P5 - 简单流水线CPU - 设计文档

这一部分的工作，主要是将原有的单周期设计进行流水线化改造。

事实上，转发逻辑并不困难，无脑全转发即可，本人开发的大部分时间都用于跟暂停作斗争了（笑，折腾半天才发现，自己摸索出的方法在教程里已经写了，不看教程导致的）

思考题

流水线架构

数据通路

- 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。

  再次明确，我们在上机时，应“宁加勿改”，能通过加东西解决的，绝对不对已有模块的功能等进行修改！

  这是一个应试做法。在实行指令集的标准指令时，我当然不会这么做。

  首先是对控制器。我们要引出一个对新加指令进行特判的控制信号。同时，在控制器内，我们还要将新指令，加入模块控制信号的生成中。

  然后是对于顶层设计。在这一设计思路的指导下，我们只对模块的输入动手。要干的即是，通过特判信号，控制要修改的模块的输入是否为新加指令指定的输入；特判信号有效则读新指令指定的输入，反之则按原设计指定输入。

  对于无法使用已有模块进行处理的指令操作，我们在顶层设计里单独加一小段逻辑，对其进行实现即可。

  然后就是对转发、暂停的处理了。这部分的工作，主要是对控制器的有关部分进行修改。

  对于转发，我们将对寄存器进行写入操作的指令归为“破坏性指令”。我们要先干的，是判断新加的指令是否具有破坏性；如果是，则将其加入转发控制逻辑的破坏性指令列表中。随后还要指定破坏性指令的写入目标。这里根据新加指令的写入目标对应一下即可。

  对于暂停，我们主要是对新指令的 `Tnew, Tuse` 进行分析，并将其加入对应指令的分类。

  上述即为解决的完整流程。

译码器

- 确定你的译码方式，简要描述你的译码器架构，并思考该架构的优势以及不足。

  本设计中采用分布式译码，并采取控制信号驱动型译码。

  具体实现是，除C级以外，各级都有一套独立的控制单元。这些控制单元均采用同一套CTRL，不过视各级需求不同，选择性地引出控制单元的输出。

  至于为何采取这种方式，主要是方便课上临时添加指令。我的课上加指令解法是，由控制器引出指示是否为新加指令的信号，并由该信号在顶层设计内进行对应的特判。如果采用当前设计，我们只需要在需要修改的层级引出这一特判信号，**不用对层间寄存器进行修改**；而统一编码则要求对层间寄存器进行修改。

  而且这样能尽可能减少层间要传递的信号，这一点在教程中已经提到了。

  不足之处？我们在各层级内使用的控制器都是一整套完整的控制器，其中有很多生成的控制信号可能在一个层级根本用不上；这当然可以各级单独造控制器解决，但这又会增加修改的次数，不利于课上上机进行临时添加指令。

  而且，不引出输出信号在编译的时候是会抛Warning的。

  还有一点就是，我们C级没有控制器，指令的解码**实际是在IFU读出指令后的下一周期，在D级进行的。** 这对暂停的处理带来了一些不便。

流水线冒险

冒险的解决

- 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。

  首先，我们要知道，这一做法提前了`beq`的`Tuse`，使之变为0。

  这就会导致：在前序有任何产生冒险写入操作时，`beq`必定会暂停，降低了运行效率。

  示例如下：

```
ori $2,$0,4
  ori $3,$0,4
  beq $2,$3,next
```

```
ori $2,$0,16
ori $3,$0,16
ori $7,$0,0
ori $8,$0,4
sw $2,0($7)
sw $3,0($8)
ori $2,$0,65535
ori $3,$0,65534
lw $2,0($7)
lw $3,0($8)
beq $2,$3,next # 暂停两个周期
```

- 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？

  很简单。延迟槽内指令是紧随在跳转指令之后，必定要被执行的。如果写回PC+4，延迟槽内指令会被再执行一次。
- 我们要求大家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？

  在实际电路中，DM、ALU获得稳定输出值是需要时间的。若直接将其作为转发数据的来源，在现实里很有可能得到的是不稳定的输出，**与预期取得的值不符。**

  我们在设计运行的时钟频率时，一般会保证在一个周期内，各单元都能得到稳定输出。故选择在下一周期，由流水寄存器获得转发值，这样取得的转发值是可以保证符合预期的。
- [P5 选做] 在冒险的解决中，我们引入了 AT 法，如果你有其他的解决方案，请简述你的思路，并给出一段指令序列，简单说明你是如何做到尽力转发的。

  AT法的确是最一般化的方法，没办法了...

