

# Project4 VerilogHDL 完成单周期处理器开发

## 一、 设计说明

1. 处理器应 MIPS-Lite3 指令集。
  - a) MIPS-Lite3 = {MIPS-Lite2, addi, addiu, slt, j, jal, jr}。
  - b) MIPS-Lite2 指令集: addu, subu, ori, lw, sw, beq, lui。
  - c) addi 可以不支持溢出。
2. 处理器为单周期设计。

## 二、 设计要求

3. 单周期处理器由 datapath(数据通路)和 controller(控制器)组成。
  - a) 数据通路由如下 module 组成: PC(程序计数器)、NPC(NextPC 计算单元)、GPR (通用寄存器组, 也称为寄存器文件、寄存器堆)、ALU(算术逻辑单元)、EXT(扩展单元)、IM(指令存储器)、DM(数据存储器)。
  - b) IM: 容量为 4KB(32bit×1024 字)。
  - c) DM: 容量为 4KB(32bit×1024 字)。
4. Figure1 为供你参考的数据通路架构图。
  - a) 我们不确保 Figure1 是完全正确的; 我们也不确保 Figure1 能够满足 MIPS-Lite3
  - b) 鼓励你从数据通路的功能合理划分的角度自行设计更好的数据通路架构。
  - c) 如果你做了比较大的调整, 请务必注意不要与要求 5 矛盾。

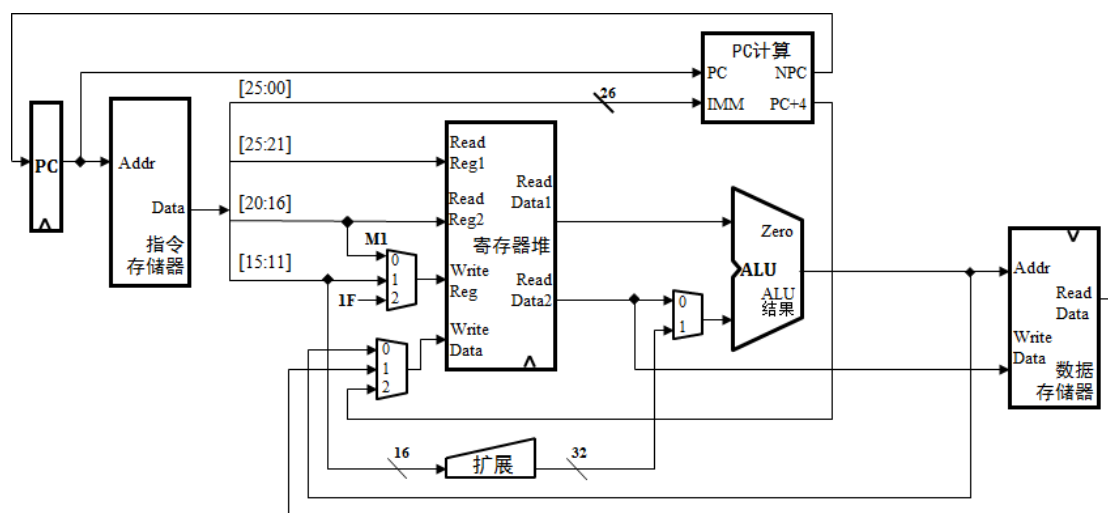


Figure1 数据通路(供参考)

5. 整个 project 必须采用模块化和层次化设计。

- a) Figure2 为参考的目录结构和文件命名。其中红色框的目录名称及文件名称不允许调整(control、datapath、mips.v、code.txt 都属于同一层；control 目录下包括 ctrl.v；datapath 目录下包括 im.v、dm.v，等等)。

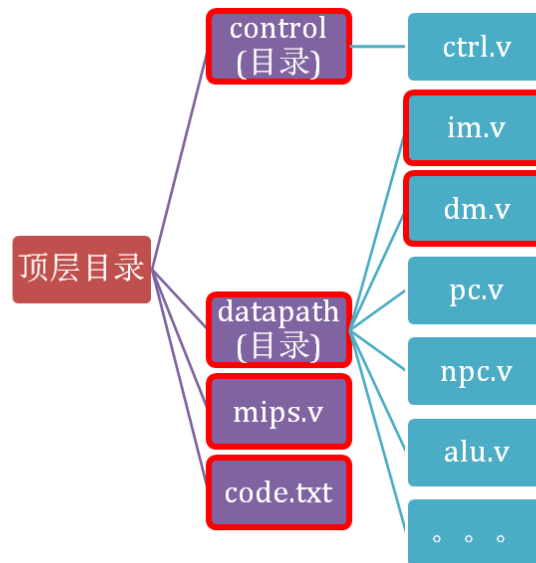


Figure2 参考的 project 目录组织

- b) 顶层设计文件命名为 mips.v。
- c) 建议 datapath 中的每个 module 都由一个独立的 VerilogHDL 文件组成。
- d) 建议所有 mux (包括不同位数、不同端口数等) 都建模在一个 mux.v 中。  
可以有多个 module。
6. code.txt 中存储的是指令码

