CPU Project Design

CPU Component Recall

- Instruction memory(pc).
- Registers module.
- ALU.
- Data memory.
- Controller.
- Some multiplexers in between.
- MemORIO (I/O device)

Modules Specification

1. Controller

Input

- `[5:0] Opcode` // from instruction memory: instruction[31:26]
- `[5:0] Function_opcode` // from instruction memory: r-form instructions[5:0]
- `[21:0] Alu_resultHigh` // from the Executer: Alu_Result[31..10] (highest 22 bits to distinguish between I/O and memory)

Output

- `Jr` // 1 means "jr", 0 means not
- `Jmp` // 1 means "jump", 0 means not
- `Jal` // 1 means "jal", 0 means not
- `Branch` // 1 means "beq" , otherwise it's not
- `nBranch` // 1 means "bne" , otherwise it's not
- `RegDST` // 1 means destination register is "rd", otherwise it's "rt"
- `MemOrIOtoReg` // 1 means that data needs to be read from **memory** or **I/O** to the register (i.e. reading is needed)
- `RegWrite` // 1 means to write to the register
- `MemRead` // 1 means to read from the memory
- `MemWrite` // 1 means to write to the memory
- `IORead` // 1 means I/O read
- `IOWrite` // 1 means I/O write
- `ALUSrc` // 1 means the 2nd operand is immediate (except for "beq","bne")
- `Sftmd` // 1 means to shift
- `I_format` // 1 means the instruction is I-type except for "beq","bne","LW" or "SW"
- `[1:0] ALUOp` // if the instruction is R-type or I-format, `ALUOp[1]` is 1, if instruction is "beq" or "bne", `ALUOp[0]` is 1

2.Executer(ALU)

Input

- `[31:0] Read_data_1` // from IF
- `[31:0] Read_data_2` // from IF
- `[31:0] Sign_extend` // from IF, sign-extended immediate
- `[5:0] Function_opcode` // instructions[5:0]
- `[5:0]Exe_opcode` // instructions[31:26]
- `[1:0]ALUOp` // from controller, check I/R format and beq/bne
- `[4:0]Shamt` // instruction[10:6], the amount of shift bits
- `Sftmd` // from controller, 1 means it is a shift instruction
- `ALUSrc` // from controller, 1 means the 2nd operand is immediate (except for "beq","bne")
- `I_format` // from controller, 1 means the instruction is I-type except for "beq","bne","LW" or "SW"
- `Jr` // from controller, 1 means "jr", 0 means not
- `[31:0] PC_plus_4`// pc+4

Output

- `Zero` // 1 means the calculation result is 0
- `[31:0] Addr_Result` // result of address
- `[31:0] ALU_Result` // result of data

3. Data memory

Input

-`[31:0] address` // from MemOrIO, rooted from Executer. -`[31:0] write_data` // from IF, Read_data_2 -`Memwrite` // from controller, 1 means write, 0 means read -`clk` // system clock

Output

-`[31:0] read_data` // data read out from memory

4. IFetch

Input

- `clk,rst`
- `[31:0] Addr_result` // address calculated by Executer
- `Zero` // 1 means the ALU_result is 0
- `[31:0] Read_data_1` // used by "jr"
- // from controller
- `Branch` // 1-beq
- `nBranch` // 1-bne
- `Jmp` // 1-j
- `Jal` // 1-jal
- `Jr` // 1-jr

Output

- `[31:0] instruction`// instruction read out
- `[31:0] branch_base_addr` // actually **pc+4**, to ALU, for branch use
- `[31:0] link_addr` // actually **pc+4**, for "jal" use
- `[31:0] pc`

> need a next_pc inside to update pc using FSM

5. Decoder

Input

- `[31:0] Instruction` // from IF
- `[31:0] read_data` // data from Dmemory or I/O port
- `[31:0] ALU_result` // from Executer
- `Jal` // from controller, 1 means "jal", 0 means not
- `RegWrite` // from controller, 1 means to write to the register
- `MemToReg` // from controller(MemOrIOtoReg), 1 means the data to wirte is from Dmemory, 0 means from AlU
- `RegDst` // from controller, 1 means destination register is "rd"(instruction[20:16]), otherwise "rt" (instruction[15:11])
- `clk`
- `rst`
- `[31:0] opcplus4` // from controller(link_addr), for "Jal" use.

Output

- `[31:0] read_data_1`;
- `[31:0] read_data_2`;
- `[31:0] sign_extend`; // sign-extended immediate

6. MemOrIO

Input

- `memRead` // read memory, from Controller
- `memWrite` // write memory, from Controller
- `ioRead` // read IO, from Controller
- `ioWrite` // write IO, from Controller
- `[31:0] addr_in` // from **alu_result** in ALU
- `[31:0] mem_read_data` // data read from data memory
- `[15:0] io_read_data` // data read from I/O, 16 bits
- `[31:0] reg_read_data` // data read from decoder(register)

Output

- `[31:0] addr_out` // address to data memory and I/O
- `[31:0] reg_write_data` // data be written to registers (may be from memory or I/O)
- `[31:0] write_data` // data written to memory or I/O

- `LEDCtrl` // LED Control Signal (enable)
- `SwitchCtrl` // Switch Control Signal (enable)
- `TubeCtrl` // Tube Control Signal (enable)

7. myclk

Input

- `clk`
- `rst`

Output

- `myclk` // a clk with frequency we need

Some Modules for I/O

8. IOread

Input

- `reset`
- `ioRead` // from controller, 1 means read from I/O
- `switchctrl` // from MemOrIO, 1 means enable switch
- `[15:0] ioRead_switch_data` // data from switches

Output

- `[15:0] ioRead_data` // give the data to MemOrIO

9. switch_driver

Input

- `clk`
- `rst`
- `switchCtrl` // from MemOrIO, 1 means enable switch read
- `[1:0] switchaddr` // 到switch模块的地址低端!!!
- `switchread` // from controller, 1 means read from switch
- `[15:0] switch_in` // data from board switches

Output

`[15:0] switch_out` // data to MemOrIO

10. LED_driver

Input

- `clk`

- `rst`
- `ledWrite` // from controller(IOWrite), 1 means write to LED
- `ledCtrl` // from MemOrIO, 1 means enable LED
- `[1:0] led_addr` // 到LED模块的地址低端!!!
- `[15:0] led_data` // data to be ready

Output

- `[15:0] led_out` // data signal to board LEDs

11. Tube_driver

Input

- `clk`
- `rst`
- `TubeCtrl` // from MemOrIO, 1 means enable Tube
- `[31:0] in_num` // data from MemOrIO, to be displayed

Output

- `[7:0] segment_led` // the content to be displayed
- `[7:0] seg_en` //enable signal

To be continued:

UART

有些别人的代码，可以参考，但是看不懂，不知道怎么用，先放着。 我们应该先完成所有部分再考虑如何加入`UART`。过于抽象！ 原理大概是例化很多uart模块，通过一个kickoff 信号控制。

Reference Respositories

Certseeds, 2catycm (UART modules)

For more details, see UART.md and the folder `uart_reference`.