Bridge of Life Education

SOC Design

Lab Plan

Jiin Lai

Lab List

Mandatory

1. Lab#1 – Vitis/Vivado Tools installation & Test (1w - individual)
2. Lab#2 – HLS FIR (1w - individual)
3. Lab#3 – Verilog FIR & XSIM/GTKWave (2w – individual)
4. Lab#4 – Caravel SOC Simulation (2w - team)
    1. Lab 4-2 – Caravel SOC User Project (FIR)
    2. Lab 4-1 – management FW
5. Lab#5 – Caravel FPGA - build from each IP (2w – team)

Optional choose 2 from the following list (3w team)

1. Lab#A – Interrupt Service
2. Lab#B – Executing in User Project memory
3. Lab#C – UART
4. Lab#D – SDRAM
5. Lab#D – Software Emulation (Bit banging)

Lab#6 - Workload Optimized SOC (WLOS) - Baseline (Till semester end – team)

Final Project - Workload Optimized SOC

Note: all the reference lab repository (except Lab#1, Lab#2) is in https://github.com/bol-edu/caravel-soc_fpga-lab

Lab Platform

- PYNQ-Z2, KV260 (MPSOC) – Embedded System
  - Vivado HLS – C-sim, Co-sim, IP-generation
  - Vivado Design Suite – IP integration, block-design, generate bit-stream
  - Download to PYNQ and run Jupyter Notebook

- Note: KV260 is twice of PYNQ-Z2 FPGA capacity

Build your Lab work on both PYNQ-Z2 and KV260

Tools – Xilinx Vitis 2022.1

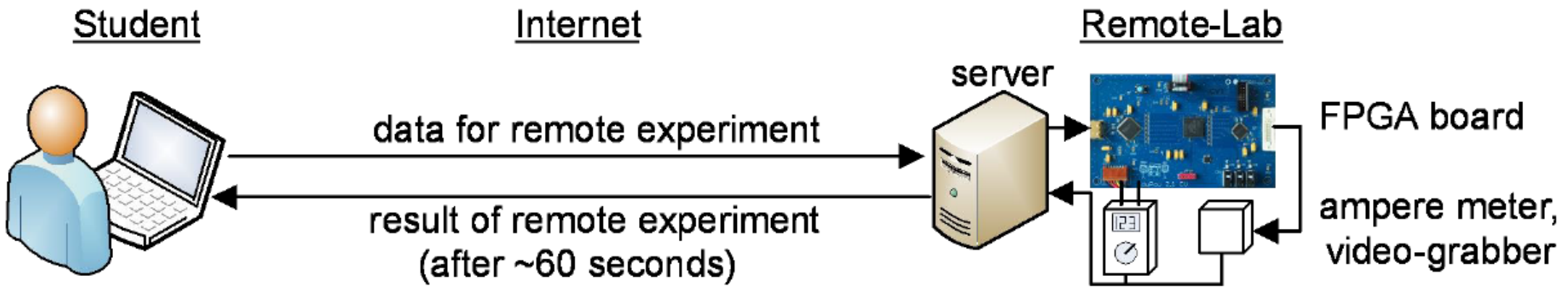
- Vitis/Vivado/Vitis HLS Software 2022.1
  - Vitis HLS - C-Simulation, Synthesis, C++/RTL Co-Simulation
  - Vivado – IP Integration, Block-Design, Bit-stream generation
  - Vitis – IDE/Makefile
    - Software-Emulation, Hardware-Emulation, Hardware
    - Vitis_Analyzer - Design analysis
- Tool Installation (Before Class)

Makefile or GUI

- To ease the replication of the work in Github, we use makefile. Thus tools run in Linux system is needed.
- During development stage, a GUI environment is preferred for design analysis and optimization. Window system can be used.

Online FPGA

- FPGA design software (Xilinx Vitis HLS) installed on student's computer
- Use Jupyter Notebook to access remote FPGA
- Locally compiled, simulated, and upload FPGA binary to remote FPGA
- Result of experiments shown in Jupyter Notebook
- Lab facilities
  - Cluster of Pynq-Z2/KV260 boards
  - Automate the FPGA board management (Demo)



Online FPGA Access

User Manual: (PYNQ-Z2/KV260)

