单周期CPU

设计文档

2.0

贾博驿

2023年11月5日

设计实现指令

R型指令：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6 | 5 | 5 | 5 | 5 | 6 |
| opcode | rs | rt | rd | shamt | funct |

I型指令：

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 5 | 5 | 16 |
| opcode | rs | rt | imm16 |

J型指令：

|  |  |
| --- | --- |
| 6 | 26 |
| opcode | instr\_index |

以下除分支与跳转类指令，所有的RTL省略PC←PC+4。

**算数与逻辑运算类**

1. 算数类
2. R型

|  |  |  |
| --- | --- | --- |
| 名称 | funct | RTL |
| add | 100000 | GPR[rd] ← GPR[rs] + GPR[rt] |
| addu | 100001 | GPR[rd] ← GPR[rs] + GPR[rt] |
| sub | 100010 | GPR[rd] ← GPR[rs] - GPR[rt] |
| subu | 100011 | GPR[rd] ← GPR[rs] - GPR[rt] |

注：实现的add实际上为addu、sub实际上为subu。二者按照都按照无符号指令处理。

1. I型

|  |  |  |
| --- | --- | --- |
| 名称 | opcode | RTL |
| addiu | 001001 | GPR[rt]← GPR[rs] + sign\_extend(immediate) |

1. 逻辑类
2. R型

|  |  |  |
| --- | --- | --- |
| 名称 | funct | RTL |
| and | 100100 | GPR[rd]←GPR[rs] AND(bitwise) GPR[rt] |
| or | 100101 | GPR[rd]←GPR[rs] OR(bitwise) GPR[rt] |
| xor | 100110 | GPR[rd]←GPR[rs] XOR(bitwise) GPR[rt] |
| nor | 100111 | GPR[rd]←GPR[rs] NOR(bitwise) GPR[rt] |
| sll | 000000 | GPR[rd]←GPR[rt](31-shamt)..0 || 0shamt |
| srl | 000010 | GPR[rd]←0shamt || GPR[rt]31..shamt |
| sra | 000011 | GPR[rd]←(GPR[rt]31)shamt || GPR[rt]31..shamt |
| sllv | 000100 | s ← GPR[rs]4..0 GPR[rd]←GPR[rt](31-s)..0 || 0s |
| srlv | 000110 | s ← GPR[rs]4..0 GPR[rd]←0s || GPR[rt]31..s |
| srav | 000111 | s ← GPR[rs]4..0 GPR[rd]←(GPR[rt]31)s || GPR[rt]31..s |

1. I型

|  |  |  |
| --- | --- | --- |
| 名称 | opcode | RTL |
| andi | 001100 | GPR[rt] ← GPR[rs] AND(bitwise) zero\_extend(immediate) |
| ori | 001101 | GPR[rt] ← GPR[rs] OR(bitwise) zero\_extend(immediate) |
| xori | 001110 | GPR[rt] ← GPR[rs] XOR(bitwise) zero\_extend(immediate) |

**寄存器操作类**

1. R型

|  |  |  |
| --- | --- | --- |
| 名称 | funct | RTL |
| slt | 101010 | GPR[rd] ← 0GPRLEN-1 || GPR[rs] < GPR[rt] |
| sltu | 101011 | GPR[rd] ← 0GPRLEN-1 || (0 || GPR[rs]) < (0 || GPR[rt]) |

注：slt类指令成立时置1，**不成立时置0**。

1. I型

|  |  |  |
| --- | --- | --- |
| 名称 | opcode | RTL |
| lui | 001111 | GPR[rt] ← immediate || 016 |
| slti | 001010 | GPR[rt] ← 0GPRLEN-1|| GPR[rs] < sign\_extend(immediate) |
| sltiu | 001011 | GPR[rt] ← GPR[rt] ← 0GPRLEN-1 || (0 || GPR[rs]) < (0 || sign\_extend(immediate)) |

