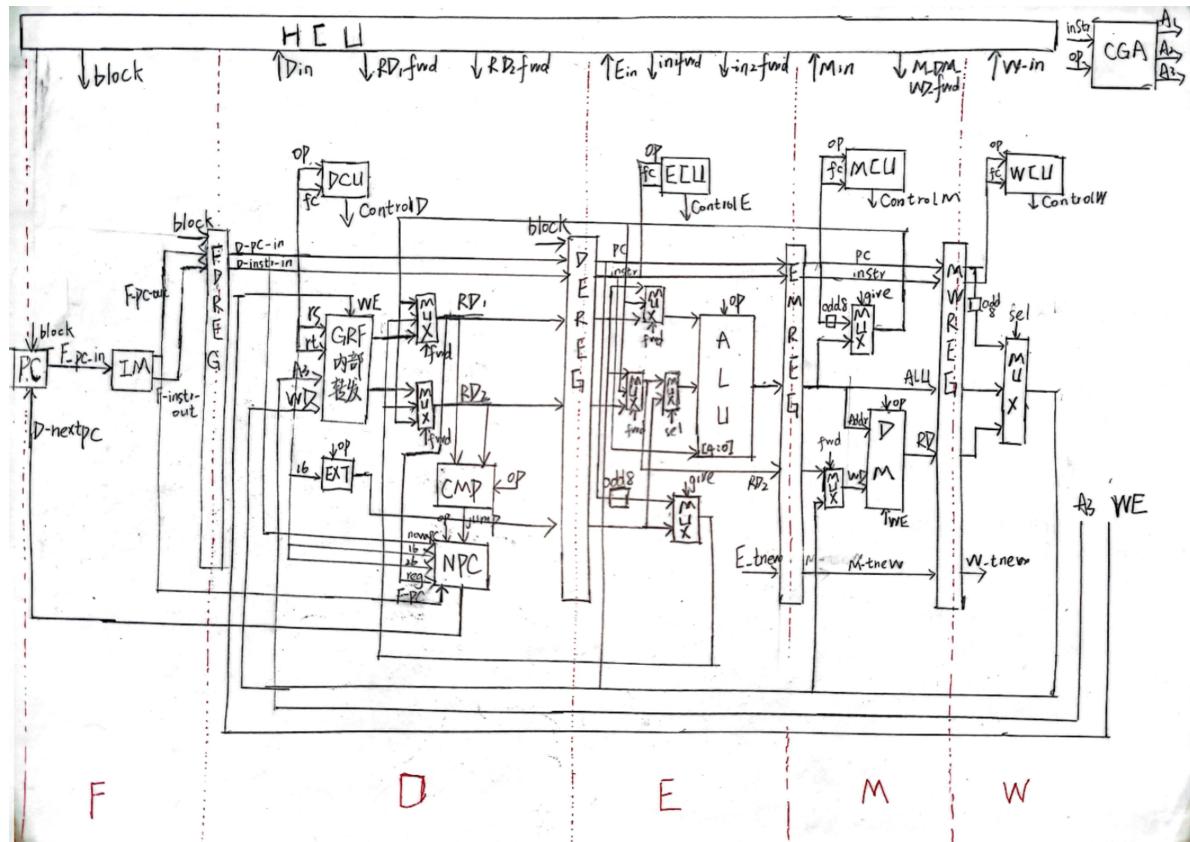


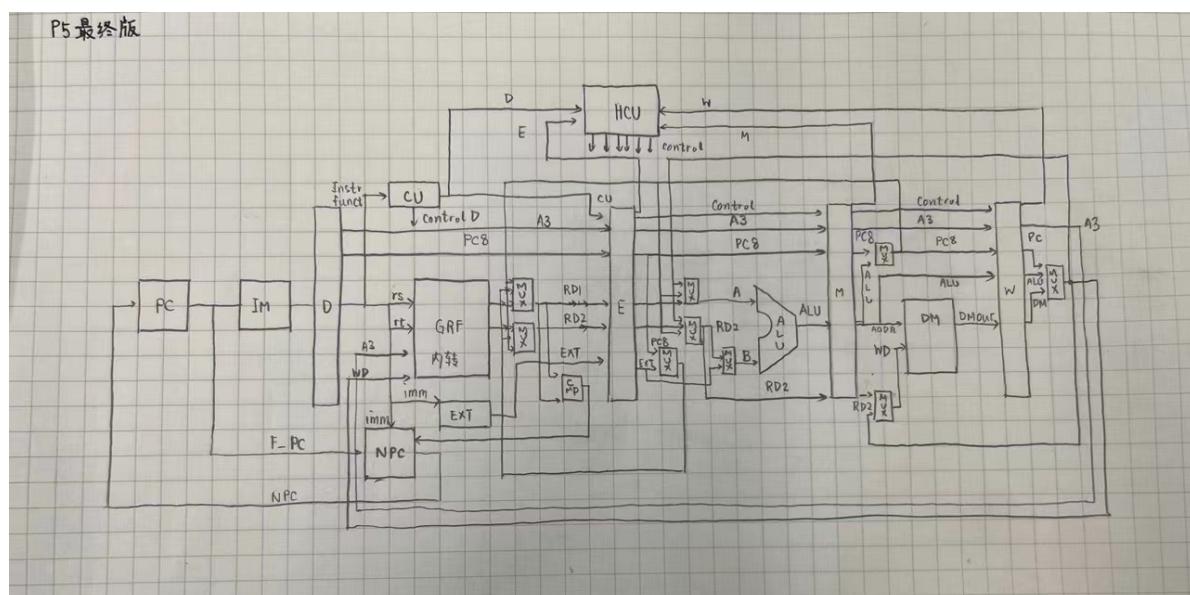
设计草稿

顶层电路

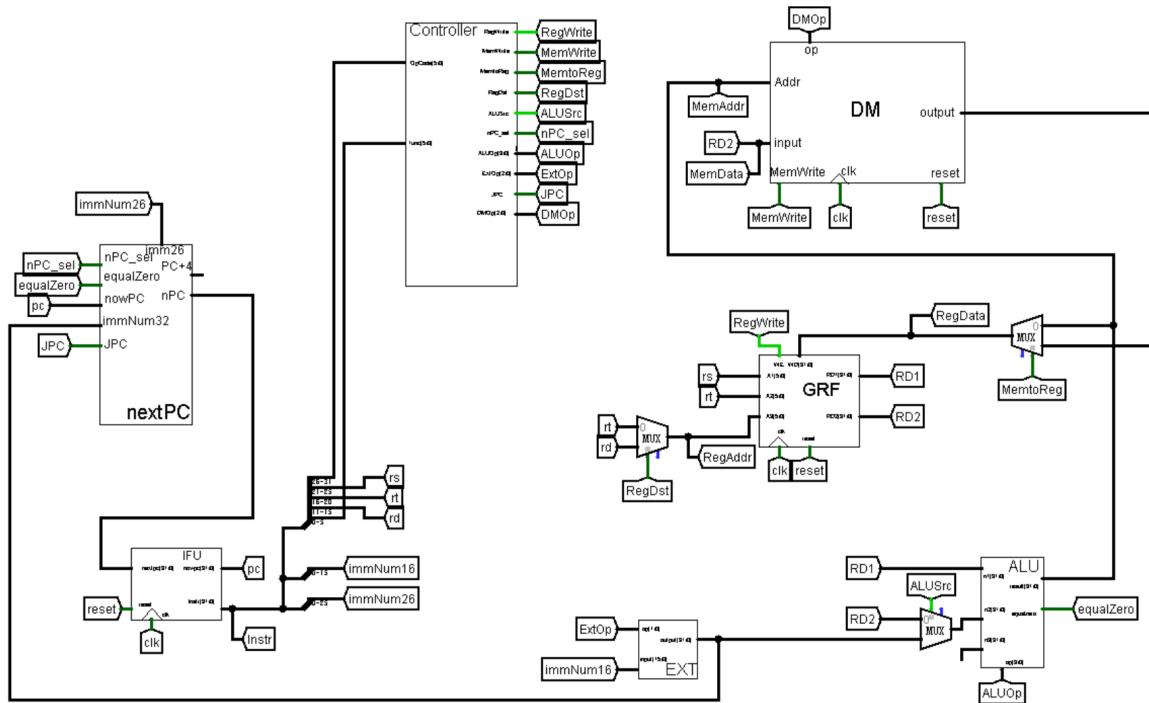
扫描版本



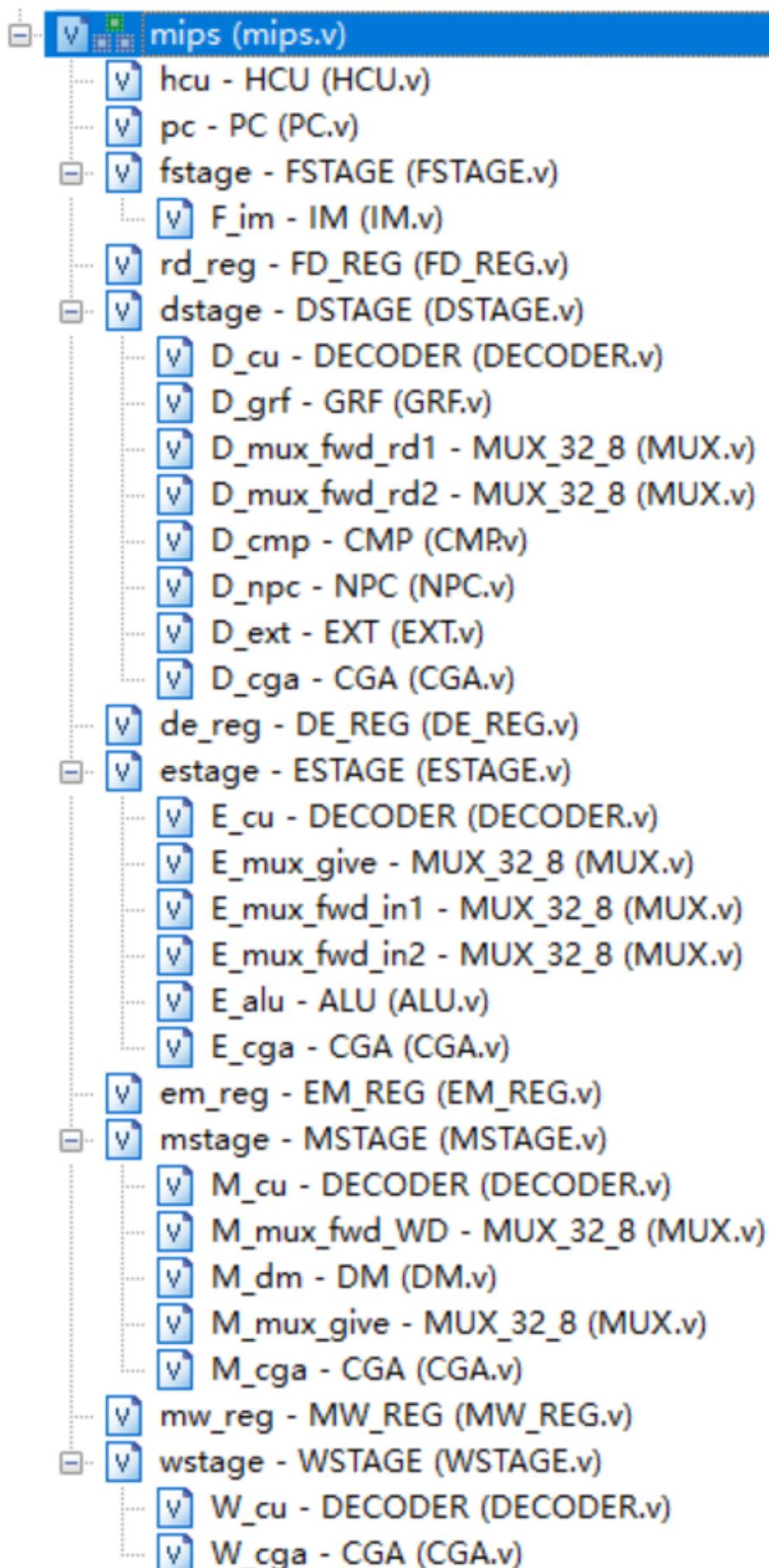
参照版



单周期CPU电路图



电路架构



指令细分

指令类即具有实现相似性的若干指令，例如对于 MIPS-C3 指令集（包含了 P5、P6 课下要求的指令集），可以进行如下分类：

- 寄存器立即数计算： addi, addiu, slti, sltiu, andi, ori, xori, sll, srl, sra
- 寄存器寄存器计算： add, addu, sub, subu,slt, sltu, and, or, nor, xor, sllv, srlv, sraw
- 根据寄存器分支： beq, bne, bgez, bgtz, blez, bltz
- 写内存： sw, sh, sb
- 读内存： lw, lh, lhu, lb, lbu
- 读乘除法寄存器： mfhi, mflo
- 写乘除法寄存器： mthi, mtlo, mult, multu, div, divu
- 跳转并链接： jal, jalr
- 跳转寄存器： jr, jalr
