Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

INTRODUCTION TO SoC

Practical session - Semester 251

WEEK 2

Carry Look-ahead Adder (CLA) & RISC-V Assembly

1 Introduction

1.1 Aims

- Get familiar with Vivado software.
- Get familiar with the FPGA Arty-Z7 board.
- Get familiar with RISC-V toolchain
- Understand the hierarchical design principle.
- Practice in writing test benches for a designed module.

1.2 Preparation

- Read the laboratory materials before class.
- Revise Verilog basics.
- Each group prepare at least one laptop with Vivado software installed.
- Download prebuilt RISC-V toolchain.
- Recommend using Linux for the Lab session.

1.3 Documents and lab materials

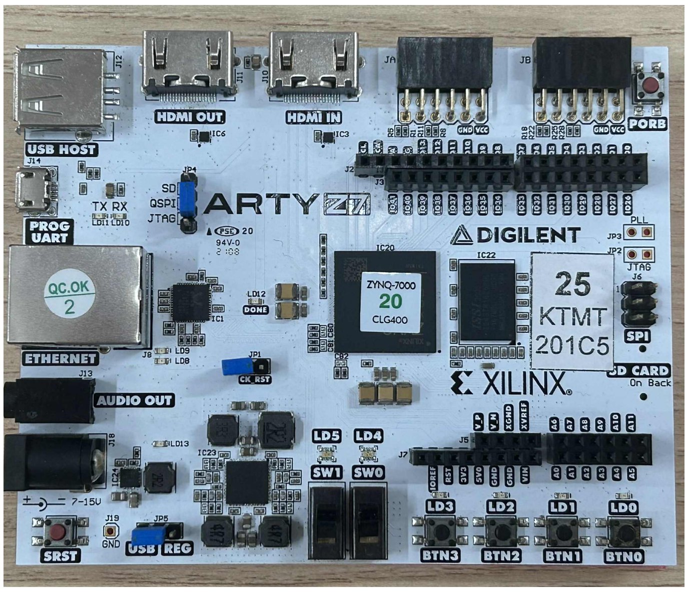

Figure 1: Arty-Z7 FPGA board

- M. Morris Mano, Michael D. Ciletti, Digital System with an Introduction to the Verilog HDL, VHDL, and SystemVerilog, Pearson Education, Inc., 2017
- Lecture slides
- *Arty-Z7-20-Master.xdc*: Arty-Z7 constraint file.
- *Guide_for_Installing_Vivado.pdf*: Guide for installing Vivado and getting started with Vivado and Arty-Z7.

1.4 Procedure

For each exercise (also for further labs):

- Read the requirements, then determine the input/output signals of your circuits.
- Make a design idea of the circuit, then use Verilog to model the circuit.
- Analysis & Synthesis the circuit with Vivado software.
- Write a test bench to simulate the circuit on Vivado Simulator.
- Generate the bitstream and program the Arty-Z7 to evaluate the circuit.

1.5 Report requirements

- Lab exercises will be reviewed directly in class.
- Write a PDF report (with circuit/simulation screenshots inserted).
- Must explain the flow of code, capture the waveform, and the simulation results on board Arty-Z7
- Submit on BK-elearning by deadline

2 Carry Look-ahead Adder (CLA)

- You'll implement a two-level Carry-Look Ahead (CLA) adder, as discussed in class.
- This adder will then be used in your processor implementation in a later homework.

2.1 Literature review

To go faster than a carry-select adder, we need to improve how carries are computed. Since carry computation is the main bottleneck, one way to speed it up is by using redundancy to determine carries more quickly. Although this method requires more hardware, it significantly reduces delay. This faster approach is known as the **carry look ahead adder (CLA)**.

2.1.1 Generate & Propagate

Let's look at the single-bit carry-out function:

$$CO = (AB) \mid (AC) \mid (BC)$$

We can factor out terms that use only A and B (which are available immediately):

$$CO = (AB) \mid (A \mid B) \cdot C$$

Define:

$$G = AB$$

$$P = A \mid B$$

Here, G is the **generate** term (it produces a carry-out regardless of the incoming carry C), and P is the **propagate** term (it allows the carry to propagate).

