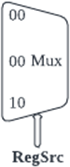
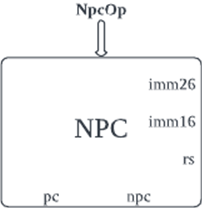
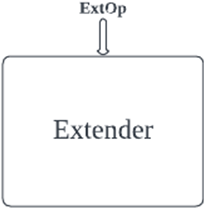
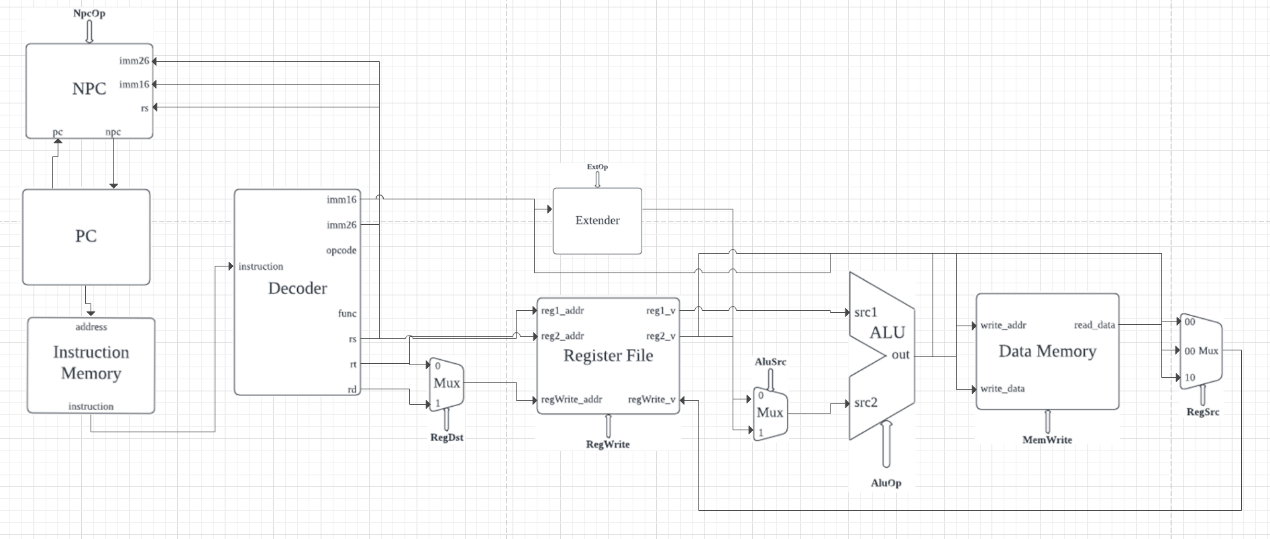
**单周期**

首先根据小组讨论，我们利用图示软件确定了CPU各个模块的输入输出以及控制信号，然后画出了初步的CPU数据通路图。

模块图：



数据通路图



接着，列出需要列出二进制指令集和控制信号表，新建definition文件定义操作码、控制信号和二进制编码的对应关系，指令操作码、控制信号和二进制编码的对应关系如下所示：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | reg\_dst | reg\_src | alu\_src | alu\_op | en\_men\_write | reg\_src | extend | npc\_op |
| lui | 01 | 011 | - | - | 0 | imm | shift | next |
| addiu | 01 | 001 | 1 | add | 0 | alu | unsigned | next |
| addi | 01 | 001 | 1 | add | 0 | alu | signed | next |
| add | 11 | 001 | 0 | add | 0 | alu | - | next |
| sub | 11 | 001 | 0 | sub | 0 | alu | - | next |
| lw | 0 | 010 | 1 | add | 0 | mem | unsigned | next |
| sw | - | 010 | 1 | add | 1 | - | unsigned | next |
| beq | - | - | 0 | sub | 0 | - | - | offset |
| j | - | - | - | - | 0 | - | - | immediate |

