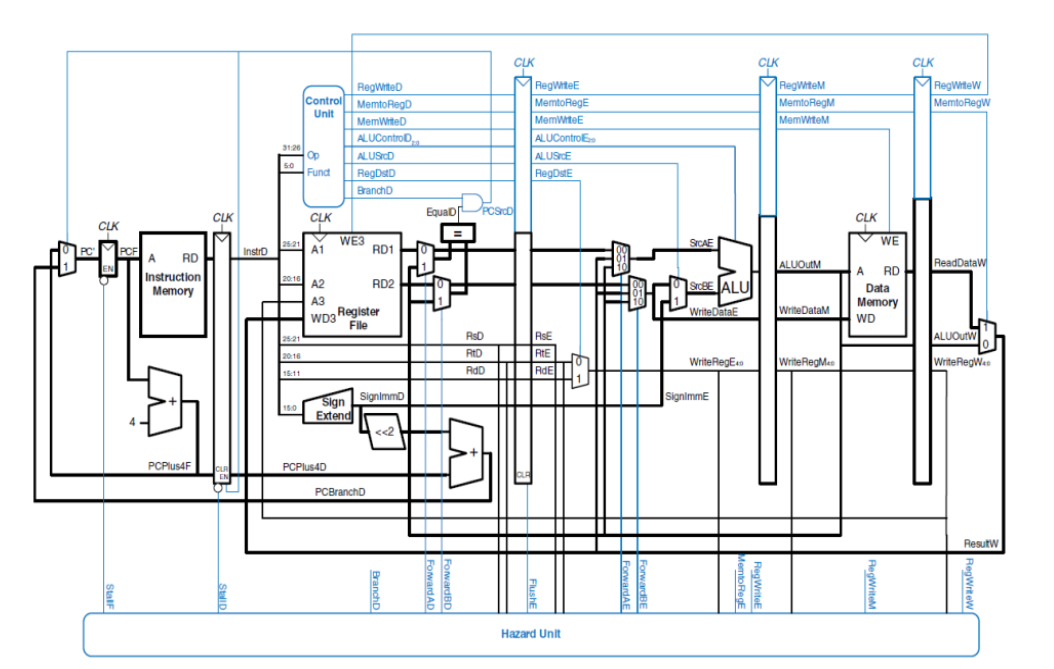
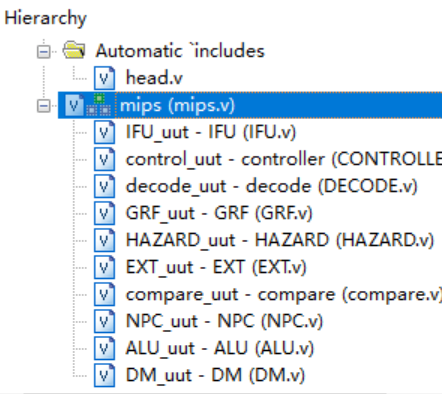
**P6 流水线CPU 实验报告**

**第一部分 CPU模型架构**

1. **流水线CPU顶层架构视图和模块架构视图**





1. **流水线CPU数据通路和控制器**
2. **各指令对应的数据通路表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令 | | addu | subu | ori |
| PC | | pc->IM | pc->IM | pc->IM |
| IM | | IM->INS | IM->INS | IM->INS |
| ADD8(4) | | PC+8->PC8 | PC+8->PC8 | PC+8->PC8 |
| D级 | INS | new | new | new |
| PC8 | new | new | new |
| decode | | rd,rs,rt | rd,rs,rt | rt,rs,imm16 |
| EXT | |  |  | ZE(imm16) |
| NPC | |  |  |  |
| GRF | | RF[rs]->RS\_OUT RF[rt]->RT\_OUT | RF[rs]->RS\_OUT RF[rt]->RT\_OUT | RF[rs]->RS\_OUT RF[rt]->RT\_OUT |
| compare | |  |  |  |
| E级 | WB | new(rd) | new(rd) | new(rt) |
| EXT\_OUT |  |  | new(EXT) |
| RS\_OUT | new(GRF) | new(GRF) | new(GRF) |
| RT\_OUT | new(GRF) | new(GRF) | new(GRF) |
| PC8 | old | old | old |
| ALU | | RS\_OUT+RT\_OUT unsigned | RS\_OUT-RT\_OUT unsigned | RS\_OUT|EXT\_OUT |
| M级 | WB | old | old | old |
| ALU\_result | new(ALU) | new(ALU) | new(ALU) |
| DIN |  |  |  |
| DM | |  |  |  |
| W级 | WrData | old(ALU\_result) | old(ALU\_result) | old(ALU\_result) |
| WB | old | old | old |
| GRF | | WrData->RF[WB] | WrData->RF[WB] | WrData->RF[WB] |

|  |  |  |  |
| --- | --- | --- | --- |
| lui | lw | sw | beq |
| pc->IM | pc->IM | pc->IM | pc->IM |
| IM->INS | IM->INS | IM->INS | IM->INS |
| PC+8->PC8 | PC+8->PC8 | PC+8->PC8 | PC+8->PC8 |
| new | new | new | new |
| new | new | new | new |
| rt,imm16 | rt,rs,imm16 | rt,rs,imm16 | rt,rs,imm16 |
| LUI(imm16) | SE(imm16) | SE(imm16) | SE(imm16) |
|  |  |  | B\_type PC8+SE |
| RF[rs]->RS\_OUT (32'b0,rs=00000) | RF[rs]->RS\_OUT | RF[rs]->RS\_OUT RF[rt]->RT\_OUT |  |
|  |  |  | RF[rs]==RF[rt] |
| new(rt) | new(rt) |  |  |
| new(EXT) | new(EXT) | new(EXT) |  |
| new(GRF) | new(GRF) | new(GRF) |  |
|  |  | new(GRF) |  |
| old | old | old | new(npc) |
| RS\_OUT|EXT\_OUT | RS\_OUT+RT\_OUT unsigned | RS\_OUT+RT\_OUT unsigned |  |
| old | old |  |  |
| new(ALU) | new(ALU) | new(ALU) |  |
|  |  | old(RT\_OUT) |  |
|  | read:DM[ALU\_re] | DIN->DM[ALU\_re] |  |
| old(ALU\_result) | new(DM) |  |  |
| old | old |  |  |
| WrData->RF[WB] | WrData->RF[WB] |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| j | jal | jr | jalr |
| pc->IM | pc->IM | pc->IM | pc->IM |
| IM->INS | IM->INS | IM->INS | IM->INS |
| PC+8->PC8 | PC+8->PC8 | PC+8->PC8 | PC+8->PC8 |
| new | new | new | new |
| new | new | new | new |
| imm26 | imm26 | rs | rd,rs |
|  |  |  |  |
| J\_type PC[31:28]|imm26|00 | J\_type PC[31:28]|imm26|00 | JR\_type RF[rs] | JR\_type RF[rs] |
|  | PC8->RF[31] | RF[rs]->RS\_OUT | RF[rs]->RS\_OUT PC8->RF[rd] |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| new(npc) | new(npc) | new(npc) | new(npc) |

1. **CPU各模块规格**
2. **IFU**
3. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 方向 | 位数 | 端口描述 |
| clk | IN | 1 | 时钟信号 |
| Reset | IN | 1 | 同步复位信号 |
| if\_jump | IN | 1 | pc跳转执行信号 |
| stall\_pc | IN | 1 | pc暂停信号 |
| next\_pc | IN | 32 | pc跳转地址 |
| instr | OUT | 32 | 从IM中读取到的指令 |
| PC | OUT | 32 | PC当前值 |
| PC4 | OUT | 32 | PC+4值 |
| PC8 | OUT | 32 | PC+8值 |

