北航2021计组P6设计文档

黄雨石20376156

一、CPU设计方案

(一) 总体设计概述

本`CPU`为`verilog`实现的流水线`MIPS - CPU`，支持的`MIPS-lite3`指令集包含`{ lb, lbu, lh, lhu, lw, sb, sh, sw, add, addu, sub, subu, mult, multu, div, divu, sll, srl, sra, sllv, srlv, srav, and, or, xor, nor, addi, addiu, andi, ori, xori, lui, slt, slti, sltiu, sltu, beq, bne, blez, bgtz, bltz, bgez, j, jal, jalr, jr, mfhi, mflo, mthi, mtlo }`共50条指令 处理器为五级流水线设计。为了实现这些功能，`CPU`主要包含了`PC、CMP、GRF、ALU、DP、EXT、 CTRL、BE`以及每级模块之间的流水寄存器模块、确定转发和暂停机制的处理冒险的模块单元等

(二)关键模块定义

- **F级**

  1.PC

端口名	方向	描述
npc[31:0]	I	D级npc当前值
pc[31:0]	O	F级pc当前值
pwe	I	使能信号
reset	I	复位信号
clk	I	时钟信号

- **D级**

  1.D_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
instr_in[31:0]	I	F级指令
instr_out[31:0]	O	F级流水过来的指令
pc_in[31:0]	I	F级pc值
pc_out[31:0]	O	F级流水过来的pc值

2.D_CMP

端口名	方向	描述
rs_d[31:0]	I	D级rs寄存器数据
rt_d[31:0]	I	D级rt寄存器数据
type[2:0]	I	选择信号
b_j	O	是否跳转

- type:
  - 3'd0: 跳转指令为`beq`
  - 3'd1:跳转指令为`bne`
  - 3'd2:跳转指令为`blez`
  - 3'd3:跳转指令为`bgez`
  - 3'd4:跳转指令为`bltz`
  - 3'd5:跳转指令为`bgtz`

3.D_EXT

端口名	方向	描述
imm16[15:0]	I	D级16位立即数
ext_sel	O	选择信号
ext_out[31:0]	I	扩展后结果

- ext_sel:
  - 1'b0:0拓展
  - 1'b1:符号拓展

4.D_GRF

端口名	方向	描述
rs[4:0]	I	rs寄存器地址
rt[4:0]	I	rt寄存器地址
gwa[4:0]	I	写入寄存器的地址
gwd[31:0]	I	写入寄存器的值
rs_d	O	rs寄存器的值
rt_d	O	rt寄存器的值
gwe	I	寄存器写使能
reset	I	复位信号
clk	I	时钟信号
W_pc	I	W级pc值

5.D_NPC

端口名	方向	描述
D_pc[31:0]	I	D级pc值
F_pc[31:0]	I	F级pc值
imm26[25:0]	I	26位立即数
imm16[15:0]	I	16位立即数
rs_d[31:0]	I	rs寄存器的值
npc_sel[2:0]	I	选择信号
b_j	I	分支指令是否跳转
npc[31:0]	O	F级pc下一周期的值

  - npc_sel:
    - 3'd0: `F_pc + 4`
    - 3'd1 `j_i` 类指令 `{D_pc[31:28], imm26, 2'b0}`
    - 3'd2 `j_r` 类指令: `rs_d`
    - 3'd3: `branch` 类指令 `D_pc + 4 + {{14{imm16[15]}}, imm16, 2'b0}`

- **E级**

1.E_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
instr_in[31:0]	I	D级指令
instr_out[31:0]	O	D级流水过来的指令
pc_in[31:0]	I	D级pc值
pc_out[31:0]	O	D级流水过来的pc值
ext_in[31:0]	I	D级立即数拓展结果
ext_out[31:0]	O	D级流水过来的立即数拓展结果
rs_d_in[31:0]	I	D级rs寄存器值
rs_d_out[31:0]	O	D级流水过来的rs寄存器值
rt_d_in[31:0]	I	D级rt寄存器值
rt_d_out[31:0]	O	D级流水过来的rt寄存器值