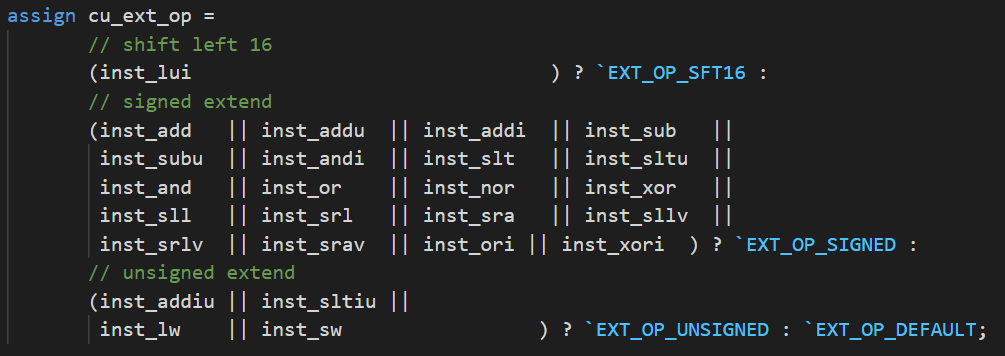
控制信号表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 信号 | 作用模块 | 含义 | 编码 | 信号名 |
| en\_reg | register\_file | 写寄存器文件 | 1 |  |
|  |  | / | 0 |  |
| extend | extend | 不扩展 | 00 |  |
|  |  | LUI左移16位 | 01 |  |
|  |  | 符号扩展16->32 | 10 |  |
|  |  | 无符号扩展16->32 | 11 |  |
| npc\_op | npc | / | 000 |  |
|  |  | PC+4 | 001 |  |
|  |  | 26位立即数扩展 | 010 |  |
|  |  | PC+offset+4 | 011 |  |
|  |  | rs | 100 |  |
| alu\_src | alu\_src\_mux | register源 | 0 |  |
|  |  | immediate源 | 1 |  |
| alu\_op | alu | / | 0000 |  |
|  |  | ADD | 0001 |  |
|  |  | SUB | 0010 |  |
|  |  | SLT | 0011 |  |
|  |  | AND | 0100 |  |
|  |  | OR | 0101 |  |
|  |  | XOR | 0110 |  |
|  |  | NOR | 0111 |  |
|  |  | SLL | 1000 |  |
|  |  | SRL | 1001 |  |
|  |  | SRA | 1010 |  |
|  |  | SLLV | 1011 |  |
|  |  | SRLV | 1100 |  |
|  |  | SRAV | 1101 |  |
| reg\_src | reg\_src\_mux | / | 000 |  |
|  |  | 来自ALU | 001 |  |
|  |  | 来自data\_mam | 010 |  |
|  |  | 来自extend | 011 |  |
|  |  | 来自ALU | 100 |  |
| reg\_dst | reg\_dst\_mux | / | 00 |  |
|  |  | 写到rt | 01 |  |
|  |  | 写到rs | 10 |  |
|  |  | 写到寄存器文件中的某个 | 11 |  |

控制信号编码表

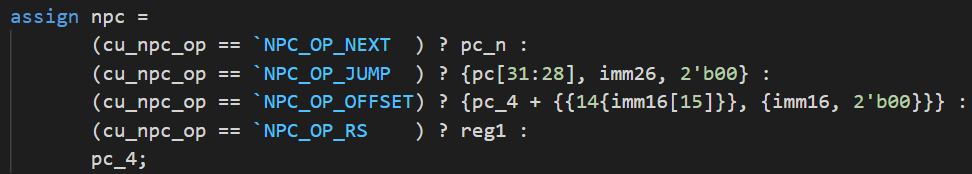
## 控制单元（CU）

控制模块CU负责根据译码结果产生控制信号，按照之前列出的控制信号表，确定控制信号，根据之前在definetion文件中定义的宏为不同的模块生成控制信号。以扩展模块extend的控制信号extop为例，根据表格lui指令应该使用特殊的左移16扩展信号，对于运算指令中含有字符‘u’的以及lw、sw指令应该进行无符号扩展，其余算数指令应该进行符号扩展，由此便可以写出extop控制信号的扩展表：



