Verilog 流水线 CPU 设计文档

接口设计

mips.v 顶层模块

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
i_inst_rdata	I[31:0]	i_inst_addr 对应的 32 位指令
m_inst_rdata	I[31:0]	m_data_addr 对应的 32 位数据
i_inst_addr	O[31:0]	需要进行取指操作的流水级 PC（一般为 F 级）
m_data_addr	O[31:0]	数据存储器待写入地址
m_data_wdata	O[31:0]	数据存储器待写入数据
m_data_byteen	O[3:0]	字节使能信号
m_inst_addr	O[31:0]	M 级 PC
w_grf_we	O	GRF 写使能信号
w_grf_addr	O[4:0]	GRF 中待写入寄存器编号
w_grf_wdata	O[31:0]	GRF 中待写入数据
w_inst_addr	O[31:0]	W 级 PC

将存储器 IM 与 DM 实现外置。

F 级

无独立模块，pc 寄存器产生下一地址，流出到外置 IM 中产生对应指令，再传回 mips.v。

D 级

E 级 alu 的比较前移至 D，使得新的地址能够在 D 级就产生。同样，该指令在 W 级需要的写寄存器序号，也可以在这里产生。

D 级寄存器

- pc_F → pc_D
- instr_F → instr_D

npc.v 下一地址

Sig	Type	Descript.
pc	I[31:0]	当前指令的地址

Sig	Type	Descript.
j26	I[25:0]	当前指令的 jump 地址域
imm	I[15:0]	当前指令的 Imm16 数据域
rO1	I[31:0]	GPR[rs]
zero	I	转发后的 GPR[rs] == GPR[rt] 判断信号
npcOp	I[2:0]	控制信号
npc	O[31:0]	下一指令的地址

grf.v 寄存器堆（读取部分）

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
readAddr1	I[4:0]	读寄存器 1 地址
readAddr2	I[4:0]	读寄存器 2 地址
readData1	O[31:0]	读取值 1
readData2	O[31:0]	读取值 2

E 级

计算得到 pc+8 结果，流水到 W 级作为写入数据信号，这样做以减少转发电路的大小。

新增乘除槽以支持乘除法指令。将乘除模块的结果合并到 aluRes 中，减少了流水信号的数量。

E 级寄存器

- pc_D → pc_E
- instr_D → instr_E
- mux_rO1_D → rO1_E
    - 转发后的 GPR[rs]
- mux_rO2_D → rO2_E
    - 转发后的 GPR[rt]
- rIR_D → rIR_E
    - 流水到 W 级使用的写寄存器序号信号

alu.v 算术逻辑单元

Sig	Type	Descript.
aluOp	I[3:0]	选择运算类型
shamt	I[4:0]	移位量
srcA	I[31:0]	寄存器值 1
srcB	I[31:0]	运算值 2

Sig	Type	Descript.
pc	I[31:0]	当前指令的地址
zero	O	结果是否为 0
aluRes	O[31:0]	运算结果

ALU 编码表

aluOp	功能
000 (SLL)	A << shamt
001 (OR)	A OR B
010 (ADD)	A + B
011 (LUI)	{B[15:0], 16'b0}
100 (SUB)	A - B
101 (LINK)	PC + 8
110 (XOR)	A ^ B
111 (AND)	A & B
1000 (SLT)	$signed(A) < $signed(B)
1001 (SLTU)	A < B

hilo.v 乘除模块

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
from	I[1:0]	regFrom 控制信号，用于 mfhi/mflo 指令
start	I[2:0]	写 hi/lo 控制信号，用于乘除法指令和 mthi/mtlo 指令
srcA	I[31:0]	同 alu
srcB	I[31:0]	同 alu
isbusy	O	是否正在进行乘除运算
result	O[31:0]	结果

封装了 hi/lo 寄存器而没有将之暴露在顶层设计中，根据需要传入 from/start 信号而产生结果。

M 级

DM 外置。

M 级寄存器

- pc_E → pc_M

- instr_E → instr_M
- aO_E → aO_M
    - 合并乘除槽结果和 pc+8 结果后的所有计算的结果
- rO1_E → rO1_M
- mux_rO2_E → rO2_M
    - 二次转发后的 aluSrcB
- rIR_E → rIR_M

W 级

W 级寄存器

- pc_M → pc_W
- instr_M → instr_W
- aO_M → aO_W
- mO_M → mO_W
    - DM 的读取数据
- rIR_M → rIR_W

grf.v 寄存器堆（写入部分）

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
regWrite	I	写使能
writeAddr	I[4:0]	写寄存器地址
writeData	I[31:0]	写入值

ctrl.v 控制单元

Sig	Type	Descript.
opcode	I[5:0]	opcode 字段
funct	I[5:0]	funct 字段
ext	O	符号扩展信号，在顶层设计中，选择将 imm16 符号扩展/零扩展
aluSrc	O	I 型指令用，选择立即数作为 srcB 进行计算
memWrite	O	Store 类指令用，将寄存器中的值赋给内存
regWrite	O	寄存器堆写使能
regDst	O[1:0]	写寄存器地址选择
regFrom	O[1:0]	写寄存器数据选择
loadOp	O[2:0]	按字节/半字访问支持
mdOp	O[2:0]	乘除类指令分类
npcOp	O[2:0]	计算下一指令的地址

