NCTU-EE DCS – 2017

Final Project Part 2

Design: Pipelined CPU

Data Preparation

1. Extract test data from TA's directory:

   % tar xvf ~dcsta02/ Final_Project_Pipeline.tar

2. The extracted directory contains:
   a. Exercise/ :your design

Design Description and Example Wave

An arithmetic logic unit (ALU) is a digital electronic circuit that performs arithmetic and bitwise logical operations on integer binary numbers. In this exercise, a string of instructions will be given. Your job is to decode these instructions and execute. In order to increase clock frequency and throughputs, pipeline technology can be used. Instructions will be given continuously with pipeline. You should try to design a simple 19-bits CPU with 16 registers. The basic required instruction set are following: (similar to MIPS)

Type	Instruction Format				
Logic	opcode (3-bit)	rs (4-bit)	rt (4-bit)	rd (4-bit)	func (4-bit)
Operation	opcode (3-bit)	rs (4-bit)	rt (4-bit)	rd (4-bit)	rl (4-bit)
Immediate	opcode (3-bit)	rs (4-bit)	rt (4-bit)	immediate (8-bit)	

Register s(rs), Register t(rt), Register d(rd) and Register l(rl) represent the address of registers. Since the instruction takes 4 bits to store the address, it means we have 16 registers, from r0 to r15. Each register reserve 16 bits to store data, e.g. rs = 4'b0000 means one of operands is "r0". rt = 4'b0101 means one of operands is "r5", and so on. And the outputs are corresponding to the registers, ex: out = r0 when we check your answer.

There will be a 16*16 data memory in our pattern. You can access this memory according to the instruction.

Function	Type	Definition	Instruction
AND	Logic	rd = rs & rt	000-ssss-tttt-dddd-0000
OR	Logic	rd = rs | rt	000-ssss-tttt-dddd-0001
ADD	Operation	rd = rs + rt	000-ssss-tttt-dddd-0010
SUB	Operation	rd = |rs – rt|	000-ssss-tttt-dddd-0011
MUL	Operation	{rd, rl} = rs * rt	001-ssss-tttt-dddd-llll
ADDI	Immediate	rt = rs + immediate	010-ssss-tttt-iiii-iiii
SUBI	Immediate	rt = |rs – immediate|	011-ssss-tttt-iiii-iiii
STORE	Immediate	DM[rs[3:0]]=rt+immediate	100-ssss-tttt-iiii-iiii
LOAD	Immediate	rt=DM[rs[3:0]]+immediate	101-ssss-tttt-iiii-iiii

Ex1:

instruction[18:0] = 19’b001_0000_0001_0010_1111

{r2, r15} = r0 * r1

out = r2&r15

out_addr = 4‘d2 + 4‘d15

Ex2:

instruction[18:0] = 19’b100_0010_0011_1110_0000

addr[3:0] = r2[3:0]

wen = 1’b1

din[15:0] = r3 + 8’b11100000

out[15:0] = r3 + 8’b11100000

out_addr = r2[3:0]

Inputs

1. All input signals will be **synchronized** at **negative edge** of the clock.

I/O	Signal name	Description
Input	**clk**	Clock
Input	**rst_n**	Asynchronous active-low reset. You should set all your outputs and registers to 0 when rst_n is low
Input	**in_valid**	Input signals is valid when in_valid is high. in_valid will keep high continuously for 150 cycles.
Input	**instruction[18:0]**	instruction[18:0] is valid when in_valid is high.
Input	**dout[15:0]**	When ren is high, you can get data dout[15:0] from memory at next cycle.

Outputs

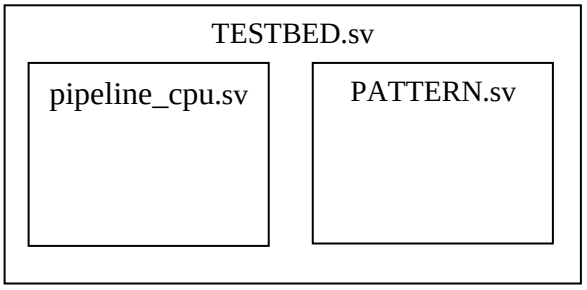
1. Output signals are synchronized at clock positive edge.
2. The TA's pattern will capture your output for checking at clock negative edge.

I/O	Signal name	Description
Output	**out_valid**	**out_valid** should be low after initial reset and can be raised when **in_valid is high**. **out_valid** should be set to high when your output value is ready.
Output	**out[15:0]**	We will check out[15:0] when **out_valid is high.** out[15:0] is the result from the instruction.
Output	**out_addr[3:0]**	We will check out_addr[3:0] when out_valid is high. out_addr is the address of destination register.
Output	**ren**	Read enable (ren) is an input signal of memory. When ren is high, you can read data from memory at next cycle. ren should be high for only one cycle per instruction.
Output	**wen**	Write enable (wen) is an input signal of memory. When wen is high, you can write data into memory. wen should be high for only one cycle per instruction.
Output	**addr[3:0]**	When ren or wen is high, the address of memory will be accessed.
Output	**din[15:0]**	When wen is high, din[15:0] will be written into memory.

Specifications

1. Top module name: **pipeline_cpu** (Filename: **pipeline_cpu.sv**)
2. Input ports: **clk, rst_n, in_valid, instruction[18:0], dout[15:0]**
3. Output ports: **out_valid, ren, wen, addr[3:0], din[15:0], out[15:0], out_addr[3:0]**
4. It is an **asynchronous active-low reset** and **positive edge clk** architecture
5. The clock period of the design is **3.5ns**.
6. The input delay is set to **0.5*(clock period)**.
7. The output delay is set to **0.5*(clock period)**, and the output loading is set to **0.05**.
8. Overflow is allowed in your design.
9. After synthesis, please check **syn.log** file that can not include any **Latch & Error**.
10. After synthesis, you can check **pipeline_cpu**.area and **pipeline_cpu**.timing. The area report is valid when the slack in the end of **timing report** is **non-negative**.
11. The latency for one case should be smaller than **(pipeline stage + instruction numbers + 5) cycles**.

Block Diagram



Note

1. Grading policy:
   a. Pass the RTL simulation
   b. Synthesis successful
      (**no Latch and Error**)
      (**slack in the timing report is non-negative**)
   c. Using For loop is not allowed.
   d. All registers should have reset function.
   e. Plagiarism is not allowed.

f. Function correct 70%, area 30%

2. Template folders and reference commands:

   a. 01_RTL/ (RTL simulation)**./01_run**

   b. 02_SYN/ (Synthesis)**./01_run_dc**

(Check the design which contain **latch** or not in **syn.log**

(Check the design's timing in /Report/**pipeline_cpu.timing**)

Sample Waveform