计算机组成原理实验报告

18375200 刘裕炜

一、CPU设计方案综述

（一）总体设计概述

本CPU为Verilog实现的五级流水线MIPS - CPU，支持的指令集包含{addu、subu、ori、lw、sw、beq、lui、j、jal、jr、nop}。为了实现这些功能，CPU主要包含了IFU、GRF、ALU、DM、指令译码等模块。

（二）关键模块定义

1、GRF

表1 GRF模块接口

信号名	方向	描述
RAddr0[4:0]	I	要读取寄存器的第一个地址
RAddr1[4:0]	I	要读取寄存器的第二个地址
WAddr[4:0]	I	写入地址
WriteData[31:0]	I	写入数据
WritePC[31:0]	I	执行写入的指令地址
RegWrite	I	写入使能
clk	I	时钟
reset	I	同步复位信号
RData0[31:0]	O	第一个地址寄存器的值
RData1[31:0]	O	第二个地址寄存器的值

表2 GRF功能定义

序号	功能名称	功能描述
1	读寄存器	RData0输出RAddr0地址所寻址的寄存器，RData1输出RAddr1地址所寻址的寄存器
2	写寄存器	RegWrite有效且reset不为1时，在时钟上升沿将WD的数据写入A3地址所寻址的寄存器
3	同步复位	clk上升沿时若reset信号有效，将32个GPR单元清零

2、IM

表3 IM模块接口

信号名	方向	描述
OpAddr[31:0]	I	指令地址
OpCode[31:0]	O	指令数

3、ALU

表4 ALU模块接口

信号名	方向	描述
In0[31:0]	I	第一个运算数
In1[31:0]	I	第二个运算数
ALUOp[2:0]	I	ALU控制信号
Res[31:0]	O	运算结果

表5 ALU功能定义

序号	功能名称	功能描述
1	加法	Res输出In0+In1的值
2	减法	Res输出In0-In1的值
3	按位或	Res输出In0和In1按位或的值
4	判断相等	若In0和In1相等，Zero输出1，否则输出0

4、DM

表6 DM模块接口

信号名	方向	描述
Addr[31:0]	I	操作地址
WData[31:0]	I	写入数据
MemWrite	I	写入使能
clk	I	时钟
reset	I	异步复位信号
WritePC[31:0]	I	执行写入的指令地址
RData[31:0]	O	操作地址对应内存单元存储的值

5、NPC

表7 NPC模块接口

信号名	方向	描述
PC[31:0]	I	当前PC值
branch_addr[31:0]	I	分支地址
jump_addr[31:0]	I	跳转地址
branch	I	branch信号
jump	I	jump信号
Stall	I	阻塞信号
NextPC[31:0]	O	下一条PC值
PC_4[31:0]	O	PC+4