注：sltiu对立即数的处理是先**有符号扩展**再进行无符号比较。

**分支与跳转类**

1. 分支类（I型）

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | opcode | rt | RTL |
| (all) |  |  | if condition then  PC ← PC + 4 + sign\_extend(offset || 02)  endif |
| bltz | 000001 | 00000 | condition ← GPR[rs] < 032 |
| bgez | 000001 | 00001 | condition ← GPR[rs] ≥ 032 |
| beq | 000100 | rt | condition ← (GPR[rs] = GPR[rt]) |
| bne | 000101 | rt | condition ← (GPR[rs] ≠ GPR[rt]) |
| blez | 000110 | 00000 | condition ← GPR[rs] ≤ 032 |
| bgtz | 000111 | 00000 | condition ← GPR[rs] > 032 |

1. 跳转类
2. R型

|  |  |  |
| --- | --- | --- |
| 名称 | funct | RTL |
| jr | 001000 | PC ←GPR[rs] |
| jalr | 001001 | GPR[rd] ← PC + 4, PC← GPR[rs] |

1. J型

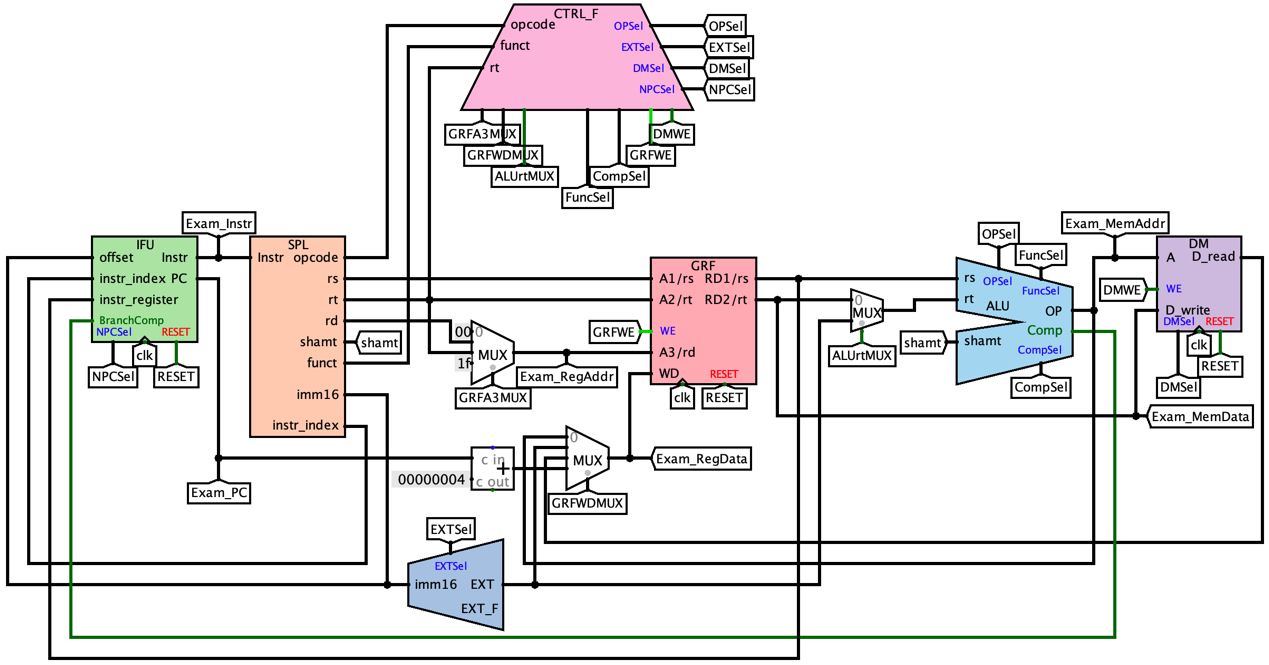
|  |  |  |
| --- | --- | --- |
| 名称 | opcode | RTL |
| j | 000010 | PC ← PC31..28 || instr\_index || 02 |
| jal | 000011 | GPR[31]← PC + 4, PC ← PC31..28 || instr\_index || 02 |

