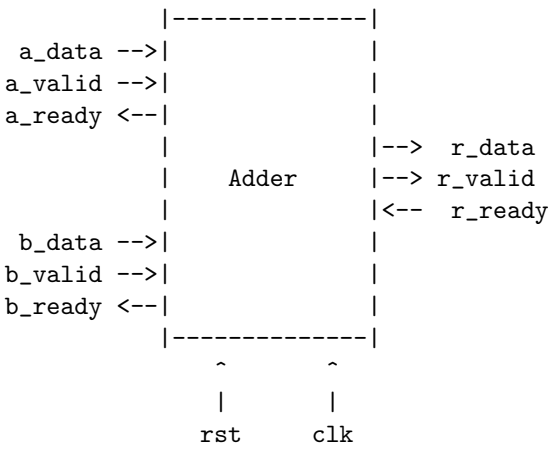
1 Introducción

Este documento es un informe de los trabajos desarrollados con la finalidad de ser evaluados para formar parte del equipo de **Novo Space**. Consta con dos ejercicios que nos acercan al lenguaje de programación de **Python** y familiarizarse con tecnologías de interés para el prototipado y uso por parte de la empresa.

2 Primer ejercicio

2.1 Enunciado

En el diseño de una radio para satélites de comunicación, es necesario realizar operaciones algebraicas sobre fuentes de datos provenientes de diferentes antenas. Estas operaciones deben realizarse lo más rápido posible (500 millones de operaciones por segundo aproximadamente). Es por ello, que se decidió que el procesamiento de estos datos lo haga una **FPGA**. La primer operación que se tiene que resolver es la suma. Realizar un sumador con lógica de números signados complemento a 2 que cumpla con la siguiente interfaz:



Tanto las dos entradas como la salida cumplen un protocolo genérico stream con las siguientes características:

- Las señales `_data` tienen N bits definidas durante la instanciación.
- La cantidad de bits de `r_data` quedan a definir por el diseñador. Algunas posibles alternativas son:
  - A definir durante la instanciación
  - Igual que la entrada
  - Un bit mas que la entrada
- El dato `_data` es leído por el sumidero cuando `_valid` y `_ready` están en 1
- La señal `_valid` no puede depender depender de la señal `_ready` para ir a 1
- Todas los datos que salgan por el puerto r deben ser un resultado valido entre los datos del puerto a y puerto b. No se debe realizar corroboración de overflow.

2.2 Resolución

2.2.1 Introducción

En primera instancia se realizó una investigación de como operan los **FPGA**, sus características, por qué son de nuestro interés y como es que se utilizan. En gran síntesis, los **FPGA** (*Field-Programmable Gate Array*) como bien lo describe su nombre, son dispositivos programables que cuentan con una matriz de bloques

lógicos interconectados. La programación de estos dispositivos se realiza mediante un lenguaje de descripción especializado. Las principales características son su alta velocidad no por utilizar frecuencias de clock del orden de los GHz como los **CPU** y **GPU**, si no por su particularidad de procesamiento en paralelo, su baja disipación de potencia y cantidad de puertos I/O. Los lenguajes de programación para **FPGA** son conocidos como **HDL** (*Hardware Description Language*), siendo los más utilizados: **VHDL**, **Verilog** y **ABEL**. El que nos representa mayor interés es **Verilog**, no porque lo utilicemos directamente para la resolución del ejercicio, ya el *framework* que utilizamos en **Python** interpreta lo descrito en este lenguaje y genera un archivo **Verilog**. Esta herramienta se llama **nMigen** y suele ser más amigable a la hora de encontrarse por primera vez frente a la programación de un **FPGA** ya que se programa sobre **Python**. A demás se utilizó el *framework* **Cocotb** para la verificación y el simulador **GTKWave**. El génesis del ejercicio consta de la instalación de todos los paquetes.

2.2.2 Análisis del Example.py

Debido a la juventud de **nMigen** y su constante desarrollo, la documentación oficial no se encuentra finalizada. Por lo que gran parte de la investigación se realizó en foros y trabajos en repositorio, que serán citados al final. Luego se realizó un análisis del ejemplo. Se trata de un incrementador sincrónico, que posee un puerto de entrada y otro de salida. Es de gran interés la clase `Stream` ya que tiene el mismo principio de funcionamiento del `Adder` que se requiere diseñar: El método `Send` envía el dato de entrada esperando que `ready` de ese puerto se encuentre `HIGH`. Esto se va a utilizar para los puertos de entrada. El otro atributo `recv` recibe los datos del **FPGA** siempre que se encuentre válido el *Flag* de `valid`. El resto del ejercicio va a ser explicado en esta sección ya que es muy similar al sumador.

