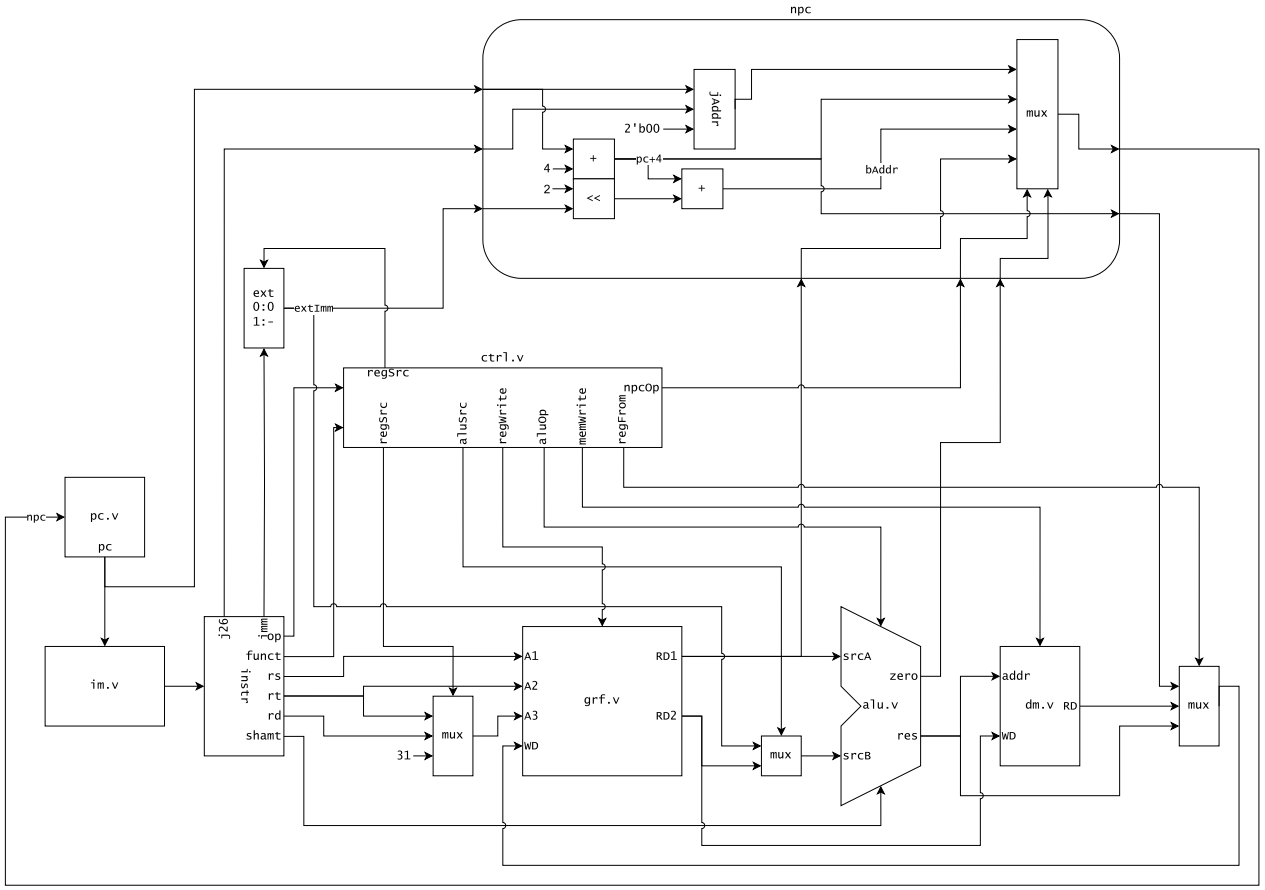
Verilog 流水线 CPU 设计文档

数据通路结构图



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

pc.v 程序计数器

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
npc	I[31:0]	下一指令的地址
pc	O[31:0]	当前指令的地址

im.v 指令存储器

Sig	Type	Descript.
pc	I[31:0]	待取指令的地址
instr	O[31:0]	指令

grf.v 寄存器堆

Sig	Type	Descript.
clk	I	时钟信号
reset	I	同步复位
regWrite	I	写使能
readAddr1	I[4:0]	读寄存器 1 地址
readAddr2	I[4:0]	读寄存器 2 地址
writeAddr	I[4:0]	写寄存器地址
writeData	I[31:0]	写入值
readData1	O[31:0]	读取值 1
readData2	O[31:0]	读取值 2