1. 模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 具体描述 |
| 1 | 同步复位 | 复位信号为1时，在时钟上升沿复位pc |
| 2 | pc顺序执行 | pc执行pc+4 |
| 3 | pc跳转执行 | pc跳转至next\_pc地址 |
| 4 | pc暂停执行 | pc保持当前值不变，即暂停 |
| 5 | 读取指令 | 根据pc所指地址从IM读取响应指令并输出 |

1. **GRF**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位数 | 描述 |
| clk | IN | 1 | 时钟信号 |
| WrData | IN | 32 | 回写32位数据 |
| RWAddr | IN | 5 | 回写寄存器地址 |
| rs | IN | 5 | 读寄存器地址1 |
| rt | IN | 5 | 读寄存器地址2 |
| RegWr | IN | 1 | 寄存器堆写使能信号 |
| Reset | IN | 1 | 复位信号 |
| RS\_OUT | OUT | 32 | RS对应输出32位数据 |
| RT\_OUT | OUT | 32 | RT对应输出32位数据 |

（2）模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 同步复位 | 当Reset信号有效时同步复位所有寄存器 |
| 2 | 读寄存器 | 读出rs, rt对应寄存器内存储的数据 |
| 3 | 写寄存器 | 将WrData输入的数据写入RW对应的寄存器 |

1. **ALU**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位数 | 描述 |
| ALUOp | IN | 4 | ALU控制信号，控制进行何种运算 |
| inputA | IN | 32 | ALU的第一个操作数 |
| inputB | IN | 32 | ALU的第二个操作数 |
| shift | IN | 5 | 移位操作移位位数 |
| SHIFTV | IN | 1 | 可变移位选择信号 |
| ALU\_result | OUT | 32 | ALU计算结果 |

（2）模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 加 | 将两个操作数相加 |
| 2 | 减 | 将两个操作数相减 |
| 3 | 或 | 将两个操作数按位或 |
| 4 | 逻辑左移 | 将操作数B逻辑左移shift位 |
| 5 | 逻辑右移 | 将操作数B逻辑左移shift位 |
| 6 | 与 | 将两个操作数按位与 |
| 7 | 或非 | 将两个操作数按位或再取反 |
| 8 | 异或 | 将两个操作数按位异或 |
| 9 | 有符号比较大小 | 对两个操作数进行有符号比较 |
| 10 | 无符号比较大小 | 对两个操作数进行无符号比较 |
| 11 | 算术右移 | 将操作数B算术右移shift位 |

1. **DM**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位数 | 描述 |
| clk | IN | 1 | 时钟信号 |
| DIN | IN | 32 | 写入32位数据 |
| DMAddr | IN | 32 | 写入32位地址 |
| DMWr | IN | 1 | DM写使能信号 |
| Reset | IN | 1 | 同步复位信号 |
| L\_S\_SL | IN | 3 | load/store类型选择信号 |
| DOUT | OUT | 32 | 读出32位数据 |
| LoadSel | OUT | 2 | Load类型地址决定值 |

1. 模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 读内存数据 | DMWr信号无效时，读出DMAddr地址存储的数据 |
| 2 | 写内存数据 | DMWr信号有效时，向DMAddr地址写入DIN数据 |
| 3 | 同步复位 | 当Reset信号有效时，同步复位DM |

1. **EXT**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位数 | 描述 |
| EXT\_Op | IN | 2 | 扩展类型选择信号 |
| imm16 | IN | 16 | 要执行扩展操作的16位立即数 |
| EXT\_OUT | OUT | 32 | 扩展结果 |

1. 模块功能

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 功能描述 |
| 1 | 符号扩展 | EXT\_Op信号为01时，对16位立即数进行符号扩展 |
| 2 | 无符号（零）扩展 | EXT\_Op信号为00时，对16位立即数进行零扩展 |
| 3 | 加载立即数到高位 | EXT\_Op信号为11时，将16位立即数加载到高位 |

1. **CMP**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 方向 | 位数 | 端口描述 |
| RS\_OUT | IN | 32 | RS寄存器值 |
| RT\_OUT | IN | 32 | RT寄存器值 |
| opcode | IN | 6 | 比较指令解码1 |
| rt | IN | 5 | 比较指令解码2 |
| judge | OUT | 1 | 判断结果信号 |

1. 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 具体描述 |
| 1 | 相等时跳转 | Rs和RT相等时判断信号为1 |
| 2 | 不等时跳转 | Rs和RT不等时判断信号为1 |
| 3 | 小于0时跳转 | Rs小于0时判断为1 |
| 4 | 小于等于0时跳转 | Rs小于等于0时判断为1 |
| 5 | 大于0时跳转 | Rs大于0时判断为1 |
| 6 | 大于等于0时跳转 | Rs大于等于0时判断为1 |

1. **NPC**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 方向 | 位数 | 端口描述 |
| judge | IN | 1 | 比较判断信号 |
| NPC\_SL | IN | 3 | NPC跳转模式选择信号 |
| IMM16 | IN | 16 | 16位立即数 |
| IMM26 | IN | 16 | 26位立即数 |
| RS\_OUT | IN | 32 | RS寄存器值 |
| PC\_4 | IN | 32 | PC+4的值 |
| next\_pc | OUT | 32 | 跳转pc地址 |
| if\_jump | OUT | 1 | 跳转判断信号 |

1. 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 具体描述 |
| 1 | B型跳转 | 当比较判断信号为1，NPC\_SL为B型跳转时，相对跳转 |
| 2 | J型跳转 | NPC\_SL为J型时跳转到26位立即数所指地址 |
| 3 | JR跳转 | NPC\_SL为JR型时跳转到RS寄存器所指地址 |

1. **LoadExt**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 方向 | 位数 | 端口描述 |
| mem\_out | IN | 32 | DM读出数据 |
| L\_S\_SL | IN | 3 | Load类型选择信号 |
| DMAddr | IN | 2 | 地址偏移信号 |
| DM\_data | OUT | 32 | 写回数据 |

1. 功能描述

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 具体描述 |
| 1 | 按字读取 | 将DM读出数据整字读出 |
| 2 | 按半字读取 | 将DM读出数据取半字读出 |
| 3 | 按字节读取 | 将DM读出数据取字节读出 |

1. **乘除单元设计**
2. 端口定义

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 方向 | 位数 | 端口描述 |
| clk | IN | 1 | 时钟信号 |
| reset | IN | 1 | 同步复位信号 |
| is\_signed | IN | 1 | 符号判断信号 |
| Other\_Reg\_Wr | IN | 2 | 写乘除寄存器信号 |
| M\_D\_Cal | IN | 2 | 乘除计算信号 |
| inputA | IN | 32 | 第一个操作数 |
| inputB | IN | 32 | 第二个操作数 |
| busy | OUT | 1 | 乘除运算忙碌信号 |
| HI | OUT | 32 | HI寄存器的值 |
| LO | OUT | 32 | LO寄存器的值 |