2.2.3 Desarrollo del sumador

A partir de la clase `Stream` sustraída del ejemplo, se definió una nueva clase `Adder`, con sus atributos a, b, r del tipo `Stream` correspondientes a los puertos de entrada y salida y una tercer variable `aux` que será de gran ayuda para localizar la condición lógica en la que alguno de los puertos no se encuentra válido y no se produzca una perdida de datos o defasaje entre puertos de entrada.
Luego un el método `elaborate` (que es llamado por **nMigen**) se tienen un dominio sincrónico y otro combinacional. Si no se utilizaría el protocolo genérico de *stream*, aquí solo sumaríamos los datos de `a_data` + `b_data` y los colocaríamos en `r_data`, en cualquiera de los dos dominios. Pero como esos datos deben ser validados previamente, se realizan las condiciones lógicas que me permiten leer los datos de los puertos de entrada y no enviarles el *flag* de *ready* hasta que ambos valores sean válidos y el bloque próximo al puerto r envíe el *flag* de `r_ready`. Cuando los puertos de entrada se encuentran aceptados el puerto de salida será válido y el próximo bloque recibirá el dato. Se utiliza la señal `aux` ya que de no utilizarla, se producía un retraso inicial de 2 periodos en el caso de tener un *flag low* de *valid* en alguno de los puertos de entrada.

2.2.4 Simulación

Luego al bloque de simulación, para ello se utilizó el *framework* **Cocotb**, nos permite utilizar funciones como generador de *clock* o detección de flancos.
Para la simulación, se definen una señal de *clock*, en este caso de 10ns, y se resetea el sumador. En la simulación se busca corroborar el correcto funcionamiento, por ello se enviarán 2 ráfagas de datos, con tamaño $N = 2^{width-1}$ de la siguiente manera:

Primera ráfaga

- Señal en el puerto A: [1, 2, 3, ..., 1+N] = $(a_1, a_2, a_3, ..., a_n) \in Z^{1xN} / [2^{width-1} <= a_i < 2^{width-1}]$
- Señal en el puerto B: [2, 2, 2, ..., 2] = $(b_1, b_2, b_3, ..., b_n) \in Z^{1xN} / [2^{width-1} <= b_i < 2^{width-1}]$

Segunda ráfaga

- Señal en el puerto A: [2, 2, 2, ..., 2] = $(a_1, a_2, a_3, ..., a_n) \in Z^{1xN} / [2^{width-1} <= a_i < 2^{width-1}]$
- Señal en el puerto B: [1, 2, 3, ..., N] = $(b_1, b_2, b_3, ..., b_n) \in Z^{1xN} / [2^{width-1} <= b_i < 2^{width-1}]$

Y se espera recibir

- Señal en el puerto R: [3, 4, 5, ...] = $(r_1, r_2, r_3, ..., r_n) \in Z^{1x2.N} / [2^{width-1} <= r_i < 2^{width-1}]$

Algunas consideraciones

- La cantidad de bits a la salida se definió igual a la de los puertos de entrada ya que se trata de números signados complemento a 2 y el módulo de los números negativos varía en función de la cantidad de bits. Estos pueden tomar los valores $2^{width-1} <= a_i < 2^{width-1}$.

- El sumador no comprueba *overflow*, por lo que el resultado será coherente si la suma de `data_a` y `data_b` no supere los límites del espacio.
- En la simulación se realizan sumas de valores que superan los límites del espacio. Por eso se realiza un truncamiento en base 2 de los valores esperados al tamaño *width* utilizado.
- Para probar condiciones criticas en las que alguno de los puertos de entrada no posea su *flag* de *valid* se realizó un retraso entre los envíos de datos. Se envían primero los datos al puerto A, luego se espera un ciclo de *clock* y se envían los datos al puerto de B. Esto no debería producir perdida de datos o defasaje entre las señales de los puerto de entrada.
- Se envían y esperan dos ráfagas de datos para corroborar que los *flags* del puerto de salida tomen los valores que corresponde.
- La señal esperada es la suma de ambas señales `a_data + b_data` truncada en base 2 del tamaño de los puertos. Una ráfaga seguida de la otra.