- 加载高位： lui
- 空指令： nop

上面的指令中并没有无条件跳转指令 j，这是因为 j 指令不产生数据也不使用数据，已被移出群

- 计算类型指令，修改ALU即可
- 跳转一般是**条件跳转** +** 条件写**。跳转指令一个好处在于它是在 D 级决定是否跳转的，也就是说在 D 级你可以获得全部的正确信息。所以我们的方案是 D 级生成一个 D_check 信号然后流水它。然后每一级根据这个信号判断写入地址/写入值之类的。
- 条件存储一般是最难的。但是掌握了套路之后也还好。条件存储的特点是必须要到 M 级才知道要写啥，这就给转发之类的造成了困难，所以我们的策略是**如果 D 级要读寄存器，而且新指令可能要写这个寄存器，那么就 stall**。具体来说是这样的：

```
//lwso
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (D_lwso ? D_rs_addr ==
5'd31 : D_rs_addr == E_RFDst);

// lrm
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (D_lrm ? D_rs_addr :
D_rs_addr == E_RFDst);
```

在 CU 中的写法则与条件跳转类似：

```
// lwso
M_check = D_lwso && condition;
// CU
assign RFDst = // ...
    lwso ? (check==1'd1 ? 5'd31 : 5'd0) : // 注意不是直接一个 check
    5'd0;
```

Tuse表格 (用不到时记为最大值即可)

rs操作	Tuse_rs	rt操作	Tuse_rt
calc_r(rs)\shiftS	1	calc_r(rt)	1
calc_i(rs)	1	shiftS(rt)	1
load(rs)	1	store(rt)	2
store(rs)	1	branch(rt)	0
branch(rs)	0		
jr/jalr(rs)	0		

错误可能原因

- 指令执行流错误 (pc错误)
- 阻塞错误
- 转发错误
- 写入GRF错误
- 写入DM错误

测试方案

- 简单修改p4时的生成mips代码的代码，进行大范围的随机测试
- 从需求者位置和供应者位置方面考虑手动构造了一些测试的数据
- 手动构造了一些测试数据来测试分支指令和跳转指令

```
.text
#lui
lui $0,0x1234
ori $t0,0xabcd
lui $t0,0xfffff
#ori
ori $t1,$t0,0x1234
ori $0,$t1,0xfffff
sll $t1,$t0,1
#add
ori $t2,0x1234
add $t2,$t2,$t2
add $0,$t2,$t2
ori $t3,0x0001
lui $t4,0xfffff
ori $t4,$t4,0xffffe
add $t4,$t3,$t4
nop
#sub
lui $t5,0xfffff
ori $t5,0xfffff
ori $t6,0xabcd
sub $t6,$t5,$t6
```

```
sub $t5,$t5,$t5
nop
#sw
ori $t7,4
sw $t7,-4($t7)
lw $s4,-4($t7)
sw $t0,4($0)
sw $t1,8($0)
sw $t2,12($0)
sw $t3,12($0)
sw $0,16($0)
nop
#jal
jal func
ori $v0,1
#beq
ori $s0,0x1
ori $s1,0xfffff
beq $s0,$s1,b1
nop
b2:
sub $s1,$s1,$s1
beq $0,$s1,b3
lui $v1,0xfffff
b1:
beq $0,$0,b2
add $s1,$s1,$t0
b3:
ori $s2,11451
add $s3,$s2,$0
lw $s7,0($0)
lw $s6,4($0)
lw $s5,8($0)
b4:
beq $s3,$s2,b4
ori $v0,$s2,0xab
func:
ori $a0,0x1234
add $a1,$a0,$a0
ori $a2,4
add $a2,$ra,$a2
jr $ra
ori $v0,$a2,0xffab
```