**内存操作类（I型）**

|  |  |  |
| --- | --- | --- |
| 名称 | opcode | RTL |
| (all) |  | addr←sign\_extend(offset) + GPR[base] |
| lb | 100000 | byte←addr1..0  GPR[rt]←sign\_extend(memory[addr]7+8\*byte..8\*byte) |
| lh | 100001 | half← addr1  GPR[rt]←sign\_extend(memory[addr]15+8\*half..8\*half) |
| lw | 100011 | GPR[rt]←memory[addr] |
| lbu | 100100 | byte←addr1..0  GPR[rt]←zero\_extend(memory[addr]7+8\*byte..8\*byte) |
| lhu | 100101 | half← addr1  GPR[rt]←zero\_extend(memory[addr]15+8\*half..8\*half) |
| sb | 101000 | memory[sign\_extend(offset) + GPR[base]]7..0 ← GPR[rt]7..0 |
| sh | 101001 | memory[sign\_extend(offset) + GPR[base]]15..0 ← GPR[rt]15..0 |
| sw | 101011 | memory[sign\_extend(offset) + GPR[base]]31..0 ← GPR[rt] |

注：针对halfword操作需要addr为2的非负整数倍，针对word操作需要addr为4的非负整数倍。针对byte和halfword操作时只修改对应byte和halfword的值，同一个word的其他部分不变。

模块设计



**IFU**

1. 描述

取指令模块。内有PC子模块、NPC子模块和ROM。接收分支和跳转指令的控制信号和数据并实现跳转。输出指令和PC寄存器取值。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| RESET | I | 异步复位信号 |
| clk | I | 时钟信号 |
| NPCSel[1:0] | I | NPC子模块控制信号 |
| BranchComp | I | NPC子模块分支指令控制信号 |
| offset[15:0] | I | 分支指令跳转 |
| instr\_index[25:0] | I | j、jal指令跳转 |
| instr\_register[31:0] | I | jr、jalr指令跳转 |
| Instr[31:0] | O | 当前指令 |
| PC[31:0] | O | 当前PC寄存器取值 |

1. 控制信号

BranchComp：接ALU的Comp，表示分支指令的条件是否满足。

NPCSel：接CTRL，使用Priority Encoder，00接VCC。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 00 | PC←PC+4 |
| 分支指令 | 01 | 使用offset，结合BranchComp判断 |
| j、jal | 10 | 使用instr\_index |
| jr、jalr | 11 | 使用instr\_register |

**IFU\_PC**

1. 描述

程序计数器子模块。Logisim中使用内置D触发器组装。为实现复位直接变为0x00003000，第13、12号位的复位信号接入异步置位接口，其余信号接入异步复位接口。Verilog中直接使其在复位信号有效时的值为32'h00003000。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| RESET | I | 异步复位信号 |
| clk | I | 时钟信号 |
| D[31:0] | O | 下一指令地址 |
| Q[31:0] | O | 当前指令地址 |

**IFU\_NPC**

1. 描述

下一条指令地址计算子模块。输入IFU接收的分支和跳转指令的控制信号和数据，使用MUX，计算出下一条指令的地址。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| PC[31:0] | 1 | 当前指令地址 |
| NPCSel[1:0] | I | 同IFU |
| BranchComp | I | 同IFU |
| offset[15:0] | I | 同IFU |
| instr\_index[25:0] | I | 同IFU |
| instr\_register[31:0] | I | 同IFU |
| NPC[31:0] | O | 下一指令地址 |

**SPL**

1. 描述

指令分线器模块。按照三种指令的格式将一条指令分为不同的部分，供其他模块使用。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| Instr[31:0] | 1 | 当前指令 |
| opcode[5:0] | O | 指令的操作数 |
| rs[4:0] | O | R、I型指令的rs |
| rt[4:0] | O | R、I型指令的rt |
| rd[4:0] | O | R型指令的rd |
| shamt[4:0] | O | R型指令的shamt |
| funct[5:0] | O | R型指令的funct |
| imm16[15:0] | O | I 型指令的立即数 |
| instr\_index[25:0] | O | J型指令的跳转地址 |

**GRF**

1. 描述

寄存器堆模块，共有31个寄存器（0号寄存器直接接地）。通过A1、A2输入地址，可以取出指定寄存器存储的数据。当WE使能时，通过A3输入地址，WD输入数据，可以修改指定寄存器存储的数据。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| RESET | I | 异步复位信号 |
| clk | I | 时钟信号 |
| WE | I | 写入使能信号 |
| A1[4:0] | I | 读取的第一个寄存器的编号 |
| A2[4:0] | I | 读取的第二个寄存器的编号 |
| A3[4:0] | I | 写入的寄存器的编号 |
| WE[31:0] | I | 写入寄存器的数据 |
| RD1[31:0] | O | 读取的第一个寄存器的值 |
| RD2[31:0] | O | 读取的第二个寄存器的值 |

