**计算机体系结构实验课程期末综合实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验名称 | 优化 CPU 系统 | | | 班级 | 李雨森班 |
| 学生姓名 | 姚知言 | 学号 | 2211290 | 指导老师 | 董前琨 |
| 实验地点 | 实验楼A306 | | 实验时间 | 2024年11—12月 | |

1. **实验目的**

加深对计算机组成原理和体系结构理论知识的理解。

培养对 CPU 设计的兴趣，在理解现有 CPU 架构的基础上，引发对体系结构的思考和创新。

培养创新思维能力，并通过实践验证新想法。

1. **实验内容说明**

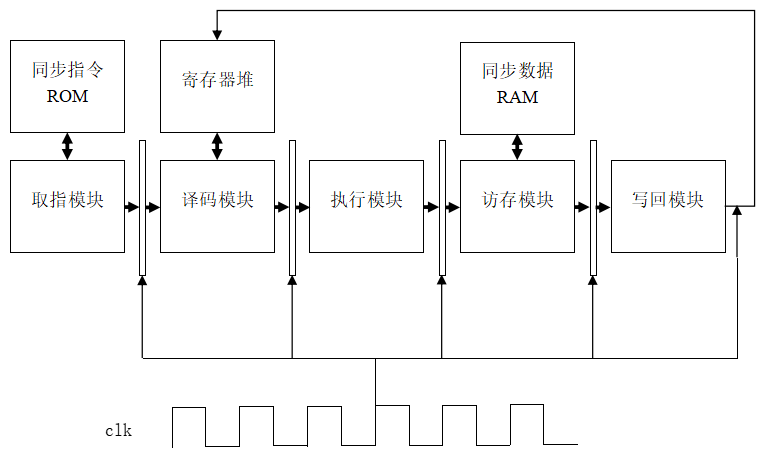
（1）分析静态 5 级流水 CPU 中的流水线阻塞情况，包括数据相关、控制相关、结构

相关等，使用前递技术优化流水线设计，尽可能减少流水线阻塞情况。

（2）扩展MIPS指令集，针对R型，I型，J型指令均进行扩展，并验证结果。

（3）分析MEM阶段的执行流程，设计cache，减少MEM阶段对内存访问的延迟。

1. **实验原理图**



1. **实验步骤**

（1）前递技术优化流水线设计

**(a)pipeline\_cpu.v**

decode ID\_module( // 译码级

.rf\_rs\_value (rf\_rs\_value ), // I, 32

.rf\_rt\_value (rf\_rt\_value ), // I, 32

...

//forwarding

.EXE\_f\_result (EXE\_f\_result ),

.EXE\_f\_valid (EXE\_f\_valid),

.EXE\_f\_wdest (EXE\_f\_wdest),

.MEM\_f\_result (MEM\_f\_result ),

.MEM\_f\_valid (MEM\_f\_valid),

.MEM\_f\_wdest (MEM\_f\_wdest)

);

将寄存器传入的rs/rt value改为rf\_rs/rt\_value，在后续选择把rf的结果或者前递结果赋值。

在decode阶段增加EX和MEM的前递输入信号，valid表示前递有效的使能信号，wdest表示前递的寄存器号，result表示前递的结果。

wire [31:0] rf\_rs\_value;

wire [31:0] rf\_rt\_value;

regfile rf\_module( // 寄存器堆模块

...

.rdata1 (rf\_rs\_value ), // O, 32

.rdata2 (rf\_rt\_value ), // O, 32

..

);

同步调整文件中的wire命名和rf端口绑定。

exe EXE\_module( // 执行级

...

//forwarding

.EXE\_result (EXE\_result)

);

mem MEM\_module( // 访存级

...

//forwarding

.MEM\_result (MEM\_result)

);

在exe阶段和mem阶段增加结果的输出信号。（虽然这一结果也包括在总线中，但这样实现较为直观）

//forwarding

wire [31:0] EXE\_result;

wire [31:0] MEM\_result;

reg [31:0] EXE\_f\_result,MEM\_f\_result;

reg [4:0] EXE\_f\_wdest,MEM\_f\_wdest;

reg EXE\_f\_valid,MEM\_f\_valid;

//ID与EXE、MEM、WB交互

wire [ 4:0] EXE\_wdest;

wire [ 4:0] MEM\_wdest;

wire [ 4:0] WB\_wdest;

设计新的wire读取前递结果，设计新的reg锁存结果。将wdest的wire定义位置提前，以便使用。

always @(posedge clk)

begin

//if (!resetn || cancel)

//begin

// MEM\_valid <= 1'b0;

//end

//else if (MEM\_allow\_in)

//begin

// MEM\_valid <= EXE\_over;

//end

if(resetn && !cancel && MEM\_allow\_in && EXE\_over) begin

MEM\_valid <= 1'b1;

EXE\_f\_result <= EXE\_result;

EXE\_f\_valid <= 1'b1;

EXE\_f\_wdest <= EXE\_wdest;

end

else begin

MEM\_valid <= 1'b0;

EXE\_f\_result <= 32'b0;

EXE\_f\_valid <= 1'b0;

EXE\_f\_wdest <= 5'b0;

end

end

更改MEM的valid赋值语句块，保留原valid赋值不变，增加前递结果的锁存。

always @(posedge clk)

begin

//if (!resetn || cancel)

//begin

// WB\_valid <= 1'b0;

//end

//else if (WB\_allow\_in)

//begin

// WB\_valid <= MEM\_over;

//end

if(resetn && !cancel && WB\_allow\_in && MEM\_over) begin

WB\_valid <= 1'b1;

MEM\_f\_result <= MEM\_result;

MEM\_f\_valid <= 1'b1;

MEM\_f\_wdest <= MEM\_wdest;

end

else begin

WB\_valid <= 1'b0;

MEM\_f\_result <= 32'b0;

MEM\_f\_valid <= 1'b0;

MEM\_f\_wdest <= 5'b0;

end

end

WB类似。

**(b)decode.v**

module decode( // 译码级

...

input [ 31:0] rf\_rs\_value, // 第一源操作数值(rf)

input [ 31:0] rf\_rt\_value, // 第二源操作数值(rf)

...