2.ALU

端口名	方向	描述
a[31:0]	I	数据a
b[31:0]	I	数据b
alu_out[31:0]	O	alu计算结果
alu_sel[4:0]	I	选择信号

- alu_sel:
  - 5'd0: `add:a+b`
  - 5'd1: `sub:a-b`
  - 5'd2: `or:a|b`
  - 5'd3: `and:a&b`
  - 5'd4: `xor:a^b`
  - 5'd5: `sll:a<<b`
  - 5'd6: `srl:a>>b`
  - 5'd7: `sra:$signed($signed(a)>>>b)`
  - 5'd8: `slt:($signed(a)<$signed(b))?32'b1:32'b0`
  - 5'd9: `sltu:a<b`
  - 5'd10: `lui:b<<16`

3.E_MDU

端口名	方向	描述
clk	I	时钟信号
reset	I	复位信号
d1[31:0]	I	数据1
d2[31:0]	I	数据2
md_sel[2:0]	I	选择信号
md_stall	O	阻塞信号
md_out[31:0]	O	输出结果

- md_sel:
  - 4'd0: `mult`
  - 4'd1: `multu`
  - 4'd2 `div`
  - 4'd3: `divu`
  - 4'd4: `mfhi`
  - 4'd5 `mflo`
  - 4'd6 `mthi`
  - 4'd7 `mtlo`

- **M级**

  1.M_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
instr_in[31:0]	I	E级指令
instr_out[31:0]	O	E级流水过来的指令
pc_in[31:0]	I	E级pc值
pc_out[31:0]	O	E级流水过来的pc值
alu_in[31:0]	I	E级alu的计算结果
alu_out[31:0]	O	E级流水过来的alu的计算结果
rt_d_in[31:0]	I	流水到E级的rt寄存器的值
rt_d_out[31:0]	O	流水到M级的rt寄存器的值

2.M_BE

端口名	方向	描述
dwa[31:0]	I	地址
dwd_temp[31:0]	I	写入数据（中间）
be_sel[2:0]	I	选择信号
byteen[3:0]	O	替换信号
dwd[31:0]	O	写入数据（最终）

- be_sel:
  - 3'd0: `sw`
  - 3'd1: `sh`
  - 3'd2: `sb`

3.M_DP

端口名	方向	描述
dm_temp[31:0]	I	DM中间结果
dp_sel[2:0]	I	选择信号
dp_a[31:0]	I	地址
dm_out[31:0]	O	DM最终结果

- dm_sel:

    - 3'd0: `lw`
    - 3'd1: `lh`
    - 3'd2: `lb`
    - 3'd3: `lhu`
    - 3'd4: `lbu`
- **W级**

  1.W_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
alu_in[31:0]	I	流水到M级的alu计算结果
alu_out[31:0]	O	流水到W级的alu计算结果
pc_in[31:0]	I	M级pc值
pc_out[31:0]	O	W级pc值
dm_in[31:0]	I	M级dm输出结果
dm_out[31:0]	O	流水到W级dm输出结果
instr_in[31:0]	I	M级指令
instr_out[31:0]	O	W级指令