- https://drive.google.com/file/d/1Vx0e3m9EviOuqPVhowYVbgVunZxD828i/view

Basic Lab – Tool Setup

Lab#1 – Vitis/Vivado Tools installation & Test

- Vitis/Vivado Tool installation
  - https://github.com/bol-edu/course-lab_1
  - Ref: HLS Tools Installation Guide 2022.1_Ubuntu.md
    - https://github.com/bol-edu/course-lab_1/blob/2022.1/HLS%20Tools%20Installation%20Guide%202022.1_ubuntu.md
  - Preinstall Vitis Tool, you can download from Google Drive, if you have problem in downloading from Google Drive, please copy from a flash drive (ask your TA)
    - https://drive.google.com/drive/folders/1aVujPu872Siyw-uODWrvf4p0axVsdPsw
  - 2022.1-Workbook-Lab1.pdf

- Setup
  - TeraTerm/PuTTY/WinSCP Installation on PC/Laptop
  - Vivado Installation Including Vivado HLS on PC/Laptop
  - PYNQ with Jupyter Notebook (Host Program)
- Implementation
  - IP Design (Multiplier) - Vivado HLS C/C++ to RTL
  - Generate Bitstream for FPGA - Vivado RTL to Bitstream
  - Host Program - Jupyter Notebook

Lab#2 – HLS FIR

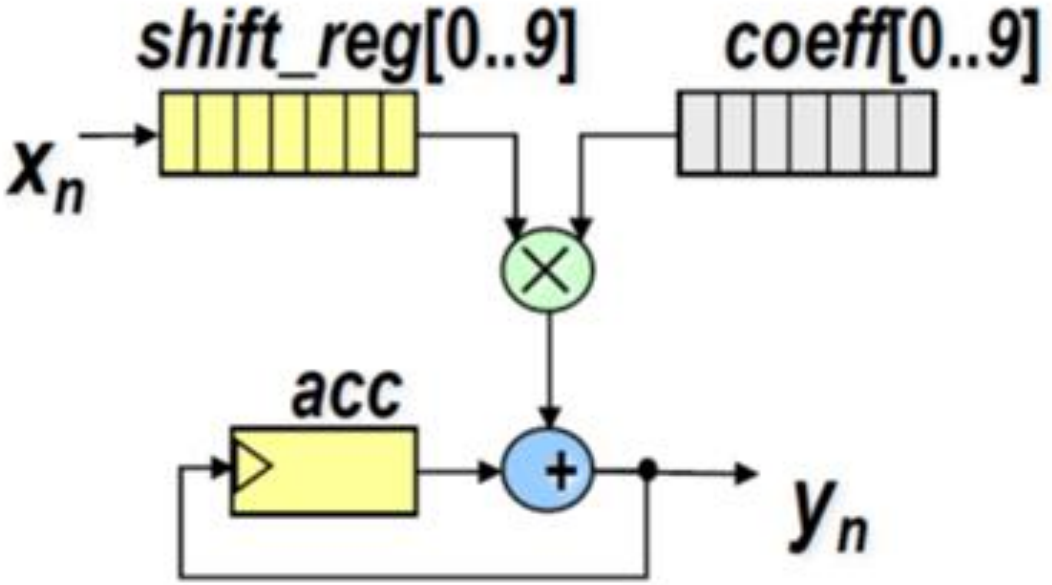
- Reference
  - https://github.com/bol-edu/course-lab_2
  - Using 2022.1-Workbook-Lab2-KV260.pdf - you can use KV260
- **Implementation**
  - FIR design
  - Interface with Host by AXI-Master, and Stream Interface
- **Anatomy of the embedded system software flow, including**
  - FPGA accelerator IP
  - OS kernel
  - PYNQ API layer
  - Application layer
- **Codes (host python, kernel) used a template for future project**

Lab#3 – Verilog FIR design with XSIM & GTKWave simulation (2-w)

A good exercise to sharpen your Verilog design capability

- Purpose:
  - Learn Xilinx Vivado Simulation (XSIM) and Waveform display (GTKWave)
  - Design FIR based on Lab2 HLS FIR specification (axi-stream interface)
- Reference：
  - Run xsim: https://github.com/bol-edu/soclab-nthusp23/tree/main/lab/02.xsim-gcd
  - Github fir: https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-fir
- Implementation
  - Run Xilinx Vivado Simulation (XSIM) of GCD
  - Design a FIR with Verilog implementation
  - Develop Testbench (test data from Lab#2 HLS FIR)
- Reference:
  - UG900: Vivado Logic Simulation
    - https://docs.xilinx.com/r/2022.1-English/ug900-vivado-logic-simulation/Overview
  - UG901: Vivado Synthesis
    - https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Vivado-Synthesis