//forwarding

input [31:0] EXE\_f\_result,

input [4:0] EXE\_f\_wdest,

input EXE\_f\_valid,

input [ 31:0] MEM\_f\_result,

input [4:0] MEM\_f\_wdest,

input MEM\_f\_valid

);

同步进行输入端口的修改。

wire [31:0] rs\_value,rt\_value;

assign rs\_wait = ~inst\_no\_rs & (rs!=5'd0)

& ( (rs==EXE\_wdest) | (rs==MEM\_wdest) | (rs==WB\_wdest) )

& (~EXE\_f\_valid | ~(rs==EXE\_f\_wdest))

& (~MEM\_f\_valid | ~(rs==MEM\_f\_wdest));

assign rt\_wait = ~inst\_no\_rt & (rt!=5'd0)

& ( (rt==EXE\_wdest) | (rt==MEM\_wdest) | (rt==WB\_wdest) )

& (~EXE\_f\_valid | ~(rt==EXE\_f\_wdest))

& (~MEM\_f\_valid | ~(rt==MEM\_f\_wdest));

assign rs\_value = (rs==5'd0) ? 32'd0 :

(EXE\_f\_valid & (rs==EXE\_f\_wdest)) ? EXE\_f\_result :

(MEM\_f\_valid & (rs==MEM\_f\_wdest)) ? MEM\_f\_result :

rf\_rs\_value;

assign rt\_value =(rt==5'd0) ? 32'd0 :

(EXE\_f\_valid & (rt==EXE\_f\_wdest)) ? EXE\_f\_result :

(MEM\_f\_valid & (rt==MEM\_f\_wdest)) ? MEM\_f\_result :

rf\_rt\_value;

更改wait条件，增加前递找不到结果才等待。

定义rs/rt\_value，若前递可以获得结果则通过前递结果赋值，否则读取寄存器的值。

注意：这里对于rs=0 or rt=0的特殊判定很重要！因为在不需要写回的指令中会将对应的EXE/MEM的wdest清零，若此时rs/rt恰好为0号寄存器，会导致错误的匹配。但0号寄存器的数值应该一直保持为0，所以也不需要去特意修改wait，直接把32’d0赋值就可以。

**(c)exe.v**

module exe( // 执行级

...

//forwarding

output [31:0] EXE\_result

);

还需要删除原有的exe\_result定义，将后续exe\_result更改为EXE\_result。

**(d)mem.v**

module mem( // 访存级

...

//forwarding

output [ 31:0] MEM\_result

);

还需要删除原有的mem\_result定义，将后续mem\_result更改为MEM\_result。

（2）指令添加设计

在本实验中实现了以下五条指令。

a.R型指令：

nand，nand rd,rs,rt：rd=^(rs&rt)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Bit | 31-26 | 25-21 | 20-16 | 15-11 | 10-6 | 5-0 |
| R-type | op | rs | rt | rd | shamt | func |
| nand | 000000 | rs | rt | rd | 00000 | 111111 |

b.I型指令：

subiu， rt <- rs - (zero-extend)immediate；

nori， rt <- not(rs | (zero-extend)immediate)。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bit | 31-26 | 25-21 | 20-16 | 15-0 |
| I-type | op | rs | rt | immediate |
| subiu | 111101 | rs | rt | immediate |
| nori | 111110 | rs | rt | immediate |

c.J型指令：

jc，跳转到address置寄存器31为0，reg[31]=0,PC={next\_pc[31:28],target <<2}。

|  |  |  |
| --- | --- | --- |
| Bit | 31-26 | 25-0 |
| R-type | op | address |
| jc | 111111 | address |

以下是代码实现：

**(a)pipeline\_cpu.v**

wire [168:0] ID\_EXE\_bus;

reg [168:0] ID\_EXE\_bus\_r;

因为ID->EXE的标记位增加，需要调整总线位宽。

**(b)decode.v**

module decode( // 译码级

...

output [168:0] ID\_EXE\_bus, // ID->EXE总线

...

);

需要调整总线位宽，因为增加了一个标记位INST\_JC，同时ALU的控制信号也多了一位。

wire inst\_NAND, inst\_SUBIU, inst\_NORI, inst\_JC;

assign inst\_NAND = op\_zero & sa\_zero & (funct == 6'b111111);

assign inst\_SUBIU = (op == 6'b111101);

assign inst\_NORI = (op == 6'b111110);

assign inst\_JC = (op == 6'b111111);

增加的指令以及指令译码。

assign inst\_jbr = inst\_J | inst\_JAL | inst\_jr

| inst\_BEQ | inst\_BNE | inst\_BGEZ

| inst\_BGTZ | inst\_BLEZ | inst\_BLTZ

| inst\_JC;

跳转指令新增JC。

wire inst\_nand;

assign inst\_nand = inst\_NAND;

新增nand信号，用于ALU指令分类。

assign inst\_imm\_zero = inst\_ANDI | inst\_LUI | inst\_ORI | inst\_XORI | inst\_NORI | inst\_SUBIU;

NORI和SUBIU都是需要向前补充0来进行无符号扩展的。

注：原代码中ADDIU在有符号扩展分类中，不知道是设计错误还是故意为之，个人倾向于认为这是一处错误，但在此我并没有进行修改。

assign inst\_wdest\_rt = inst\_imm\_zero | inst\_ADDIU | inst\_SLTI

| inst\_SLTIU | inst\_load | inst\_MFC0;

assign inst\_wdest\_31 = inst\_JAL | inst\_JC;

assign inst\_wdest\_rd = inst\_ADDU | inst\_SUBU | inst\_SLT | inst\_SLTU

| inst\_JALR | inst\_AND | inst\_NOR | inst\_OR

| inst\_XOR | inst\_SLL | inst\_SLLV | inst\_SRA

| inst\_SRAV | inst\_SRL | inst\_SRLV

| inst\_MFHI | inst\_MFLO | inst\_NAND;

填写目的寄存器信号，NAND的目的寄存器是rd，JC的目的寄存器是31，另外两条因为包括在inst\_imm\_zero中了，不需要再进行添加。

assign inst\_no\_rt = inst\_ADDIU | inst\_SLTI | inst\_SLTIU

| inst\_BGEZ | inst\_load | inst\_imm\_zero

| inst\_J | inst\_JAL | inst\_MFC0

| inst\_SYSCALL | inst\_JC;

将JC加入指令不包括rt的集合，以免错误读取。SUBIU和NORI同样是因为包括在inst\_imm\_zero中了，不需要再进行添加，而NAND的rs和rt位置都存储寄存器地址，不应该在此添加。

assign j\_taken = inst\_J | inst\_JAL | inst\_jr | inst\_JC;

将JC加入无条件跳转的信号中。

wire [12:0] alu\_control;

assign alu\_control = {inst\_nand,

inst\_add, // ALU操作码，独热编码

inst\_sub,

inst\_slt,

inst\_sltu,

inst\_and,

inst\_nor,

inst\_or,

inst\_xor,

inst\_sll,

inst\_srl,

inst\_sra,

inst\_lui

};

添加nand到alu\_control中，并更改其位宽。

assign ID\_EXE\_bus = {multiply,mthi,mtlo,

alu\_control,inst\_JC,alu\_operand1,alu\_operand2,

mem\_control,store\_data,

mfhi,mflo,

mtc0,mfc0,cp0r\_addr,syscall,eret,

rf\_wen, rf\_wdest,

pc};

在总线中增加jc的信号，以便后续正确修改31寄存器，与jal区分开来。

**(c)exe.v**

module exe(

...

input [168:0] ID\_EXE\_bus\_r,// ID->EXE总线

...