思考题

第一题 我们使用提前分支判断的方法尽早产生结果来减少因不确定而带来的开销，但实际上这种方法并非总能提高效率，请从流水线冒险的角度思考其原因并给出一个指令序列的例子。

以beq为例，改指令需要比较rs与rt所对应寄存器的值，但是实际上并不一定产生这两个数据。

```
lw $t1,0($0)
beq $t1,$0,label1
.....
label1:
```

第二题 因为延迟槽的存在，对于 jal 等需要将指令地址写入寄存器的指令，要写回 PC + 8，请思考为什么这样设计？

- 你考虑到jal的存在，所以无论是否跳转，都会在跳转指令之后执行位于pc+4的nop指令或者与数据无关的指令，所以应该写会pc+8

第三题 我们要求所有转发数据都来源于流水寄存器而不能是功能部件（如 DM、ALU），请思考为什么？

- 从寄存器到ALU等组合逻辑部件的输出需要消耗时间，增加某一级的延迟，那么最小时钟周期就有可能增大，背离了设计五级全速流水线CPU的初衷。

第四题 我们为什么要使用 GPR 内部转发？该如何实现？

- 当W级要写回寄存器值，而D级正要取出该寄存器的值时就会发生数据冲突，为了能得到正确的值，所以要设计内部转发。
- 仅需在GRF模块中判断一下A3与A1或者A2是否相等并注意不为5'd0即可。

第五题 我们转发时数据的需求者和供给者可能来源于哪些位置？共有哪些转发数据通路？

- 供给者来自于EXT、PCADD8、ALU、DM
- 需求者来自于D_GRF_RD1、D_GRF_RD2、E_ALU_A1、E_ALU_A2、M_DM_WD
- 转发数据通路：
 - E_give、M_give、W_give -> D_GRF_RD1
 - E_give、M_give、W_give -> D_GRF_RD2
 - M_give、W_give -> E_ALU_A1
 - M_give、W_give -> E_ALU_A2
 - W_give -> M_DM_WD

第六题 在课上测试时，我们需要你现场实现新的指令，对于这些新的指令，你可能需要在原有的数据通路上做哪些扩展或修改？提示：你可以对指令进行分类，思考每一类指令可能修改或扩展哪些位置。

提示：可以加一个以新增指令命名的信号，进行特判，防止影响到其他指令（比较保险）

- 对于计算类指令：改变DECODER---改变HCU---扩展ALU和EXT
- 对于跳转类指令：改变DECODER---改变HCU---修改NPC和CMP
- 对于访存类指令：改变DECODER---改变HCU---扩展DM指令

第七题 简要描述你的译码器架构，并思考该架构的优势以及不足。

- 我采用的是分布式译码
- 优势：减少了流水线寄存器的复杂度，上机时能让改动较少
- 不足：需要实例化多个寄存器，让代码变得臃肿，并且与实际工业生产的状况不符

第八题 请详细描述你的测试方案及测试数据构造策略。

- 简单修改p4时的生成mips代码的代码，进行大范围的随机测试
- 从需求者位置和供应者位置方面考虑手动构造了一些测试的数据
- 手动构造了一些测试数据来测试分支指令和跳转指令