

实验十一 RV32I 单周期 CPU

2022 年春季学期

“陛下，我们把这台计算机命名为‘秦一号’。请看，那里，中心部分，是 CPU，是计算机的核心计算元件。由您最精锐的五个军团构成，对照这张图您可以看到里面的加法器、寄存器、堆栈存贮器；外围整齐的部分是内存，构建这部分时我们发现人手不够，好在这部分每个单元的动作最简单，就训练每个士兵拿多种颜色的旗帜，组合起来后，一个人就能同时完成最初二十个人的操作，这就使内存容量达到了运行‘秦 1.0’操作系统的最低要求；”

— 《三体》，刘慈欣

本实验的目标是利用 FPGA 实现 RV32I 指令集中除系统控制指令之外的其余指令。利用单周期方式实现 RV32I 的控制通路及数据通路，并能够顺利通过功能仿真。在完成功能仿真后可以在 DE10-Standard 开发板上进行单步测试。通过对单周期 CPU 的实际设计来理解 CPU 内部的控制及数据通路原理，并掌握基本的系统级测试和 Debug 技术。

11.1 RISC-V 指令集简介

RISC-V 是由 UC Berkeley 推出的一套开源指令集。该指令集包含一系列的基础指令集和可选扩展指令集。在本实验中我们主要关注其中的 32 位基础指令集 RV32I。RV32I 指令集中包含了 40 条基础指令，涵盖了整数运算、存储器访问、控制转移和系统控制几个大类。本实验中无需实现系统控制的 ECALL/EBREAK、内存同步 FENCE 指令及 CSR 访问指令，所以共需实现 37 条指令。RV32I 中的程序计数器 PC 及 32 个通用寄存器均是 32 位长度，访存地址线宽度也是 32 位。RV32I 的指令长度也统一为 32 位，在实现过程中无需支持 16 位的压缩指令格式。

11.1.1 RV32I 指令编码

RV32I 的指令编码非常规整，分为六种类型，其中四种类型为基础编码类型，其余两种是变种：

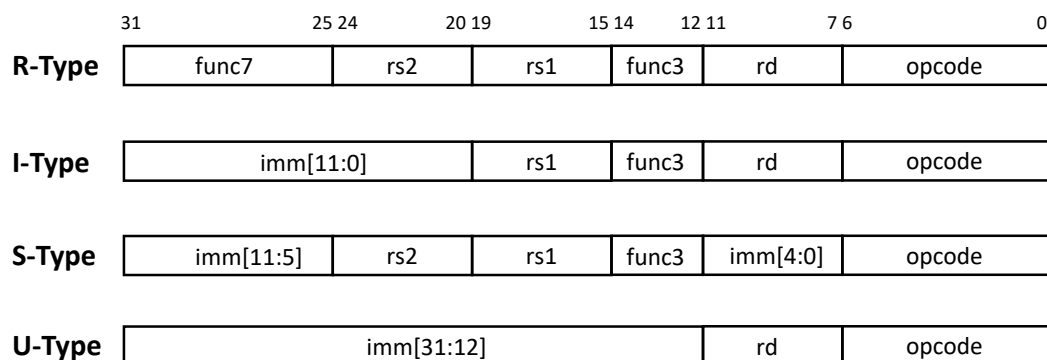


图 11-1: RV32I 指令的四种基本格式

- **R-Type**: 为寄存器操作数指令, 含 2 个源寄存器 rs1,rs2 和一个目的寄存器 rd。
- **I-Type**: 为立即数操作指令, 含一个源寄存器和一个目的寄存器和一个 12bit 立即数操作数
- **S-Type**: 为存储器写指令, 含两个源寄存器和一个 12bit 立即数。
- **B-Type**: 为跳转指令, 实际是 S-Type 的变种。与 S-Type 主要的区别是立即数编码。S-Type 中的 imm[11:5] 变为 {imm[12],imm[10:5]}, imm[4:0] 变为 {imm[4:1],imm[11]}。
- **U-Type**: 为长立即数指令, 含一个目的寄存器和 20bit 立即数操作数。
- **J-Type**: 为长跳转指令, 实际是 U-Type 的变种。与 U-Type 主要的区别是立即数编码。U-Type 中的 imm[31:12] 变为 {imm[20], imm[10:1], imm[11], imm[19:12]}。

其中四种基本格式如图 11-1 所示:

在指令编码中, opcode 必定为指令低 7bit, 源寄存器 rs1, rs2 和目的寄存器 rd 也都在特定位置出现, 所以指令解码非常方便。

🔍 **思考**: 为什么会有 S-Type/B-Type, U-Type/J-Type 这些不同的立即数编码方案? 指令相关的立即数为何在编码时采用这样“奇怪”的 bit 顺序?

11.1.2 RV32I 中的通用寄存器

RV32I 共 32 个 32bit 的通用寄存器 x0~x31(寄存器地址为 5bit 编码), 其中寄存器 x0 中的内容总是 0, 无法改变。其他寄存器的别名和寄存器使用约定

表 11-1: RV32I 中通用寄存器的定义与用法

Register	Name	Use	Saver
x0	zero	Constant 0	–
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	–
x4	tp	Thread Pointer	–
x5~x7	t0~t2	Temp	Caller
x8	s0/fp	Saved/Frame pointer	Callee
x9	s1	Saved	Callee
x10~x11	a0~a1	Arguments/Return Value	Caller
x12~x17	a2~a7	Arguments	Caller
x18~x27	s2~s11	Saved	Callee
x28~x31	t3~t6	Temp	Caller

参见表 11-1。需要注意的是，部分寄存器在函数调用时是由调用方（Caller）来负责保存的，部分寄存器是由被调用方（Callee）来保存的。在进行 C 语言和汇编混合编程时需要注意。

11.1.3 RV32I 中的指令类型

本实验中需要实现的 RV32I 指令含包含以下三类：

- **整数运算指令：**可以是对两个源寄存器操作数，或一个寄存器一个立即数操作数进行计算后，结果送入目的寄存器。运算操作包括带符号数和无符号数的算术运算、移位、逻辑操作和比较后置位等。
- **控制转移指令：**条件分支包括 beq, bne 等等，根据寄存器内容选择是否跳转。无条件跳转指令会将本指令下一条指令地址 PC+4 存入 rd 中供函数返回时使用。
- **存储器访问指令：**对内存操作是首先寄存器加立即数偏移量，以计算结果为地址读取/写入内存。读写时可以是按 32 位字，16 位半字或 8 位字节来进行读写。读写时区分无符号数和带符号数。注意：RV32I 为Load/Store型架构，内存中所有数据都需要先 load 进入寄存器才能进行操作，不能像 x86 一样直接对内存数据进行算术处理。

表 11-2: RV32I 中整数运算指令编码列表

	31	25 24	20 19	15 14	12 11	7 6	0	
lui	imm[31:12]				rd	0110111		U-Type
auipc	imm[31:12]				rd	0010111		U-Type
addi	imm[11:0]		rs1	000	rd	0010011		I-Type
slti	imm[11:0]		rs1	010	rd	0010011		I-Type
sltiu	imm[11:0]		rs1	011	rd	0010011		I-Type
xori	imm[11:0]		rs1	100	rd	0010011		I-Type
ori	imm[11:0]		rs1	110	rd	0010011		I-Type
andi	imm[11:0]		rs1	111	rd	0010011		I-Type
slli	0000000	shamt	rs1	001	rd	0010011		I-Type
srli	0000000	shamt	rs1	101	rd	0010011		I-Type
srai	0100000	shamt	rs1	101	rd	0010011		I-Type
add	0000000	rs2	rs1	000	rd	0110011		R-Type
sub	0100000	rs2	rs1	000	rd	0110011		R-Type
sll	0000000	rs2	rs1	001	rd	0110011		R-Type
slt	0000000	rs2	rs1	010	rd	0110011		R-Type
sltu	0000000	rs2	rs1	011	rd	0110011		R-Type
xor	0000000	rs2	rs1	100	rd	0110011		R-Type
srl	0000000	rs2	rs1	101	rd	0110011		R-Type
sra	0100000	rs2	rs1	101	rd	0110011		R-Type
or	0000000	rs2	rs1	110	rd	0110011		R-Type
and	0000000	rs2	rs1	111	rd	0110011		R-Type

11.1.3.1 整数运算指令

RV32I 的整数运算指令包括 21 条不同的指令，其指令编码方式参见表 11-2。

这些整数运算指令所需要完成的操作参见表 11-3。说明中 **R[reg]** 表示对地址为 **reg** 的寄存器进行操作，**M[addr]** 表示对地址为 **addr** 的内存进行操作，**SEXT(imm)** 表示对 **imm** 进行带符号扩展到 32 位，**←** 表示赋值，**<<** 及 **>>** 分别表示逻辑左移和右移，**>>>** 表示算术右移 (注意 **verilog** 与 **java** 中定义的不同)，比较过程中带 **s** 和 **u** 下标分别表示带符号数和无符号数比较。