- a) 用 VerilogHDL 建模 IM 时，必须以读取文件的方式将 code.txt 中指令加载至 IM 中。

- b) code.txt 的格式如 Figure3 所示。每条指令占用 1 行，指令二进制码以文本方式存储。

```
1 34010001
2 34020008
3 34100000
4 34110008
5 3c12aabb
6 12200009
```

Figure3 code.txt 文件格式

7. 为使得代码更加清晰可读，建议多使用宏定义，并将宏定义组织在合理的头文件中。

8. PC 复位后初值为 0x0000\_3000，目的是与 MARS 的 Memory Configuration 相配合。
- a) 教师用测试程序将通过 MARS 产生，其配置模式如 Figure4 所示。



Figure4 MIPS 存储配置模式(MARS memory configuration)

三、 模块定义【WORD】

9. 仿照下面给出的 PC 模块定义，给出所有功能部件的模块定义。
10. PC 模块定义(参考样例)

(1) 基本描述

PC 主要功能是完成输出当前指令地址并保存下一条指令地址。复位后，PC 指向 0x0000\_3000，此处为第一条指令的地址。

(2) 模块接口

| 信号名       | 方向 | 描述      |
|-----------|----|---------|
| NPC[31:2] | I  | 下条指令的地址 |
| clk       | I  | 时钟信号    |

|          |   |                         |
|----------|---|-------------------------|
| Reset    | I | 复位信号。<br>1: 复位<br>0: 无效 |
| PC[31:2] | O | 30 位指令存储器地址(最低 2 位省略)   |

### (3) 功能定义

| 序号 | 功能名称       | 功能描述                          |
|----|------------|-------------------------------|
| 1  | 复位         | 当复位信号有效时，PC 被设置为 0x0000_3000。 |
| 2  | 保存 NPC 并输出 | 在每个 clock 的上升沿保存 NPC，并输出。     |

11. 下列模块必须严格满足如下的接口定义：

a) 你必须在 VerilogHDL 设计中建模这 3 个模块。

b) 不允许修改模块名称、端口各信号以及变量的名称/类型/位宽。

| 文件     | 模块接口定义   |
|--------|--|
| mips.v | <pre> module mips(clk, rst) ;     input          clk ;    // clock     input          rst ;    // reset </pre>   |
| im.v   | <pre> im_4k( addr, dout ) ;     input  [11:2]  addr ;    // address bus     output [31:0]  dout ;    // 32-bit memory output  reg      [31:0]  im[1023:0] ; </pre>   |
| dm.v   | <pre> dm_4k( addr, din, we, clk, dout ) ;     input  [11:2]  addr ;    // address bus     input  [31:0]  din ;    // 32-bit input data     input          we ;    // memory write enable     input          clk ;    // clock     output [31:0]  dout ;    // 32-bit memory output  reg      [31:0]  dm[1023:0] ; </pre> |

## 四、 测试要求

12. 所有指令都应被测试充分。

13. 构造至少包括 40 条以上指令的测试程序，并测试通过。

a) MIPS-Lite3 定义的每条指令至少出现 1 次以上。

b) 必须有函数，并至少 1 次函数调用。

14. 函数相关指令(jal 和 jr)是较为复杂的指令，其正确性不仅涉及到自身的正确性，还与堆栈调整等操作相关。因此为了更充分的测试，你必须在测试程序中组织一个循环，并在循环中多次函数调用，以确保正确实现了这 2 条指令。

15. 详细说明你的测试程序原理及测试结果。【WORD】

- a) 应明确说明测试程序的测试期望，即应该得到怎样的运行结果。
- b) 每条汇编指令都应该有注释。

## 五、 问答【WORD】

16. C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。

- a) 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的 Operation 部分。

## 六、 其他要求

- 17. 打包文件：VerilogHDL 工程文件、code.txt、code.txt 所对应的汇编程序、项目报告。
- 18. 时间要求：各班实验指导教师指定。
- 19. 本实验要求文档中凡是出现了【WORD】字样，就意味着该条目需要在实验报告中清晰表达。
- 20. 实验报告请按照《计算机组成原理实验报告撰写规则.doc》要求排版。

