哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2020 夏季

课程名称：计算机设计与实践

实验名称：多周期 CPU 设计

实验性质：综合设计型

实验学时：42 地点：线上

学生班级：1801101

学生学号：180110115

学生姓名：方澳阳

评阅教师：

报告成绩：

实验与创新实践教育中心制

2020 年 5 月

注：

本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明设计的成果和特色。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

完成多周期 CPU 设计与实现的同学根据多周期设计的内容完成实验报告。

设计的功能描述（含所有实现的指令描述及模块的功能）

指令描述:

R-型指令

1 add

指令名：加法指令

汇编格式：add rd, rs, rt

汇编举例：add $4, $2, $3

功能描述：rd ← rs + rt; PC ← NPC (PC + 4)。 32 位整数加法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。由 于本设计无溢出检测，因此该指令功能同 ADDU。

2 addu

指令名: 无符号数加法指令

汇编格式：addu rd, rs, rt

汇编举例：addu $4, $2, $3

功能描述：rd ← rs + rt; PC ← NPC (PC + 4)。 32 位无符号整数加法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

3 sub

指令名：减法指令

汇编格式：sub rd, rs, rt

汇编举例：sub $4, $2, $3

功能描述：rd ← rs - rt; PC ← NPC (PC + 4)。 32 位整数减法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

4 subu

指令名：无符号减法指令

汇编格式：subu rd, rs, rt

汇编举例：subu $4, $2, $3

功能描述：rd ← rs - rt; PC ← NPC (PC + 4)。 32 位无符号整数减法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

5 and

指令名： 逻辑与

汇编格式：and rd, rs, rt

汇编举例：and $4, $2, $3

功能描述：rd ← rs and rt; PC ← NPC (PC + 4)。 32 位数按位逻辑与，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

6 or

指令名： 逻辑或

汇编格式：or rd, rs, rt
汇编举例：or $4, $2, $3
功能描述：rd ← rs or rt; PC ← NPC (PC + 4)。 32 位数按位逻辑或，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

7 xor
指令名： 逻辑异或
汇编格式：xor rd, rs, rt
汇编举例：xor $4, $2, $3
功能描述：rd ← rs xor rt; PC ← NPC (PC + 4)。 32 位数按位逻辑异或，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

8 nor
指令名： 逻辑或非
汇编格式：nor rd, rs, rt
汇编举例：nor $4, $2, $3
功能描述：rd ← rs nor rt; PC ← NPC (PC + 4)。 32 位数按位逻辑或非，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

9 slt
指令名: 小于则设置指令
汇编格式：slt rd, rs, rt
汇编举例：slt $4, $2, $3
功能描述： if ((rs)<(rt)) then (rd)←1; else (rd)←0; PC ← NPC (PC + 4)。如果 rs 的值小于 rt 值，则设置 rd 为 1，否则 rd 为 0。

10 sltu
指令名: 无符号小于则设置指令
汇编格式：sltu rd, rs, rt
汇编举例：sltu $4, $2, $3
功能描述： if ((rs)<(rt)) then (rd)←1; else (rd)←0; PC ← NPC (PC + 4)。无符号数小于判断，如果 rs 的值小于 rt 值，则设置 rd 为 1，否则 rd 为 0。

11 sll
指令名: 逻辑左移
汇编格式：sll rd, rt, shamt
汇编举例：sll $4, $2, 10
功能描述： (rd)←(rt)<<shamt; PC ← NPC (PC + 4)。逻辑左移，将 rt 寄存器中的 32 位数逻辑左移后赋给 rd，低位用 0 填充，移位的位数是 shamt。

12 srl
指令名: 逻辑右移
汇编格式：srl rd, rt, shamt
汇编举例：srl $4, $2, 10

功能描述： (rd)←(rt)>>shamt; PC ← NPC (PC + 4)。逻辑右移，将 rt 寄存器中的 32 位数逻辑右移后赋给 rd，移位的位数是 shamt。80000000H 逻辑右移 1 位的结果是 40000000H。

13 sra
指令名: 算术右移
汇编格式：sra rd, rt, shamt
汇编举例：sra $4, $2, 10
功能描述： (rd)←(rt)>>shamt; PC ← NPC (PC + 4)。算术右移，将 rt 寄存器中的 32 位数算术右移后赋给 rd，移位的位数是 shamt。算术右移时，符号位不仅要参与移位，还要保留，如 80000000H 算术右移 1 位的结果是 0C0000000H。

14 sllv
指令名: 按寄存器值逻辑左移指令
汇编格式：sllv rd, rt, rs
汇编举例：sllv $4, $2, $3
功能描述： (rd)←(rt)<<(rs); PC ← NPC (PC + 4)。按寄存器值逻辑左移指令，将 rt 寄存器中的 32 位数逻辑左移后赋给 rd，低位用 0 填充，移位的位数在 rs 寄存器中。

15 srlv
指令名: 按寄存器值逻辑右移指令
汇编格式：srlv rd, rt, rs
汇编举例：srlv $4, $2, $3
功能描述： (rd)←(rt)>>(rs); PC ← NPC (PC + 4)。按寄存器值逻辑右移指令，将 rt 寄存器中的 32 位数逻辑右移后赋给 rd，移位的位数在 rs 寄存器中。

16 srav
指令名: 按寄存器值算术右移指令
汇编格式：srav rd, rt, rs
汇编举例：srav $4, $2, $3
功能描述： (rd)←(rt)>>(rs); PC ← NPC (PC + 4)。按寄存器值算术右移指令，将 rt 寄存器中的 32 位数算术右移后赋给 rd，移位的位数在 rs 寄存器中。算术右移时，符号位不仅要参与移位，还要保留。

17 jr
指令名: 按寄存器内容转移指令
汇编格式：JR rs
汇编举例：JR $31 功能描述：(PC)←(rs); PC ← NPC (PC + 4)。将 rs 寄存器的内容当地址，赋给 PC，从而完成转移，通常可做过程返回语句。实际系统中只用了低 16 位地址线。

I-型指令

18 addi
指令名：有符号立即数加法指令
汇编格式：ADDI rt, rs, immediate
汇编举例：ADDI $4, $2, -100
功能描述：(rt)←(rs) + (Sign-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位有符号立即数扩展到 32 位，然后加上 rs 中的数，结果给 rt 寄存器。 如果结果溢出会产生内部异常中断，本设计不支持异常处理。

19 addiu
指令名：无符号立即数加法指令
汇编格式：ADDIU rt, rs, immediate
汇编举例：ADDIU $4, $2, -100
功能描述：(rt)←(rs) + (Sign-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位有符号立即数扩展到 32 位，然后加上 rs 中的数，结果给 rt 寄存器。 与 ADDI 的不同是不会因溢出而产生内部异常中断，本设计未实现内部异常处理，因此次指令与 ADDI 相同。

20 andi
指令名：立即数逻辑与指令
汇编格式：ANDI rt, rs, immediate
汇编举例：ADDI $4, $2, 1 功能描述：(rt)←(rs) AND (Zero-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位立即数零扩展到 32 位，然后同 rs 中的数按位逻辑与，结果给 rt 寄存 器。

21 ori
指令名：立即数逻辑或指令
汇编格式：ORI rt, rs, immediate
汇编举例：ORI $4, $2, 5 功能描述：(rt)←(rs) ORI (Zero-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位立即数零扩展到 32 位，然后同 rs 中的数按位逻辑或，结果给 rt 寄存 器。

22 xori
指令名：立即数逻辑异或指令
汇编格式：XORI rt, rs, immediate
汇编举例：XORI $4, $2, 5 功能描述：(rt)←(rs) XORI (Zero-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位立即数零扩展到 32 位，然后同 rs 中的数按位逻辑异或，结果给 rt 寄 存器。

23 sltiu
指令名：小于无符号立即数则设置指令
汇编格式：SLTIU rt, rs, immediate
汇编举例：SLTIU $3, $2, 10
功能描述：if ((rs)<(sign_extend)immediate) then (rt)←1; else (rt)←0; PC ← NPC (PC + 4)。如果 rs 的值小于立即数 immediate 值（进行的是符号扩展），则设

置 rt 为 1，否则 rt 为 0。

24 lui
指令名：立即数赋值指令
汇编格式：LUI rt, immediate
汇编举例：LUI $2, 10
功能描述：：(rt)←immediate<<16 & 0FFFF0000H 即(rt)←immediate×65536; PC ← NPC (PC + 4)。首先 16 位立即数赋给 rt 寄存器的高 16 位，低 16 位用 0 填充。也就是将 16 位立 即数乘以 65536 后赋值给 rt 寄存器。

