IOB-SoC

A RISC-V based System on Chip

José T. de Sousa

IObundle Lda

March 12, 2020

Outline

- Introduction
- Project setup
- Project editing guidelines
- Getting started with the timer IP hardware design
- Edit the hardware definitions file `system.vh`
- Instantiate the timer IP in file `rtl/src/system.v`
- Add the timer IP firmware
- Edit file `firmware.c` to drive the timer IP
- Edit the Makefile to compile the timer IP firmware
- Setup RTL simulation using the Icarus Verilog simulator
- Simulate the system
- Conclusions and future work

Introduction

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

Project setup

- Use a Linux machine virtual or not
- Install the latest stable version of the open source Icarus Verilog simulator
- Make sure you can access *github.io* and *bitbucket.org* using ssh keys
- Go to *bitbucket.org/jjts/iob-soc*
- Fork the repository to create your own remote repository
- Follow the README instructions to install the RISC-V toolchain if you have not already
- Create a directory for your project
  `>mkdir mysoc && cd mysoc`
- Follow the instructions in the README to clone the repository into your project directory

Project editing guidelines

- Your SoC will be inspired in IOB-SoC, not superimposed on it
  - You will partially reproduce the file hierarchy of the iob-soc repository by copying to the `mysoc` directory only the directories and files you need to change
- Do not edit any files in the iob-soc clone repository unless you want to contribute to it
- If you wish to submit fixes or improvements to the original iob-soc repository do the following:
  - Edit the relevant files in the iob-soc directory (clone)
  - Commit and push your changes to your fork
  - Submit a Pull Request to the original iob-soc repository
  - If you have benefited from IOB-SoC then also let others benefit from your work

Getting started with the timer IP hardware design

- Create the timer IP hardware or alternatively download the one from
  `git clone git@bitbucket.org:jjts/iob-timer.git`
- Copy the rtl directory from the iob-soc clone:
  `cp -r iob-soc/rtl .`
- Edit the system header file `./rtl/include/system.vh` as in the next slide

Edit the hardware definitions file system.vh

```
//
// HARDWARE DEFINITIONS
//

//Optional memories (passed as command line macro)
`define USE_BOOT
`define USE_DDR

// slaves
// minimum 4 slaves: boot, uart, ram and reset controller
// Optionally, to use DDR, you need 2 additional slaves: cache and cache controller
// ***NEW*** increment the number of slaves to host your new timer IP
`define N_SLAVES 5

//bits reserved to identify slave (the 2**N_SLAVES-1 combination is reserved and cannot be us
`define N_SLAVES_W 3

//peripheral address prefixes
`define BOOT_BASE 0
`define UART_BASE 1
`define SOFT_RESET_BASE 2
`define MAINRAM_BASE 3
`define CACHE_BASE 4
`define CACHE_CTRL_BASE 5
//***new*** base for timer
`define TIMER_BASE 6
...
```

Instantiate the timer IP in file rtl/src/system.v

```
`timescale 1 ns / 1 ps
`include "system.vh"

module system (
               input                 clk,
               input                 reset,
               ...
               ...
               ...

   //***NEW*** timer instance
   time_counter #(.COUNTER_WIDTH(32))
   timer (
          rst(reset_int),
          clk(clk),
          .addr(m_addr[2]),
          .data_in(m_wdata),
          .data_out(s_rdata[`TIMER_BASE]),
          .valid(s_valid[`TIMER_BASE]),
          .ready(s_ready[`TIMER_BASE])
    );

endmodule
```

Add the timer IP software

- Copy the software directory from the iob-soc clone:
  `cp -r iob-soc/software .`
- Edit the `./software/firmware/firmware.c` file as in the next slide

Edit file `firmware.c` to drive the timer IP

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

#define UART (UART_BASE<<(DATA_W-N_SLAVES_W))
#define SOFT_RESET (SOFT_RESET_BASE<<(ADDR_W-N_SLAVES_W))
#define TIMER (TIMER_BASE<<(ADDR_W-N_SLAVES_W))

int main()
{

  //***NEW*** variable to read timer cycle count
  int cycles = timer_get_count(TIMER_BASE);

  uart_init(UART,UART_CLK_FREQ/UART_BAUD_RATE);

  uart_printf("Hello world!\n");

  uart_txwait();

