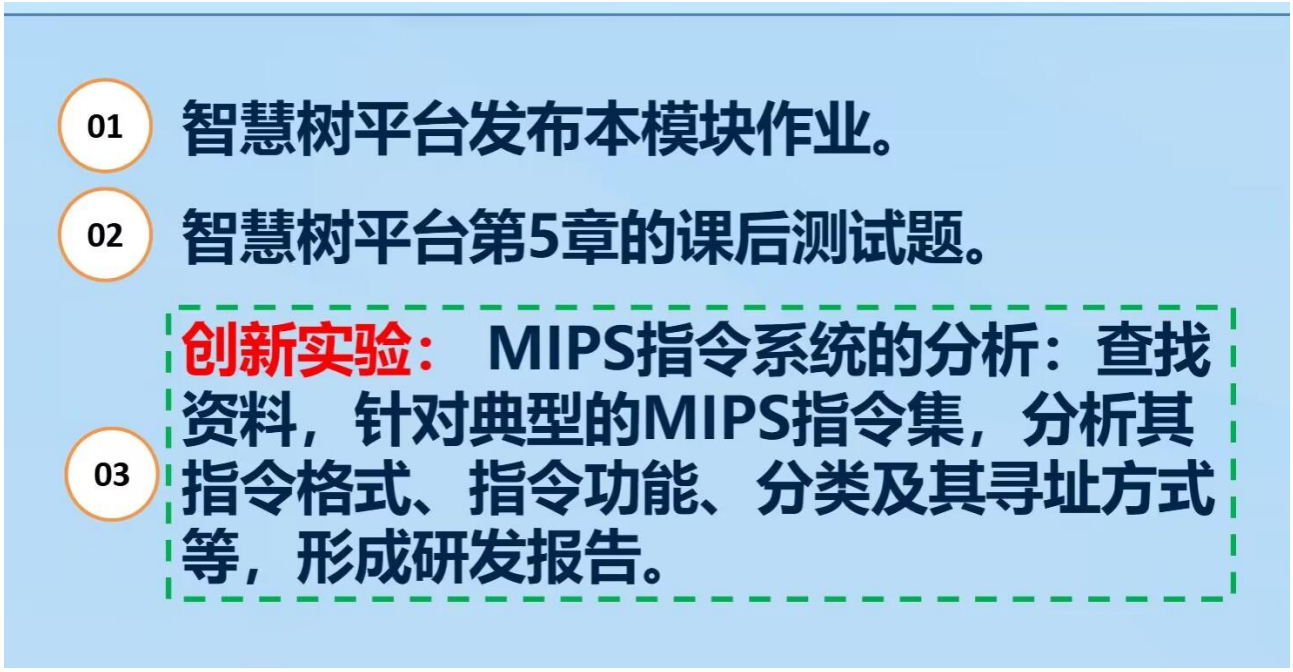
山东大学计算机科学与技术学院

计算机组成与设计课程实验报告

学号：202300130183	姓名： 宋浩宇	班级： 23 级人工智能
实验题目： 创新实验		
实验学时： 2	实验日期： 2024/12/24	

实验目的：



硬件环境：

13th Gen Intel(R) Core(TM) i9-13980HX 2.20 GHz

32.0 GB (31.6 GB 可用)

康芯 KX-CDS FPGA 平台

芯片 Cyclong IV E EP4CE6E22C8

软件环境：

Windows 11 家庭中文版 23H2 22631.4317

Intel Quartus II 13.0sp1(64 bit)

实验内容与设计：

1、实验内容

MIPS 指令系统的分析:查找资料，针对典型的 MIPS 指令集，分析其指令格式、指令功能、分类及其寻址方式等，形成研发报告。

2、实验参考资料

MIPS32 指令集

MIPS 指令可以分成以下各类：

空操作 no-op;