25 lw
指令名：存储器读（字操作）
汇编格式：LW rt, offset(rs)
汇编举例：LW $3, 10($2) 或 LW $3, buff($2)
功能描述：(rt)←Memory[(rs)+(sign_extend)offset]; PC ← NPC (PC + 4)。以 rs 寄存器的内容为基地址，offset 通过符号扩展后形成 32 位的偏移，将基地址 加上偏移形成一个 32 位的地址，以此地址从 RAM 中读出一个字（4 字节）赋给 rt 寄存器。本系统中只使用了低 16 位地址，汇编中，offset 可以是变量名。

26 sw
指令名：存储器写（字操作）
汇编格式：SW rt, offset(rs)
汇编举例：SW $3, 10($2)
功能描述：Memory[(rs)+(sign_extend)offset]←(rt); PC ← NPC (PC + 4)。以 rs 寄存器的内容为基地址，offset 通过符号扩展后形成 32 位的偏移，将基地址 加上偏移形成一个 32 位的地址，将 rt 寄存器的内容写入到 RAM 中该地址开始的一个字（4 字节）单元。本系统中只使用了低 16 位，汇编中，offset 可以是变量名。这是唯 一一个源操作数做第一操作数的指令。

27 beq
指令名：相等则转移指令
汇编格式：BEQ rt, rs, immediate
汇编举例：BEQ $3, $2, 10
功能描述：if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2); else PC ← NPC (PC + 4)。如果 rt 和 rs 的值相等，则转移到新的地址。新地址是当前指令的下一条指令地址（PC+4）加上一个 32 位偏移量。该 32 位偏移量是将 16 位 offset 符号扩展到 32 位，然后左移 2 位（即乘 4）后取低 32 位得到。实际系统中只用了低 16 位地址线。

28 bne
指令名：不相等则转移指令
汇编格式：BNE rt, rs, immediate
汇编举例：BNE $3, $2, 10
功能描述：if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2); else PC ←

NPC (PC + 4)。如果 rt 和 rs 的值不等，则转移到新的地址。新地址是当前指令的下一条指令地址（PC+4）加上一个 32 位偏移量。该 32 位偏移量是将 16 位 offset 符号扩展到 32 位，然后左移 2 位（即乘 4）后取低 32 位得到。实际系统中只用了低 16 位地址线。

29 bgtz
指令名：大于 0 转移指令
汇编格式：BGTZ rs, offset
汇编举例：BGTZ $1, 40
功能描述：if ((rs)>0) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1; else PC ← NPC (PC + 4)。如果 rs 的值大于 0 则跳转到指定分支。

J-型指令
30 j
指令名：无条件转移指令
汇编格式：J target
汇编举例：J 100
功能描述：(PC)←((Zero-Extend) address<<2); PC ← NPC (PC + 4)。无条件转移到新的地址。新地址是 26 位 address 零扩展到 32 位，然后左移 2 位（即 乘 4）后取低 32 位得到。实际系统中只用了低 16 位地址线。要注意指令中的 address 是汇编语句中操作数 target 除以 4 的结果。在做 CPU 设计或汇编（编译）器设计的时候都要注意这一点。

31 jal
指令名：过程调用指令
汇编格式：JAL target
汇编举例：JAL 100
功能描述：($31)←(PC)+4; (PC)←((Zero-Extend)address<<2); PC ← NPC(PC+4)。先将下条指令的地址（(PC)+4）保存在$31（$ra）作为过程的返回地址，然后无条 件转移到新的地址。新地址是 26 位 address 零扩展到 32 位，然后左移 2 位（即乘 4） 后取低 32 位得到。实际系统中只用了低 16 位地址线。要注意指令中的 address 是汇编 语句中操作数 target 除以 4 的结果。

模块功能：

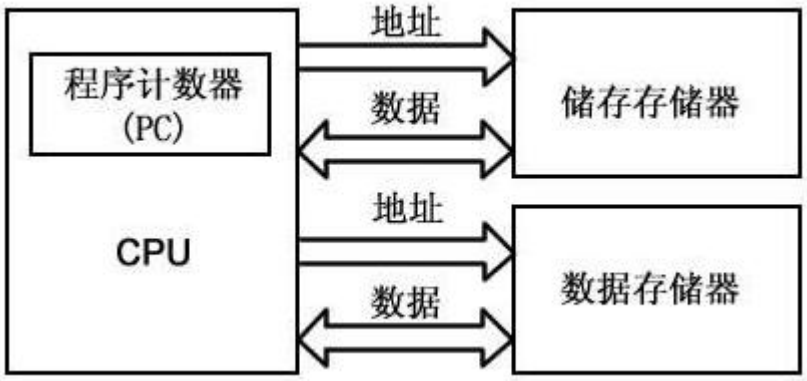
1. minisys
顶层模块，负责所有模块的实现，以及数据通路的连接。
2. cpuclk
时钟模块，负责将 100MHz 时钟信号分频为 23MHz。调用 ip 核实现。
3. dmemory32
数据存储器模块，可以存放 64KB 的数据。
4. ifetc32
取指模块，内部实现了一个 64KB 的指令存储器.
5. Reg_mux
寄存器模式选择模块，根据 CU 给出的寄存器模式，为寄存器堆选择合适的输入端口.
6. regfile
寄存器堆，有 32 个寄存器,可以同时读出两个数据，或写入一个数据
7. ALU_mux
ALU 模式选择模块，根据 CU 给出的 ALU 模式，为寄存器堆选择合适的操作数
8. ALU
计算模块，可以执行加减、移位、或与异或等操作。
9. NPC
计算下一个地址是什么。
10. extend
有符号和无符号数的拓展模块。
11. control32
控制器模块，根据指令需要执行的阶段分为了 5 个阶段。取指、译码、执行、访存、写回。
由一个具有 5 个状态的状态机实现。