## 七、 成绩及实验测试要求

- 21. 实验成绩包括但不限于如下内容：初始设计的正确性、增加新指令后的正确性、实验报告等。
- 22. 实验测试时，你必须已经完成了处理器设计及开发。
  - a) 允许实验报告可以未完成。
- 23. 实验测试时，你需要展示你的设计并证明其正确性。
  - a) 应简洁的描述你的验证思路，并尽可能予以直观展示。
- 24. 实验指导教师会临时增加 1~2 条指令，你需要在规定时间内完成对原有设计的修改，并通过实验指导教师提供的测试程序。
  - a) 考查时，教师将用专用 testbench 和 code.txt 检测代码执行情况。

## 八、开发与调试技巧

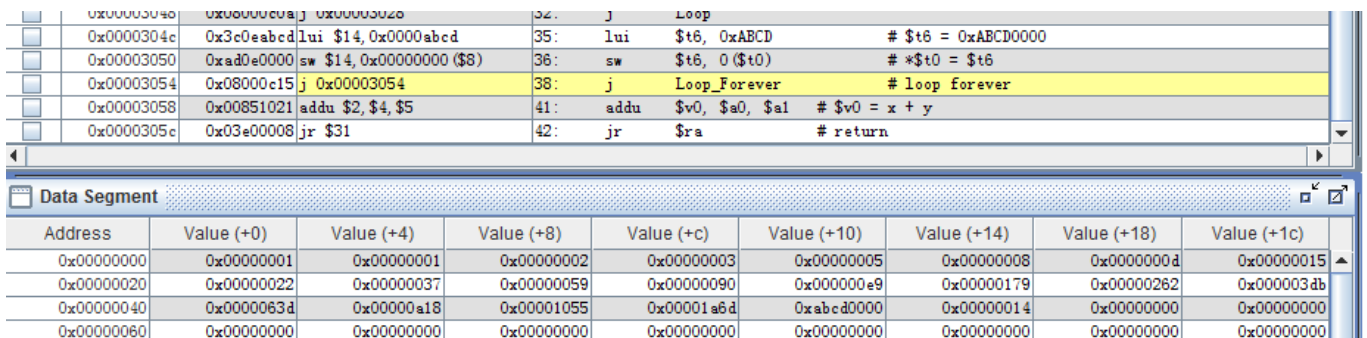
25. 对于每条指令，请认真阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》！

a) 如果测试时，你无法清楚的解释所要求的指令，测试成绩将减一档！

26. 建议先在 MARS 中编写测试程序并调试通过。注意 memory configuration 的具体设置。

a) 为了便于你初步测试你的处理器，我们提供了一个汇编程序 (proj4\_test\_fibonacci-2.asm)及其对应的指令码文件(code.txt)。

b) proj4\_test\_fibonacci-2.asm 首先初始化了 F0 和 F1，然后以函数调用的方式计算 F2 至 F19，并在 F19(其值为 0x1A6D)后写入 0xABCD\_0000，然后进入死循环，运行结果如图所示。



The screenshot shows the MARS MIPS simulator. The top window displays the assembly code for proj4\_test\_fibonacci-2.asm. The bottom window shows the Data Segment memory values.

| Address    | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+1c) |
|------------|------------|------------|------------|------------|-------------|-------------|-------------|-------------|
| 0x00000000 | 0x00000001 | 0x00000001 | 0x00000002 | 0x00000003 | 0x00000005  | 0x00000008  | 0x0000000d  | 0x00000015  |
| 0x00000020 | 0x00000022 | 0x00000037 | 0x00000059 | 0x00000090 | 0x000000e9  | 0x00000179  | 0x00000262  | 0x000003db  |
| 0x00000040 | 0x0000063d | 0x00000a18 | 0x00001055 | 0x00001a6d | 0xabc00000  | 0x00000014  | 0x00000000  | 0x00000000  |
| 0x00000060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000  | 0x00000000  | 0x00000000  | 0x00000000  |

Figure5 proj4\_test\_fibonacci-2.asm 运行结果

c) 你应该加载 code.txt 至指令存储器以测试你的处理器设计。假设你的处理器设计是正确的，那么数据存储器的前 22 个 word 的值应该如图所示。

d) 你需要参照 Figure4 设置 MARS，否则该程序将无法运行。

e) 注意：你不能将我们提供 proj4\_test\_fibonacci.asm 直接作为你写的测试程序提交。你需要编写一个不同的测试程序。

27. 利用\$readmemh 系统任务可以给存储器初始化数据。例如可以把 code.txt 文件中的数据加载至 my\_memory 模块。

```
reg [31:0] my_memory[1023:0] ;
```

```
initial
```

```
$readmemh( "code.txt", my_memory ) ;
```

