国立大学法人 電気通信大学 The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

「Course」 RISC-V Computer System Integration

「Lecture 6」 Rocket Computer System: Custom Hardware with External IOs

Hoang Trong Thuc

2022/12

国立大学法人 電気通信大学 The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

Outline

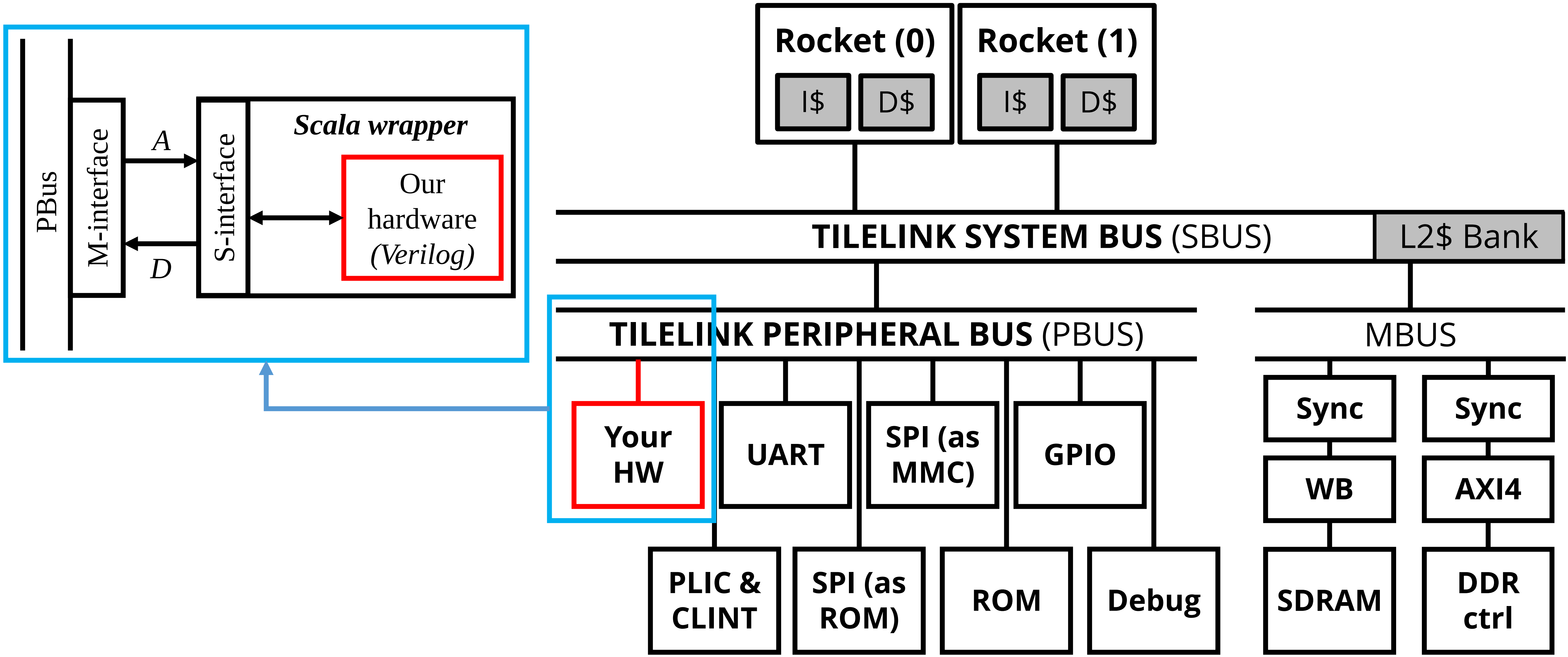
1. Last lecture brief review
2. Include external IOs
3. Practice: adder-subtractor with switch control

Outline

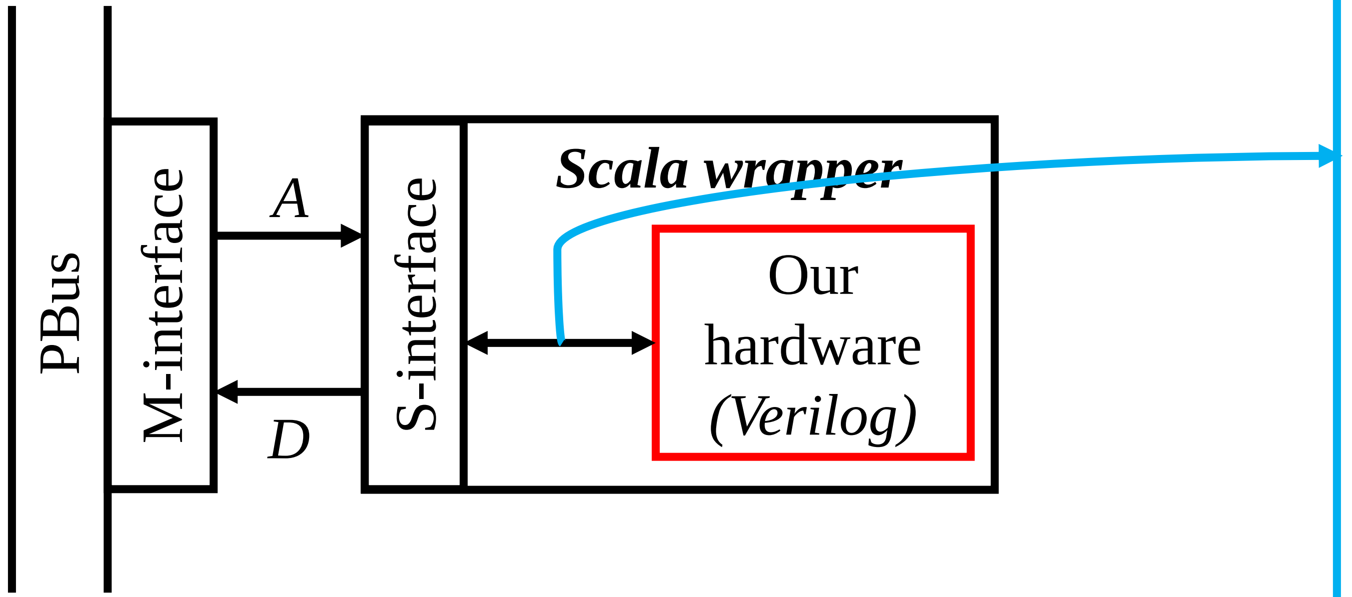
1. Last lecture brief review
2. Include external IOs
3. Practice: adder-subtractor with switch control

1. **Last lecture brief review** (1/10) The end goal

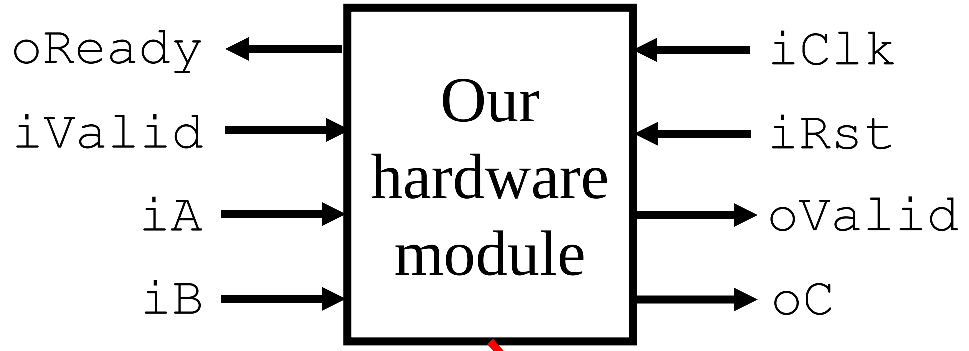
We add our custom hardware to the system's peripheral bus



1. Last lecture brief review (2/10) Custom hardware's signals



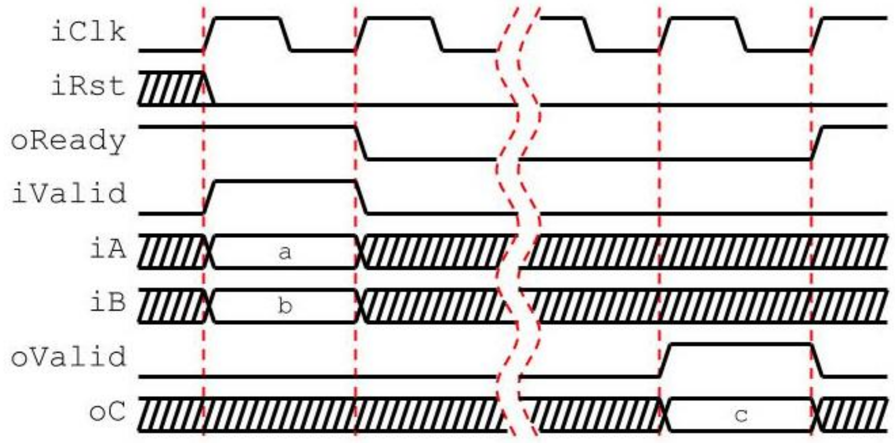
For a very simple case, iA and iB as inputs and oC as output. Then, our hardware's signals will look like this:



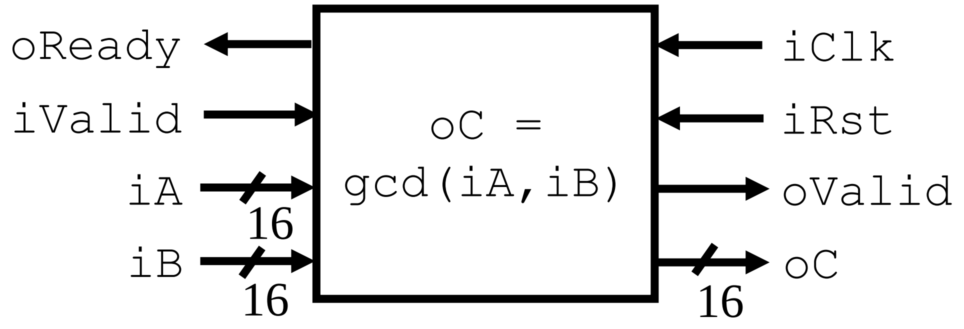
GCD pseudo-code:

```
Function: c=gcd(a,b)
BEGIN
  while(b!=0); do
    if(a>b)  swap(a,b)
    else     b=b-a
  done
  c=a
END
```

GCD waveform:

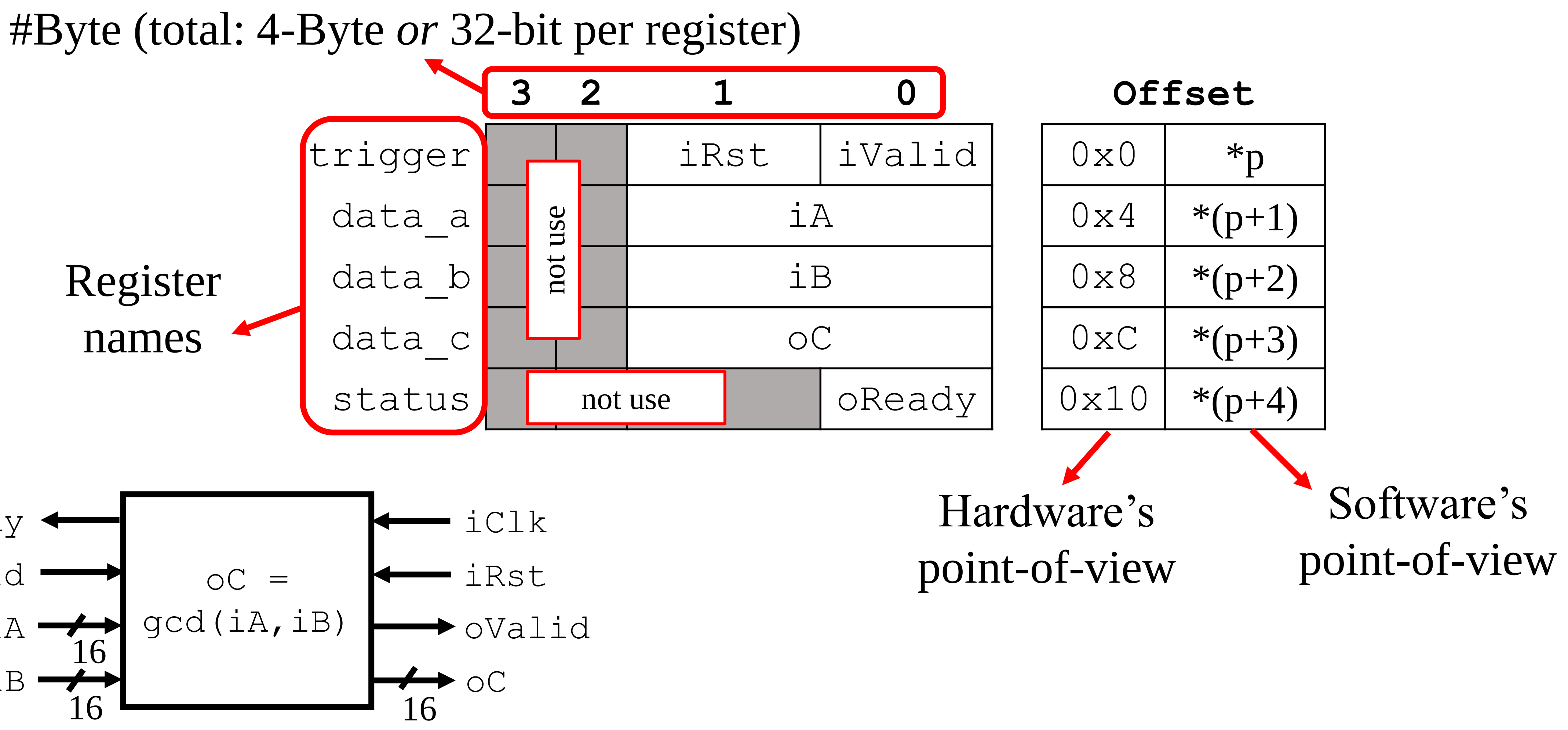


GCD was chosen to be the example:



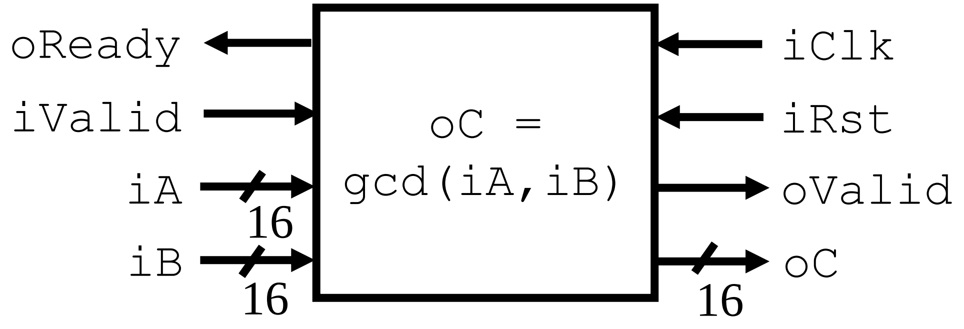
1. Last lecture brief review (3/10) GCD register-map

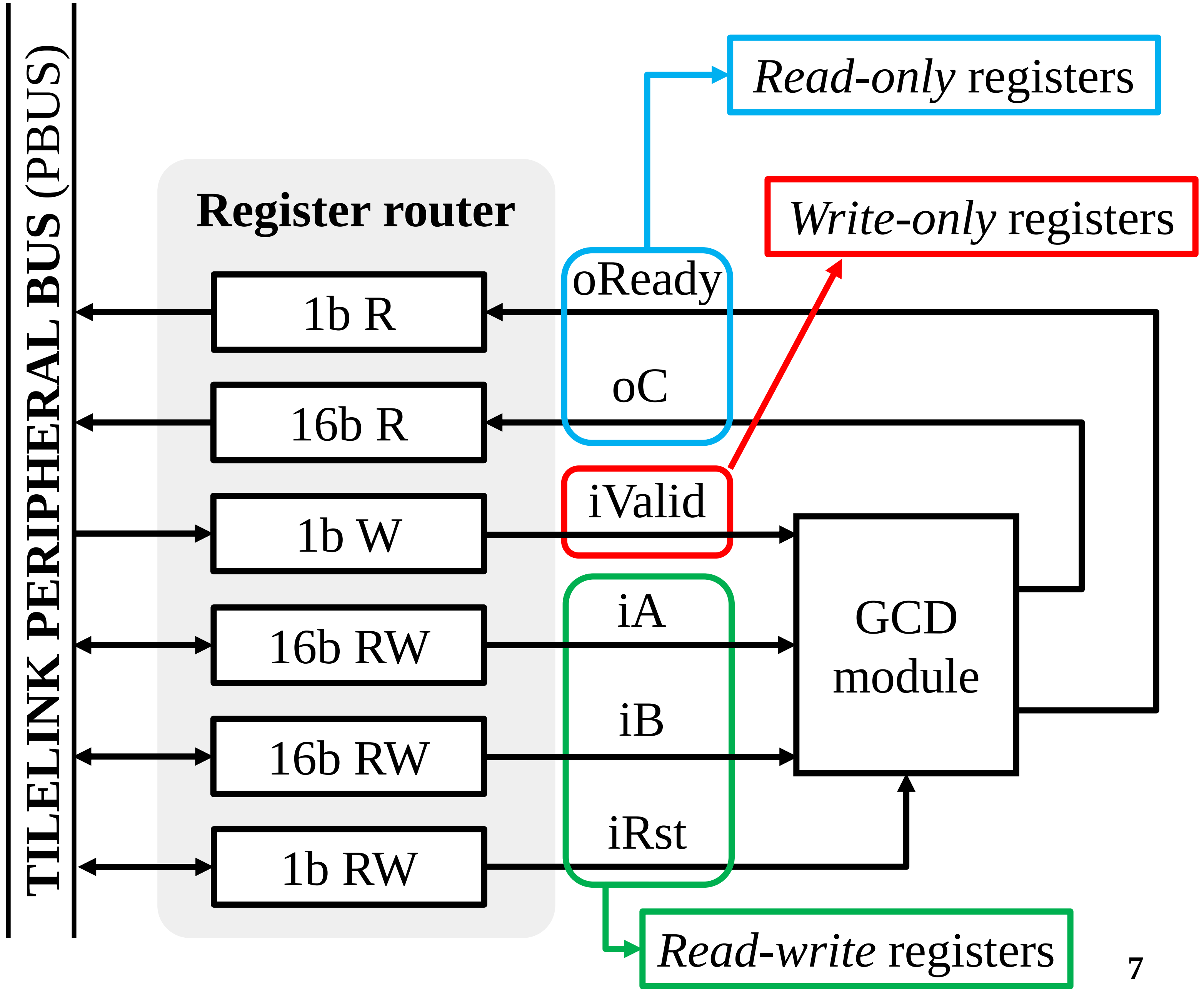
With the GCD module, the memory-register map can be chosen like this:



1. Last lecture brief review (4/10) Register-router

	3	2	1	0
trigger	▨	▨	iRst	iValid
data_a	▨	▨	iA	
data_b	▨	▨	iB	
data_c	▨	▨	oC	
status	▨	▨	▨	oReady

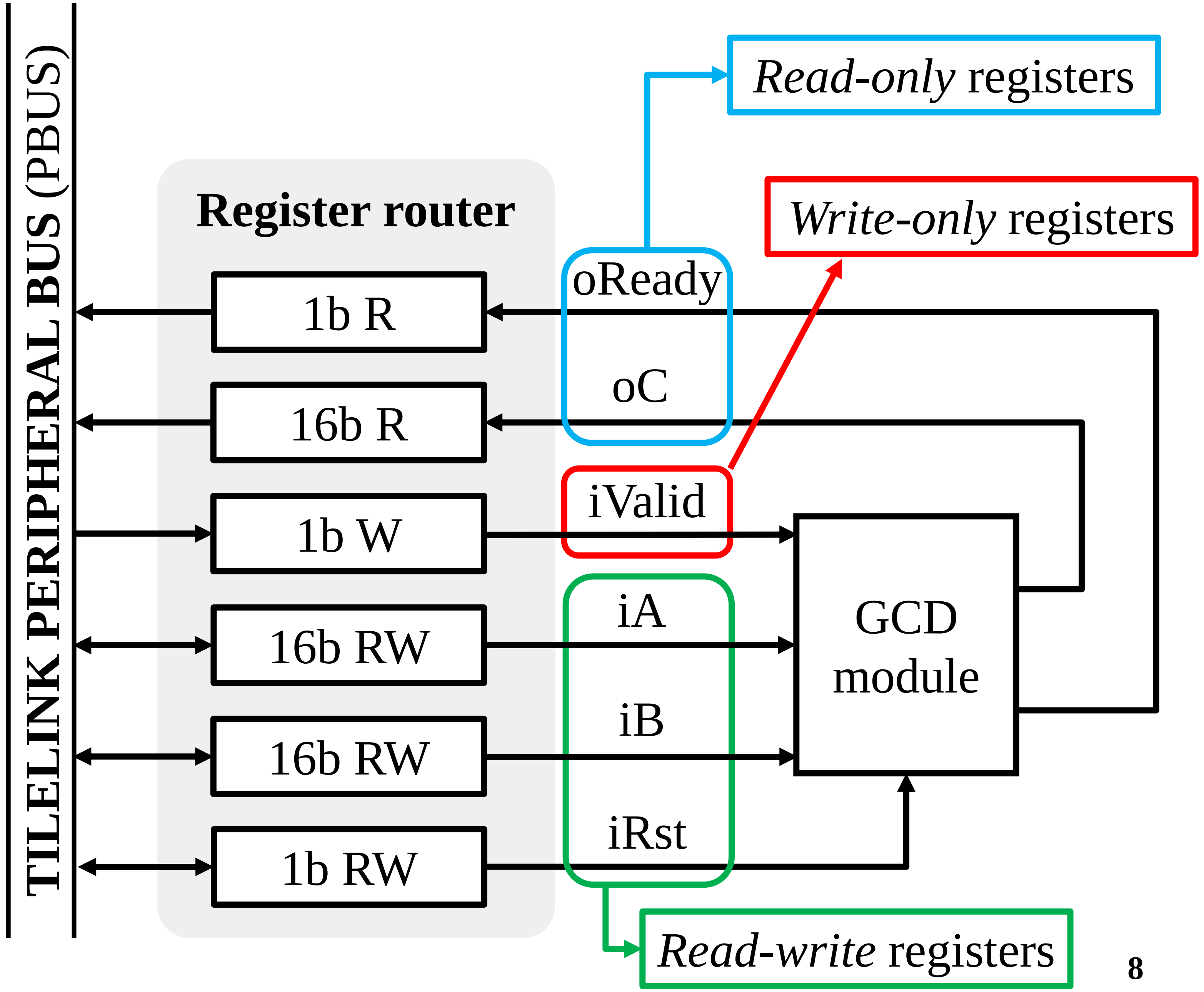




1. Last lecture brief review (5/10) Software example

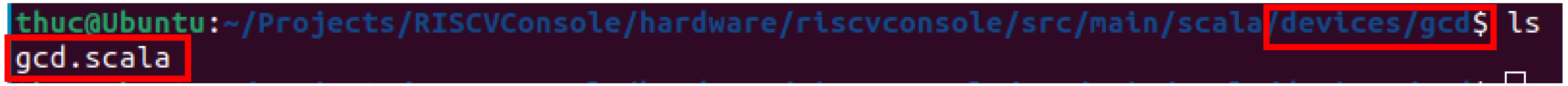
Software example

```
// Reset the hardware
_REG32(gcd_reg, GCD_TRIGGER) =
0x00000100;
_REG32(gcd_reg, GCD_TRIGGER) =
0x00000000;

// Write A & B values
_REG32(gcd_reg, GCD_DATA_A) = 7;
_REG32(gcd_reg, GCD_DATA_B) = 3;

// Trigger and wait
_REG32(gcd_reg, GCD_TRIGGER) =
0x00000001;
while(!(_REG32(gcd_reg,
GCD_STATUS) && 0x00000001));

// Get and print the result
int c = _REG32(gcd_reg,
GCD_DATA_C);
kprintf("GCD of 7 and 3 is %d\n",
c);
```



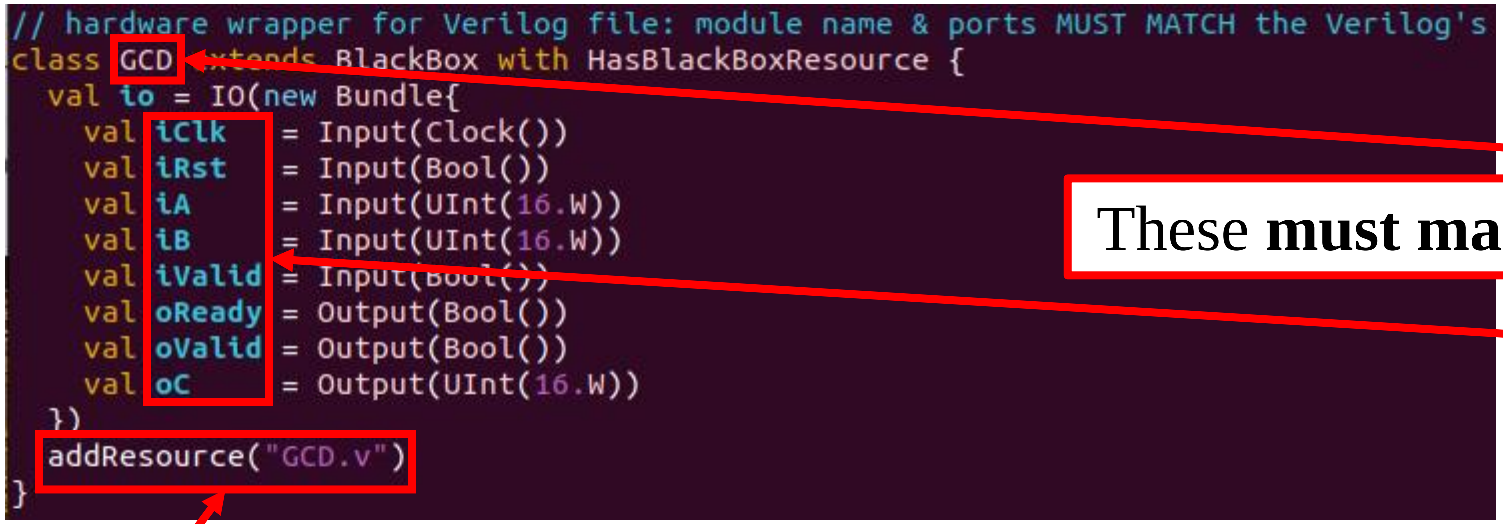
1. Last lecture brief review (6/10) GCD Scala file

The ***Scala wrapper file*** must be created under the `devices/` folder:



```
thuc@Ubuntu:~/Projects/RISCVConsole/hardware/riscvconsole/src/main/scala/devices/gcd$ ls
gcd.scala
```

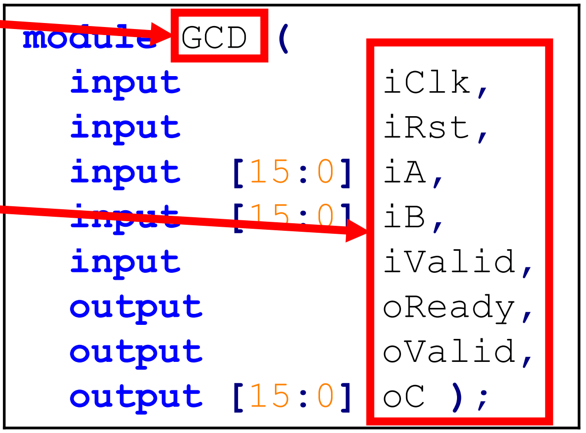
In the **`gcd.scala`** file, make a "fake" class that contains only the ports:



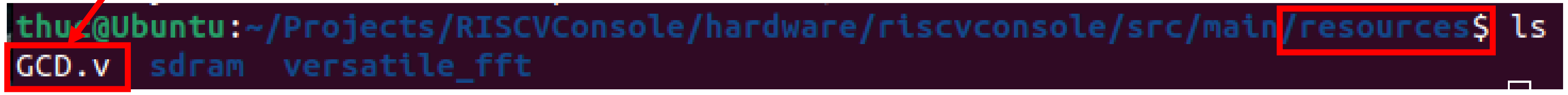
```
// hardware wrapper for Verilog file: module name & ports MUST MATCH the Verilog's
class GCD extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle{
    val iClk   = Input(Clock())
    val iRst   = Input(Bool())
    val iA     = Input(UInt(16.W))
    val iB     = Input(UInt(16.W))
    val iValid = Input(Bool())
    val oReady = Output(Bool())
    val oValid = Output(Bool())
    val oC     = Output(UInt(16.W))
  })
  addResource("GCD.v")
}
```