2.2.5 Resultados de la simulación

```
Seeding Python random module with 1620797054
Found test adder.burst
Running test 1/1: burst
Starting test: "burst"
Description: None
Test Passed: burst
Passed 1 tests (0 skipped)
*******************************************************************
** TEST         PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
*******************************************************************
** adder.burst    PASS          690.00          0.02     33751.31  **
*******************************************************************

*******************************************************************************

**                                ERRORS : 0

*******************************************************************************

**                               SIM TIME : 690.00 NS

**                              REAL TIME : 0.15 S

**                         SIM / REAL TIME : 4654.54 NS/S

*******************************************************************************
```

Abbildung 1: Devolución de la verificación formal **Cocotb**

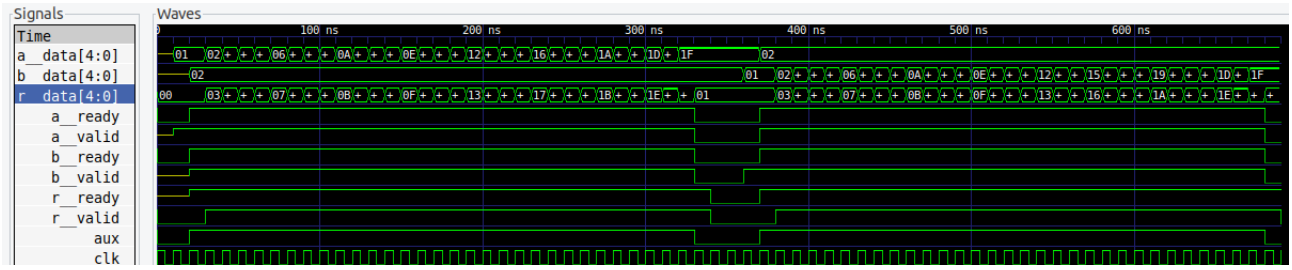


Abbildung 2: Rango completo

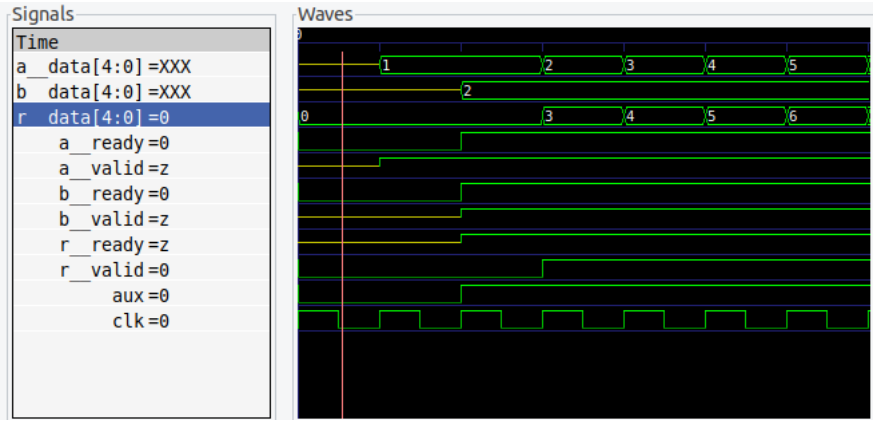


Abbildung 3: Inicio t(0)

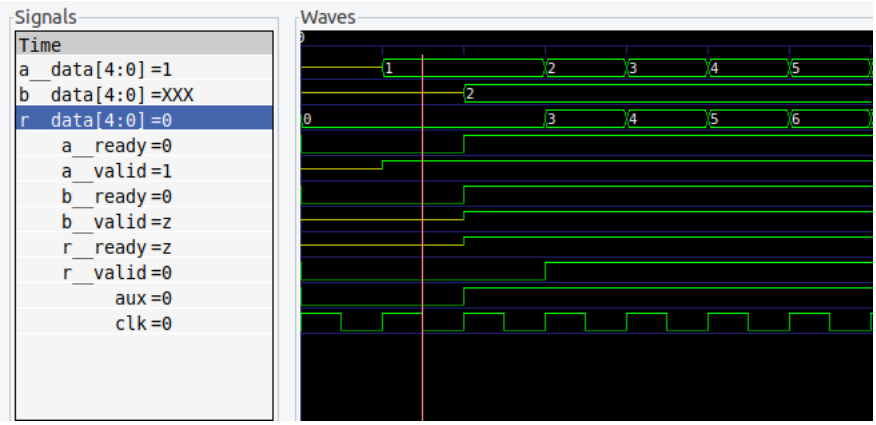


Abbildung 4: Enviado `data_a` esperando a `data_b` t(1)