设计的主要特色

1. 实现了多周期的 CPU 设计。
2. 在多周期的基础上完成了拓展的指令，即实现了 31 条指令。
3. 通过了 31 条指令测试包的测试。

设计的体系结构

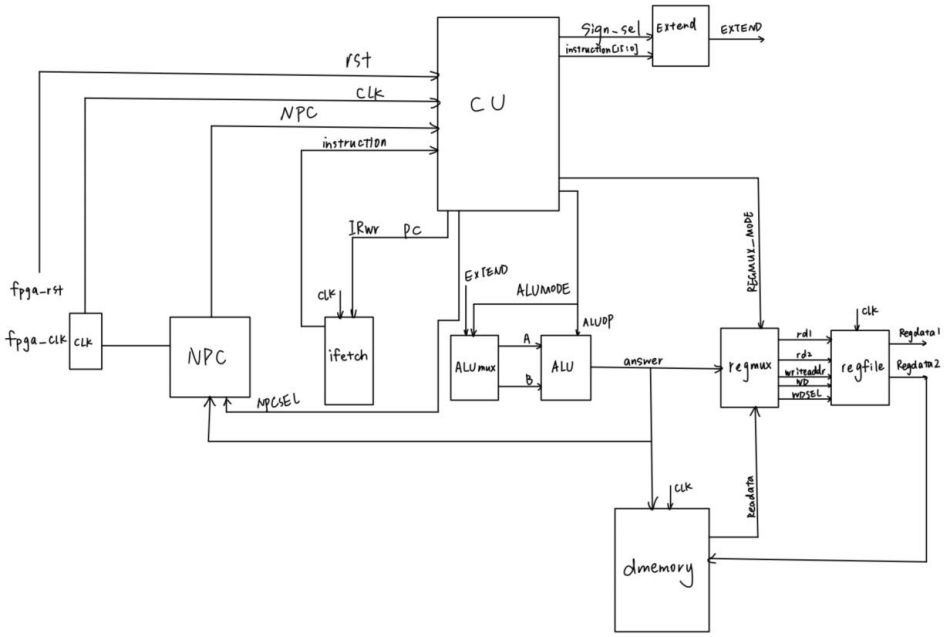
（包括体系结构框图和对结构图的简要解释）



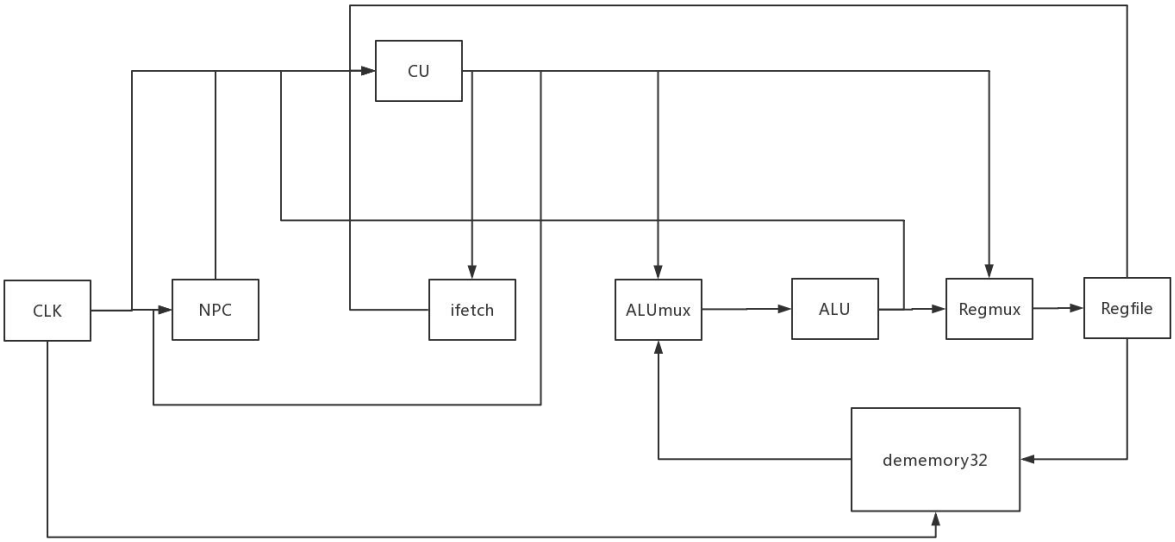
使用两个独立的存储器模块，分别存储指令和数据。
每个存储模块都不允许指令和数据并存，以便实现并行处理。
具有一条独立的地址总线和一条独立的数据总线，利用公用地址总线访问两个存储模块（程序存储模块和数据存储模块），公用数据总线则被用来完成程序存储模块或数据存储模块与 CPU 之间的数据传输；
哈佛结构的微处理器具有较高的执行效率，其程序指令和数据指令分开组织和存储的，执行时可以预先读取下一条指令。
哈佛结构较复杂，对外围设备的连接与处理要求高，不适合外围存储器的扩展。

数据通路设计

（包含数据通路主要部件说明及数据通路表）



简化版数据通路



数据通路主要由 10 个模块组成，分别是 CLK, NPC, ifetch, ALUmux, ALU, REGmux, Regfile,dmemory32, Extend,CU.
模块功能已经在上面陈述，同时还有一些控制信号，例如输入给 minisys 的 fpga_rst 和 fpga_clk。

reg_mux 以及 ALU_mux 主要是为了减少 CU 的体积，将一部分组合电路移到 CU 之外，CU 通过较少的控制信号，可以控制这些 mux 产生对应指令需要的信号。

数据通路表如下：

	所属单元	取指单元					
	部件	PC	NPC			IM（指令存储器）	IR
	输入信号	DI	PC	Imm	RA（寄存器）	A	
R型指令	add	NPC.NPC	PC.DO			PC.DO	IM
	addu	NPC.NPC	PC.DO			PC.DO	IM
	sub	NPC.NPC	PC.DO			PC.DO	IM
	subu	NPC.NPC	PC.DO			PC.DO	IM
	and	NPC.NPC	PC.DO			PC.DO	IM
	or	NPC.NPC	PC.DO			PC.DO	IM
	xor	NPC.NPC	PC.DO			PC.DO	IM
	nor	NPC.NPC	PC.DO			PC.DO	IM
	slt	NPC.NPC	PC.DO			PC.DO	IM
	sltu	NPC.NPC	PC.DO			PC.DO	IM
	sll	NPC.NPC	PC.DO			PC.DO	IM
	srl	NPC.NPC	PC.DO			PC.DO	IM
	sra	NPC.NPC	PC.DO			PC.DO	IM
	sllv	NPC.NPC	PC.DO			PC.DO	IM
	srlv	NPC.NPC	PC.DO			PC.DO	IM
	srav	NPC.NPC	PC.DO			PC.DO	IM
	jr	NPC.NPC	PC.DO		RF.RD1	PC.DO	IM
I型指令	addi	NPC.NPC	PC.DO			PC.DO	IM
	addiu	NPC.NPC	PC.DO			PC.DO	IM
	andi	NPC.NPC	PC.DO			PC.DO	IM
	ori	NPC.NPC	PC.DO			PC.DO	IM
	xori	NPC.NPC	PC.DO			PC.DO	IM
	sltiu	NPC.NPC	PC.DO			PC.DO	IM
	lui	NPC.NPC	PC.DO			PC.DO	IM
	lw	NPC.NPC	PC.DO			PC.DO	IM
	sw	NPC.NPC	PC.DO			PC.DO	IM
	beq	NPC.NPC	PC.DO	IR[15:0]		PC.DO	IM
	bne	NPC.NPC	PC.DO	IR[15:0]		PC.DO	IM
	bgtz	NPC.NPC	PC.DO	IR[15:0]		PC.DO	IM
J型指令	j	NPC.NPC	PC.DO	IR[25:0]		PC.DO	IM
	jal	NPC.NPC	PC.DO	IR[25:0]		PC.DO	IM
综合		NPC.NPC	PC.DO	IR[15:0] IR[25:0]	RF.RD1	PC.DO	IM

译码单元							
RF（寄存器堆）				A	B	S_EXT	E
A1（读出1）	**A2（读出2）**	**A3（写入）**	**WD（写回值）**				
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
	IR[20:16]	IR[15:11]	ALU.C		RF.RD2		
	IR[20:16]	IR[15:11]	ALU.C		RF.RD2		
	IR[20:16]	IR[15:11]	ALU.C		RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]	IR[15:11]	ALU.C	RF.RD1	RF.RD2		
IR[25:21]							
IR[25:21]		IR[20:16]	ALU.C	RF.RD1		IR[15.0]	EXT.Ext
IR[25:21]		IR[20:16]	ALU.C	RF.RD1		IR[15.0]	EXT.Ext
IR[25:21]		IR[20:16]	ALU.C	RF.RD1		IR[15.0]	EXT.Ext
IR[25:21]		IR[20:16]	ALU.C	RF.RD1		IR[15.0]	EXT.Ext
IR[25:21]		IR[20:16]	ALU.C	RF.RD1		IR[15.0]	EXT.Ext
IR[25:21]		IR[20:16]	ALU.C	RF.RD1		IR[15.0]	EXT.Ext
		IR[20:16]	ALU.C			IR[15.0]	EXT.Ext
IR[25:21]		IR[20:16]	DM.RD	RF.RD1		IR[15.0]	EXT.Ext
IR[25:21]	IR[20:16]			RF.RD1	RF.RD2	IR[15.0]	EXT.Ext
IR[25:21]	IR[20:16]			RF.RD1	RF.RD2		
IR[25:21]	IR[20:16]			RF.RD1	RF.RD2		
IR[25:21]				RF.RD1			
		0x1F	NPC.PC4	RF.RD1			
IR[25:21]	IR[20:16]	IR[15:11]	ALUout	RF.RD1	RF.RD2	IR[15.0]	EXT.Ext
		IR[20:16]	DR				
		0x1F	NPC.PC4				

执行单元			数据存储器	
ALU		ALUOut	DM（数据存储器）	DR
A	B		A	
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
IR[10:6]	B	ALU.C		
IR[10:6]	B	ALU.C		
IR[10:6]	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	B	ALU.C		
A	E	ALU.C		
A	E	ALU.C		
A	E	ALU.C		
A	E	ALU.C		
A	E	ALU.C		
A	E	ALU.C		
0x10H	E	ALU.C		
A	E	ALU.C	ALUout	DM.RD
A	E	ALU.C	ALUout	
A	B	ALU.C		
A	B	ALU.C		
A	0x0H	ALU.C		
A	B	ALU.C	ALUout	DM.RD
IR[10:6]	E			
0x10H	0x0H			

各部件的设计与实现

（含模块功能，输入、输出信号及变量定义，关键代码等。控制器设计包含控制信号表）

1. minisys

顶层模块，负责所有模块的实现，以及数据通路的连接。

输入： fpga_rst, 复位信号

fpga_clk 100MHz 时钟信号

2. cpuclk

时钟模块, 负责将 100MHz 时钟信号分频为 23MHz。调用 ip 核实现。

输入： clk_in1, 输入的 100MHz 时钟信号

输出： clk_out1, 输出 23MHz 时钟信号

3. dmemory32

数据存储器模块, 可以存放 64KB 的数据。

输入：

address, 32 位的地址, 可以是写入的地址, 也可以是读出的地址, 具体视控制信号 Memwrite 而定.

write_data, 32 位写入的数据

Memwrite, 写使能控制信号, 高电平为写入, 低电平为读出

clock, 输入的时钟信号

输出:

read_data, 读出的 32 位数据.

4. ifetc32

取指模块, 内部实现了一个 64KB 的指令存储器.

输入:

clock, 时钟信号

IRWr, 读指令到寄存器使能信号

PC, 欲读取的指令的地址

输出:

IR, 读取出来的指令

5. Reg_mux

寄存器模式选择模块, 根据 CU 给出的寄存器模式, 为寄存器堆选择合适的输入端口.

