IOB-SoC

A RISC-V-based System on Chip

IObundle Lda

July 21, 2020

Outline

- Introduction
- Project setup
- Create an IP core to instantiate in your SoC
- Edit the ./system.mk configuration file to declare a new peripheral
- Edit file `firmware.c` to drive the new peripheral
- Run the firmware in internal SRAM
- Run the firmware in external DDR
- Simulate and implement the system
- Implement in FPGA
- Implement in ASIC (WIP)
- Implement in RTL simulation
- Conclusions and future work

Introduction

- Building processor-based systems from scratch is challenging
- The IOB-SoC template eases this task
- Provides a base Verilog SoC equipped with
  - a RISC-V CPU
  - a memory system including boot ROM, RAM and AXI4 interface to DDR
  - a UART communications module
- Users can add IP cores and software to build more complex SoCs
- Here, the addition of a timer IP and its software driver is exemplified

Project setup

- Use a Linux machine or VM
- Install the latest stable version of the open source Icarus Verilog simulator (`iverilog.icarus.com`)
- Make sure you can access `github.com` using an ssh key
- At `github.com` create your SoC repository by cloning `github.com/IObundle/iob-soc`
- Follow the instructions in the README file to clone the repository in your Linux machine

Create an IP core to instantiate in your SoC

- Create a timer IP core repository or, alternatively, use the one at www.github.com/IObundle/iob-timer.git
- An IP core can be integrated in an IOb-SoC if it provides the following 2 files:
  1. hardware/hardware.mk
  2. software/embedded/embedded.mk
- Add the IP core repository as a git submodule of your IOb-SoC repository:

```
git submodule add https://github.com/IObundle/iob-timer.git submodules/TIMER
```

- To configure the system to host the IP core, edit the `./system.mk` file as in the next slide

Edit the ./system.mk configuration file to declare a new peripheral

```
#FIRMWARE
FIRM_ADDR_W:=13

#SRAM
SRAM_ADDR_W=13

#DDR
USE_DDR:=0
RUN_DDR:=0
DDR_ADDR_W:=30
CACHE_ADDR_W:=24

#ROM
BOOTROM_ADDR_W:=12

#Init memory (only works in simulation or FPGA not running DDR)
INIT_MEM:=1

#Peripheral list (must match respective submodule name)
PERIPHERALS:=UART TIMER

...
```

Edit the `firmware.c` file to drive the new peripheral

`./software/firmware/firmware.c`

```
#include "system.h"
#include "periphs.h"
#include "iob-uart.h"
#include "iob_timer.h"

int main()
{
  unsigned long long elapsed;
  unsigned int elapsedu;

  //read current timer count, compute elapsed time
  elapsed = timer_get_count(TIMER_BASE);
  elapsedu = timer_time_us(TIMER_BASE);

  //init uart
  uart_init(UART_BASE, FREQ/BAUD);

  uart_printf("\nHello world!\n");

  uart_txwait();

  uart_printf("\nExecution time: %d clocks in %dus @%dMHz (%d MBaud)\n\n",
              (unsigned int)elapsed, elapsedu, FREQ/1000000, BAUD/1000000);

  uart_txwait();
  return 0;
}
```

Run the firmware in internal SRAM

1. Run the firmware in internal RAM and disable (re)programming
   - Assign `USE_DDR=0` and `INIT_MEM=0`
   - Loading programs after the FPGA is programmed is disabled: if the firmware is modified the FPGA must be recompiled
   - This option is only valid for FPGA which permits memory initialisation
2. Run the firmware in internal RAM and enable (re)programming
   - Assign `USE_DDR=0 INIT_MEM=1`
   - Loading programs after the FPGA is programmed is enabled
   - This option is valid for FPGA and ASIC
   - Firmware is (re)loaded via UART

Run the firmware in external DDR

1. Run the firmware in external DDR and disable (re)programming
   - Assign `USE_DDR=1` and `INIT_MEM=0`
   - This option is only allowed in simulation which permits memory initialisation
   - An FPGA or ASIC implementation will not work
2. Run the firmware in external DDR memory and enable (re)programming
   - Define `USE_DDR=1 INIT_MEM=1`
   - This option is valid for FPGA and ASIC
   - Firmware is (re)loaded via UART
   - Third party DDR controller IP core is required

Simulate and implement the system

- To simulate the system just type `make`
- The firmware, bootloader and system verilog description are compiled as you can see from the printed messages
- The last prints should look like the following

```
IOb-SoC Bootloader:

Reboot CPU and run program...

Hello world!

Execution time: 6583 clocks in 66us @100MHz (30 MBaud)
```

Implement in FPGA

- Add your FPGA folder in `./hardware/fpga`. Use the other folders in there as examples
- In the file `./system.mk`:
  1. Add your FPGA board name in `FPGA_BOARD`
  2. Add your server URL connected to the board in `FPGA_BOARD_SERVER`
  3. Add further server info such as username and work directories
- To implement and load the hardware design in the FPGA, type `make fpga-load`
- To load and run your firmware in the FPGA, type `make run-firmware`

Implement in ASIC (WIP)

- Add your ASIC folder in `./hardware/asic`. Use the other folders in there as examples
- In the file `./system.mk`:
  1. Add your ASIC node name in `ASIC_NODE`
  2. Add your server URL connected to the microcontroller in `ASIC_SERVER`
  3. Add further server info such as username and work directories
- To implement and run the ASIC, type `make asic`

Implement in RTL simulation

- Add your simulation folder in `./hardware/simulation`. Use the other folders in there as examples
- In the file `./system.mk`:
  1. Add your simulator name in `SIMULATOR`
  2. Add your server URL with the RTL simulator in `SIM_SERVER`
  3. Add further server info such as username and work directories
- To implement and run the RTL simulator, type `make`

Conclusions and future work

- Conclusions
  - A tutorial on SoC creation using IOb-SoC is presented
  - The addition of a peripheral IP core (timer) is illustrated
  - A simple software driver for the IP core is exemplified
  - How to compile and run the system is explained
  - Options for implementing the main memory are presented
  - Implementation of FPGA and ASIC is explained
- Future work
  - Non-volatile (flash) external memory support
  - Real Time Operating System (RTOS)