Programación de la FPGA con Chisel.

Generación del Código Verilog

Pimero: aprender Chisel. Lo mejor es seguir el *bootcamp* para conocer los conceptos básicos del lenguaje, y el libro de Martin Schoeberl.

https://github.com/freechipsproject/chisel-bootcamp

https://www.imm.dtu.dk/~masca/chisel-book.html

Segundo: Una vez hemos terminado la descripción hardware de nuestro módulo, lo convertimos a Verilog añadiendo la siguiente instrucción al final del *.scala* de Chisel (fuera de la clase):

```
import chisel3._
import chisel3.util._

class MyModule extends Module {

//Chisel module definition goes here.

}

/**
 * An object extending App to generate the Verilog code.
 */
object SafeCode extends App {
  (new chisel3.stage.ChiselStage).emitVerilog(new SafeCode())
}
```

Será necesario que nuestro módulo tenga definido en su entrada salida (IO) cualquier conexión que queramos hacer con las interfaces de la FPGA (botones, leds…). Un detalle importante es que cuando luego lo conectemos con la FPGA, el reset puede estar invertido (esto ocurre de hecho en la Arty 7 que usamos). Para solucionarlo, añadimos esta línea, que invierte el reset:

```
  withReset(~reset.asBool){
```

Tenemos que ponerlo después del IO Bundle, y dentro de esa llave estará todo nuestro código (que es lo que queremos que funcione con el reset invertido).

```
class MyCPU (dataWidth: Int) extends Module {
  val io = IO(new Bundle {
   //IO goes here.
  })
  withReset(~reset.asBool){

  //Chisel module definition goes here
}
```

Al ejecutar sbt en la carpeta donde se encuentra nuestro código (el que tiene emitVerilog), generaremos nuestro ".v".

```
$ sbt run
```

Proyecto en Vivado

Con este código Verilog ya podemos irnos a Vivado a trabajar con la FPGA. Es recomendable familiarizarse primero con Vivado, existen un par de tutoriales sencillos para ello (son para una versión anterior, pero me parecen más sencillos de seguir que los de la nueva):

https://digilent.com/reference/programmable-logic/guides/getting-started-with-vivado

https://digilent.com/reference/vivado/getting_started/start

Hay que tener un par de conceptos claros después de esto:

- La funcionalidad de nuestro modulo es lo que creamos con el Verilog. Vivado tiene una interfaz tipo CAD que permite conectar distintos módulos, e incluso es posible importar IPs (módulos de una librería) a nuestro diseño. Para empezar, restringirse a un único módulo con Chisel es un buen comienzo.
- Es importante conectar correctamente nuestro modulo (la funcionalidad) con nuestra plataforma, que será la placa con la FPGA.