细心的同学可能会注意到基本的整数运算指令并没有完全覆盖到所有的运算操作。RV32I 中基本指令集可以通过伪指令或组合指令的方式来实现基本指

表 11-3: 整数运算指令操作说明

指令	操作
lui rd,imm20	$R[rd] \leftarrow \{imm20, 12'b0\}$
auipc rd,imm20	$R[rd] \leftarrow PC + \{imm20, 12'b0\}$
addi rd,rs1,imm12	$R[rd] \leftarrow R[rs1] + SEXT(imm12)$
slti rd,rs1,imm12	<i>if</i> $R[rs1] <_s SEXT(imm12)$ <i>then</i> $R[rd] \leftarrow 32'b1$ <i>else</i> $R[rd] \leftarrow 32'b0$
sltiu rd,rs1,imm12	<i>if</i> $R[rs1] <_u SEXT(imm12)$ <i>then</i> $R[rd] \leftarrow 32'b1$ <i>else</i> $R[rd] \leftarrow 32'b0$
xori rd,rs1,imm12	$R[rd] \leftarrow R[rs1] \oplus SEXT(imm12)$
ori rd,rs1,imm12	$R[rd] \leftarrow R[rs1] \mid SEXT(imm12)$
andi rd,rs1,imm12	$R[rd] \leftarrow R[rs1] \& SEXT(imm12)$
slli rd,rs1,shamt	$R[rd] \leftarrow R[rs1] \ll shamt$
srli rd,rs1,shamt	$R[rd] \leftarrow R[rs1] \gg shamt$
srai rd,rs1,shamt	$R[rd] \leftarrow R[rs1] \ggg shamt$
add rd,rs1,rs2	$R[rd] \leftarrow R[rs1] + R[rs2]$
sub rd,rs1,rs2	$R[rd] \leftarrow R[rs1] - R[rs2]$
sll rd,rs1,rs2	$R[rd] \leftarrow R[rs1] \ll R[rs2][4:0]$
slt rd,rs1,rs2	<i>if</i> $R[rs1] <_s R[rs2]$ <i>then</i> $R[rd] \leftarrow 32'b1$ <i>else</i> $R[rd] \leftarrow 32'b0$
sltu rd,rs1,rs2	<i>if</i> $R[rs1] <_u R[rs2]$ <i>then</i> $R[rd] \leftarrow 32'b1$ <i>else</i> $R[rd] \leftarrow 32'b0$
xor rd,rs1,rs2	$R[rd] \leftarrow R[rs1] \oplus R[rs2]$
srl rd,rs1,rs2	$R[rd] \leftarrow R[rs1] \gg R[rs2][4:0]$
sra rd,rs1,rs2	$R[rd] \leftarrow R[rs1] \ggg R[rs2][4:0]$
or rd,rs1,rs2	$R[rd] \leftarrow R[rs1] \mid R[rs2]$
and rd,rs1,rs2	$R[rd] \leftarrow R[rs1] \& R[rs2]$

令中未覆盖到的功能，具体可以参考 11.1.4 节。对于乘除法这样的功能是通过软件实现的，在下一实验中会介绍。

11.1.3.2 控制转移指令

RV32I 中包含了 6 条分支指令和 2 条无条件转移指令。表 11-4 列出了这些控制转移指令的编码方式。

表 11-4: RV32I 中控制转移指令编码列表

	31	25 24	20 19	15 14	12 11	7 6	0	
jal	imm[20], imm[10:1], imm[11], imm[19:12]				rd	1101111		J-Type
jalr	imm[11:0]		rs1	000	rd	1100111		I-Type
beq	imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	1100011		B-Type
bne	imm[12,10:5]	rs2	rs1	001	imm[4:1,11]	1100011		B-Type
blt	imm[12,10:5]	rs2	rs1	100	imm[4:1,11]	1100011		B-Type
bge	imm[12,10:5]	rs2	rs1	101	imm[4:1,11]	1100011		B-Type
bltu	imm[12,10:5]	rs2	rs1	110	imm[4:1,11]	1100011		B-Type
bgeu	imm[12,10:5]	rs2	rs1	111	imm[4:1,11]	1100011		B-Type

表 11-5: 控制转移指令操作说明

指令		操作
jal	rd, imm20	$R[rd] \leftarrow PC + 4$ $PC \leftarrow (PC + \text{SEXT}(\{\text{imm20}, 1'b0\}))$
jalr	rd, rs1, imm12	$R[rd] \leftarrow PC + 4$ $PC \leftarrow (R[rs1] + \text{SEXT}(\text{imm12})) \& 0\text{xffffffe}$
beq	rs1, rs2, imm12	<i>if</i> $R[rs1] == R[rs2]$ <i>then</i> $PC \leftarrow PC + \text{SEXT}(\{\text{imm12}, 1'b0\})$ <i>else</i> $PC \leftarrow PC + 4$
bne	rs1, rs2, imm12	<i>if</i> $R[rs1] != R[rs2]$ <i>then</i> $PC \leftarrow PC + \text{SEXT}(\{\text{imm12}, 1'b0\})$ <i>else</i> $PC \leftarrow PC + 4$
blt	rs1, rs2, imm12	<i>if</i> $R[rs1] <_s R[rs2]$ <i>then</i> $PC \leftarrow PC + \text{SEXT}(\{\text{imm12}, 1'b0\})$ <i>else</i> $PC \leftarrow PC + 4$
bge	rs1, rs2, imm12	<i>if</i> $R[rs1] \geq_s R[rs2]$ <i>then</i> $PC \leftarrow PC + \text{SEXT}(\{\text{imm12}, 1'b0\})$ <i>else</i> $PC \leftarrow PC + 4$
bltu	rs1, rs2, imm12	<i>if</i> $R[rs1] <_u R[rs2]$ <i>then</i> $PC \leftarrow PC + \text{SEXT}(\{\text{imm12}, 1'b0\})$ <i>else</i> $PC \leftarrow PC + 4$
bgeu	rs1, rs2, imm12	<i>if</i> $R[rs1] \geq_u R[rs2]$ <i>then</i> $PC \leftarrow PC + \text{SEXT}(\{\text{imm12}, 1'b0\})$ <i>else</i> $PC \leftarrow PC + 4$

11.1.3.3 存储器访问指令

RV32I 提供了按字节、半字和字访问存储器的 8 条指令。所有访存指令的寻址方式都是寄存器间接寻址方式，访存地址可以不对齐 4 字节边界，但是在实现中可以要求访存过程中对齐 4 字节边界。在读取单个字节或半字时，可以按要求对内存数据进行符号扩展或无符号扩展后再存入寄存器。表 11-6 列出了

表 11-6: RV32I 中存储访问指令编码列表

	31	25 24	20 19	15 14	12 11	7 6	0	
lb	imm[11:0]		rs1	000	rd	0000011		I-Type
lh	imm[11:0]		rs1	001	rd	0000011		I-Type
lw	imm[11:0]		rs1	010	rd	0000011		I-Type
lbu	imm[11:0]		rs1	100	rd	0000011		I-Type
lhu	imm[11:0]		rs1	101	rd	0000011		I-Type
sb	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011		S-Type
sh	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011		S-Type
sw	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011		S-Type

表 11-7: 存储访问指令操作说明

	指令	操作
lb	$rd, imm12(rs1)$	$R[rd] \leftarrow SEXT(M_{1B}[R[rs1] + SEXT(imm12)])$
lh	$rd, imm12(rs1)$	$R[rd] \leftarrow SEXT(M_{2B}[R[rs1] + SEXT(imm12)])$
lw	$rd, imm12(rs1)$	$R[rd] \leftarrow M_{4B}[R[rs1] + SEXT(imm12)]$
lbu	$rd, imm12(rs1)$	$R[rd] \leftarrow \{24'b0, M_{1B}[R[rs1] + SEXT(imm12)]\}$
lhu	$rd, imm12(rs1)$	$R[rd] \leftarrow \{16'b0, M_{2B}[R[rs1] + SEXT(imm12)]\}$
sb	$rs2, imm12(rs1)$	$M_{1B}[R[rs1] + SEXT(imm12)] \leftarrow R[rs2][7:0]$
sh	$rs2, imm12(rs1)$	$M_{2B}[R[rs1] + SEXT(imm12)] \leftarrow R[rs2][15:0]$
sw	$rs2, imm12(rs1)$	$M_{4B}[R[rs1] + SEXT(imm12)] \leftarrow R[rs2]$

存储访问类指令的编码方式。表 11-7列出了存储访问类指令的具体操作。

11.1.4 常见伪指令

RISC-V 中规定了一些常用的伪指令。这些伪指令一般可以在汇编程序中使用，汇编器会将其转换成对应的指令序列。表 11-8介绍了 RISC-V 的常见伪指令列表。

表 11-8: 常见伪指令说明

伪指令	实际指令序列	操作
<code>nop</code>	<code>addi x0, x0, 0</code>	空操作
<code>li rd, imm</code>	<code>lui rd, imm[32:12]+imm[11]</code> <code>addi rd, rd, imm[11:0]</code>	加载 32 位立即数，先加载高位，再加上低位，注意低位是符号扩展
<code>mv rd, rs</code>	<code>addi rd, rs, 0</code>	寄存器拷贝
<code>not rd, rs</code>	<code>xori rd, rs, -1</code>	取反操作
<code>neg rd, rs</code>	<code>sub rd, x0, rs</code>	取负操作
<code>seqz rd, rs</code>	<code>sltiu rd, rs, 1</code>	等于 0 时置位
<code>snez rd, rs</code>	<code>sltu rd, x0, rs</code>	不等于 0 时置位
<code>sltz rd, rs</code>	<code>slt rd, rs, x0</code>	小于 0 时置位
<code>sgtz rd, rs</code>	<code>slt rd, x0, rs</code>	大于 0 时置位
<code>beqz rs, offset</code>	<code>beq rs, x0, offset</code>	等于 0 时跳转
<code>bnez rs, offset</code>	<code>bne rs, x0, offset</code>	不等于 0 时跳转
<code>blez rs, offset</code>	<code>bge x0, rs, offset</code>	小于等于 0 时跳转
<code>bgez rs, offset</code>	<code>bge rs, x0, offset</code>	大于等于 0 时跳转
<code>bltz rs, offset</code>	<code>blt rs, x0, offset</code>	小于 0 时跳转
<code>bgtz rs, offset</code>	<code>blt x0, rs, offset</code>	大于 0 时跳转
<code>bgt rs, rt, offset</code>	<code>blt rt, rs, offset</code>	rs 大于 rt 时跳转
<code>ble rs, rt, offset</code>	<code>bge rt, rs, offset</code>	rs 小于等于 rt 时跳转
<code>bgtu rs, rt, offset</code>	<code>bltu rt, rs, offset</code>	无符号 rs 大于 rt 时跳转
<code>bleu rs, rt, offset</code>	<code>bgeu rt, rs, offset</code>	无符号 rs 小于等于 rt 时跳转
<code>j offset</code>	<code>jal x0, offset</code>	无条件跳转，不保存地址
<code>jal offset</code>	<code>jal x1, offset</code>	无条件跳转，地址缺省保存在 x1
<code>jr rs</code>	<code>jalr x0, 0 (rs)</code>	无条件跳转到 rs 寄存器，不保存地址
<code>jalr rs</code>	<code>jalr x1, 0 (rs)</code>	无条件跳转到 rs 寄存器，地址缺省保存在 x1
<code>ret</code>	<code>jalr x0, 0 (x1)</code>	函数调用返回
<code>call offset</code>	<code>aupic x1, offset[32:12]+offset[11]</code> <code>jalr x1, offset[11:0] (x1)</code>	调用远程子函数
<code>la rd, symbol</code>	<code>aupic rd, delta[32:12]+delta[11]</code> <code>addi rd, rd, delta[11:0]</code>	载入全局地址，其中 delta 是 PC 和全局符号地址的差
<code>lla rd, symbol</code>	<code>aupic rd, delta[32:12]+delta[11]</code> <code>addi rd, rd, delta[11:0]</code>	载入局部地址，其中 delta 是 PC 和局部符号地址的差
<code>l{b h w} rd, symbol</code>	<code>aupic rd, delta[32:12]+delta[11]</code> <code>l{bhlw} rd, delta[11:0] (rd)</code>	载入全局变量
<code>s{b h w} rd, symbol, rt</code>	<code>aupic rd, delta[32:12]+delta[11]</code> <code>s{bhlw} rd, delta[11:0] (rt)</code>	载入局部变量

