CENG 531 - ADVANCED COMPUTER ARCHITECTURE

MURAT MİRAN KÖKSAL
1595354

PROJECT: PHASE I
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROLDATA REGISTER</td>
<td>2</td>
</tr>
<tr>
<td>PROCEDURE S_IF PROLOGUE</td>
<td>2</td>
</tr>
<tr>
<td>PROCEDURE S_ID PROLOGUE</td>
<td>2</td>
</tr>
<tr>
<td>PROCEDURE S_ID INTERMISSION</td>
<td>3</td>
</tr>
<tr>
<td>PROCEDURE S_EX INTERMISSION</td>
<td>5</td>
</tr>
<tr>
<td>PROCEDURE S_WB EPILOGUE</td>
<td>5</td>
</tr>
<tr>
<td>CPI COMPARISON</td>
<td>6</td>
</tr>
</tbody>
</table>
**CONTROLDATA REGISTER**

is_branch_taken:

1-bit controller for nullifying the extra instruction that enters the pipeline after executing a taken branch.

set_nostat:

1-bit controller used in writeback stage to prevent the squashed instruction from being counted as an executed instruction.

p_is_branch:

1-bit controller which is actually the same as the global latch is_branch. p_is_branch is used instead of is_branch in the modified code to prevent possible conflict or complication of data.

p_load_flag_e & p_load_flag_m:

1-bit controllers which have the same functions as the global latches load_flag_e and load_flag_m. They are used instead of load_flag_e and load_flag_m because of the same reason explained above.

**PROCEDURE S_IF PROLOGUE**

Lines added to reset the new elements of the controldata register.

```
set_nostat = 0;
p_is_branch = 0;
p_load_flag_e = 0;
p_load_flag_m = 0;
```

**PROCEDURE S_ID PROLOGUE**

Following lines are added before decoding the fetched instruction.

```
if is_branch_taken == 1 then
begin
    set_nostat = 1;
    ir = 0;
end;
```

This code snippet simply checks the value of is_branch_taken in the controldata register and if it is 1 takes the steps required to squash the instruction which is about to be decoded in ID stage. is_branch_taken of IF stage is set by prologue of EX stage when a branch is found to be taken. The reason for doing so is that whenever a branch gets to EX stage, it is known that ID stage has the delay instruction which is provided by the compiler and that instruction has to be executed; but the instruction which is read from the memory can be either correct or incorrect according to the direction of the branch in EX. The instruction in IF will always be the one which comes right after the delay instruction in the program and it is okay to execute that instruction when the branch is not taken. But in case of a taken branch, the instruction which has just been fetched from the memory must be squashed. The only stage we can squash that extra instruction is ID stage where decoding of instructions happens. We have to make sure whether the branch in EX is taken or not before decoding the incorrect instruction in ID. Therefore, whenever the direction of the branch in EX stage is found to be taken, is_branch_taken of IF stage is set by EX so that on the next cycle it will be known by the ID prologue that the instruction specified by the instruction register ir of that stage is an incorrect one and ir will be overwritten by the code of sll instruction.
Also, set_nostat is set to 1 when an incorrect instruction is removed since we don’t want it to be counted in writeback stage as an executed instruction.

**PROCEDURE s_ID INTERMISSION**

As mentioned in the section where modifications to the controldata register are explained, is_branch, load_flag_e and load_flag_m are changed to p_is_branch, p_load_flag_e and p_load_flag_m to remove possible risk of complications that may occur when global latches are used.

Aim of the modifications was to make the description resolve branches at EX cycle as opposed to ID cycle. There are different types of branches in the mips instruction set which are shown in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition type</th>
<th>Instruction type</th>
<th>Instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>equal</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>BGEZ</td>
<td>gez</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>BGEZAL</td>
<td>gez</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>BGTZ</td>
<td>gtz</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>BLEZ</td>
<td>lez</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>BLTZ</td>
<td>ltz</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>BLTZAL</td>
<td>ltz</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>BNE</td>
<td>neq</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>J</td>
<td>z</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>JAL</td>
<td>z</td>
<td>1</td>
<td>integer</td>
</tr>
<tr>
<td>JALR</td>
<td>u</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>JR</td>
<td>u</td>
<td>0</td>
<td>integer</td>
</tr>
<tr>
<td>BC1F</td>
<td>equal</td>
<td>0</td>
<td>float</td>
</tr>
<tr>
<td>BC1T</td>
<td>equal</td>
<td>0</td>
<td>float</td>
</tr>
</tbody>
</table>

As seen in the table, except two instructions, all of the branches are of integer class which means they are send to EX in the end of ID. The other two branches, which are of float class, are send to FADD1, so there is no need to modify them since what we want to do is to resolve branches at the end of EX. Still, float branches can be modified to resolve at FADD cycles which require deletion of unwanted instructions in case of a taken branch. But current modifications do not involve that.