Thus, the carry-out can be expressed as:

$$CO = G \mid PC$$

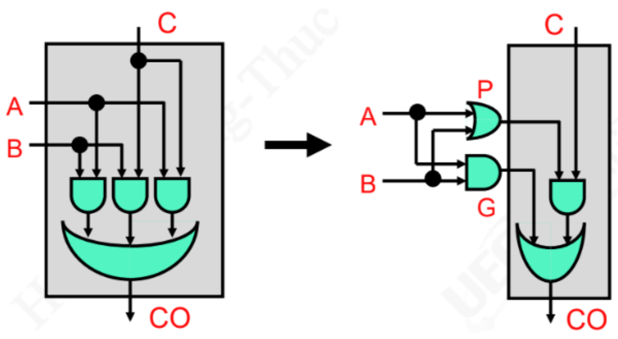

Figure 2: Single bit carry-out

2.1.2 Multi-level CLA

To calculate the **propagate and generate** signals based on inputs A and B (and not on the carry-in), we can use a tree structure. In this approach, pairs of bits are combined hierarchically to determine overall propagate and generate signals for larger groups of bits. This tree-based design allows carries to be computed in parallel, reducing the overall delay to a logarithmic scale relative to the number of bits. Although this method requires more hardware, it provides a good balance between speed and area efficiency.

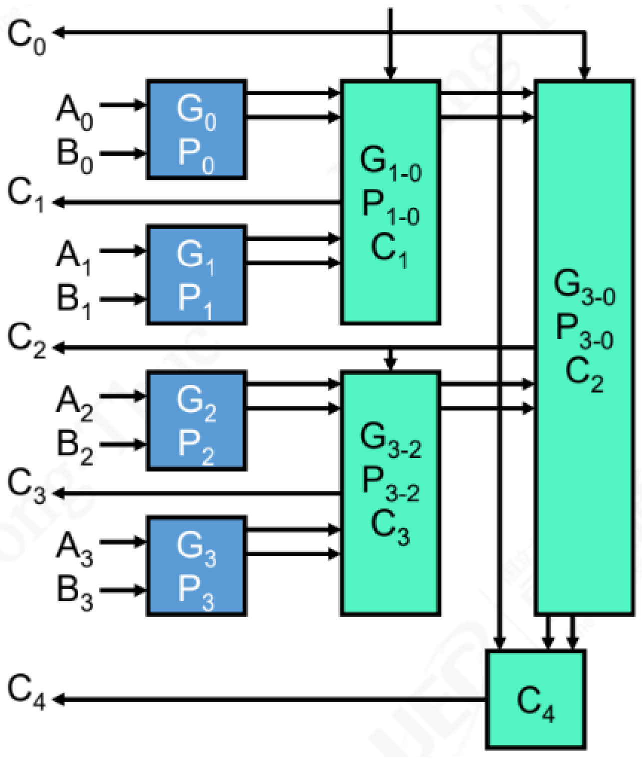

Figure 3: A 4b CLA

Windowed G/P: abstraction for multi-level CLA

Individual carry equations:

$$C_1 = G_0 \mid P_0C_0$$

$$C_2 = G_1 \mid P_1C_1$$

Infinite hardware CLA equations:

$$C_1 = G_0 \mid P_0C_0$$

$$C_2 = G_1 \mid P_1G_0 \mid P_1P_0C_0$$

Group terms into "windows":

$$C_2 = (G_1 \mid P_1G_0) \mid (P_1P_0)C_0$$

$$C_2 = G_{1-0} \mid P_{1-0}C_0$$

where G_{1-0} and P_{1-0} are **window G & P**:

- 1 bit summarizing gen/prop for all bits in the window
- G$_{1-0}$: carry-out generated by bits [1:0]
- P$_{1-0}$: carry-out propagated by bits [1:0]

2.2 CLA sub module

We supply some skeleton code in **cla.v**, including module port definitions and the simple *gp1* generate-propagate module at the leaf of the CLA hierarchy.