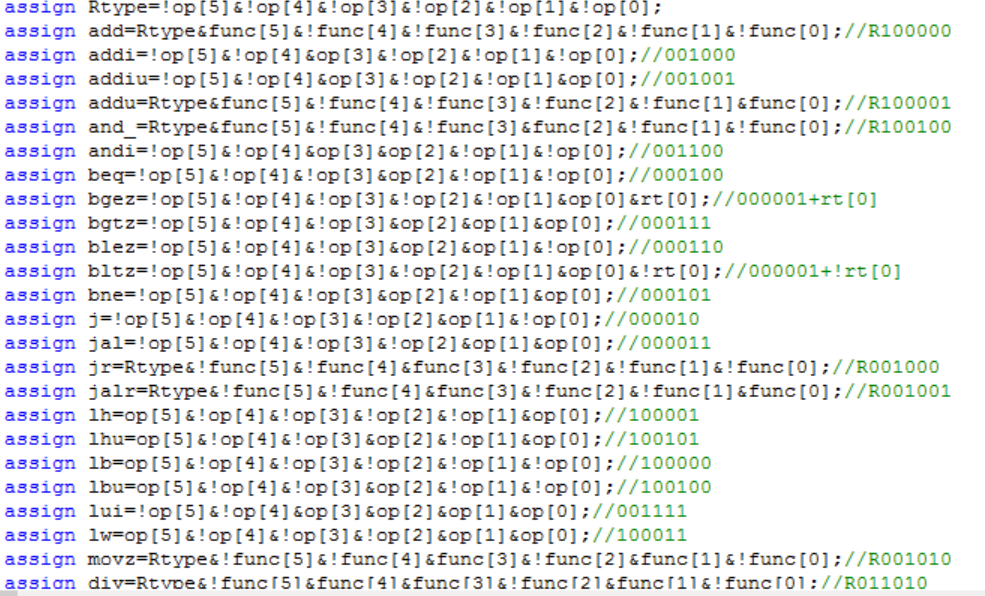
1. 功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 具体描述 |
| 1 | 进行有（无）符号乘除运算 | 根据符号判断和乘除计算信号进行对应的计算 |
| 2 | 写HI和LO寄存器 | 根据写寄存器信号写入对应的寄存器 |

1. **CPU控制器设计**
2. **端口定义和功能描述**

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | 位数 | 描述 |
| opcode | IN | 6 | 指令高六位[31:26]，用于判断指令类型 |
| func | IN | 6 | 指令低六位[5:0]，用于判断R指令 |
| RegDst | OUT | 2 | 写入寄存器选择信号,00选择rd，01选择rt，10选择ra |
| NPC\_SL | OUT | 2 | PC跳转选择信号，00顺序执行，01为B型指令，10为J型指令，11为JR指令 |
| RegData | OUT | 2 | 寄存器写入数据选择信号，00为ALU，01为DM，10为PC\_4 |
| RegWr | OUT | 1 | 寄存器写使能信号 |
| ExtOp | OUT | 2 | 扩展类型选择信号，01是符号扩展，00是零扩展，10是加载立即数到高位 |
| ALUsrc | OUT | 1 | ALU输入端2选择信号，为1选择EXT数据，为0选择rt寄存器数据 |
| DMWr | OUT | 1 | DM模式选择信号，为1向DM写入数据，为0从DM读出数据 |
| ALUOp | OUT | 3 | ALU功能选择信号，ADD:000，SUB:001，OR:010，SLL:011 |
| L\_S\_SL | OUT | 3 | Load/Store类型选择信号 |
| newdatatype | OUT | 2 | 新产生数据来源信号，目前有三种来源ALU,DM,PC |
| rt\_tuse | OUT | 2 | D级指令还有几个周期使用rt寄存器 |
| rs\_tuse | OUT | 2 | D级指令还有几个周期使用rs寄存器 |
| M\_D\_Read | OUT | 1 | HI和LO寄存器读取选择信号 |
| M\_D\_Cal | OUT | 2 | 乘除计算信号 |
| is\_signed | OUT | 1 | 有符号计算信号 |
| SHIFTV | OUT | 1 | 可变移位信号 |

