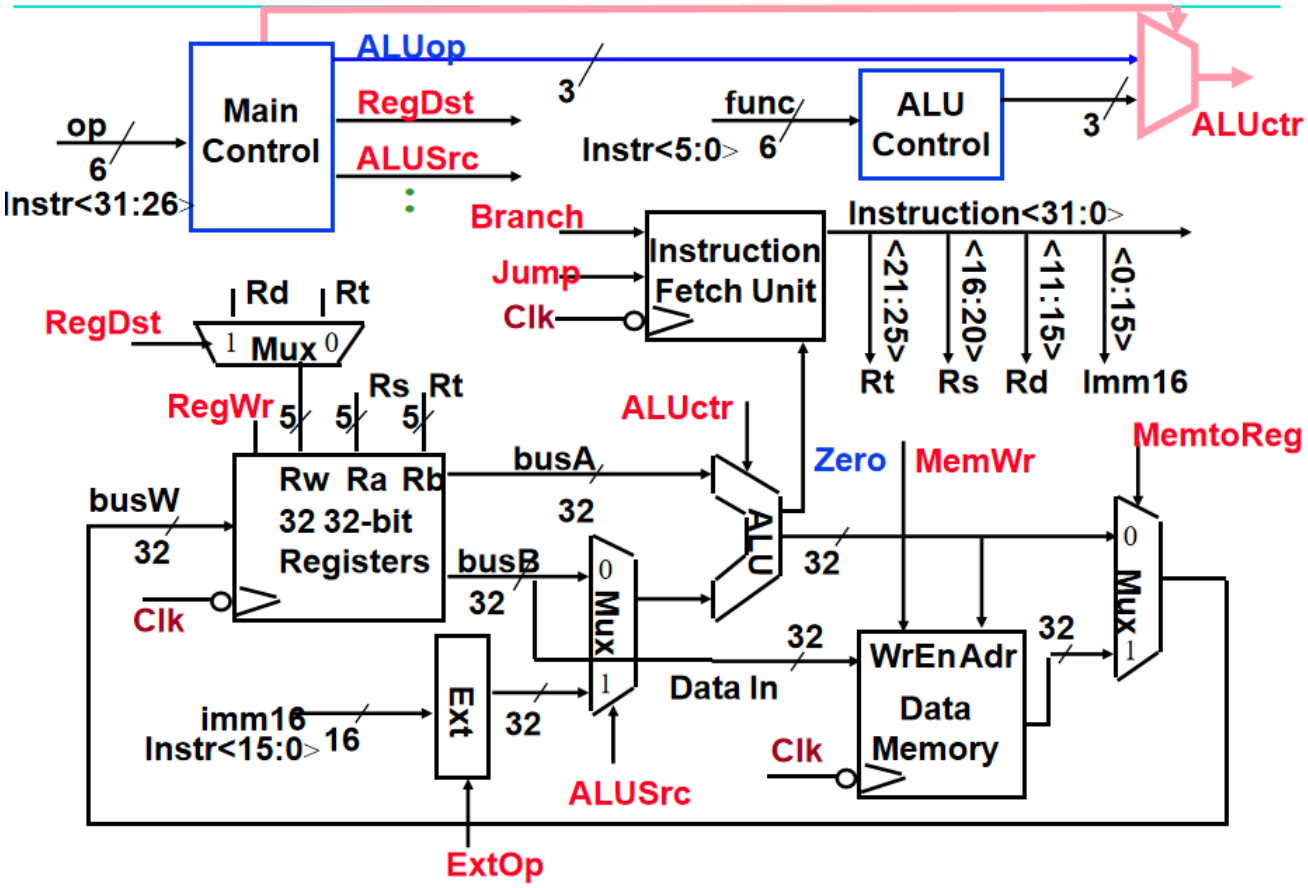