These **must match**

In **Verilog** file:



```
module GCD (
  input          iClk,
  input          iRst,
  input  [15:0]  iA,
  input  [15:0]  iB,
  input          iValid,
  output         oReady,
  output         oValid,
  output [15:0]  oC );
```

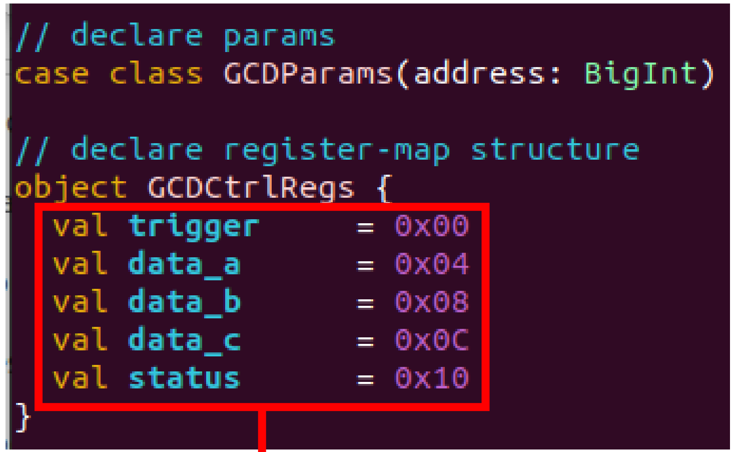


```
thuc@Ubuntu:~/Projects/RISCVConsole/hardware/riscvconsole/src/main/resources$ ls
GCD.v  sdram  versatile_fft
```

The `addResource()` will automatically find the given file-name in the `resources/` folder

1. Last lecture brief review (7/10) Wires & regs declaration

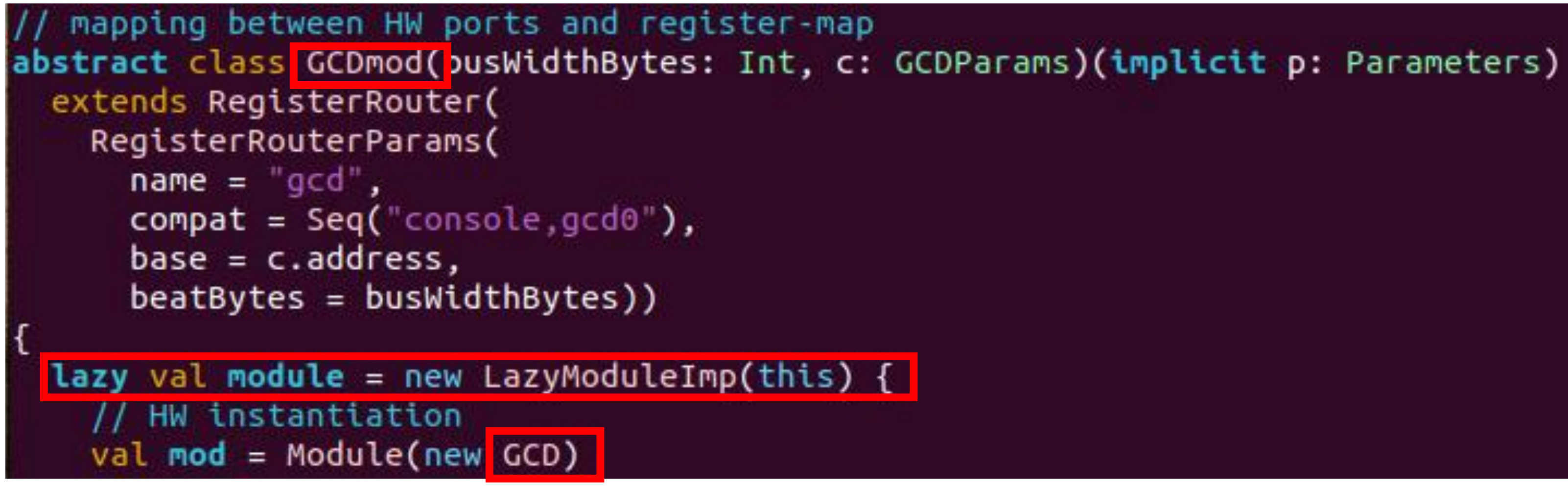
In the `gcd.scala` file:



```scala
// declare params
case class GCDParams(address: BigInt)

// declare register-map structure
object GCDCtrlRegs {
  val trigger       = 0x00
  val data_a        = 0x04
  val data_b        = 0x08
  val data_c        = 0x0C
  val status        = 0x10
}
```

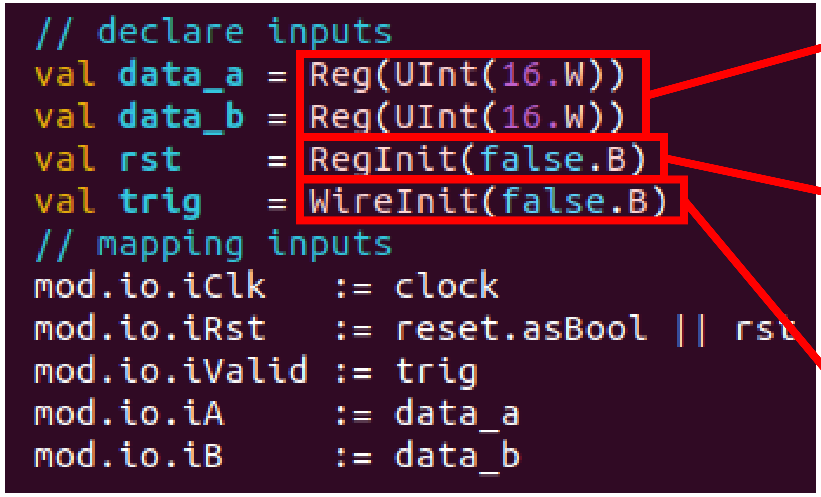
Declare registers with its offset.

Create a *Scala wrapper* called **GCDmod** that includes the previous **GCD** class as sub-module.



```scala
// mapping between HW ports and register-map
abstract class GCDmod(busWidthBytes: Int, c: GCDParams)(implicit p: Parameters)
  extends RegisterRouter(
    RegisterRouterParams(
      name = "gcd",
      compat = Seq("console,gcd0"),
      base = c.address,
      beatBytes = busWidthBytes))
{
  lazy val module = new LazyModuleImp(this) {
    // HW instantiation
    val mod = Module(new GCD)
```

Declare the ***wires*** and ***regs*** in the `GCDmod` and connect them to the `GCD` sub-module:



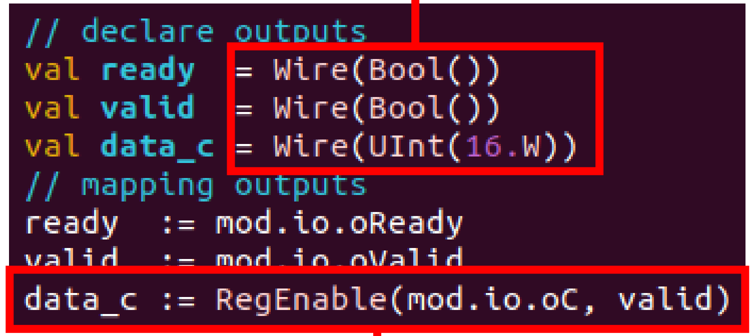
```scala
// declare inputs
val data_a = Reg(UInt(16.W))
val data_b = Reg(UInt(16.W))
val rst    = RegInit(false.B)
val trig   = WireInit(false.B)
// mapping inputs
mod.io.iClk    := clock
mod.io.iRst    := reset.asBool || rst
mod.io.iValid  := trig
mod.io.iA      := data_a
mod.io.iB      := data_b
```

Normal ***regs***

reg with init value of `0`

wire with init value of `0`

All outputs must be ***wires***



```scala
// declare outputs
val ready   = Wire(Bool())
val valid   = Wire(Bool())
val data_c  = Wire(UInt(16.W))
// mapping outputs
ready   := mod.io.oReady
valid   := mod.io.oValid
data_c  := RegEnable(mod.io.oC, valid)
```

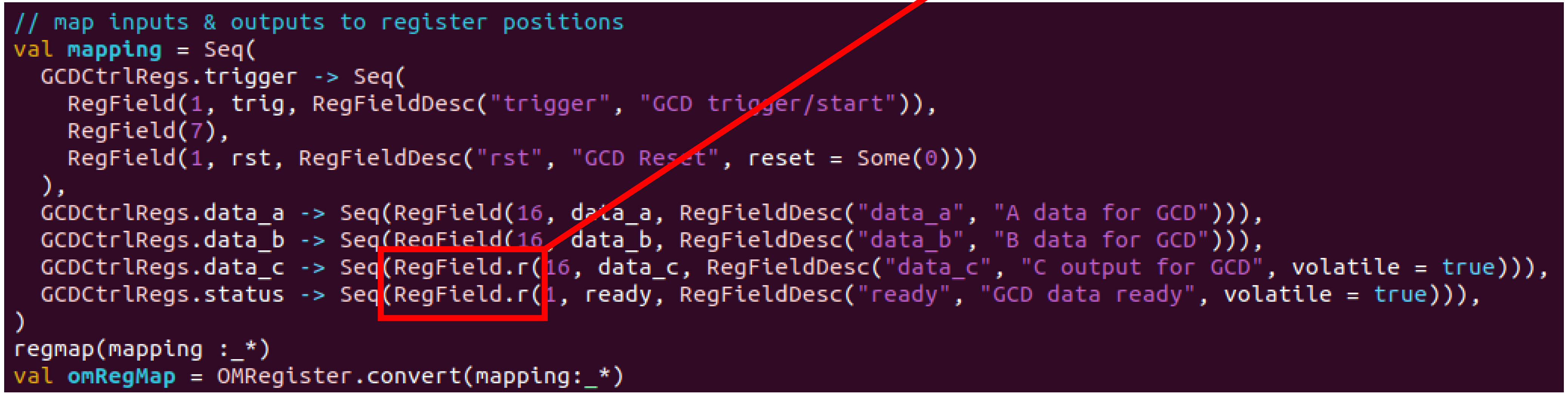
`data_c` captures `oC` at `valid`

1. Last lecture brief review (8/10) Map to registers

In the `gcd.scala` file:

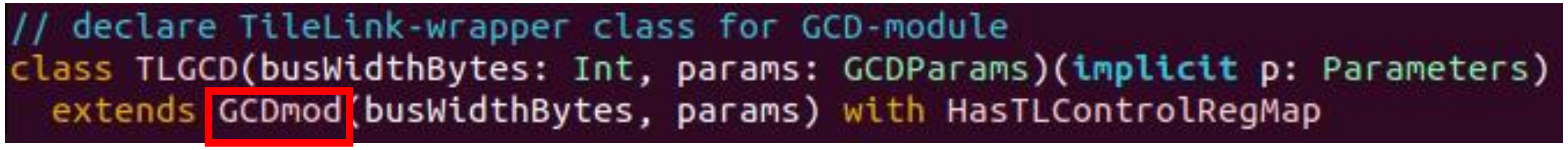
`RegField.r` instead of `RegFiled` means **read-only**.

Mapping between signals and registers:



```
// map inputs & outputs to register positions
val mapping = Seq(
  GCDCtrlRegs.trigger -> Seq(
    RegField(1, trig, RegFieldDesc("trigger", "GCD trigger/start")),
    RegField(7),
    RegField(1, rst, RegFieldDesc("rst", "GCD Reset", reset = Some(0)))
  ),
  GCDCtrlRegs.data_a -> Seq(RegField(16, data_a, RegFieldDesc("data_a", "A data for GCD"))),
  GCDCtrlRegs.data_b -> Seq(RegField(16, data_b, RegFieldDesc("data_b", "B data for GCD"))),
  GCDCtrlRegs.data_c -> Seq(RegField.r(16, data_c, RegFieldDesc("data_c", "C output for GCD", volatile = true))),
  GCDCtrlRegs.status -> Seq(RegField.r(1, ready, RegFieldDesc("ready", "GCD data ready", volatile = true))),
)
regmap(mapping :_*)
val omRegMap = OMRegister.convert(mapping:_*)
```

Finally, create a ***TileLink wrapper*** called **TLGCD** that extends from the previous **GCDmod** class:

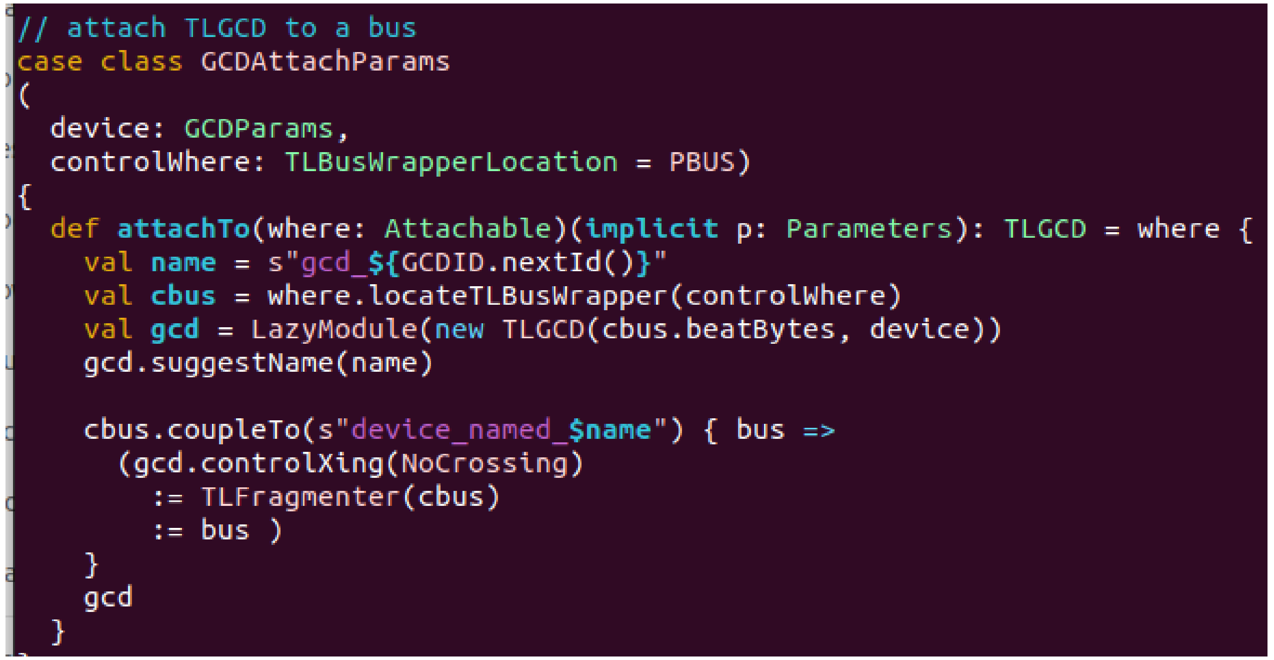


```
// declare TileLink-wrapper class for GCD-module
class TLGCD(busWidthBytes: Int, params: GCDParams)(implicit p: Parameters)
  extends GCDmod(busWidthBytes, params) with HasTLControlRegMap
```