6、指令译码器

指令译码器位于ID模块，用于将指令码译码，得到具体的指令类型，然后将指令类型流水至各级。

表8 控制器模块接口

信号名	方向	描述
Instr[31:0]	I	指令码
InstrType[59:0]	O	指令类型（独热码）

（三）关键功能实现

1、GRF的同时读写

在GRF中设置了内部转发，当同时对同一个寄存器进行读写时，将待写入数据直接交给读出端口。

2、分支和跳转

（1）分支

beq指令的相等判断前移至ID级，取出数据后立即判断得到是否转发的信号，并将转发信号和转发地址不经流水线寄存器直接连回NPC，这样可以做到只留一个延迟槽。

（2）跳转

j和jr指令在ID级译码后直接将跳转地址和信号连回NPC。

jal指令在ID级译码后将跳转地址和信号连回NPC，将PC+8的值继续流水，在EX级合并入R型计算指令的数据通路，视作在EX级产生结果。

3、冲突处理策略

T_{use} ： 该寄存器自当前时刻起算，再过多少个时钟周期，它的值需要被准备好供使用。

T_{new} ： 自当前时刻起算，再过多少个时钟周期，准备存入该寄存器的结果可以从某个流水线寄存器中取出。

采取计算 $T*{new}$和$T*{use}$的方法来确定是否转发或阻塞。为达到转发和阻塞效果，需要将待读取地址Rs、Rt，待写入地址Rt，Rd随指令一起流水。且在ID级解码产生信号类型时也应该一并产生各寄存器的 $T*{new}$ 和 $T*{use}$ ，在ID级判断是否阻塞，当阻塞解除后，将 $T*{new}$ 和 $T*{use}$ 逐级流水，并在进入新一级流水时将非0的值减1。

指令类型	T_{use}	$T_{new}\ in\ ID$	$T_{new}\ in\ EX$	$T_{new}\ in\ Mem$
R型计算指令	1	2	1	0
I型计算指令	1	2	1	0
load.rs (addr)	1	\	\	\
load.rt	\	3	2	1
store.rs (addr)	1	\	\	\
store.rt (value)	2	\	\	\
lui	\	1	0	0
jal	\	1	0	0
jr	0	\	\	\
beq	0	\	\	\

（1）lui和jal指令

lui和jal指令仅对立即数处理，因此在ID阶段就可以得到结果，在EX阶段将其并入R型指令的数据通路。

（2）阻塞方法

对同一个寄存器，存在某级$T_{new}>T_{use}$ 时，说明数据不能在使用的时刻前产生，需要阻塞。阻塞时，冻结IF/ID和PC寄存器，同步清零ID/EX寄存器，并更新前一指令的 T_{new} 值。

（3）转发方法

在排除阻塞情况后，对每个支持转发的元件数据输入的端口，其MUX的控制信号中，0表示原先数据通路来源的数据，其它接口为相应转发来源，转发信号通过比对需求数据的寄存器编号和后续流水线中最近的命中的寄存器编号决定是否执行转发。

转发节点如下：

- 转发供给：
  - ID/EX寄存器中ResFromID_ID_to_EX，对应于在ID阶段产生的结果；
  - EX/Mem寄存器中的ALUOut_EX_to_Mem，对应于在EX及其之前阶段产生的结果；
  - Mem/WB寄存器中的RegWriteData_Mem_to_WB，对应于在Mem及其之前阶段产生的结果。
- 转发需求：
  - ID级中GRF寄存器出口，与GRF两个读出数据合并，进行选择；
  - EX级中ALU的两个数据入口；
  - EX级中准备带入Mem级的写入数据；
  - Mem级中DM的写入数据入口。

具体转发方案：

- 每级流水中传下来的Rs，Rt的地址，若非0，则为需要读取且需要注意冲突的寄存器地址。每级流水中的RegWriteAddr若非0则为最终要写的寄存器地址。
- ID级中GRF寄存器出口仅在当指令在ID级立即需要后面级流水已经计算结束但还没回写的数据时进行转发，因此只需连接ResFromID_ID_to_EX和ALUOut_EX_to_Mem两个来源，并在转发控制单元内对这两个来源的寄存器号进行比对即可。对于Mem/WB寄存器的来源，由于GRF自身有内部转发机制因此不用单独转发。
- EX级中ALU入口对应于需要在ALU利用存在冲突的寄存器内容执行计算的情况。需要连接ALUOut_EX_to_Mem和RegWriteData_Mem_to_WB两个来源。
  - 注意ALUIn1处立即数优先级应当比转发高，即如果指令需要立即数参与运算，则应当忽略转发的结果。
- 对于DM的写入数据需要读取的寄存器：
  - 如果其上一次被写的指令与当前store类指令间隔有两条及以上，则在store指令的ID阶段，之前对其写入的指令最迟已经走到WB阶段，此时要么是无冒险，要么是同时读写，可以由GRF内部转发解决。
  - 如果其与上次写入间隔一条指令，则无法在store指令的Mem级进行转发，因为此时上一条写入指令的WB阶段已执行结束，但是上次写入发生在本次读取之后，存在冒险，需要转发。可以考虑将转发行为提前，在store指令的EX阶段，对待写入DM的值的寄存器在WB级流水检索是否存在可用转发源，若存在则将其转发过来，替代原本从GRF读出的旧值，若不存在则不作处理。
    - 注意这里接入ALUIn1的数据时可以选择接入转发前直接读出的数据ALUIn1或转发后的数据ALUIn1_bypass，但是我们后面在Mem入口处已经设定了转发，这里如果接入ALUIn1_bypass的话，一是难以判断是否转发，二是与后面重复，因此我们选择接入直接读出的数据ALUIn1。
    - 考虑将其与ALUIn1的转发通道合并，将ALUIn1的转发MUX改为不管Tuse的值是否为0都扫描后面有无可用转发源