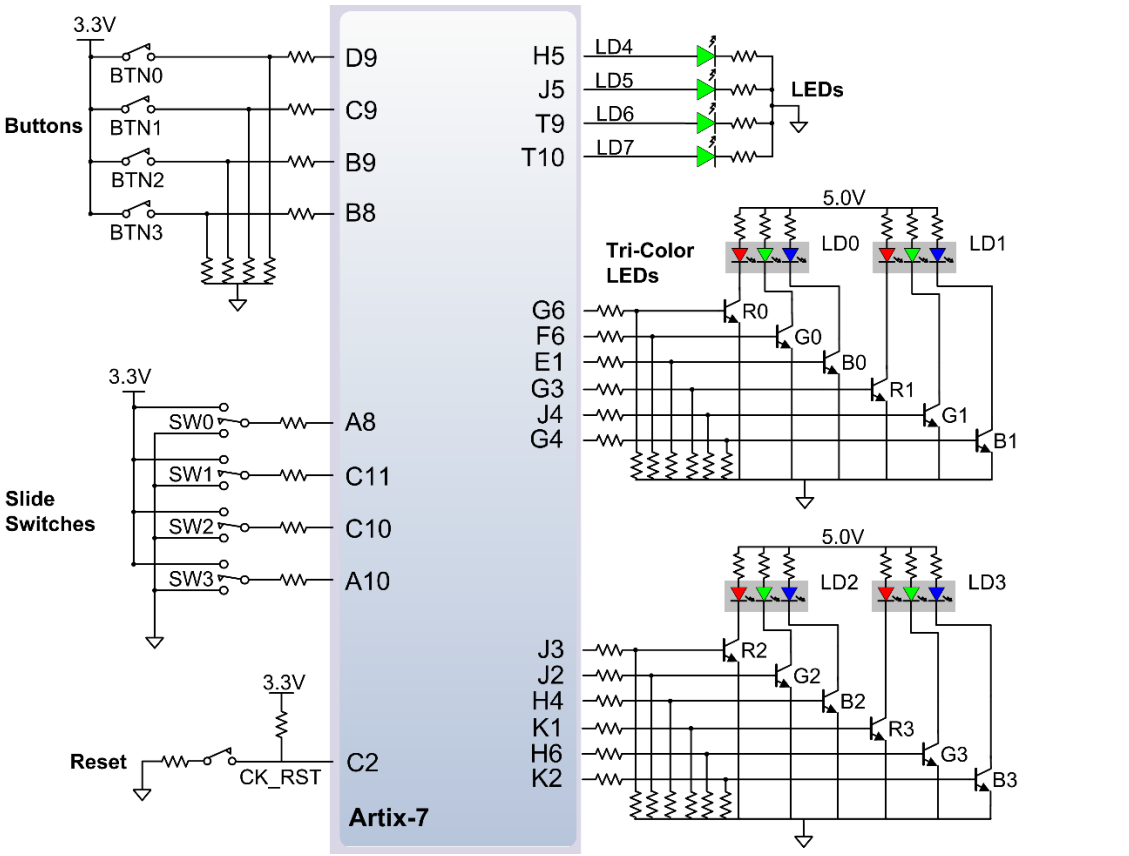
Para empezar a crear un módulo podemos hacerlo abriendo Vivado y seleccionando un nuevo proyecto. Podemos poner el nombre que queramos, y elegimos RTL, sin poner todavía ninguna fuente (marcamos la opción "*Do not specify sources at this time*"). Esto nos permite hacer las cosas a nuestro ritmo, aunque tampoco es muy problemático cargarlo desde el principio.

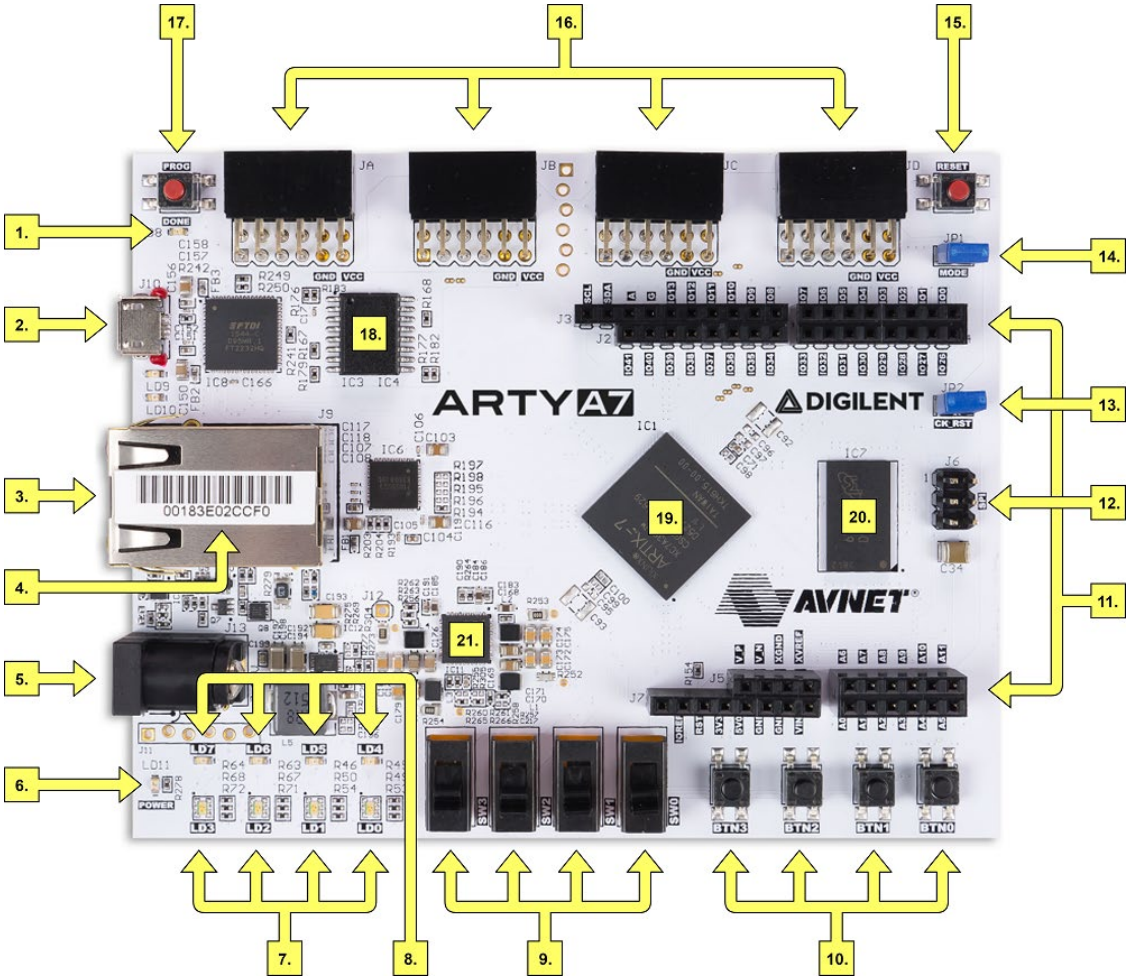
Nos va a pedir asociar el proyecto con un hardware. Elegimos el de nuestra placa de desarrollo (en mi caso, Arty A7 100T). Si no está disponible en la sección de *boards*, se puede descargar desde la web (en mi caso digilent https://github.com/Digilent/vivado-boards/archive/master.zip)