Below is the original mips code at the end of the ID intermission stage. Modification on the code will be explained on this code snippet.

```c
if (i_type == branch_type_0) |
  (i_type == branch_type_1) then
begin
  if c_what ^= condition_z then
    branch_target=my_pc +
    sign_extend_14(immediate);

if i_class == float_class then
begin
  if fpr_tag[F_tag_CpC] then
    begin
      # builtin printf("Stall on CPC\n");
      stall float_cc;
    end;
    branch_input=fpr[$CpC] == tf;
    branch_instruction_addr=my_pc;
  end
else
```
begin
    do_forwarding_to_id;
    condition_code(lop, rop);
    branch_instruction_addr = my_pc;
end;
end;

Now that integer class branches will be resolved at EX cycle, it is clear that only float class branches will be processed at ID cycle. Therefore, there is no more need to check the condition type in ID because condition_z branches are integer class branches and they will be processed in EX cycle after modifications. So, the condition of “if c_what ^= condition_z then” is removed. As a result, branch target of float branches (integer class or float class including condition_z branches) will be calculated in ID. There is one thing to be careful about here. Even though integer class branches will be processed in EX, they will still be processed in ID before being sent to EX. Because ID cycle provides necessary stalls for all instructions including branches to prevent hazards. If a branch instruction in ID is not stalled, it will be able to enter the if block above and get its target address calculated. By simply moving the address calculation into “if i_class == float_class then” block, it is guaranteed that in ID cycle only the target address of float class branches are calculated. Now the code looks like following:

```c
if (i_type == branch_type_0) |
    (i_type == branch_type_1) then
begin
    if i_class == float_class then
    begin
        branch_target = my_pc + sign_extend_14(immediate);
    ...

Integer branches will fall into else block of “if i_class == float_class then”. The first two lines there are the ones that find the branch direction. Then, they are the lines which should be disallowed, because they do the job which we want to do in EX. The third line is unnecessary and can be removed. According to these, whole modified code snippet is as following. do_forwarding_to_id is still there but does no harm.

```c
if (i_type == branch_type_0) |
    (i_type == branch_type_1) then
begin
    if i_class == float_class then
    begin
        branch_target = my_pc + sign_extend_14(immediate);
        if fpr_tag[F_tag_CpC] then
        begin
            # builtin printf("Stall on CPC\n");
            stall float_cc;
        end;
        branch_input = fpr[$CpC] == tf;
        branch_instruction_addr = my_pc;
    end
    else
    begin
        do_forwarding_to_id;
    end;
end;
```
PROCEDURE s_EX intermission
This procedure is not included in the unmodified mips definition and added to the modified one.

procedure s_EX intermission
begin
  p_is_branch = (i_type == branch_type_0) | (i_type == branch_type_1);

  if (p_is_branch == 1) & (i_class == integer_class) then
    begin
      if c_what ^= condition_z then
        branch_target=my_pc + sign_extend_14(immediate);

      condition_code(lop,rop);

      is_branch_taken = branch_input;
      is_branch_taken [s_IF] = branch_input;
    end;
  end s_EX;

  Note that the procedure belongs to intermission stage of EX cycle. The selection of minor-cycles is important because it may cause timing problems which as a result causes segmentation faults. These lines can also be put into prolog stage.

  Content of the procedure is very similar to the ones in ID intermission. Since we only move branch instructions which are of integer class, we need to put the necessary codes in the first place. If an instruction is integer class branch, then it will get into block where target address is calculated and branch direction is found. Previously, the condition of branch computation was removed in ID (for float class branches) because there was no need for it. Here we keep the condition in place because condition_z branches are integer class branches and their target is calculated in ID stage (in mips instruction set description). After address computation, condition code of branch_input is set which is basically the control bit for the direction of the branch. This bit is written to is_branch_taken element of controldata register of both this cycle (which is unnecessary and not used in following stages of MEM and WB) and IF cycle (which was explained in previous sections).

  Normally, it can be dangerous and should not be allowed to write to controldata register of another pipeline stage. Before doing this, I tried a couple of different solutions which had nothing to do with other pipeline stages’ registers. But they all failed probably because of timing and synchronization problems between prologue, intermission, and epilogue procedures. The last modification I made before this one caused segmentation fault in simulations even though it seemed to be correct, and I was unable to find the problem. In the end, I tried this one and it worked without problem.

PROCEDURE s_WB epilogue
Writeback epilogue is the last stage an instruction visits. In the original mips description, it is the stage where instructions are retired from the processor. In the modified version the stage still does the same thing but after checking the control bit, set_nostat. Remember from the ID prologue procedure that incorrect instructions are marked by setting set_nostat so that after those instructions are nullified they will not be considered as executed instructions when they retire. Also, is_branch_taken is reset in this stage. The code is as following:
is_branch_taken = 0;
if set_nostat == 1 then
    retire nostat
else
    retire stat;

### CPI COMPARISON

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPI (unmodified – mips_1)</th>
<th>CPI (modified – mips_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>1.290691</td>
<td>1.377080</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>1.445955</td>
<td>1.605474</td>
</tr>
<tr>
<td>126.gcc</td>
<td>1.443177</td>
<td>1.560291</td>
</tr>
<tr>
<td>129.compress</td>
<td>1.218427</td>
<td>1.294637</td>
</tr>
<tr>
<td>130.li</td>
<td>1.334951</td>
<td>1.454516</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>Benchmark did not work because of some library dependency.</td>
<td></td>
</tr>
<tr>
<td>134.perl</td>
<td>1.468078</td>
<td>1.568197</td>
</tr>
<tr>
<td>147.vortex</td>
<td>1.456300</td>
<td>1.551932</td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>1.385670</td>
<td>1.468294</td>
</tr>
<tr>
<td>102.swim</td>
<td>1.350756</td>
<td>1.405759</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>1.398705</td>
<td>1.455541</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>1.458090</td>
<td>1.522100</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>1.192154</td>
<td>1.210329</td>
</tr>
<tr>
<td>110.applu</td>
<td>1.404129</td>
<td>1.430604</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>1.312863</td>
<td>1.366246</td>
</tr>
<tr>
<td>141.apsi</td>
<td>1.462512</td>
<td>1.532201</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>1.353128</td>
<td>1.374684</td>
</tr>
<tr>
<td>146.wave5</td>
<td>1.446746</td>
<td>1.507401</td>
</tr>
</tbody>
</table>

Arithmetic mean 1.377784 1.452075
Harmonic mean 1.372473 1.444472