- 如果其与上次写入为相邻两条指令，则在store的Mem阶段上一条指令已经走到WB，必然已经得到结果，可以从WB向Mem转发，因此DM入口处再检测一次是否能从WB转发即可。
- 需要注意的是，即使存在连续几条指令都对同一个寄存器写入，最后一条指令需要将该寄存器的值存回DM的情况，上述转发方案仍然成立，因为每次store指令的转发都是用最新产生的数据覆写store的数据流。

4、加指令工程化步骤

（1）修改头文件，加入新指令的独热码；

（2）修改InstrDecoder，加入新指令译码；

（3）修改AT_Cal，加入新指令的AT计算环节；

（4）

```
* 若加入的是跳转，则修改ID中的Jump信号产生模块
* 若加入指令在ID级产生值，则修改ID中产生结果的模块
* 若加入的是分支，修改ID中Branch信号产生模块
* 若需写寄存器，到Mem中改RegWrite信号
```

二、测试方案

测试思路

CPU在执行过程中，可能遇到的指令情况有无冲突，有冲突两种情况。对于冲突处理又有转发和暂停两种情况。因此可以列出下表：

	本条指令	Tuse=0 ID需要	Tuse=1 EX级需要	Tuse=2 Mem需要
前1条指令在ID	Tnew=1 ID结束产生	转发	转发	转发
前1条指令在ID	Tnew=2 EX结束产生	暂停	转发	转发
前1条指令在ID	Tnew=3 Mem结束产生	暂停	暂停	转发
前2条指令在ID	Tnew=1	转发	转发	转发
前2条指令在ID	Tnew=2	转发	转发	转发
前2条指令在ID	Tnew=3	暂停	转发	转发
前3条指令在ID	Tnew=1	GRF内部转发	GRF内部转发	GRF内部转发
前3条指令在ID	Tnew=2	GRF内部转发	GRF内部转发	GRF内部转发
前3条指令在ID	Tnew=3	GRF内部转发	GRF内部转发	GRF内部转发
前4条指令在ID	Tnew=1	无冲突	无冲突	无冲突
前4条指令在ID	Tnew=2	无冲突	无冲突	无冲突
前4条指令在ID	Tnew=3	无冲突	无冲突	无冲突

因此需要测试的点主要是连续3条指令内，最后一条与前两条之间存在冲突的情况。

测试1（Fibonacci数列生成）

```
.data
number: .space 400
.text
addu $s0, $0, $0
ori $s1, $0, 45
ori $s2, $0, 0
ori $s3, $0, 4
ori $s4, $0, 1
ori $t0, $0, 0
ori $t1, $0, 1
Loop_Start:
    beq $s0, $s1, Loop_End
    addu $t2, $t1, $t0
    sw $t2, 0($s2)
```

```
    addu $s2, $s2, $s3
    addu $s0, $s0, $s4
    subu $t0, $t1, $zero
    addu $t1, $t2, $0
    jal Loop_Start
Loop_End:
lw $t7, -20($s2)
```