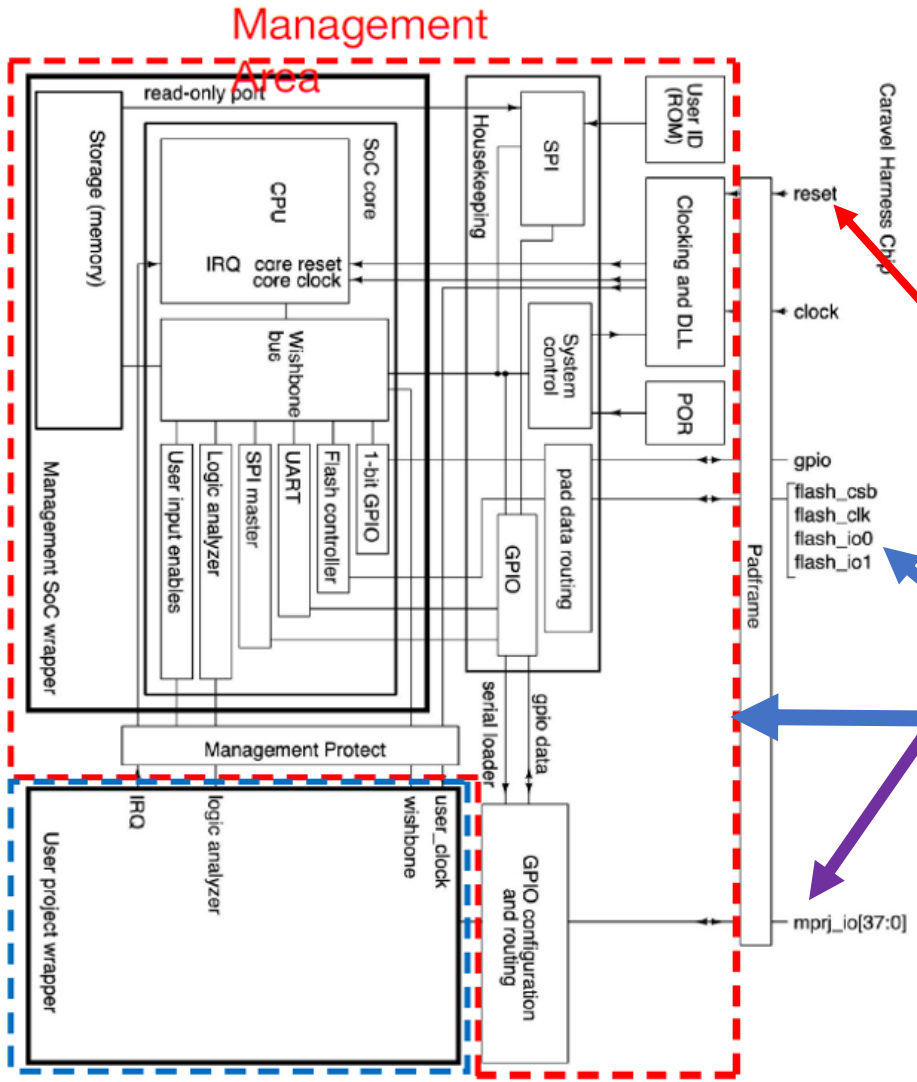
Specification FIR - 11 taps



- Data_Width 32
- Tape_Num 11
- Data_Num xx
- Interface
  - data_in stream (Xn)
  - data_out: stream (Yn)
  - coef[Tape_Num-1:0] axilite
  - len: axilite
  - ap_start: axilite
  - ap_done: axilite
- **Using one Multiplier and one Adder**
- Shift register implemented with SRAM (Shift_RAM, size = 10 DW) – size = 10 DW
- Tap coefficient implemented with SRAM (Tap_RAM = 11 DW) and initialized by axilite write
- Operation
  - ap_start to initiate FIR engine (ap_start valid for one clock cycle)
  - Stream-in Xn. The rate is depending on the FIR processing speed. Use axi-stream valid/ready for flow control
  - Stream out Yn, the output rate depends on FIR processing speed.