28. 有时我们需要较为集中的在顶层 testbench 中观察甚至修改下层模块的变量，那么你可以通过使用层次路径名来非常方便的达到这一目的。例如：

```

module testbench ;
    Chil C1(...) ;

    $display(C1.Art) ;
endmodule

module Chil(...) ;
    reg Art;

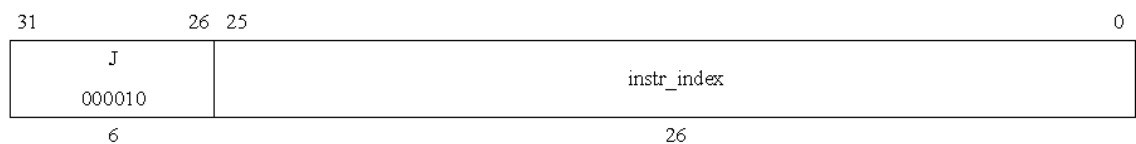
    ...
endmodule

```

## 九、 补充说明

由于 MIPS-Lite3 部分指令(主要是 JAL、JR)涉及非常复杂的运行模式，故你在阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》时可能存在困难。为此我们做了简化处理以便于你理解。简化主要是去除所有与 exception、delay slot 有关的描述。J、JAL、JR 所有简化后的描述陈列如下。

### 29. J 指令



Format: J target

MIPS32

#### Purpose:

To branch within the current 256 MB-aligned region

#### Description:

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr\_index* field shifted left 2 bits.

#### Operation:

I:  
 $I+1: PC \leftarrow PC_{GPRLEN-1..28} || instr\_index || 0^2$

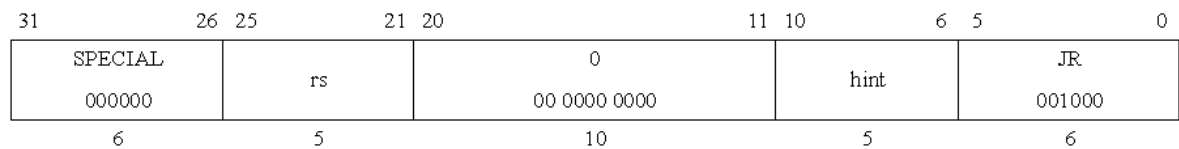
#### Exceptions:

None

#### Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

### 30. JR 指令



Format: JR rs

MIPS32

**Purpose:**

To execute a branch to an instruction address in a register

**Description:**  $PC \leftarrow GPR[rs]$

Jump to the effective target address in GPR *rs*.

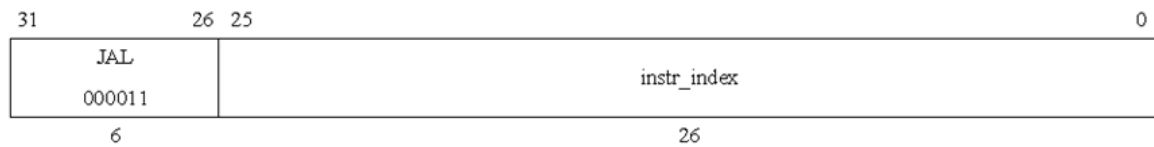
**Restrictions:**

The effective target address in GPR *rs* must be naturally-aligned.

**Operation:**

I: temp  $\leftarrow GPR[rs]$   
 I+1: PC  $\leftarrow temp$

### 31. JAL 指令



Format: JAL target

MIPS32

**Purpose:**

To execute a procedure call within the current 256 MB-aligned region

**Description:**

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr\_index* field shifted left 2 bits.

**Operation:**

I:  $GPR[31] \leftarrow PC + 4$   
 I+1: PC  $\leftarrow PC_{GPRLEN-1..28} || instr\_index || 0^2$

**Exceptions:**

None

**Programming Notes:**

Forming the branch target address by concatenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

### 32. LW 指令

a) Operation

Addr  $\leftarrow \text{sign\_extend}(\text{offset}) + GPR[\text{base}]$



$\text{GPR}[\text{rt}] \leftarrow \text{Memory}[\text{Addr}]$

### 33. SW 指令

#### a) Operation

$\text{Addr} \leftarrow \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}]$

$\text{Memory}[\text{Addr}] \leftarrow \text{GPR}[\text{rt}]$