);

...

wire [12:0] alu\_control;

...

assign {multiply,

mthi,

mtlo,

alu\_control,

jc,

alu\_operand1,

alu\_operand2,

mem\_control,

store\_data,

mfhi,

mflo,

mtc0,

mfc0,

cp0r\_addr,

syscall,

eret,

rf\_wen,

rf\_wdest,

pc } = ID\_EXE\_bus\_r;

与decode对应，更改总线位宽和信息对应。

assign EXE\_result = mthi ? alu\_operand1 :

mtc0 ? alu\_operand2 :

multiply ? product[63:32] :

jc ? 32'd0:

alu\_result;

新增如果是JC指令，需要将result清零。

**(d)alu.v**

module alu(

input [12:0] alu\_control, // ALU控制信号

...

);

控制信号位宽对应调整。

wire alu\_nand;

assign alu\_nand = alu\_control[12];

为nand增加并绑定新的控制信号。

wire [31:0] nand\_result;

assign nand\_result = ~and\_result;

添加为nand计算结果。

assign alu\_result = alu\_nand ? nand\_result:

(alu\_add|alu\_sub) ? add\_sub\_result[31:0] :

alu\_slt ? slt\_result :

...

32'd0;

在选择结果输出中增加nand。

（3）cache实现

在本实验中，我设计了一个较为简单的cache，其技术细节为：

共有32块，每块4字节，采用直相联和写穿透的方式实现。

同时，仅在写指令的时候同步对cache进行刷新，对于读指令时的cache miss情况，暂时不做cache更新。

以下是代码实现：

**(a)cache.v[新增]**

module cache(

input [31:0]addr, //输入地址（读/写）

input wen, //写使能

input [31:0] w\_data, //写数据

output now\_valid, //当前cacheline的有效位

output [24:0] now\_label, //当前cacheline存储的地址

output [31:0] r\_data //当前cacheline存储的数据

);

//cacheline定义：最高位表示valid,中间25位表示地址标记（32-5-2），末32位表示存储的数据

reg [57:0] cacheline [31:0];//[57] valid,[56:32]addr,[31:0] data

//cache初始化为0

integer i;

initial begin

for (i = 0; i < 32; i = i + 1) begin

cacheline[i] = 57'b0;

end

end

wire [4:0] now\_line; //当前输入对应的cacheline

wire [24:0] write\_label; //对应的标记位

assign now\_line = addr[6:2];

assign now\_valid = cacheline[now\_line][57];

assign write\_label = addr[31:7];

assign now\_label = cacheline[now\_line][56:32];

assign r\_data = cacheline[now\_line][31:0];

always @(\*) begin

if(wen) begin

cacheline[now\_line] ={1'b1,write\_label,w\_data};

end

end

//always块，在写指令时更新cache

endmodule

其实在一开始是打算把hit是否命中的判定也放在这个里面的，但是后续调试过程中，如果使用reg赋值，会导致多耽误一个周期（也就是并没有优化时钟周期，显而易见）。使用wire调试的过程中会导致仿真卡死（并没有报错），原因可能是赋值嵌套。最后选择将是否命中的判定拆到mem.v中进行。

**(b)mem.v**

wire wen;

wire [24:0] now\_label;

assign wen = MEM\_allow\_in && inst\_store;

写使能和now\_label的定义，写使能初始化为MEM的第一拍(allow\_in)和指令写。（其实这里也曾经考虑过把读cache miss的情况也加进去，但是最终没能成功）

wire load\_sign;

wire [31:0] load\_result;

wire [31:0] mem\_rdata,cache\_rdata;

wire now\_valid;

wire hit = inst\_load && now\_valid && (now\_label == dm\_addr[31:7]);

定义hit判定条件：读指令，当前cacheline的有效位为1，且地址标签匹配。再定义一些与cache交互的端口。

assign mem\_rdata = (hit) ? cache\_rdata :dm\_rdata;

assign load\_sign = (dm\_addr[1:0]==2'd0) ? mem\_rdata[ 7] :

(dm\_addr[1:0]==2'd1) ? mem\_rdata[15] :

(dm\_addr[1:0]==2'd2) ? mem\_rdata[23] : mem\_rdata[31] ;

assign load\_result[7:0] = (dm\_addr[1:0]==2'd0) ? mem\_rdata[ 7:0 ] :

(dm\_addr[1:0]==2'd1) ? mem\_rdata[15:8 ] :

(dm\_addr[1:0]==2'd2) ? mem\_rdata[23:16] :

mem\_rdata[31:24] ;

assign load\_result[31:8]= ls\_word ? mem\_rdata[31:8] : {24{lb\_sign & load\_sign}};

新定义mem\_rdata，在cache命中的时候使用cache存储的数据，否则使用内存读取的数据。在后续load\_sign和load\_result中，把赋值的dm\_rdata替换为mem\_rdata。

assign MEM\_over = ( inst\_load && !hit) ? MEM\_valid\_r : MEM\_valid;

MEM\_over判定更新，现在只有读指令cache未命中的时候才需要两个周期了，其他都只需要一个周期。

cache cache\_module(

.addr (dm\_addr ),

.wen (wen),

.w\_data (dm\_wdata),

.now\_valid (now\_valid),

.now\_label (now\_label),

.r\_data(cache\_rdata)

);