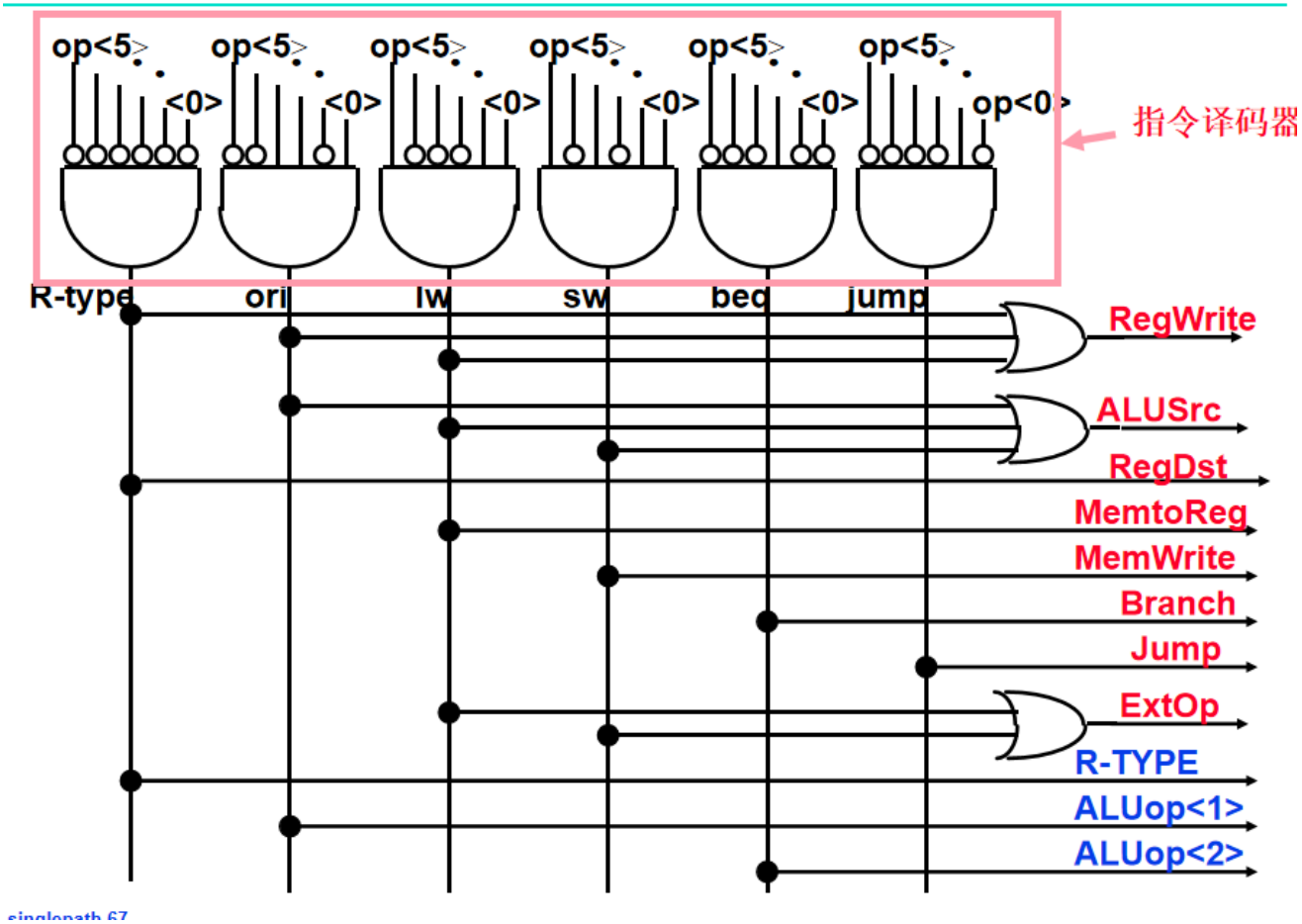
实验六：单周期CPU