1. 控制信号

WE：接CTRL、使用或门。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 0 | 不涉及寄存器写入的指令 |
| R型：除jr | 1 |  |
| I型：算数与逻辑运算类、寄存器操作类 | 1 |  |
| I型：内存操作类读取类 | 1 |  |
| jal | 1 |  |

GRFA3MUX：接CTRL，使用Priority Encoder，00接VCC。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 00 | 固定0(0x00)，保护 |
| R型：全体（忽略不使能的jr） | 01 | 接入rd |
| I型：全体（忽略不使能的类型） | 10 | 接入rt |
| jal | 11 | 固定31(0x1f) |

GRFWDMUX：接CTRL，使用Priority Encoder，00接VCC。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 00 | 接入ALU |
| lui | 01 | 接入EXT |
| I型：内存操作读取类 | 10 | 接入DM |
| jalr、jal | 11 | 接入PC + 4 |

**ALU**

1. 描述

算数逻辑模块，完成算数运算（加、减）、逻辑运算（与、或、异或、或非）、移位运算（左移、逻辑右移、算数右移）、比较运算（两数相等、不等，有符号小于，无符号小于，一数有符号小于、小于等于、大于、大于等于零）。提供的计算结果用于保存到寄存器、判断分支等。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| OPSel[1:0] | I | OP输出控制信号 |
| FuncSel[2:0] | I | 算数、逻辑、移位部分控制信号 |
| CompSel[2:0] | I | 比较部分控制信号 |
| rs[31:0] | I | 第一个操作数 |
| rt[31:0] | I | 第二个操作数 |
| shamt[4:0] | I | 移位立即数 |
| OP[31:0] | O | 计算结果 |
| Comp | O | 比较结果 |

1. 控制信号

OPSel：接CTRL，使用Priority Encoder，00接VCC。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| add、addu、sub、subu、addiu | 00 | 算数运算（加、减） |
| I型：内存操作类 | 00 | 算数运算（加、减） |
| and、or、xor、nor、andi、ori、xori | 01 | 逻辑运算（与、或、异或、或非） |
| sll、srl、sra、sllv、srlv、srav | 10 | 移位运算（左移、逻辑右移、算数右移） |
| slt、sltu、slti、sltiu | 11 | 比较运算（无符号拓展Comp为32位） |

FuncSel：接CTRL，使用MUX，根据OPSel选择。

OPSel == 00时，使用或门。（只用设置001）

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| add、addu、addiu | 000 | 加法 |
| I型：内存操作类读取类 | 000 | 加法 |
| sub、subu | 001 | 减法 |

OPSel == 01时，使用MUX，MUX控制信号为是否是IJ型。R型时接funct[2:0]、IJ型时接opcode[2:0]。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| and(100100)、andi(001100) | 100 | 与 |
| or(100101)、ori(001101) | 101 | 或 |
| xor(100110)、xori(001110) | 110 | 异或 |
| nor(100111) | 111 | 或非 |

OPSel == 10时，接funct[2:0]。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| sll(000000) | 000 | 逻辑左移立即数 |
| srl(000010) | 010 | 逻辑右移立即数 |
| sra(000011) | 011 | 算数右移立即数 |
| sllv(000100) | 100 | 逻辑左移寄存器 |
| srlv(000110) | 110 | 逻辑右移寄存器 |
| srav(000111) | 111 | 算数右移寄存器 |

OPSel == 11时， FuncSel无效，置为000。

Comp：接CTRL、使用Priority Encoder，00接VCC。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| bltz(00000) | 000 | 有符号小于0 |
| bgez(00001) | 001 | 有符号大于等于0 |
| slt(101010)、slti(001010) | 010 | 有符号两数小于 |
| sltu(101011)、sltiu(001011) | 011 | 无符号两数小于 |
| beq(000100) | 100 | 两数相等 |
| bne(000101) | 101 | 两数不等 |
| blez(000110) | 110 | 有符号小于等于0 |
| bgtz(000111) | 111 | 有符号大于0 |

ALUrtMUX：接CTRL、使用或门。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 0 | 接入GRF |
| I型：除分支与跳转类 | 1 | 接入EXT |