You will need to implement three additional modules to have a complete 32-bit adder.

You should start with the 'gp4' module, which computes aggregate generate/propagate (g/p) signals for a 4-bit slice of the addition.

The module takes the bit-level g/p signals from 'gp1' as input, and computes whether all 4 bits collectively generate or propagate a carry.

The module also computes the actual carry-out values for the low-order 3 bits of its input.

You should also think about why 'gp4' doesn't compute 4 bits worth of carry-out.

The gp4 module will form the top layer of your CLA hierarchy.

Verilog's reduction operators will come in handy in building your 'gp4' module.

They perform a bit-wise reduction like so:

```
wire [3:0] w;
wire or_reduction;
assign or_reduction = (| w);
assign or_reduction = (w[3] | w[2] | w[1] | w[0]); // equivalent to code above
```

Listing 1: Bitwise Reudction algorithms

Reductions can be combined with indexing to reduce just a portion of a bus:

```
wire [3:0] w;
wire or_reduction;
assign or_reduction = (| w[2:0]);
assign or_reduction = (w[2] | w[1] | w[0]); // equivalent to code above
```

Listing 2: Reduction Algorithm

You can run ‘gp4‘ tests via

```
MAKEFLAGS=-j4 pytest --exitfirst --capture=no testbench.py -k runCocotbTestsGp4
```

though their coverage is low so we encourage you to add other test cases as well.

Once you have the ‘gp4‘ module working, you can move on to the ‘gp8‘ module which will form the base of your CLA hierarchy.

The ‘gp8‘ logic is a generalization of ‘gp4‘ to a larger window size. Though it is harder, you may consider implementing a parameterized ‘gpn‘ module that computes generate/propagate/carry-out over an N-bit window.

You can then instantiate this appropriately for both ‘gp4‘ and ‘gp8‘.

2.3 CLA Module

Finally, you will build the 32-bit adder module ‘cla‘. Use the ‘gp1‘, ‘gp4‘ and ‘gp8‘ modules to build your CLA tree and to compute the final sum.

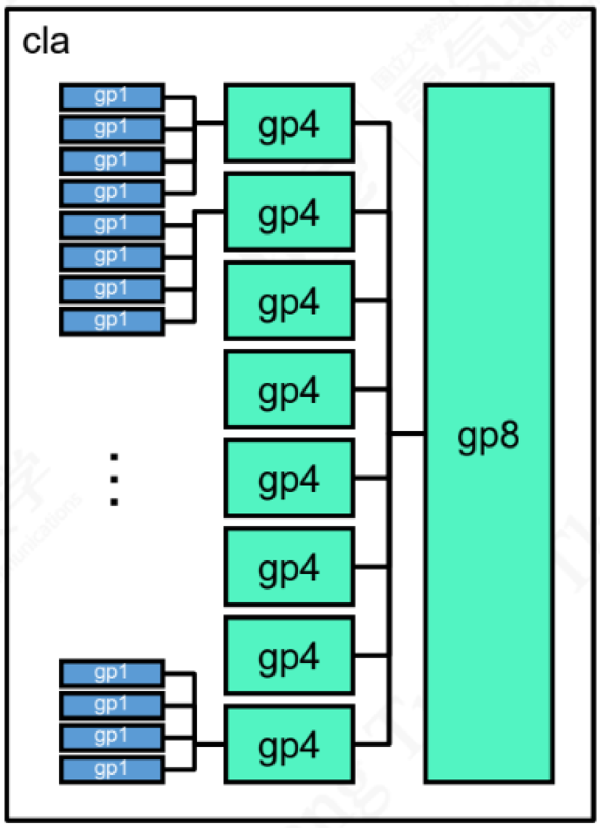

Figure 4: 32b CLA using sub-gp module

You can test your ‘cla‘ module via:

```
MAKEFLAGS=-j4 pytest --exitfirst --capture=no testbench.py -k runCocotbTestsCla
```

2.4 Submitting your code

Submit your **cla.v** file

2.5 FPGA demo

Now that your code works in simulation, you can run the FPGA board demo to see it operate in real life!