输入:

mode, 从 CU 过来的选择信号, 选择不同的寄存器模式, 即是否写、写地址是什么、写数据是什么

zero_g, 从 ALU 过来的信号, 判断计算结果的情况, 有大于\小于\等于三个状态

IMD20_16, 指令中第 16:20 位的 5 位地址

IMD15_11, 指令中第 11:15 位的 5 位地址

WD_ALU, 来自 ALU 计算出来的数据

WD_DMRD, 来自数据存储器读出来的数据

WD_PC4, 来自 NPC 的当前 pc+4 的数据

输出:

Writeaddr，写地址
WD，写数据
WDsel，读写控制信号

关键代码:

```
always@*
    begin
        case(mode)
        `REG_MODE0: begin WriteAddr = IMD15_11; WD = WD_ALU; WDsel = 1; end
        `REG_MODE1: begin WriteAddr = IMD20_16; WD = (zero_g==`SMALLER) ? 32'b1:32
'b0; WDsel = 1; end
        `REG_MODE2: begin WriteAddr = IMD15_11; WD = (zero_g==`SMALLER) ? 32'b1:32
'b0; WDsel = 1; end
        `REG_MODE3: begin WriteAddr = IMD20_16; WD = WD_ALU; WDsel = 1; end
        `REG_MODE4: begin WriteAddr = IMD20_16; WD = WD_DMRD; WDsel = 1; end
        `REG_MODE5: begin WriteAddr = 5'b11111; WD = WD_PC4; WDsel = 1; end  //$31
        `REG_MODE6: begin WDsel = 0; end
        endcase
    end
```

6. regfile
寄存器堆，有 32 个寄存器,可以同时读出两个数据，或写入一个数据
输入:
clk，时钟
fpga_rst，复位信号，用于清空寄存器堆
rd1，第一个读寄存器的地址
rd2，第二个读寄存器的地址
WriteAddr，从 reg_mux 模块过来的写地址
WD，从 reg_mux 模块过来的写数据
WDsel，从 reg_mux 模块过来的写使能信号

输出:
regdata1，第一个读寄存器读出的数据
regdata2，第二个读寄存器读出的数据

关键代码:

```
reg [31:0] imem[31:0]; //32

integer i=0; //be careful, i should be integer!
always @(negedge clk)
begin
    Regdata1 <= imem[rd1];
    Regdata2 <= imem[rd2];
```

```
end
always @(posedge clk)
begin
    imem[WriteAddr] <= WDsel?WD:imem[WriteAddr];

    if(fpga_rst)
    begin
        for(i = 0;i <= 31;i = i+1)
        begin
            imem[i]<=0;
        end
    end
end
```

7. ALU_mux

ALU 模式选择模块，根据 CU 给出的 ALU 模式，为寄存器堆选择合适的操作数

输入:

ALU_MODE, 从 CU 过来的选择信号，选择不同的 ALU 模式，即操作数的什么

RF_RD1，从寄存器堆读出的第一个数据

RF_RD2，从寄存器堆读出的第二个数据

IM_10_6，指令的 6:10 位，shamt 数据

EXTEND，拓展后的数据

输出:

OP_1，第一个操作数

OP_2，第二个操作数

关键代码:

```
    always@*
    begin
        case(ALU_MODE)
        `ALU_MODE0: begin OP_1 = RF_RD1; OP_2 = RF_RD2; end
        `ALU_MODE1: begin OP_1 = IM_10_6; OP_2 = RF_RD2; end
        `ALU_MODE2: begin OP_1 = RF_RD1; OP_2 = EXTEND; end
        `ALU_MODE3: begin OP_1 = 32'h10; OP_2 = EXTEND; end
        `ALU_MODE4: begin OP_1 = RF_RD1; OP_2 = 32'b0; end
        endcase
    end
```

8. ALU

计算模块

输入:
clk，时钟信号
ALUop，操作类型选择信号
first，第一个操作数
second，第二个操作数

输出:
answer，执行运算后的答案
zero，答案是大于还是小于还是等于 0.

关键代码:

```
assign zerog = ans==0?`EQUAL:((ans[31]==0)?`GREATER:`SMALLER);

always@*
begin
    case (ALUop)
        `ADD:
            ans = first+second;
        `SUB:
            ans = first-second;
        `SLT:
            ans = ($signed(first)) < ($signed(second)) ? 32'hffffffff:32'h1;
        `SLL:
            ans = second<<first;
        `SRL:
            ans = second>>first;
        `SRA:
            ans = ($signed(second))>>>first;
        `AND:
            ans = first&second;
        `OR:
            ans = first|second;
        `XOR:
            ans = first^second;
        `NOR:
            ans = ~(first|second);
    endcase
end
```

9. NPC
计算下一个地址是什么

输入:
clk, 时钟信号
rst, 复位信号
ModelSel, 模式选择信号, CU 给出的控制信号, 选择是哪一种模式
PC, 当前的地址
Offset, 从寄存器堆读出的地址
Instruction, 当前的指令
zerog, ALU 运算结果是大于小于还是等于 0 的信号

输出:
npc, 下一个 PC 的值
PC4, 当前 PC+4 的值.

代码:

```
always @*
    begin
        if(rst)
        begin
            Npc = 0;
        end
        else
        begin
            case (ModeSel)
            `Jr:begin
                Npc = Offset;
            end
            `J_Jal:begin
                Npc = {4'b0,Instruction[25:0],2'b0};
            end
            `Beq:begin
                Npc = zerog == `EQUAL? PC+(pc_ext<<2) : PC+4;
            end
            `Bne:begin
                Npc = zerog == `EQUAL? PC+4 : PC+(pc_ext<<2);
            end
            `Bgtz:begin
                Npc = zerog == `GREATER? PC+(pc_ext<<2):PC+4;
            end
            `Normal: Npc = PC+4;
        endcase
        end

    end
```

10. extend
有符号和无符号拓展模块
输入:
imm16, 16 位的数据
Sign_sel, 有符号选择信号

输出
ext: 32 位的拓展后的数据
代码:

```
    wire [31:0]imm32_s;
    wire [31:0]imm32_u;

    assign ext = Sign_Sel ? imm32_s : imm32_u;
    assign imm32_u = {16'h0000,imm16};
    assign imm32_s = (imm16[15]==1)?{16'hffff,imm16}:{16'h0000,imm16};
```

11. control32
控制器模块, 根据指令需要执行的阶段分为了 5 个阶段。取指、译码、执行、访存、写回。由一个具有 5 个状态的状态机实现。
输入:
input clk, 时钟信号
input rst, 复位信号
input [31:0]instruction, 取出的指令
input [31:0]NPC, 下一条指令的地址
输出:
output reg[31:0]PC_curr, 当前的 pc 值
output reg[3:0]ALUop, alu 操作符
output reg[2:0]ALU_MODESEL, alu 操作选择信号
output reg[2:0]reg_mux_mode, 寄存器堆选择信号
output reg[2:0]NPC_SEL, npc 选择信号
output reg Memwrite, 存储器写信号
output reg Sign_Sel, 有符号拓展信号
output reg IRWrsel, 读指令使能信号
output [31:0]PC_pre 输出给 debug_pc 的信号
关键代码:

```
always@(negedge clk)
begin
    PC_curr <= rst?0:(select_npc?NPC:PC_curr);
end
always@ *
begin
    case(curr_state)
```

```
        `sifetch:       begin next_state=`sidecode;                         end
        `sidecode:
        begin
            case(op)
            6'b000000:  begin next_state = (funct==`ins_jr)?`sifetch:`sexecute; end
            `ins_j:     begin next_state = `sifetch;                        end
            `ins_jal:   begin next_state = `swb;                            end
            default:    begin next_state = `sexecute;                       end
            endcase
        end
        `sexecute:
        begin
            case(op)
            `ins_beq:   next_state = `sifetch;
            `ins_bne:   next_state = `sifetch;
            `ins_lw:    next_state = `smem;
            `ins_sw:    next_state = `smem;
            default:    next_state = `swb;
            endcase
        end
        `smem:
        begin
            case(op)
            `ins_lw:    next_state = `swb;
            `ins_sw:    next_state = `sifetch;
            default:    next_state = `sifetch;
            endcase
        end
        `swb:    begin  next_state = `sifetch; end
        default: begin  next_state = `sifetch; end
    endcase
end
always@(posedge clk)
begin
    curr_state <= rst?0:next_state;
end
reg flag = 0;
always@*
begin
    case (curr_state)
        `sifetch:
        begin
            IRWrsel=1;
            select_npc = !flag?0:1;
```