- **控制**

  1.CTRL

端口名	方向	描述
instr[31:0]	I	指令
rs[4:0]	O	rs寄存器地址
rt[4:0]	O	rt寄存器地址
rd[4:0]	O	rd寄存器地址
imm16[15:0]	O	16位立即数
imm26[25:0]	O	26位立即数
shamt[4:0]	O	移位数
load	O	读取dm类指令
store	O	储存到dm类指令
count_i	O	有立即数的计算类指令
count_r	O	无立即数的计算类指令
branch	O	分支类指令
shifts	O	用shamt的移位指令
shiftv	O	用寄存器值进行移位的指令
j_r	O	用寄存器值进行跳转类指令
j_i	O	用立即数进行跳转类指令
j_l	O	跳转并链接类指令
md	O	乘除指令
mf	O	读lo, hi 寄存器
mt	O	写lo,hi寄存器
type[2:0]	O	CMP选择信号
ext_sel	O	EXT选择信号
gwe	O	GRF写使能信号
gwa_res[4:0]	O	GRF写入地址选择信号
gwd_sel[2:0]	O	GRF写入地址值选择信号
npc_sel[2:0]	O	NPC选择信号
alu_sel[4:0]	O	ALU选择信号
alu_a_sel[1:0]	O	ALU数据a选择·信号
alu_b_sel[2:0]	O	ALU数据b选择信号
be_sel[2:0]	O	BE选择信号

端口名	方向	描述
dp_sel[2:0]	O	DP选择信号

- gwa_res:
  - `count_r:rd`
  - `count_i | load:rt`
  - `jal:5'd31`
- gwd_sel:
  - 3'd0: `alu`
  - 3'd1: `dm`
  - 3'd2: `pc+8`
- alu_a_sel:
  - 2'b0: `rs_d`
  - 2'b1: `rt_d`
- alu_b_sel:
  - 3'd0: `rt_d`
  - 3'd1: `ext_out`
  - 3'd2: `rs_d[4:0]`
  - 3'd3: `shamt`
- 分类

```
assign load = lw | lh | lb | lhu | lbu | 0;

    assign store = sw | sh | sb | 0;

    assign count_r = add | addu | sub | subu | sll | sllv | srl |
                     srlv | sra | srav | And | Or | Xor | Nor |
                         slt | sltu | 0;

    assign count_i = addi | addiu | andi | ori | xori | lui |
                         slti | sltiu | 0;

    assign branch = bltzal | beq | bne | blez | bgtz | bgez | bltz | 0;

    assign shifts = sll | srl | sra | 0; //shift shamt

    assign shiftv = sllv | srlv | srav | 0; //shift $x

    assign j_r = jalr | jr | 0; //jump $x

    assign j_i = jal | j | 0; //jump imm26

    assign j_l = bltzal | jal | jalr | 0; //link

    assign md = mult | multu | div | divu | 0;

    assign mt = mthi | mtlo | 0;

    assign mf = mfhi | mflo | 0;
```

- **冒险**

  1.SU

端口名	方向	描述
D_instr[31:0]	I	D级指令
E_instr[31:0]	I	E级指令
M_instr[31:0]	I	M级指令
md_stall	I	乘除槽繁忙信号
stall	O	阻塞信号

(三)转发暂停控制

- **转发**

  此处我采用的是教程中的转发机制，**即只要下一级的写入地址一样且不为0就转发（W到D级我们使用内部转发）**对于写入地址我做了如下处理

```
assign gwa_res = (count_r)? rd : (count_i | load)? rt :(jal)? 5'd31 : 5'b0;
```

  由于已经对指令进行了分类,我们对于要写入寄存器的指令，我的grf的写入地址就直接是它们要写入的实际地址，而对于不需要写入寄存器我们直接认为它写入的地址为0

```
assign fw_D_rs_d = (D_rs == 0)? 0:
                   (D_rs == E_gwa_res)? fw_E_wd:
                   (D_rs == M_gwa_res)? fw_M_wd:
                   D_rs_d;
```

  因此我们首先判断该级读出地址是否为0，如果为0那么读出的就是0，如果不为0那么对于不写寄存器的指令就会因为其写入地址为0而不转发，因此在此转发机制下不考虑写使能的判断，同时原级也视为一种优先级最低的转发，然后按最近的级数优先级递减，对于还没有产生结果转发了会怎么样呢？当然在之后它产生了结果它还是会被再转发一次，即二次转发**（在此不考虑直到数据必须被使用时还没产生正确结果的情况，因为这个时候就需要暂停阻塞了，即插入nop直到判断在数据必须被使用前产生了结果）**