11.2 RV32I 电路实现

11.2.1 单周期电路设计

在了解了 RV32I 指令集的指令体系结构（Instruction Set Architecture, ISA）之后，我们将着手设计 CPU 的微架构（micro architecture）。同样的一套指令体系结构可以用完全不同的微架构来实现。不同的微架构在实现的时候只要保证程序员可见的状态，即 PC、通用寄存器和内存等，在指令执行过程中遵守 ISA 中的规定即可。具体微架构的实现可以自由发挥。在本实验中，我们首先来实现单周期 CPU 的微架构。所谓单周期 CPU 是指 CPU 在每一个时钟周期中需要完成一条指令的所有操作，即每个时钟周期完成一条指令。

每条指令的执行过程一般需要以下几个步骤：

1. **取指令**：使用本周期新的 PC 从指令存储器中取出指令，并将其放入指令寄存器（IR）中
2. **指令译码**：对取出的指令进行分析，生成本周期执行指令所需的控制信号，并计算下一条指令的地址
3. **读取操作数**：从寄存器堆中读取寄存器操作数，并完成立即数的生成
4. **运算**：利用 ALU 对操作数进行必要的运算
5. **访问内存**：包括读取或写入内存对应地址的内容
6. **寄存器写回**：将最终结果写回到目的寄存器中

每条指令执行过程中的以上几个步骤需要 CPU 的控制通路和数据通路配合来完成。其中控制通路主要负责控制信号的生成，通过控制信号来指示数据通路完成具体的数据操作。数据通路是具体完成数据存取、运算的部件。控制通路和数据通路分离的开发模式在数字系统中经常可以碰到。其设计的基本指导原则是：控制通路要足够灵活，并能够方便地修改和添加功能，控制通路的性能和时延往往不是优化重点。反过来，数据通路需要简单且性能强大。数据通路需要以可靠的方案，快速地移动和操作大量数据。在一个简单且性能强大的数据通路支持下，控制通路可以灵活地通过控制信号的组合来实现各种不同的应用。

图 11-2 提供了 RV32I 单周期 CPU 的参考设计，下面我们就针对该 CPU 的控制通路和数据通路来分别进行说明。

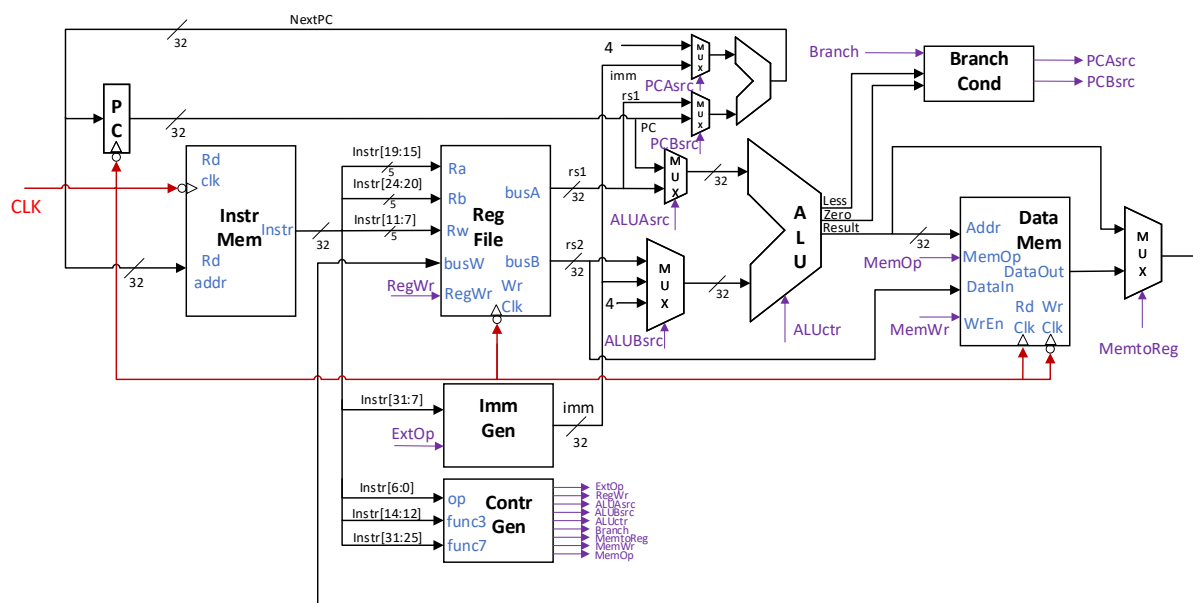


图 11-2: 单周期 CPU 的电路图

11.2.2 控制通路

11.2.2.1 PC 生成

程序计数器 **PC** 控制了整个 **CPU** 指令执行的顺序。在顺序执行的条件下，下一周期的 **PC** 为本周期 **PC+4**。如果发生跳转，**PC** 将会变成跳转目标地址。本设计中每个时钟周期是以时钟信号 **CLK** 的下降沿为起点的。在上一周期结束前，利用组合逻辑电路生成本周期将要执行的指令的地址 **NextPC**。在时钟下降沿到达时，将 **NEXT PC** 同时加载到 **PC** 寄存器和指令存储器的地址缓冲中去，完成本周期指令执行的第一步。**NextPC** 的计算涉及到指令译码和跳转分析，后续在 11.2.2.5 节中会详细描述。在系统 **reset** 或刚刚上电时，可以将 **PC** 设置为固定的地址，如全零，让系统从特定的启动代码开始执行。

11.2.2.2 指令存储器

指令寄存器 **Instruction Memory** 专门用来存放指令。虽然在冯诺伊曼结构中指令和数据是存放在统一的存储器中，但大多数现代 **CPU** 是将指令缓存和数据缓存分开的。在本实验中我们也将指令和数据分开存储。本实验中的指令存储器类似 **CPU** 中的指令缓存，采用 **FPGA** 中的 **SRAM** 来实现。由于

表 11-9: 控制信号 ExtOP 的含义

ExtOP	立即数类型
000	immI
001	immU
010	immS
011	immB
100	immJ

DE10-Standard 开发板上的大容量 SRAM 只支持在沿上进行读写，所以本设计采用时钟下降沿来对指令存储器进行读取操作，指令存储器的读取地址是 NextPC。指令存储器只需要支持读操作，在本实验中，可以要求所有代码都是 4 字节对齐，即 PC 低两位可以认为总是 2'b00。由于指令存储器每次总是读取 4 个字节，所以可以将存储器的每个单元大小设置为 32bit。指令存储器可以使用单端口 RAM 来实现，总容量一般可以设置为 256KB（64K 条指令），可以满足大部分演示代码的需求。指令存储器可以用 mif 文件来进行初始化，mif 文件可以用实验指导提供的工具链生成。如果指令存储器生成是选择了 In System Memory 动态 SRAM 内容更新的功能，可以不用编译整个工程，直接在 Quartus 中加载新的代码。

11.2.2.3 指令译码及立即数生成

在读取出本周期的指令 instr[31:0] 之后，CPU 将对 32bit 的指令进行译码，并产生各个指令对应的立即数。RV32I 的指令比较规整，所以可以直接取指令对应的 bit 做为译码结果：

```

1 assign op   = instr[6:0];
2 assign rs1  = instr[19:15];
3 assign rs2  = instr[24:20];
4 assign rd   = instr[11:7];
5 assign func3 = instr[14:12];
6 assign func7 = instr[31:25];

```

同样的，也可以利用立即数生成器 imm Generator 生成所有的立即数。注意，所有立即数均是符号扩展，且符号位总是 instr[31]:

```

1 assign immI = {{20{instr[31]}}, instr[31:20]};

```

```

2 assign immU = {instr[31:12], 12'b0};
3 assign immS = {{20{instr[31]}}}, instr[31:25], instr[11:7]};
4 assign immB = {{20{instr[31]}}}, instr[7], instr[30:25], instr[11:8], 1'b0};
5 assign immJ = {{12{instr[31]}}}, instr[19:12], instr[20], instr[30:21], 1'b0};