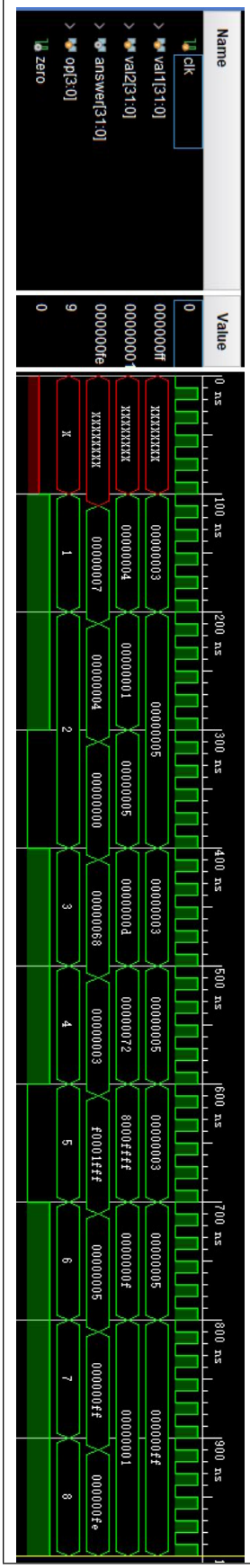
```
        flag = !flag?1:flag;
        reg_mux_mode = `REG_MODE6; //不写寄存器
        Memwrite = 0;
    end
    `sidecode:
    begin
        IRWrsel=0;
        reg_mux_mode = `REG_MODE6; //不写寄存器
        Memwrite = 0;
        select_npc=0;
        case(op)
        6'b000000:  begin  NPC_SEL = (funct==`ins_jr)?`Jr:`Normal; end
        `ins_j:     begin  NPC_SEL = `J_Jal;  end
        `ins_jal:   begin  NPC_SEL = `J_Jal;  end
        default:    begin  NPC_SEL = `Normal; end
        endcase
    end
    `sexecute:
    begin
        IRWrsel=0;
        select_npc=0;
        reg_mux_mode = `REG_MODE6; //不写寄存器
        Memwrite = 0;
        case(op)
        6'b000000:
            begin
                case (funct)
                    `ins_add:  begin ALUop = `ADD;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_addu: begin ALUop = `ADD;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_sub:  begin ALUop = `SUB;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_subu: begin ALUop = `SUB;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_and:  begin ALUop = `AND;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_or:   begin ALUop = `OR;   ALU_MODESEL = `ALU_MODE0;end
                    `ins_xor:  begin ALUop = `XOR;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_nor:  begin ALUop = `NOR;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_slt:  begin ALUop = `SLT;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_sltu: begin ALUop = `SUB;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_sll:  begin ALUop = `SLL;  ALU_MODESEL = `ALU_MODE1;end
                    `ins_srl:  begin ALUop = `SRL;  ALU_MODESEL = `ALU_MODE1;end
                    `ins_sra:  begin ALUop = `SRA;  ALU_MODESEL = `ALU_MODE1;end
                    `ins_sllv: begin ALUop = `SLL;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_srlv: begin ALUop = `SRL;  ALU_MODESEL = `ALU_MODE0;end
                    `ins_srav: begin ALUop = `SRA;  ALU_MODESEL = `ALU_MODE0;end
                    default:   begin end
```

```
                endcase
            end
        `ins_addi:  begin ALUop = `ADD; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 1; end
        `ins_addiu: begin ALUop = `ADD; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 1; end
        `ins_andi:  begin ALUop = `AND; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 0; end
        `ins_ori:   begin ALUop = `OR;  ALU_MODESEL = `ALU_MODE2; Sign_Sel = 0; end
        `ins_xori:  begin ALUop = `XOR; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 0; end
        `ins_sltiu: begin ALUop = `SUB; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 0; end
        `ins_lui:   begin ALUop = `SLL; ALU_MODESEL = `ALU_MODE3; Sign_Sel = 0; end //将立即数左移 16 位
        `ins_lw:    begin ALUop = `ADD; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 1; end
        `ins_sw:    begin ALUop = `ADD; ALU_MODESEL = `ALU_MODE2; Sign_Sel = 1; end
        `ins_beq:   begin ALUop = `SUB; ALU_MODESEL = `ALU_MODE0; NPC_SEL = `Beq; Sign_Sel = 1; select_npc=0;end
        `ins_bne:   begin ALUop = `SUB; ALU_MODESEL = `ALU_MODE0; NPC_SEL = `Bne; Sign_Sel = 1; select_npc=0;end
        `ins_bgtz:  begin ALUop = `SLT; ALU_MODESEL = `ALU_MODE4; NPC_SEL = `Bgtz;Sign_Sel = 1; select_npc=0;end
        default:    begin select_npc=0; end
        endcase
    end
    `smem:
    begin
        IRWrsel=0;
        select_npc=0;
        reg_mux_mode = `REG_MODE6; //不写寄存器
        Memwrite = op==`ins_sw?1:0;
    end
    `swb:
    begin
        IRWrsel=0;
        select_npc=0;
               case(op)
                6'b000000:
                    begin
                        case (funct)
                            `ins_slt:  begin reg_mux_mode = `REG_MODE2; end
                            `ins_sltu: begin reg_mux_mode = `REG_MODE2; end
                            `ins_sll:  begin reg_mux_mode = instruction[31:0]==0?`REG_MODE6:`REG_MODE0; end
                            `ins_jr:   begin reg_mux_mode = `REG_MODE6; end
                            default:   begin reg_mux_mode = `REG_MODE0;end
                        endcase
                    end
                `ins_sltiu: begin  reg_mux_mode = `REG_MODE1;  end
                `ins_lw:    begin  reg_mux_mode = `REG_MODE4;  end
                `ins_sw:    begin  reg_mux_mode = `REG_MODE6;  end
                `ins_beq:   begin  reg_mux_mode = `REG_MODE6;  end
                `ins_bne:   begin  reg_mux_mode = `REG_MODE6;  end
```

```
                    `ins_bgtz:  begin  reg_mux_mode = `REG_MODE6;  end
                    `ins_j:     begin  reg_mux_mode = `REG_MODE6;  end
                    `ins_jal:   begin  reg_mux_mode = `REG_MODE5;  end//regmode5 是把{PC+4}写到$31 里
                    default:    begin  reg_mux_mode = `REG_MODE3; end
                    endcase
        end
        default: begin select_npc=0; end
    endcase
end
endmodule
```

设计主要测试结果（仿真截图或下载照片）

（自己实现的仿真部分代码及截图（至少包括除时钟和存储器外的 2 个模块），贴主要说明问题的时序图并对时序进行分析，可以竖贴）



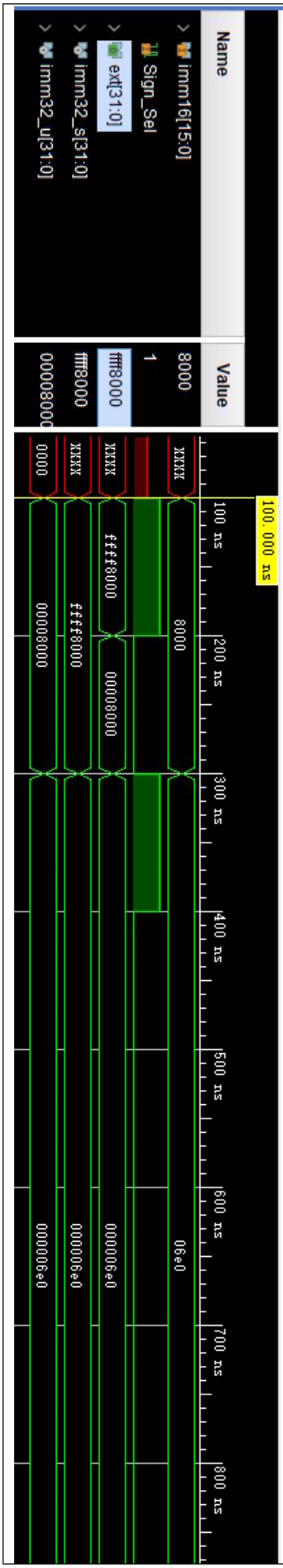
ALU 模块

```
module ALU_sim(

    );
    reg clk=0;
    reg [31:0]val1;
    reg [31:0]val2;
    wire [31:0]answer;
    reg [3:0]op;
    wire zero;
    executs32 test(clk,op,val1,val2,answer,zero);

    always #10 clk=~clk;

    initial begin
    #100 begin op=`ADD;val1=32'd3;val2=32'd4; end  //7
    #100 begin op=`SUB;val1=32'd5;val2=32'd1; end  //4
    #100 begin op=`SUB;val1=32'd5;val2=32'd5; end  //4
    #100 begin op=`SLL;val1=32'b11;val2=32'b1101; end  //1101000
    #100 begin op=`SRL;val1=32'd5;val2=32'b1110010; end //11
    #100 begin op=`SRA;val1=32'd3;val2=32'h8000ffff; end  //f0001fff
    #100 begin op=`AND;val1=32'd5;val2=32'hf; end
    #100 begin op=`OR;val1=32'hff;val2=32'd1; end
    #100 begin op=`XOR;val1=32'hff;val2=32'd1; end
    #100 begin op=`NOR;val1=32'hff;val2=32'd1; end
    #100 begin op=`SLT;val1=32'hff;val2=32'd1; end

    end

endmodule
```

输入:
clk，时钟信号
ALUop，操作类型选择信号
first，第一个操作数
second，第二个操作数

输出:
answer，执行运算后的答案
zero，答案是大于还是小于还是等于 0.