Sig	Type	Descript.
aluOp	O[3:0]	运算类型选择
tUseRs	O	GPR[rs] 的 T_use 值，D 级暂停使用
tUseRt	O[1:0]	GPR[rt] 的 T_use 值，D 级暂停使用
tNew	O[1:0]	当前指令的 T_new 值，分为 E/M 级，D 级暂停使用

编码

控制单元编码表

Op	ext	aluSrc	memWrite	regWrite	regDst	loadOp	regFrom
R-Type				1	1		
add/sub/xor/and/or/slt/sltu				*	*		
j				*	*		
jr				*	*		
mult/multu/div/divu				*	*		
mthi/mtlo				*	*		
mflo				*	*		2
mfhi				*	*		3
jal				1	2		
ori/lui/andi		1		1			
bne/beq	1						
addi	1	1		1			
lw	1	1		1		0	1
lb	1	1		1		2	1
lh	1	1		1		4	1
sb	1	1	1				
sh	1	1	2				
sw	1	1	3				

npcOp 编码表

Op	npcOp
beq	1
j	2
jal	2

Op	npcOp
jr	3
bne	4

aluOp 编码表

Op	aluOp
ori/or	1
add/addi/lw/lb/lh/sw/sb/sh	2
lui	3
sub/beq/bne	4
jal	5
xor	6
and/andi	7
slt	8
sltu	9

mdOp 编码表

Op	mdOp
mult	1
mtlo	2
multu	3
mthi	4
div	5
mflo	6
divu	7
mfhi	8

Stall 分析表

Op	tUseRs	tUseRt	tNewE
add/sub/xor/and/or/slt/sltu/mfhi/mflo	1	1	1
mult/multu/div/divu/mthi/mtlo	1	1	zz
sw/sh/sb	1	2	zz
ori/lui/addi/andi	1	zz	1
lw/lh/lb	1	zz	2

Op	tUseRs	tUseRt	tNewE
beq	0	0	zz
jr	0	zz	zz
j	z	zz	zz
jal	z	zz	0

测试用例

```
ori $8, $0, 0x301c
ori $8, $0, 0x301c
ori $8, $0, 0x301c
mthi $8
ori $8, $0, 0x301c
ori $8, $0, 0x301c
ori $8, $0, 0x301c
```

这一用例测试了乘除模块中 `busy` 信号的初始化问题。如果对应的寄存器没有初始化，则 `mthi` 后的指令会一直保持阻塞。

```
lui $9, 4660
ori $9, $9, 22136
sw $9, 0($0)
sh $9, 4($0)
sh $9, 6($0)
sb $9, 8($0)
sb $9, 9($0)
sb $9, 10($0)
sb $9, 11($0)
lw $10, 0($0)
lh $11, 0($0)
lh $12, 2($0)
lb $13, 0($0)
lb $14, 1($0)
lb $15, 2($0)
lb $24, 3($0)
```

这一用例测试了按字节/半字访问的功能是否正常。

```
addi $9, $9, 10
addi $9, $9, 10
addi $9, $9, 10
addi $9, $9, 10
divu $9, $9
mflo $10
```

这一用例测试了乘除法指令是否成功阻塞。

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？
  - 因为需要乘除法需要进行阻塞，在进行乘除法计算期间，还需要计算其他指令。
  - 独立的 hi/lo 寄存器使得乘除法指令可以被打断，此时中间结果保存在 hi/lo 寄存器中，以便下一次运算时取出。
  - 本身是一种并行计算的思想。
- 真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。
  - 采用特定算法，将乘除法分解为多个阶段，多周期累加积/商，最终得到结果。
- 请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？
  - 将 E 级的 busy 信号传给 hazard 单元，判断 D 级是否为乘除指令的同时判断 E 级是否 busy，将这一结果直接和原 stall 信号相或。
- 请问采用字节使能信号的方式处理写指令有什么好处？
  - 独热编码，可以非常清晰地表明我们需要操作的字节，并将按字访问、半字访问、字节访问以及其他可能的访问方式统一起来，简化代码且不失灵活性。
- 请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？
  - 不是。实际获得的数据是一个完整的字，写入的数据只是其中一个字节，其余字节的数据没有改变。目前的存储器是按字访问的，如果编码改成按字节访问，则按字节读写更自然，更有效率。
- 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？
  - AT 法分析暂停的需求。
  - 将带使能、清空、重置等信号的寄存器封装成 mips.v 中的一个子模块，方便构建流水寄存器时调用。
  - 将 hi/lo 寄存器封装在乘除模块中，不直接暴露，通过信号从同一个输出中访问值。
  - 分布式译码，ctrl 带参调用，参数指明了这是哪一级的控制信号，以获取正确的 T_new。
  - 构建一个 split 模块，将各级指令独立切分成各个数据域，按需调用。
- 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？
  - 大致分为两类，一种和 p5 一样的普通冲突，一种是和乘除法相关的冲突。第一种在 AT 法的分析下得到完美解决，第二种冲突见上处理 Busy 信号带来的周期阻塞。相应的测试样例见上。
- 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。
  - 使用单元测试方法，独立验证各个指令的数据通路是否正确。其次根据指令的 T_use 和 T_new 进行分类，进行最小组合，测试其是否成功暂停和转发。
  - 测试时应着重考虑跳转类指令与其他指令的组合情况。