  那么我们要转发的接收位置一共有以下几种

  - **D级**
    1. **CMP:rs_d, rt_d**
    2. **NPC:rs_d(j_r类指令)**
  - **E级**
    1. **ALU:a ,b**
  - **M级**
    2. **BE:dwd_temp**
  - **W级**
    1. **GRF:rs_d, rt_d(内部转发解决)**

  那么对于具体要转发过来的东西，我做了如下处理：

```
assign fw_E_wd = (E_gwd_sel == `gwd_pc8)? E_pc + 8:
                            32'd0;

    assign fw_M_wd = (M_gwd_sel == `gwd_alu)? M_alu_out:
                            (M_gwd_sel == `gwd_pc8)? M_pc + 8:
                            (M_gwd_sel == `gwd_md)? M_md_out:
                            32'd0;

    assign fw_W_wd = (W_gwd_sel == `gwd_alu)? W_alu_out:
                            (W_gwd_sel == `gwd_dm)? W_dm_out:
                            (W_gwd_sel == `gwd_pc8)? W_pc + 8:
                            (W_gwd_sel == `gwd_md)? W_md_out:
                            32'd0;
```

即有可能被转发的结果接着流水到下一级，再由多路选择器判断该级需要转发回去的是该指令从上一级流水过来的数据还是该级产生的数据**（由于没有遵守必须从寄存器转发的要求，因此需满足从非寄存器输出端口的转发一定转发到该级之前）**，该判断的目的是对要转发的数据进行一下简单的筛选，保证一级只有一个数据被转发，但仍然可能转发错误的数据**(在上面已经说明转发了错误的数据对于结果没有影响)**

在该方案中有个细节十分重要，就是我们参与流水的寄存器的值必须是转发结果，如果直接流水当级寄存器的值，即使正确的寄存器的值在之后仍会被转发过来替代该值，但当正确寄存器的值已经写入寄存器而该寄存器的值却还没到要使用的阶段，那么对于该条指令就不可能得到正确结果，因为已经写入寄存器的值除内部转发外不会再转发比如(假设所有非0号寄存器中的值均不为0)

```
addu $1, $2, $3
nop
sw $1, 0($0)
```

综上，对于该方法转发单元的处理不必单独列出转发表判断何时转发，每次转发的数据，接收到的数据是否正确都不需要考虑了，同时我将转发单元集成到mip.v的顶层模块中更加直观便于修改

- **暂停**

  对于暂停机制我采取了教程中的**AT法**

  首先，定义一下 Tuse 和 Tnew：

  - Tuse：指令进入 **D 级**后，其后的某个功能部件**再**经过多少时钟周期就**必须**要使用寄存器值。对于有两个操作数的指令，其**每个操作数的 Tuse 值可能不等（如 store 型指令 rs、rt 的 Tuse 分别为 1 和 2 ）。**
  - Tnew：位于 **E 级及其后各级**的指令，再经过多少周期就能够产生要写入寄存器的结果。在我们目前的 CPU 中，W 级的指令Tnew 恒为 0；对于同一条指令，Tnew@M = max(Tnew@E - 1, 0)

  具体机制如下：

  当两条指令发生数据冲突**（前面指令的写入寄存器，等于后面指令的读取寄存器）**，我们就可以根据 Tnew 和 Tuse 值来判断策略。

  1. Tnew=0，说明结果已经算出，如果指令处于 WB 级，则可以通过寄存器的内部转发设计解决，不需要任何操作。如果指令不处于 WB 级，则可以通过转发结果来解决。
  2. Tnew<=Tuse，说明需要的数据可以及时算出，可以通过转发结果来解决。
  3. Tnew>Tuse，说明需要的数据不能及时算出，必须暂停流水线解决。

对于Tuse,Tnew,将Tuse分为Tuse_rt,Tuse_rs这样便于通过先前的指令分类分别判断，因为对于同一条指令的两个操作数的Tuse可能不同，如上述的store,Tnew则分为Tnew_E,Tnew_M,**(Tnew_W不需要，因为可以通过内部转发解决)**

对于stall的暂停信号同样分为stall_rs,stall_rt,并且继续做如下细分，stall_rs_E,stall_rs_M.....