由仿真文件中的操作符号以及波形可以看出来，在执行各种操作时，例如 add, sub, sll, srl, sra...等时,计算的结果都是正确的。



EXTEND 模块

输入:
imm16, 16 位的数据
Sign_sel, 有符号选择信号

输出
ext: 32 位的拓展后的数据

内部实时计算着有符号和无符号拓展的值, 只需要通过 Sign_sel 判断哪个值就选择哪个值即可.

根据波形图也可看出功能正常.

```
module bit16_extend_sim(

    );
        reg [15:0]imm16;
    reg sign_sel;
    wire [31:0]ext;

    extend test(
    imm16,
    sign_sel,
    ext
    );

    initial begin
    #100  begin imm16=16'h8000; sign_sel = 1;end
    #100  begin imm16=16'h8000; sign_sel = 0;end
     #100 begin imm16=16'h06E0;   sign_sel = 1;end
    #100 begin imm16=16'h06E0;   sign_sel = 0;end

    end
endmodule
```

设计过程中遇到的问题及解决方法

（包括设计过程中的错误及测试过程中遇到的问题）

1. 问题大致可以分为两类，一类问题是跳转指令，即 J,Jal,Jr,Beq,Bne,Bgtz 等指令。另一类是其他的指令，出现问题的有 slt，lw，sw 等。

2. 一开始有的错误就是 PC 很早就开始跳变，然而 ref 的 PC 却很晚才开始变，这个通过删除 golden_trace 文件中的前几行得到了解决。但是在多周期中，由于 PC 是在取指阶段，取完指之后就变成了下一条指令的地址，这样的话就导致了在第一条指令执行时 ref 是 00000000，然而此时在取指结束后 PC 已经变成了 00000004，这样就导致了 PC 总是比 ref 快了一条指令。然后花了很多时间想了很多办法，首先想着用一个 PC_pre 来保存上一个 pc,但是由于在多周期的情况下，每条指令执行的周期长度不一样，比如 J 指令只需要两个周期，而 add 指令要 4 个周期，这样就导致从 add 指令到 J 指令时,PC_pre 中的 add 指令的地址只有 2 个周期,而 J 指令却有 4 个周期。这样又对不上 ref 的 PC 了。最后为了凑 ref 的 PC，用了一个 flag 寄存器让 PC 在第一条指令执行的时候不变化，这样的话，ref 是 00000000 的时候，PC 也是 00000000，这样就对上了。

3. 在解决了 5 号错误之后，就到了那些跳转指令出现错误了。因为除了跳转指令之外都可以在自己实现的仿真模块里进行测试，因此其他指令出现的错误几率比较小。还有很多指令出现问题的原因经常是线连错了（比如在顶层模块中，实例化 ALU 模块，但是把 CU 和 ALU 相连的线连错了），又或者是因为自己在模块里新加了一根 output 的线，但是忘记在顶层模块连线，导致错误。

4. 测试的逻辑是写寄存器才测试，然后 j 指令因为不写寄存器所以不会触发测试，但是 jal 指令因为在更新 PC+4 后需要写入寄存器，就会触发测试，但是 golden trace 里面对应的值是 PC 跳转后的值，而当前 PC 的值是需要写入 RF 的原 PC+4 的值，就会对不上。解决方法：令 debug_pc = Jal?NPC:PC, 避开这一情况.

5. sltiu 出现的错误是，使用三目运算符时，写的指令全部变成 XXXXXXXX，而使用 ifelse 时，写 0 和写 1 的次序不对。后来花了很多时间去测试，最后发现是在顶层模块我命名了两个变量名，zerog 和 zero_g,有一个变量名我没有使用，导致出现 XXXXXXXX。解决方法：删除那个不使用的变量名后正常。

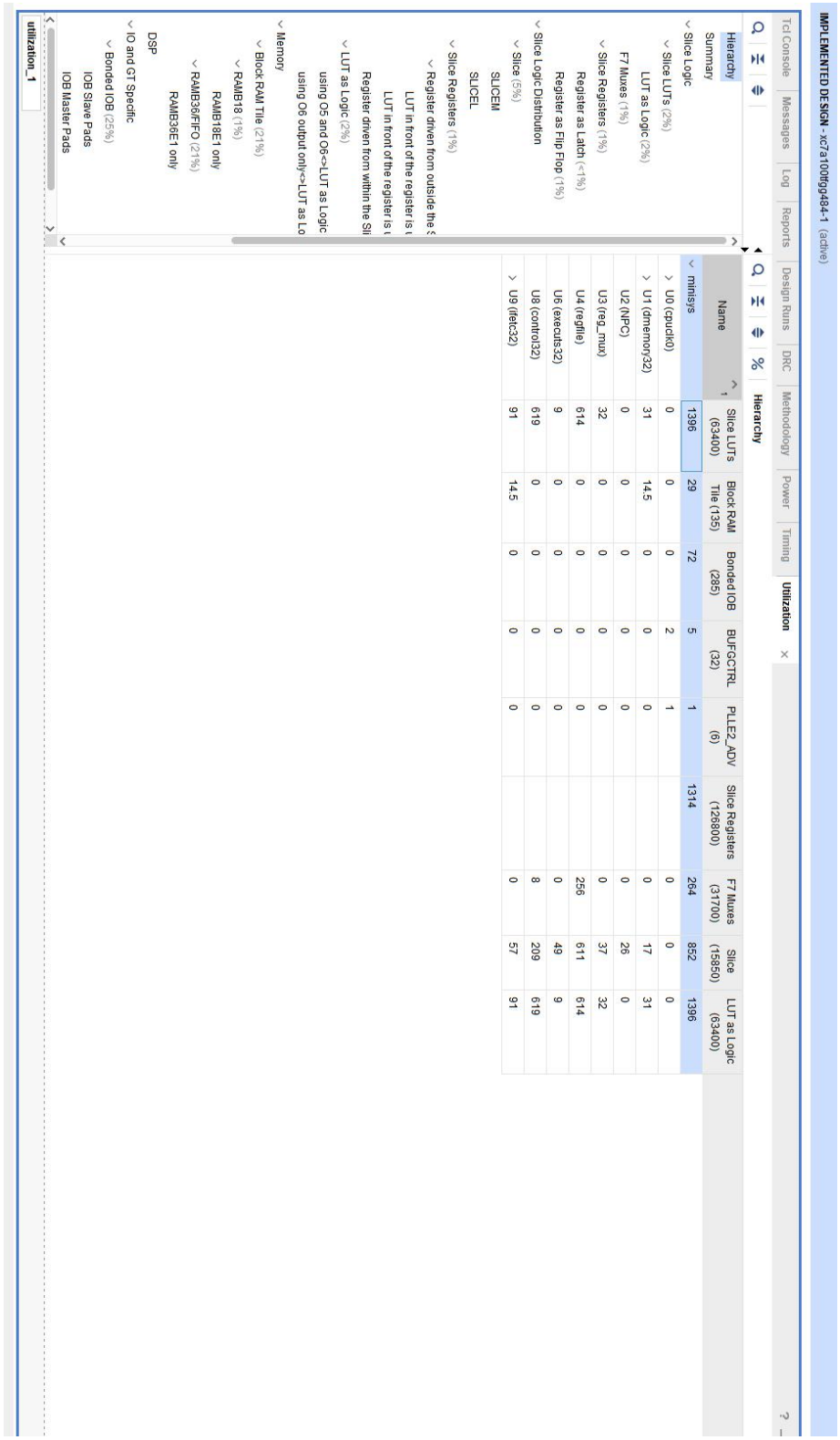
6. 其中，slt 出现问题是因为最开始在单周期中的测试包中没有测试 slt 这条指令，而我也没有意识到 slt 是有符号数的运算，导致在新的测试包出来之后，我对 slt 的操作是和 sltiu 的操作一样的，导致错误。解决方法：单独在 ALU 中增加一个 slt 指令的有符号计算。