**EXT**

1. 描述

位扩展模块，将16位立即数无符号扩展、有符号扩展、右补16位0为32位。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| EXTSel[1:0] | I | 扩展方式控制信号 |
| imm16[15:0] | I | 16位立即数 |
| EXT[31:0] | O | 32位扩展结果 |

1. 控制信号

EXTSel：接CTRL，使用Priority Encoder，00接VCC。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 00 | 有符号拓展 |
| andi、ori、xori | 01 | 无符号拓展 |
| lui | 10 | 右补16位0 |

**DM**

1. 描述

内存模块。包括RAM和位处理两个部分。位处理部分用于处理针对byte和halfword的读取和写入。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| RESET | I | 异步复位信号 |
| clk | I | 时钟信号 |
| WE | I | 写入使能信号 |
| DMSel[2:0] | I | 读取存储模式控制信号 |
| A[31:0] | I | 读取或写入的地址 |
| D\_write[31:0] | I | 写入的数据 |
| D\_read[31:0] | O | 读取的数据 |

1. 控制信号

WE：接CTRL，使用或门。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| 其余指令 | 0 | 不涉及写入寄存器的指令 |
| sb、sh、sw | 1 |  |

DMSel：接CTRL，接opcode[2:0]。

|  |  |  |
| --- | --- | --- |
| 指令 | 取值 | 描述 |
| lb(100000)、sb(101000) | 000 | 针对byte操作，**读取时**有符号拓展 |
| lh(100001)、sh(101001) | 001 | 针对halfword操作，**读取时**有符号拓展 |
| lw(100011) | 011 | 针对word操作 |
| lbu(100100) | 100 | 针对byte操作，读取时无符号拓展 |
| lhu(100101) | 101 | 针对halfword操作，读取时无符号拓展 |

**CTRL**

**（CTRL\_AND、CTRL\_OR）**

1. 描述

控制模块。通过opcode、funct和rt判断指令的类型，输出各个模块的控制信号。 Logisim中分为两个子模块，AND子模块通过与逻辑，判断指令类型。OR子模块通过或门、Priority Encoder、MUX等元件，将判断的指令类型转换为控制信号。Verilog中将其合并在一个模块中。

1. 接口

|  |  |  |
| --- | --- | --- |
| 名称 | 方向 | 描述 |
| opcode[5:0] | I | 指令的操作数 |
| funct[5:0] | I | R型指令funct |
| rt[4:0] | I | 分支指令rt |
| OpSel[1:0] | O | 接入ALU |
| EXTSel[1:0] | O | 接入EXT |
| DMSel[2:0] | O | 接入DM |
| NPCSel[1:0] | O | 接入IFU |
| FuncSel[2:0] | O | 接入ALU |
| CompSel[2:0] | O | 接入ALU |
| GRFWE | O | 接入GRF |
| DMWE | O | 接入DM |
| GRFA3MUX[1:0] | O | 接入GRF的GRFA3MUX |
| GRFWDMUX[1:0] | O | 接入GRF的GRFWDMUX |
| ALUrtMUX | O | 接入ALU的ALUrtMUX |

**P3思考题**

1. 单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能？

发挥状态存储功能的是GRF、DM，其中包含的大量的寄存器表示当前CPU的状态。

发挥状态转移功能的是IFU、ALU、EXT、(Controller)CTRL。其中IFU中包含的ROM可以等效为一种输入信号。其余的部分均为组合逻辑电路，作为次态逻辑和输出逻辑。

1. IM 使用 ROM，DM 使用 RAM，GRF 使用 Register，这种做法合理吗？

目前的设计情况下是合理的。目前没有加入其他的I/O设备，无法从其他地方读取存储的程序，需要通过非易变的ROM存储程序。DM要求大量的存储空间，读写并不频繁，使用RAM是合理的，也符合真实情况。GRF中的寄存器数量要求不多，读写频繁，使用读写速度快的Register是合理的。