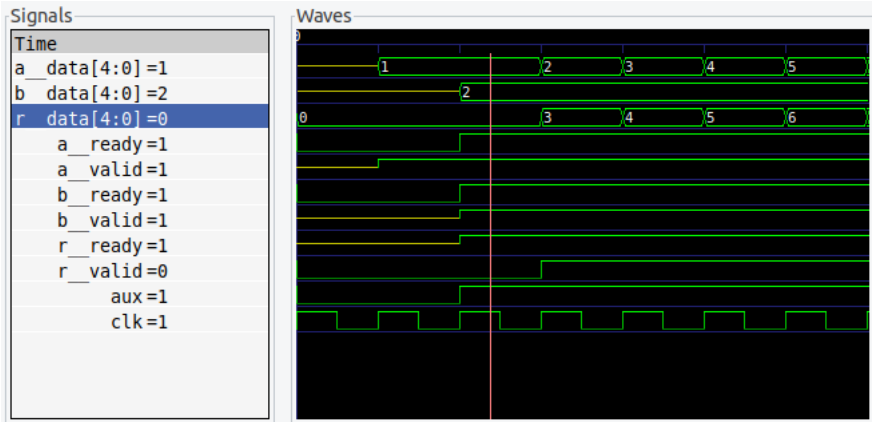


Abbildung 5: Enviados `data_a` y `data_b` esperando estar listos para escuchar t(2)

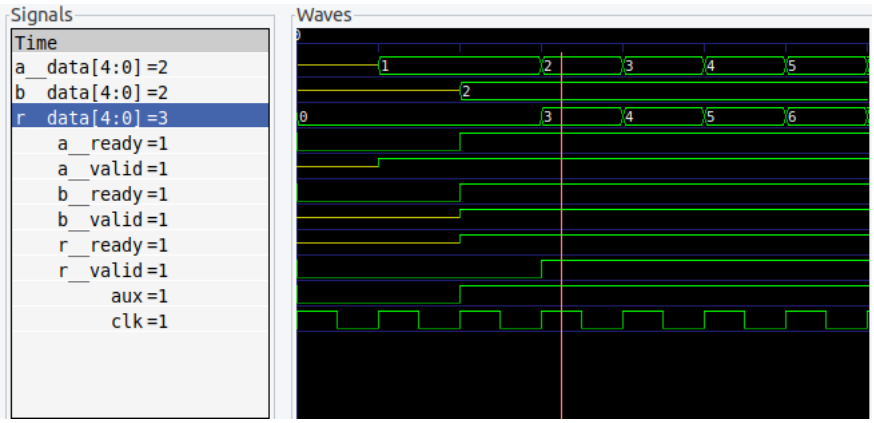


Abbildung 6: Primera respuesta válida del sumador t(3)

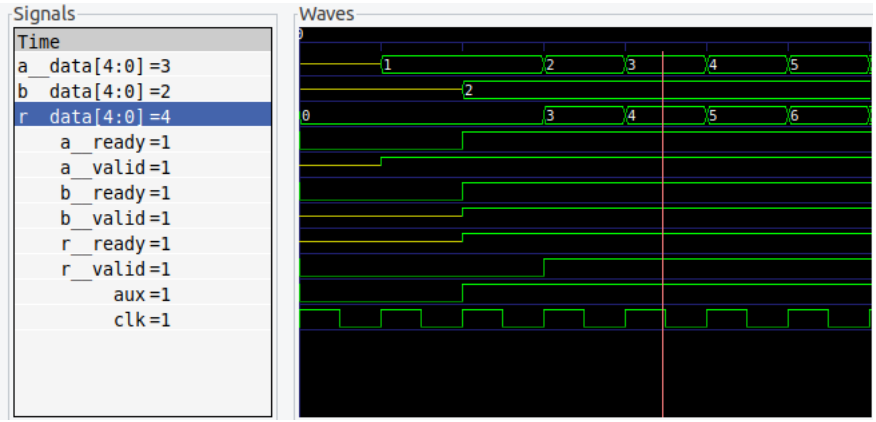


Abbildung 7: Segunda respuesta válida del sumador t(4)