寄存器 / 寄存器传输：用得很广，包括条件传输在内；

常数加载：作为数值和地址的整型立即数；

算术 / 逻辑指令；

整数乘法、除法和求余数；

整数乘加；

加载和存储；

跳转、子程序调用和分支；

断点和自陷；

CP0 功能：CPU 控制指令

浮点；

用户态的受限访问：rdhwr 和 synci

注：64 位版本开头以“d”表示，无符号数以“u”结尾，立即数通常以“i”结尾，字节操作以“b”结尾，双字操作以“d”结尾，字操作以“w”结尾

1、空操作：nop:相当于 sll zero,zero,o，

ssnop: equals sll zero,zero,1.

这个指令不得与其它指令同时发送，这样就保证了其运行要花费至少一个时钟周期。这在简单的流水线的 CPU 上无关紧要，但在复杂些的实现上对于实现强制的延时很有用。

2、寄存器 / 寄存器传送：

move: 通常用跟$zero 寄存器的 or 来实现，或者用 addu。

movf, movt, movn, movz: 条件传送。

3、常数加载：

dla、la: 用来加载程序中某些带标号的位置或者变量的地址的宏指令；

dli、li: 装入立即数常数，这是一个宏指令；

lui: 把立即数加载到寄存器高位。

4、算术 / 逻辑运算：

add、addi、dadd、daddi、addu、addiu、daddu、daddiu、dsub、sub、subu：加法指令和减法指令；

abs，dabs：绝对值；

dneg、neg、negu：取相反数；

and、andi、or、ori、xor、nor：逐位逻辑操作指令；

drol、rol、ror：循环移位指令；

sll、srl、sra：移位。

5、条件设置指令：

slt、slti、sltiu、sltu、seq、sge、sle、sne：条件设置。

6、整数乘法、除法和求余数：
div、mul、rem 等等。

7、整数乘加（累加）：
mad 等。

8、加载和存储：
lb、ld、ldl、ldr、sdl、sdr、lh、lhu、ll、sc、pref、sb 等操作。

9、浮点加载和存储：
l.d、l.s、s.d、s.s 等

常用 MIPS 指令集及格式：

MIPS 指令集(共 31 条）									
助记符	指令格式						示例	示例含义	操作及其解释
Bit #	31..26	25..21	20..16	15..11	10..6	5..0			
R-type	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+$3	rd <- rs + rt ；其中 rs＝$2，rt=$3, rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt ；其中 rs＝$2，rt=$3, rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt ；其中 rs＝$2，rt=$3, rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-$3	rd <- rs - rt ；其中 rs＝$2，rt=$3, rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2 & $3	rd <- rs & rt ；其中 rs＝$2，rt=$3, rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2 | $3	rd <- rs | rt ；其中 rs＝$2，rt=$3, rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2 ^ $3	rd <- rs xor rt ；其中 rs＝$2，rt=$3, rd=$1(异或）
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1=~($2 | $3)	rd <- not(rs | rt) ；其中 rs＝$2，rt=$3, rd=$1(或非）
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ；其中 rs＝$2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ；其中 rs＝$2，rt=$3, rd=$1 (无符号数）

sll	000000	0000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt ； shamt 存放移位的位数， 也就是指令中的立即数，其中 rt=$2, rd=$1
srl	000000	0000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ； (logical) ，其中 rt=$2, rd=$1
sra	000000	0000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	rd <- rt >> shamt ；(arithmetic) 注意符号位保留 其中 rt=$2, rd=$1
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	rd <- rt << rs ；其中 rs＝$3，rt=$2, rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ； (logical)其中 rs＝$3，rt=$2, rd=$1
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ； (arithmetic) 注意符号位保留 其中 rs＝$3，rt=$2, rd=$1
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	PC <- rs
I-type	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate ；其中 rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,100	$1=$2+100	rt <- rs + (zero-extend)immediate ； 其中 rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ； 其中 rt=$1,rs=$2
ori	001101	rs	rt	immediate			andi $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ；其中 rt=$1,rs=$2
xori	001110	rs	rt	immediate			andi $1,$2,10	$1=$2 ^ 10	rt <- rs xor (zero-extend)immediate ； 其中 rt=$1,rs=$2
lui	001111	0000	rt	immediate			lui $1,100	$1=100*65536	rt <- immediate*65536 ；将 16 位立即数放到目标寄存器高 16 位，目标寄存器的低 16 位填 0
lw	100011	rs	rt	immediate			lw $1,10($2)	$1=memory[$2 +10]	rt <- memory[rs + (sign-extend)immediate] ； rt=$1,rs=$2
sw	101011	rs	rt	immediate			sw $1,10($2)	memory[$2+10] =$1	memory[rs + (sign-extend)immediate] <- rt ； rt=$1,rs=$2

