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

INTRODUCTION TO SoC

Practical session - Semester 251

WEEK 3

Single Cycle Datapath

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

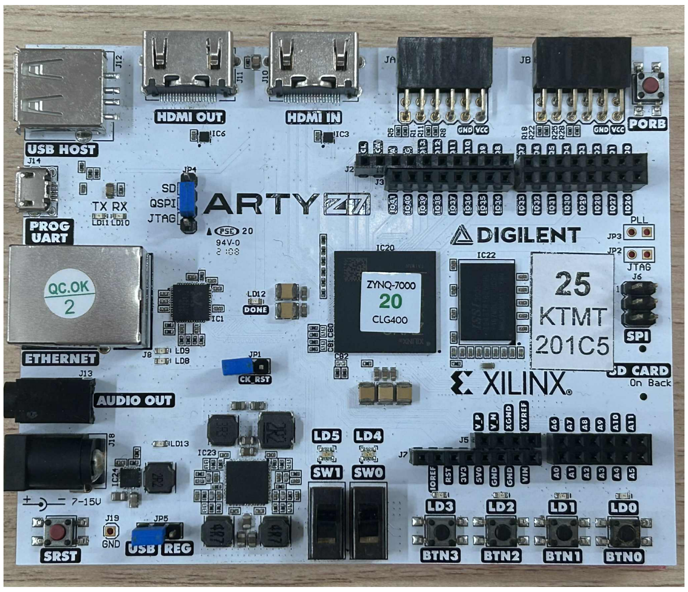

Figure 1: Arty-Z7 FPGA board

- 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 Single-Cycle Datapath

- `DatapathSingleCycle.v` has your starter code, including the memory, the program counter, and signals useful for decoding rv32im instructions.
- This homework builds upon previous assignments, and will include **your divider and CLA**.

2.1 Literature review

A computer can be viewed as a **finite state machine** consisting of:

- **Registers** — a few of them, but very fast.
- **Memory** — lots of it, but slower.
- **Program Counter (PC)** — holds the address of the next instruction to execute.

A computer executes instructions through the following steps:

1. **Fetch**: Retrieve the next instruction from memory.
2. **Decode**: Determine what the instruction does.
3. **Read Inputs**: Access operands from registers and/or memory.
4. **Execute**: Perform the operation (e.g., addition, multiplication, etc.).
5. **Write Outputs**: Store the result back to registers or memory.
6. **Next Instruction**: Update the Program Counter (PC) to point to the next instruction.

A **program** is simply data stored in memory.

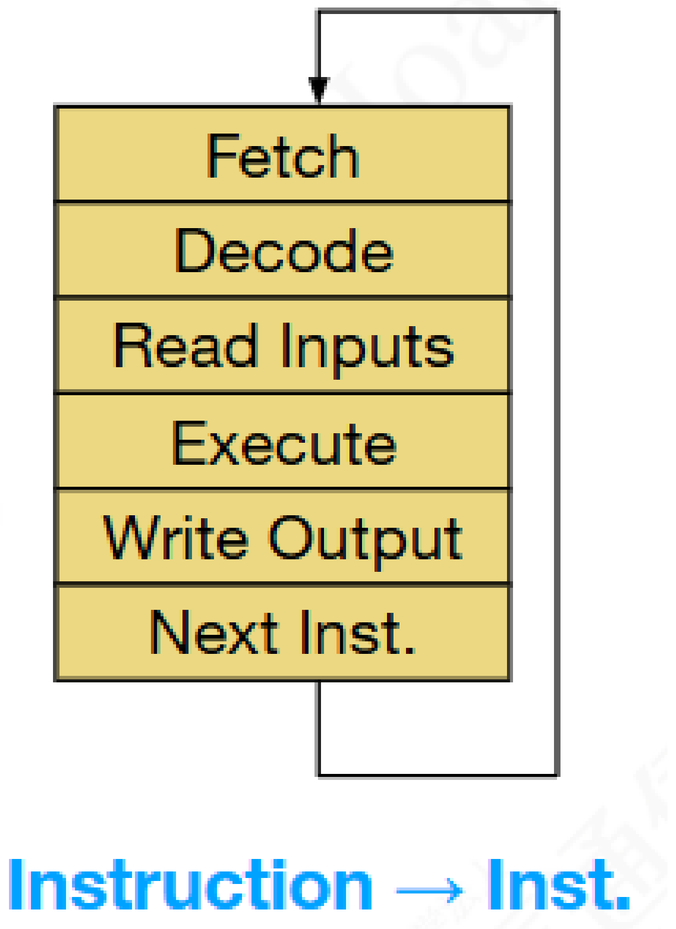

Figure 2: Execution model

2.1.1 Datapath

The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers.