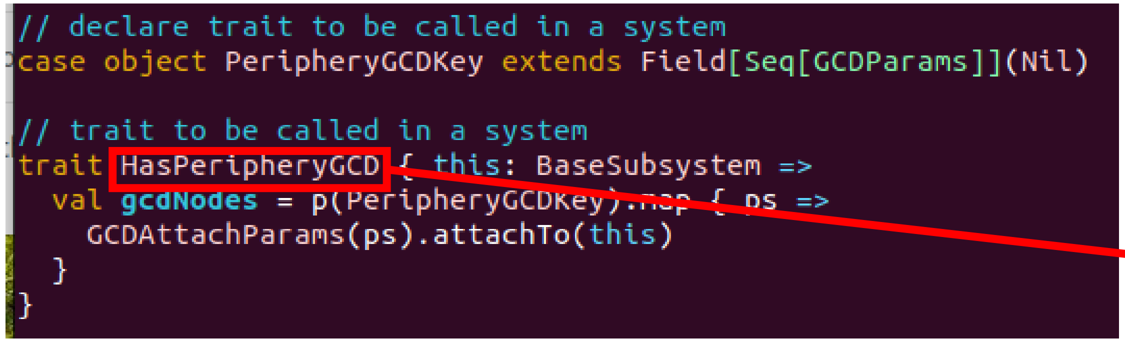
1. Last lecture brief review (9/10) System traits

In the `gcd.scala` file:

Create a **trait** to be called later in the RISCVConsole System:



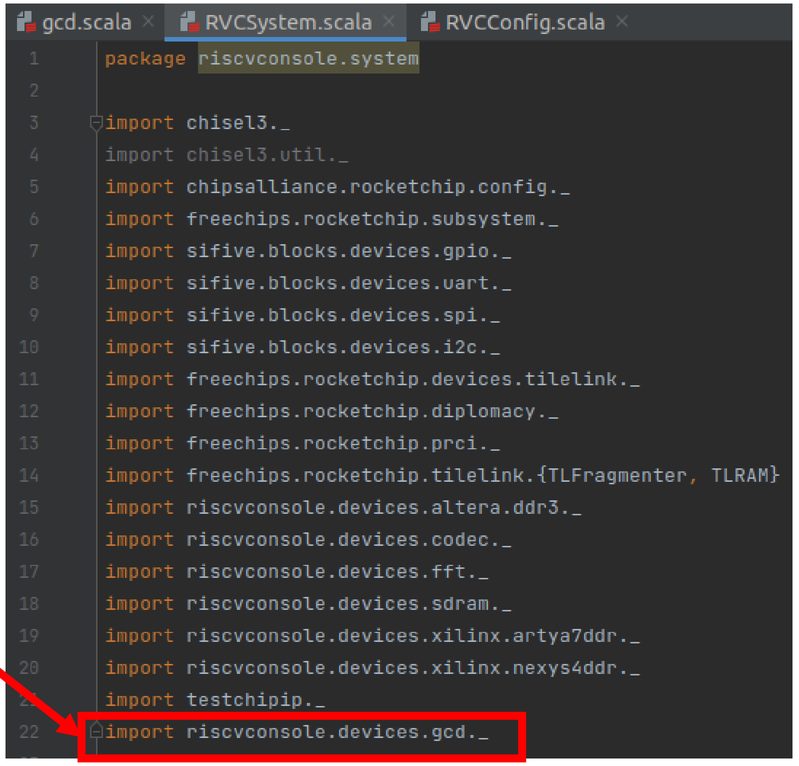
```scala
// attach TLGCD to a bus
case class GCDAttachParams
(
  device: GCDParams,
  controlWhere: TLBusWrapperLocation = PBUS)
{
  def attachTo(where: Attachable)(implicit p: Parameters): TLGCD = where {
    val name = s"gcd_${GCDID.nextId()}"
    val cbus = where.locateTLBusWrapper(controlWhere)
    val gcd = LazyModule(new TLGCD(cbus.beatBytes, device))
    gcd.suggestName(name)

    cbus.coupleTo(s"device_named_$name") { bus =>
      (gcd.controlXing(NoCrossing)
        := TLFragmenter(cbus)
        := bus )
    }
    gcd
  }
```



```scala
// declare trait to be called in a system
case object PeripheryGCDKey extends Field[Seq[GCDParams]](Nil)

// trait to be called in a system
trait HasPeripheryGCD { this: BaseSubsystem =>
  val gcdNodes = p(PeripheryGCDKey).map { ps =>
    GCDAttachParams(ps).attachTo(this)
  }
}
```

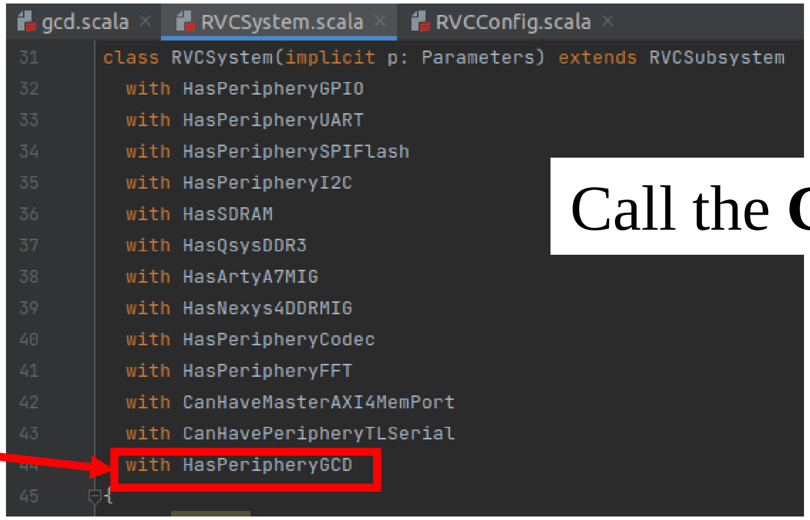
In the `RVCSystem.scala` file:

Import the **GCD package**:



```scala
package riscvconsole.system

import chisel3._
import chisel3.util._
import chipsalliance.rocketchip.config._
import freechips.rocketchip.subsystem._
import sifive.blocks.devices.gpio._
import sifive.blocks.devices.uart._
import sifive.blocks.devices.spi._
import sifive.blocks.devices.i2c._
import freechips.rocketchip.devices.tilelink._
import freechips.rocketchip.diplomacy._
import freechips.rocketchip.prci._
import freechips.rocketchip.tilelink.{TLFragmenter, TLRAM}
import riscvconsole.devices.altera.ddr3._
import riscvconsole.devices.codec._
import riscvconsole.devices.fft._
import riscvconsole.devices.sdram._
import riscvconsole.devices.xilinx.artya7ddr._
import riscvconsole.devices.xilinx.nexys4ddr._
import testchipip._
import riscvconsole.devices.gcd._
```

Call the **GCD trait**:

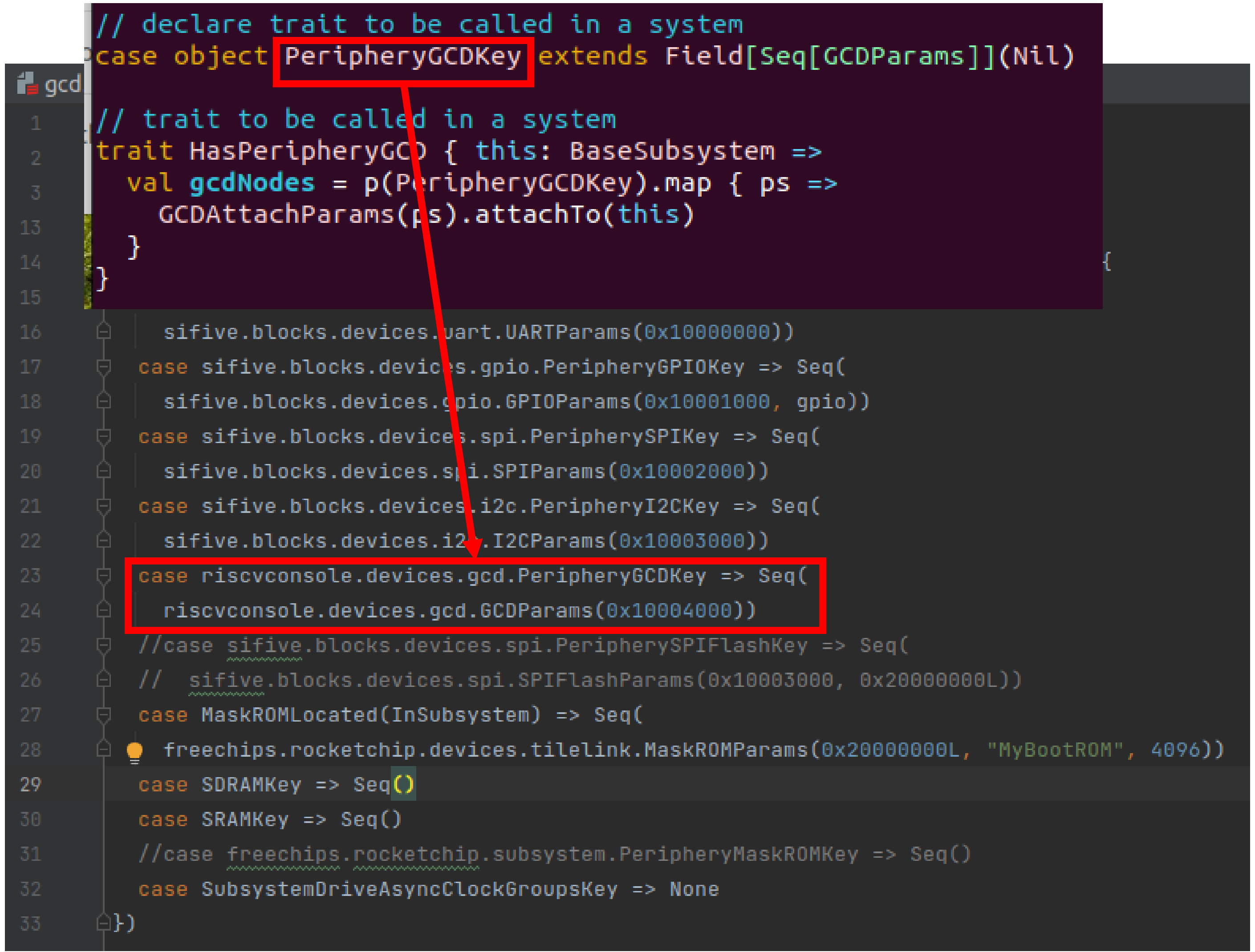


```scala
class RVCSystem(implicit p: Parameters) extends RVCSubsystem
  with HasPeripheryGPIO
  with HasPeripheryUART
  with HasPeripherySPIFlash
  with HasPeripheryI2C
  with HasSDRAM
  with HasQsysDDR3
  with HasArtyA7MIG
  with HasNexys4DDRMIG
  with HasPeripheryCodec
  with HasPeripheryFFT
  with CanHaveMasterAXI4MemPort
  with CanHavePeripheryTLSerial
  with HasPeripheryGCD
{
```

1. Last lecture brief review (10/10) Assign the address

In the `RVCConfig.scala` file:

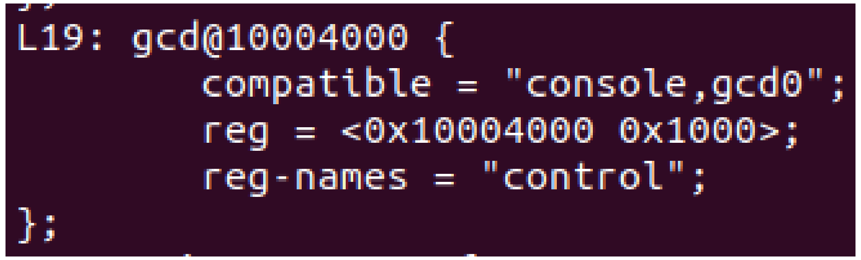
Put the **GCDKey** and assign an address.:



```
// declare trait to be called in a system
case object PeripheryGCDKey extends Field[Seq[GCDParams]](Nil)

// trait to be called in a system
trait HasPeripheryGCD { this: BaseSubsystem =>
  val gcdNodes = p(PeripheryGCDKey).map { ps =>
    GCDAttachParams(ps).attachTo(this)
  }
}
```

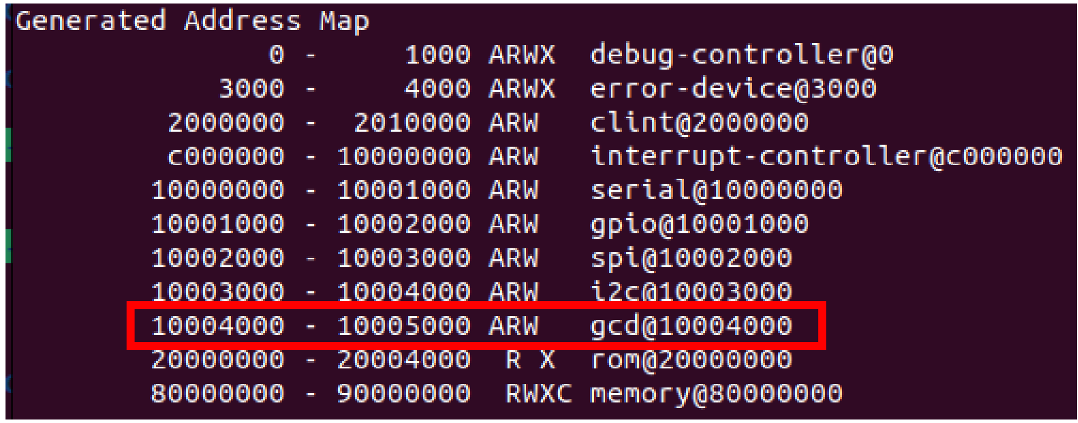
```
    sifive.blocks.devices.uart.UARTParams(0x10000000))
  case sifive.blocks.devices.gpio.PeripheryGPIOKey => Seq(
    sifive.blocks.devices.gpio.GPIOParams(0x10001000, gpio))
  case sifive.blocks.devices.spi.PeripherySPIKey => Seq(
    sifive.blocks.devices.spi.SPIParams(0x10002000))
  case sifive.blocks.devices.i2c.PeripheryI2CKey => Seq(
    sifive.blocks.devices.i2c.I2CParams(0x10003000))
  case riscvconsole.devices.gcd.PeripheryGCDKey => Seq(
    riscvconsole.devices.gcd.GCDParams(0x10004000))
  //case sifive.blocks.devices.spi.PeripherySPIFlashKey => Seq(
  //  sifive.blocks.devices.spi.SPIFlashParams(0x10003000, 0x20000000L))
  case MaskROMLocated(InSubsystem) => Seq(
    freechips.rocketchip.devices.tilelink.MaskROMParams(0x20000000L, "MyBootROM", 4096))
  case SDRAMKey => Seq()
  case SRAMKey => Seq()
  //case freechips.rocketchip.subsystem.PeripheryMaskROMKey => Seq()
  case SubsystemDriveAsyncClockGroupsKey => None
})
```

