Project Functional Description and Design Requirements

Our goal is to design a 3-stage pipelined processor. The 3 stages are called ID EX and WB. The ID stage contains PC register, IMEM memory, BIOS memory and Regfile. The PC register， IMEM and BIOS MEM are designed to stay at the same stage. The first stage also contains control unit, immediate number generator and ALU Ctrl signal generator. The second stage EX, begins with a ID\_EX register module and ends with a EX\_WB register module. This stage has no data store. It contains ALU, forwarding unit and branch comparator. It also contains small module: mux1 and mux2 to select ALU two input signal, change\_mem\_wr module to choose proper data part to write in memory. WB stage includes DMEM memory, BIOS memory, UART memory and CONV2D memory. It also contains a write back data mux and memory read\_data change part. The whole block diagram is shown below.

一些文字和图案

描述已自动生成

Datapath of CPU:

Take R-type ISA and I-type calculating ISA as an example: The IMEM will read the value of PC before the first stage. In the first stage, ID module will decode the fetched instruction and send the information to register, IMM and control unit. Meanwhile, Mux will choose proper PC value to update. The control unit is responsible for controlling the second stage and third stage. The IMM generates data based on type of instructions. After that, all values are sent and stored in ID\_EX module. In ID\_EX module, a data hazard should be dealt with, which is between the 3-stage result and 1-stage regfile data (because actually the write back data is stored in regfile in the “fourth” stage, but ID\_EX must store the regfile data after 1 stage). Then mux on EX module will choose one of the proper data: reg1 data, write back data, pc value (for mux 1) and reg2 data, write-back data and imm data (for mux2). Then ALU will process data based on alu control. The output of ALU will connect with all the memory modules and EX\_WB, but the we use extra control signal will decide which part of memory should be written in. Then in 3-stage data is written back through WB module.

The difference of treating other kind of ISA are shown below:

For J-type and jalr ISA: We calculate the result of jump address through ALU, and output the jump judge signal in the second stage. The next instruction after jump-ISA is useless, so we use a flush signal to clear the data in ID\_EX module.

For branch ISA: The branch comparator lays in the 2-stage, considering the effect of write back data hazard. The branch judge signal will be set out from 2-stage. Branch address is generated in 1-stage IMM but is sent out from 2-stage.

For store type ISA: The data should be chosen the right bit part based on address offset.

For load type ISA: Similar with store type, the address offset value is passed to 3-stage and data is changed in the WB module.

For CSR: If the write back address is equal to 0, the write back enable signal will be set to 0, which means data can’t be written back to regfile.

We use these ways to deal with hazard problem:

For all data hazard, we use forwarding unit to control mux. The unit includes comparation of address, and ISA type judgement, so that which mux should choose forward data can be judged.

For branch control hazard, we use “not taken” prediction, and flush the next instruction’s EX and WB stage. The data hazard in branch is dealt with in the same way of other data hazard.

For jal write back data forwarding, the operation of label instruction begins at the third stage of jal. We need to add wb data to ID\_EX so that the ID\_EX module can store right wb data after the first satge.

Important Sub-pieces of CPU:

Pc output mux: the pc output mux can choose these four data: the normal “pc + 4” data, the branch address, and the judge address. It is controlled by both branch judge signal and jump judge signal.

Control Unit: The control unit should output signal to control: ALU control part; judging UART ISA; CSR written back; branch comparator; write back data type judge (for WB module to choose); setting offset of store data; forwarding unit to judge whether this ISA is possible to cause hazard; jump signal judgement; whether the data will be written back based on ISA type. Since the output signals are various, we use extra lines to gather same type of ISA. For example, except the store and branch ISA, all the instruction data have “wb” process, so we put those to one judge signal.

IMM module: The IMM module will generate all kinds of imm data, so we don’t design some other submodule such as shift number.

ALU control: We use two modules to control ALU. One is control unit, which is responsible for judging what type this ISA is. Another is ALU control part. This can generate 4-bit signal based on ISA type (control unit output) and calculation type (if this ISA is a R-type or calculating I-type signal).

Two mux in EX stage: Then mux on EX module will choose one of the proper data: reg1 data, write back data, pc value (for mux 1) and reg2 data, write-back data and imm data (for mux2).

ALU: ALU unit have two output: one is original output data which can be generated from different types of operation like add, sub. Another is address output data, which is only connect with two input value’s sum. This will reduce fanout, considering that one output will drive many data memory or ALU result registers.

Forwarding Unit fully control the two muxs. It will not only generate output based on data hazard, but also based on the ISA type.

Branch Comparator: We add two separated mux before the input of branch comparator. The comparator output branch judge signal, but do not responsible for branch address. We assume that all branch ISA are “not taken”, and deal with wrong prediction by flush signal.

WB module: The write back data is chosen from these data: UART output data, CONV accelerator output data, ALU calculation result, PC calculation result, Data memory output and BIOS memory output.

%remain how to deal with load forwarding

Other detailed connection:

The pc data is shifted by 2 when connected with IMEM data memory, but the “+4” unit is connected with original pc output.

Conclusion:

1. Details are important. In the first checkpoint, we forget to connect the write back signal to regfile. Although we passed some Riscv151 testbench, it is actually not right answer. The best way to debug simulation is firstly check warning. It usually shows whether your signal connection is correct. The synthesis and implementation part is also a good tool to check: whether the signal has different drivers, whether there is a latch; whether the circuit is combinational logic, etc.
2. Consider more elements when designing the circuit. We firstly treat the data hazard in the same way, ignoring the effect that load forwarding can cause long-time critical path. When doing the work of improving frequency, we find that changing load forwarding is a relatively difficult job, which should be done on previous part. It is better to consider more elements to get a better performance.
3. Consider more about hardware, not just language. We firstly design the FIFO of conv2D accelerator too big. The result is that we failed implementation many times. Besides, the mux part should have less inputs in order to increase the fanout.

Division of labor:

My work in the team:

Before Chekpoint 1:

Design the small submodules of CPU and connect them. (For the preparation of integration, like EX big module).

Test the Riscv151\_testbench.v and correct the wrong connection.

Before Checkpint 2:

Design the original UART signal controller part.

Work with partner to debug UART receiver, transmitter and controller part.

Before Checkpoint 3:

Design the conv2D controller part.

Work with partners to debug the circuit to pass testbench and bitstream.

Before Checkpoint 4:

Try the innovation part: add dynamic branch predict to the original circuit, and test it.

Change the organization of ALU and control unit, to reduce the fanout.

Try to delete load forwarding part to increase the frequency. (Finally failed).