Una vez creado el proyecto, tenemos que cargar las dos componentes fundamentales que mencionábamos antes: las CONSTRAINTS (que definen los parámetros de implementación y además nos permite definir toda la interfaz de la placa) y las SOURCES (que son los ficheros con la funcionalidad).

No definimos las *constraints* desde cero, si no que cargamos la correspondiente a la placa que queremos programar. En mi caso, al ser una placa Digilent, bajo las XDC (ficheros de *constraints*) de toda la gama https://github.com/Digilent/digilent-xdc/archive/master.zip. Una vez descargados los ficheros XDC, podemos añadirlo a nuestro proyecto. Pinchamos en la opción *Add Source* (menú file, o el botón **+** en la ventana de *Sources*) y elegimos añadir un *Constraint*. En la ventana siguiente, seleccionamos añadir archivo y buscamos el correspondiente a nuestra placa (Arty A7 100T). **IMPORTANTE**: marcar la opción "*Copy constraints files into Project*", de esta forma el fichero XDC se copia al proyecto y será el que modifiquemos. De no hacerlo así, estaremos modificando el fichero de Constraints base (que luego querremos usar en otros proyectos).

Una vez cargado el fichero aparecerá disponible en la ventana de Sources. Podemos abrirle con doble click y aparece en la ventana de diseño. Veremos que todas las líneas están comentadas, pero nos sirve para ver de que forma se buscan los distintos interfaces de nuestra placa (leds, botones, switches, puerto ethernet…), así como el reloj. Habrá que descomentar las líneas asociadas con aquellos componentes que queremos utilizar. Podemos ver además el PACKAGE_PIN asociado a cada uno de ellos, que se corresponde con los de la figura.


3.3V
BTN0
BTN1
BTN2
BTN3
Buttons
D9
C9
B9
B8
H5
J5
T9
T10
LD4
LD5
LD6
LD7
LEDs
5.0V
Tri-Color LEDs
LD0
LD1
G6
F6
E1
G3
J4
G4
R0
G0
B0
R1
G1
B1
3.3V
SW0
SW1
SW2
SW3
Slide Switches
A8
C11
C10
A10
5.0V
LD2
LD3
J3
J2
H4
K1
H6
K2
R2
G2
B2
R3
G3
B3
3.3V
Reset
CK_RST
C2
Artix-7


1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
ARTY A7
DIGILENT
AVNET
00183E02CCF0