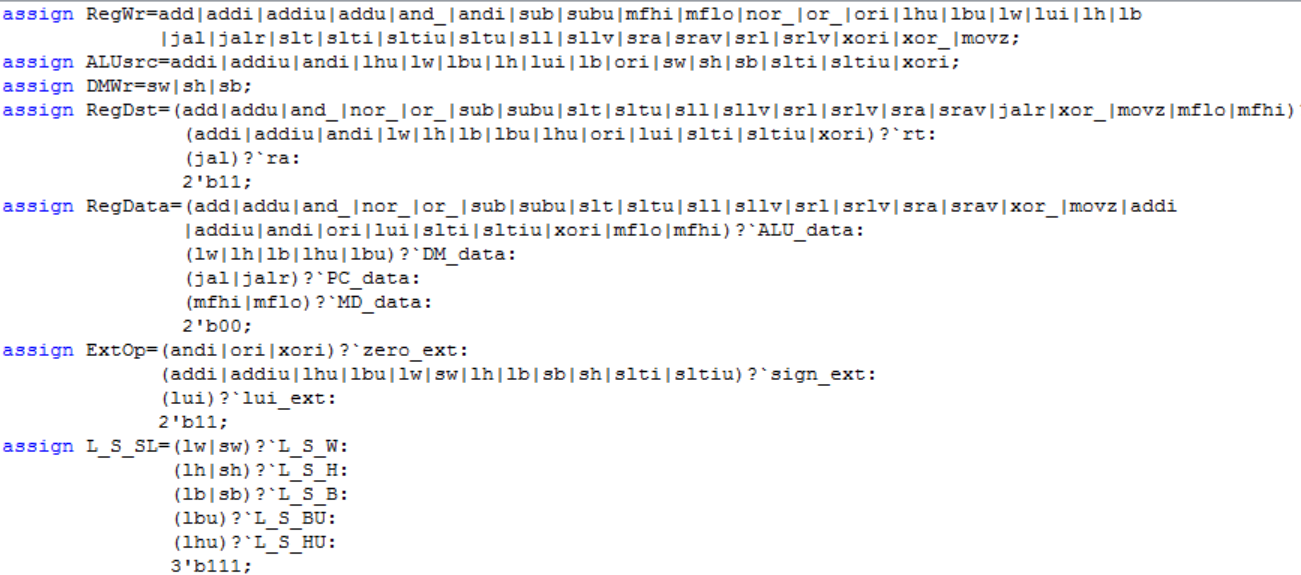
1. **控制器的指令解码（与逻辑AND）**

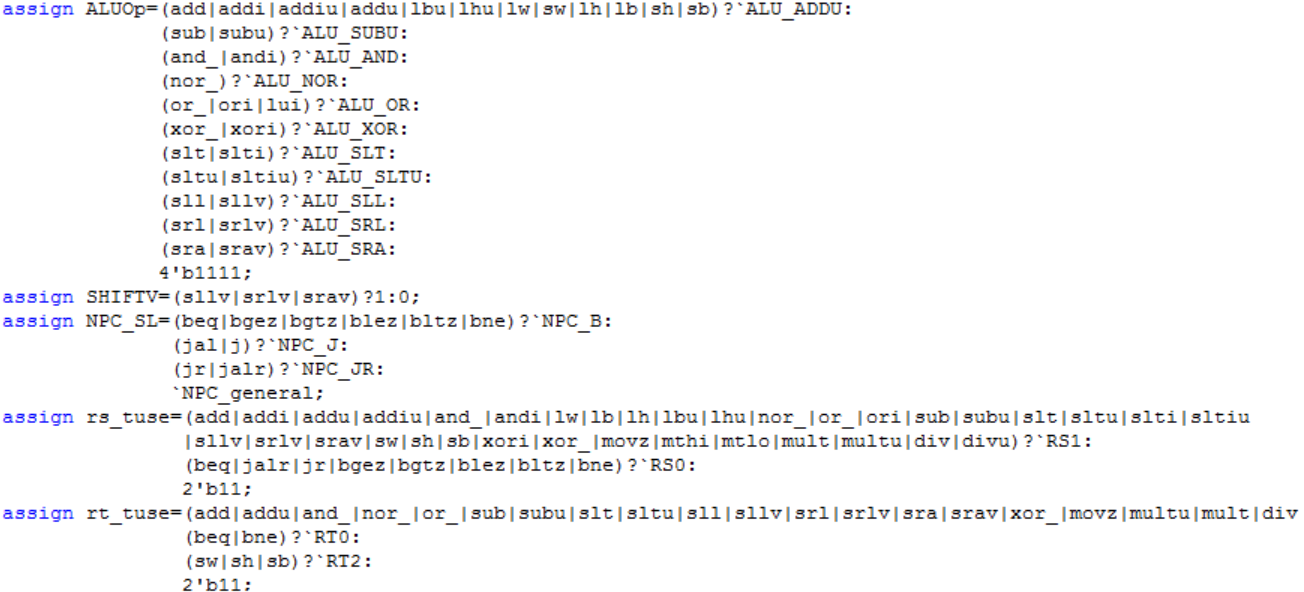


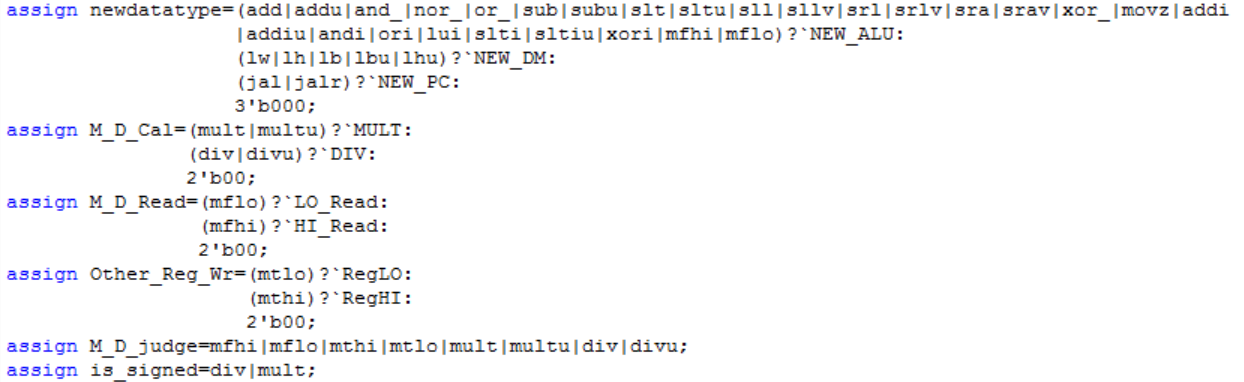
1. **控制器指令与控制信号真值表**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | addu | subu | ori | lw | sw | beq | lui | jal | jr | j |
| RegDst | 00 | 00 | 01 | 01 | X | X | 01 | 10 | X | X |
| NPC\_SL | 00 | 00 | 00 | 00 | 00 | 01 | 00 | 10 | 11 | 10 |
| RegData | 00 | 00 | 00 | 10 | X | X | 00 | 10 | X | X |
| RegWr | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| ExtOp | X | X | 00 | 01 | 01 | X | 00 | X | X | X |
| ALUsrc | 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | X |
| DMWr | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| ALUOp | 000 | 001 | 010 | 000 | 000 | 001 | 010 | X | X | X |

**(4) 控制器的控制信号生成（或逻辑OR）**







1. **冒险单元（HAZARD）**
2. **Tuse-Tnew时间模型**
3. **Tuse**

我们定义某条指令位于流水线D级的时候，再经过Tuse个时钟周期就必须要使用相应的数据。而在D级只有rs和rt两个使用数据的寄存器地址，由此得到指令集的Tuse表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | Tuse | | 功能部件 |
| rs | rt |
| Addu | 1 | 1 | ALU |
| Add | 1 | 1 | ALU |
| Subu | 1 | 1 | ALU |
| Sub | 1 | 1 | ALU |
| And\_ | 1 | 1 | ALU |
| Nor\_ | 1 | 1 | ALU |
| Or\_ | 1 | 1 | ALU |
| Xor\_ | 1 | 1 | ALU |
| Movn | 1 | 1 | ALU |
| Movz | 1 | 1 | ALU |
| Andi | 1 |  | ALU |
| Addiu | 1 |  | ALU |
| Addi | 1 |  | ALU |
| Slti | 1 |  | ALU |
| Sltiu | 1 |  | ALU |
| Ori | 1 |  | ALU |
| xori | 1 |  | ALU |
| Lui |  |  | ALU |
| Lw | 1 |  | DM |
| Lh | 1 |  | DM |
| Lhu | 1 |  | DM |
| Lb | 1 |  | DM |
| Lbu | 1 |  | DM |
| Sw | 1 | 2 | NW |
| Sh | 1 | 2 | NW |
| Sb | 1 | 2 | NW |
| Beq | 0 | 0 | NW |
| Beql | 0 | 0 | NW |
| Bne | 0 | 0 | NW |
| Bgez | 0 |  | NW |
| Bgezal | 0 |  | PC |
| Bgtz | 0 |  | NW |
| Blez | 0 |  | NW |
| Bltz | 0 |  | NW |
| Bltzal | 0 |  | PC |
| J |  |  | NW |
| Jal |  |  | PC |
| Jalr | 0 |  | PC |
| Jr | 0 |  | NW |
| Sll |  | 1 | ALU |
| Sllv | 1 | 1 | ALU |
| Sra |  | 1 | ALU |
| Srav | 1 | 1 | ALU |
| Srl |  | 1 | ALU |
| Srlv | 1 | 1 | ALU |
| Slt | 1 | 1 | ALU |
| Sltu | 1 | 1 | ALU |
| Mult | 1 | 1 | NW |

1. **Tnew**

对于那些能产生新数据的指令，经过分析发现他们产生数据的来源目前有三种：ALU,DM,PC。我们定义对于这些这令在某一流水级还需经过Tnew个时钟周期便可将新数据写入下级流水寄存器，由此得到Tnew表如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令 | 功能部件 | E | M | W |
| addu | ALU | 1 | 0 | 0 |
| subu | ALU | 1 | 0 | 0 |
| ori | ALU | 1 | 0 | 0 |
| lui | ALU | 1 | 0 | 0 |
| sll | ALU | 1 | 0 | 0 |
| slav | ALU | 1 | 0 | 0 |
| lw | DM | 2 | 1 | 0 |
| jal | PC8 | 0 | 0 | 0 |
| jalr | PC8 | 0 | 0 | 0 |

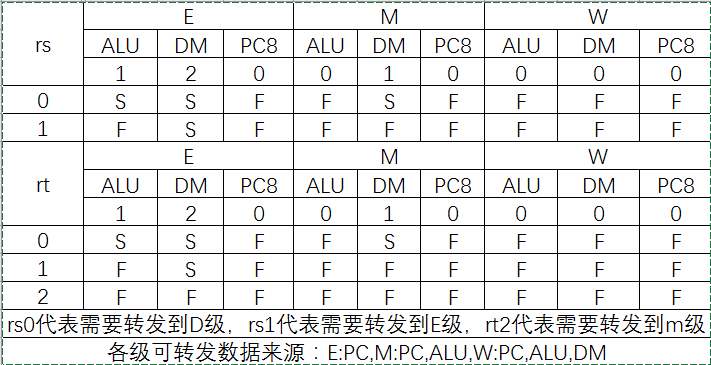
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| E | | | M | | | W | | |
| ALU | DM | PC8 | ALU | DM | PC8 | ALU | DM | PC8 |
| 1 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

1. **暂停和转发策略**

得到Tuse和Tnew表之后，便可以比较两个时间确定暂停和转发策略。

1. 当Tnew<=Tuse时：后续指令执行需要的数据已经在数据通路中产生，可以全部采用显式转发策略进行解决。
2. 当Tnew>Tuse时：后续指令执行需要的数据未在数据通路中产生，需要暂停流水线cpu运行，等待满足Tnew<=Tuse时再通过相应的转发进行解决。