Caravel SOC – Simulation & FPGA

Caravel Simulation Verification System



Verilog testbench
counter_la_tb.v

```
Module counter_tb;
Initial beigin
 wait(checkbits == 16'hAB40);
 wait(checkbits = ..
End

Initial begin
  RSB <= 1'b0;
  RSB <= 1'b1;
end

Caravel uut(    );

Spiflash #( .FILENAME("counter_la.hex"} spiflash ( );

tbuart tburat (     );

endmodule
```

Lab#4-0 – Caravel SOC Simulation

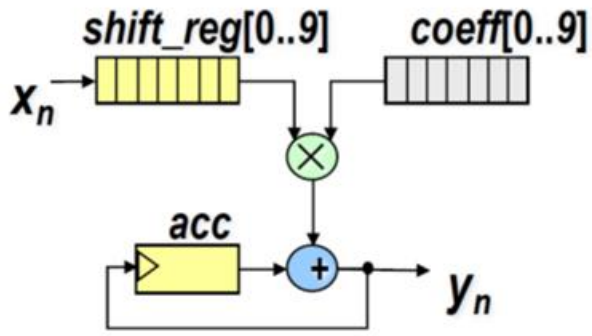
- Reference
  - https://github.com/bol-edu/caravel-soc (iverilog)
  - https://github.com/bol-edu/caravel-soc_fpga#run-xilinx-vivado-simulation-of-caravel-soc-fpga (Xsim)
    - 再git clone https://github.com/bol-edu/caravel-soc_fpga
    - 按照連結 https://github.com/bol-edu/caravel-soc_fpga#run-xilinx-vivado-simulation-of-caravel-soc-fpga
    - 就可執行xsim範例
- Run the following testbench
  - counter_la, coutner_wb, gcd_la
- Learn the following
  - Caravel testbench structure
  - Caravel SOC design & embedded programming
  - GDB + GDBWave Debugging (caravel-soc, iverilog) - (optional)
  - Trace Verilog code with Vim + vtags (caravel-soc, iverilog) – (optional)
  - Using CPU trace (https://github.com/bol-edu/caravel-soc/tree/cpu-trace) - caravel-soc, iverilog (optional)
- Observe the following - suggested for report content
  - spiflash access & code execution (observe CPU trace)
  - cpu wb cycles interaction with user project area.
  - cpu interface with user project with la
  - User project output mprj pin, and interaction with Testbench
- Submit report (follow the submission guideline)