Next time, `$ make default` will see a new **gcd** module is included in the device-tree:



```
L19: gcd@10004000 {
        compatible = "console,gcd0";
        reg = <0x10004000 0x1000>;
        reg-names = "control";
};
```

The **gcd** is added in the address map:



```
Generated Address Map
               0 -     1000 ARWX  debug-controller@0
            3000 -     4000 ARWX  error-device@3000
         2000000 -  2010000 ARW   clint@2000000
         c000000 - 10000000 ARW   interrupt-controller@c000000
        10000000 - 10001000 ARW   serial@10000000
        10001000 - 10002000 ARW   gpio@10001000
        10002000 - 10003000 ARW   spi@10002000
        10003000 - 10004000 ARW   i2c@10003000
        10004000 - 10005000 ARW   gcd@10004000
        20000000 - 20004000  R X  rom@20000000
        80000000 - 90000000  RWXC memory@80000000
```

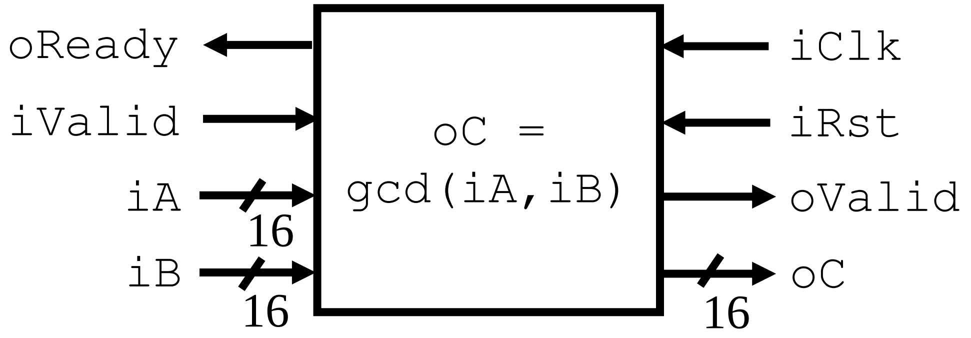
Outline

1. Last lecture brief review
2. Include external IOs
3. Practice: adder-subtractor with switch control

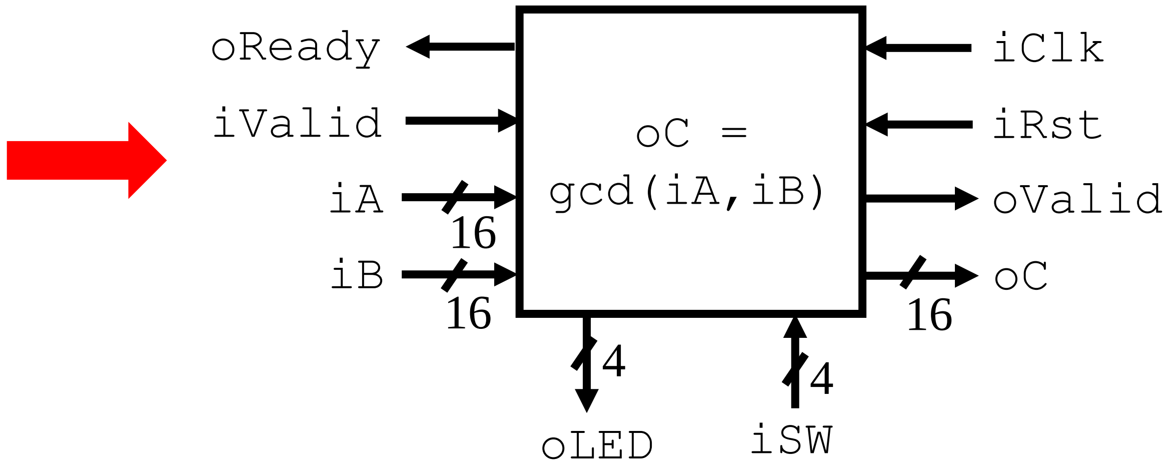
2. Include external IOs (1/10) GCD with external IOs

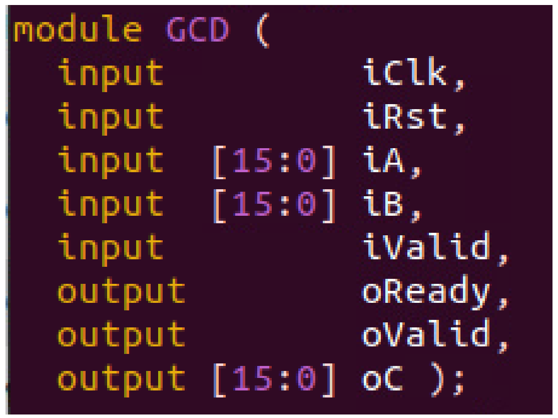
Now let's say the GCD module has **4bit** input switches and **4bit** output LEDs:

Then:



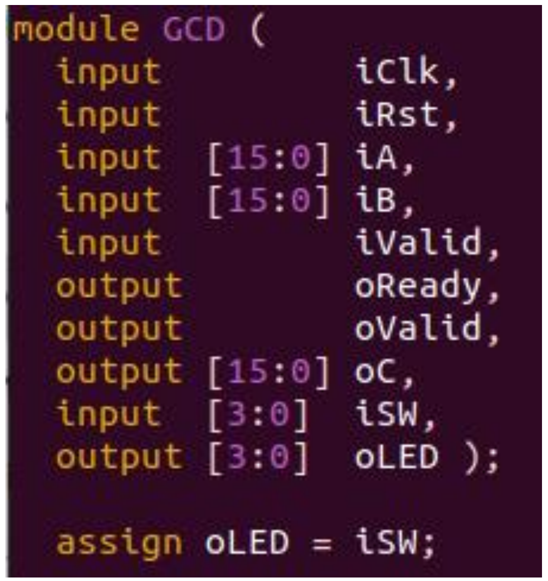
Now:





```
module GCD (
  input         iClk,
  input         iRst,
  input  [15:0] iA,
  input  [15:0] iB,
  input         iValid,
  output        oReady,
  output        oValid,
  output [15:0] oC );
```

Let's make thing simple, just assign switches to LEDs:



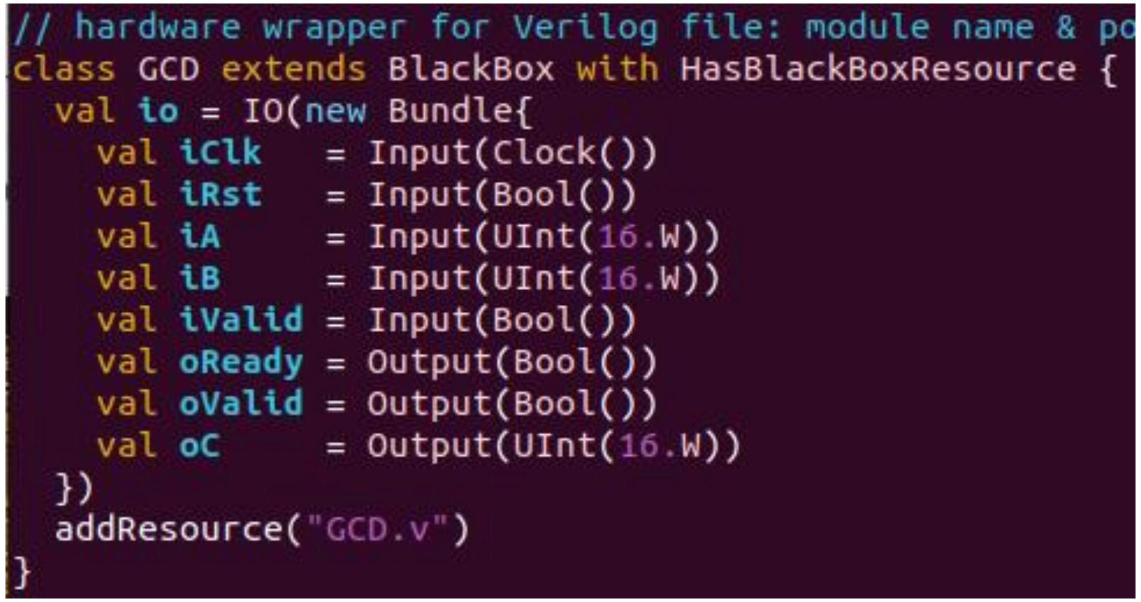
```
module GCD (
  input         iClk,
  input         iRst,
  input  [15:0] iA,
  input  [15:0] iB,
  input         iValid,
  output        oReady,
  output        oValid,
  output [15:0] oC,
  input  [3:0]  iSW,
  output [3:0]  oLED );

  assign oLED = iSW;
```

2. Include external IOs (2/10) In `gcd.scala`: GCD

In the `gcd.scala` file:

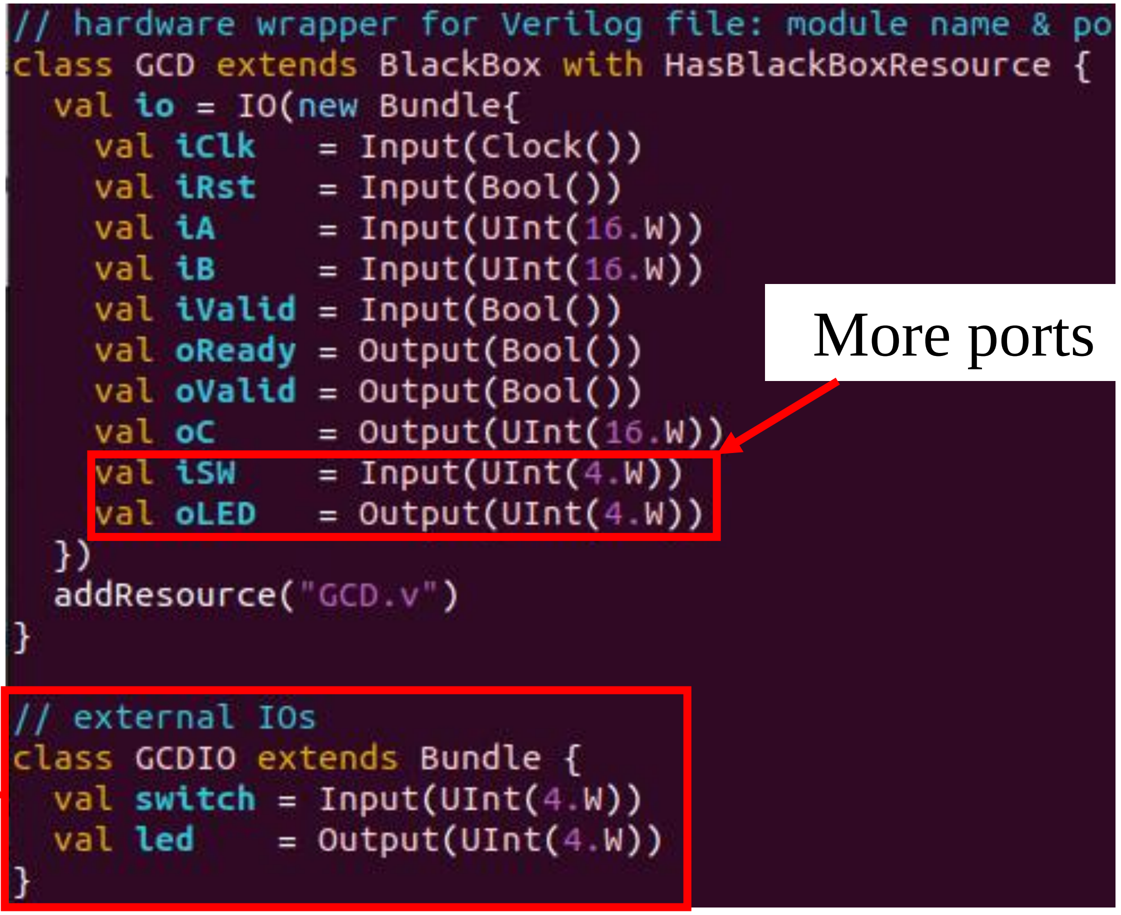
First, in the **GCD** class: *add more ports*

Then:



```scala
// hardware wrapper for Verilog file: module name & po
class GCD extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle{
    val iClk    = Input(Clock())
    val iRst    = Input(Bool())
    val iA      = Input(UInt(16.W))
    val iB      = Input(UInt(16.W))
    val iValid  = Input(Bool())
    val oReady  = Output(Bool())
    val oValid  = Output(Bool())
    val oC      = Output(UInt(16.W))
  })
  addResource("GCD.v")
}
```

Now:



```scala
// hardware wrapper for Verilog file: module name & po
class GCD extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle{
    val iClk    = Input(Clock())
    val iRst    = Input(Bool())
    val iA      = Input(UInt(16.W))
    val iB      = Input(UInt(16.W))
    val iValid  = Input(Bool())
    val oReady  = Output(Bool())
    val oValid  = Output(Bool())
    val oC      = Output(UInt(16.W))
    val iSW     = Input(UInt(4.W))
    val oLED    = Output(UInt(4.W))
  })
  addResource("GCD.v")
}

// external IOs
class GCDIO extends Bundle {
  val switch = Input(UInt(4.W))
  val led    = Output(UInt(4.W))
}
```

More ports

A new class (let's call **GCDIO**) for external IOs

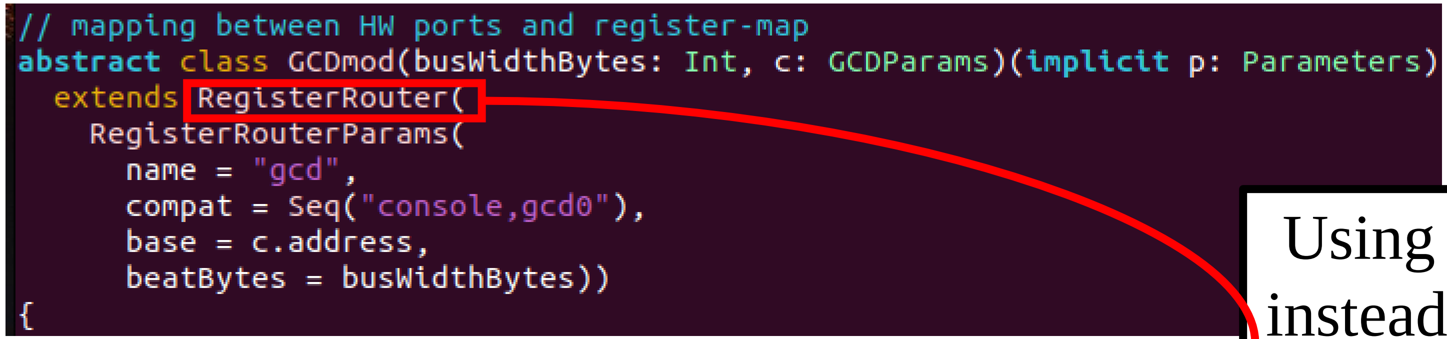
**Note:* the new `gcd.scala` file is provided in the material. You can copy it to your `devices/gcd/` folder.

