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

INTRODUCTION TO SoC
Practical session - Semester 251

ASSIGNMENT

Pipelined 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

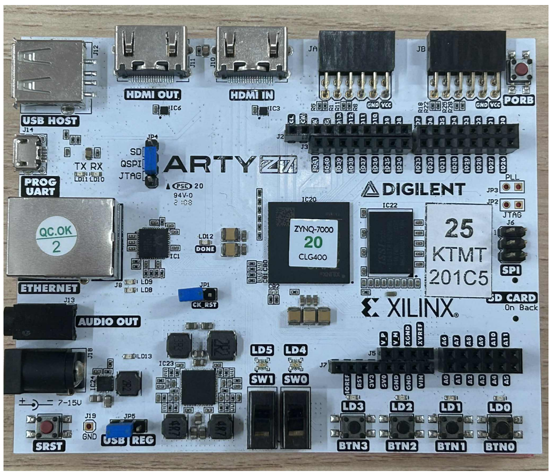
Figure 1: Arty-Z7 FPGA board

- 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.

2 Pipelined Datapath

- In this homework you will build on your design from previous homework and pipeline the entire datapath into 5 stages.
- There are two milestones for this assignment.

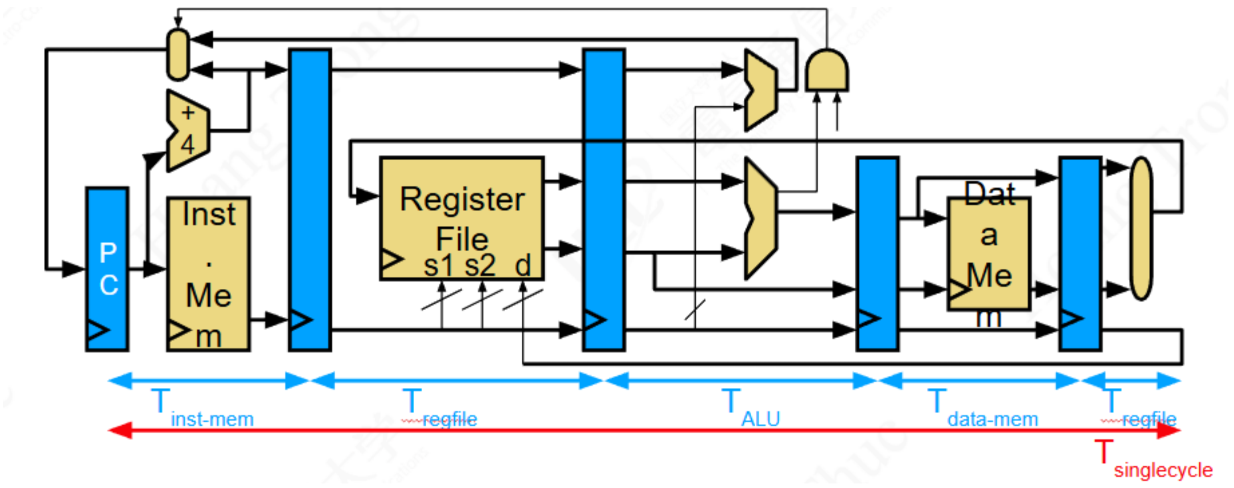

Figure 2: 5-stage Pipelined Datapath

2.1 Milestone 1: RV32I ALU Instructions and Branches

For this first milestone, you will need to handle the RV32I ALU instructions (except 'auipc') and also branch instructions.

You will need to implement MX and WX bypasses, and also a **WD bypass** so that if an instruction i_d in Decode reads a register x and another instruction i_w in Writeback writes to x in that same cycle, i_d will receive its value of x from i_w.