```

在生成各类指令的立即数之后，根据控制信号 **ExtOP**，通过多路选择器来选择立即数生成器最终输出的 **imm** 是以上五种类型中的哪一个。控制信号 **ExtOP** 为 3bit，可以按表 11-9 方式编码，未列出的编码可视为无关。

11.2.2.4 控制信号生成

在确定指令类型后，需要生成每个指令对应的控制信号，来控制数据通路部件进行对应的动作。控制信号生产部件 (Control Signal Generator) 是根据 **instr** 中的操作码 **opcode**，及 **func3** 和 **func7** 来生成对应的控制信号的。其中操作码实际只有高 5 位有用，**func7** 实际只有次高位有用。生成的控制信号具体包括：

- **ExtOP**: 宽度为 3bit，选择立即数产生器的输出类型，具体含义参见表 11-9。
- **RegWr**: 宽度为 1bit，控制是否对寄存器 **rd** 进行写回，为 1 时写回寄存器。
- **ALUAsrc**: 宽度为 1bit，选择 ALU 输入端 A 的来源。为 0 时选择 **rs1**，为 1 时选择 **PC**。
- **ALUBsrc**: 宽度为 2bit，选择 ALU 输入端 B 的来源。为 00 时选择 **rs2**，为 01 时选择 **imm**(当是立即数移位指令时，只有低 5 位有效)，为 10 时选择常数 4（用于跳转时计算返回地址 **PC+4**）。
- **ALUctr**: 宽度为 4bit，选择 ALU 执行的操作，具体含义参见表 ??。
- **Branch**: 宽度为 3bit，说明分支和跳转的种类，用于生成最终的分支控制信号，含义参见表 11-12。
- **MemtoReg**: 宽度为 1bit，选择寄存器 **rd** 写回数据来源，为 0 时选择 ALU 输出，为 1 时选择数据存储器输出。
- **MemWr**: 宽度为 1bit，控制是否对数据存储器进行写入，为 1 时写回存储器。
- **MemOP**: 宽度为 3bit，控制数据存储器读写格式，为 010 时为 4 字节读写，为 001 时为 2 字节读写带符号扩展，为 000 时为 1 字节读写带符号扩展，为 101 时为 2 字节读写无符号扩展，为 100 时为 1 字节读写无符号扩展。

表 11-10: RV32I 指令控制信号列表第一部分

指令	op[6:2]	func3	func7[5]	ExtOP	RegWr	Branch	MemtoReg	MemWr	MemOP
lui	01101	×	×	001	1	000	0	0	×
auipc	00101	×	×	001	1	000	0	0	×
addi	00100	000	×	000	1	000	0	0	×
slti	00100	010	×	000	1	000	0	0	×
sltiu	00100	011	×	000	1	000	0	0	×
xori	00100	100	×	000	1	000	0	0	×
ori	00100	110	×	000	1	000	0	0	×
andi	00100	111	×	000	1	000	0	0	×
slli	00100	001	0	000	1	000	0	0	×
srli	00100	101	0	000	1	000	0	0	×
srai	00100	101	1	000	1	000	0	0	×
add	01100	000	0	×	1	000	0	0	×
sub	01100	000	1	×	1	000	0	0	×
sll	01100	001	0	×	1	000	0	0	×
slt	01100	010	0	×	1	000	0	0	×
sltu	01100	011	0	×	1	000	0	0	×
xor	01100	100	0	×	1	000	0	0	×
srl	01100	101	0	×	1	000	0	0	×
sra	01100	101	1	×	1	000	0	0	×
or	01100	110	0	×	1	000	0	0	×
and	01100	111	0	×	1	000	0	0	×
jal	11011	×	×	100	1	001	0	0	×
jalr	11001	000	×	000	1	010	0	0	×
beq	11000	000	×	011	0	100	×	0	×
bne	11000	001	×	011	0	101	×	0	×
blt	11000	100	×	011	0	110	×	0	×
bge	11000	101	×	011	0	111	×	0	×
bltu	11000	110	×	011	0	110	×	0	×
bgeu	11000	111	×	011	0	111	×	0	×
lb	00000	000	×	000	1	000	1	0	000
lh	00000	001	×	000	1	000	1	0	001
lw	00000	010	×	000	1	000	1	0	010
lbu	00000	100	×	000	1	000	1	0	100
lhu	00000	101	×	000	1	000	1	0	101
sb	01000	000	×	010	0	000	×	1	000
sh	01000	001	×	010	0	000	×	1	001
sw	01000	010	×	010	0	000	×	1	010

表 11-11: RV32I 指令控制信号列表第二部分

指令	op[6:2]	func3	func7[5]	ALUAsrc	ALUBsrc	ALUctr	ALU 动作
lui	01101	×	×	×	01	0011	拷贝立即数
auipc	00101	×	×	1	01	0000	PC + 立即数
addi	00100	000	×	0	01	0000	rs1 + imm
slti	00100	010	×	0	01	0010	slt rs1, imm
sltiu	00100	011	×	0	01	1010	sltu rs1, imm
xori	00100	100	×	0	01	0100	rs1 \oplus imm
ori	00100	110	×	0	01	0110	rs1 imm
andi	00100	111	×	0	01	0111	rs1 & imm
slli	00100	001	0	0	01	0001	rs1 << imm[4:0]
srli	00100	101	0	0	01	0101	rs1 >> imm[4:0]
srai	00100	101	1	0	01	1101	rs1 >>> imm[4:0]
add	01100	000	0	0	00	0000	rs1 + rs2
sub	01100	000	1	0	00	1000	rs1 - rs2
sll	01100	001	0	0	00	0001	rs1 << rs2[4:0]
slt	01100	010	0	0	00	0010	slt rs1, rs2
sltu	01100	011	0	0	00	1010	sltu rs1, rs2
xor	01100	100	0	0	00	0100	rs1 \oplus rs2
srl	01100	101	0	0	00	0101	rs1 >> rs2[4:0]
sra	01100	101	1	0	00	1101	rs1 >>> rs2[4:0]
or	01100	110	0	0	00	0110	rs1 rs2
and	01100	111	0	0	00	0111	rs1 & rs2
jal	11011	×	×	1	10	0000	PC + 4
jalr	11001	000	×	1	10	0000	PC + 4
beq	11000	000	×	0	00	0010	根据 rs1, rs2 设置 Zero
bne	11000	001	×	0	00	0010	根据 rs1, rs2 设置 Zero
blt	11000	100	×	0	00	0010	根据 rs1, rs2 设置带符号 Less
bge	11000	101	×	0	00	0010	根据 rs1, rs2 设置带符号 Less
bltu	11000	110	×	0	00	1010	根据 rs1, rs2 设置无符号 Less
bgeu	11000	111	×	0	00	1010	根据 rs1, rs2 设置无符号 Less
lb	00000	000	×	0	01	0000	rs1 + imm
lh	00000	001	×	0	01	0000	rs1 + imm
lw	00000	010	×	0	01	0000	rs1 + imm
lbu	00000	100	×	0	01	0000	rs1 + imm
lhu	00000	101	×	0	01	0000	rs1 + imm
sb	01000	000	×	0	01	0000	rs1 + imm
sh	01000	001	×	0	01	0000	rs1 + imm
sw	01000	010	×	0	01	0000	rs1 + imm

扩展。

这些控制信号控制各个数据通路部件按指令的要求进行对应的操作。所有指令对应的控制信号列表如表 11-10 和表 11-11 所示。根据这些控制信号可以得出系统在给定指令下的一个周期内所需要做的具体操作。注意：这里的控制信号在定义上可能和教科书上的 9 条指令 CPU 控制信号有一些区别。如果没有学习过组成原理的同学请参考相关教科书，分析各类指令在给定控制信号下的数据通路的具体操作。这里只进行一个简略的说明：

- **lui**: ALU A 输入端不用，B 输入端为立即数，按 U-Type 扩展，ALU 执行的操作是拷贝 B 输入，ALU 结果写回到 rd。
- **auipc**: ALU A 输入端为 PC，B 输入端为立即数，按 U-Type 扩展，ALU 执行的操作是加法，ALU 结果写回到 rd。
- **立即数运算类指令**: ALU A 输入端为 rs1，B 输入端为立即数，按 I-Type 扩展。ALU 按 ALUctr 进行操作，ALU 结果写回 rd。
- **寄存器运算类指令**: ALU A 输入端为 rs1，B 输入端为 rs2。ALU 按 ALUctr 进行操作，ALU 结果写回 rd。
- **jal**: ALU A 输入端 PC，B 输入端为常数 4，ALU 执行的操作是计算 PC+4，ALU 结果写回到 rd。PC 计算通过专用加法器，计算 PC+imm，imm 为 J-Type 扩展。
- **jalr**: ALU A 输入端 PC，B 输入端为常数 4，ALU 执行的操作是计算 PC+4，ALU 结果写回到 rd。PC 计算通过专用加法器，计算 rs1+imm，imm 为 I-Type 扩展。
- **Branch 类指令**: ALU A 输入端为 rs1，B 输入端为 rs2，ALU 执行的操作是比較大小或判零，根据 ALU 标志位选择 NextPC，可能是 PC+4 或 PC+imm，imm 为 B-Type 扩展，PC 计算由专用加法器完成。不写回寄存器。
- **Load 类指令**: ALU A 输入端为 rs1，B 输入端为立即数，按 I-Type 扩展。ALU 做加法计算地址，读取内存，读取内存方式由存储器具体执行，内存输出写回 rd。
- **Store 类指令**: ALU A 输入端为 rs1，B 输入端为立即数，按 S-Type 扩展。ALU 做加法计算地址，将 rs2 内容写入内存，不写回寄存器。

11.2.2.5 跳转控制

代码执行过程中，NextPC 可能会有多种可能性：

表 11-12: 控制信号 Branch 的含义

Branch	跳转类型
000	非跳转指令
001	无条件跳转 PC 目标
010	无条件跳转寄存器目标
100	条件分支，等于
101	条件分支，不等于
110	条件分支，小于
111	条件分支，大于等于