设计思路





模块结构：

1. `singleCPU` 模块：

- **模块声明：**
  - 输入：时钟信号 `clk`、32位宽的指令信号 `I`。
  - 输出：多个控制和数据信号，包括 `busA`、`busB`、`AluB`、`AluF`、`busW`、`Datain`、`Dataout`、`RegWr`、`Branch`、`Jump`、`ExtOp`、`AluSrc`、`MemWr`、`MemtoReg`、`RegDst`。
- **ALU 控制信号 `ALUctr` 的声明：** 使用了一个 3 位宽的线 `ALUctr`，用于表示ALU的控制信号。
- **模块实例化：** 实例化了 `dataroad` 模块，并将输入输出连接。
- **指令解析：** 使用 `assign` 语句从输入的指令信号 `I` 中提取操作码 `op` 和功能码 `func`。
- `decoding` **模块实例化：** 实例化了 `decoding` 模块，并将输入输出连接。

2. `dataroad` 模块：

- **输入输出声明：** 输入包括时钟信号 `clk` 和多个控制信号，输出包括多个数据信号。
- **模块内部实现：** 该模块的内部包含了整个MIPS单周期处理器的数据通路控制逻辑，包括指令内存、寄存器、ALU、数据存储器等。

3. `decoding` 模块：

- **输入输出声明：** 输入包括操作码 `op` 和功能码 `func`，输出包括多个控制信号。
- **模块内部实现：** 该模块实现了对操作码和功能码的解析，产生对应的控制信号，用于控制MIPS单周期处理器的各个模块的行为。

通过这样的层次结构，整个代码实现了一个基本的MIPS单周期处理器，其中 `dataroad` 模块控制了数据通路，`decoding` 模块解析了指令并产生了相应的控制信号，而 `singleCPU` 模块则作为顶层模块将这些模块连接在一起。

数据通路模块（`dataroad`）：

1. `fetchins` 模块：

- **初始化指令内存：** 在 `initial` 块中，初始化了一个包含几条MIPS指令的指令内存 `inst_mem`。
- **指令获取：** 根据程序计数器（`pc`）的值，从指令内存中获取当前指令 `inst`。
- **跳转和分支目标计算：** 根据指令中的跳转和分支条件，计算下一个地址 `next_addr`。
- **程序计数器更新：** 在每个时钟的下降沿，将程序计数器 `pc` 更新为下一个地址 `next_addr`。

2. `Reg` 模块：

- **寄存器读写控制：** 根据输入的控制信号，判断是否进行寄存器的读取和写入操作。
- **寄存器读取和写入：** 根据输入的寄存器地址 `Ra` 和 `Rb`，从寄存器中读取数据。根据写入信号，将数据写入寄存器。

3. `MUX2to1` 模块：

- **2选1多路复用器：** 根据输入的选择信号 `ctr`，选择其中一个输入作为输出。

4. `alu` 模块：

- **ALU运算：** 根据输入的操作数 `A`、`B` 和ALU控制信号 `ALUctr` 进行ALU运算。
- **运算结果：** 输出运算结果 `Result`、零标志位 `Zero`、溢出标志位 `Overflow`。

5. `DataMemory` 模块：

- **数据存储器：** 根据输入的写入使能信号 `WrEn`，在时钟沿上升或写入使能信号上升沿时，将数据写入或读取数据存储器。

6. 主模块：

- **指令获取和解析：** 使用 `fetchins` 模块获取当前指令，并解析指令的操作码和操作数。
- **寄存器操作：** 使用 `Reg` 模块进行寄存器的读取和写入。
- **ALU运算：** 使用 `alu` 模块进行ALU运算，计算ALU的输出。
- **数据存储器操作：** 使用 `DataMemory` 模块进行数据存储器的读取和写入。
- **多路复用器操作：** 使用 `MUX2to1` 进行数据选择。

代码展示

```
module singleCPU(clk, I, busA, busB, AluB, AluF, busW, Datain, Dataout,
                 RegWr, Branch, Jump, ExtOp, AluSrc, MemWr, MemtoReg, RegDst);
    parameter n = 32;
    input clk;
    output [n-1:0] I, busA, busB, AluB, AluF, busW, Datain, Dataout;

    output wire RegWr, Branch, Jump, ExtOp, AluSrc, MemWr, MemtoReg, RegDst;
    wire [2:0] ALUctr;
    wire[5:0] op, func;
    dataroad dataroadbj(clk, RegWr, Branch, Jump, ExtOp, AluSrc, ALUctr,
MemWr, MemtoReg, RegDst,
    I, busA, busB, AluB, AluF, busW, Datain, Dataout);

    assign op = I[31:26];
    assign func = I[5:0];
    decoding qzbj(op, func, RegWr,Branch, Jump, ExtOp, AluSrc, ALUctr, MemWr,
MemtoReg, RegDst);
endmodule
```

波形分析

测试指令的功能

首先验证7条指令的功能。寄存器内容为:

```
regs[0]=32'h0000000a;
regs[1]=32'h00000005;
regs[2]=32'h00000000;
regs[3]=32'h00000003;
regs[4]=32'h00000000;
regs[5]=32'h00000000;
regs[6]=32'h00000000;
regs[7]=32'h00000000;
regs[8]=32'h00000000;
regs[9]=32'h00000002;
regs[10]=32'h0000000a;
regs[11]=32'h0000000b;
regs[12]=32'h0000000c;
regs[13]=32'h00000004;

for(i=14;i<32;i=i+1)
```

```
    regs[i]=32'h0;
```