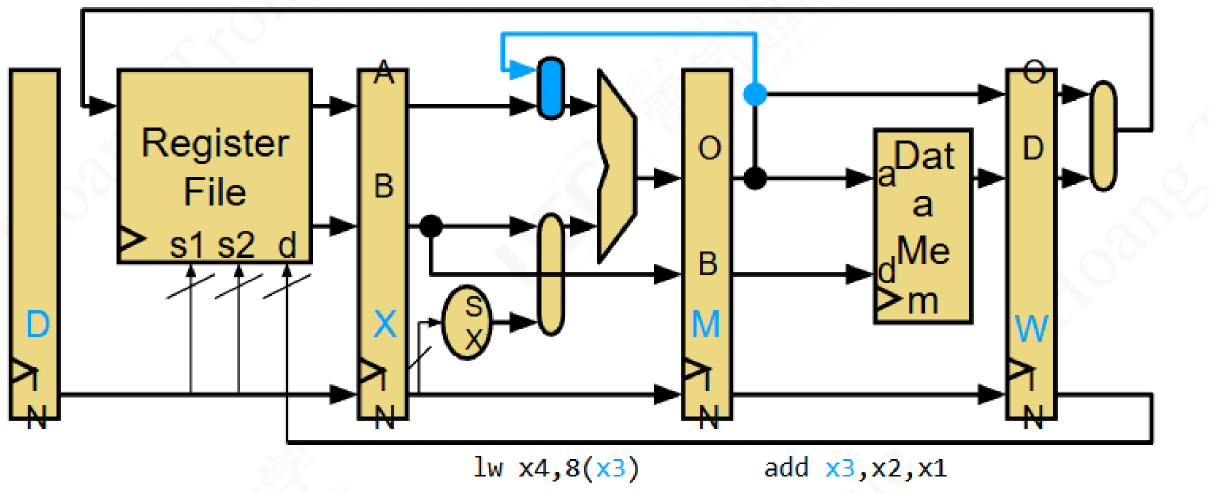

Figure 3: Your MX in Pipeline datapath

You can implement this WD bypass inside your register file, or outside it in the Decode stage of the datapath; either is fine.

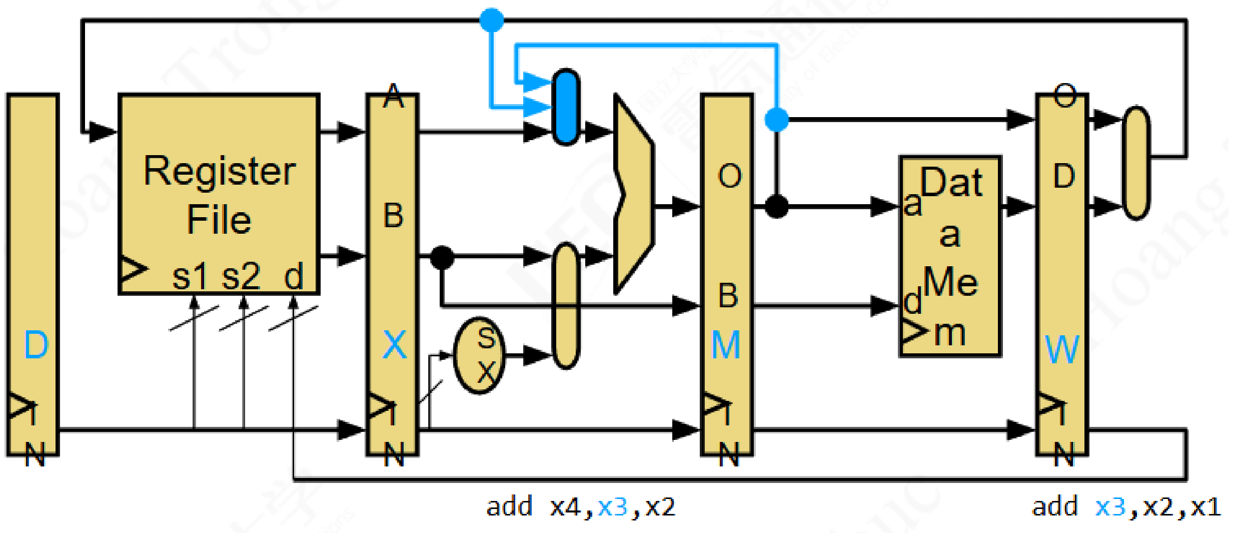

Figure 4: Your WX in Pipeline datapath

For this milestone, your datapath should always, by default, fetch the next sequential PC.