根据上述原则得到策略矩阵如下：



使用数据：

见上文Tuse图

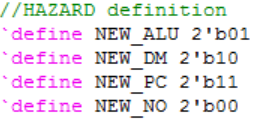
产生数据：

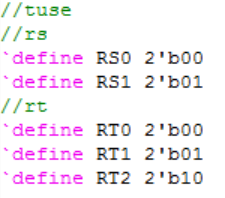
在E级流水线寄存器可转发的数据:PC

在M级流水线寄存器可转发的数据:PC,ALU

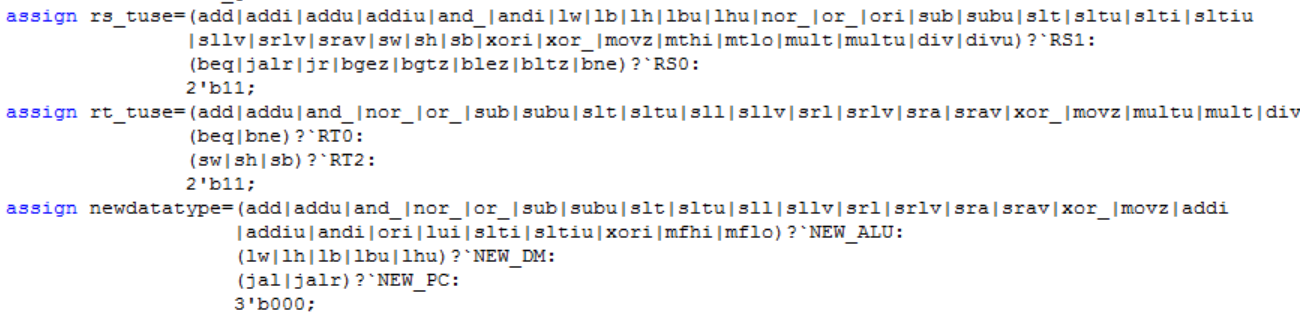
在E级流水线寄存器可转发的数据:PC,ALU,DM

对应的Verilog宏定义如下：





对应的控制器实现如下：

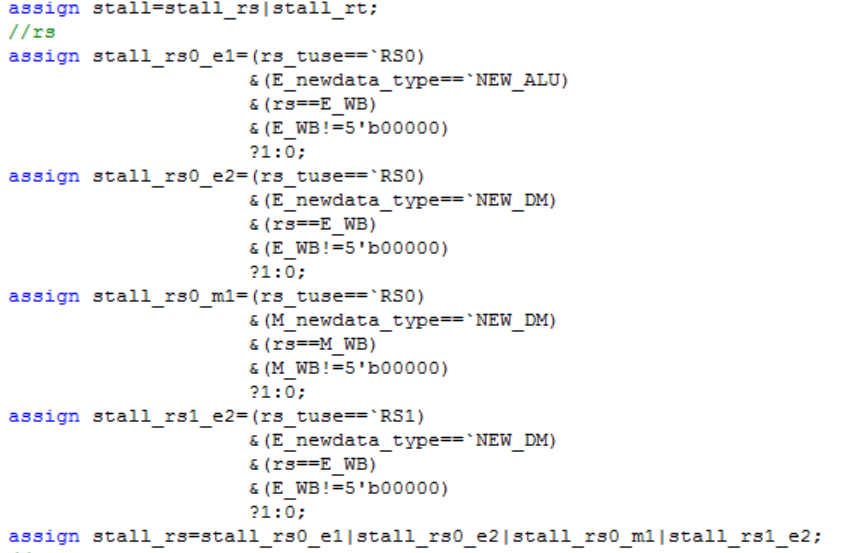


1. **暂停功能具体实现**
2. 暂停策略
3. 使pc暂停，不让其值改变；
4. 使D级流水线暂停，不让其值改变；
5. 清零E级流水线，相当于插入nop指令
6. 暂停情况实现

rs和rt都是相同的四种暂停情况，故此处以rs为例。

根据策略矩阵中S情况对应的Tuse和Tnew，构造如图所示的暂停条件：

1. 满足此时对应的Tuse和Tnew条件
2. 当前指令使用数据的寄存器和前级产生新数据要写入的寄存器相同
3. 为了避免0号寄存器带来的误转发，还要保证产生数据的寄存器不是0号寄存器。

当暂停条件满足时，HAZARD单元使stall\_pc,stall\_D,reset\_E三个信号为1，执行相应操作完成暂停。

1. **转发功能具体实现**
2. 转发条件：

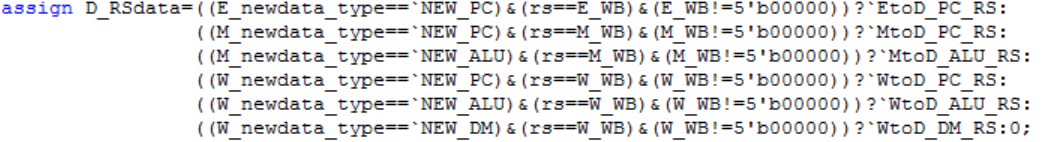
i.下一级寄存器确实产生了新数据；

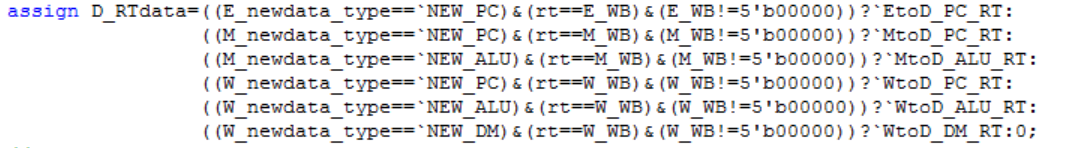
ii.当前指令使用数据的寄存器和前级产生新数据要写入的寄存器相同.

iii.为了避免0号寄存器带来的误转发，还要保证产生数据的寄存器不是0号寄存器。

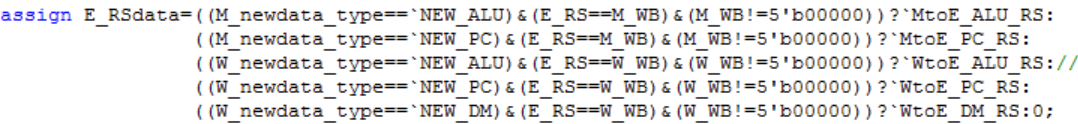
(2) 各级的转发情况

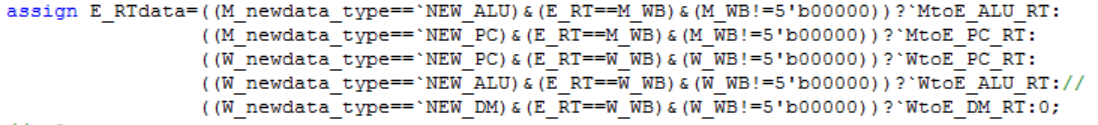
D级：rs和rt位置都需要转发，D级可以接受来自E,M,W三个级的转发数据，E级的转发来源PC,M级的转发来源PC,ALU，W级的转发来源PC,ALU,DM，故总共有可能的六个转发条件，Verilog实现如下：



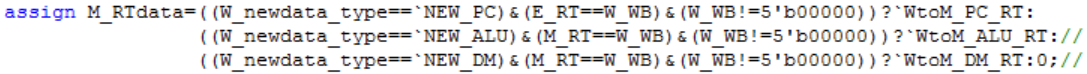


E级：rs和rt位置都需要转发，E级可以接受来自M,W两个级的转发数据，M级的转发来源PC,ALU，W级的转发来源PC,ALU,DM，故总共有可能的五个转发条件，Verilog实现如下：





M级：目前指令集只有rt位置需要转发，M级可以接受来自W级的转发数据， W级的转发来源PC,ALU,DM，故总共有可能的三个转发条件，Verilog实现如下：



**第二部分 测试验证**

1. **基础指令和基础模块测试**
2. **测试目标**
   1. 指令集中四十条指令都能做到完全正确执行。
   2. 各个数据通路模块的运行和读出写入操作都完全正确。
3. **测试程序及期望**