如果增加了I/O设备，实现读取的存储程序，应当将IM改为使用RAM。

1. 你是否在实际实现时设计了其他的模块？

设计了专门的Splitter，名称为SPL。

设计了专门的Register，用于IFU中的PC，名称为IFU\_PC。

将IFU分为IFU\_PC和IFU\_NPC 两部分，将CTRL分为CTRL\_AND和CTRL\_OR两部分。

以上模块的介绍和实现思路见文档正文部分。

1. 事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？

nop指令如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6 | 5 | 5 | 5 | 5 | 6 |
| opcode | rs | rt | rd | shamt | funct |
| 000000 | 00000 | 00000 | 00000 | 00000 | 00000 |

opcode为000000且funct为000000的指令为sll，按照sll指令的机制可以解释为：

sll $0, $0, 0

也就是将$0逻辑左移0位存入$0。$0本身固定为0x00000000，GRF不会发生改变，这一条指令也不涉及内存的读写，DM不会发生改变。也即CPU的状态不会发生改变，也即空指令的效果。而以上过程完全是借由sll指令的机制实现的，故不需要将其添加到真值表。

1. 阅读 Pre 的“MIPS 指令集及汇编语言”一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处？

对于ori指令，没有覆盖到存储到$0的情况，立即数的位数不够大。

对于lui指令，没有覆盖到存储到$0的情况，没有覆盖立即数为0情况。

对于add指令，没有覆盖到存储到$0的情况，没有覆盖使用 $0的情况，没有覆盖到32位整型值的正极限和负极限，没有覆盖到运算溢出的情况。

对于sw指令，没有覆盖到存储$0的情况，没有覆盖到offset为负数的情况，没有覆盖base为正数、负数的情况，base为固定值，未覆盖base为可变值的情况。

对于lw指令，没有覆盖到存储到$0的情况，没有覆盖到offset为负数的情况，没有覆盖base为正数、负数的情况，base为固定值，未覆盖base为可变值的情况。

对于beq指令，没有覆盖使用$0的情况，没有覆盖到寄存器的值为负数、0、较大的正数的情况。没有覆盖跳转的地址为该条指令指令之前，该条指令本身的情况。

针对于这一次的要求，未覆盖sub指令、nop指令。

**P4思考题**

1. 阅读下面给出的 DM 的输入示例中（示例 DM 容量为 4KB，即 32bit × 1024字），根据你的理解回答，这个 addr 信号又是从哪里来的？地址信号 addr 位数为什么是 [11:2] 而不是 [9:0] ？



这个addr信号从ALU来的。因为DM所涉及的lw、sw等指令的共同操作是

addr←sign\_extend(offset) + GPR[base]

这个操作由EXT先进行符号拓展，再由ALU进行加法运算得到。所以addr信号应接入ALU。因为在仅实现lw、sw两条指令并且仅执行合法指令的前提下，addr的后两位均为00，而表示DM的所有地址即0-1023只需要10位二进制数即可，所以位数只需要[11:2]即可。

1. 思考上述两种控制器设计的译码方式，给出代码示例，并尝试对比各方式的优劣。

第一种译码方式：指令对应的控制信号如何取值。

always @(\*) begin

if (opcode = 6'b000000) begin

case (funct)

`R\_add :

OPSel <= 2'b00

// others

default:

// others

endcase

end

else begin

case (opcode)

`I\_ori :

OPSel <= 2'b01

// others

default:

// others

endcase

end

end

第二种译码方式：控制信号每种取值所对应的指令。本设计文档之前所用的方式就是这种。示例代码如下：

assign NPCSel = (I\_beq) ? 2'b01 :

(J\_jal) ? 2'b10 :

(R\_jr) ? 2'b11 :

2'b00;

第一种方式设计优点：增加新的指令方便，只需要设计好各个模块控制信号的取值即可直接通过添加一个case添加一条新的指令。缺点：对于每个指令都要指定所有的控制信号端口的值，若指令数量增加，代码将十分冗长。

第二种方式设计优点：翻译形成控制模块方便，可以利用opcode和funct的特征进行批量地编码，指令数量多的时候代码也不会非常长。缺点：增加新的指令复杂，有可能要考虑到指令之间的在判断逻辑上的相互影响。

1. 在相应的部件中，复位信号的设计都是同步复位，这与 P3 中的设计要求不同。请对比同步复位与异步复位这两种方式的 reset 信号与 clk 信号优先级的关系。