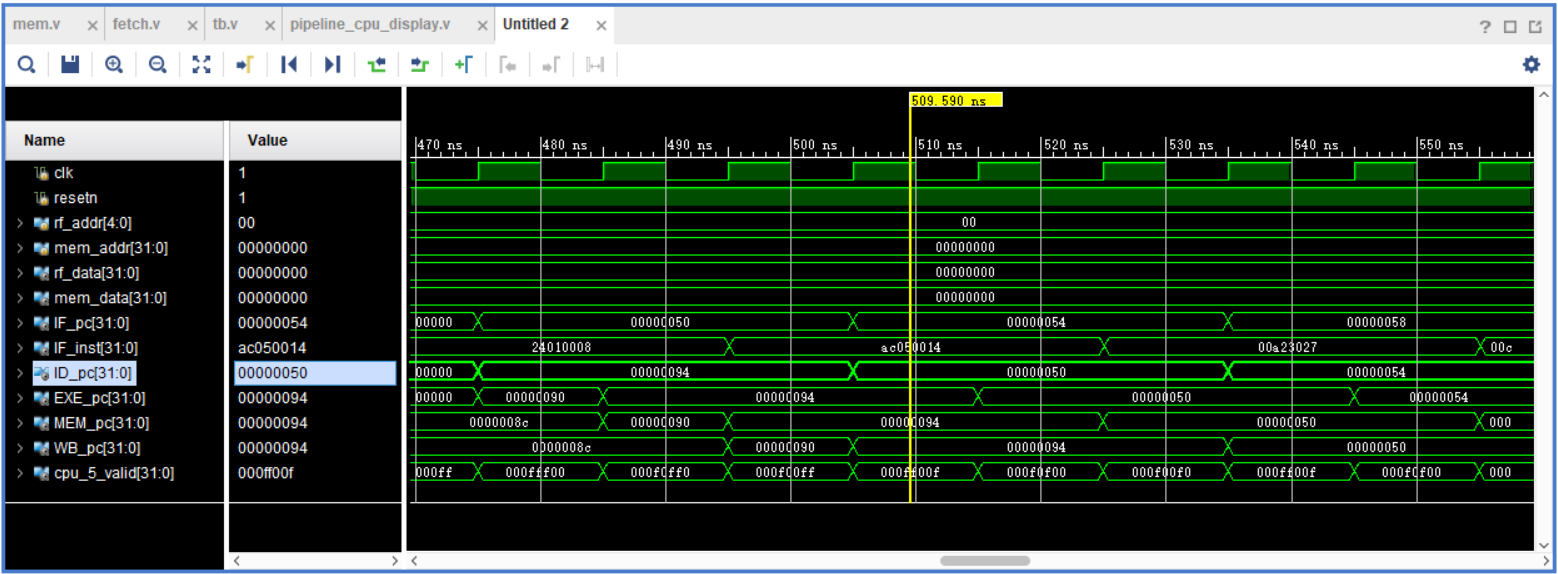
cache模块的调用，在上一部分已经进行了解释。

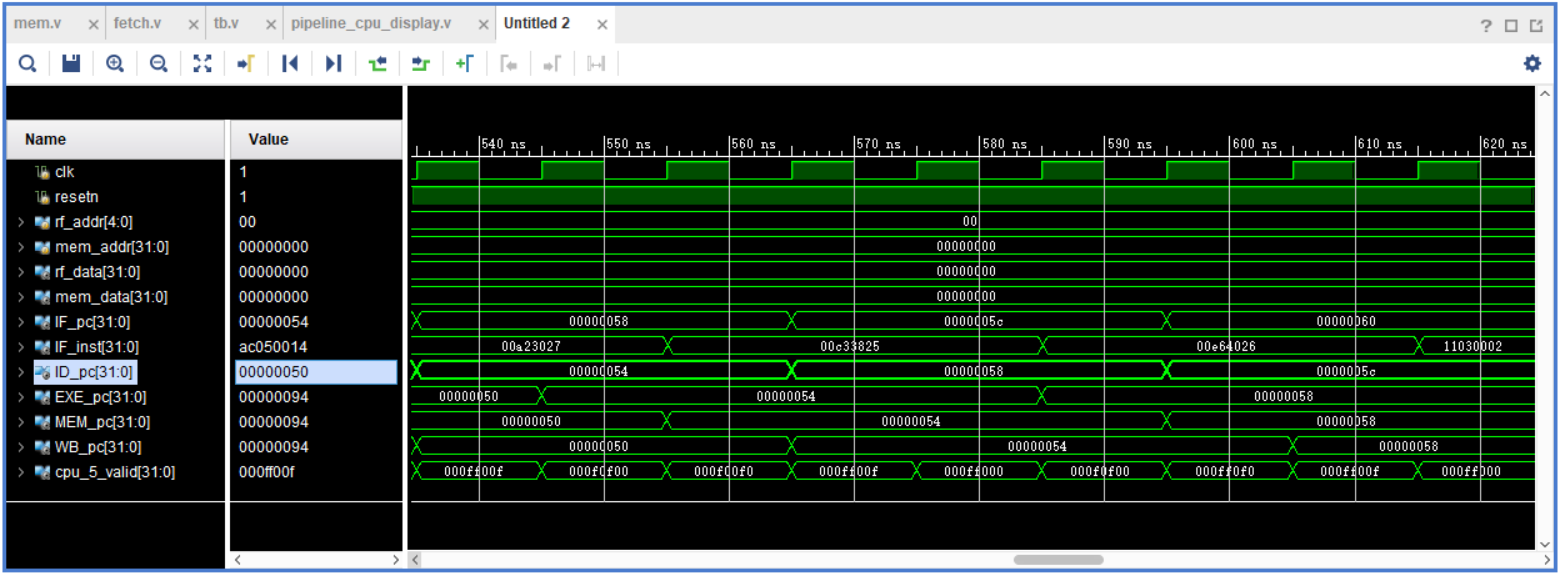
1. **实验结果分析**

（1）前递技术优化对比

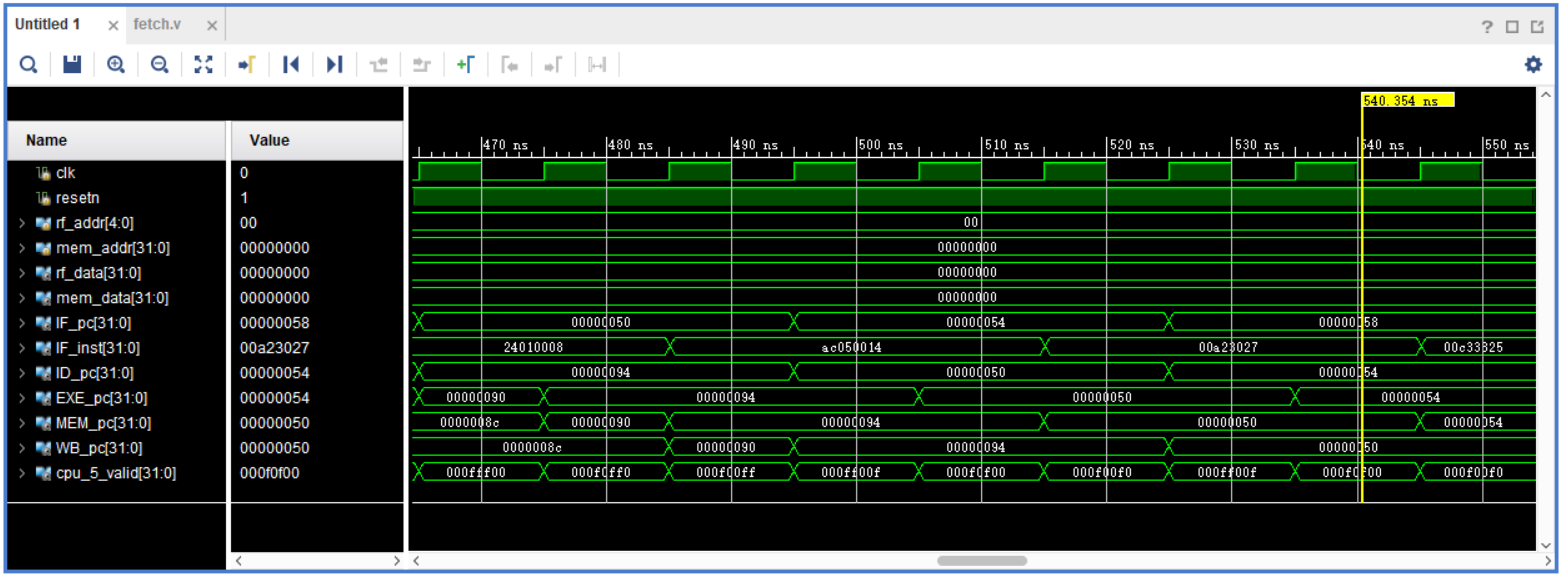
搭建前递前后，对50H-60H这五条指令的分析：

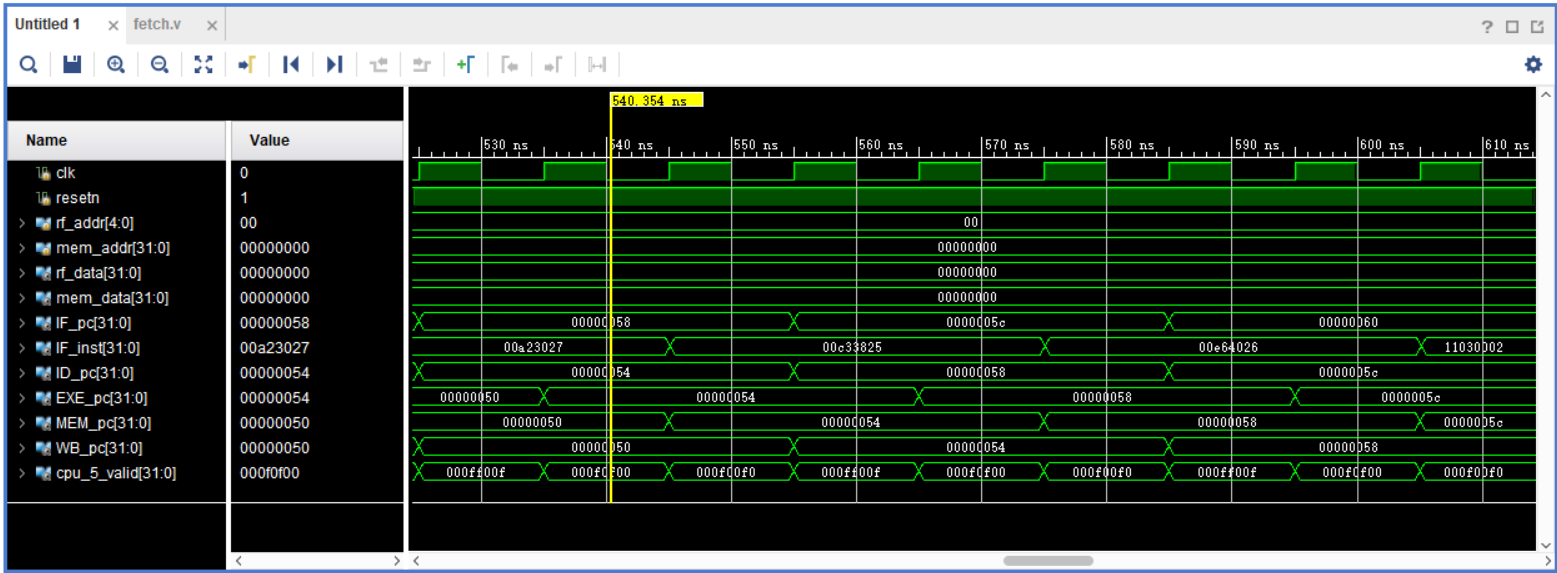
搭建前的仿真结果如下（这一结果也在上一次报告中展示）：



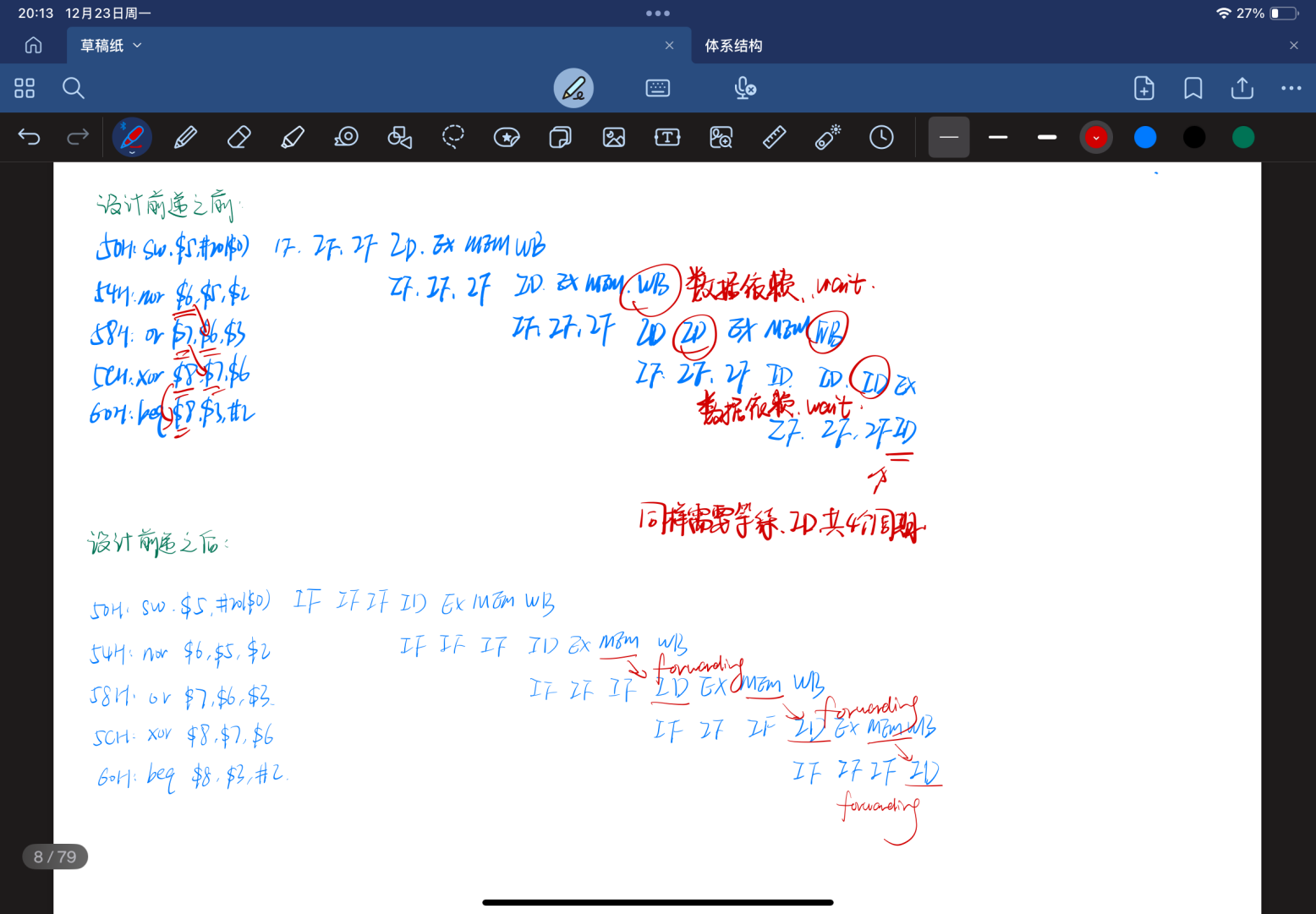


搭建后的仿真结果如下：





分析如下：



设计前递之前，如果是连续的数据依赖，每一条指令的ID都会多一个周期（因为需要等待上一条指令的WB），从而造成CPU执行效率低下。

设计前递之后，由于前递结果的设计所有指令（若不涉及分支跳转，也不涉及读内存）均可在7周期内完成执行。在不涉及分支跳转下，平均每条指令的运行时间为耗时最长的IF阶段的3周期。前递的设计在数据依赖的情况下可以很好的提升CPU的运行性能。

（2）指令集扩展验证

在本部分中，通过我重新设计的.coe文件作为inst\_rom的指令，进行验证。

设计的指令如下：

nand $13,$25,$7 00000|11001|00111|01101|00000|111111，0327683F

subiu $13,$5,#5 111101|00101|01101|0000000000000101，F4AD0005

nori $13,$25,#6 111110|11001|01101|0000000000000110，FB2D0006

jc #34H 111111|00000000000000000000001101，FC00000D

设计文件如下：

; Initialization file for a

; 32-bit wide ROM

memory\_initialization\_radix = 16;