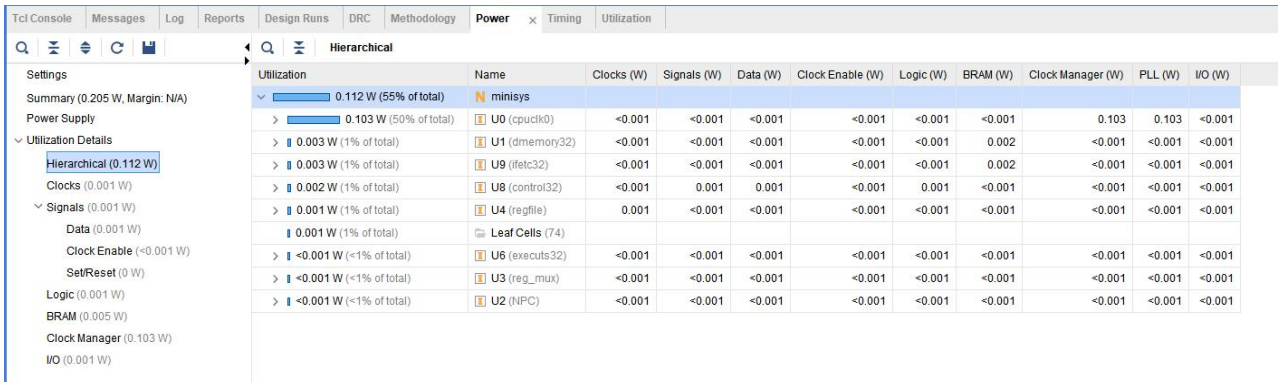
7. lw 出现问题是因为读存储器的时序不对，导致在写回的时候使用的是上次读出来的数据。解决方法：调整读存储器的的时序后正常。

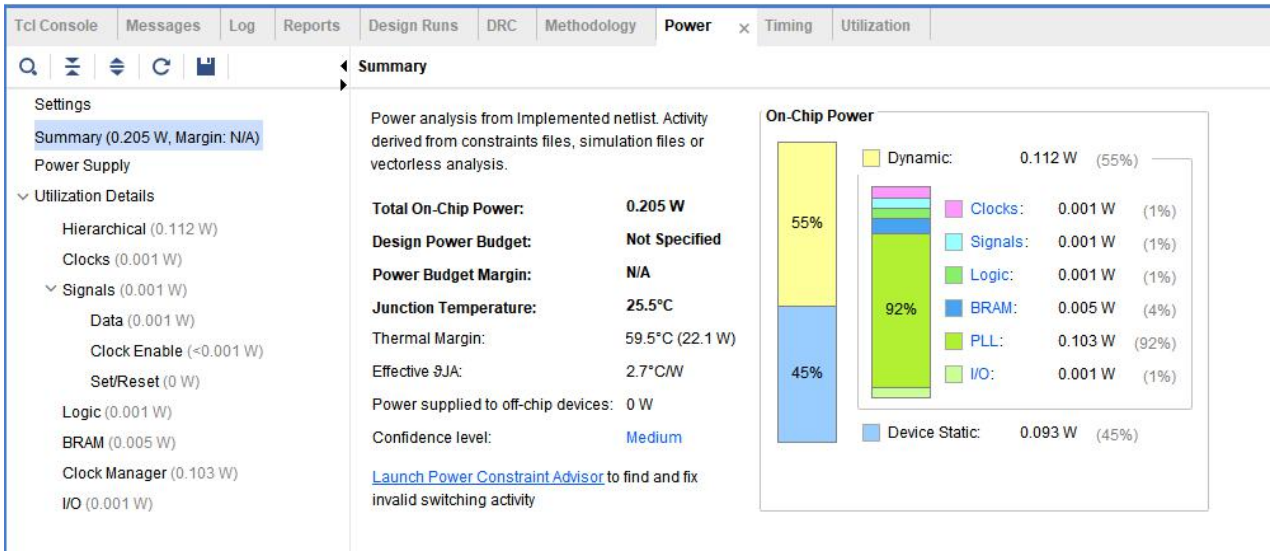
设计的性能分析

（资源使用情况、主频、功耗数据和自我分析）

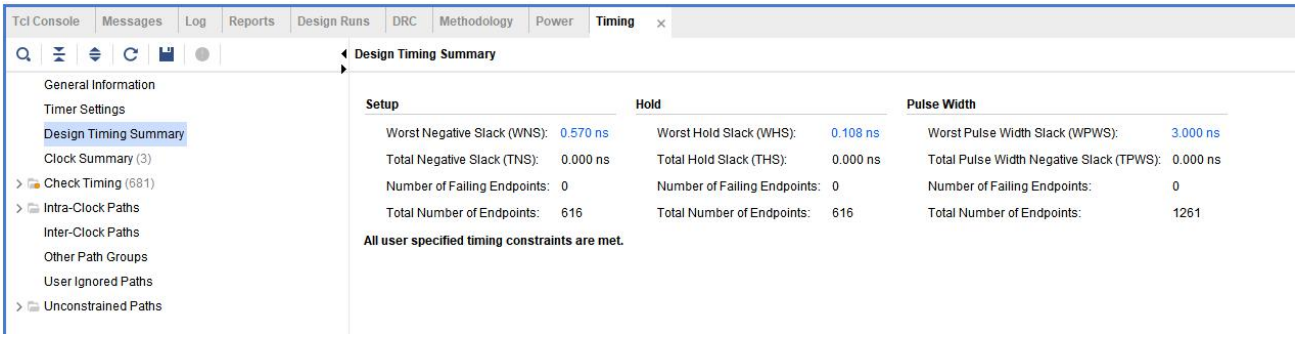
23MHz 主频:




Tcl Console
Messages
Log
Reports
Design Runs
DRC
Methodology
Power
Timing
Utilization
Hierarchical
Settings
Summary (0.205 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (0.112 W)
Clocks (0.001 W)
Signals (0.001 W)
Data (0.001 W)
Clock Enable (<0.001 W)
Set/Reset (0 W)
Logic (0.001 W)
BRAM (0.005 W)
Clock Manager (0.103 W)
I/O (0.001 W)
Utilization
Name
Clocks (W)
Signals (W)
Data (W)
Clock Enable (W)
Logic (W)
BRAM (W)
Clock Manager (W)
PLL (W)
I/O (W)
0.112 W (55% of total)
minisys
0.103 W (50% of total)
U0 (cpuclk0)
0.003 W (1% of total)
U1 (dmemory32)
0.003 W (1% of total)
U9 (ifetc32)
0.002 W (1% of total)
U8 (control32)
0.001 W (1% of total)
U4 (regfile)
0.001 W (1% of total)
Leaf Cells (74)
<0.001 W (<1% of total)
U6 (executs32)
<0.001 W (<1% of total)
U3 (reg_mux)
<0.001 W (<1% of total)
U2 (NPC)


Tcl Console
Messages
Log
Reports
Design Runs
DRC
Methodology
Power
Timing
Utilization
Summary
Settings
Summary (0.205 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (0.112 W)
Clocks (0.001 W)
Signals (0.001 W)
Data (0.001 W)
Clock Enable (<0.001 W)
Set/Reset (0 W)
Logic (0.001 W)
BRAM (0.005 W)
Clock Manager (0.103 W)
I/O (0.001 W)
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.
Total On-Chip Power: 0.205 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 25.5°C
Thermal Margin: 59.5°C (22.1 W)
Effective ϑJA: 2.7°C/W
Power supplied to off-chip devices: 0 W
Confidence level: Medium
Launch Power Constraint Advisor to find and fix invalid switching activity
On-Chip Power
55%
45%
92%
Dynamic: 0.112 W (55%)
Clocks: 0.001 W (1%)
Signals: 0.001 W (1%)
Logic: 0.001 W (1%)
BRAM: 0.005 W (4%)
PLL: 0.103 W (92%)
I/O: 0.001 W (1%)
Device Static: 0.093 W (45%)

100MHz


Tcl Console
Messages
Log
Reports
Design Runs
DRC
Methodology
Power
Timing
Design Timing Summary
General Information
Timer Settings
Design Timing Summary
Clock Summary (3)
Check Timing (681)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths
Setup
Worst Negative Slack (WNS): 0.570 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 616
Hold
Worst Hold Slack (WHS): 0.108 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 616
Pulse Width
Worst Pulse Width Slack (WPWS): 3.000 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 1261
All user specified timing constraints are met.