测试2

python自动生成，手工做修改以检测jal和beq执行情况。

```
lui $0, 0x3000
lui $1, 0x3001
lui $2, 0x3002
lui $3, 0x3003
lui $4, 0x3004
lui $5, 0x3005
lui $6, 0x3006
lui $7, 0x3007
lui $8, 0x3008
lui $9, 0x3009
lui $10, 0x300a
lui $11, 0x300b
lui $12, 0x300c
lui $13, 0x300d
lui $14, 0x300e
lui $15, 0x300f
lui $16, 0x3010
lui $17, 0x3011
lui $18, 0x3012
lui $19, 0x3013
lui $20, 0x3014
lui $21, 0x3015
lui $22, 0x3016
lui $23, 0x3017
lui $24, 0x3018
lui $25, 0x3019
lui $26, 0x301a
lui $27, 0x301b
lui $28, 0x301c
lui $29, 0x301d
lui $30, 0x301e
lui $31, 0x301f
ori $0, $0, 0x1
ori $1, $0, 0x2
ori $2, $0, 0x3
ori $3, $0, 0x4
ori $4, $0, 0x5
ori $5, $0, 0x6
ori $6, $0, 0x7
ori $7, $0, 0x8
ori $8, $0, 0x9
ori $9, $0, 0xa
ori $10, $0, 0xb
ori $11, $0, 0xc
ori $12, $0, 0xd
```

```
ori $13, $0, 0xe
ori $14, $0, 0xf
ori $15, $0, 0x10
ori $16, $0, 0x11
ori $17, $0, 0x12
ori $18, $0, 0x13
ori $19, $0, 0x14
ori $20, $0, 0x15
ori $21, $0, 0x16
ori $22, $0, 0x17
ori $23, $0, 0x18
ori $24, $0, 0x19
ori $25, $0, 0x1a
ori $26, $0, 0x1b
ori $27, $0, 0x1c
ori $28, $0, 0x1d
ori $29, $0, 0x1e
ori $30, $0, 0x1f
ori $31, $0, 0x20
addu $31, $31, $31
addu $30, $30, $30
addu $29, $29, $29
addu $28, $28, $28
addu $27, $27, $27
addu $26, $26, $26
addu $25, $25, $25
addu $24, $24, $24
addu $23, $23, $23
addu $22, $22, $22
addu $21, $21, $21
addu $20, $20, $20
addu $19, $19, $19
addu $18, $18, $18
addu $17, $17, $17
addu $16, $16, $16
addu $15, $15, $15
addu $14, $14, $14
addu $13, $13, $13
addu $12, $12, $12
addu $11, $11, $11
addu $10, $10, $10
addu $9, $9, $9
addu $8, $8, $8
addu $7, $7, $7
addu $6, $6, $6
addu $5, $5, $5
addu $4, $4, $4
addu $3, $3, $3
addu $2, $2, $2
addu $1, $1, $1
addu $0, $0, $0
subu $0, $0, $15
subu $1, $1, $15
subu $2, $2, $15
subu $3, $3, $15
subu $4, $4, $15
subu $5, $5, $15
subu $6, $6, $15
```

```
subu $7, $7, $15
subu $8, $8, $15
subu $9, $9, $15
