103011227 report

作者

周延儒

project2 pipeline

This is the report for the project 2 ,pipeline. In the previous project , we have already implemented a simulator, and however , it lack performance due to low design complexity as well ass poor throughput. For example, loading data takes the longest time, and thus it is the crucial time for the single cycle cpu performance. So to improve the performance , we introduce the technique ,pipeline,to dramatically improve performance with higher design complexity and high throughput ,too.

First of all , we have to divide the CPU into 5 stages to "pipeline" it.

The 5 stages are IF,ID,EX,DM(MEM I declare in my program),,and WB.

IF: instruction

ID: decode the instruction ,branch/jump instruction, hazard detection(I get to know whether to stall in the previous stage)

EX: ALU for the instruction , calculating the corresponding result.if the instruction is for store/load data, it calculates the data address.

WB: write the corresponding value to the register file.

Moreover, encountering the hazards, we can resolve it with stall ,forwarding ,and flush.

For data harzard:

Stall

Forwarding: in this project , we have EX/DM to ID ,and EX/DM to EX forwarding. Plus, we don’ t the DM/WB forwarding datapath.

Control hazard:

Flush: when the branch instruction takes, we have to flush the instruction we fetch in the IF stage , and insert NOP instruction .

Structural Hazard:

In this project ,there is no kind of hazard.

We declare additional variables for this project

```
int  stalledT;  represent the stall for T or S
int stalledS;

int forwardT; represent the forward for T or S
int forwardS;

int is_fwd;    represent the forward for EX/DM to ID
int is_fwd_ex represent the forward for EX/DM to EX

//foward to ID branch
int ex_dm_fwd_s; represent what is forward to EX/DM to ID
int ex_dm_fwd_t;

//forward to ex
int ex_dm_fwd_ex_s; represent what is forward to EX/DM to EX
int ex_dm_fwd_ex_t;



struct IF_ID_register ID_ins regiser for the instruction we fetch and decode
struct ID_EX_register EX_ins  register for execution for ALU
struct EX_MEM_register MEM_ins;register for the memory access
int dm_val; the dm value for the non store/load instruction , eg add 1,2,3, the dm_val is the value of the value of 2 plus value of 3
int MDR; the memory data register value for the load/store instruction ,eg lw 4(0), the MDR is the value of memory[4+0];
```

`int check_T_stall() ,check_S_stall check whether to stall for ID (notice that if the value is 1 ,it doesn't mean it must stall. We may solve it by forwarding )`

`int ex_write_reg; record the value of the rd for the R-type, rs for the I type . That is , the the output results for the ALU`

`int dm_write_reg; record the value similar to the ex_write_reg, the only diffencet is for the DM(Mem)`

`for the 5 stages we have void WB(),void DM, void EX(),void ID(), void IF()`

Project flow :

First of all , as what we run in the prevoid project single_cycle ,we reads the instructions from the image.bin to my instruction storage `struct instruct_mem [256] im` ,and the data from bimage.bin to the correspingind storage `struct data_mem [] dm`

Once reading all the preliminary data , the pipeline starts to work. It prcess in the order of simulating the pipeline is WB→ DM → EX → ID → IF.

If there is no stall we have to deal with , it begin next stege.

If there is a stall we need to solve, we insert nop to avoid stall. And in the same time we still have to check whether to insert nop to stall again . After we receive 5 halt instruction or the any error we need halt to avoid crashing, the pipeline finish the process , and the CPU has done the duty ,and exit.

The snapshot.rpt is also distinct form that of the single_cycle. In additional to print the register files, we need print the stages status . Most interesting part is that the stages status is the what the next stage is going to do. As a result , I print the register files first ,and print the next stage status.

Detailed description:

WB(): in this stage , it will write back the dm_val,or the MDR depending on the DM_opcode to the correspinging register file.

First , if the instruction is branch type ,JR,NOP ,etc, since we don't have to write anything to the register file , we just return.

Then, we have to check if any data is written to register zero , if so , we print the error message “write 0 error” .