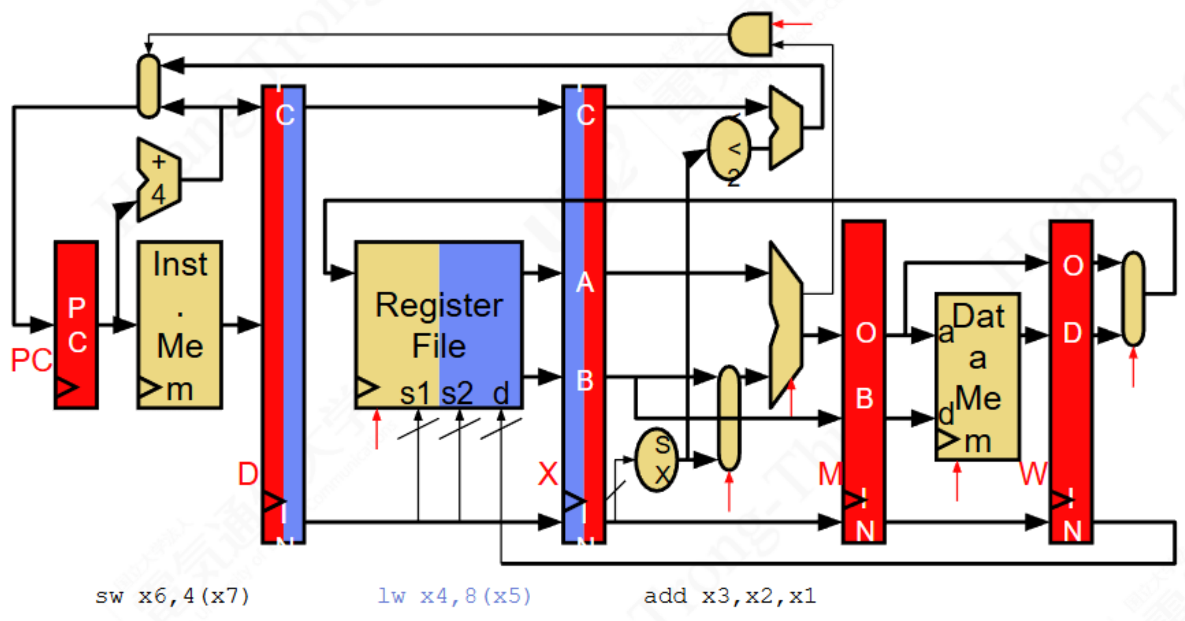

Figure 5: Your fetching instruction in Pipeline datapath

Branch directions should be determined in Execute.

On a taken branch, your datapath will flush the instructions in Fetch and Decode (replacing them with NOPs/bubbles) and then fetch the correct-path instruction in the following cycle (when the branch moves to the Memory stage).

The pipelining lecture slides discuss the cycle timing in detail.

The riscv-tests (except for **‘simple‘**) all exercise a variety of bypasses, so you’ll need all of that working before you can run them.

2.2 Datapath trace outputs

To validate your pipelined design more fully, we have also introduced cycle-level test traces in this homework.

Your datapath will need to set the ‘trace_*‘ output signals to identify, in each cycle, what is happening in the Writeback stage.

See the documentation in `‘DatapathPipelined.v‘` on these ports for details.

We have provided cycle-by-cycle traces of the expected behavior of your processor in the ‘trace-*.json‘ files.

2.3 Milestone 2: All RV32IM Instructions

For the second milestone, you will need to handle the rest of the RV32IM instructions.

With the presence of load instructions comes the possibility of load-use dependencies, which must be handled via stalling.

You should implement stalling in the Decode stage, i.e., if there is a load in Execute and a dependent instruction in Decode, in the next cycle the load should advance to Memory, the dependent instruction should remain in Decode, and Execute should be filled with a NOP.

You will also need to add WM bypassing to your pipeline.