The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

During instruction execution, the **fetch** phase retrieves the instruction from memory and translates the operation code (opcode) into a set of control signals that direct the rest of the system.

The **datapath** is responsible for performing the actual computations. It includes components such as registers, arithmetic logic units (ALUs), and other hardware elements that manipulate data according to the instruction's requirements.

The **control unit** determines which computation should be performed and how data should flow through the datapath. It routes data between registers and selects the appropriate ALU operation based on the decoded instruction.

Overall, the processor operates in a continuous **Fetch** → **Decode** → **Execute** cycle, where each instruction passes through these three fundamental stages.

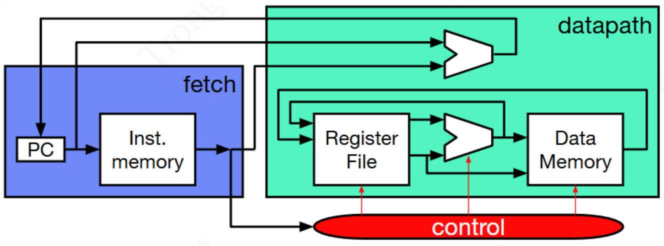

Figure 3: Datapath

2.1.2 Single cycle Datapath

The single-cycle microarchitecture executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction. Moreover, the processor requires separate instruction and data memories, which is generally unrealistic.

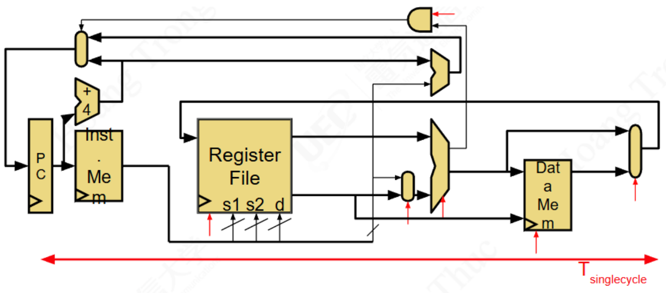

Figure 4: A Single Cycle Datapath

2.2 Performance analysis

This section introduces the basis for analyzing computer performance. There are many ways to measure performance, and marketing materials often emphasize metrics that make a system look faster—such as clock frequency or number of cores—without reflecting real-world results. In reality, different processors accomplish varying amounts of work per clock cycle, depending on the program.

The most reliable way to measure performance is by timing how long your program takes to execute—the computer that finishes it fastest has the best performance. If your program isn't available, use a set of representative programs, known as benchmarks, whose total execution time indicates how a processor performs.

The equation for execution time is decribed as follow:

$$\text{Execution Time} = \frac{\text{seconds}}{\text{program}} = \left(\frac{\text{instructions}}{\text{program}}\right) \times \left(\frac{\text{cycles}}{\text{instruction}}\right) \times \left(\frac{\text{seconds}}{\text{cycle}}\right)$$

The term **instructions per program**, also known as the *dynamic instruction count*, refers to the total number of instructions executed by a program at runtime. This quantity is determined by the program itself, the compiler, and the instruction set architecture (ISA).

The term **cycles per instruction** (CPI) represents the average number of clock cycles required to execute each instruction, typically ranging from about 2 to 0.5. CPI depends on the program, compiler, ISA, and the underlying microarchitecture.

The term **seconds per cycle**, also called the *clock period*, is the duration of a single clock cycle, typically between 2 ns and 0.25 ns. Its reciprocal is the **clock frequency**, which ranges from approximately 0.5 GHz to 4 GHz, and is determined by the microarchitecture and technology parameters.

To achieve the minimum possible execution time, each of these terms should be minimized. However, in practice, they often conflict with one another, making performance optimization a complex trade-off.

The number of seconds per cycle is the clock period, Tc. The clock period is determined by the critical path through the logic in the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder.

2.3 Instructions Design

2.3.1 ALU & Branch Instructions

You should start by completing your register file in the `RegFile` module.