The demo code is contained in the **system.v** module.

It uses your adder to add 26 to a 4-digit binary number represented by four of the board's buttons (B0, B1, B2, B3).

When a button is not pressed, it represents a logic 0; pressing it changes that bit to a logic 1 instead.

The resulting sum is displayed on the board's LEDs. (`LD0`, `LD1`, `LD2`, `LD3`, `LD4`, `LD5`)

3 RISC-V Assembly

In this assignment, you will write a simple program in **RISC-V (RV) assembly** to familiarize yourself with the instruction set architecture (ISA).

Starting with the code provided in `uppercase.S`, your program must:

1. Take as input a NUL-terminated ASCII string located in memory.
2. Convert the string into uppercase **in-place**.
3. When the conversion is complete, use a `j end_program` instruction to jump into an infinite loop.

You may find the following resource helpful:

- Table of ASCII character codes

3.1 Using Spike simulation

3.1.1 Buiding your code

You will use the **RISC-V toolchain** (`riscv64-unknown-elf-...`) to build and test your assembly code.

You have two options for obtaining the RISC-V toolchain:

1. **Compile your own toolchain** from source.
2. **Download a prebuilt toolchain** from the following repository:

   `https://github.com/stnolting/riscv-gcc-prebuilt`

The '`make build`' command will assemble your '`uppercase.S`' file into a RV executable, which can then be run.

3.1.2 Run your code

Once you have an RV executable, you can run it using **Spike** [?]:

https://github.com/riscv-software-src/riscv-isa-sim

Spike is a program that simulates a RISC-V processor and allows you to control execution, inspect memory, and view registers.

Running the Program

Use the following command to launch your program with Spike: `make start`

Useful Spike commands

- `'run 1'` — run one instruction
- `'pc 0'` — print out the PC for core 0 (there is only one core in our setup, but Spike requires you to specify the core number)
- `'reg 0'` — print out all register values for core 0
- `'reg 0 a0'` — print out the value of the `a0` register for core 0
- `'str 0 2000'` — print out the contents of memory at address `0x2000`, treating it as a NUL-terminated ASCII string
- `'help'` — print out the list of available commands

At the very beginning of execution, Spike runs a few instructions before reaching your code at the `_start` label.

The ASCII string to convert is located in memory starting at address `0x2000`.

When learning RISC-V assembly, you may find the **RISC-V ISA Reference Sheet** helpful.

3.1.3 Test your code

The `make run` command will run your program under Spike, displaying the input string both before and after execution.

This is the command that will be used to test your code.

You should also experiment with other input strings beyond the one provided to you, to **ensure that your program works correctly for all printable ASCII characters** (decimal values `32--126`).

3.2 Using web-based online RISC-V emulators

There are many online RISC-V emulators available. Two recommended options are:

- https://creatorsim.github.io/creator/
- https://thaumicmekanism.github.io/venus/

Note that these emulators interpret certain assembler directives slightly differently from Spike.

3.2.1 Using CreatorSim

For the emulator at https://creatorsim.github.io/creator/ (select `RV32IMFD`):

- Remove all occurrences of the `.section` directive.
- Rename the `_start` label to `main`.

3.2.2 Using Venus

For the emulator at https://thaumicmekanism.github.io/venus/:

- Remove all occurrences of the `.section` directive.
- Replace the `.asciz` directive with `.string`.

3.3 Submitting your code

Submit your **uppercase.S** file

- In addition to the source codes, students must also prepare a PDF file to explain the codes, the reasoning behind them, and show the results.
- The results include screenshots of the waveforms and photos of the board output (if the assignment involves an FPGA board).
- Email tile format: **[Introduction to SoC] [student ID] Practice #no.**
- Send source codes and PDF report to:
  hoangtt@uec.ac.jp, cc: anhpkn@hcmut.edu.vn, tnthinh@hcmut.edu.vn