Lab 4-1 – management FW lab (1-w) - Team

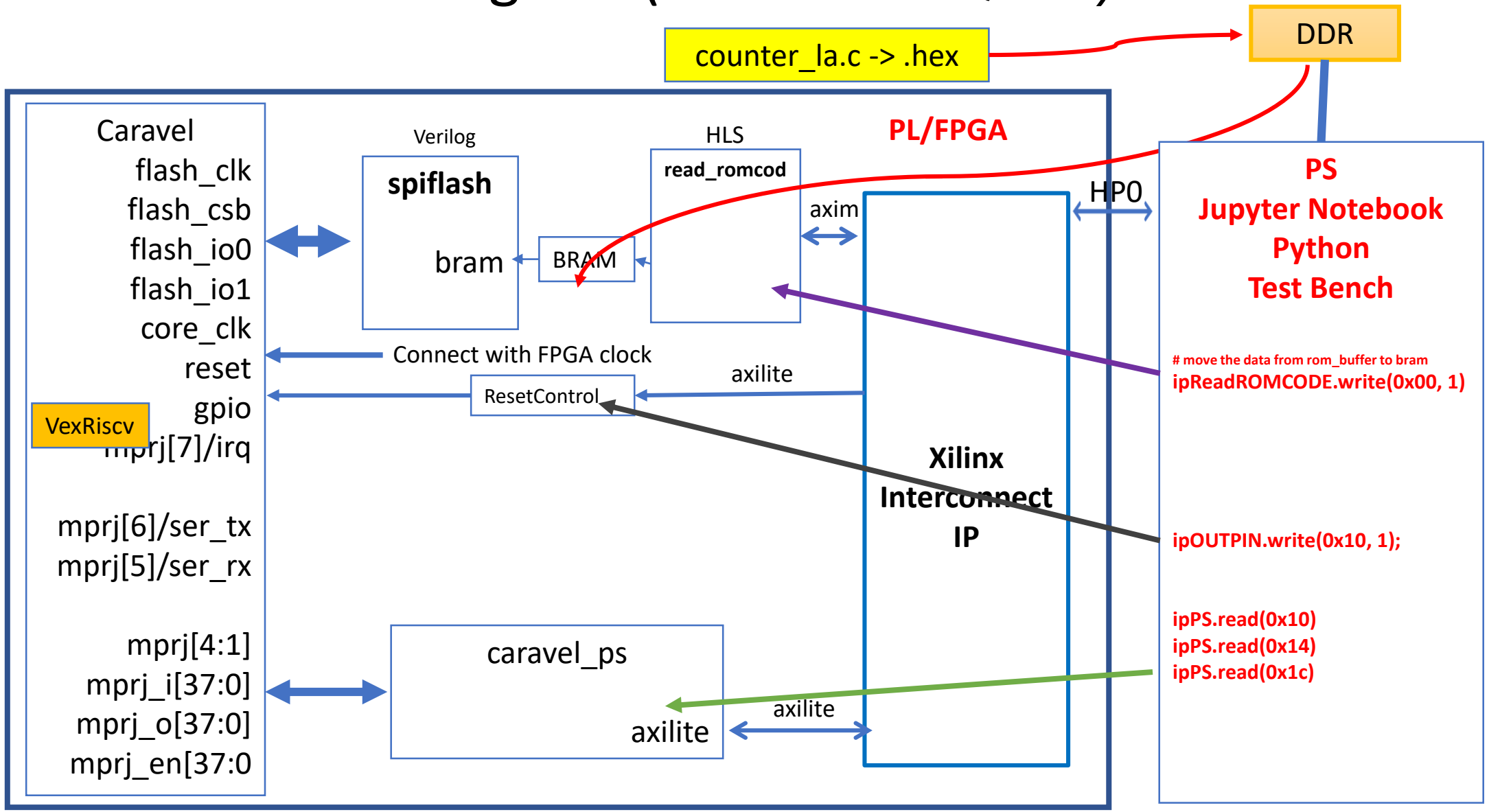
- Management SOC firmware examples (refer to class lecture) – based on mpw-8c branch
  - https://github.com/efabless/caravel/tree/b5010be8a7b89dd52e6da2c6f75f1ab05de43963/verilog/dv/caravel/mgmt_soc
- Each team sign up a topic below
  1. hkspi
  2. irq
  3. timer
  4. uart
  5. user_pass_thru
     - https://github.com/bol-edu/caravel-soc/files/11024947/Housekeeping.function.by.Willy.pdf
- Lab work and Report
  - Cross check firmware with RTL code & documentation
  - Observe the interaction of firmware code and hardware waveform
  - Change the code for different behavior and observe it in waveform, e.g. change timer value

Lab 4-2 – Caravel SOC User Project (FIR) – 2w Team

- Use the design from Lab#3 (Verilog FIR)
- Add Wishbone (WB) interface to communicate with RISCV
- RISCV-FIR Handshake Specification
  - RISCV Program coeff memory by WB
  - RISCV WB write x[n] input (note: you should make sure FIR is ready accept the data)
  - RISCV check if y[n] is ready, if yes: read y[n], otherwise wait (note: you need to have flag to indicate y[n] is ready)
  - When finish, write final Y (lower 16-bit) through mprj pin
  - Testbench check correctness, output last Y output (Y[15:0]) on mproj pins. use one mprj pin to indicate the last output result is ready (note, initially,
- Implementation, develop the following
  - Firmware code – move data (x, y), check correctness (Note: coefficient, x[n], y[n] defined inline, n=16)
  - Testbench – check mprj output from FIR
- Report
  - The programming interface between RISCV and user project
  - What is the throughput?
  - Suggest performance improvement based on the observation, e.g.
    - Speed up the FIR processing speed
    - Reduce the RISCV overhead in moving data, e.g. enlarge Shift_RAM size = 12 DW (one more than TapeRAM size) – what is the effect ?



CaravelFPGA Block Diagram (Based PYNQ-Z2)



Lab#5 – Caravel FPGA - build from each IP

- Reference
  - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/labi

Note： labi integrates the following three ip

  - Lab1 – ROMCode : https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab1
  - Lab2 – Spiflash : https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab2
  - Lab3 – GPIO pins: https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab3
  - The Caravel user project : gcd
- Implementation
  - Build the three IP: caravel_ps, read_romcode, output_pin
  - Build the FPGA bitstream – integrate the Caravel soc and the three IPs
  - Validate it on FPGA with the Jupyter Notebook run on PS
- Observe the following
  - AXI master to download the firmware code
  - Boot up with spiflash access
  - cpu wb cycles interaction with user project area
  - cpu interface with user project with la
  - User project output mprj pin
- Submission guideline – report

https://github.com/bol-edu/caravel-soc_fpga (IP is ready)

Lab#6 – Workload Optimized SOC (WLOS)

This is the baseline implementation for the Workload, i.e. mostly done in software. The execution environment development will be used for final project development.