memory\_initialization\_vector =

; EX\_JUMP\_ADDR

24010001

24010001

24010001

24010001

24010001

24010001

24010001

24010001

24010001

24010001

24010001

24010001

24010001 开始地址之前的都用addiu $1, $0,#1 即[$1]=1填充

; START\_ADDR

24010001 34H:addiu $1, $0,#1,[$1]=1

24050011 38H:addiu $5,$0,#17,[$5]=17 以下三条指令是为验证扩展指令准备寄存器

2407AAAA 3CH:addiu $7,$0,#0x4BB4,[$7]=0x4BB4

24190005 40H:addiu $25,$0,#5,[$25]=5

0327683F 44H:nand $13,$25,$7 以下四条指令是扩展指令集的指令

F4AD0005 48H:subiu $13,$5,#5

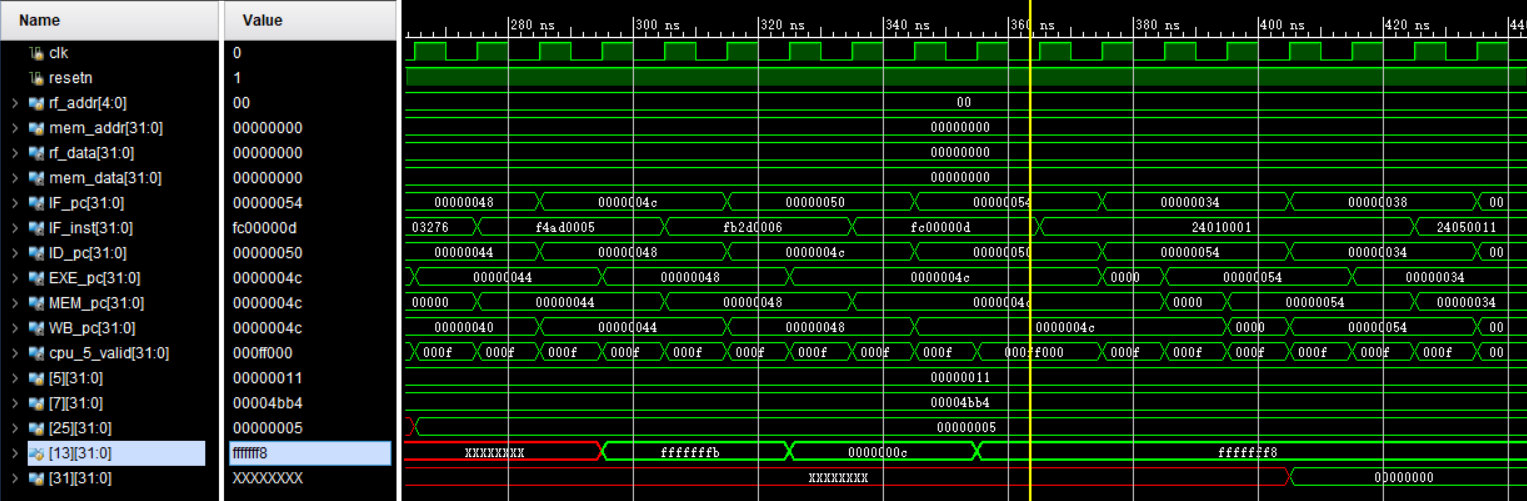
FB2D0006 4CH:nori $13,$25,#6

FC00000D 50H:jc #34H

24010001 由于程序存在跳转之前会多向后执行一条指令的bug，因此多预留两条

24010001 addiu $1, $0,#1 即[$1]=1，以保证稳定运行

仿真结果如下：



44H结束之后，$13的结果为0xfffffffb。

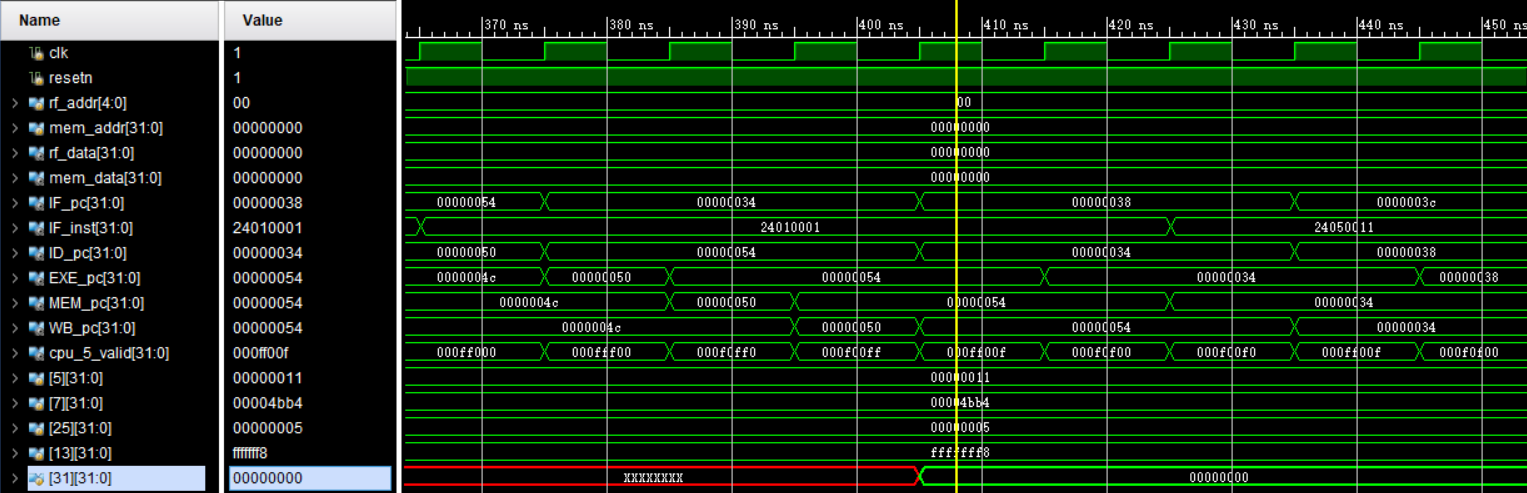
$25为0x00000005，$7为0x00004BB4，两者相与为0x00000004，与非为0xfffffffb，是正确的。

48H结束之后，$13的结果为0x0000000c。

$5为0x00000011=17，17-5=12，因此也是正确的。

4CH结束之后，$13的结果为0xfffffff8。

5和6进行或运算，结果为7，再进行非运算，结果为0xfffffff8，同样是正确的。



50H结束后，跳转到了34H，$31被清零，也是正确的。

至此，所有扩展的指令都得到了正确的验证。

（3）cache优化对比

测试coe文件设计：

在34H之前的指令仍然通过24010001进行填充，不再赘述。

; START\_ADDR

24010001 34H:addiu $1, $0,#1,[$1]=1

24020010 38H:addiu $2,$0,#0x10,[$2]=0x10

24030100 3CH:addiu $3,$0,#0x100,[$3]=0x100

AC010004 40H:sw $1,#4($0) cache(1)更新

AC030000 44H:sw $3,#0($0) cache(0)更新，不影响cache(1)

8C040004 48H:lw $4,#4($0) cache(1) hit!

AC020084 4CH:sw $2,#132($0) #132和#4的映射cache块相同(1)，会换出#4

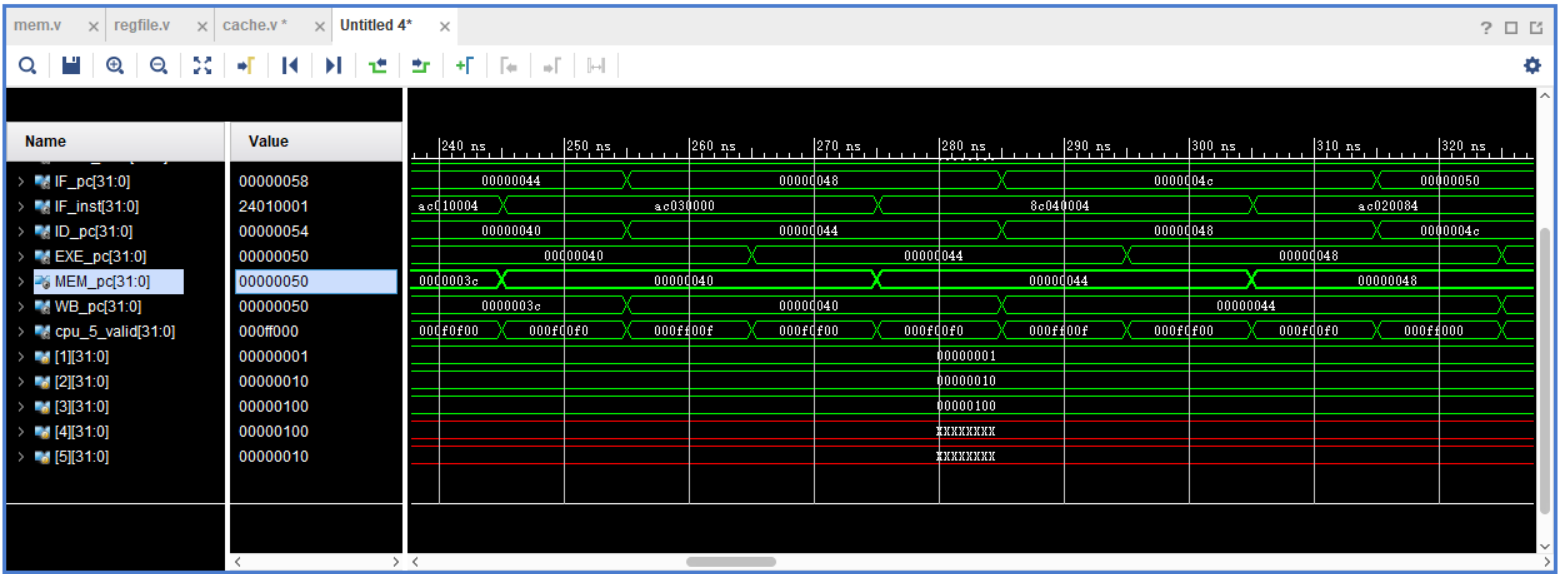
8C050004 50H:lw $5,#4($0) cache(1) miss!