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
zero	O	结果是否为 0
aluRes	O[31:0]	运算结果

ALU 编码表

aluOp	功能
000	A << B(shamt)
001	A OR B
010	A + B
011	{B[15:0], 16'b0}
100	A - B

aluOp	功能
101	
110	A ^ B
111	

dm.v 数据存储器

Sig	Type	Descript.
clk	I	内置时钟
reset	I	异步复位
memWrite	I	写使能
memAddr	I[31:0]	地址
writeData	I[31:0]	写入值
readData	O[31:0]	读取值

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，0 表示 rt，1 表示 rd，2 表示 $31
regFrom	O[1:0]	写寄存器数据选择，0 表示从 alu 获取，1 表示获取内存，2 表示获取 pc + 4
npcOp	O[2:0]	计算下一指令的地址，0 表示 pc + 4，1 表示 B 类指令，2 表示 J 类指令，3 表示 JR 类指令
aluOp	O[3:0]	运算类型选择

编码

控制单元编码表

Op	OpCode/funct	ext	aluSrc	memWrite	regWrite	regDst	regFrom	npcOp	aluOp
R-Type	000000				1	1			
add	100000								2
sub	100010								4
xor	100110								5
jr	001000							3	
j	000010							2	
jal	000011				1	2	2	2	
beq	000100	1						1	4
ori	001101		1		1				1
lui	001111		1		1				3
lw	100011	1	1		1		1		2
sw	101011	1	1	1					2

测试用例

```
ori $8, $0, 0x301c
jr $8
lui $10, 10
lui $11, 11
nop
add $12, $10, $11
```

```
ori $28, $0, 0
ori $29, $0, 0
ori $1, $0, 4112
lui $2, 34595
ori $3, $0, 30806
lui $4, 34303
ori $5, $0, 1
lui $6, 65535
ori $7, $0, 65535
add $1, $1, $2
add $9, $1, $3
sub $8, $1, $2
sub $0, $7, $0
beq $28, $17, ADDR_0x3044
nop
beq $0, $0, ADDR_0x3094
nop
ADDR_0x3044:
beq $1, $2, ADDR_0x3094
nop
ori $2, $0, 12
nop
nop
nop
jal ADDR_0x306c
sw $1, 0($2)
beq $0, $0, ADDR_0x3094
add $1, $1, $2
ADDR_0x306c:
add $1, $1, $2
add $1, $1, $2
add $1, $1, $2
sw $31, 0($2)
lw $1, 0($2)
nop
nop
nop
jr $1
sw $31, 0($2)
ADDR_0x3094:
beq $0, $0, ADDR_0x3094
nop
```

思考题

- 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。
    - `T_use < T_new` 时需要阻塞流水线，将 `beq` 等跳转指令提前到 `D` 级，使得其 `T_use` 为 `0`. 这样就增大了阻塞的概率。

```
beq $t1, $t2, label
ori $t1, $0, 233
# other code
label:
# other code
```

- 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？
    - `PC + 4` 为延迟槽地址，已执行，直接执行其下内容。
- 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？
    - 从功能部件直接转发，会增加某一级的延迟，反而降低总效率。
- 我们为什么要使用 `GPR` 内部转发？该如何实现？
    - 其实就是 `W` 级数据写回到 `GPR` 内部，实际上从单周期来，架构已经自然实现该目标。
- 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？
    - 需求者：`grf` 的输出、`alu` 的输入、`dm` 的数据写入
    - 供给者：`alu` 的输出、`grf` 的写入
- 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。
    - 先增加数据通路，再设计阻塞和转发的数据通路，最后设计控制信号和阻塞转发的信号。
- 简要描述你的译码器架构，并思考该架构的优势以及不足。
    - 分布式译码，需要实例化多个控制器，但可以降低流水线之间传递的信号量。
    - 控制信号驱动型，和 `Logisim` 类似，代码简洁，但较难找错误。