- 顺序执行：NextPC = PC + 4;
- jal: NextPC = PC + imm;
- jalr: NextPC = rs1 + imm;
- 条件跳转：根据 ALU 的 Zero 和 Less 来判断，NextPC 可能是 PC + 4 或者 PC + imm;

在设计中使用一个单独的专用加法器来进行 PC 的计算。同时，利用了跳转控制模块来生成加法器输入选择。其中，PCASrc 控制 PC 加法器输入 A 的信号，为 0 时选择常数 4，为 1 时选择 imm。PCBsrc 控制 PC 加法器输入 B 的信号，为 0 时选择本周 PC，为 1 时选择寄存器 rs1。

跳转控制模块根据控制信号 Branch 和 ALU 输出的 Zero 及 Less 信号来决定 PCASrc 和 PCBsrc。其中控制信号 Branch 的定义如表 11-12所示。跳转控制模块的输出见表 11-13。

11.2.3 数据通路

单周期数据通路可以复用上个实验中设计的寄存器堆、ALU 和数据存储器，这里就不再详细描述。

11.2.4 单周期 CPU 的时序设计

在单周期 CPU 中，所有操作均需要在一个周期内完成。其中单周期存储部件的读写是时序设计的关键。在 CPU 架构中 PC、寄存器堆，指令存储器和数据存储器都是状态部件，需要用寄存器或存储器来实现。在实验五中，我们也

表 11-13: PC 加法器输入选择逻辑

Branch	Zero	Less	PCAsrc	PCBsrc	NextPC
000	×	×	0	0	PC + 4
001	×	×	1	0	PC + imm
010	×	×	1	1	rs1 + imm
100	0	×	0	0	PC + 4
100	1	×	1	0	PC + imm
101	0	×	1	0	PC + imm
101	1	×	0	0	PC + 4
110	×	0	0	0	PC + 4
110	×	1	1	0	PC + imm
111	×	0	1	0	PC + imm
111	×	1	0	0	PC + 4

看到，如果要求上述存储部件以非同步方式进行读写会消耗大量资源，无法实现大容量的存储。因此，需要仔细地规划各个存储器的读写时序和实现方式。

图 11-3 描述了本实验建议的时序设计方案。在该设计中，PC 和寄存器堆由于容量不大，可以采用非同步方式读取，即地址改变后立即输出对应的数据。对指令存储器和数据存储器来说，一般系统至少需要数百 KB 的容量。此时建议用时钟沿来控制读写。假定我们是以时钟下降沿为每个时钟周期的开始。对于存储器和寄存器的写入统一安排在下降沿上进行。对于数据存储器的读出，由于地址计算有一定延时，可以在时钟上升沿进行读取。

下面通过图 11-3 来描述单个时钟周期内 CPU 的具体动作。其中绿色部分为本周期正确数据，黄色为上一周期的旧数据，蓝色为下一周期的未来数据。

- 周期开始的下降沿将同时用于写入 PC 寄存器和读取指令存储器。由于指令存储器要在下降沿进行读取操作，而 PC 的输出要等到下降沿后才能更新，所以不能拿 PC 的输出做为指令存储器的地址。可以采用 PC 寄存器的输入，NextPC 来做为指令存储器的地址。该信号是组合逻辑，一般在上个周期末就已经准备好。
- 指令读出后将出现在指令存储器的输出端，该信号可以通过组合逻辑来进行指令译码，产生控制信号，寄存器读写地址及立即数等。
- 寄存器读地址产生后，直接通过非同步读取方式，读取两个源寄存器的数据，与立即数操作数一起准备好，进入 ALU 的输入端。

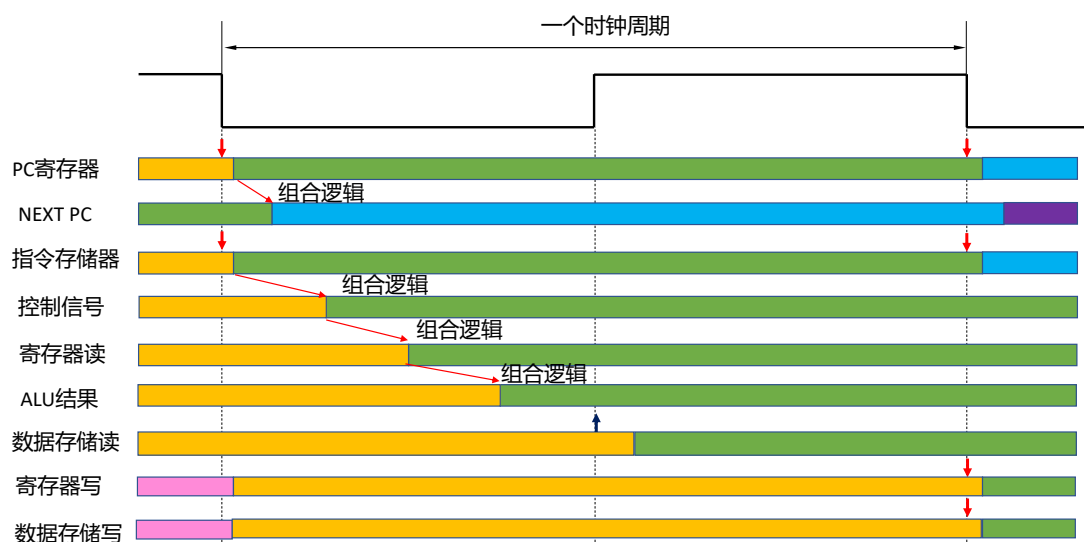


图 11-3: 单周期 CPU 的时序设计

- ALU 也是组合逻辑电路，在输入端数据准备好后就开始运算。由于数据存储器的读取地址也是 ALU 来计算的，所以要求 ALU 输出结果能在时钟半个周期的上升沿到来之前准备好。
- 时钟上升沿到来的时候，数据存储器的读地址如果准备好了，就可以在上沿进行内存读取操作。
- 最后，在下一周期的时钟下降沿到来的时候，同时对目的寄存器和数据进行写入操作。这样下一周期这些存储器中的数据就是最新的了。

实验采用的 DE10-Standard 开发板上的 M10K 支持读写时钟分离的 ram，且能够在主时钟 50MHz 下进行单周期读写操作。在此时序设计下，主要的关键路径在 Load 指令的读取地址生成，该路径需要在半个周期内完成，如果出现时序无法满足的情况，可以考虑降低时钟频率。

11.2.4.1 模块化设计

CPU 设计过程中需要多个不同的模块分工协作，建议在开始编码前划分好具体模块的功能和接口。对于模块划分我们提供了以下的参考建议。顶层实体内包含的模块主要是：

- **CPU 模块：**主要对外接口包括时钟、Reset、指令存储器的地址/数据线、数据存储器的地址及数据线和自行设计的调试信号。

- ALU 模块: 主要对外接口是 ALU 的输入操作数、ALU 控制字、ALU 结果输出和标志位输出等。
 - ◆ 加法器模块
 - ◆ 桶型移位器模块
- 寄存器堆模块: 主要对外接口是读写寄存器号输入、写入数据输入、寄存器控制信号、写入时钟、输出数据。
- 控制信号生成模块: 主要对外接口是指令输入及各种控制信号输出。
- 立即数生成器模块: 主要对外接口是指令输入, 立即数类型及立即数输出。
- 跳转控制模块: 主要对外接口是 ALU 标志位输入、跳转控制信号输入及 PC 选择信号输出。
- PC 生成模块: 主要对外接口是 PC 输入、立即数输入, rs1 输入, PC 选择信号及 NEXTPC 输出。
- 指令存储器模块: 主要对外接口包括时钟、地址线和输出数据线。
- 数据存储器模块: 主要对外接口包括时钟、输入输出地址线、输入输出数据、内存访问控制信号和写使能信号。
- 外围设备: 用于 Reset 或显示调试结果等等。

以上模块划分不是唯一的, 同学们可以按照自己的理解进行划分。设计中将存储器部分与 CPU 分开放在顶层的主要目的是在后续的计算机系统实验中简化外设对存储器的访问。设计时请先划分模块, 确认模块的连接, 再单独开发各个模块, 建议对模块进行单元测试后再整合系统。

11.3 软件及测试部分

RISC-V CPU 是一个较为复杂的数字系统, 在开发过程中需要对每一个环节进行详细的测试才能够保证系统整体的可靠性。

11.3.1 单元测试

在开发过程中, 需要首先确保每一个子单元都正常工作, 因此在完成各个单元的代码编写后需要进行对应的测试。具体可以包括:

- **代码复查**：检查代码编写过程中是否有问题，尤其是变量名称、数据线宽度等易出错的地方。检查编译中的警告，判断是否警告会带来错误。
- **RTL 复查**：利用 RTL Viewer 检查系统编译输出的 RTL 是否符合设计构想，有没有悬空或未连接的引脚。
- **TestBench 功能仿真**：通过针对独立单元的 testbench 进行功能仿真，尤其需要注意 ALU、寄存器堆、及内容的功能正确性。对于存储元件需要分析时序正确性，即数据是否在正确的时间给出，写入时是否按预期写入等。

11.3.2 单步功能仿真

在完成基本单元测试后，可以进行 CPU 整体的联调。整体联调的主要目的是验证各个指令基本功能的正确性。实验指导提供了 testbench 的示例帮助大家进行单步指令的执行和验证。

在这个 Testbench 中，首先定义了一部分测试中需要用到的变量：

```

1 `timescale 1 ns / 10 ps
2 module testbench_cpu();
3   integer numcycles; //number of cycles in test
4   reg clk,reset; //clk and reset signals
5   reg[8*30:1] testcase; //name of testcase