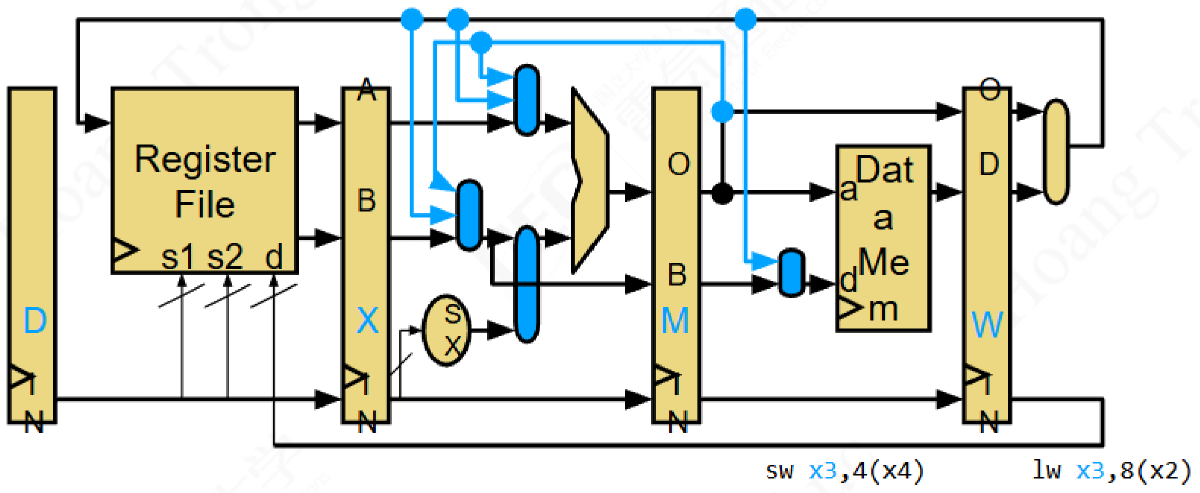

Figure 6: Your WM bypassing in Pipeline datapath

2.4 Divide/remainder operations

You will need to support the division and remainder operations. For simplicity, we'll discuss only division, as the remainder is handled identically.

Your divide operations should use the pipelined divider from homework #4.

Since a divide takes 8 cycles, its quotient will not be available until much later than other ALU operations.

Divide operations should proceed to the M stage after they complete the divider pipeline.

This keeps dividing similar to other inst. and allows for regular MX bypassing in cases like the one below, where a (non-divide) consumer inst. immediately follows a divide inst.:

```
    div x1,x2,x3 F D X0 X1  X2 X3 X4 X5 X6 X7 M W
    addi x4,x1,0   F D  *  *  *  *  *  *  *  X M W
```

Your divide pipeline should also permit back-to-back execution of consecutive ***independent*** divide inst.

```
    div x1,x2,x3 F D X0 X1  X2 X3 X4 X5 X6 X7 M  W
    div x4,x5,x6   F D  X0 X1 X2 X3 X4 X5 X6 X7 M W
```

For *dependent* divide inst., the younger inst. must stall until the older inst. completes the entire divide pipeline.

```
div x1,x2,x3 F D X0 X1 X2 X3 X4 X5 X6 X7 M  W
div x4,x5,x1   F D  *  *  *  *  *  *  *  X0 X1 X2 X3 X4 X5 X6 X7 M W
```

Cycle-level tracing is also enabled for this milestone, this time for the LW and dhrystone tests.

2.5 Memory

Due to pipelining, our memory now works slightly differently than it did in homework #3 and homework #4.

The homework #5 memory uses the same clock as the datapath, and all memory reads/writes occur on the negative edge (half-way through the cycle).

2.6 Implementation Tips

The pipelined design is substantially more complex than our previous designs.

Our reference implementation is over 50% larger than the multi-cycle design.

As signals multiply across the pipeline stages (e.g., you'll want to track the PC for each inst. in each stage now), we recommend a strict naming convention(e.g., use the 'f_' prefix for all Fetch signals, 'd_' for all Decode signals, etc.) to make it easy to identify which signal corresponds to which stage.

Instead of copying over all of your homework #4 code at once and adding pipeline stages to it, we recommend you pull in just the parts needed to get each test case working.

This will keep your design as small as possible as long as possible, making it easier to understand and debug.

The testcases we have provided are relatively limited.

Adding your own tests will help you uncover bugs before they crop up in a larger riscv-test or dhrystone which are harder to understand.

Alternatively, if you do discover a bug via a larger test, consider trying to reproduce the bug via a smaller test.

This can make it easier to check your fix and to ensure that it stays fixed as you work on other parts of the design.

2.7 Submitting your code

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

For this homework, you will again run synthesis and place-and-route to see how much your improved divider design affects overall frequency.

We will need you to run the waveform for this assignment.

Submit your **DatapathPipelined.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