Next , based on the DM_opcode , write the data to the register file,

If the opcode is for load instruction , assign the register with MDR

Otherwise ,assign it with dm_value ,(eg add 2,3,4 dm_val will 3+4 , so 2 will be assigned the value of dm_val)

DM(): (notice that I declare the DM as MEM , they are equivalent.) in this stage , as what we do above , MEM_opcode is EX_ins.opcode. dm_val will be the ALU output result , EX_ins.output_result, and dm_write_reg = ex_write_reg;(dm_write_reg is the value of the register we want to store)

Then , according to the MEM_ins.opcode , we load the data or store the data to MDR or nothing.

If it's the load_data , we assign the value MDR to the mem[Ex_in.output_result(the memory address location)] from dm

IF it’ s the store_data ,we store the value of dm_val_reg to the to the corresponding memory location(EX_ins.output_result or dm_val ,they are the same) in the dm.

Besides , if the instruction is NOP, we can just ignore ,and return .

EX(): in stage is one of the sophistecated part of the stages. For we need to decide whether to use the data by forwarding
According to the type of instruction the , it calculate correspondingly .
First of all, for R-type ,rd =f(rs,rt); take add for example:
If dm_write_reg !=rt,rs , it means there is not depencies . So we can just use ALU to calculates the result.
The EX_ins.output_result the result, and the ex_write_reg is the rd .
If dm_write_reg==rt,rs :this case have different operation,
If dm_write_reg is for storing data, it doesn't need to forwarding
If dm_write_reg is for loading data , it means it loads data from the data memory dm to the dm_write_reg . However, it doesn't get the correct value at this time, so it cannot forward , and it should be detected stall before.
If dm_write_reg is for nop ,JR,branch ,etc , just calculate with no error, for it is nothing for the this instruction.
If it is not the case above , we can forward the data, besides because we don't have DM/WB forward path , we doesn't have to consider this situation . We can get the correct result with the forward data. Moreover, depending on the rs,rt,forward, we set the `ex_dm_fwd_ex_s=EX_ins.rs or ex_dm_fwd_ex_t=EX_ins.rt; is_fwd_ex=1;`

Then for I type,rt rs . there are 3 case arithmetic,store,load: take addi for example addi rt=rs+c_immediate:

In this case we just have to consider rs.

If rs!=dm_write _ reg, still there is no dependencies , so we just calculate

If rs==dm_write_reg:

If dm_write_reg is for load data, we cannot forward ,and hazard is suppose to detect and we insert NOP to stall to avoid .

If dm_write_reg is for store data , as what we dealed with ,just calculate without forwarding.

IF dm_write_reg is NOP,branch , also, just calculate.

If elsewise, we can forward the data , the technique is similar to that of R-type.

For branch type or ,since it is processed by the IDIF, we don't have to do anything.

Plus, if it s NOP, nothing should be done.

In the end, for j and jal. For j ,also nothing is done. Nonetherless, since the jal store the PC value to register 31, so ex_write_reg=31,and the output_result is the pc value. The following cycles may utilizes it as forwarding data.

ID(),IF() , in this stage, ID is for the Conditional branches and unconditional branches are determined during the ID stage.

IF the there is any forwarding , it can forward the data determined whether the branch take action.

Most weird part is that in the IF(),the original function is supposed to fetch the instruction only . Nonetheless, to inplement the function, we can just print the instruction in the store in the IM.

I make a change in the IF() stage. In my opinion , this is on the consequence of the difference of print the registefile and stages in terms of the stage. Printing thr register file is in the current stage,and on the contrary, printing the stages in the next stage. As as result, it truly confused me for a long time. Being changed , the IF function will determined whether **Hazard** happen . If so ,set stall signal to be 1. If not decide if we have to flush the instruction or not . Apart from the modification of the IF(), I aslo change the calling the function . In each cycle, I call it `IF(pc-1, im);`so it can perform the ID function in the right instruction instead of the incorrect one . Nevertheless, aftering finishing the entire design , I’ m of the opinion that IF is quite redundant , I should have implement the hazard detect in the ID() . I think the result is that I wanted to add additional function the IF such as it can deal with the data in advance. It turns out we needn’ t do so.

Back to the IF ID, the most crucial part is the hazard detection , and branch processing.