beq	000100	rs	rt	immediate	beq $1,$2,10	if($1==$2) goto PC+4+40	if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2
bne	000101	rs	rt	immediate	bne $1,$2,10	if($1!=$2) goto PC+4+40	if (rs != rt) PC <- PC+4 + (sign-extend)immediate<<2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if (rs <(sign-extend)immediate) rt=1 else rt=0 ； 其中 rs=$2，rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if (rs <(zero-extend)immediate) rt=1 else rt=0 ； 其中 rs=$2，rt=$1
J-type	op	address					
j	000010	address			j 10000	goto 10000	PC <- (PC+4)[31..28],address,0,0 ；address=10000/4
jal	000011	address			jal 10000	$31<-PC+4; goto 10000	$31<-PC+4 ； PC <- (PC+4)[31..28],address,0,0 ； address=10000/4

更全的 MIPS 汇编指令
Arithmetic Instructions
abs des, src1 # des gets the absolute value of src1.
add(u) des, src1, src2 # des gets src1 + src2.
addi $t2,$t3,5 # $t2 = $t3 + 5 加 16 位立即数
addiu $t2,$t3,5 # $t2 = $t3 + 5 加 16 位无符号立即数
sub(u) des, src1, src2 # des gets src1 - src2.
div(u) src1, reg2 # Divide src1 by reg2, leaving the quotient in register
lo and the remainder in register hi.
div(u) des, src1, src2 # des gets src1 / src2.
mul des, src1, src2 # des gets src1 * src2.
mulo des, src1, src2 # des gets src1 * src2, with overflow.
mult(u) src1, reg2 # Multiply src1 and reg2, leaving the low-order word
in register lo and the high-order word in register hi.
rem(u) des, src1, src2 # des gets the remainder of dividing src1 by src2.
neg(u) des, src1 # des gets the negative of src1.
and des, src1, src2 # des gets the bitwise and of src1 and src2.
nor des, src1, src2 # des gets the bitwise logical nor of src1 and src2.
not des, src1 # des gets the bitwise logical negation of src1.
or des, src1, src2 # des gets the bitwise logical or of src1 and src2.
xor des, src1, src2 # des gets the bitwise exclusive or of src1 and src2.
rol des, src1, src2 # des gets the result of rotating left the contents of src1 by src2 bits.
ror des, src1, src2 # des gets the result of rotating right the contents of src1 by src2 bits.
sll des, src1, src2 # des gets src1 shifted left by src2 bits.
sra des, src1, src2 # Right shift arithmetic.
srl des, src1, src2 # Right shift logical.
sllv des, src1, src2 # $t0 = $t1 << $t3，shift left logical
srlv des, src1, src2 # $t0 = $t1 >> $t3，shift right logical
srav des, src1, src2 # $t0 = $t1 >> $t3，shift right arithm.
Comparison Instructions
seq des, src1, src2 # des 1 if src1 = src2, 0 otherwise.
sne des, src1, src2 # des 1 if src1 != src2, 0 otherwise.
sge(u) des, src1, src2 # des 1 if src1 >= src2, 0 otherwise.
sgt(u) des, src1, src2 # des 1 if src1 > src2, 0 otherwise.
sle(u) des, src1, src2 # des 1 if src1 <= src2, 0 otherwise.
slt(u) des, src1, src2 # des 1 if src1 < src2, 0 otherwise.
slti $t1,$t2,10 # 与立即数比较
Branch and Jump Instructions
b lab # Unconditional branch to lab.
beq src1, src2, lab # Branch to lab if src1 = src2 .
bne src1, src2, lab # Branch to lab if src1 != src2 .