  之前的一个改进型AT法是错的...

解决冒险的流水线实现

- 我们为什么要使用 GPR 内部转发？该如何实现？

  因为我们的指令执行到W级后，仍需要一个周期才能完成实际的写入操作，而此时可能已有指令请求待写入值。

  实现即是将转发值转发到GRF对应的输出。
- 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？

详见设计草稿内的转发逻辑分析。

流水线测试教程

测试数据示例

- [P5 选做] 请详细描述你的测试方案及测试数据构造策略。

  请直接看测试方案代码。

  总之就是逐层把指令都测一遍。

  构造策略就是边搭边测转发和暂停。不转发和不暂停作为特例给出。

覆盖率分析

- [P5、P6 选做] 请评估我们给出的覆盖率分析模型的合理性，如有更好的方案，可一并提出。
  略。

设计草稿

是次作业仍然可以认为是对本人P4设计的迭代开发，主要工作是插入流水线寄存器。

其余重要改动有：控制器内生成转发、暂停相关信号，NPC放入D级，**对NPC而非PC**加入对暂停的处理。

我们按字母顺序设计层级命名，分为`Command, Data, Execute, Memory, Write`五级。

- C(Command)级：`IFU`
- D(Data)级：`GRF`，`EXT`,`CMP`,`NPC`
- E(Execute)级：`ALU`
- M(Memory)级：`DM`
- W(Writeg)级: `GRF的A3, WE, WD部分

除了C级，各级均有自己的一套控制器。

命名规则

继续沿用P4的规定：

- 模块：模块取名全大写，实例化时首字母大写，其余部分小写。如`ALU Alu()`
- 控制信号取值：仅在使能信号处直接使用0或1，其余全部使用宏封装；宏名称满足`<module>_[<ctrl_name>_]<operation>`或`<mux>_<signal_name>`的命名格式，且控制信号

的命名全使用小写字符。如 `npc_op_norm` 与 `alu_add`

- `CTRL` 的控制信号输出端口：全小写命名，遵循 `<module/mux_name>_<ctrl_name>` 的形式，如 `malub_c`， `npc_cond`。此外， `CTRL` 的其它输出端口也是全小写命名。
- `wire` 型变量：前序全部添加下划线以作区分，如 `_wire`。若为控制信号，则命名为全小写。
- 模块端口：全大写。
- 模块的控制信号端口： `<module_name><ctrl_name>`，其中 `<ctrl_name>` 首字母大写，如 `ALUOp`
- 接入模块端口的 `wire`： `_<module_name>_<socket_name>`。如 `_ALU_A` 与 `_ALU_Op`。
- 上一条规则对于 `CTRL` 需要稍加修改。若为控制信号，则为 `_CTRL_<module_name>_<ctrl_name>`，如一根接上 `CTRL` 输出的线： `_CTRL_ALU_Op`
- 出于阅读便利（偷懒），stall和forward的格式可能不完全遵循上述规则

转发逻辑

自然，本设计中采用了全速转发。

（注：我们这里是分布译码）为此我们在控制器中引出 `forward` 信号，指示当前执行指令是否为“破坏性指令”，并引出 `target` 信号指示当前破坏性指令的写入目标。

在流水线各层级，传播D级GRF的读取地址 `GRF_A*`。

在转发终点，将转发来源的 `target` 与当前所在层级的 `GRF_A*` 进行比较。若比较后相等，且转发来源的 `forward` 信号有效，**且写入目标不为 `$0` （否则会转发出非0值，与 `$0` 行为不符）**，则进行转发。

我们需要明确：当一个模块产生新值时，其就会成为一个新的转发供给者。 这一部分可以与 `Tnew` 串起来：

- `Tnew = 0`：`_jal`，来自 `(_IFU_PC + 8)`
- `Tnew = 1`：`_add | _sub | _sll | _ori | _lui`，来自 `ALU`
- `Tnew = 2`：`_lw`，来自 `DM`