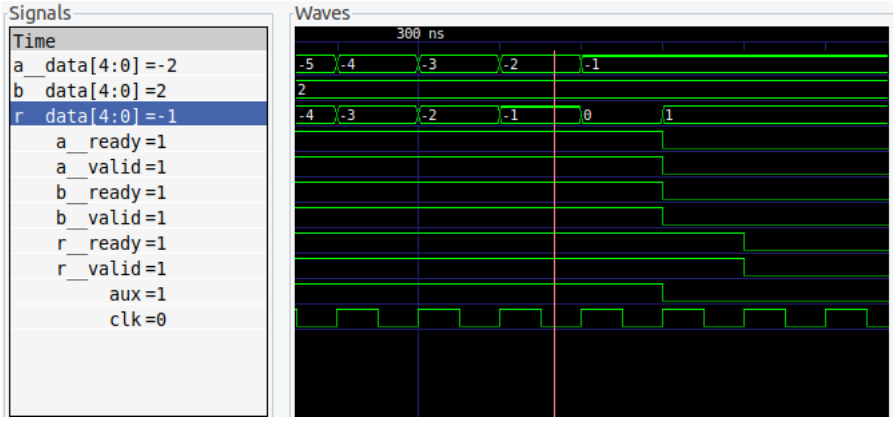


Abbildung 8: *Overflow*

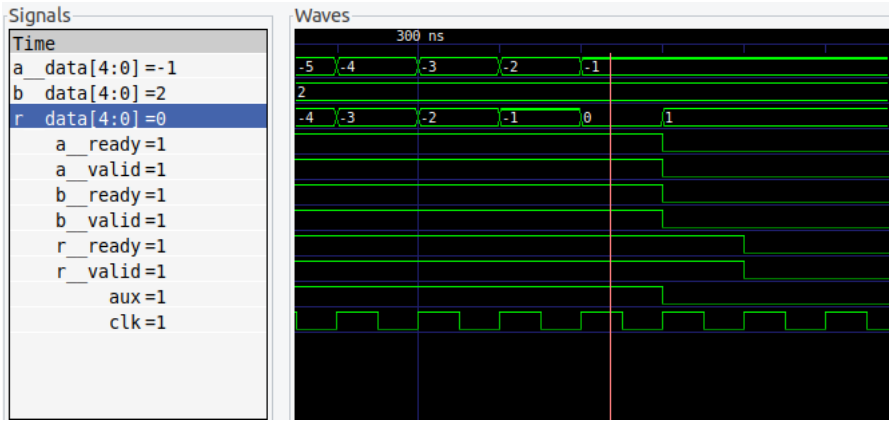


Abbildung 9: Ultima entrada de la primera ráfaga

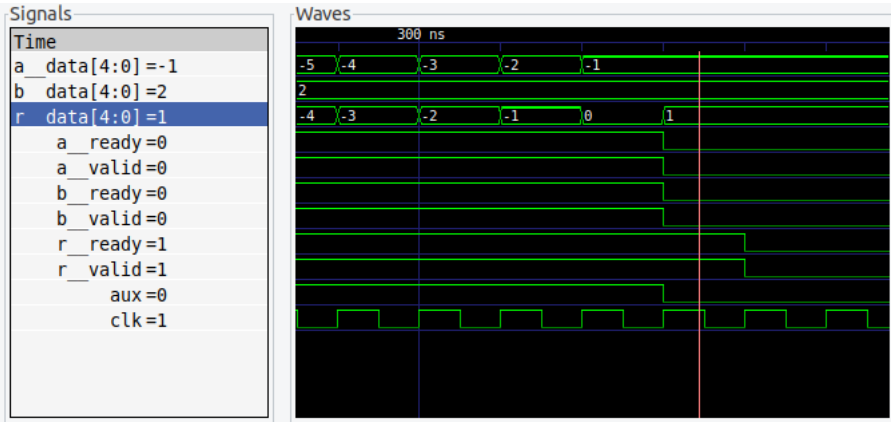


Abbildung 10: Última respuesta de la primera ráfaga

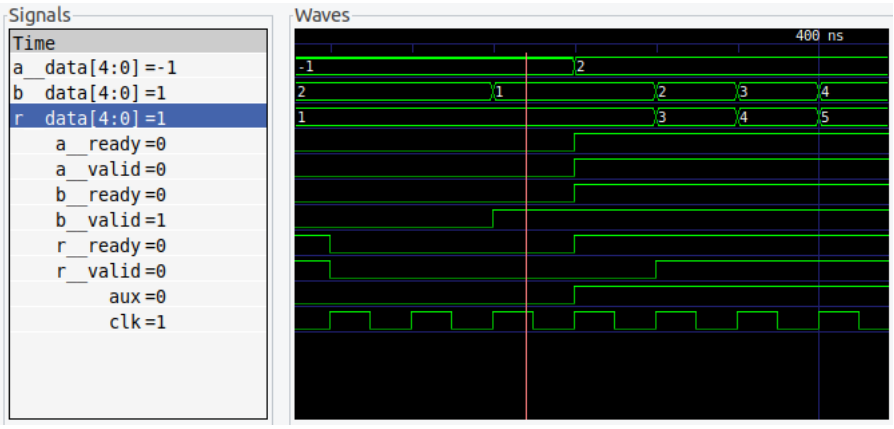


Abbildung 11: Segunda ráfaga **data_b** primero

3 Segundo ejercicio

3.1 Enunciado

Muchas veces las herramientas que se utilizan en el día a día no son suficientes para lo que se desea hacer y hay que realizar nuestras propias herramientas o agregar un *layer* de adaptación con otra ya existente. Actualmente, las herramientas de un *vendor* de **FPGA** no soportan completamente la sintaxis de **verilog** lo cual hace incompatible el *output* de **nMigen**. Más puntualmente no soporta la inicialización *inline* de memorias:

Sintaxis no soportada

```
reg [7:0] mem [15:0];
initial begin
  mem[0] = 8'h3d;
  mem[1] = 8'hc9;
  mem[2] = 8'hbf;
  mem[3] = 8'hd5;
  mem[4] = 8'h52;
  mem[5] = 8'he0;
  mem[6] = 8'h05;
  mem[7] = 8'hc8;
  mem[8] = 8'hbc;
  mem[9] = 8'h6e;
  mem[10] = 8'h98;
  mem[11] = 8'h9f;
  mem[12] = 8'h4b;
  mem[13] = 8'h4c;
  mem[14] = 8'h39;
  mem[15] = 8'h55;
end
```

Sinaxis soportada

```
reg [7:0] mem [15:0];
$readmemh("memdump0.mem", mem);
```

memdump0.mem

```
3d
c9
bf
d5
52
e0
05
c8
bc
6e
98
9f
4b
4c
39
55
```