bge(u) src1, src2, lab # Branch to lab if src1 >= src2 .
bgt(u) src1, src2, lab # Branch to lab if src1 > src2 .
ble(u) src1, src2, lab # Branch to lab if src1 <= src2 .
blt(u) src1, src2, lab # Branch to lab if src1 < src2 .
beqz src1, lab # Branch to lab if src1 = 0.
bnez src1, lab # Branch to lab if src1 != 0.
bgez src1, lab # Branch to lab if src1 >= 0.
bgtz src1, lab # Branch to lab if src1 > 0.
blez src1, lab # Branch to lab if src1 <= 0.
bltz src1, lab # Branch to lab if src1 < 0.
bgezal src1, lab # If src1 >= 0, then put the address of the next instruction
into $ra and branch to lab.
bgtzal src1, lab # If src1 > 0, then put the address of the next instruction
into $ra and branch to lab.
bltzal src1, lab # If src1 < 0, then put the address of the next instruction
into $ra and branch to lab.

```
j label # Jump to label lab.
jr src1 # Jump to location src1.
jal label # Jump to label lab, and store the address of the next instruction in $ra.
jalr src1 # Jump to location src1, and store the address of the next instruction in $ra.
Load, Store, and Data Movement
(reg) $ Contents of reg.
const $ A constant address.
const(reg) $ const + contents of reg.
symbol $ The address of symbol.
symbol+const $ The address of symbol + const.
symbol+const(reg) $ The address of symbol + const + contents of reg.
la des, addr # Load the address of a label.
lb(u) des, addr # Load the byte at addr into des.
lh(u) des, addr # Load the halfword at addr into des.
li des, const # Load the constant const into des.
lui des, const # Load the constant const into the upper halfword of des,
# and set the lower halfword of des to 0.
lw des, addr # Load the word at addr into des.
lwl des, addr
lwr des, addr
ulh(u) des, addr # Load the halfword starting at the (possibly unaligned) address addr into
des.
ulw des, addr # Load the word starting at the (possibly unaligned) address addr into des.
sb src1, addr # Store the lower byte of register src1 to addr.
sh src1, addr # Store the lower halfword of register src1 to addr.
sw src1, addr # Store the word in register src1 to addr.
swl src1, addr # Store the upper halfword in src to the (possibly unaligned) address addr.
swr src1, addr # Store the lower halfword in src to the (possibly unaligned) address addr.
ush src1, addr # Store the lower halfword in src to the (possibly unaligned) address addr.
usw src1, addr # Store the word in src to the (possibly unaligned) address addr.
move des, src1 # Copy the contents of src1 to des.
mfhi des # Copy the contents of the hi register to des.
mflo des # Copy the contents of the lo register to des.
mthi src1 # Copy the contents of the src1 to hi.
mtlo src1 # Copy the contents of the src1 to lo.
Exception Handling
rfe # Return from exception.
syscall # Makes a system call. See 4.6.1 for a list of the SPIM system calls.
break const # Used by the debugger.
nop # An instruction which has no effect (other than taking a cycle to execute).
```