The stall detection varies from instruction type to type.
Fortunately , it can be categoried into several parts:

- R-type :
  - Of course , it the most common arithmetic instruction: Rd=f(rs,rt) =>so we focus on the relation between rs,rt and the ex_write_reg and dm_write_reg:
    - If ex_write_reg =rs,rt dm_write_reg :

If the ex_write_reg is not for load data, it can forward,and no hazard is detected :

```
Example:
ID: ADD
EX: ADD

EX: ADD fwd_EX-DM_rs_$3
DM: ADD
```

If the ex_write_reg is for the load data, since we need to get the value from the data_memory , we have to stall ,and insert NOP to solve this problem

```
  IF: 0xFFFFFFFF to_be_stalled
  ID: ADD to_be_stalled
EX: LW
```

IF ex_write_reg is storing date or branch or Jr etc , Since nothing is going happen

```
ID: ADD
EX: SW
```

- If ex_write_reg!=rs,rt, dm_write_reg ==rs,rt

  Since we don't have DM/WB forwarding , we have no choice but to stall .

  But if the dm_write_reg is store data or branch ,j Nothing is done.

```
   IF: 0xFFFFFFFF to_be_stalled
   ID: ADD to_be_stalled
   EX: NOP
    DM: ADD
```

- If Ex_write_reg==rs,rt and dm_write_reg, in this part ,we need to know what kind of instruction the ex_write_reg and dm_write_reg, sometimes is the double dat hazard. In this project we can solve by forward

```
EX: ADD fwd_EX-DM_rs_$3
 DM: ADD
WB: ADD
```

Otherwise depending on the rule we discuss ,we can determined whether to stall or not.

- Ex_write_reg !=rs,rt dm_write_reg!=rs,rt : of course, there is no dependencies, and so nothing is done.

- I type:

In here there only rs ,rt . So the case is simpler than that of the R-type

- arithmetic type

  for type such as addi,ori,etc . the stall harzard detecting is as same as that of R-type. We only have to care about the dependencies of the rs.

- Load store type:

  For type such sw, lw, We calculate the address with rs and c_immediate ,and the data loaded or stored .

  We need to know the value of rs rt . The rule is the same as we discuss:

  Eg:

```
EX: LW fwd_EX-DM_rs_$3
DM: ADD
We forward rs

ID: LW to_be_stalled
EX: NOP
DM: ADD
Since we don't have DM/WB forwarding path , we need to stall
```

- LUI type: since rt=C<<16 ,there won’ t be any hazard.
- Branch type: this type is the most tricky type in my opion . we need compare rs ,rt type to determined whether flush or . Before execute the comparison , we need make sure whether any stall exist.

IF ex_write_reg ==rs,rt:

If ex_write_reg is not for the store data ,where it take no effort here, we have to stall ,for we can not get the correct value when ALU operate and ID compared. Fortunately , after stall for cycle, we have the chance to forward under the condition that the expected forwarding register is not for load data:

Eg:

```
ID: BNE to_be_stalled
EX: ADD
DM: NOP

Next stage:

 ID: BNE fwd_EX-DM_rt_$3
 EX: NOP
DM: ADD
```

If ex_write_reg !=rs,rt && dm_write_reg ==rs,rt

Adopting the technique we learn in the R-type , we can decide whether to stall or forward.

If dm_write_reg is for non load_data, we can forward the rs,rt .if dm_write_reg is fo store date, nothing happen (store data takes no effect)

The example is actually that above

If dm_write_reg is for load date, because we haven’ t get the correct value so we have to stall

```
IF: 0x00000000 to_be_stalled
ID: BNE to_be_stalled
EX: NOP
DM: LW
```

If there exist double data hazard , we can stall one time and forward

```
ID: BNE to_be_stalled
EX: ADD
 DM: ADD
 Next stage(add all write to $3)

ID: BNE fwd_EX-DM_rt_$3
EX: NOP
 DM: ADD
```

After determined the harad, according to the insruciton , it decide whether to go the the pc value. If the branch is taken , we need to flush the instruction we have fetched . Otherwise ,continue the process. Notice that the branch instruction takes no effect on the following EX,DM,WB, so it couldn't any forward path from them to the corresponding instruction

- J type :

  Because we just jump to the corresponding PC value, so there is no hazard to stall. Nontheless , the jal instruction will store the PC value in the WB, so there may be forwarding path from jal to the following instructions in the next few cycles.