## NPC模块

|  |  |  |
| --- | --- | --- |
| input | pc | 当前程序计数器值 |
| input | imm16 | 16位立即数译码 |
| input | imm26 | 26位立即数译码 |
| input | rs | rs寄存器值 |
| input | control\_npc | npc模块控制信号 |
| output | npc | 下一条指令地址 |

NPC模块负责根据控制信号control\_npc判定下一条指令地址npc。下一条指令的地址可能有以下几种情况：①当前指令地址+4（pc+4） ②无条件跳转地址（imm26）③条件跳转（imm16+4或rs寄存器），因此输入要同时传入imm16，imm26，rs的值。在npc模块中还需要对不同长度的立即数做扩展，直接跳转的扩展方式为pc[31:28], imm26, 2'b00，相对寻址的扩张方式为符号扩展pc\_n + {{14{imm16[15]}}, {imm16, 2'b00}},关键代码如下所示：

## PC模块

|  |  |  |
| --- | --- | --- |
| input | wire clk | 时钟信号 |
| input | wire rst | 复位信号 |
| input [31:0] | wire npc | 下一条指令地址 |
| output | reg pc | 程序计数器值 |

PC模块接收NPC模块传入的下一条指令地址npc，储存并输出程序计数器的值。clk与rst分别代表时钟信号和复位信号，npc是由NPC模块传入的下一条指令地址。PC模块在时钟上升沿将npc的值更新到pc寄存器并输出，在rst信号上升沿刷新PC的值，代码中采用了异步刷新的方式。

## 指令寄存器 （Instruction Memory）

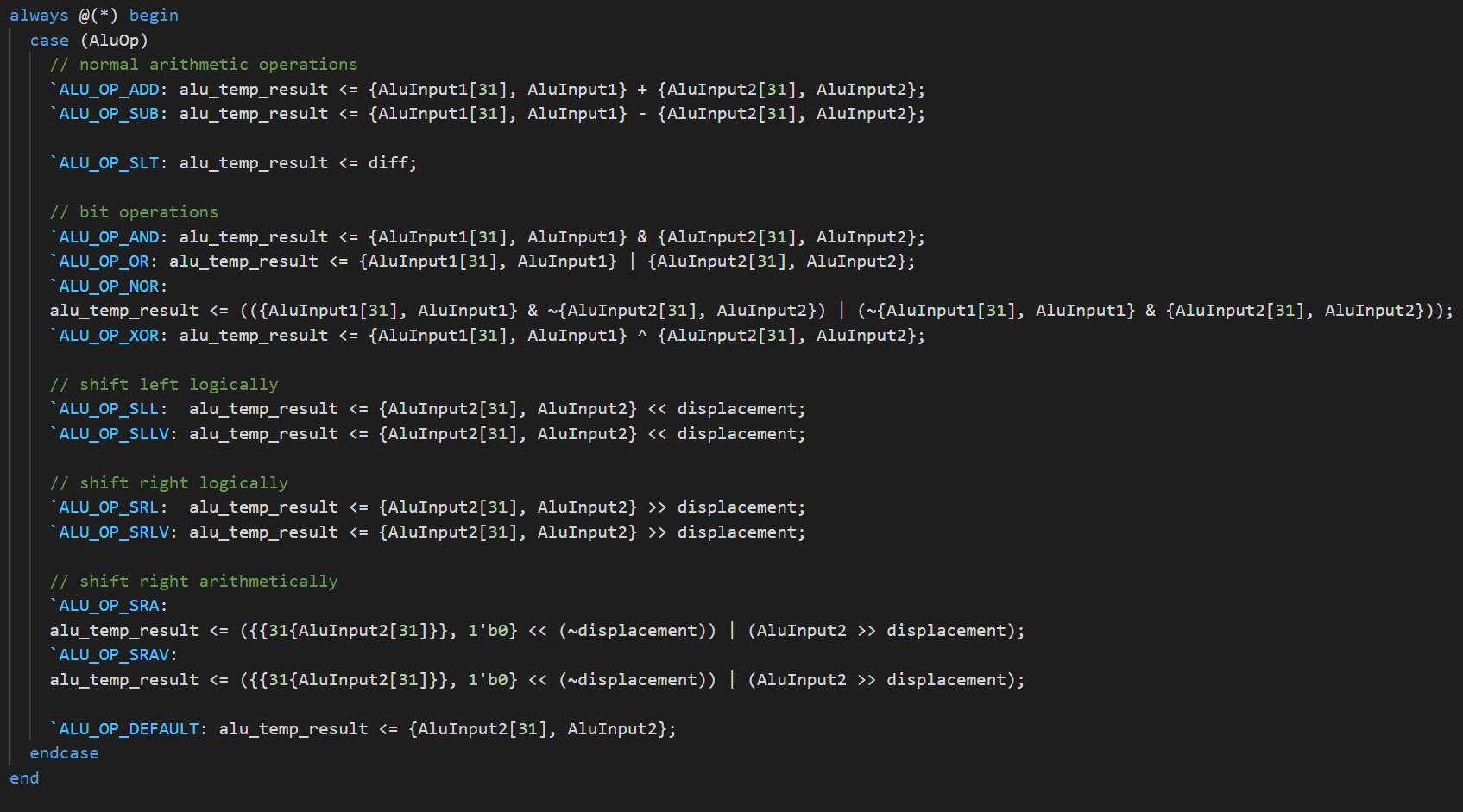
|  |  |  |
| --- | --- | --- |
| input | instruction\_addr | 指令地址 |
| output | [31:0] instruction | 指令内容 |