  //***NEW*** code section to read current timer count, compute and
  //print the elapsed time and clock frequency
  cycles = timer_get_count(TIMER) - cycles;

  uart_printf("Execution time: %dus @%dMHz,115200BAUD\n",(time*1000000)/UART_CLK_FREQ);

  uart_txwait();
  return 0;
}
```

Edit the Makefile to compile the timer IP firmware

```
TOOLCHAIN_PREFIX = riscv32-unknown-elf-
PYTHON_DIR := ../python/
SUBMODULES_DIR := ../../iob-soc/submodules
UART_DIR := $(SUBMODULES_DIR)/iob-uart/c-driver

#***NEW*** specify timer directory and addi it to include path
TIMER_DIR := ../../iob-timer
INCLUDE = -I. -I$(UART_DIR) -I$(TIMER_DIR)
DEFINE = -DUART_BAUD_RATE=$(BAUD) -DUART_CLK_FREQ=$(FREQ)

#***NEW*** add timer.c to the source list
SRC =  firmware.S firmware.c $(UART_DIR)/iob-uart.c  $(TIMER_DIR)/iob_timer.c

all: firmware.lds $(SRC) system.h $(UART_DIR)/iob-uart.h
	$(TOOLCHAIN_PREFIX)gcc -Os -ffreestanding  -nostdlib -o firmware.elf $(DEFINE) $(INCL
	$(TOOLCHAIN_PREFIX)objcopy -O binary firmware.elf firmware.bin
	$(eval MEM_SIZE=`./get_firmsize.sh`)
	$(PYTHON_DIR)/makehex.py firmware.bin $(MEM_SIZE) > progmem.hex
	$(eval MEM_SIZE=`$(PYTHON_DIR)/get_memsize.py MAINRAM_ADDR_W`)
	$(PYTHON_DIR)/makehex.py firmware.bin $(MEM_SIZE) > firmware.hex

system.h: ../../rtl/include/system.vh
	sed s/\`/\#/g ../../rtl/include/system.vh > system.h

clean:
	@rm -rf firmware.bin firmware.elf firmware.map *.hex *.dat
	@rm -rf uart_loader system.h
	@rm -rf ../uart_loader

.PHONY: all clean
```

Setup RTL simulation using the Icarus Verilog simulator

- Copy the simulation folder from the iob-soc repository
  `cp -r iob-soc/simulation .`
- Edit file "simulation/icarus/Makefile" as below, to compile the system and simulate it

```
#simulation baud rate
BAUD := 1000000
FREQ := 100000000

#paths
ROOT_DIR := ../..
SUBMODULES_DIR := $(ROOT_DIR)/iob-soc/submodules
FIRM_DIR := $(ROOT_DIR)/software/firmware

(...)

#***NEW*** add timer directory to the hw include paths
TIMER_DIR := $(ROOT_DIR)/iob-timer
HW_INCLUDE := -I. -I$(RTL_DIR)/include -I$(UART_DIR)/rtl/include -I$(CACHE_DIR)/rtl/header -

#***NEW*** add the timer IP verilog source to the list of sources
VSRC = $(RTL_DIR)/testbench/system_tb.v  $(RTL_DIR)/src/*.v $(RTL_DIR)/src/memory/behav/*.v $

(...)
```

Simulate 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

```
./a.out
VCD info: dumpfile system.vcd opened for output.
Hello world!
Total execution time: 1262 us @100MHz
```

- Congratulations! You have created your first RISC-V system using IOB-SoC!!

Main memory options

- The main memory type options are enabled by defining or undefining the USE_DDR and USE_BOOT macros
- Option 1: place the firmware image in FPGA memory during design compilation
  - Do not define either macro: not reprogrammable
  - This option is only valid for FPGA: needs recompilation if firmware changes
- Option 2: place the firmware in internal RAM
  - Define macro USE_BOOT only: reprogrammable
  - This option is valid for FPGA and ASIC
  - Firmware can be (re)loaded via UART
- Option 3: place the firmware in external DDR memory
  - Define both macros: reprogrammable
  - This option is valid for FPGA and ASIC
  - Firmware can be (re)loaded via UART
  - Third party DDR controller IP core is required

Conclusions and future work

- Conclusions
  - Presented project and editing guidelines
  - Designed of a peripheral IP (timer)
  - Instantiated peripheral in the system
  - Designed a simple software driver for the peripheral
  - Compiled the software
  - Simulated the system's RTL code running the software in the memories
  - Presented options for the main memory
- Future work
  - Non volatile (flash) external memory support
  - Real Time Operating System (RTOS)