riscy

May 2, 2025

1 Base Riscy Notebook

This notebook is supposed to exemplify how to set up and launch the execution of your core in the FPGA.

It is recommended to make a duplicate of this notebook that you can edit and change all the paths/names from `base_riscy` to `riscy` so you can always keep the initial version of both the overlay and the control of the core in a working state.

1.1 Initialization of the notebook

A series of packages need to be imported, as well as defining a few variables and functions

[1]:
```
from pynq import Overlay
from pynq import PL
from pynq import MMIO

zynq_system = Overlay("/home/xilinx/jupyter_notebooks/riscy/overlays/
↪riscv_wrapper.bit", download=False)

ins_mem       = MMIO(0x40000000, 0x8000)
data_mem      = MMIO(0x42000000, 0x8000)
riscv_control = MMIO(0x40008000, 0x1000)
reg_bank      = MMIO(0x40009000, 0x1000)

#Register Bank Addresses:
end_of_test_addr = 0x0 #only bit 0 out of 32.
exec_clk_cycles = 0x4

#Data mem addresses:
end_seq_addr = 0x2000
c_result_check_addr = 0x2004
expected_result_addr = 0x2030
calculated_result_addr = 0x2040

def parse_coe_file(file_path):
    data_values = []
    with open(file_path, 'r') as file:
        lines = file.readlines()
        start_parsing = False

        for line in lines:
            line = line.strip()

            if start_parsing:
                values = line.split(',')
                data_values.extend([v.strip() for v in values if v.strip()])

            if "memory_initialization_vector=" in line:
                start_parsing = True
                line = line.split("=")[1]  # Get the values after '='
                values = line.split(',')
                data_values.extend([v.strip() for v in values if v.strip()])
                print(data_values)

    return data_values

def parse_and_process_file(filename, write_func):
    data_values = parse_coe_file(filename)
    count = 0
    offset = 0x0

    #print(f"Length of memory file: {len(data_values)} x 32-bits")
    while count < len(data_values):
        write_func(offset, int(data_values[count], 16))
        #print(f"Write {hex(offset)}: {hex(int(data_values[count], 16))}")
        offset += 0x4
        count += 1
```

```
/usr/local/lib/python3.6/dist-packages/pynq/pl_server/device.py:594:
UserWarning: Users will not get PARAMETERS / REGISTERS information through TCL
files. HWH file is recommended.
  warnings.warn(message, UserWarning)
```

1.1.1 Downloading the FPGA image (bitstream)

Now we will download the bistream into the PL of the FPGA:

[2]:
```
zynq_system.download()
```

1.2 Control of the RISCY Core

Two top level pins of the RISCY core are connected to a control block (reboot_riscv_0 in the bd).

This block has a base address of `0x40008000`, but the relative offset of the 32-bit register controlling the connected riscv instance pins is `0x10`. This 32-bit register is connected as follows:

- bit 4 = fetch_enable port

- bit 0 = reboot port

The default value of the control register is: `0x00000000`.

1.2.1 Rebooting the RISCY Core

We will start by asserting high and deasserting the register bit that is connected to the reboot pin of the riscv core:

[3]:
```
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
riscv_control.write(0x10, 0x00000001)
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
riscv_control.write(0x10, 0x00000000)
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
```

```
Read: 0x0
Read: 0x1
Read: 0x0
```

1.2.2 Programming the instruction memory

Now that the riscv core is rebooted and stalling we can proceed to write our desired initialization sequences in the instruction memory (blk_mem_gen_1):

[4]:
```
parse_and_process_file("/home/xilinx/jupyter_notebooks/riscy/mem_files/code.
  ↪coe", ins_mem.write_reg)
```

```
['00000013']
```