2. Include external IOs (3/10) In `gcd.scala`: GCDmod

In the `gcd.scala` file:

Then, in the **GCDmod** class:
change to IORegisterRouter & add new IOs

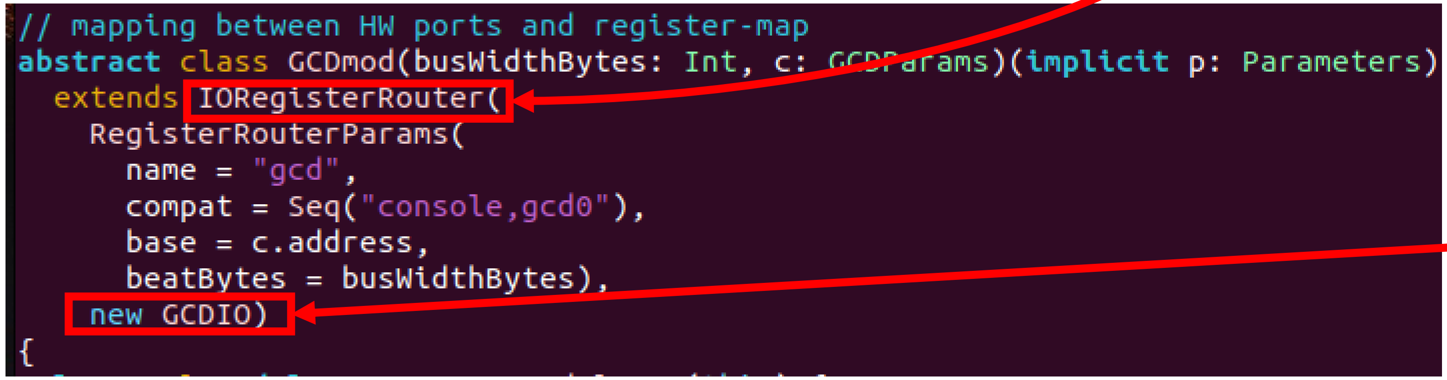
Then:



```
// mapping between HW ports and register-map
abstract class GCDmod(busWidthBytes: Int, c: GCDParams)(implicit p: Parameters)
  extends RegisterRouter(
    RegisterRouterParams(
      name = "gcd",
      compat = Seq("console,gcd0"),
      base = c.address,
      beatBytes = busWidthBytes))
{
```

Using `IORegisterRouter` instead of `RegisterRouter`

Now:



```
// mapping between HW ports and register-map
abstract class GCDmod(busWidthBytes: Int, c: GCDParams)(implicit p: Parameters)
  extends IORegisterRouter(
    RegisterRouterParams(
      name = "gcd",
      compat = Seq("console,gcd0"),
      base = c.address,
      beatBytes = busWidthBytes),
    new GCDIO)
{
```

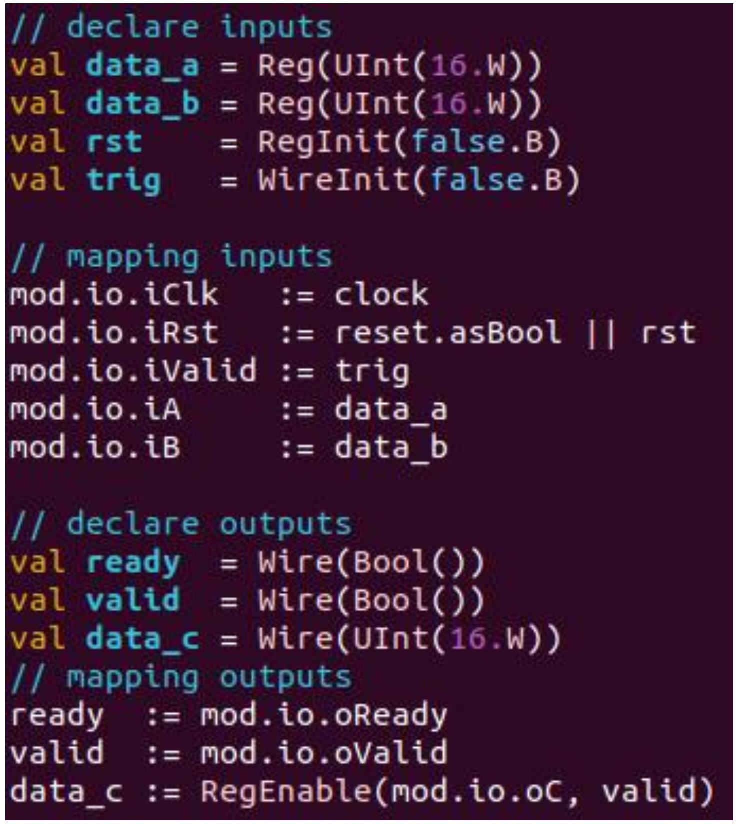
Add a new **GCDIO** *(just declared earlier)*

2. Include external IOs (4/10) In `gcd.scala`: GCDmod

In the `gcd.scala` file:

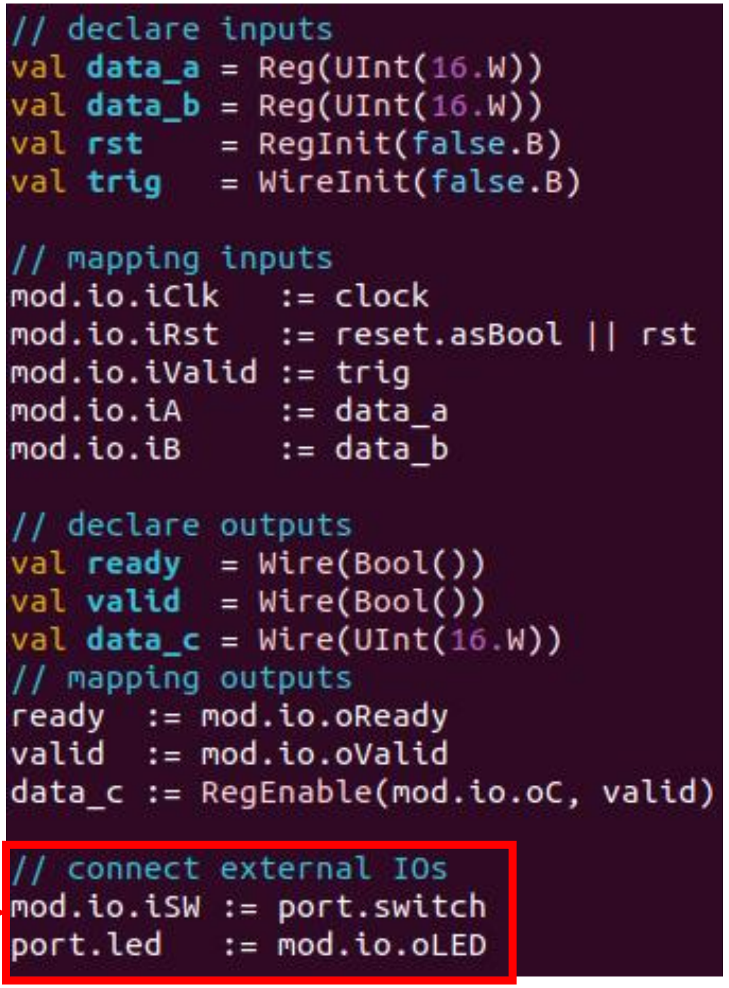
In the **GCDmod** class:
remember to connect the submodule's IOs

Then:



```
// declare inputs
val data_a = Reg(UInt(16.W))
val data_b = Reg(UInt(16.W))
val rst    = RegInit(false.B)
val trig   = WireInit(false.B)

// mapping inputs
mod.io.iClk    := clock
mod.io.iRst    := reset.asBool || rst
mod.io.iValid  := trig
mod.io.iA      := data_a
mod.io.iB      := data_b

// declare outputs
val ready   = Wire(Bool())
val valid   = Wire(Bool())
val data_c  = Wire(UInt(16.W))
// mapping outputs
ready   := mod.io.oReady
valid   := mod.io.oValid
data_c := RegEnable(mod.io.oC, valid)
```

Now:



```
// declare inputs
val data_a = Reg(UInt(16.W))
val data_b = Reg(UInt(16.W))
val rst    = RegInit(false.B)
val trig   = WireInit(false.B)

// mapping inputs
mod.io.iClk    := clock
mod.io.iRst    := reset.asBool || rst
mod.io.iValid  := trig
mod.io.iA      := data_a
mod.io.iB      := data_b

// declare outputs
val ready   = Wire(Bool())
val valid   = Wire(Bool())
val data_c  = Wire(UInt(16.W))
// mapping outputs
ready   := mod.io.oReady
valid   := mod.io.oValid
data_c := RegEnable(mod.io.oC, valid)

// connect external IOs
mod.io.iSW  := port.switch
port.led    := mod.io.oLED
```

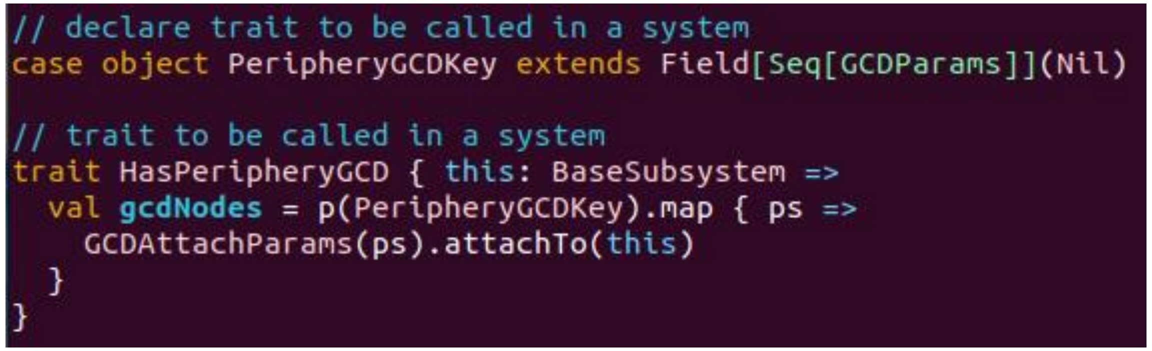
Connect **mod** (sub-module inside)'s ports to **GCDmod** (module outside)'s ports

2. Include external IOs (5/10) In `gcd.scala`: trait

In the `gcd.scala` file:

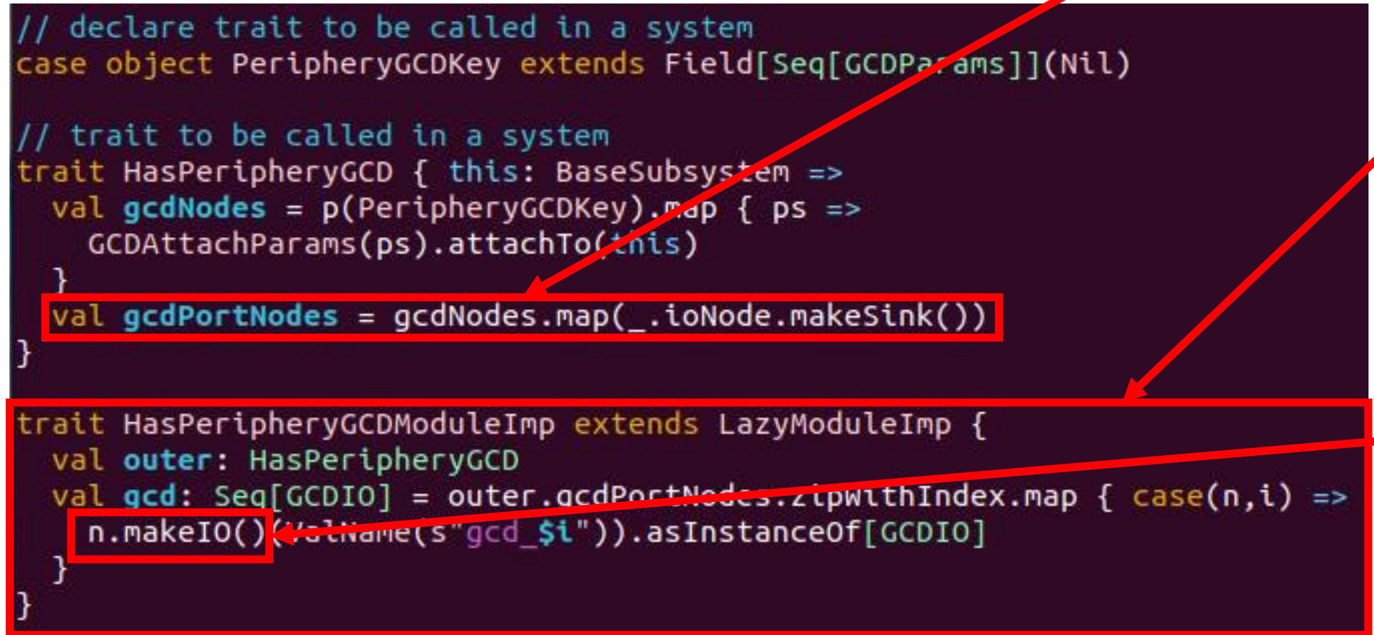
Finally, in the **HasPeripheryGCD** trait:
`makeSink()` for the RVCSystem