1. Workload
    1. PS/Python code performs system initialization, firmware download, handshake with UART
    2. Firmware code - **matmul, qsort, fir, interrupt service routine**
    3. The firmware code and data runs from 10T BRAM (Lab#B)
    4. UART has two parts
        1. User-project UART (when receive a character, interrupt CPU
        2. Xilinx IP UART-lite to transmit / receive text between PS and user-project
    5. Firmware interrupt service – perform UART loopback function
        1. Receive text characters from UART/RX
        2. Send the text received to UART/TX
2. Implementation: TBD
3. Github reference:
    - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-wlos_baseline

Labs to prepare for Final Project (3w) team
Choose 2 lab from Lab# A,B,C,D

Lab#A – lab-interrupt : interrupt service

- Reference:
  - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-interrupt
- Implementation
  - Modify user_proj_example.gcd.v, generate interrupt after gcd is done
  - Interrupt service routine read the gcd result from la and output on mprj pins
  - Modify testbench to get the gcd result and display the value
- Observation & write report on
  - Identify the rtl code from interrupt generation and the signal path reaches to processor,
  - Identify related interrupt control and status registers.
  - How to enter/exit an interrupt service routine
  - Waveform to demonstrate the processing steps from interrupt generation, interrupt service routine, and return from interrupt

Lab#B – lab-exmem : Executing function from User Project memory

- Reference:
  - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-exmem
- Implementation
  - Implement the matrix-multiplication, and execute it on user memory
  - reference code: https://github.com/bol-edu/course-lab_riscv/blob/main/lab/synthesizable/benchmarks/src/matmul.c
- Observation & write report on
  - List relevant code changes from rtl/firmware
  - Memory map & linker (lds)
  - How to move code from spiflash to ser project area memory
  - How to execute code from user project memory
  - Show the Operation sequence and its waveform

Lab#C – lab-uart: UART

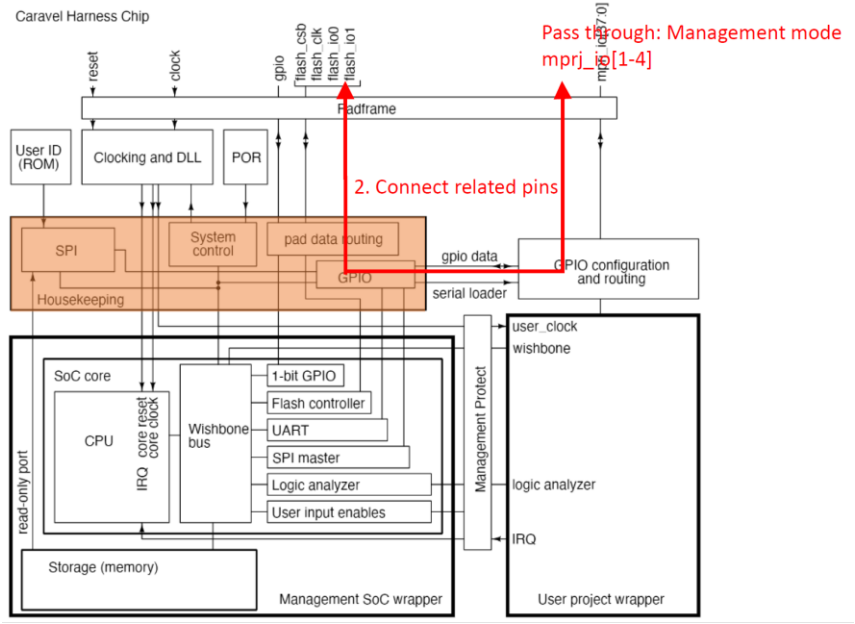
- Reference:
  - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-uart
- Implementation
  - Replace user project with the gcd design
    - https://github.com/bol-edu/caravel-soc_fpga-lab/blob/main/lab-exmem/rtl/user/user_proj_example.gcd.v
  - The gcd result is output on la signals
  - FW read the gcd result from la signals, and print the value out on UART
  - PS/Jupyter notebook receives and prints the message (the gcd result)
- Observation & write report on
  - rtl and firmware changes
  - UART operation waveform
  - Caravel RISCV and UART interface signals and mmio
  - UART Interrupt service on PS Python