Se pide desarrollar un *script* de **Python** que reemplace las estructuras de inicialización *inline* por la inicialización soportada por la herramienta y exporte los archivos con los valores de inicialización. Se otorga un *testcase* (testcase.v) y los resultados esperados (en la carpeta expected).

Regex de ayuda

```
r'  reg \[(.*)\] (\S*) \[(.*)\];\n  initial begin\n((    \S*\[\S*\] = \S*;\n)*)  end\n'
```

3.2 Resolución

Analizando el regex que nos dieron de ayuda, se modificó de forma tal que un grupo me quede:

```
reg [7:0] mem [15:0];
```

Ya que este no se modificaría en la sintaxis soportada. La expresión regular que utilicé fue:

```
'(  reg \[(.*)\] (\S*) \[(.*)\];\n)*  initial begin\n((    \S*\[\S*\] = \S*;\n)*)  end\n'
```

Con la ayuda de la librería `argparse` se define el nombre del archivo **verilog** a modificar. Cuando se ejecuta el *script* se puede definir el nombre del archivo que contenga el **Verilog** y el nombre del nuevo archivo con la sintaxis corregida. El *help* del *script* es lo suficiente descriptivo:

```
python3 ej2.py --h
usage: ej2.py [-h] [--in_file IN_FILE] [--out_file OUT_FILE]

EJ:2 This script is responsible for modifying the nmigen output file by
changing the syntax type of memory initializations.

optional arguments:
  -h, --help           show this help message and exit
  --in_file IN_FILE    Enter the in file name with its extension, default case
                       "testcase.v"
  --out_file OUT_FILE  Enter the out file name with its extension, default
                       case "fixed_IN_FILE"
```