那么当Tuse,Tnew全部被计算出来后可以做如下处理(注意乘除槽处理)

```
assign stall_rs_E = (Tuse_rs < Tnew_E)&&(D_rs != 5'd0 && D_rs == E_gwa_res);
    assign stall_rs_M = (Tuse_rs < Tnew_M)&&(D_rs != 5'd0 && D_rs ==
M_gwa_res);
    assign stall_rs = stall_rs_E | stall_rs_M | 0;

    assign stall_rt_E = (Tuse_rt < Tnew_E)&&(D_rt != 5'd0 && D_rt ==
E_gwa_res);
    assign stall_rt_M = (Tuse_rt < Tnew_M)&&(D_rt != 5'd0 && D_rt ==
M_gwa_res);
    assign stall_rt = stall_rt_E | stall_rt_M | 0;

    assign stall_hl = md_stall && (D_mf | D_md | D_mt);

    assign stall = stall_rs | stall_rt | stall_hl;
```

对于Tuse:

- Tuse_rt:

```
assign Tuse_rt = (D_branch)? 3'd0:
                 (D_count_r | D_md)? 3'd1:
                 (D_store)? 3'd2:
                 3'd3;
```

- Tuse_rs:

```
 assign Tuse_rs = (D_branch | D_j_r)? 3'd0 :
     ((D_count_r & !D_shifts) | D_count_i | D_load | D_store | D_mt |
D_md)? 3'd1:
         3'd3;
```

对于Tnew:

- Tnew_E

```
 assign Tnew_E = (E_count_i | E_count_r | E_mf)? 3'd1:
                          (E_load)? 3'd2:
                          3'd0;
```

- Tnew_M

```
 assign Tnew_M = (M_load)? 3'd1:
                           3'd0;
```