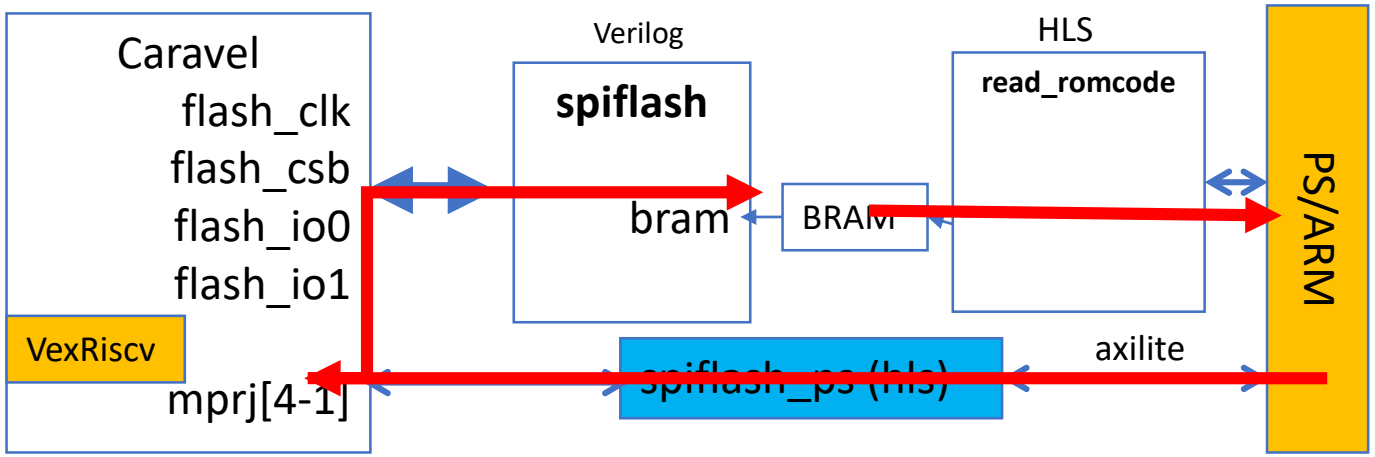
Lab#D – lab-sdram : SDRAM controller & SDRAM device

1. Refer to lab-exmem, but replace the BRAM with SDRAM (SDRAM controller + SDRAM)
    1. SDRAM device - convert a SDRAM model to hardware implement, memory array using BRAM
    2. SDRAM controller - support non-page mode
    3. The combined SDRAM Controller + SDRAM device is to replace a Wishbone BRAM
    4. The reference code for SDRAM controller and SDRAM device model is available
2. Reference:
    - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-sdram
3. Implementation
    - Firmware matmul executed in SDRAM
    - Modify SDRAM controller to support non-page mode or auto-precharge.
    - Shorten the refresh period, 100 times shorter
4. Observe and write report on
    - Study the SDRAM controller design – how page mode is implemented, sdram bus protocol …
    - Performance difference on page-mode, non-page-mode
    - Observe dram access conflicts with DRAM refresh
    - When refresh period is 100 times shorter, what is the performance difference ?

Lab#E : Software Emulation – Bit Banging

- Bit Banging is a method to employs software as a substitute for dedicated hardware to generate transmitted signals or process received signals. It is a low cost implementation, and often used in embedded system.
- Caravel SOC provides a Pass-through mode to allow an external FTDI (USB-SPI) device (connected to mprj[1-4]) to update firmware code.





https://github.com/bol-edu/caravel-soc/files/11024963/caravel_SPI.pdf
https://github.com/bol-edu/caravel-soc/files/11024947/Housekeeping.function.by.Willy.pdf
https://en.wikipedia.org/wiki/Bit_banging