Tcl Console | Messages | Log | Reports | Design Runs | DRC | Methodology | Power | Timing | Utilization

Hierarchy

- Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (2%)
    - LUT as Logic (2%)
  - F7 Muxes (1%)
  - Slice Registers (1%)
    - Register as Latch (<1%)
    - Register as Flip Flop (1%)
- Slice Logic Distribution
  - Slice (4%)
    - SLICEM
    - SLICEL

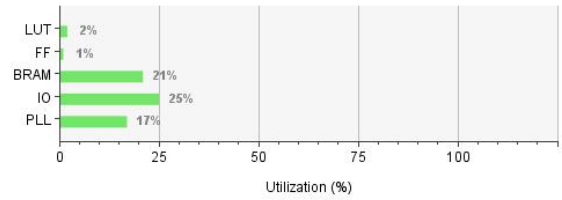
Name	Slice LUTs (63400)	Block RAM Tile (135)	Bonded IOB (285)	BUFGCTRL (32)	PLLE2_ADV (6)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)
minisys	1396	29	72	5	1	1314	264	670	1396
U0 (cpuclk0)	0	0	0	2	1		0	0	0
U1 (dmemory32)	31	14.5	0	0	0		0	18	31
U2 (NPC)	0	0	0	0	0		0	27	0
U3 (reg_mux)	32	0	0	0	0		0	33	32
U4 (regfile)	614	0	0	0	0		256	458	614
U6 (executs32)	9	0	0	0	0		0	53	9
U8 (control32)	619	0	0	0	0		8	203	619
U9 (ifetc32)	91	14.5	0	0	0		0	50	91

Tcl Console | Messages | Log | Reports | Design Runs | DRC | Methodology | Power | Timing | Utilization

Summary

- Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (2%)
    - LUT as Logic (2%)
  - F7 Muxes (1%)
  - Slice Registers (1%)
    - Register as Latch (<1%)
    - Register as Flip Flop (1%)
- Slice Logic Distribution
  - Slice (4%)
    - SLICEM
    - SLICEL
  - Slice Registers (1%)
    - Register driven from outside the S
      - LUT in front of the register is u

Resource	Utilization	Available	Utilization %
LUT	1396	63400	2.20
FF	1314	126800	1.04
BRAM	29	135	21.48
IO	72	285	25.26
PLL	1	6	16.67



Tcl Console | Messages | Log | Reports | Design Runs | DRC | Methodology | Power | Timing | Utilization

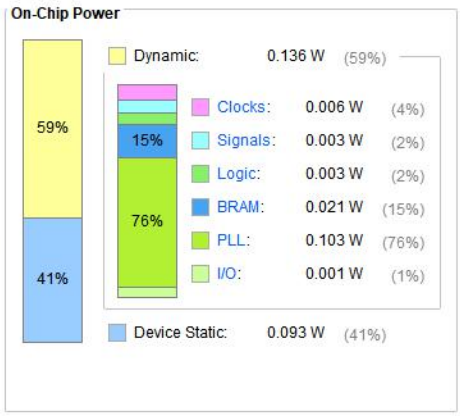
Summary

- Settings
- Summary (0.228 W, Margin: N/A)
- Power Supply
- Utilization Details
  - Hierarchical (0.136 W)
  - Clocks (0.006 W)
  - Signals (0.003 W)
    - Data (0.003 W)
    - Clock Enable (<0.001 W)
    - Set/Reset (0 W)
  - Logic (0.003 W)
  - BRAM (0.021 W)
  - Clock Manager (0.103 W)
  - I/O (0.001 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.228 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (22.1 W)
Effective ϑJA:	2.7°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



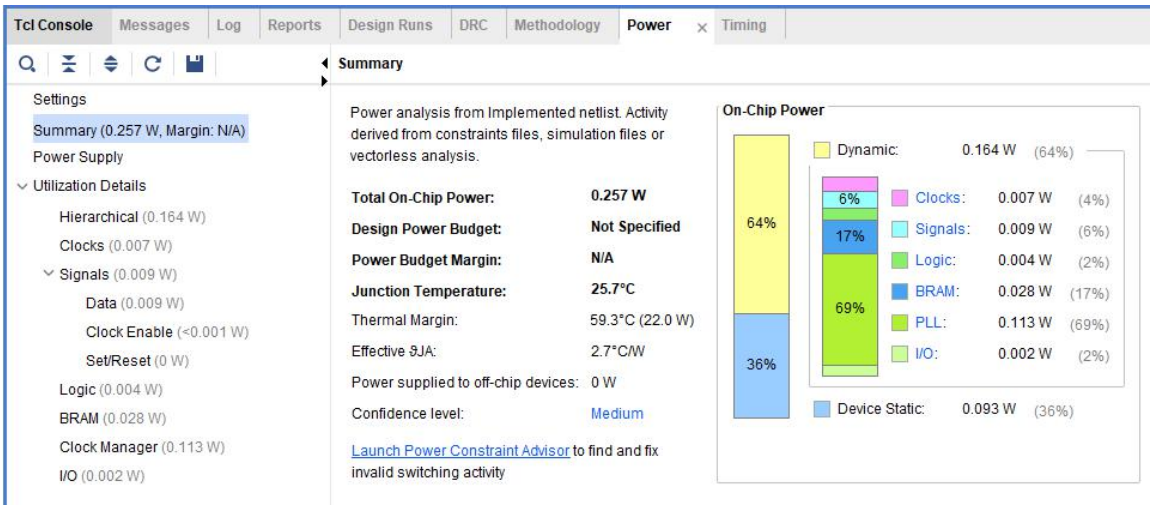
经过多次测试，在频率提高到 135MHz 时是可以综合成功的

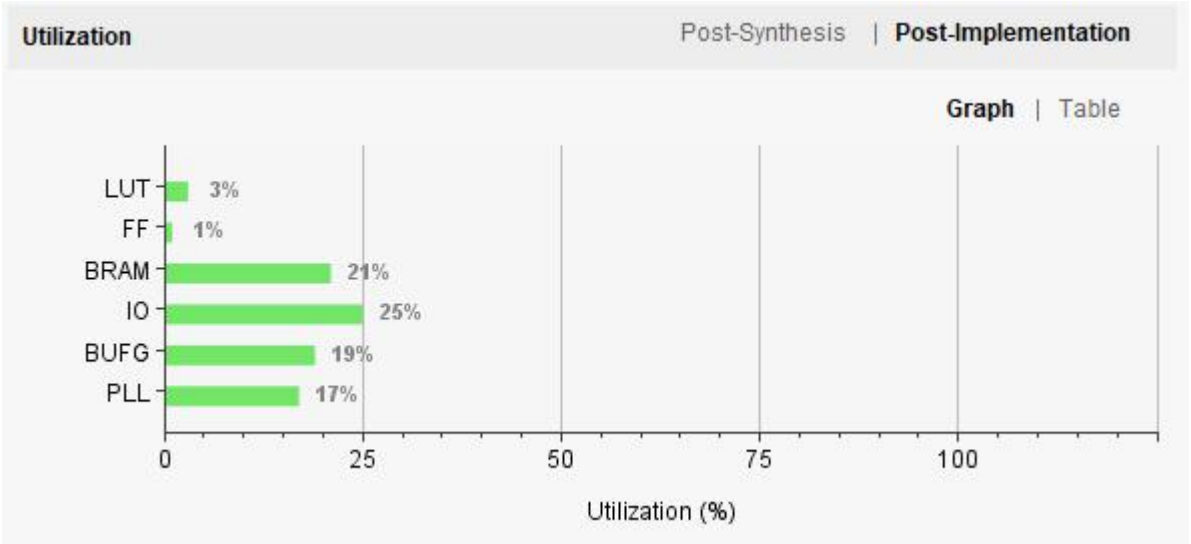
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.168 ns	Worst Hold Slack (WHS):	0.159 ns	Worst Pulse Width	
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width N	
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing	
Total Number of Endpoints:	1672	Total Number of Endpoints:	1672	Total Number of En	

All user specified timing constraints are met.

功率达到了 0.257W





项目总结

（包括设计的总结和还需改进的内容以及收获）

这次的单周期以及多周期 CPU 的设计让我加深了对计算机硬件层面的认识。设计过程中遇到了很多的错误，有一些是自己的粗心大意导致的，比如线连错了、搞混了有符号计算和无符号计算。还有一些是为了凑测试文件的时序的错误，这一方面希望学校以后开展这方面实验的时候可以优化一下测试文件。或者在实验开始之前就和同学们讲清楚具体测试需要的要求，否则在中途再去修改会比较麻烦。

自己在这次实验中,对 verilog 这个语言以及形式建模综合方法有了更多的了解。从一开始的一无所知到现在的初窥一二，心里还是觉得挺高兴的。尤其得知自己高中同学所在的大学他们只需要设计一个单周期的能实现一条指令的 CPU 时，突然感觉自己实现了 31 条指令的多周期 CPU 是挺厉害的。

当然，自己这次实验设计的多周期 CPU 也只是仅仅达到了多周期的要求而已，提高时钟频率到 150MHz 以后就出现了问题，门延迟依旧比较高。以及也没有实现流水线的 CPU。离设计真正高效的 CPU 还有很大很大的距离，由此也希望自己能够在某个领域继续深造下去，这样才能为中国乃至世界的进步贡献自己的一份力量。