ins\_mem模块是存储指令的模块，根据传入指令地址输出32位指令信号，传入后续的译码阶段。此外，为了储存指令，建立instruction.txt以16进制的方式储存，之后在testbench中便可以通过$readmem函数进行读取。

**算数运算单元（ALU）**

|  |  |  |
| --- | --- | --- |
| input | input1 | Alu输入1 |
| input | input2 | Alu输入2 |
| input | AluOp | Alu控制信号 |
| output | AluOut | Alu运算结果 |
| output | overflow | 是否发生溢出 |

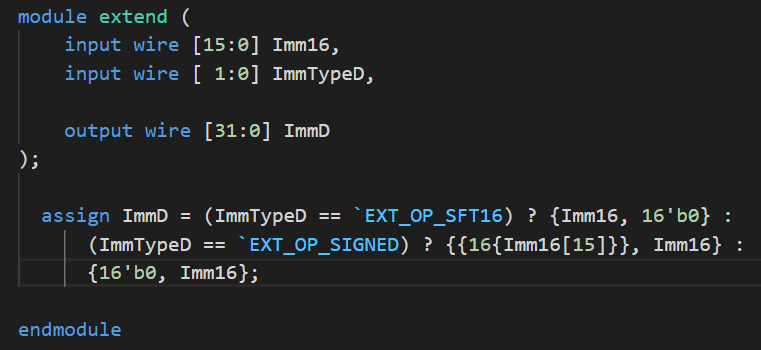
Alu依据控制信号AluOp对两个输入input1和input2做算数运算操作，根据之前准备实现的算数运算指令类型以及算数运算的特性，分情况写出对应的输出，具体代码如下所示