subu $10, $10, $15
subu $11, $11, $15
subu $12, $12, $15
subu $13, $13, $15
subu $14, $14, $15
subu $15, $15, $15
subu $16, $16, $15
subu $17, $17, $15
subu $18, $18, $15
subu $19, $19, $15
subu $20, $20, $15
subu $21, $21, $15
subu $22, $22, $15
subu $23, $23, $15
subu $24, $24, $15
subu $25, $25, $15
subu $26, $26, $15
subu $27, $27, $15
subu $28, $28, $15
subu $29, $29, $15
subu $30, $30, $15
subu $31, $31, $15
ori $5, $0, 4
sw $0, 0($5)
sw $1, 4($5)
sw $2, 8($5)
sw $3, 12($5)
sw $4, 16($5)
sw $5, 20($5)
sw $6, 24($5)
sw $7, 28($5)
sw $8, 32($5)
sw $9, 36($5)
sw $10, 40($5)
sw $11, 44($5)
sw $12, 48($5)
sw $13, 52($5)
sw $14, 56($5)
sw $15, 60($5)
sw $16, 64($5)
sw $17, 68($5)
sw $18, 72($5)
sw $19, 76($5)
sw $20, 80($5)
sw $21, 84($5)
sw $22, 88($5)
sw $23, 92($5)
sw $24, 96($5)
sw $25, 100($5)
sw $26, 104($5)
sw $27, 108($5)
sw $28, 112($5)
sw $29, 116($5)
sw $30, 120($5)
sw $31, 124($5)
```

```
lw $0, 128($zero)
lw $1, 124($zero)
lw $2, 120($zero)
lw $3, 116($zero)
lw $4, 112($zero)
lw $5, 108($zero)
lw $6, 104($zero)
lw $7, 100($zero)
lw $8, 96($zero)
lw $9, 92($zero)
lw $10, 88($zero)
lw $11, 84($zero)
lw $12, 80($zero)
lw $13, 76($zero)
lw $14, 72($zero)
lw $15, 68($zero)
lw $16, 64($zero)
lw $17, 60($zero)
lw $18, 56($zero)
lw $19, 52($zero)
lw $20, 48($zero)
lw $21, 44($zero)
lw $22, 40($zero)
lw $23, 36($zero)
lw $24, 32($zero)
lw $25, 28($zero)
lw $26, 24($zero)
lw $27, 20($zero)
lw $28, 16($zero)
lw $29, 12($zero)
lw $30, 8($zero)
lw $31, 4($zero)
ori $15, $zero, 0x0004
ori $16, $zero, 0x0100
ori $17, $zero, 0x0000
Loop0:
    beq $16, $0, EndLoop
    jal Loop1
Loop1:
    jal Loop2
Loop2:
    jal Loop3
Loop3:
    jal Loop4
Loop4:
    jal Loop5
Loop5:
    jal Loop6
Loop6:
    jal Loop7
Loop7:
    jal Loop8
Loop8:
    jal Loop9
Loop9:
    jal Loop10
Loop10:
    subu $16, $16, $15
```

```
    jal Loop0
EndLoop:
nop
```