Once your register file is working, you can start implementing your processor.

The instance of the RegFile module must be named ‘rf‘ for the tests to work.

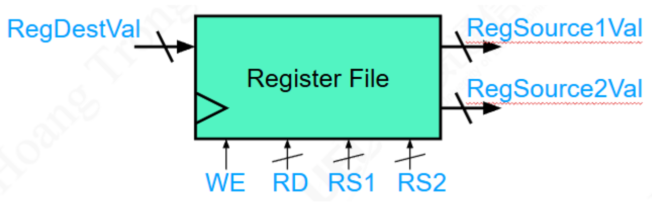

Figure 5: Register File

Your processor must support ALU instructions (*‘lui‘* through *‘and‘* but not *‘auipc‘*) on our **[ISA sheet]**, *branches* (‘beq‘ through ‘bgeu‘), and the *‘ecall‘* instruction.

‘ecall‘ just needs to set the *‘halt‘* output of the processor to 1.

You should start with *‘lui‘*, which loads an immediate into the upper bits of a register.

The first step is to fetch the *‘lui‘* instruction from memory.

The memory unit lives outside your datapath, as you can see in the *‘Processor‘* module.

The memory supports simultaneous access via a read-only port for fetching instructions, and a read-write port for loads and stores.

Your datapath sends an address (on the *‘pc_to_imem‘* output) to the memory, and the memory responds with the value at the requested addresses (on the *‘inst_from_imem‘* input).

You can ignore the other memory inputs/outputs until the next practice when you'll implement loads and stores.

You will need to use your CLA adder to implement the *‘addi‘*, *‘add‘*, and *‘sub‘* instructions.

In other situations in which you need to add things (e.g., incrementing the PC or computing branch targets), you can use the ‘+‘ operator.

2.3.2 Remaining Instructions

You will need to support the remaining rv32im instructions.

The memory instructions, with multi-byte loads and stores, will likely be where you spend the most time.

Note that your processor does not need to support misaligned memory accesses.

You should instantiate your divider and use it to implement the divide and remainder instructions.

You can use the ‘*‘ operator for multiply.

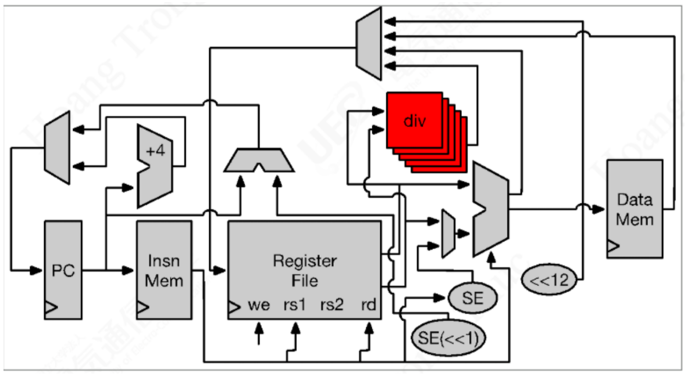

Figure 6: Critical path through the divider

2.3.3 Check timing closure

For this homework, in addition to the usual testing in simulation,you will also run the FPGA tools to generate a bitstream.

Doing so allows you to check how fast your design runs (its clock frequency) and how many FPGA resources it consumes.

We will not require you to reach a particular clock frequency or resource consumption level.

To see information about critical paths and the frequency of your design, the frequency part of the report will look something like this:

```
"fmax": {
    "$glbnet$clk_proc": {
        "achieved": 5.247085094451904,
        "constraint": 4.166666507720947
    },
    "clk_mem": {
        "achieved": 22.045854568481445,
        "constraint": 4.0100250244140625
    }
},
```

Listing 1: Design example

The key metric is whether you have achieved *timing closure*, i.e., whether the *achieved* clock frequency of your design meets its *constraint* (the frequency you requested).

If the achieved value is higher or equal the constraint, you have achieved timing closure. Otherwise, timing closure has failed and you need to run with a slower clock.

If your design does not meet timing, go ahead and ****submit it anyway****.

However, clock frequency and LUT usage should be gathered in the report, too.

2.4 Submitting your code

You cannot use the '-', '/' or '%' operators in your code.

Submit your **DatapathSingleCycle.v** file together with clock frequency and LUT usage reports.

- 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