**扩展（Extend）**

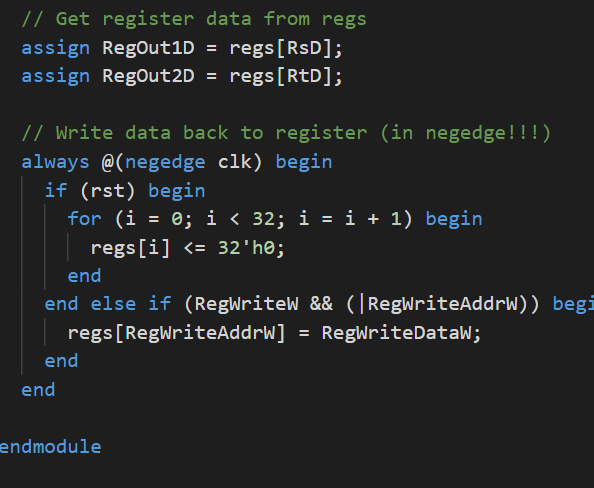
|  |  |  |
| --- | --- | --- |
| input | imm16 | 16位立即数输入 |
| input | ExtOp | 扩展控制信号 |
| output | out | 扩展结果 |

译码得到的16位立即数需要进行扩展，依据控制信号ExtOp将其扩展为32位，分情况讨论16位立即数的扩展操作。左移16的扩展情况下，将16位立即数放到高位，低16位置0；无符号和符号扩展分别将高16位置为0或符号位，具体代码如下。



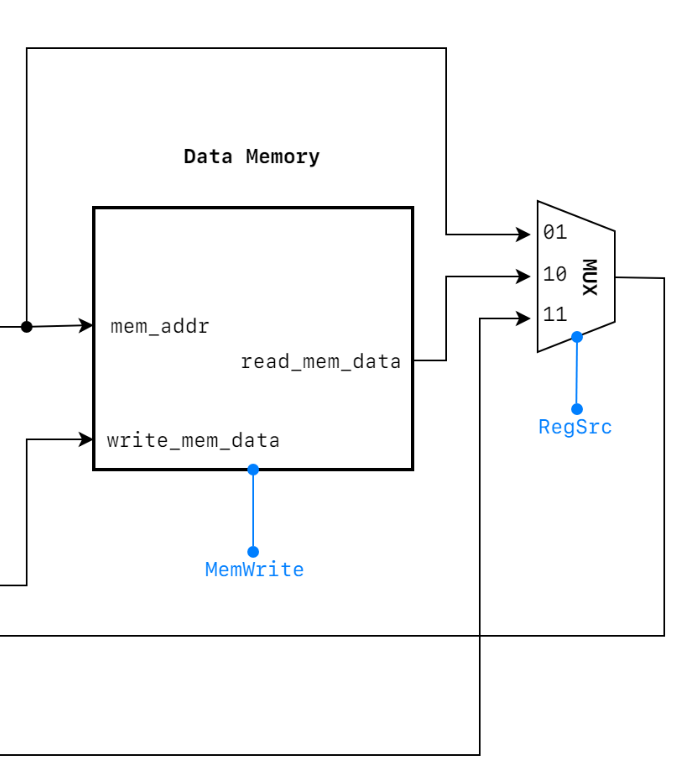
**寄存器文件（Register File）**

寄存器文件的默认行为是在时钟下降沿（避免冲突）将根据输入的两个寄存器地址输出对应寄存器的值。此外寄存器写文件的控制信号为RegWrite，用来判断是否允许写寄存器，其从译码结果获取寄存器地址，从写回阶段获取写入值，并依据RegWrite判断是否改写寄存器文件，关键代码如下。

****

**数据存储器（Data Memory）**

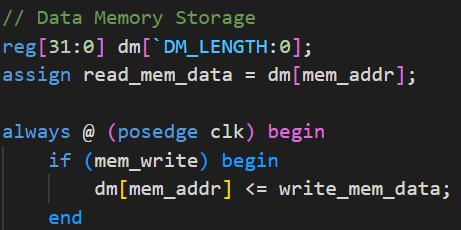
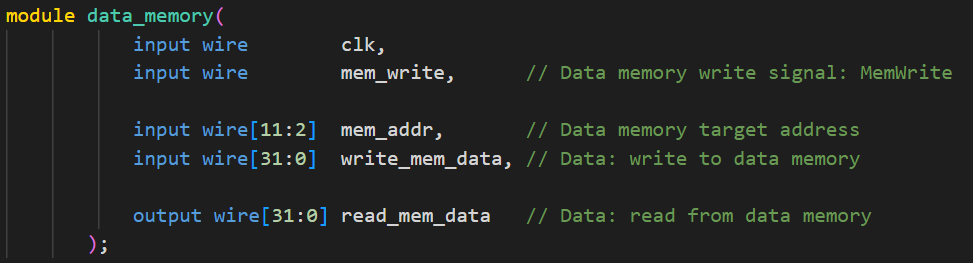
存储器访问阶段，如果指令设计读取、存储内存，则需要对存储器中相应地址进行读取或者写入。结果写回阶段，则将得到的数据（访问存储器或者修改其它寄存器的值获得）写回相应的寄存器。