转发的来源（供给者）共有两个，一个是M级寄存器中，上一周期ALU的计算结果 **（吗？见后）**；另一个是W级寄存器中，即将写入GRF的数据。

M级向D，E级转发，W级向D，E，M级转发。

对于D级，我们采用“GPR 内部转发”，转发终点设为GRF的输出端口。

对于D级之后的层级，考虑到扩展需求，我们不将转发终点设定为具体的模块端口，而是直接对层间寄存器传播的 `D_GRF_RD*` 进行替换（当然加了MUX）。在各个模块处，将原来接的 `D_GRF_RD*` 替换为经转发处理后的 `_D_GRF_RD*_F`。

需求者需要列举？用了`GRF_RD*`的数据的都是需求者。D级的转发比较特殊，需要注意一下：D级的比较单元的输入，应该是**转发结果**。

不难注意到，M级与W级向D,E级的转发存在冲突。考虑如下指令序列：

```
add $2,(),()
add $2,$2,()
add (),$2,()
```

为了实现连续转发，我们决定设定转发的优先级：**M级转发为最高优先，W级次之，再次之则不转发。** 这是因为，以上述指令为例，第二条指令再次产生了新值，将由M级转发；若转发的结果不是M级而是W级，则转发的值并非最新值，不符合程序的设计预期。

然而！ 对于M级转发，转发来源**并不是唯一的！对当前的`jal`如此，在P6加了乘除法指令后更是如此！！**

一番分析之后，可以认定W级接上`_GRF_WD`就稳了，因为`_GRF_WD`前的MUX已经起到了指定来源的作用。故我们只用对M级转发进行处理。

为此，我们在控制器内再引出一个`forward_src`信号，指示当前的实际转发来源。当前一共有两个：

- `forward_src_alu`：`E_ALU_C`
- `forward_src_pc_8`：`(E_IFU_PC + 8)`

因为我的PC+8是在存进`DM`时再算的，故这里要手动加上8.

随后，造一MUX，根据这一控制信号选择转发供给者。转发终点处，接M级转发输入来源时，改为接上这个MUX即可。

暂停逻辑

STALL！STALL！要SPIN辣（误）

按照教程，我们采取冻结PC与CD层间寄存器，清空DE层间寄存器的方案。

在控制单元，我们引出`t_use_rd*, t_new`指示**各GRF读出数据**的使用时间，和新值生成时间。

整个暂停逻辑放在D级执行。

针对这一五级流水线设计的特殊性，我们对AT法进行了一小点修改。

> (以下复制于思考题解答)

> 在分析整个流水线的执行过程，我们发现，在第一次读出`Tuse,Tnew`时，最大可能的`Tuse, Tnew`均为2；**在`Tnew`小于2时，我们可以保证所有的冒险都可以通过转发解决。**
>
> 因此，我们不用对`Tnew`进行任何计算，直接将`Tnew`传入层间寄存器。在下一周期，我们再由当前的`Tuse`与上一周期的`Tnew`进行比较，进行是否暂停的判断。此后我们便**不再传播`Tnew`，`Tuse`，或在各级对`Tuse`与`Tnew`间的关系进行比较，因为此时的比较已经没有意义了。**
>
> 这样也可以避免减出负数，要上`$signed()`的问题。我不想跟这玩意纠缠。

若比较为真，且当前 `GRF_A*` 与上一条指令的写入目标相同，**且写入目标不为 `$0` （因为写入结果不会产生任何影响）**，则执行暂停。

前文提到，我们的PC冻结由NPC入手。在暂停信号有效时，我们让NPC直接输出来自IFU的PC值，不加4。

模块规格

主要是对NPC进行完善。

其他较P4中设计并无变化，故不加详细说明，直接复制P4中端口定义与控制信号。

控制模块同样单独给出。

NPC

计算下一PC值。

此时的NPC横跨了C级与D级，为了C级的IFU能够顺利运行，需要另开一端口，直接传入C级IFU的PC值，以进行PC+4。原有的PC值接口用于跳转与流水线处理。

端口定义

端口名	方向	说明
C_PC[31:0]	I	读入C级IFU的PC值
PC[31:0]	I	读入当前PC值，原接口
IMM16[15:0]	I	传入 `branch` 类指令的立即数
IMM26[25:0]	I	传入J型指令的26位立即数
IMM32[31:0]	I	传入 `jal` 等指令需要的32位立即数
EQ	I	来自CMP的比较结果
LT	I	来自CMP的比较结果

端口名	方向	说明
NPCOp[2:0]	I	控制信号
NPCCond[2:0]	I	指示B类指令的比较条件
NPC[31:0]	O	计算出的NPC值
clk	I	时钟信号
rst	I	**同步**复位信号