All after we check whether we have to stall or not, the pipeline program will print the stage information . If stall happens , we insert nop ,and go to the **stall type stag**e. In this stage , EX will be assigned zero ,and operate WB(),DM(). However, we still need to check whether to stall again or we can solve by forward.
So we apply the technique again to know what we should next .

After all the functions are done , the desigin is completed.

Testcase:

Based on the prevous testcase, I improve the testcase

These are my instruction

```
instruction(<#string a#>, <#int rs#>, <#int rt#>, <#int rd#>,
<#int shame#>, <#int function#>, <#int c_immediate#>)
    instruction("addi", 3, 4, 0, 0, 8, 10);
    instruction("addi", 3, 4, 0, 0, 8, 10);
    instruction("add", 3, 4, 5, 0, 32,0);
    instruction("add", 5, 8, 7, 0, 32, 0);
    instruction("BNQ", 3, 4, 0, 0, 0, -2);
    instruction("addi", 0, 8, 0, 0, 0, 0xfff0);
    instruction("bgtz", 8, 0, 0, 0, 0, 1);

    instruction("ori", 6, 7, 0, 0, 0, 0x4fab);
    instruction("nori", 2, 3, 0, 0, 0, 0xffab);
    instruction("and", 3, 10, 11, 0, 0x24, 0);
   instruction("sh", 0, 8, 0, 0, 0, 128);
    instruction("lw", 0, 15, 0, 0, 0, 128);
    instruction("sb", 0, 11, 0, 0, 0, 255);
    instruction("lw", 0,25, 0, 0, 0, 252);

    instruction("addi", 2, 14, 0, 0, 0, 0x0010);
    instruction("addi", 3, 17, 0, 0, 0, 0x0010);
    instruction("lw", 14, 17, 0, 0, 0, 0);
    instruction("addi", 11, 12, 0, 0, 0, 0x0050);
    instruction("addi", 11, 12, 0, 0, 0, 0x0050);
    instruction("addi", 11, 12, 0, 0, 0, 0x0050);
    instruction("addi", 12, 11, 0, 0, 0, 0x0abc);
    instruction("add", 12, 11, 25, 0, 32, 0);


    instruction("addi", 0, 18, 0, 0, 0, 0x0010);
    instruction("addi", 0, 19, 0, 0, 0, 0x0003);
    instruction("sw", 18, 19, 0, 0, 0, 0);
        instruction("lw", 18, 25, 0, 0, 0, 1);

    instruction("add", 0, 0, 26, 0, 32, 0);
    instruction("sw", 0, 26, 0, 0, 0, 0);
    instruction("lw", 0, 30, 0, 0, 0, 0);
    instruction("sub" ,7, 8, 16, 0, 0x22, 0);
    instruction("jal", 0, 0, 0, 0, 0, 48);
    instruction("jr", 31, 0, 0, 0, 0x08, 0);

    instruction("nop", 20, 0, 0, 0, 0, 0);
    instruction("ori", 0, 20, 0, 0, 0, 0xabcd);
    instruction("ori", 0, 21, 0, 0, 0, 0xabcd);
```

```
instruction("beq", 20, 21, 0, 0, 0, 2);
instruction("beq", 20, 21, 0, 0, 0, 1);

instruction("addi", 0, 20, 0,0, 0, 0x2cde);
instruction("sw", 0, 20, 0, 0, 0, 0);
instruction("lb", 0, 10, 0, 0, 0, 0x0003);

instruction("lw", 0, 27, 0, 0, 0, 0);
instruction("sll", 0, 11, 6, 10, 0x00, 0);
instruction("addi", 0, 28, 0, 0, 0, 0x00ff);
instruction("bgtz", 28, 0, 0, 0, 0, 3);


instruction("addi", 0, 23, 0, 0, 0, 0x9fab);
instruction("addi", 22, 22, 0, 0, 0, 0x0001);
instruction("bne", 23, 22, 0, 0, 0, -5);
instruction("beq", 22, 23, 0, 0, 0, 3);
instruction("nand", 15, 16, 17, 0, 0x28, 0);
instruction("nor", 17, 3, 2, 0, 0x27, 0);
```

In my opinion , the most tricky part is NOP instruction .

Since nop is the special case of the sll, so I decide to insert nop instruction . Morever , the rd is don'tcare , I assign rd be non-zero and zero respective .

The total cycle is around 28000s cycle since I have bne beq for loop .