[5]:
```
# We can also read the instruction memory:
data=0x0
addr=0x0
for i in range(20):
    data = ins_mem.read(addr)
    print(f"Address is: {hex(addr)}; Value is: {hex(data)}.")
    addr += 0x4
```

```
Address is: 0x0; Value is: 0x13.
Address is: 0x4; Value is: 0x13.
Address is: 0x8; Value is: 0x13.
Address is: 0xc; Value is: 0x13.
Address is: 0x10; Value is: 0x13.
Address is: 0x14; Value is: 0x13.
Address is: 0x18; Value is: 0x13.
Address is: 0x1c; Value is: 0x13.
Address is: 0x20; Value is: 0x13.
Address is: 0x24; Value is: 0x13.
Address is: 0x28; Value is: 0x13.
Address is: 0x2c; Value is: 0x13.
Address is: 0x30; Value is: 0x13.
Address is: 0x34; Value is: 0x13.
Address is: 0x38; Value is: 0x13.
Address is: 0x3c; Value is: 0x13.
Address is: 0x40; Value is: 0x13.
Address is: 0x44; Value is: 0x13.
Address is: 0x48; Value is: 0x13.
Address is: 0x4c; Value is: 0x13.
```

1.2.3 Programming the data memory

Same as before with the data memory (blk_mem_gen_2 in the bd):

[6]:
```
parse_and_process_file("/home/xilinx/jupyter_notebooks/riscy/mem_files/data.
↪coe", data_mem.write_reg)
```

```
['7B777C63']
```

[7]:
```
# We can also read the data memory:
data=0x0
addr=0x0
for i in range(20):
    data = data_mem.read(addr)
    print(f"Address is: {hex(addr)}; Value is: {hex(data)}.")
    addr += 0x4
```

```
Address is: 0x0; Value is: 0x7b777c63.
Address is: 0x4; Value is: 0xc56f6bf2.
Address is: 0x8; Value is: 0x2b670130.
Address is: 0xc; Value is: 0x76abd7fe.
Address is: 0x10; Value is: 0x7dc982ca.
Address is: 0x14; Value is: 0xf04759fa.
Address is: 0x18; Value is: 0xafa2d4ad.
Address is: 0x1c; Value is: 0xc072a49c.
Address is: 0x20; Value is: 0x2693fdb7.
Address is: 0x24; Value is: 0xccf73f36.
Address is: 0x28; Value is: 0xf1e5a534.
Address is: 0x2c; Value is: 0x1531d871.
Address is: 0x30; Value is: 0xc323c704.
Address is: 0x34; Value is: 0x9a059618.
Address is: 0x38; Value is: 0xe2801207.
Address is: 0x3c; Value is: 0x75b227eb.
Address is: 0x40; Value is: 0x1a2c8309.
Address is: 0x44; Value is: 0xa05a6e1b.
Address is: 0x48; Value is: 0xb3d63b52.
Address is: 0x4c; Value is: 0x842fe329.
```

1.2.4 Starting the execution of the RISCY Core

Now we can proceed to assert high to the fetch enable pin of the RISCY core:

[8]:
```
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
riscv_control.write(0x10, 0x0000_0010)
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
```

```
Read: 0x0
Read: 0x10
```

1.2.5 Check the results of the execution of the core

First we check that the core finished execution. Then we check expected and calculated results by the core. Core also compares them and saves a sequence to indicate if the match or not. Check numnber of clk cycles it took to execute the C code.

[9]:
```
data = reg_bank.read(end_of_test_addr)
if data == 1:
    print("Test execution finished!")
else:
    print("Test execution DID NOT finish!")
```

```
Test execution finished!
```

[10]:
```
for i in range(4):
    data = data_mem.read(expected_result_addr + (i * 4))
    print(f"[{i}] Expected result: {hex(data)}")
```

```
[0] Expected result: 0xfba50914
[1] Expected result: 0x714bf41f
[2] Expected result: 0x2e25aabe
[3] Expected result: 0xaaf9080f
```

[11]:
```
for i in range(4):
    data = data_mem.read(calculated_result_addr + (i * 4))
    print(f"[{i}] Calculated result: {hex(data)}")
```

```
[0] Calculated result: 0xfba50914
[1] Calculated result: 0x714bf41f
[2] Calculated result: 0x2e25aabe
[3] Calculated result: 0xaaf9080f
```

[12]:
```
data = data_mem.read(c_result_check_addr)
print(f"C code result check: {hex(data)}")
print(f"0xcafebabe = match; 0xbaaaaaad = no match.")
```

```
C code result check: 0xcafebabe
0xcafebabe = match; 0xbaaaaaad = no match.
```

[13]:
```python
data = reg_bank.read(exec_clk_cycles)
print(f"It took: {data} clk cycles.")
```

```
It took: 161441 clk cycles.
```