Callout	Description	Callout	Description	Callout	Description
1	FPGA programming DONE LED	8	User RGB LEDs	15	chipKIT processor reset
2	Shared USB JTAG / UART port	9	User slide switches	16	Pmod connectors
3	Ethernet connector	10	User push buttons	17	FPGA programming reset button
4	MAC address sticker	11	Arduino/chipKIT shield connectors	18	SPI flash memory
5	Power jack for optional external supply	12	Arduino/chipKIT shield SPI connector	19	Artix FPGA
6	Power good LED	13	chipKIT processor reset jumper	20	Micron DDR3 memory
7	User LEDs	14	FPGA programming mode	21	Dialog Semiconductor DA9062 power supply

Lo siguiente será cargar nuestro fichero fuente de Verilog. Seguimos el mismo procedimiento que para el fichero de *Constraints*, con *Add Source*, pero eligiendo la segunda opción (*Add or créate design sources*). De nuevo, copiamos el fuente al proyecto y podremos verlo en la ventana de *Sources*. Podemos echar un vistazo al Verilog haciendo doble click sobre el fichero y de nuevo se abrirá en el área de diseño. Aquí es donde es necesario encajar la interfaz de nuestro módulo (Verilog) con la interfaz de la FPGA (las constraints). Para ello, nos fijamos en el nombre que tienen los distintos puertos de entrada y salida de nuestro módulo (si lo hemos hecho con Chisel, todos empezarán por: "io_<nombre>"). Este es el nombre que tenemos que asignar a los distintos componentes en el fichero de constraints, en el nombre entre llaves que está después de "*get_ports*".

Así, por ejemplo, si nuestro módulo tiene una salida en Chisel correspondiente a un LED con este aspecto:

```
val led_r = Output(UInt(1.W))
```

En Verilog lo veremos como:

```
output      io_led_r,
```

Y nosotros tendremos que modificar el fichero de constraints, con la línea correspondiente a ese LED, para que el puerto tenga ese mismo nombre:

```
set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 } [get_ports { io_led_r }]; #IO_L19P_T3_35 Sch=led0_r
```

Debemos hacer esto para cada uno de los puertos de entrada y salida de nuestro módulo, para que se asocien con el componente adecuado de nuestra FPGA.

IMPORTANTE: Podrás ver como por defecto Chisel ha creado dos entradas en nuestro circuito: *clock* y *reset*. Estás entradas existen, aunque nosotros no las definamos, y hay que asociarlas con el botón correspondiente en el caso del *reset* (el especifico de la placa es el C2, pero se puede usar cualquier otro si le ponemos el nombre de "reset"). Para el reloj, hay que des comentar las dos líneas asociadas al reloj, la de *set_property* (donde hay que poner el nombre adecuado al puerto, en este caso, "*clock*") y la de *create_clock* (en esta podemos definir el período del mismo, y de nuevo hay que especificar el puerto con el nombre "clock" o el que se corresponda en nuestro diseño).

Sintesis, Implementación y Generación de Bitstream

Una vez acabado la unión entre modulo y constraints, podemos pasar a compilar. Para ello, lo más fácil es darle a Generate Bitstream. Esto pasa por todas las fases necesarias: *Synthesis*, *Implementation* (incluido place and route) y finalmente *Bitstream Generation*. Es posible ver el diseño (Open Synthesized Design y Open Implemented Design) para ver como se mapea nuestro diseño en la FPGA. Es posible ver también consumos de potencia, así como un análisis temporal del diseño. Adicionalmente, se le puede pedir un "*Report Utilizacion*", donde podemos ver como se está usando la FPGA (y si andamos muy justos o tenemos margen). Cuando finaliza este paso, tendremos un fichero **.bit** (se encuentra en la carpeta "**.runs**" dentro de nuestro proyecto, dentro de la subcarpeta "**impl_#**"). Ahora ya podemos pasar a programar el hardware.

Para programar la FPGA, podemos hacerlo desde el mismo Vivado. Para ello, una vez generado el Bitstream, podremos abrir el *Hardware Manager*. Aquí, seleccionamos "*Open Target*" y le damos a *Auto Connect* si la FPGA está enchufada en local. Si no, hay que elegir "Open New Target", especificando la IP y puerto donde se encuentra conectada la FPGA (tiene un wizard para la conexión). Si todo ha ido bien, detectará automáticamente la placa y podremos ejecutar "Program Device". Saldrá un desplegable con todo el harware que hayamos cargado (que probablemente sea solo uno), elegimos el nuestro y se procede a la carga del .bit en el mismo.