控制信号

NPCOp	3'b000	3‘b001	3‘b010	3'b011
功能	C_PC+4	branch	j26	j32
说明	正常操作	执行 `branch` 类指令	执行J型指令	执行 `jal` 类32位立即数指令

NPCCond	3'b000	3'b001	3'b010	3'b011	3'b100
功能	EQ	LT	GT	LE	GE
说明	==	<	>	<=	>=

IFU

仍然采用了P3中计算相对PC的取指令方法。后续如有这方面的扩展，需要留意。

端口定义

端口/位宽	方向	说明
NPC[31:0]	I	输入NPC单元计算得出的NPC值
INS[31:0]	O	读出PC指向的IM地址之指令
PC[31:0]	O	传出当前PC值
clk	I	时钟信号
reset	I	**同步**复位

CMP

进行**有符号**比较。

端口定义

名称/位宽	方向	说明
A[31:0]	I	输入RS
B[31:0]	I	输入RT或IMM
EQ	O	(A==B)
LT	O	(A < B)

EXT

端口定义

端口/位宽	方向	说明
A[15:0]	I	输入的16位立即数
EXTOp	I	指示拓展方式
B[31:0]	O	扩展结果

控制信号

ext_signed	1'b0	1'b1
功能	ext_ze	ext_se
说明	零扩展	符号扩展

GRF

端口定义

端口/位宽	方向	说明
A1[4:0]	I	要读取的$[rs]之地址
A2[4:0]	I	要读取的$[rt]之地址
A3[4:0]	I	要写入的$[rd]之地址
WD[31:0]	I	写入的数据
We	I	写使能信号

端口/位宽	方向	说明
RD1[31:0]	O	读取的$[rs]
RD2[31:0]	O	读取的$[rt]
rst	I	**同**步复位信号
clk	I	时钟信号

ALU

端口定义

名称/位宽	方向	说明
A[31:0]	I	输入RS
ALUOp[3:0]	I	ALU控制信号
B[31:0]	I	输入RT或IMM
SHAMT[4:0]	I	输入移位位数
C[31:0]	O	输出结果

控制信号

ALUOp	4'b0000	4'b0001	4'b0010	4'b0011
功能	alu_add	alu_sub	alu_or	alu_sll
说明	加运算	减运算	或运算	左移运算

DM

端口定义

端口/位宽	方向	说明
ADDR[31:0]	I	DM地址
WD[31:0]	I	待写入的数据
We	I	写使能信号
clk	I	时钟信号
rst	I	**同**步复位信号

端口/位宽	方向	说明
RD[31:0]	O	读出指定地址的数据

控制模块

（复制自P4设计文档）

具体实现的话，源码较文字描述更为直观，我们这里主要谈搭建思路。

仍然是分为`DEC, OR, AND`三级进行搭建。`DEC`将机器码分解成各信号区块，`OR`进行指令的翻译，`AND`进行控制信号的生成。

在`AND`层级，我们实际上做的**不是生成控制信号，而是生成各控制信号的条件布尔变量。** 在全部条件生成后，我们再通过`assign`语句生成控制信号。

以ALU的控制信号为例：

```
// ALU

wire _alu_sub, _alu_or, _alu_sll; // _alu_add = _add | _lw | _sw   set as
default

assign _alu_sub = _sub,

           _alu_or = _ori,

           _alu_sll = _sll | _lui;

// Signal

assign alu_op = (_alu_sub) ? `alu_sub :

                (_alu_or) ? `alu_or :

                (_alu_sll) ? `alu_sll : `alu_add;
```