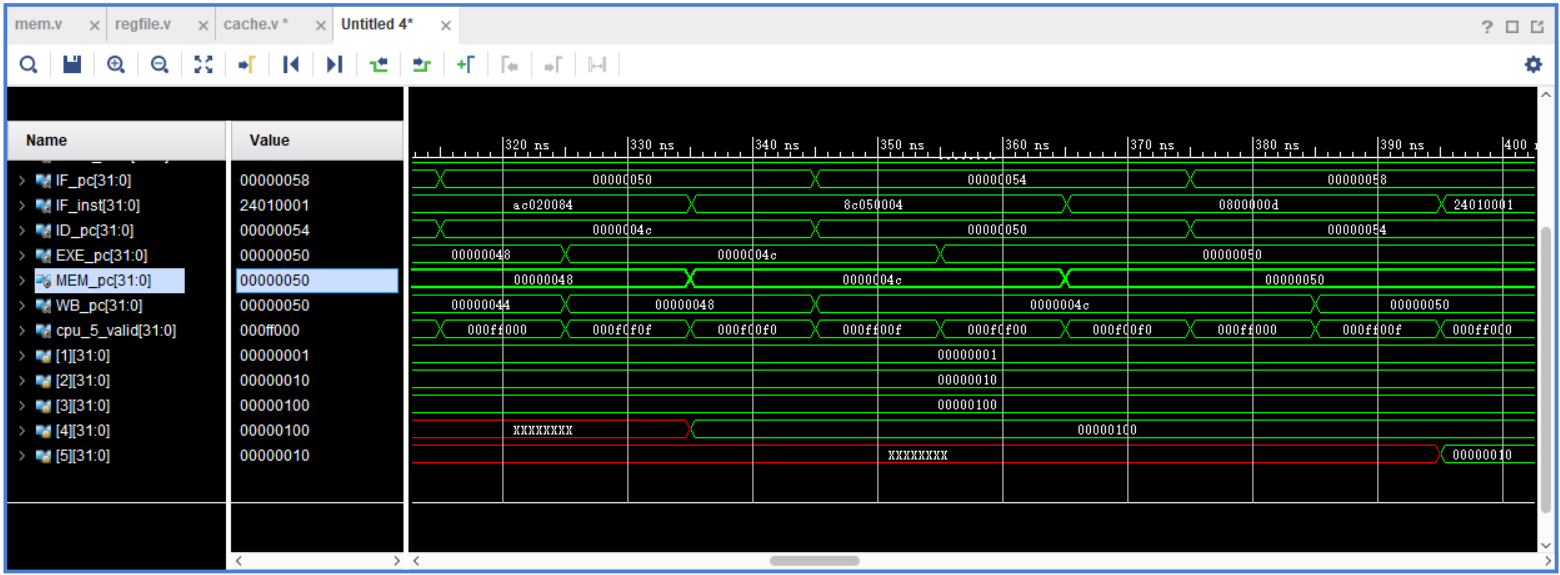
0800000D 54H:j 34H

24010001

24010001

以下是优化之前的仿真结果：





可以看到写指令40H，44H，4CH的MEM阶段都只需要1周期，但48H和50H的读指令的MEM阶段却需要2周期。

以下是优化后的结果：可以看到48H的cache HIT情况需要1周期就可以完成，而50H的cache MISS则还需要2周期完成。



这说明我们的优化是很成功的，以下将对cache的运行过程进行详细分析。（在仿真结果图中也显示了对应信号，便于观察）

$1,$2,$3的初始值分别为0x1，0x10，0x100

**40H:sw $1,#4($0)**

|  |  |  |  |
| --- | --- | --- | --- |
| Cache | valid[57] | addr[56:32] | data[31:0] |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |

对Cache[1]进行了刷新，addr标记为0，更新为寄存器$1的值。

**44H:sw $3,#0($0)**

|  |  |  |  |
| --- | --- | --- | --- |
| Cache | valid[57] | addr[56:32] | data[31:0] |
| 0 | 1 | 0 | 100 |
| 1 | 1 | 0 | 1 |

对Cache[0]进行了刷新，addr标记为0，data更新为寄存器$3的值。

**48H:lw $4,#4($0)**

检查Cache0，发生Cache Hit!

此时Hit为1，直接使用Cache结果，有效节约了访存开销。

**4CH:sw $2,#132($0)**

|  |  |  |  |
| --- | --- | --- | --- |
| Cache | valid[57] | addr[56:32] | data[31:0] |
| 0 | 1 | 0 | 100 |
| 1 | 1 | 1 | 10 |

对Cache[1]进行了刷新，addr标记为1，data更新为寄存器$2的值。

**50H:lw $4,#4($0)**

可以发现，cache[1]存储的数据并不是#4的数据，发生Cache Miss!

此时Hit=0，只能从内存中读取结果。

1. **总结感想**

在本次实验中，我对5级流水线CPU进行了改进，分别完成了前递的设计，指令集的扩展以及cache的设计。在实验过程中，我对流水线CPU有了更进一步的认识，对Verilog语法也加深了认识。通过一次一次的尝试与修改，在仿真中一个一个wire or reg变量的查询和debug我对CPU的运行过程也更加熟悉。

这学期体系结构实验，在先前组成原理实验的基础上，对CPU的运行流程的理解变得更加深刻，与理论课所讲授知识结合，增进了对体系结构的理解。