Conectarse a una consola UART

Descargar Tera Term (https://ttssh2.osdn.jp/index.html.en)
Abrir una conexión Serie por el USB.
Se puede probar esto con el proyecto existente en:
https://github.com/Digilent/Arty-A7/releases/tag/100/GPIO/2022.1-1
Explicado en:
https://digilent.com/reference/programmable-logic/arty-a7/demos/gpio?redirect=1

Creación de Modulos (para importar luego en Chisel)

Creamos el fichero de Chisel al uso, pero añadiendo al comienzo package <nombre_componente>, como por ejemplo este multiplexor de N bits:

```
package mux2
import chisel3._

class Mux2 (inWidth: Int) extends Module {
    require(inWidth >= 0)
    val io = IO(new Bundle {
    val sel = Input(UInt(1.W))
    val in0 = Input(UInt(inWidth.W))
    val in1 = Input(UInt(inWidth.W))
    val out = Output(UInt(inWidth.W))
    })
    when (io.sel === 0.U){
        io.out := io.in0
    } .otherwise{
        io.out := io.in1
    }
}
```

Conviene guardar los módulos en una carpeta (el nombre de la carpeta determina el nombre de nuestro modulo), en la que tendremos un fichero sbt que contenga la información de la versión de nuestro módulo y de la organización (por ejemplo, yo tengo una carpeta de nombre "*modules*"). Este sería un ejemplo de fichero sbt:

```
version := "1.0"
organization := "uc.prietop"

scalaVersion := "2.12.13"

scalacOptions ++= Seq(
  "-feature",
  "-language:reflectiveCalls",
)

// Chisel 3.5
addCompilerPlugin("edu.berkeley.cs" % "chisel3-plugin" % "3.5.3" cross CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.5.3"
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" % "0.5.3"
```

(Incluye la dependencias de chisel, porque se usan en el componente que he creado).

A continuación, ejecutamos publishLocal de sbt dentro de la carpeta del módulo:

```
$ sbt publishLocal
```

Ahora, es necesario añadir esta dependencia en el sbt del proyecto en el que vamos a incluir nuestro módulo:

```
libraryDependencies += "uc.prietop" %% "modules" % "1.0"
```

Y ya podemos incluirlo en nuestro proyecto:

```
import chisel3._
import chisel3.util._
import mux2._


class MyCPU (dataWidth: Int) extends Module {
```

(Ponemos el nombre del package que vamos a incluir dentro de nuestro módulo)

Creación de una IP

Podemos crear una IP de un proyecto abierto. Para eso vamos a **Tools-> Create and Package IP...** Se nos abrirá un wizard para realizar el empaquetado del proyecto. Elegimos empaquetar el proyecto actual. Lo guardamos en el path por defecto, que será en el directorio del propio proyecto (o elegimos otro que sea una librería donde guardemos todas las IPs), marcando la opción de incluir los ficheros .xci. En la siguiente ventana le damos a Finalizar, y tras un rato de procesado, nos aparecerá una ventana para validar nuestra IP, incluyendo errores o avisos que puedan haber saltado en el proceso. Si estamos satisfechos, seleccionamos "Package IP" abajo en el último menú de "Review and Package".

Usar una IP propia en un proyecto

En el menú de **Project Manager** (en el Flow Navigator de la Izquierda), elegimos **Settings -> IP->Repository**. En la ventana que aparece le damos al + y ponemos el directorio en el que tenemos almacenada la IP.

Ahora podemos usar una ventana CAD para realizar nuestro circuito incluyendo nuestra IP. Para crear un área de diseño vamos a **IP Integrator** (en el Flow Navigator) y seleccionamos **Create Block Design**.

Importante terminar de realizar nuestro conexionado, no solo entre modulos, si no también el externo. Hay un pop-up wizard que te ayuda a conectar el reloj y el reset, y luego es posible hacer que algunos pines sean externos dando botón derecho sobre ellos con el “lápiz” y seleccionando la opción **Make External**. Una vez finalizado el diseño, en la ventana de sources, seleccionamos el diagrama (fichero .bd) y damos botón derecho **Create HDL Wrapper**.

Cuando hagamos el proceso de síntesis, se nos pide indicar un top. Tendremos que indicar el nombre del wrapper. Luego hay que mirar el wrapper para que encaje con los nombres de los constraints.