```

其中 testcase 是我们的测试用例名，为字符串格式，用来载入不同的测试用例。

随后，在 testbench 中实例化 CPU 中的部件，这里单独实例化了 CPU 主体、指令存储和数据存储：

```

1 // CPU declaration
2 // signals
3 wire [31:0] iaddr,idataout;
4 wire iclk;
5 wire [31:0] daddr,ddataout,ddatain;
6 wire drdclk, dwrclk, dwe;
7 wire [2:0] dop;
8 wire [23:0] cpudbgdata;
9
10 //main CPU
11 rv32is mycpu(.clock(clk),
12               .reset(reset),

```

```

13         .imemaddr(iaddr), .imemdataout(idataout), .imemclk(iclk),
14         .dmemaddr(daddr), .dmemdataout(ddataout), .dmemdatain(ddatain),
15         .dmemrdclk(drdclk), .dmemwrclk(dwrclk), .dmemop(dop), .dmemwe(dwe)
16         .dbgdata(cpudbgdata));
17
18 //instruction memory, no writing
19 testmem instructions(
20     .address(iaddr[17:2]),
21     .clock(iclk),
22     .data(32'b0),
23     .wren(1'b0),
24     .q(idataout));
25
26 //data memory
27 dmem datamem(.addr(daddr),
28     .dataout(ddataout),
29     .datain(ddatain),
30     .rdclk(drdclk),
31     .wrclk(dwrclk),
32     .memop(dop),
33     .we(dwe));

```

在实际实现中请同学们根据自己设计的 CPU 接口自行进行更改。在测试过程中，建议可以用自己写的 memory 模块替代 IP 核生成的 memory 模块，方便对内存进行各类操作。

随后，定义了一系列的辅助 task，帮助我们完成各类测试操作：

```

1 //useful tasks
2 task step; //step for one cycle ends 1ns AFTER the posedge of the next cycle
3     begin
4         #9 clk=1'b0;
5         #10 clk=1'b1;
6         numcycles = numcycles + 1;
7         #1 ;
8     end
9 endtask
10
11 task stepn; //step n cycles

```

```

12     input integer n;
13     integer i;
14     begin
15         for (i =0; i<n ; i=i+1)
16             step();
17     end
18 endtask
19
20 task resetcpu; //reset the CPU and the test
21     begin
22         reset = 1'b1;
23         step();
24         #5 reset = 1'b0;
25         numcycles = 0;
26     end
27 endtask

```

其中 `step` 任务是将 CPU 时钟前进一个周期，在单周期 CPU 中等价于单步执行一条指令。注意我们这里的周期是以上升沿开始的，在实际测试中可以将时间步进到下一个周期的上升沿后一个时间单位，这主要是由于我们单周期 CPU 是在下一上升沿进行写入，对数据的验证要在上升沿略后一些的时间进行。`stepn` 任务用于执行 `n` 条指令，`resetcpu` 用于将 `cpu` 重置，从预定开始执行的地址重新执行。

Testbench 中还定义了载入任务：

```

1 task loadtestcase; //load instructions to instruction mem
2     begin
3         $readmemh({testcase, ".hex"},instructions.ram);
4         $display("---Begin test case %s-----", testcase);
5     end
6 endtask

```

该任务用于载入指令文件。指令文件为文本格式，建议放在 `simulate/modelsim` 子目录下，用相对目录名来定位文件。这里采用 `$readmemh` 读入到指定的指令存储中去，由于指令存储空间的声明不在顶层实体 `instructions` 中，需要使用 `instructions.ram` 来访问实体内部声明的变量 `ram`。在编写 `testbench` 时请同学们自己按照自己的设计来定位实际应该访问的变量的位置。

同时还需要定义一系列的断言任务，辅助检查寄存器或者内存中的内容，并在出错时提供提示信息：

```

1 task checkreg;//check registers
2     input [4:0] regid;
3     input [31:0] results;
4     reg [31:0] debugdata;
5     begin
6         debugdata=mycpu.myregfile.regs[regid]; //get register content
7         if(debugdata==results)
8             begin
9                 $display("OK: end of cycle %d reg %h need to be %h, get %h",
10                     numcycles-1, regid, results, debugdata);
11             end
12         else
13             begin
14                 $display("!!!Error: end of cycle %d reg %h need to be %h, get %h",
15                     numcycles-1, regid, results, debugdata);
16             end
17         end
18 endtask

```

在这个任务中访问了 CPU 内部定义的寄存器堆 myregfile 中的 regs 变量，并根据所需要的访问 regid 来提取数据，并和预期数据进行比较。如果不正确，任务会提示比较结果，方便进行 debug。同样的，也可以编写类似的内存内容比较模块，对内存中的内容进行检查。

假定需要测试 CPU 中加法语句的正确性，同学们可以编写一小段汇编

```

1 addi t1,zero,100
2 addi t2,zero,20
3 add t3,t1,t2

```

在这段汇编执行过程中，我们可以检查各个寄存器结果，观察代码执行的正确性。利用上学期用过的rars仿真器来将这段汇编转换为二进制，并写入 add.hex 文件中，无需添加文件头 v2.0 raw。示例文件的具体内容如下：

```

1 06400313
2 01400393
3 00730E33

```

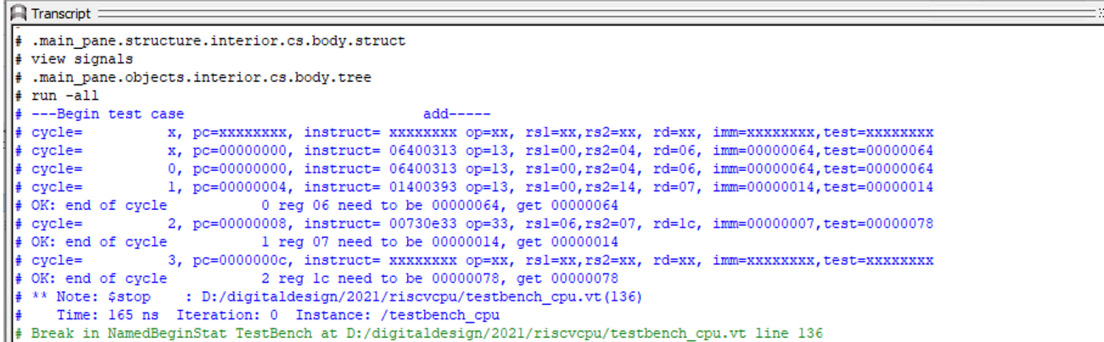
Testbench 具体的执行部分如下：

```

1  initial begin:TestBench
2      #80
3      // output the state of every instruction
4      $monitor("cycle=%d, pc=%h, instruct= %h op=%h,
5              rsl=%h,rs2=%h, rd=%h, imm=%h",
6              numcycles, mycpu.pc, mycpu.instr, mycpu.op,
7              mycpu.rsl,mycpu.rs2,mycpu.rd,mycpu.imm);
8
9      testcase = "add";
10     loadtestcase();
11     resetcpu();
12     step();
13     checkreg(6,100); //t1==100
14     step();
15     checkreg(7,20);  //t2=20
16     step();
17     checkreg(28,120); //t3=120
18     $stop
19 end

```

执行过程中，首先使用 \$monitor 来定义我们需要观察的变量，只要这些变量发生变化 modelsim 会自动地打印出变量的内容。这样，可以在每条指令执行时看到对应的 PC 及指令解码的关键信息。同学们也可以自定义需要观察的信号。在载入 add 测试用例后，testbench 单步执行了 3 次，每次执行完就按照我们预期的执行结果检查了 t1,t2 和 t3 寄存器。modelsim 的实际输出如图 11-4：



```

Transcript
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ---Begin test case          add-----
# cycle=      x, pc=xxxxxxx, instruct= xxxxxxxx op=xx, rsl=xx,rs2=xx, rd=xx, imm=xxxxxxx,test=xxxxxxx
# cycle=      x, pc=00000000, instruct= 06400313 op=13, rsl=00,rs2=04, rd=06, imm=00000064,test=00000064
# cycle=      0, pc=00000000, instruct= 06400313 op=13, rsl=00,rs2=04, rd=06, imm=00000064,test=00000064
# cycle=      1, pc=00000004, instruct= 01400393 op=13, rsl=00,rs2=14, rd=07, imm=00000014,test=00000014
# OK: end of cycle      0 reg 06 need to be 00000064, get 00000064
# cycle=      2, pc=00000008, instruct= 00730e33 op=33, rsl=06,rs2=07, rd=1c, imm=00000007,test=00000078
# OK: end of cycle      1 reg 07 need to be 00000014, get 00000014
# cycle=      3, pc=0000000c, instruct= xxxxxxxx op=xx, rsl=xx,rs2=xx, rd=xx, imm=xxxxxxx,test=xxxxxxx
# OK: end of cycle      2 reg 1c need to be 00000078, get 00000078
# ** Note: $stop      : D:/digitaldesign/2021/riscvcpu/testbench_cpu.vt(136)
#      Time: 165 ns Iteration: 0 Instance: /testbench_cpu
# Break in NamedBeginStat TestBench at D:/digitaldesign/2021/riscvcpu/testbench_cpu.vt line 136

```

图 11-4: 单周期 CPU 的仿真输出

从图中可以看到初始化结束后代码从全零地址开始执行，每个周期结束后会对寄存器进行检查。注意这里检查点在上升沿到来后，所以在第 n 周期结束时，PC 和指令已经是下一条指令的内容了。

请自行按照自己的设计修改单步仿真 `testbench`，并自行设计测试用例来对 CPU 进行初步联调。

11.3.3 系统功能仿真

单步功能仿真用于简单验证 CPU 中各条指令的基本情况，确保 CPU 可以完成基础功能。但是，为了排除 CPU 中潜在的 bug，我们需要对 CPU 的实现进行详细的测试，避免后面在搭建整个计算机系统时由于 CPU 的问题出现难以定位的 bug。在本实验中使用 RISC-V 的官方测试集来对 CPU 进行全面的系统仿真。