由于正则表达式进行条件匹配的格式限制，我们必须留一个情况作默认值。此处ALU的默认控制信号取值就是`alu_add`。

至于转发和暂停的控制，执行逻辑部分已经讲清楚了，还要看建议直接看源码。

数据通路与MUX

数据通路

对于IFU输入必来自NPC此类的显然连接，我们在表中略去,在下单独给出。

IFU.NPC = NPC.NPC
CTRL.INS = IFU.INS
NPC.PC = IFU.PC
ALU.A = GRF.RD1
CMP.A1 = GRF.RD1，CMP.A2=GRF.RD2

注意！此处进行了流水线改造，请自动将各级接入的 `GRF.RD*` 替换为转发结果！

输入/指令	add	sub	sll(nop)	ori	lw	sw	b
NPC.NPCOp	npc_norm	npc_norm	npc_norm	npc_norm	npc_norm	npc_norm	n
NPC.Cond	/	/	/	/	/	/	E
NPC.IMM32	/	/	/	/	/	/	/
GRF.A3	CTRL.rd	CTRL.rd	CTRL.rd	CTRL.rt	CTRL.rt	/	/
GRF.We	1	1	1	1	1	0	0
GRF.WD	ALU.C	ALU.C	ALU.C	ALU.C	DM.RD	/	/
EXT.EXTOp	/	/	/	0	1	1	/
ALU.ALUOp	alu_add	alu_sub	alu_sll	alu_ori	alu_add	alu_add	/
ALU.B	GRF.A2	GRF.A2	GRF.A2	EXT.B	EXT.B	EXT.B	/
ALU.SHAMT	/	/	CTRL.shamt	/	/	/	/
DM.ADDR	/	/	/	/	ALU.C	ALU.C	/
DM.We	0	0	0	0	0	1	0
DM.WD	/	/	/	/	/	GRF.A2	/

(1):此处的移位方式并不重要，用sign_extend也是可以的...

MUX

MALUB_c	3'b000	3'b001
MALUB	GRF.A2	EXT.B

MALUSA_c	3'b000	3'b001
MALUSA	CTRL.SA	5'd16

MGRFA3_c	3'b000	3'b001	3'b010
MGRFA3	CTRL.rd	CTRL.rt	5'd31

MGRFWD_c	3'b000	3'b001	3'b010
MGRFWD	ALU.C	DM.RD	(IFU.PC+4)

测试方案

我们在前序设计测试程序的时候，其实已经完成了指令功能与转发+暂停+延迟槽功能的测试。直接沿用即可...

```
.text
# 操控$5为0xffff_ffff,$6为0x8000_0000
# 测试$0
ori $0,$0,65535
# 若通过，测试ori；zero_ext
ori $3,$0,4096 # 高第三位为1，测试$0是否不影响转发
ori $3,$3,65534 #最后一位0，高第三位为1，测试是否转发
# 若通过，测试lui
lui $3,65535
lui $0,65535
# 若通过，测试减
ori $4,$0,5 #init
ori $2,$0,1
sub $3,$4,$2 #正常减法
ori $2,$0,6
sub $3,$3,$2 #跨到-1
ori $2,$0,1
sub $3,$6,$2 # -2137483648 - 1，溢出到2137483647
sub $0,$3,$1 # $0
# 若通过，测试加
ori $4,$0,5 #init
ori $2,$0,1
add $3,$4,$2 #正常+法
```

```
ori $2,$0,1
sub $3,$0,$2 #$3跨到-1
add $3,$3,$2 #-1跨到0
ori $2,$0,1
add $3,$5,$2 # 2137483647 + 1，溢出到-2137483648
add $0,$3,$2 # $0
# sll
ori $3,$0,65535
sll $3,$3,16
sll $0,$3,1
sll $0,$0,0 #nop，无事发生
# 通过，测试sw
ori $3,$0,1
ori $2,$0,4
sw $3,0($2)
sw $3,4($2)
sw $3,-4($2)
# lw
# 完成上述测试后，0，4，8应存有1
ori $4,$0,1
ori $2,$0,4
lw $3,4($2)
lw $3,0($2)
lw $3,-4($3) # 暂停
lw $0,0($2) # $0
# beq
ori $2,1
ori $3,2
ori $4,1
that:
beq $2,$3,that
nop

ori $2,$0,16
ori $3,$0,16
ori $7,$0,0
ori $8,$0,4
sw $2,0($7)
sw $3,0($8)
ori $2,$0,65535
```

```
ori $3,$0,65534
lw $2,0($7)
lw $3,0($8)
beq $2,$3,next # 暂停两个周期
ori $2,$0,16
ori $3,$0,65535
next:
ori $3,$0,16
ori $2,$0,255
ori $3,$0,255
ori $4,$0,126
ori $5,$0,127
ori $4,$0,127
beq $4,$5,next1 # 暂停一个周期
ori $4,$0,0
ori $4,$0,511
ori $5,$0,511
next1:
ori $2,$0,511
ori $3,$0,511

jal test # 测试跳转，以及下方延迟槽，以及jal有关转发是否正确
ori $31,$31,0
ori $3,$0,65535
test:
ori $3,$31,0 # 继续测转发
ori $10,$0,65535
ori $21,$10,65535
```