Lab#E : Software Emulation – Bit Bang

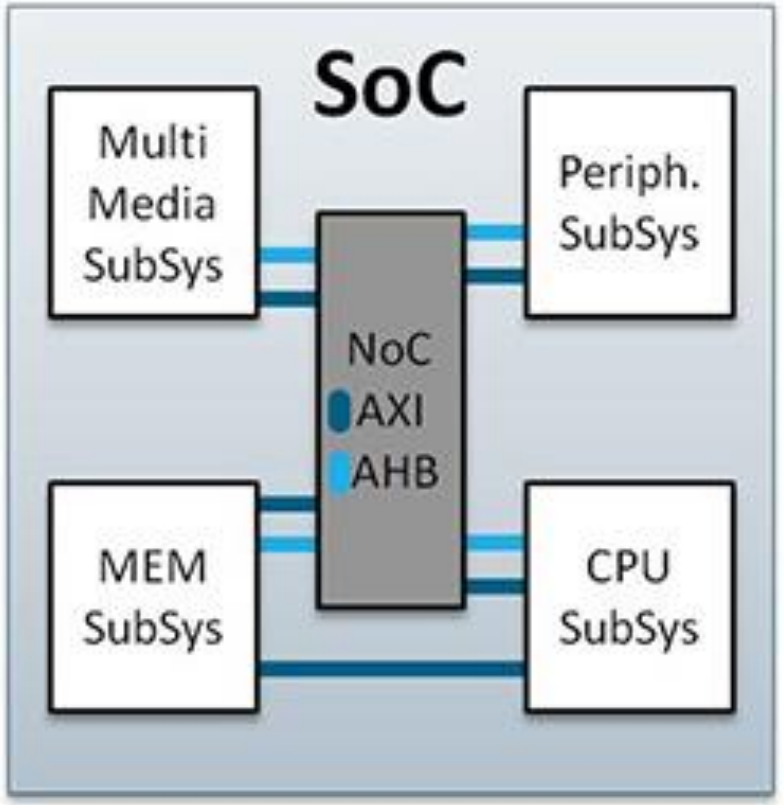
- Reference: Lab#4-1, Lab#5
- Implementation
  - Add write function to spiflash.v
  - Integrate the new spiflash.v to CaravelFPGA
  - Develop Python code program caravel-ps (mprj registers) to generate spi write transaction to upload firmware code to spiflash
  - Upload firmware code to spiflash (Note: Caravel is held in reset state)
  - After reset release, RISC-V will execute the new firmware code.
- Submission & Report
  - Github with spiflash.v, Python code
  - Report on the development process – issue found, fix, code documentation

https://github.com/bol-edu/caravel-soc/files/11024963/caravel_SPI.pdf
https://github.com/bol-edu/caravel-soc/files/11024947/Housekeeping.function.by.Willy.pdf
https://en.wikipedia.org/wiki/Bit_banging

Final Project – Workload Optimized SOC Suggested Area to Improve

Objectives

1. Understand the role of each components in the SOC, SOC contains
   - Computing - processor, coprocessor(accelerator)
   - Memory – Cache, System Buffers, DDR
   - IO - UART, SPI …
   - Interconnect / BUS - data-mover (DMA)
2. Optimize the component



Computing - RISCV

- Baseline – Caravel VexRISCV
- Design your own RISCV, and optimize the following area
- Enhanced RISCV (in user area) – after spiflash code load into BRAM
  - Branch prediction
  - Add instruction Multiplication (R32VIM)
  - Add Cache
  - Out-of-Order

Note: The RISCV is implemented in user project area.

After FW code loaded into user project memory, the execution is handed over to user-project RISCV

You will design a communication mechanism between Caravel RISCV and User-RISCV

Computing - FIR

- Baseline – RISCV execute FIR code (Note: data is in SDRAM)
- Based on Lab#3 (Verilog FIR), improve with
  - Increase Shifter RAM size to improve data throughput
  - Add an DMA engine like Xilinx AXI DMA to stream-in, stream-out data to/from FIR-ACC

Memory - SDRAM

The baseline code runs on BRAM with 10 clocks read/write access time. You can improve the memory system with the following options,

1. You can replace the memory with page-mode SDRAM (Lab#D)
2. If you implement FIR with DMA, you will need to support two requests (one from CPU, another from FIR DMA), an bus arbiter is needed
3. Since SDRAM can do burst transfer, you may consider add write-buffer, and/or prefetch buffer to further reduce the memory access latency.
4. Since there may be two requests from different address range, it may create page conflict when switch between two requests, you may consider to customize your address map to reduce the conflict.
5. Any other idea ….

IO – UART

Currently UART workload implementation has a lot of CPU overhead, i.e. frequent interrupt, you may consider the following optimization

1. Add FIFO to current UART implementation, doing so, can greatly reduce CPU overhead
2. Even further perform HW loopback with CPU involved.

Other Enhancement

- Matmul accelerator
- Qsort accelerator
- Any other idea ….