Then:



```scala
// declare trait to be called in a system
case object PeripheryGCDKey extends Field[Seq[GCDParams]](Nil)

// trait to be called in a system
trait HasPeripheryGCD { this: BaseSubsystem =>
  val gcdNodes = p(PeripheryGCDKey).map { ps =>
    GCDAttachParams(ps).attachTo(this)
  }
}
```

Now:



```scala
// declare trait to be called in a system
case object PeripheryGCDKey extends Field[Seq[GCDParams]](Nil)

// trait to be called in a system
trait HasPeripheryGCD { this: BaseSubsystem =>
  val gcdNodes = p(PeripheryGCDKey).map { ps =>
    GCDAttachParams(ps).attachTo(this)
  }
  val gcdPortNodes = gcdNodes.map(_.ioNode.makeSink())
}

trait HasPeripheryGCDModuleImp extends LazyModuleImp {
  val outer: HasPeripheryGCD
  val gcd: Seq[GCDIO] = outer.gcdPortNodes.zipWithIndex.map { case(n,i) =>
    n.makeIO()(ValName(s"gcd_$i")).asInstanceOf[GCDIO]
  }
}
```

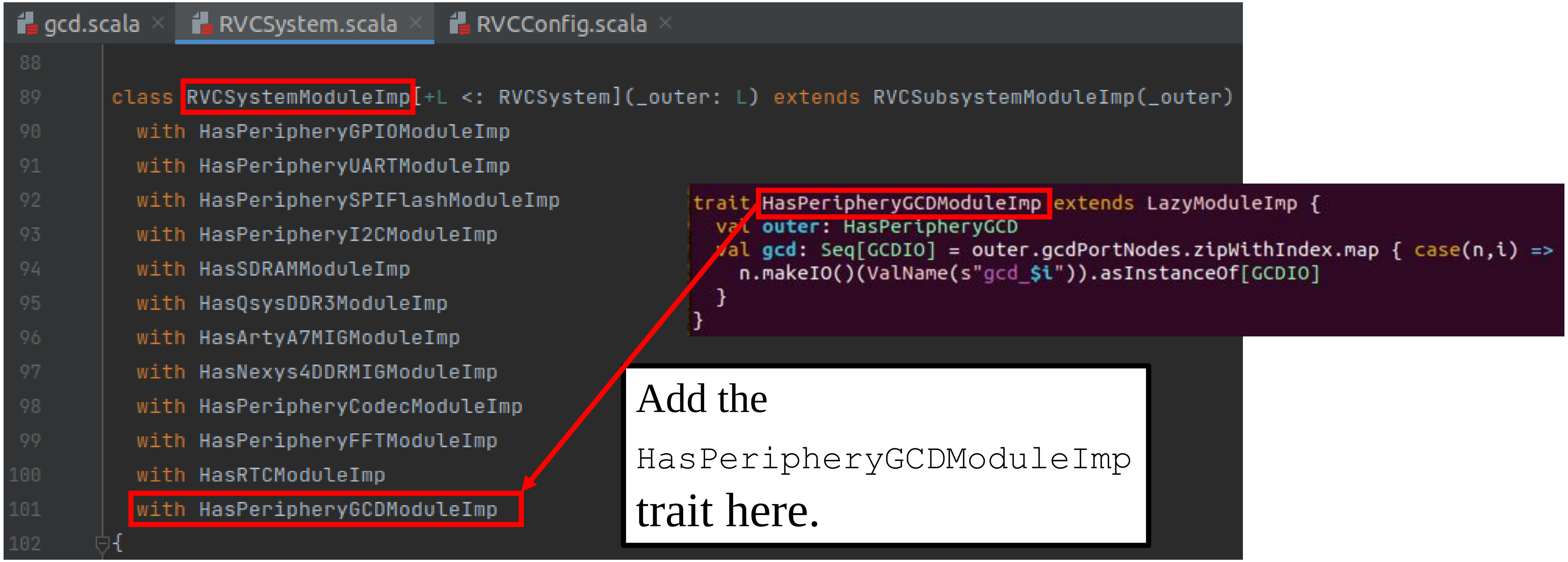
This `makeSink()` function will later create a "*sink*" port for the RVCSystem.

- The `HasPeripheryGCDModuleImp` trait will be used for the final system's module implementation.
- This `makeIO()` function will later create external IOs in the top module.

2. Include external IOs (6/10) In `system.scala`

In the `system.scala` file:

`RISCVConsole/hardware/riscvconsole/src/main/scala/riscvconsole/RVCSystem.scala`



```scala
class RVCSystemModuleImp[+L <: RVCSystem](_outer: L) extends RVCSubsystemModuleImp(_outer)
  with HasPeripheryGPIOModuleImp
  with HasPeripheryUARTModuleImp
  with HasPeripherySPIFlashModuleImp
  with HasPeripheryI2CModuleImp
  with HasSDRAMModuleImp
  with HasQsysDDR3ModuleImp
  with HasArtyA7MIGModuleImp
  with HasNexys4DDRMIGModuleImp
  with HasPeripheryCodecModuleImp
  with HasPeripheryFFTModuleImp
  with HasRTCModuleImp
  with HasPeripheryGCDModuleImp
{
```

```scala
trait HasPeripheryGCDModuleImp extends LazyModuleImp {
  val outer: HasPeripheryGCD
  val gcd: Seq[GCDIO] = outer.gcdPortNodes.zipWithIndex.map { case(n,i) =>
    n.makeIO()(ValName(s"gcd_$i")).asInstanceOf[GCDIO]
  }
}
```

Add the `HasPeripheryGCDModuleImp` trait here.

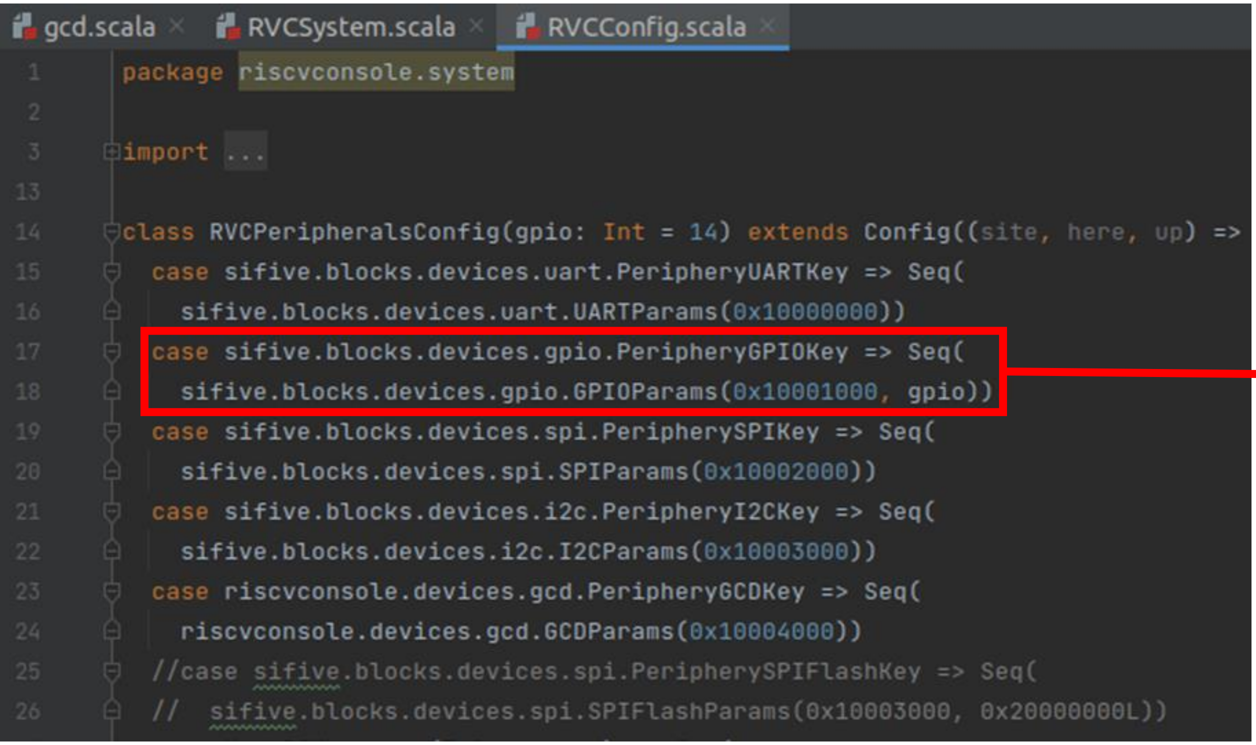
2. Include external IOs (7/10) In `config.scala`

In the `config.scala` file:

`RISCVConsole/hardware/riscvconsole/src/main/scala/riscvconsole/RVCConfig.scala`

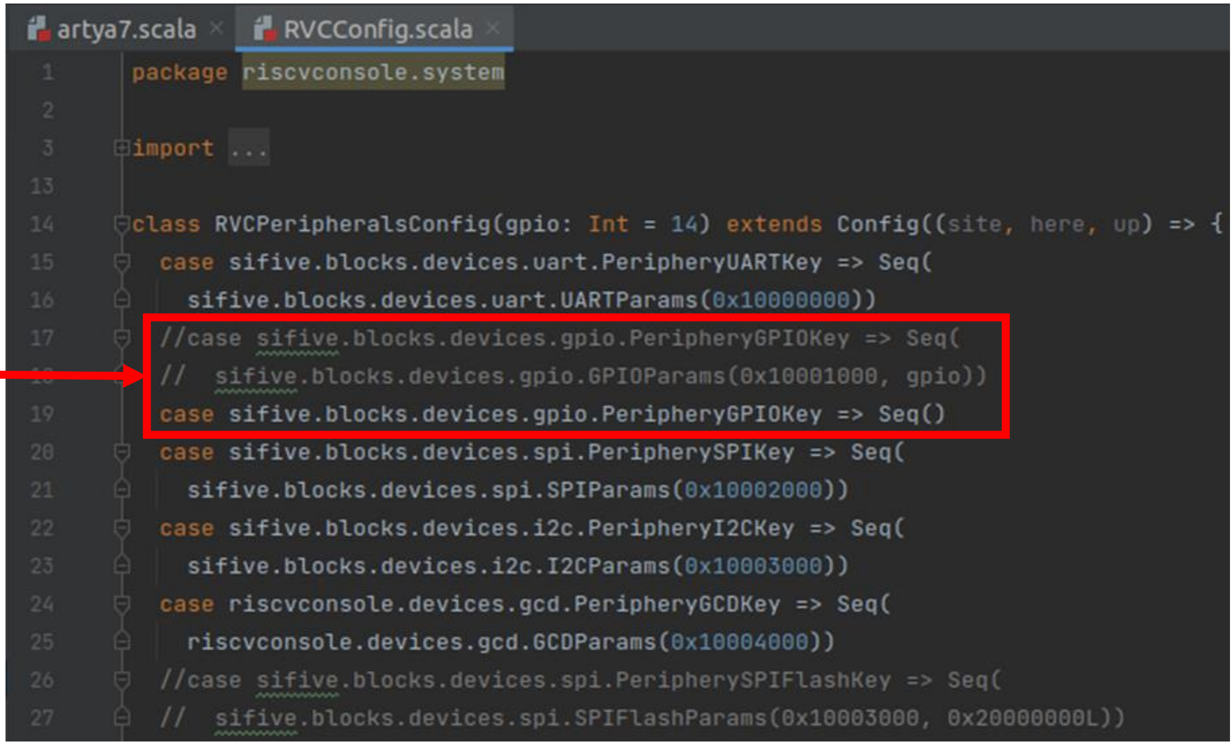
Because we will use the *LEDs* and *switches* for the **GCD** module, let’s <u>disable</u> the **GPIO** module in the system.

Then:



```
package riscvconsole.system

import ...

class RVCPeripheralsConfig(gpio: Int = 14) extends Config((site, here, up) =>
  case sifive.blocks.devices.uart.PeripheryUARTKey => Seq(
    sifive.blocks.devices.uart.UARTParams(0x10000000))
  case sifive.blocks.devices.gpio.PeripheryGPIOKey => Seq(
    sifive.blocks.devices.gpio.GPIOParams(0x10001000, gpio))
  case sifive.blocks.devices.spi.PeripherySPIKey => Seq(
    sifive.blocks.devices.spi.SPIParams(0x10002000))
  case sifive.blocks.devices.i2c.PeripheryI2CKey => Seq(
    sifive.blocks.devices.i2c.I2CParams(0x10003000))
  case riscvconsole.devices.gcd.PeripheryGCDKey => Seq(
    riscvconsole.devices.gcd.GCDParams(0x10004000))
  //case sifive.blocks.devices.spi.PeripherySPIFlashKey => Seq(
  //  sifive.blocks.devices.spi.SPIFlashParams(0x10003000, 0x20000000L))
```

Now:

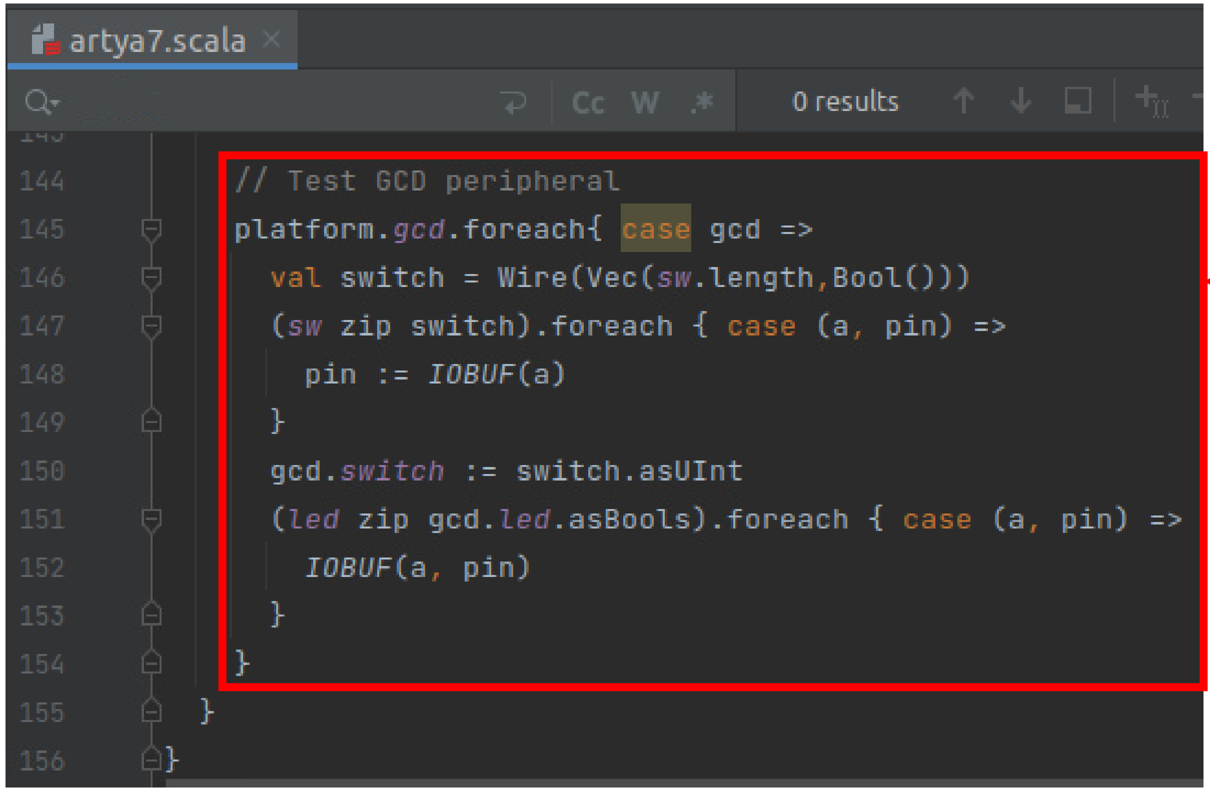


```
package riscvconsole.system

import ...

class RVCPeripheralsConfig(gpio: Int = 14) extends Config((site, here, up) => {
  case sifive.blocks.devices.uart.PeripheryUARTKey => Seq(
    sifive.blocks.devices.uart.UARTParams(0x10000000))
  //case sifive.blocks.devices.gpio.PeripheryGPIOKey => Seq(
  //  sifive.blocks.devices.gpio.GPIOParams(0x10001000, gpio))
  case sifive.blocks.devices.gpio.PeripheryGPIOKey => Seq()
  case sifive.blocks.devices.spi.PeripherySPIKey => Seq(
    sifive.blocks.devices.spi.SPIParams(0x10002000))
  case sifive.blocks.devices.i2c.PeripheryI2CKey => Seq(
    sifive.blocks.devices.i2c.I2CParams(0x10003000))
  case riscvconsole.devices.gcd.PeripheryGCDKey => Seq(
    riscvconsole.devices.gcd.GCDParams(0x10004000))
  //case sifive.blocks.devices.spi.PeripherySPIFlashKey => Seq(
  //  sifive.blocks.devices.spi.SPIFlashParams(0x10003000, 0x20000000L))
```

2. Include external IOs (8/10) In `artya7.scala`

In the `artya7.scala` file:

`RISCVConsole/hardware/riscvconsole/src/main/scala/fpga/artya7.scala`



```
    // Test GCD peripheral
    platform.gcd.foreach{ case gcd =>
      val switch = Wire(Vec(sw.length,Bool()))
      (sw zip switch).foreach { case (a, pin) =>
        pin := IOBUF(a)
      }
      gcd.switch := switch.asUInt
      (led zip gcd.led.asBools).foreach { case (a, pin) =>
        IOBUF(a, pin)
      }
    }
  }
}
```

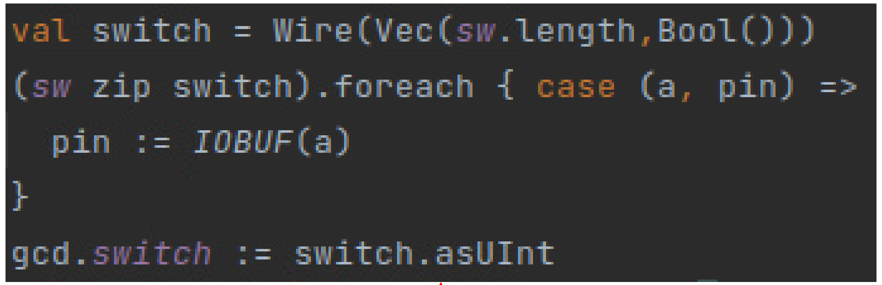
There is **only one class** in the file, and at the end of that class, add these:

This is for the FPGA pin map.

2. Include external IOs (9/10) In `artya7.scala`

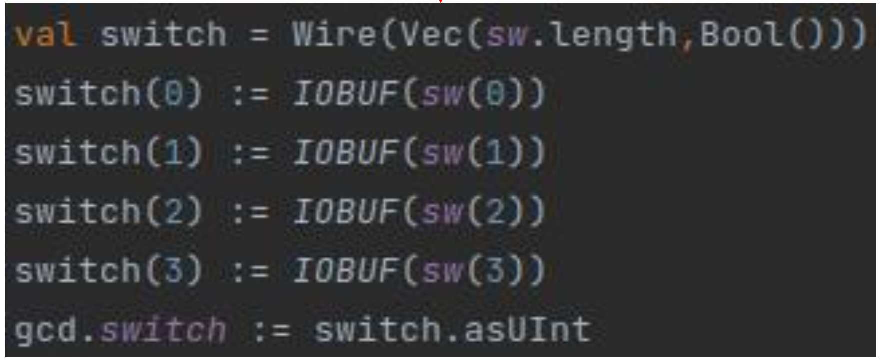
******Note:*** instead of assigning in a group, you can do the assignment individually.

For inputs:



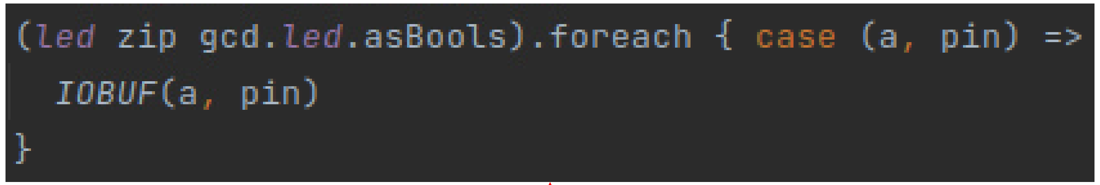
```
val switch = Wire(Vec(sw.length,Bool()))
(sw zip switch).foreach { case (a, pin) =>
  pin := IOBUF(a)
}
gcd.switch := switch.asUInt
```

SAME



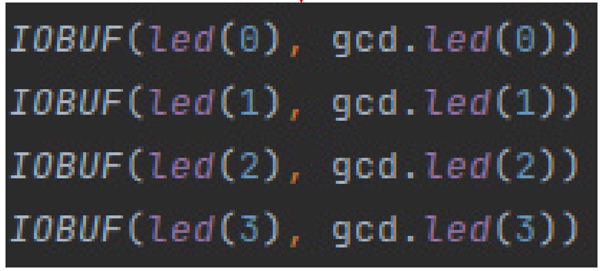
```
val switch = Wire(Vec(sw.length,Bool()))
switch(0) := IOBUF(sw(0))
switch(1) := IOBUF(sw(1))
switch(2) := IOBUF(sw(2))
switch(3) := IOBUF(sw(3))
gcd.switch := switch.asUInt
```

For outputs:



```
(led zip gcd.led.asBools).foreach { case (a, pin) =>
  IOBUF(a, pin)
}
```

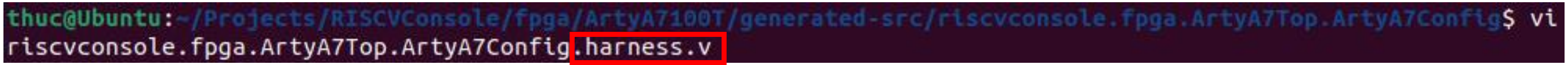
SAME



```
IOBUF(led(0), gcd.led(0))
IOBUF(led(1), gcd.led(1))
IOBUF(led(2), gcd.led(2))
IOBUF(led(3), gcd.led(3))
```

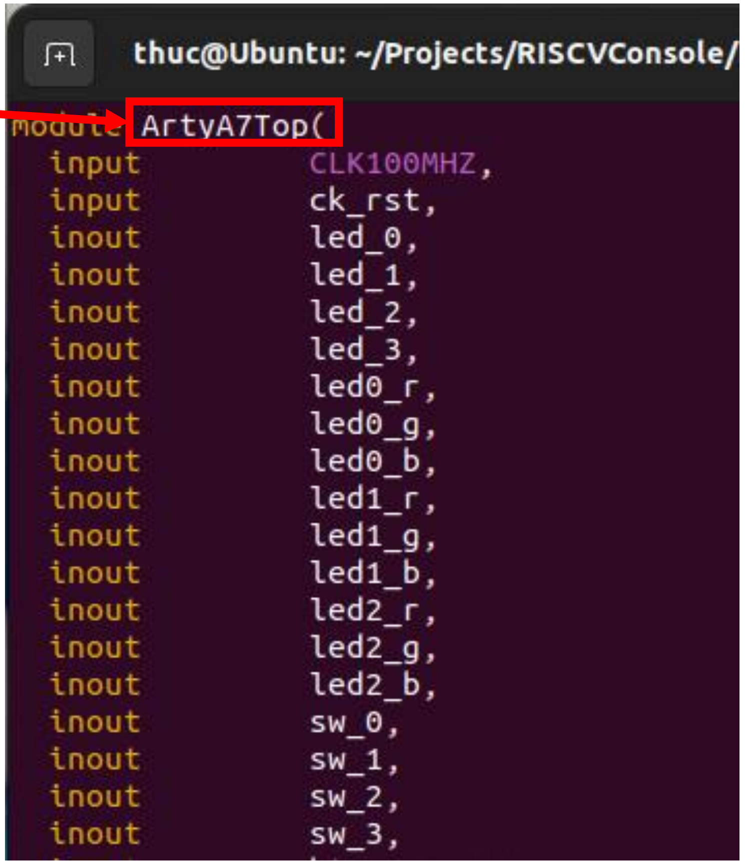
2. Include external IOs (10/10) Check the generated Verilog

Now, the `$ make default` will generate a new Verilog sources.
You can check the **ArtyA7Top** top-module to make sure that the *external IOs* were connected correctly:



```
thuc@Ubuntu:~/Projects/RISCVConsole/fpga/ArtyA7100T/generated-src/riscvconsole.fpga.ArtyA7Top.ArtyA7Config$ vi
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.harness.v
```

Contain the **ArtyA7Top** module



```
thuc@Ubuntu: ~/Projects/RISCVConsole/
module ArtyA7Top(
  input           CLK100MHZ,
  input           ck_rst,
  inout           led_0,
  inout           led_1,
  inout           led_2,
  inout           led_3,
  inout           led0_r,
  inout           led0_g,
  inout           led0_b,
  inout           led1_r,
  inout           led1_g,
  inout           led1_b,
  inout           led2_r,
  inout           led2_g,
  inout           led2_b,
  inout           sw_0,
  inout           sw_1,
  inout           sw_2,
  inout           sw_3,
```

Outline

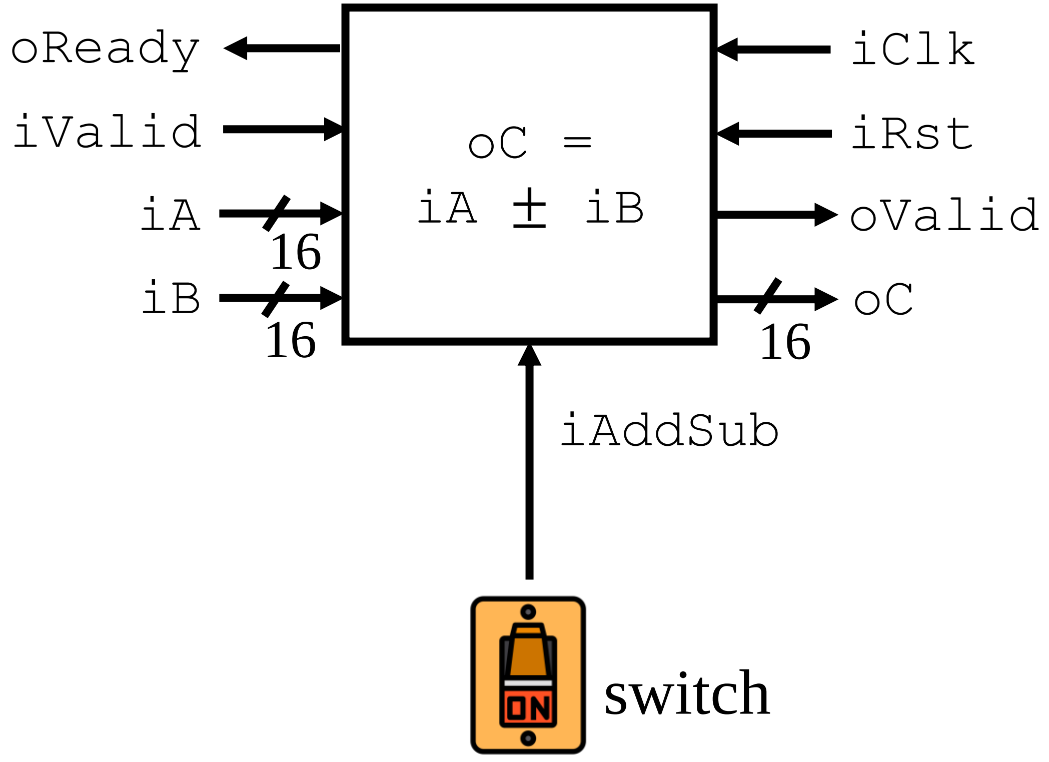
1. Last lecture brief review
2. Include external IOs
3. Practice: adder-subtractor with switch control

3. Practice: adder-subtractor with switch control (1/1)

Exercise 1:

Remake the **adder** to **adder-subtractor**;
using a outside switch to control the *add/sub* function:

Remake the Verilog file
→ remake the Scala wrapper
→ attach it to the system's PBus
→ regenerate the system
→ confirm the function in software



国立大学法人
電気通信大学
The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

THANK YOU

2022/12