11.3.3.1 官方测试集简介

RISC-V 社区开发了官方测试集，可以在 [github](https://github.com/riscv/riscv-tests) 上直接下载。地址是 <https://github.com/riscv/riscv-tests>。如果需要使用官方测试集，可以运行下列命令

```
1 $ git clone https://github.com/riscv/riscv-tests
2 $ cd riscv-tests
3 $ git submodule update --init --recursive
4 $ autoconf
5 $ ./configure --with-xlen=32
6 $ make isa
```

其中前 3 句是从 [github](https://github.com/riscv/riscv-tests) 上下载源代码，第 4-5 句是生成 `makefile`，最后是 `make` 测试集。

RISC-V 官方测试集针对不同的 RISC-V 指令变种都提供了测试。在本实验中，我们主要使用 `rv32ui` 也就是 RV32 的基本指令集，`u` 表示是用户态，`i` 表示是整数基本指令集。实验中采用的环境是无虚拟地址的环境，即只使用物理地址访问内存。所以，我们主要关注 `rv32ui-p` 开头的测试即可。

 由于官方测试集需要使用 `risc-v gcc` 工具链（如果 Build from source 约需下载 6GB 代码），我们提供了已经安装好 `risc-v` 工具链的 `vmWare` 虚拟机供大

家下载。该虚拟机中已经安装了 `riscv32-unknown-elf` 工具链（含 `CC`, `LD`, `OBJCOPY`, `OBJDUMP` 等全套编译命令）。同学们可以自行下载后使用。虚拟机用户名为 `exp`, 密码为 `lab2021`。虚拟机为 `server` 版, 不支持图形化界面。如果需要传输文件可以在宿主机上用 `sftp`（用户名和密码同上）与虚拟机连接, 可以使用 `FileZilla` 等 `ftp` 客户端连接虚拟机。虚拟机地址可以用 `ifconfig` 命令看到。

- 同学们也可以采用 PA 中提供的方案在 Ubuntu 下自行安装 `risc-v gcc` 工具链, 具体方法可以参考 PA2.2 中准备交叉编译环境。在 Ubuntu 下运行

```
1 apt-get install g++-riscv64-linux-gnu binutils-riscv64-  
linux-gnu
```

然后在 `sudo` 权限下修改以下文件:

```
1 --- /usr/riscv64-linux-gnu/include/bits/wordsize.h  
2 +++ /usr/riscv64-linux-gnu/include/bits/wordsize.h  
3 @@ -25,5 +25,5 @@  
4  #if __riscv_xlen == 64  
5  # define __WORDSIZE_TIME64_COMPAT32 1  
6  #else  
7  -# error "rv32i-based targets are not supported"  
8  +# define __WORDSIZE_TIME64_COMPAT32 0  
9  #endif
```

11.3.3.2 测试集移植

官方测试集的代码中使用了系统调用指令。因此我们需要对官方测试的代码进行一定的修改才可以用于我们的测试。

测试集的主要测试代码是用宏生成的汇编代码, 汇编源码在 `riscv-tests/isa/rv32ui` 和 `riscv-tests/isa/rv64ui` 下。宏定义在 `riscv-tests/isa/macro` 和 `riscv-tests/env/p` 下。当执行 `make isa` 后, 系统会编译各个测试用例的汇编代码, 在 `isa` 目录下生成可执行文件及反汇编 `dump` 文件。大家可以打开 `dump` 文件分析每个测试用例最终执行了哪些指令, 以及每条指令的地址。如下是 `rv32ui-p-andi` 测试用例中的一部分反汇编片段。

```
1 8000007c <test_2>:  
2 8000007c: ff0100b7          lui ra,0xff010
```

```

3 80000080:    f0008093                addi    ra,ra,-256 # ff00ff00 <
    _end+0x7f00df00>
4 80000084:    f0f0f713                andi    a4,ra,-241
5 80000088:    ff0103b7                lui     t2,0xff010
6 8000008c:    f0038393                addi    t2,t2,-256 # ff00ff00 <
    _end+0x7f00df00>
7 80000090:    00200193                li      gp,2
8 80000094:    1a771463                bne     a4,t2,8000023c <fail>
9
10 80000098 <test_3>:
11 80000098:    0ff010b7                lui     ra,0xff01
12 8000009c:    ff008093                addi    ra,ra,-16 # ff00ff0 <
    _start-0x700ff010>
13 800000a0:    0f00f713                andi    a4,ra,240
14 800000a4:    0f000393                li      t2,240
15 800000a8:    00300193                li      gp,3
16 800000ac:    18771863                bne     a4,t2,8000023c <fail>

```

我们对测试集的主要调整目标是两个：一是修改测试判断方法，改为不使用系统调用；二是生成我们要加载入 FPGA 的 hex 文件。我们主要修改两个文件来完成移植。首先是在 riscv-tests/isa/下的 Makefile 文件，我们需要 Build rules 部分增加以下内容：

```

1 RISCV_OBJCOPY ?= $(RISCV_PREFIX)objcopy -O verilog
2 RISCV_HEXGEN ?= 'BEGIN{output=0;}{ gsub("\r","",$(NF)); if ($(NF)
    ~/@/) {if ($(NF) ~/@80000000/) {output=code;} else {output=1-
    code;}; gsub("@","0x",$(NF)); addr=strtonum($(NF)); if (output==1)
    {printf "%08x\n", (addr%262144)/4;}} else {if (output==1) {
    for(i=1;i<NF;i+=4) print $(i+3)$(i+2)$(i+1)$(i);}}} '
3 RISCV_MIFGEN ?= 'BEGIN{printf "WIDTH=32;\nDEPTH=%d;\n\
    nADDRESS_RADIX=HEX;\nDATA_RADIX=HEX;\n\nCONTENT BEGIN\n",depth
    ; addr=0;} { gsub("\r","",$(NF)); if ($(NF) ~/@/) {gsub("@
    ","0x",$(NF));addr=strtonum($(NF));} else {printf "%04X : %s;\n",
    addr, $(NF); addr=addr+1;}} END{print "END\n";}'

```


此三句主要是说明如何产生我们所需的 FPGA 内存 hex 文件和 mif 文件。在编译完成后，我们会用 objdump 工具反汇编生成每个测试用例对应的 .dump 文件。但是该文件不能直接用于 FPGA 内存初始化。所以，我们需要自动生成针

对 verilog 的文本 .hex 文件和对 IP 核初始化的 mif 文件。自动生成分为三步：

首先用 objcopy 工具来生成 .tmp 文件，这个文件符合 verilog 格式要求，但是其中的数据是按 8bit 字节存储的，无法直接用于初始化 32bit 宽度的内存。

第二步是用 linux 的 awk 文本处理工具简单转换一下 verilog 的格式。awk 的指令请自行查找资料学习。本例中 awk 根据 output 变量判断是否需要打印输出。在读入一行后首先去除了最后一个 token 的换行符，然后判断是否是以 @ 开头的地址。如果是，则判断地址是否是 0x80000000 代码段起始地址。根据变量 code 来判断是生成代码初始化文件还是数据初始化文件。随后，对地址取低 18 位，并将地址除以四（从 byte 编址改成我们存储器中 4 字节编址），并打印修改后的地址。对于正常的数据行，awk 会将 token 分成四个一组重新打印。


第三步是用 awk 将文本 hex 格式改成 mif 格式，增加文件头尾和地址标识。

 **思考：**为什么我们将起始为 0x80000000 的代码段和数据段地址只取低 18 位来生成代码和数据存储器的初始化文件，我们的 CPU 仍然正确地执行并找到对应的数据？

Makefile 需要修改的第二部分是在 .dump 文件生成处：

```
1 %.dump: %
2     $(RISCV_OBJDUMP) $< > $@
3     $(RISCV_OBJCOPY) $< $<.tmp
4     awk -v code=1 $(RISCV_HEXGEN) $<.tmp > $<.hex
5     awk -v code=0 $(RISCV_HEXGEN) $<.tmp > $<_d.hex
6     awk -v depth=65536 $(RISCV_MIFGEN) $<.hex > $<.mif
7     awk -v depth=32768 $(RISCV_MIFGEN) $<_d.hex > $<_d.mif
```

此处第 2-3 句分别用 objdump 和 objcopy 工具生成文本文件。第 4 句用 awk 将 .tmp 文件转换成指令存储器的初始化 hex 文件，32bit 位宽，第 5 句用于提取数据存储器的初始化 hex 文件。随后，第 6 和第 7 句分别将指令存储器初始化文件和数据存储器初始化文件转换为 mif 文件。我们这里指令存储器大小是 256k，数据存储器是 128k。

 **思考：**如果数据存储器是用 4 片 8bit 存储器来实现的，如何生成 4 片存储器对应的初始化文件？

当然，我们还对 Makefile 进行了其他改动，减少了不必要的文件输出，我

们提供了 Makefile 的补丁文件 Makefile.patch。请在完成 riscv-tests 的 configure 之后（官方测试集下载完成第 5 行命令 `./configure --with-xlen=32` 后），将 Makefile.patch 文件拷贝到 riscv-tests/isa 目录下，并在该目录下运行：

```
1 $ patch < Makefile.patch
```

请在 patch 之前保留好原始文件的备份，并在 patch 之后打开 Makefile 观察补丁是否正确安装。

我们需要修改的第二个文件是位于 riscv-tests/env/p 目录下的 riscv_test.h。这个文件主要是定义了在使用物理地址访问内存的条件下，测试代码所用的宏。我们修改的主要目标是去除宏中用到系统调用的方法，并使用特殊的 Magic Number 来指示仿真运行结束。首先我们将测试启动部分的 RVTEST_CODE_BEGIN 宏修改掉：

```
1 #define RVTEST_CODE_BEGIN \
2     .section .text.init; \
3     .align 6; \
4     .weak stvec_handler; \
5     .weak mtvec_handler; \
6     .globl _start; \
7 _start: \
8     /* reset vector */ \
9     /*j reset_vector;*/ \
10    INIT_XREG \
11    .align 2; \
12
13 //trap_vector: \
```

