的结构，包括类型、复位值、描述、大小、宽度、是否有符号和字段。
4. **RegAddr类**：用于生成寄存器地址，支持步长和自动递增。
5. **RegisterBlockDef接口**：定义了寄存器块的定义接口，包括字大小、地址映射、基地址、寄存器定义。
6. **RegisterBlockParameters接口**：定义了寄存器块参数的接口，包括总线接口类型、字节序、总线ID宽度、总线地址宽度。
7. **构造函数**：初始化寄存器块，设置I/O信号，添加内存接口，并根据寄存器定义创建信号和逻辑。
8. **寄存器逻辑**：根据寄存器类型（RW、RO、WO、ROM、RAM）创建相应的逻辑和信号。
9. **序列化和反序列化函数**：serialize和deserialize函数用于处理BigInt类型的数据的序列化和反序列化。
10. **辅助函数**：replaceZerosWithX、padZeroes、padZeroesRight、calculateDecMask和calculatePassMask是用于处理二进制地址和掩码的辅助函数。
11. **WaveDrom图表**：代码中包含了用于生成时序图的WaveDrom JSON格式数据，用于可视化写入和读取操作的时序。

整体来看，这个库提供了一套面向对象的方法来创建和操作硬件描述语言中的寄存器块，支持参数化、信号定义、模块实例化和连接等硬件设计的基本元素。

1. 文件SRAM.ts
   1. SRAM类

SRAM 类继承自 Module 类，并实现了以下功能：

* 参数声明: 在构造函数中，定义了 SRAM 的参数并设置默认值。
* 信号定义: 根据 ports 的配置（'1rw'、'2rw'、'1r\_1w'），定义了 SRAM 的输入输出信号。
* 内存数组: 使用 this.addSignal 创建一个存储 SRAM 数据的信号数组 mem，它的宽度是 dataWidth，深度是 depth。
* 写使能掩码: 根据 writeEnableMask 的不同配置，处理不同的写使能掩码宽度，并生成相应的 Verilog 代码。
* 读写行为: 根据 readBehavior 的不同配置，生成不同的读写冲突处理代码。
  1. SRAM\_Parameters 接口

这个接口定义了SRAM模块的配置参数：

* dataWidth: SRAM 数据宽度，范围从 1 位到 256 位。
* depth: SRAM 中数据字的数量，用 bigint 表示，以支持较大的深度。
* ports: SRAM 的端口配置，可以是 '1rw'（一个读写端口）、'1r\_1w'（一个读端口和一个写端口）、'2rw'（两个读写端口）。
* writeEnableMask: 写使能掩码控制，可以是 'none'（无掩码）、'bit'（逐位掩码）、'byte'（字节掩码）。
* readBehavior: 定义在同一时钟周期内读写同一地址时的行为，默认为 'undefOnWrite'（写时读数据不确定），可以选择 'readBeforeWrite'（先读后写）。
* macroConfig: 一个自由格式的字符串参数，用于选择技术特定的 SRAM 宏生成器参数（如高速、高密度配置等）。只会影响ASIC view。

1. 专利

**专利标题：** **面向对象编程**驱动的一种用于硬件模块设计和验证的软件系统，应用**多态**、**封装**和**接口管理方法，**提高开发硬件代码生产力

~~支持错误检查和Verilator注释~~

**背景技术：**随着硬件设计和验证的复杂性增加，特别是在大规模集成电路设计和验证的背景下，对**硬件模块**的生成、管理和操作变得越来越复杂，如FIFO、寄存器块、SRAM模块等。传统的硬件描述语言（HDL）如Verilog和VHDL已广泛应用于寄存器块的定义，但这些方法通常缺乏灵活性和模块化特性。**~~Verilator注释~~**~~与HDL工具的集成通常也存在一些局限性，无法有效提高设计和验证过程中的生产力。~~因此，亟需一种更高效、可扩展的方法，特别是能够支持基于**TypeScript**的设计语言，并通过**面向对象编程**（OOP）概念来简化硬件模块的操作与管理。与此同时，提供**TypeScript API**生成**Verilator注释**，可以提升验证效率和准确性。

**发明内容：** 本发明提供了一种基于TypeScript语言的硬件模块设计方法，通过OOP的方式，将**硬件模块**的定义、操作和管理抽象为一系列**TypeScript API**。该方法实现了**硬件模块**的扩展、集成、错误检查~~自动化管理~~，并能有效提高设计人员在**设计和验证**中的生产力。该方法包括以下几个主要功能：

**模块化定义**

本发明通过继承Module类，衍生了常用的子类，如RegisterBlock、SRAM、SRAM、FIR等类。该类能够灵活地与总线接口（如AXI、APB接口）进行交互，并支持模块的基本信息定义，包括地址、类型、字段等。~~寄存器块支持自动化地址分配、字段类型定义等功能。~~

**错误检查功能**  
在各硬件模块管理过程中，本发明引入了查找信号、IO、参数，用于检查信号和参数、逻辑、子模块的定义是否正确，引入了创建补全信号、IO位宽对齐等方法，提升集成子模块效率。例如，检查寄存器的地址是否重复、字段类型是否正确、信号名称是否符合规范等。这样可以避免传统手动检查的繁琐，提高设计过程的准确性和效率。

**高效的字符串处理与开发效率**  
本发明通过**TypeScript**语言来实现寄存器块的管理，**TypeScript**具有高效的字符串处理能力，能够简洁地处理寄存器字段的动态创建、替换和修改。通过对字符串的灵活操作，设计人员能够快速修改寄存器的定义、名称和字段，从而提高开发模块的生产力。