一但要暂停，我们直接将指令冻结在D级，即在下一个时钟上升沿到下下个时钟上升沿来临之间该指令仍然在D级，E级被清空，即插入nop指令，即如下：

- 冻结 PC 的值
- 冻结 F/D 级流水线寄存器的值
- 将 D/E 级流水线寄存器清零（这等价于插入了一个 nop 指令）

如此，就完成了暂停机制的构建。

我将阻塞单元放在单独的SU模块中

二、测试方案

自己构造小范围测试测每个冲突与转发，使用github学长写的对拍测试，以下为学长数据强度


```
"forward_valid_ratio": 0.7892376681614349,
   "forward_count": 57,
   "stall_count": 9,
   "forward_coverage": 0.5327102803738317,
   "stall_coverage": 0.6,
   "grade": {
       "forward": {
           "average": 54.82758620689654,
           "warning": [
               "cal_ri <~~ jal",
               "br_r2 <~~ load",
               "br_r2 <~~ jal",
               "load <~~ cal_rr",
               "load <~~ load",
               "load <~~ lui",
               "load <~~ jal",
               "store <~~ jal",
               "jr <~~ cal_ri",
               "jr <~~ load",
               "jr <~~ jal"
           ],
           "details": {
               "cal_rr <~~ cal_rr": 96.66666666666666,
               "cal_rr <~~ cal_ri": 100.0,
               "cal_rr <~~ load": 90.0,
               "cal_rr <~~ lui": 100.0,
               "cal_rr <~~ jal": 66.66666666666667,
               "cal_ri <~~ cal_rr": 86.66666666666666,
               "cal_ri <~~ cal_ri": 100.0,
               "cal_ri <~~ load": 80.0,
               "cal_ri <~~ lui": 100.0,
               "cal_ri <~~ jal": 0,
               "br_r2 <~~ cal_rr": 100.0,
               "br_r2 <~~ cal_ri": 100.0,
               "br_r2 <~~ load": 0,
               "br_r2 <~~ lui": 100.0,
               "br_r2 <~~ jal": 0,
               "load <~~ cal_rr": 0,
               "load <~~ cal_ri": 73.33333333333333,
               "load <~~ load": 0,
               "load <~~ lui": 0,
               "load <~~ jal": 0,
               "store <~~ cal_rr": 80.0,
```

```
                "store <~~ cal_ri": 100.0,
                "store <~~ load": 73.33333333333333,
                "store <~~ lui": 73.33333333333333,
                "store <~~ jal": 0,
                "jr <~~ cal_rr": 70.0,
                "jr <~~ cal_ri": 0,
                "jr <~~ load": 0,
                "jr <~~ jal": 0
            }
        },
        "stall": {
            "average": 58.0,
            "warning": [
                "load <~~ load",
                "store <~~ load",
                "jr <~~ cal_ri",
                "jr <~~ load"
            ],
            "details": {
                "cal_rr <~~ load": 100.0,
                "cal_ri <~~ load": 100.0,
                "br_r2 <~~ cal_rr": 100.0,
                "br_r2 <~~ cal_ri": 100.0,
                "br_r2 <~~ load": 100.0,
                "load <~~ load": 0,
                "store <~~ load": 0,
                "jr <~~ cal_rr": 80.0,
                "jr <~~ cal_ri": 0,
                "jr <~~ load": 0
            }
        }
    },
```


期望输出（结果将与魔改版mars的结果进行对比）：

```
        test1:Accepted!
```

三、思考题

- **为什么需要有单独的乘除法部件而不是整合进ALU？为何需要有独立的HI、LO寄存器？**
  - 乘除法延迟远大于 ALU，整合进 ALU 那么根据木桶原理 CPU 整体周期将大幅增加。增加 HI 和 LO 寄存器可以让乘除法指令和其它指令并行执行，需要结果时再取出即可。
- **参照你对延迟槽的理解，试解释“乘除槽”。**
  - 当乘除法进行或即将开始时，乘除有关指令会被阻塞在 IF/ID 流水线寄存器，相当于处于“乘除槽”。
- **举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint：考虑C语言中字符串的情况）**
  - 当访问类型只占一个字节时，比如 `char`。
- **在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？**

如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证覆盖了所有需要测试的情况；如果你是完全随机生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了特殊的策略，比如构造连续数据冒险序列，请你描述一下你使用的策略如何结合了随机性达到强测的效果。

此思考题请同学们结合自己测试CPU使用的具体手段，按照自己的实际情况进行回答。

- 主要是数据冒险和控制冒险，分别通过暂停转发以及比较前移+延迟槽解决。
- 数据生成器（学长的）采用了特殊策略：单组数据中除了 0 和 31 号寄存器外，至多涉及 3 个寄存器。一方面，这样产生的代码中，邻近的指令几乎全部都存在数据冒险，可以充分测试转发和暂停；另一方面，当测试数据的组数一定多，几乎涉及了每个寄存器，避免了只测试部分寄存器。此外，所有跳转指令都是特殊构造的，不会进入死循环的同时如果跳转出错可以输出中体现。
- 对于一些会产生异常的指令，为防止 MARS 报错，进行了一定的规避。

- **为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？**
  - 主要采用了指令分类的方法，P6 完全沿用了 P5 的分类方法，新增的指令对应的特点都没有脱离这些分类，因此对于每条指令而言，只需译码后将其加入对应的分类，数据通路部分和 P5 完全类似，转发部分完全不用改，暂停部分只需添加一个因乘除块而导致的暂停。