3、实验结果

由于原文使用 markdown 格式编写，因此在此附上 markdown 源码和 markdown 导出的 pdf

markdown 源码：

```
### 一、指令格式分析

**R 型指令 (Register Format)**:
- 结构: op(6 位) rs(5 位) rt(5 位) rd(5 位) shamt(5 位) funct(6 位)
- 示例: `add $rd, $rs, $rt`
- 说明: R 型指令主要用于寄存器间操作,例如算术运算、逻辑运算和移位操作。`op` 字段表示操作码,`rs` 和 `rt` 是源寄存器,`rd` 是目标寄存器,`shamt` 用于移位操作的位移量,`funct` 字段进一步细化操作类型。

**I 型指令 (Immediate Format)**:
```

- 结构: op(6 位) rs(5 位) rt(5 位) immediate(16 位)
- 示例: `addi $rt, $rs, immediate`
- 说明: I 型指令用于处理立即数,常见于条件分支、加载和存储操作。`immediate` 字段可以表示一个常数值或偏移量。

J 型指令 (Jump Format):
- 结构: op(6 位) address(26 位)
- 示例: `j address`
- 说明: J 型指令用于跳转,`address` 字段提供了一个跳转地址,需要注意的是,这个地址需要经过处理以形成完整的 32 位地址。

二、指令功能及分类

1. 空操作指令
- `nop`: 无操作指令,用于填充指令流水线。
- `ssnop`: 超级标量无操作指令,用于特定处理器架构的流水线控制。

2. 寄存器/寄存器传送指令
- `move`: 将一个寄存器的值传送到另一个寄存器。
- `movf`, `movt`, `movn`, `movz`: 条件传送指令,基于条件码或寄存器内容进行传送。

3. 常数加载指令
- `dla`, `la`: 加载地址到寄存器,`dla` 用于 64 位地址。
- `dli`, `li`, `lui`: 加载立即数到寄存器,`lui` 加载立即数到寄存器的高 16 位。

4. 算术/逻辑运算指令
- 加法、减法、乘法、除法、位运算等指令,有符号和无符号操作的变体。
- `sll`, `srl`, `sra`: 移位操作,逻辑左移、逻辑右移、算术右移。

5. 条件设置指令
- `slt`, `sltiu`, `sltu` 等: 比较两个值并设置目标寄存器为 0 或 1。

6. 整数乘法/除法/求余指令
- `div`, `mul`, `rem`: 进行整数的乘法、除法和求余运算。

7. 加载/存储指令
- `lb`, `lh`, `lw` 等: 从内存加载数据到寄存器,不同字节宽度。
- `sb`, `sh`, `sw` 等: 将寄存器中的数据存储到内存。

8. 跳转/子程序调用/分支指令
- `jr`, `j`, `jal`: 跳转指令,包括无条件跳转和调用子程序。
- `beq`, `bne`: 条件分支,根据两个寄存器是否相等或不等进行跳转。

9. 其他指令
- `rfe`: 从异常返回。
- `syscall`, `break`: 系统调用和断点指令。

三、寻址方式

- **寄存器寻址**：直接从寄存器中获取操作数。
- **立即数寻址**：使用指令中的立即数作为操作数或偏移量。
- **绝对地址寻址**：直接跳转到指定的内存地址。
- **基址寻址**：使用寄存器的值加上立即数作为内存地址，常用于加载和存储操作。

四、其他特点

- **有符号与无符号操作**：许多指令有有符号和无符号版本，以处理不同类型的整数数据。
- **双精度版本**：一些指令有 64 位操作的版本，如 `dadd` 等。
- **字节/半字/字寻址**：加载和存储指令支持不同大小的数据传输。
- **条件传送**：根据条件码或寄存器的值进行条件传送。
- **逻辑移位和算术移位**：移位指令有逻辑和算术两种方式，以处理不同的移位需求。

导出的 pdf：

一、指令格式分析

R 型指令 (Register Format):

- 结构: op(6位) rs(5位) rt(5位) rd(5位) shamt(5位) funct(6位)
- 示例: `add $rd, $rs, $rt`
- 说明: R 型指令主要用于寄存器间操作,例如算术运算、逻辑运算和移位操作。`op` 字段表示操作码, `rs` 和 `rt` 是源寄存器, `rd` 是目标寄存器, `shamt` 用于移位操作的位移量, `funct` 字段进一步细化操作类型。