Una vez definidos los nombres de archivos `IN_FILE` y `OUT_FILE`, se abre el archivo `IN_FILE` en modo lectura y se escribe su contenido en un *string* el cual se le pasa a una función junto con el nombre de `OUT_FILE`. Esta función se encarga de buscar y la sintaxis no soportada y sustituirla por la soportada, generando a su vez el archivo `memdump0.mem` con el contenido de la memoria. Para esto se utiliza la librería de expresión regular **re** con su método `search` que devuelve los grupos encontrados y las pociones iniciales y finales en el texto original. Gracias a las posiciones del grupo principal se sustituye la sintaxis no soportada por el contenido de `result.groups()[0]` que contiene la primera linea de la sintaxis soportada. Para la segunda linea se realiza una concatenación de los *string* constantes a los variables, y esa información se toma de los grupos de repuesta siendo **mem** y el numero de memoria las variables, ya que podría llamarse de otra forma o poseer mas de una inicialización. Luego utilizando el método `re.findall` se buscan todos los valores de inicialización de memoria y se escriben en un archivo con el nombre correspondiente. Esta operación se repite hasta que no haya coincidencia con la sintaxis no soportada. En el caso de obtener un nuevo *match* se incrementa el numero de memoria (i) que luego se utilizará para nombrar al archivo que contenga sus valores.

- ”Muchas gracias por la propuesta, el seguimiento y la devolución en los avances

---

4 Referencias

- https://github.com/nmigen/nmigen
- https://nmigen.info/nmigen/latest/intro.html
- https://github.com/RobertBaruch/nmigen-tutorial
- https://www.youtube.com/watch?v=yDJNwxY05-s
- https://www.youtube.com/channel/UCBcljXmuXPok9kT_VGA3adg
- https://www.intel.com/content/www/us/en/programmable/documentation/lit-index.html
- https://indico.cern.ch/event/357886/contributions/849365/attachments/1145718/1648596/FPGA_1.pdf