同步复位中，clk信号的优先级高于reset信号。没有时钟信号的时候，复位信号不起作用。

异步复位中，reset信号的优先级高于clk信号。存在复位信号时，时钟信号不会引起状态变化。

1. 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。 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。

addi的RTL是

temp ← (GPR[rs]31||GPR[rs]31..0) + sign\_extend(immediate)

if temp32 ≠ temp31 then

SignalException(IntegerOverflow)

else

GPR[rt] ← temp

endif

addiu的RTL是

temp ← GPR[rs] + sign\_extend(immediate)

GPR[rt]← temp

两个指令的操作相比较，addi相比addiu只是多了对于溢出的判断，其余的操作均相同，所以在忽略溢出的前提下，addi和addiu是等价的。

P3测试工作

由于Pre中提供的测试样例、weak测试点强度均不够，故使用Python编写自动指令生成程序，辅助人工编写相关程序对CPU进行加强测试。针对要求实现的指令，add、sub、ori、lui、lw、sw、beq、nop。进行以下三项测试。同时编写MARS自动对拍程序进行自动化测试。

1. ori、lui、sw指令测试

使用Python完成。

随机生成32位整型范围内的数字，使用lui和ori将其赋值给任意寄存器（包括$0，不包括$1），随机生成0x00000000至0x00002fff范围内的地址，将其赋值给$1，随机生成-20至20范围内的偏移值，将寄存器的值保存到内存对应的地址。运行结束后，对比Logisim和MARS的内存情况。

1. lw、add（addu）、sub（subu）指令测试

使用Python完成。

先随机生成128个整型范围内的数字，按照1的方法保存至内存。随机从保存这些数字的内存中读取并赋值给任意寄存器（包括$0，不包括$1），再保存到内存中。最后从保存这些数字的内存中随机读取两个数字并赋值给任意两个不同的寄存器（包括$0，不包括$1），随机对寄存器的值进行加减赋值给另一个不同的寄存器（包括$0，不包括$1），再保存到内存中。运行结束后，对比Logisim和MARS的内存情况。

1. beq、nop指令测试

使用人工编写汇编程序完成。

先给$1到$8赋值小的正数，小的负数、大的正数、大的负数。每一次赋值之后使用nop指令。然后通过各种组合对beq指令进行测试，包括是否跳转，目标在此跳转指令之前、之后、本条共六种组合。每一条跳转指令后面跟随保存相关的寄存器的值至内存中。运行结束后，对比Logisim和MARS的内存情况。

P4测试工作

由于weak测试点强度不够。故改进P3的测试，使用Python自动生成，辅助人工编写相关程序对CPU进行加强测试。针对要求实现的指令，add、sub、ori、lui、lw、sw、beq、jal、jr、nop。进行以下四项测试。同时利用助教提供的修改版MARS进行自动对拍。

1. 模块单元测试

使用ISE编写testbench完成。

针对IFU、SPL、GRF、EXT等组件编写单元testbench，覆盖测试其所有设计功能是否能够正常完成。

1. ori、lui、sw指令测试

使用Python完成。

随机生成32位整型范围内的数字，使用lui和ori将其赋值给任意寄存器（包括$0，不包括$1），随机生成0x00000014至0x00002fff范围内的地址，将其赋值给$1，随机生成-20至20范围内的偏移值，将寄存器的值保存到内存对应的地址。对比Verilog和MARS的输出情况。

1. lw、add（addu）、sub（subu）指令测试

使用Python完成。

先随机生成256个整型范围内的数字，按照1的方法保存至内存。随机从保存这些数字的内存中读取并赋值给任意寄存器（包括$0，不包括$1），再保存到内存中。最后从保存这些数字的内存中随机读取两个数字并赋值给任意两个不同的寄存器（包括$0，不包括$1），随机对寄存器的值进行加减赋值给另一个不同的寄存器（包括$0，不包括$1），再保存到内存中。对比Verilog和MARS的输出情况。

1. beq、jal、jr、nop指令测试

使用人工编写汇编程序完成。

先给$1到$8赋值小的正数，小的负数、大的正数、大的负数。每一次赋值之后使用nop指令。对beq指令进行测试，包括是否跳转，目标在此跳转指令之前、之后、本条共六种组合。对jal和jr指令进行测试，包括目标在此跳转指令之前、之后两种组合，使用jal跳转之后，再使用jr进行跳转。每一条跳转指令后面跟随保存相关的寄存器的值至内存中。对比Verilog和MARS的输出情况。