访存和写回阶段输入输出接口设计如下：

|  |  |
| --- | --- |
| 接口名称 | 作用 |
| mem\_write | 数据存储器的写使能信号 |
| mem\_addr | ALU输出的数据地址 |
| write\_mem\_data | 地址内应存的数据 |
| read\_mem\_data | 从地址读出的数据 |
| RegSrc | 选择将 ALU 计算结果、数据存储器输出或 Extend 模块输出写入寄存器 |

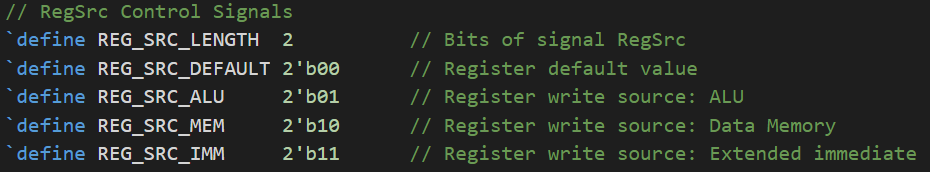
数据存储器相关接口及具体实现如下：



对于RegSrc信号，需要实现选择写入寄存器的源的功能，设计如下：

| **RegSrc[1:0]** | **功能** | **指令** |
| --- | --- | --- |
| 00 | Default | 缺省值 |
| 01 | 来自 ALU | ADDIU、ADD、SUB |
| 10 | 来自 Data Memory | LW |
| 11 | 来自 Extend 模块输出 | LUI |

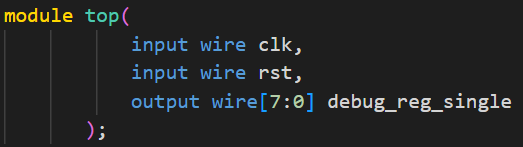
RegSrc控制信号的定义在instruction\_head.v中声明，具体如下图所示：



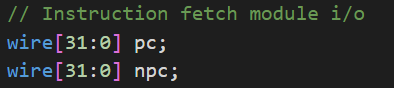
**顶层模块（top）**

在完成全体模块的编写之后，需要建立一个顶层模块将全部模块进行连接，实现模块的整体调用。顶层模块（top.v）是整个 CPU 的控制模块，通过连接各个子模块来达到运行 CPU 的目的，我们将译码也放在了这一模块中进行。单周期 CPU 的设计包括指令存储器：Instruction Memory、Data Memory、Register File、PC、NPC、CU、ALU、Extend Module、MUX这几个模块，顶层模块实例化各子模块使其接口互相连接。

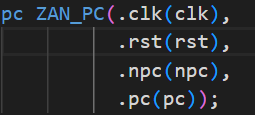
定义一个顶层模块 top.v，并设置输入信号：



同时，再声明顶端模块的 两个内部端口：



之后，比如我们需要调用 PC 模块，那么就可直接：



其中前面的pc跟定义PC模块的pc.v保持一致，后面的ZAN\_PC为当前文件调用模块名。在内部声明模块I/O端口时，通过 .调用模块端口(顶端模块端口)的语法格式进行调用。

顶端模块的功能就是将其余模块利用wire导线进行连接，因此在顶端模块内部，我们会定义用于连接各个模块输入输出的内部端口。这样就能让全部模块连接起来，成为完整的CPU电路。