这里主要将原来跳转到系统初始化的部分给注释掉了，只保留了 INIT_XREG（所有寄存器清零）。

之后，我们对测试中判断是否通过的部分进行修改。测试集中会对每一步运算进行正确性判断，如果错误会跳转到 FAIL 部分执行，如果正确是跳转到 PASS 部分执行。我们将标准集中用系统调用的判断宏修改为如下代码


```
1 #define RVTEST_PASS \
2     li a0, 0xc0ffee; \
3     .word 0xdead10cc \
4
5 // fence; \
```

```

6          li TESTNUM, 1;          \
7          li a7, 93;              \
8          li a0, 0;               \
9          ecall
10
11 #define TESTNUM gp
12 #define RVTEST_FAIL              \
13          li a0, 0xdeaddead;      \
14          .word 0xdead10cc
15
16 //          fence;              \
17 1:      beqz TESTNUM, 1b;          \
18          sll TESTNUM, TESTNUM, 1; \
19          or TESTNUM, TESTNUM, 1;  \
20          li a7, 93;              \
21          addi a0, TESTNUM, 0;     \
22          ecall

```

在修改后的代码中，如果 PASS 会将寄存器 a0 中的数据设置为 32'h00c0ffee，表示测试通过。第 3 句中在指令存储中预置了 32'hdead10cc 说明测试运行完成，在仿真中发现取指令时获取了这个数字之后就可以判断仿真完成了。同样的，如果是 FAIL 的情况，我们将 a0 置为 32'hdeaddead，并随后停止仿真。

 **思考：**为什么正常的 rv32i 指令序列中肯定不会出现 32'hdead10cc？

我们也提供了 riscv_test.h 对应的 patch，请将补丁文件拷贝到 riscv-tests/env/p 目录下对该文件进行修改。

在完成补丁后可以在 riscv-tests 目录下运行 **make isa**（直接 make 无法编译 benchmark 部分）进行编译。编译后会在 riscv-tests/isa 目录下产生每个测试用例对应的 .dump 及 FPGA 内存初始化文件。请通过 sftp 将所需的内存初始化文件拷贝出来，放置到自己的 Quartus 工程对应的仿真目录下准备进行仿真。



注意，官方测试集基于跳转指令来判断运行是否正确。如果跳转指令实现有问题，有可能会在 CPU 有 bug 时也输出正确的结果。尤其是某些情况下，信号为不定值 X 的时候，branch 指令可能会错误判断，请注意排除此类问题。

11.3.3.3 测试集 TestBench

我们需要修改 Testbench 支持对官方测试集的仿真。主要增加了以下辅助任务：

```
1 integer maxcycles =10000;
2
3 task run;
4     integer i;
5     begin
6         i = 0;
7         while( (mycpu.instr!=32'hdead10cc) && (i<maxcycles))
8             begin
9                 step();
10                i = i+1;
11            end
12        end
13    endtask
```

代码运行任务 run 会一直用单步运行代码，直到遇到我们定义的代码终止信号为止。如果代码一直不终止也会在给定最大运行周期后停止仿真。

对仿真结果测试是通过对仿真结束后 a0 寄存器中数据是否符合预期来进行判断的。当然如果程序不终止，或者 a0 数据不正常也会报错。

```
1 task checkmagnum;
2     begin
3         if(numcycles>maxcycles)
4             begin
5                 $display("!!!Error:test case %s does not terminate!", testcase);
6             end
7         else if(mycpu.myregfile.regs[10]==32'hc0ffee)
8             begin
9                 $display("OK:test case %s finshed OK at cycle %d.",
10                    testcase, numcycles-1);
11            end
12         else if(mycpu.myregfile.regs[10]==32'hdeaddead)
13             begin
14                 $display("!!!ERROR:test case %s finshed with error in cycle %d.",
15                    testcase, numcycles-1);
```

```
16         end
17     else
18     begin
19         $display("!!!ERROR:test case %s unknown error in cycle %d.",
20                 testcase, numcycles-1);
21     end
22 end
23 endtask
```

数据存储可以用我们生成的 **hex** 文件进行初始化，仿真时可以用我们提供的 **ram** 模块替代 **IP** 核生成的模块。一般只有在访存指令的测试时才需要初始化数据存储

```
1 task loaddatamem;
2     begin
3         $readmemh({testcase, "_d.hex"}, datamem.mymem.ram);
4     end
5 endtask
```

我们也提供了一个简单的可以执行单个测试用例的任务

```
1 task run_riscv_test;
2     begin
3         loadtestcase();
4         loaddatamem();
5         resetcpu();
6         run();
7         checkmagnum();
8     end
9 endtask
```

所以在仿真过程中我们只需要按顺序执行所有需要的仿真即可：

```
1     testcase = "rv32ui-p-simple";
2     run_riscv_test();
3     testcase = "rv32ui-p-add";
4     run_riscv_test();
5     testcase = "rv32ui-p-addi";
6     run_riscv_test();
7     testcase = "rv32ui-p-and";
8     run_riscv_test();
```



```

Transcript
# OK:test case          rv32ui-p-lhu finished OK at cycle      255.
# ---Begin test case          rv32ui-p-lui-----
# OK:test case          rv32ui-p-lui finished OK at cycle      56.
# ---Begin test case          rv32ui-p-lw-----
# OK:test case          rv32ui-p-lw finished OK at cycle     258.
# ---Begin test case          rv32ui-p-or-----
# OK:test case          rv32ui-p-or finished OK at cycle     479.
# ---Begin test case          rv32ui-p-ori-----
# OK:test case          rv32ui-p-ori finished OK at cycle     196.
# ---Begin test case          rv32ui-p-sb-----
# OK:test case          rv32ui-p-sb finished OK at cycle     421.
# ---Begin test case          rv32ui-p-sh-----
# OK:test case          rv32ui-p-sh finished OK at cycle     474.
# ---Begin test case          rv32ui-p-sll-----
# OK:test case          rv32ui-p-sll finished OK at cycle     484.
# ---Begin test case          rv32ui-p-slli-----
# OK:test case          rv32ui-p-slli finished OK at cycle     232.
# ---Begin test case          rv32ui-p-slt-----
# OK:test case          rv32ui-p-slt finished OK at cycle     450.
# ---Begin test case          rv32ui-p-slti-----
# OK:test case          rv32ui-p-slti finished OK at cycle     228.
# ---Begin test case          rv32ui-p-sltiu-----
# OK:test case          rv32ui-p-sltiu finished OK at cycle     228.
# ---Begin test case          rv32ui-p-sltui-----
# OK:test case          rv32ui-p-sltui finished OK at cycle     450.
# ---Begin test case          rv32ui-p-sra-----
# OK:test case          rv32ui-p-sra finished OK at cycle     503.
# ---Begin test case          rv32ui-p-srai-----
# OK:test case          rv32ui-p-srai finished OK at cycle     247.
# ---Begin test case          rv32ui-p-srl-----
# OK:test case          rv32ui-p-srl finished OK at cycle     497.
# ---Begin test case          rv32ui-p-srli-----
# OK:test case          rv32ui-p-srli finished OK at cycle     241.
# ---Begin test case          rv32ui-p-sub-----
# OK:test case          rv32ui-p-sub finished OK at cycle     448.
# ---Begin test case          rv32ui-p-sw-----
# OK:test case          rv32ui-p-sw finished OK at cycle     481.
# ---Begin test case          rv32ui-p-xor-----
# OK:test case          rv32ui-p-xor finished OK at cycle     478.
# ---Begin test case          rv32ui-p-xori-----
# OK:test case          rv32ui-p-xori finished OK at cycle     198.
# ** Note: $stop      : D:/digitaldesign/2021/riscvcpu/testbench_cpu.vt(271)
# Time: 229590 ns  Iteration: 0  Instance: /testbench_cpu
# Break in NamedBeginStat TestBench at D:/digitaldesign/2021/riscvcpu/testbench_cpu.vt line 271

```

图 11-5: 使用官方测试集的 CPU 的仿真输出

在仿真过程中可以暂时注释 \$monitor 任务，只有在出错时再检查具体测试用例为何出错。图 11-5 显示了利用官方测试集进行仿真的输出结果示例。

11.3.4 上板测试

在设计过程中建议 CPU 预留测试数据接口，上板测试前可以将对应接口接至板上的 LED 或七段数码管，显示 CPU 的内部状态。具体选择测试接口输出哪些内容可以自行决定，可以考虑 PC、寄存器结果，控制信号等等。在初次上板测试时，可以将 CPU 时钟连接到板载的 KEY 按钮上，每按一下单步执行一个周期，方便进行调试。

对于单周期 CPU，由于需要在一个周期内完成指令执行的所有步骤，很可能不能以 50MHz 运行。请观察你的 CPU 综合后 Timing Analysis 结果是否存在时序不满足，即某些 model 下 Setup Slack 为负数。此时，可以考虑调整设计减少关键路径时延，或者降低主频。

11.4 实验验收要求

11.4.1 在线测试

请自行完成单周期 CPU 的实现，并通过在线测试中单周期 CPU 的功能测试及官方测试部分。

必做 单周期功能测试

必做 单周期 CPU 官方测试

🔔 **注意** 由于我们在线测试仅针对单周期 CPU，如果同学实现了多周期或流水线 CPU 可能会在时序上与在线测试结果不一致。可以自行参考课程网站上提供的 test bench 自行编写测试代码，需要完成 RISC-V 官方测试集中 rv32ui-p 开头的所有指令的测试，由助教现场验收后可以通过。

🔔 **致谢：**感谢 2019 级李晗及高灏同学在 RISC-V 工具链方面的探索。