测试3

对jr进行测试，包括正常的 $$ra$ 寄存器和一般的 $$16$ 寄存器。还测试了一些转发和阻塞功能。

```
ori $24, $zero, 0x3000  #text base addr
ori $31, $0, 22  #jump to NopEnd
sll $31, $31, 2
addu $31, $31, $24
ori $2, $0, 4
ori $3, $0, 0x10
ori $4, $0, 1
ori $5, $5, 0
Loop:
    beq $3, $zero, EndLoop
    nop
    ori $16, $0, 18  #jump to jr_test2
    sll $16, $16, 2
    #addu $16, $16, $24
    addu $5, $5, $4
    subu $3, $3, $2
    jal Loop
jr_test1:
    addu $9, $9, $4
    addu $16, $16, $24
    jr $16
jr_test2:
    addu $10, $10, $4
    jr $ra
EndLoop:
    addu $7, $7, $4
    jal jr_test1
addu $8, $8, $4
NopEnd:
nop
lui $7, 0x1234
ori $7, $7, 0x5678
nop
ori $16, $0, 0x12
addu $7, $16, $16
ori $12, $0, 4
sw $7, 4($12)
```

测试4

连续冲突的转发和阻塞测试。

```
ori $7, $0, 1
addu $8, $8, $7
ori $15, 0
sw $15, 0($0)
ori $15, $0, 1
sw $15, 4($0)
```

```
ori $15, $0, 2
sw $15, 8($0)
ori $16, $0, 3
sw $15, 12($0)
ori $16, $0, 4
sw $15, 16($0)
ori $16, $0, 5
sw $15, 20($0)
ori $16, $0, 6
sw $15, 24($0)
ori $16, $0, 7
sw $16, 28($0)
ori $1, $0, 4
ori $2, $0, 8
ori $3, $0, 12
nop
nop
nop  #初始化DM内容

ori $1, $0, 4
nop
nop  #初始化$1内容

addu $1, $2, $3
lw $1, 0($2)
beq $1, $15, end1
nop
ori $13, $0, 0x1234

addu $1, $2, $3
jal end1
addu $10, $31, $8
end1:
nop

ori $1, $0, 4 #初始化$1内容
nop
nop
addu $1, $2, $3
lw $1, 0($1)
addu $10, $1, $0
```

三、思考题

流水线冒险

1.在采用本节所述的控制冒险处理方式下，PC的值应当如何被更新？请从数据通路和控制信号两方面进行说明。

数据通路：首先在NPC元件内部可以直接计算PC+4，然后需要将GRF的Rs寄存器值，或26位立即数左移并与PC高位拼接后的结果接到NPC中供跳转时选择，将指令立即数扩展并与PC+4求和后的结果接到NPC供分支时选择。

控制信号：在ID级接出分支和跳转控制信号，接入NPC，然后在阻塞控制器中接出阻塞信号，当阻塞信号激活时，将PC的值回写PC，否则在NPC中计算下一条指令应该是跳转、分支还是普通地PC+4并写入PC。

2.对于jal等需要将指令地址写入寄存器的指令，为什么需要回写PC+8?

因为PC+4地址的指令已经作为延迟槽机制执行过了，如果需要跳回的话应该跳到PC+4的下一条，即PC+8。

数据冒险的分析

为什么所有的供给者都是存储了上一级传来的各种数据的流水级寄存器，而不是由ALU或者DM等部件来提供数据?

由ALU或DM等部件提供数据相当于把两级流水连一块了，首先这会导致关键路径的极大增长，制约CPU的时钟周期；其次这无法实现类似前级ALU结果转发到本级ALU输入的情况，因为鲁莽地将ALU的结果再接回ALU的输入端将会直接导致电路错误。；而且这样可能会导致前一级还没有计算完成的结果就被输入到本级中，造成电路的毛刺。

AT法处理流水线数据冒险?

1.“转发（旁路）机制的构造”中的Thinking 1-4

- 如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。
    - ```
      addu $1, $2, $3
      lw $1, 0($1)
      beq $1, $15, end1
      ```
    - 这样的指令序列中，lw需要addu结果的转发，beq又需要lw结果的转发，且如果beq直接采用原始数据，就会忽略掉lw对 $$1$ 的写入，从而出错。
- 我们为什么要对GPR采用内部转发机制？如果不采用内部转发机制，我们要怎样才能解决这种情况下的转发需求呢?
    - 内部转发机制使得同时对GPR进行读写成为可能。如果不采用这样的机制，则需要在GPR的读入口处设立一个转发来源，允许从WB级向GPR出口转发。
- 为什么0号寄存器需要特殊处理?
    - 因为对0号寄存器的写入相当于无效指令，对0号寄存器的读取永远不会因为0号寄存器被写而需要阻塞或转发。
- 什么是“最新产生的数据”?
    - 从该指令向前数最近的一次对该指令要使用的寄存器进行写入的数据。

2.在AT方法讨论转发条件的时候，只提到了“供给者需求者的A相同，且不为0”，但在CPU写入GRF的时候，是有一个we信号来控制是否要写入的。为何在AT方法中不需要特判we呢？为了用且仅用A和T完成转发，在翻译出A的时候，要结合we做什么操作呢?

AT方法仅对形成结果前的输入进行转发，与写入无关。

AT方法在翻译A时，若指令不写寄存器，可以将写入地址设为 $$0$ ，在搜索时就会忽略本级结果，而WE在指令处理全过程中不需要被除了最终写入的GRF之外的任何元件读取。
      ```