I 型指令 (Immediate Format):

- 结构: op(6位) rs(5位) rt(5位) immediate(16位)
- 示例: `addi $rt, $rs, immediate`
- 说明: I 型指令用于处理立即数,常见于条件分支、加载和存储操作。`immediate` 字段可以表示一个常数值或偏移量。

J 型指令 (Jump Format):

- 结构: op(6位) address(26位)
- 示例: `j address`
- 说明: J 型指令用于跳转, `address` 字段提供了一个跳转地址,需要注意的是,这个地址需要经过处理以形成完整的32位地址。

二、指令功能及分类

1. 空操作指令

- `nop` : 无操作指令,用于填充指令流水线。
- `ssnop` : 超级标量无操作指令,用于特定处理器架构的流水线控制。

2. 寄存器/寄存器传送指令

- `move` : 将一个寄存器的值传送到另一个寄存器。
- `movf` , `movt` , `movn` , `movz` : 条件传送指令,基于条件码或寄存器内容进行传送。

3. 常数加载指令

- `dla` , `la` : 加载地址到寄存器, `dla` 用于 64 位地址。
- `dli` , `li` , `lui` : 加载立即数到寄存器, `lui` 加载立即数到寄存器的高16位。

4. 算术/逻辑运算指令

- 加法、减法、乘法、除法、位运算等指令,有符号和无符号操作的变体。

- `sll`, `srl`, `sra`: 移位操作,逻辑左移、逻辑右移、算术右移。

5. 条件设置指令

- `slt`, `sltiu`, `sltu` 等: 比较两个值并设置目标寄存器为 0 或 1。

6. 整数乘法/除法/求余指令

- `div`, `mul`, `rem`: 进行整数的乘法、除法和求余运算。

7. 加载/存储指令

- `lb`, `lh`, `lw` 等: 从内存加载数据到寄存器,不同字节宽度。
- `sb`, `sh`, `sw` 等: 将寄存器中的数据存储到内存。

8. 跳转/子程序调用/分支指令

- `jr`, `j`, `jal`: 跳转指令,包括无条件跳转和调用子程序。
- `beq`, `bne`: 条件分支,根据两个寄存器是否相等或不等进行跳转。

9. 其他指令

- `rfe`: 从异常返回。
- `syscall`, `break`: 系统调用和断点指令。

三、寻址方式

- **寄存器寻址**: 直接从寄存器中获取操作数。
- **立即数寻址**: 使用指令中的立即数作为操作数或偏移量。
- **绝对地址寻址**: 直接跳转到指定的内存地址。
- **基址寻址**: 使用寄存器的值加上立即数作为内存地址,常用于加载和存储操作。

四、其他特点

- **有符号与无符号操作**: 许多指令有有符号和无符号版本,以处理不同类型的整数数据。
- **双精度版本**: 一些指令有 64 位操作的版本,如 `dadd` 等。
- **字节/半字/字寻址**: 加载和存储指令支持不同大小的数据传输。
- **条件传送**: 根据条件码或寄存器的值进行条件传送。
- **逻辑移位和算术移位**: 移位指令有逻辑和算术两种方式,以处理不同的移位需求。

结论分析与体会：
MIPS 指令集的设计巧妙地平衡了硬件复杂度与性能需求，体现了计算机体系结构领域的深刻洞察和创新。它通过采用精简指令集、固定长度指令、流水线处理、寄存器-寄存器操作和加载/存储体系结构等策略，实现了高效的计算能力与简化的硬件设计，同时保持了指令操作的可预测性和直观性。这种设计不仅优化了 CPU 的性能和功耗，还为计算机科学教育提供了直观的学习工具，展现了人类在追求计算效率、简化设计、以及教育普及方面的智慧和创新精神。