测试程序1：程序源代码，有关注释：

ori $1 $0 0x1234

lui $2 0x9898

addu $4 $1 $2

sw $1 0($0)

lw $3 0($0)

sw $4 0($0)

lh $5 2($0)

lb $6 0($0)

lb $7 1($0)

lb $8 3($0)

lh $20 0($0)

lh $21 2($0)

sll $9 $8 3

slt $10 $1 $4

ori $11 $0 0x9876

lui $12 0x2345

addu $12 $11 $12

sb $12 1($0)

lw $12 0($0)

sb $13 2($0)

lw $13 0($0)

sb $14 3($0)

lw $14 0($0)

sh $15 0($0)

sh $16 2($0)

ori $1 $0 3

srav $17 $11 $1

ori $22 $0 0x0001 #测试22-31号寄存器并测试beq指令

ori $27 $0 0x0003

ori $28 $0 0x1234

lui $29 0x5678

addu $22 $28 $29

sw $22 0($0)

lb $21 1($0)

sh $22 4($0)

beq $8 $0 if\_1\_else #beq测试，进行跳转

nop

if\_1:

ori $30 $0 0x6666

if\_1\_else:

ori $30 $0 0x1111

ori $31 $0 0x3333

beq $29 $0 if\_2\_else #beq测试，不进行跳转，顺序执行

nop

if\_2:

ori $11 $0 0x2222

if\_2\_else:

jal func # jal 测试

nop

j end # j 测试，同时跳到末尾结束程

nop

func:

ori $12 $0 0x0022

jr $ra # jr 测试

nop

end:

lui $1 0x9889

blez $1 if\_4\_else

nop

if\_4:

lui $2 0x1122

j end1

nop

if\_4\_else:

lui $2 0x9982

end1:

测试程序2：程序源代码

addiu $1 $0 0x7654

addi $2 $0 0x1234

lui $2 0x1234

add $3 $1 $2

sub $4 $1 $2

and $5 $4 $3

andi $6 $5 0x3af7

sw $4 0($0)

sw $6 4($0)

lw $1 0($0)

lhu $1 0($0)

lhu $1 2($0)

lbu $1 0($0)

lbu $1 1($0)

lbu $1 2($0)

lbu $1 3($0)

lui $2 0xf67f

ori $1 $0 0xee7e

addu $3 $2 $1

sh $3 6($0)

sh $3 4($0)

sb $3 4($0)

sb $3 5($0)

sb $3 6($0)

sb $3 7($0)

sw $3 0($0)

lh $1 0($0)

lh $1 2($0)

lb $1 0($0)

lb $1 1($0)

lb $1 2($0)

lb $1 3($0)

xor $7 $5 $6

xori $8 $7 0x4396

or $9 $8 $6

nor $9 $8 $6

sll $10 $9 5

ori $1 $0 3

sllv $11 $9 $1

sra $12 $11 7

sll $12 $12 9

srav $12 $12 $1

srl $13 $9 4

srlv $13 $9 $1

bne $13 $0 if\_1\_else

nop

if\_1:

lui $1 0xffff

if\_1\_else:

lui $2 0x7567

blez $1 if\_2\_else

nop

if\_2:

lui $3 0xea44

if\_2\_else:

lui $4 0xf233

bgez $20 if\_3\_else

nop

if\_3:

lui $5 0x1234

if\_3\_else:

lui $5 0xb632

bgtz $5 if\_4\_else

nop

nop

if\_4:

lui $1 0

j next

nop

if\_4\_else:

lui $1 3

next:

bltz $1 if\_5\_else

nop

if\_5:

ori $10 $0 0x3110

jalr $20 $10

nop

j end1

nop

if\_5\_else:

ori $11 $0 0x3118

jalr $20 $11

nop

j end1

func1:

jr $20

nop

func2:

sll $31 $20 11

jr $20

end1:

lui $1 0xff23

lui $2 0x6732

slt $3 $1 $0

slti $4 $3 -6

sltu $5 $1 $2

sltiu $6 $1 6

andi $1 $0 0

andi $2 $0 0

ori $1 $0 -5

ori $2 $0 3

mult $1 $2

mflo $3

mfhi $4

multu $1 $2

mflo $3

mfhi $4

div $1 $2

mflo $3

mfhi $4

divu $1 $2

mflo $3

mfhi $4

**数据冒险测试**

1. **测试目标**

测试四种暂停情况和五大类转发情况，流水线CPU能够正确运行，并且尽可能多的采用转发策略解决数据冒险。

1. **测试程序及期望**

符号说明，Tuse=0,1,2,分别代表当前D级指令要在D,E,M级使用数据。Tnew，e1，与当前D级指令冲突指令在E级，它将在1个周期后把新数据写入E的下一级，即M级寄存器。

暂停情况：

1. Tuse=0,Tnew=e1

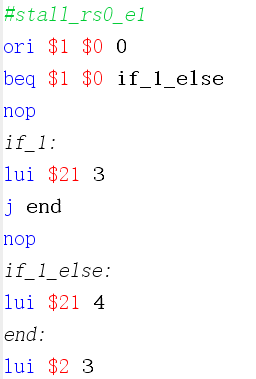
对于rs,此时使用rs的指令有beq,jr；

对于rt，此时使用rt的指令有beq。

满足e1的指令addu,subu,ori,lui

测试样例如下：

（其他指令本身行为没有问题则暂停也没有问题）



1. Tuse=0,Tnew=e2

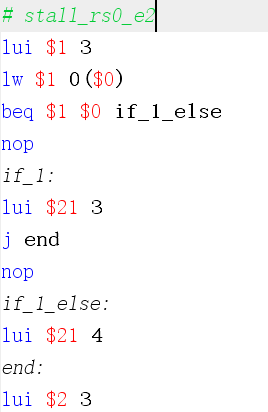
对于rs,此时使用rs的指令有beq,jr；

对于rt，此时使用rt的指令有beq。

满足e2的指令lw

测试样例如下：

（其他指令本身行为没有问题则暂停也没有问题）



1. Tuse=0,Tnew=m1

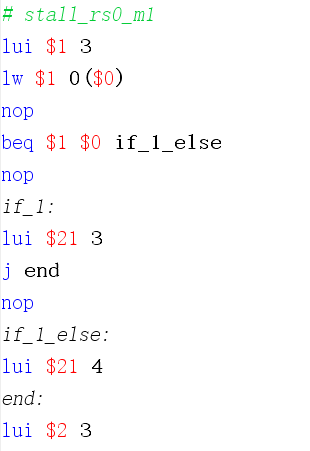
对于rs,此时使用rs的指令有beq,jr；

对于rt，此时使用rt的指令有beq。

满足m1的指令lw

测试样例如下：

（其他指令本身行为没有问题则暂停也没有问题）



1. Tuse=1,Tnew=e2

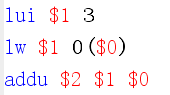
对于rs,此时使用rs的指令有addu,subu,ori,sw,lw；

对于rt，此时使用rt的指令有addu,subu。

满足e2的指令lw

测试样例如下：

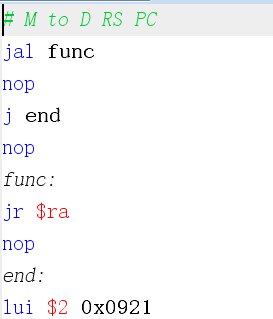
（其他指令本身行为没有问题则暂停也没有问题）



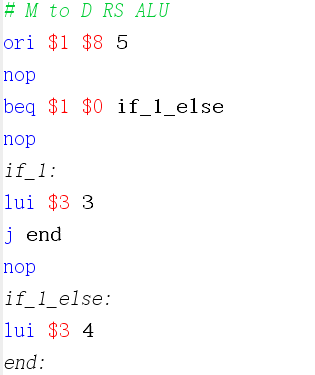
转发情况：

在指令本身行为正确的前提下，仅测试一类情况中某一样例便可验证该转发情况的正确性。

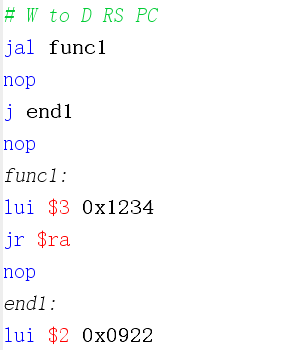
1. 向D级rs转发：
2. E级流水线寄存器转发。由于延迟槽的缘故，在E级产生新数据的jal指令后跟不会发生冲突的延迟槽指令，故这种情况在正常指令中不存在。
3. M级流水线寄存器转发，新数据来源为PC。