5 Anexo

5.1 adder.py

```
from nmigen import *
from nmigen_cocotb import run
import cocotb
from cocotb.triggers import RisingEdge
from cocotb.clock import Clock

class Stream(Record):
    def __init__(self, width, **kwargs):
        Record.__init__(
            self, [('data', width), ('valid', 1), ('ready', 1)], **kwargs)

    def accepted(self):
        return self.valid & self.ready

    class Driver:
        def __init__(self, clk, dut, prefix):
            self.clk = clk
            self.data = getattr(dut, prefix + 'data')
            self.valid = getattr(dut, prefix + 'valid')
            self.ready = getattr(dut, prefix + 'ready')

        # To the adder
        async def send(self, data):
            self.valid <= 1
            for d in data:
                self.data <= d
                await RisingEdge(self.clk)
                while self.ready.value == 0:
                    await RisingEdge(self.clk)
            self.valid <= 0

        # From the adder
        async def recv(self, count):
            self.ready <= 1
            data = []
            for _ in range(count):
                await RisingEdge(self.clk)
                while self.valid.value == 0:
                    await RisingEdge(self.clk)
                data.append(self.data.value.integer)
            self.ready <= 0
            return data


class Adder(Elaboratable):
    def __init__(self, width):
        self.a = Stream(width, name='a')
        self.b = Stream(width, name='b')
        self.r = Stream(width, name='r')
        self.aux = Signal(1)

    def elaborate(self, platform):
        m = Module()
        sync = m.d.sync
        comb = m.d.comb

        with m.If(self.r.accepted()):
            sync += self.r.valid.eq(0)
```

```
            comb += self.aux.eq(0)

        with m.If(self.a.valid & self.b.valid & self.r.ready):
            sync += self.r.data.eq(self.a.data + self.b.data)
            comb += self.aux.eq(1)


        with m.If(self.aux & self.a.accepted() & self.b.accepted()):
            sync += self.r.valid.eq(1)

        comb += self.a.ready.eq(self.a.valid & self.b.valid & self.aux)
        comb += self.b.ready.eq(self.a.valid & self.b.valid & self.aux)
        return m

# Simulation block
async def init_test(dut):
    cocotb.fork(Clock(dut.clk, 10, 'ns').start())
    dut.rst <= 1
    await RisingEdge(dut.clk)
    await RisingEdge(dut.clk)
    dut.rst <= 0

@cocotb.test()
async def burst(dut):
    await init_test(dut)

    stream_input_a = Stream.Driver(dut.clk, dut, 'a__')
    stream_input_b = Stream.Driver(dut.clk, dut, 'b__')
    stream_output = Stream.Driver(dut.clk, dut, 'r__')

    width = len(dut.a__data)
    N = (2 ** width-1)
    mask = int('1' * width, 2)
    sign_a = [1+j for j in range(N)]
    sign_b = [2 for _ in range(N)]
    expected_1 = []

    # The first sign i expect
    for i in range(0, N):
        expected_1 = expected_1 + [(sign_a[i]+sign_b[i]) & mask]
    # I must do the truncation (& mask) by the size of the variables

    # For the validation test, I want to prove that if any of
    # the ports is not valid, the data is not lost. For that use the line
    # "await RisingEdge(dut.clk)" delaying the sending of signals

    # Send the sign_a to the port a
    cocotb.fork(stream_input_a.send(sign_a))

    # wait a period
    await RisingEdge(dut.clk)

    # Send the sign_b to the port b
    cocotb.fork(stream_input_b.send(sign_b))

    # wait for the first output sign
    recved_1 = await stream_output.recv(N)

    # wait two periods
    await RisingEdge(dut.clk)
    await RisingEdge(dut.clk)
```

```
    # Swap the signals for test
    # Send the sign_b to the port a
    cocotb.fork(stream_input_b.send(sign_a))

    # wait a period
    await RisingEdge(dut.clk)
    # Send the sign_b to the port a
    cocotb.fork(stream_input_a.send(sign_b))

    # wait for the second output sign
    recved_2 = await stream_output.recv(N)

    # Test codition formal verification
    assert recved_1 + recved_2 == expected_1 + expected_1


if __name__ == '__main__':
    core = Adder(5)
    run(
        core, 'adder',
        ports=[
            *list(core.a.fields.values()),
            *list(core.b.fields.values()),
            *list(core.r.fields.values())
        ],
        vcd_file='./VCD/adder.vcd'
    )
```

5.2 ej2.py

```
import re
import sys
import argparse

PATTERN = re.compile(r'(  reg \[(.*)\] (\S*) \[(.*)\];\n)*  initial begin\n((    \S*\[\S*\] = \S*;\n)*


def fix_file(file_contents, out_file):
    '''
        This function looks for the unsupported syntax
    type in the file_contents string and sets the correct syntax by creating
    a .mem file for each memory record it finds. The inputs are the string
    file_contents and the name of the output file.
    '''

    # Number of memories
    i = 0
    result = PATTERN.search(file_contents)
    while result:
        info = result.groups()
        mem_name = info[2]
        mem_file_name = 'memdump' + str(i) + '.mem'

        fixed_syntax = info[0] + '  $readmemh(\"' + mem_file_name + '\", ' + mem_name + ');\n'

        # Search for the initial values of the memory
        mem_values = re.findall(r"= 8'h(\w*);", info[4])

        with open(mem_file_name,'w+') as mem_file:
            for value in mem_values:
                mem_file.write(value + '\n')

        file_contents = file_contents[0:result.start()] + fixed_syntax + file_contents[result.end() ::]
        result = PATTERN.search(file_contents)
        i += 1
    print(f'Modified memory blocks number: {i}')

    with open(out_file_name, 'w') as out_file:
        out_file.write(file_contents)

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='EJ:2 This script is responsible for modifying
    the nmigen output file by changing the syntax type of memory initializations. ')
    parser.add_argument('--in_file', default = 'testcase.v',
    help = 'Enter the in file name with its extension, default case \"testcase.v\"')
    parser.add_argument('--out_file',help = 'Enter the out file name with its extension,
    default case \"fixed_IN_FILE\"')

    args = parser.parse_args()

    out_file_name = args.out_file

    if out_file_name is None:
        out_file_name = 'fixed_' + args.in_file

    with open(args.in_file, 'r') as in_file:
        file_contents = in_file.read()

    fix_file(file_contents, out_file_name)
```