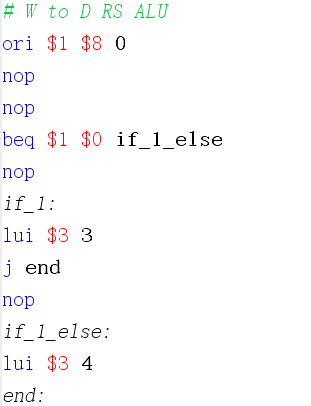
1. M级流水线寄存器转发，新数据来源为ALU



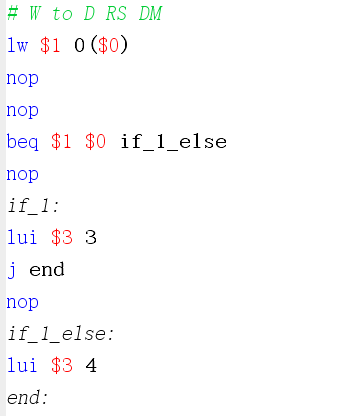
1. W级流水线寄存器转发，新数据来源为PC



1. W级流水线寄存器转发，新数据来源为ALU



1. W级流水线寄存器转发，新数据来源为DM



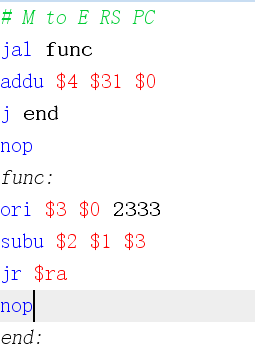
2) 向D级RT转发，类似RS情况。

3） 向E级RS转发

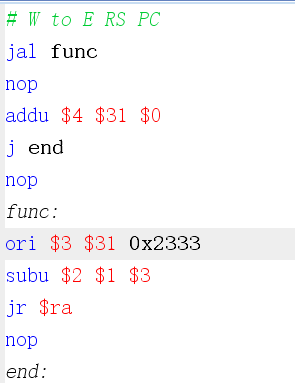
1. M级流水线寄存器转发，新数据来源为ALU



1. M级流水线寄存器转发，新数据来源为PC



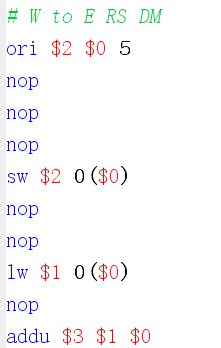
1. W级流水线寄存器转发，新数据来源为PC



1. W级流水线寄存器转发，新数据来源为ALU



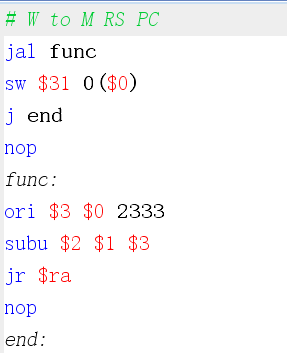
1. W级流水线寄存器转发，新数据来源为DM



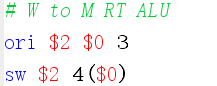
4） 向E级RT转发，类似RS情况

5） 向M级RT转发

1. W级流水线寄存器转发，新数据来源为PC



1. W级流水线寄存器转发，新数据来源为ALU



1. W级流水线寄存器转发，新数据来源为PC



6）乘除单元暂停及转发测试

lui $3 0xffff

addu $2 $3 $2

multu $1 $2

mflo $11

blez $11 if\_1\_else

nop

if\_1:

ori $3 $0 5

sllv $12 $11 $3

j end

nop

if\_1\_else:

ori $3 $0 3

srav $12 $11 $3

end:

sub $13 $2 $12

# W to D RS MD

andi $1 $0 0

andi $2 $0 0

ori $1 $0 9

ori $2 $0 0xfffb

lui $3 0xffff

addu $2 $3 $2

multu $1 $2

mflo $11

nop

blez $11 if\_2\_else

nop

if\_2:

ori $3 $0 5

sllv $12 $11 $3

j end1

nop

if\_2\_else:

ori $3 $0 3

srav $12 $11 $3

end1:

sub $13 $2 $12

# M to E RS MD

andi $1 $0 0

andi $2 $0 0

ori $1 $0 9

ori $2 $0 0xfffb

lui $3 0xffff

addu $2 $3 $2

multu $1 $2

mflo $11

multu $11 $2

mflo $11

mthi $11

mfhi $11

mtlo $11

# W to E RS MD

andi $1 $0 0

andi $2 $0 0

ori $1 $0 9

ori $2 $0 0xfffb

lui $3 0xffff

addu $2 $3 $2

multu $1 $2

mflo $11

nop

multu $11 $2

mflo $11

nop

mthi $11

mfhi $11

nop

mtlo $11

# W to M RT MD

andi $1 $0 0

andi $2 $0 0

ori $1 $0 9

ori $2 $0 0xfffb

lui $3 0xffff

addu $2 $3 $2

multu $1 $2

mflo $11

sw $11 0($0)

multu $11 $2

mfhi $11

sw $11 4($0)

1. **综合程序测试**
2. **测试目的：**

综合测试指令集和数据冒险控制单元

1. **测试程序：**

lui $t0,0x0fff #构造数据

ori $t0,$t0,123

lui $t1,,0xffff

ori $t1,$t1,0xffff

lui $t2,0x0000

ori $t2,$t2,123

lui $t3,0xffff

ori $t3,$t3,0

lui $t4,0x2342

ori $t4,$t4,0x9824

lui $t5,0x0000

ori $t5,$t5,1

lui $t6,0x0fab

ori $t6,$t6,567

lui $t7,0xfedc

ori $t7,$t7,0xab

nop

nop

nop #排空流水线

sw $t0,0($0) #存入数据

sw $t1,4($0)

sw $t2,8($0)

sw $t3,12($0)

sw $t4,16($0)

sw $t5,20($0)

sw $t6,24($0)

sw $t7,28($0)

nop

nop

nop #排空流水线

addu $t5,$t5,$t5 #运算类R1前序指令

subu $s0,$t5,$0 #运算类R1 E-M

subu $s0,$t5,$t5 #运算类R1 ID-M

subu $s0,$0,$t5 #运算类R1 ID-W

sll $t5,$t0,5 #运算类R2前序指令

subu $s1,$t5,$t3 #运算类R1 E-M

subu $s1,$t5,$t1 #运算类R1 ID-M

subu $s1,$0,$t5 #运算类R1 ID-W

ori $t5,$t6,100 #运算类I1前序指令

srav $s2,$t5,$t5 #运算类R1 E-M

srav $s2,$t5,$0 #运算类R1 ID-M

srav $s2,$s2,$t5 #运算类R1 ID-W

lui $t5,0xffff #运算类I2前序指令

slt $s3,$0,$t5 #运算类R1 E-M

slt $s3,$t5,$0 #运算类R1 ID-M

slt $s3,$t5,$t5 #运算类R1 ID-W

jal loop1 #j类前序指令

addu $0,$0,$t1 #延迟槽

lw $s6,8($0) #load类前序指令

xori $s6,$t5,000 #检验无效的rt域是否会发生多余暂停

ori $s6,$0,1 #构造正数

lw $t5,4($0) #load类前序指令

xori $s7,$t5,123 #运算类I1 ID-E

xori $s7,$t1,456 #运算类I1 ID-M

xori $s7,$t4,010 #运算类I1 ID-W

lw $s0,4($0) #load类前序指令

lui $s0,0xffff #检验无效的rs，rt域是否会发生多余的暂停

j loop2

addu $t5,$0,$0 #延迟槽

loop1:

addu $s4,$0,$ra #运算类R1 E-M

subu $s4,$0,$ra #运算类R1 ID-M

addu $s4,$s4,$ra #运算类R1 ID-W

lw $t5,0($0) #load类前序指令

addu $s5,$t5,$t1 #运算类R1 ID-E

addu $s5,$t1,$t5 #运算类R1 ID-M

addu $s5,$t5,$t5 #运算类R1 ID-W

jr $ra

nop #延迟槽

loop2:

lw $s0,16($0) #load类前序指令

sw $s0,32($0) #store类ID-E

sw $s0,36($0) #store类ID-M

lw $s0,12($0) #load类前序指令

lw $s0,8($0) #load类前序指令

addu $s0,$0,$t6 #运算类R1前序指令

sw $s0,40($0) #检验在有多种新值的情况下，转发是否完全且优先级正确

jal loop3 #j类前序指令

nop

sw $ra,52($0) #若跳转正确，该条指令不会执行

sw $ra,56($0) #若跳转正确，该条指令执行

ori $s0,$0,0x3184 #装入Loop4地址

jalr $ra,$s0 #j类ID-E

nop

addu $t1,$t1,$s3 #运算类R1前序指令

beq $t1,$t2,loop5 #B类ID-E

nop

addu $t1,$0,$s1 #运算类R1前序指令

addu $0,$s0,$s2

beq $t1,$t2,loop5 #B类ID-M

nop

beq $0,$0,loop5 #分支执行

sll $s5,$s4,5 #延迟槽

loop3:

sw $ra,44($0) #store类ID-M

sw $ra,48($0) #store类ID-W

ori $t0,$0,4

addu $ra,$ra,$t0 #改变跳转地址

jr $ra #j类ID-E

nop

loop4:

jr $ra #j类ID-M

nop

loop5: #循环

ori $t0,$0,0

loop6:

ori $t1,$0,5

ori $t2,$0,1

addu $t0,$t0,$t2

beq $t0,$t1,loop7

sw $t0,60($0)

j loop6

nop

loop7:

lw $t1,32($0) #load类前序指令

bgez $t1,loop8 #B类ID-E

nop

addu $t0,$t0,$0

loop8:

addu $t0,$0,$s5

sw $t0,64($0)

nop

nop

j loop10

loop9:

mult $ra,$ra

mfhi $t0

jr $ra

nop

loop10:

lui $t0,0x0123

ori $t0,$t0,0x4567

addi $t0,$0,0xffff

add $t0,$t0,$t0 #运算类R1前序指令

add $t0,$t0,$t0

add $t0,$t0,$t0

multu $t0,$t1 #md类指令

nop

nop

nop

sll $t0,$t0,4 #运算类R2前序指令

srl $t0,$t0,4

sra $t0,$t0,4

divu $t1,$t0 #md类指令

nop

addi $t0,$t0,0xffff #运算类I1前序指令

addi $t0,$t0,0xffff

addi $t0,$t0,0xffff

mthi $t0 #md类指令

mult $t1,$t0 #md类指令

nop

nop

lw $t2,32($0) #load类前序指令

lw $t3,36($0)

mtlo $t2 #md类指令

mult $t2,$t3 #md类指令

nop

jal loop9

nop

div $t0,$t1

**第三部分 思考题**

1. 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？

答：由于乘除法运算在实际物理实现中的延迟非常大，远远大于其他计算指令，有了单独的乘除法部件可以让无数据相关的指令在乘除法计算的延迟时间内通过ALU并行执行，这样做法大幅节省了时间。

独立的HI，LO寄存器是因为：在乘除法运算完成时，ALU并行执行的指令也可能同时完成，这样会同时产生两个值需要回写， 造成冲突，而如果强行修改GRF的写入则会带来更多麻烦。一是乘除法结果两个32位数据要和ALU结果同时回写增大了开销，整个流水级都需要改造。二是在判断转发时这一周期有三个供给数据，转发判断非常麻烦。而使用独立的HI,LO寄存器则很好的避免了开销和冲突。

1. 参照你对延迟槽的理解，试解释“乘除槽”。

答： 类比延迟槽，这其实就是在某一指令需要较大的延迟时间来执行的时候，通过所谓延迟槽，乘除槽来并行执行一些与当前指令没有数据相关的指令，这样做就可以很好地利用时间，减少阻塞。

1. 为何上文文末提到的lb等指令使用的数据扩展模块应在 MEM/WB 之后，而不能在 DM 之后?

答：DM单元延迟最大，如果在真正的实际情况中DM和内存交换数据延迟远远大于流水线周期。因此如果放置在 DM之后阶段，则 Mem阶段的延迟将进一步增加，CPU时钟频率会进一步减小。而放置在 WB 阶段，则由于GRF延迟小，因此整个WB阶段的延迟仍然小于放置在 DM之后Mem阶段的延迟，这可以使得时钟频率不会降低。

1. 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）

答：对C语言中的字符串进行读写时，每一个字符都是char型数据，占一个字节大小，所以每次操作都是进行在字节单位上的，而按字节访问内存所需时间显然小于按字访问，所以进行同样的字符串读写操作按字节访问就会在性能上更有优势。

1. 如何概括你所设计的CPU的设计风格？为了对抗复杂性你采取了哪些抽象和规范手段？

答：我的CPU设计风格类似于“Planner”型。在设计早期通过对于指令的行为分析进行分类，将数十条的指令根据一定的性质分为几大类。对于指令本身的行为功能，依据“高内距，低耦合”的思想将功能相似的指令集成在一个个单独的模块中，再通过统一的主控制器进行整合。对于冒险处理，在目前的指令集下，需求者根据流水级和寄存器的不同可分为以下五类need\_rs\_D, need\_rt\_D, need\_rs\_E ,need\_rt\_E, need\_rt\_M。对于供给者，每一级新增一个数据来源。因为一级某一时刻只能执行一条指令，例如ALU和乘除模块虽然都在E级但是一次只有一个有效，故增加多选器之后满足每一级新增一个数据来源。在此基础上，D级产生的PC8，E级产生的ALU/乘除，M级产生的DM，前级的产生数据可以传到后级，共计六个供给者。而在构造转发的时候，只需看当前级的后级有哪些，然后对应转发就好，例如E级的转发，其后级有M级的ALU和PC8，W级的ALU,PC8,DM，一共五个转发，然后按照优先级进行编码即可。只要保证转发的设计完整和指令行为的正确，也就无需写大规模的指令覆盖性测试集。在整个设计过程中，通过表格来将整个设计方案显式表达，非常清晰且便于之后修改。

1. 你对流水线CPU设计风格有何见解？

答：特定的方法针对特定的实际设计需要，没有最好的风格，只有最适合的风格。对于较小的指令集，较简单的数据通路，较短的流水线，采用“Detector”确实更加轻便快捷。但是在工程中倘若面大非常复杂的系统设计，设计和实现两个阶段应该分开进行，前期的设计准备工作非常重要，对于设计方案的显式表达非常必要，对于基础指令的分析分类是减小复杂度增加条理性的关键。暴力转发在目前的架构下没有任何问题，因为多余的转发不会影响正确数据的使用，但是如果考虑实际成本等因素，我们应该尽可能精简设计以节省开销提高性价比。而且如果是其他的流水线类型或许就会发生问题。

1. 在本实验中你遇到了哪些不同指令组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？请有条理的罗列出来。(**非常重要**)

答：见上文数据冒险测试部分。