存储器内容为

```
memory[4]=32'h0000001;
memory[5]=32'h0000002;
memory[6]=32'h0000003;
memory[7]=32'h0000004;
memory[8]=32'h0000005;
memory[11]=32'h00000064;
memory[12]=32'h000000cb;

for(i=13;i<256;i=i+1)
    memory[i]=32'h0;
```

设计一段C语言代码

```
int a = 10;
int b = 5;
int result;
//$0 a 10
//$1 b 5
//$2 result
//$3 3

result = a + b;

result = result - 3;

if (a < b) {
    result = 1;
} else {
    result = 0;
}

result = result | 0x0F;

int array[5] = {1, 2, 3, 4, 5};
int index = 2;
//$13 4
//$4 a[0] $5 a[1] $6 a[2] $7 a[3] $8 a[4]
//$9 index 2
```

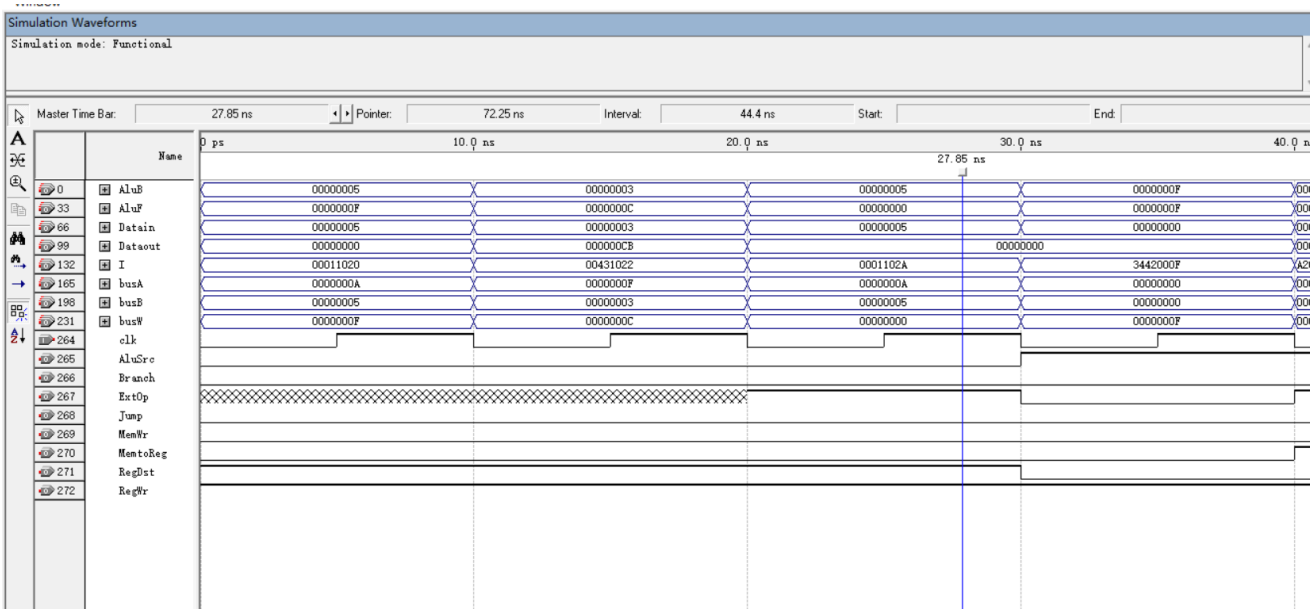
```
result = array[index];

array[index] = 10;
//$10 10

if (a == b) {
    result = 100;
} else {
    result = 200;
}
//$11 11
//$12 12
```

指令设计为:

```
0 add $2 $0 $1
000000 00010 00000 00001 00000 100000
4 sub $2 $2 $3
000000 00010 00010 00011 00000 100010
8 slt $2 $0 $1
000000 00010 00000 00001 00000 101010
12 ori $2 $2 0x0F
001101 00010 00010 0000000000001111
16 lw $2 $13 2
100011 01101 00010 0000000000000010
20 sw $10 $13 2
101011 01101 01010 0000000000000010
24 beq $0 $1 2
000100 00000 00001 0000000000000010
28 lw $2 $12 0
100011 01100 00010 0000000000000000
32 j 0xa
000010 00000000000000000000001010
36 lw $2 $11 0
100011 01011 00010 0000000000000000
```

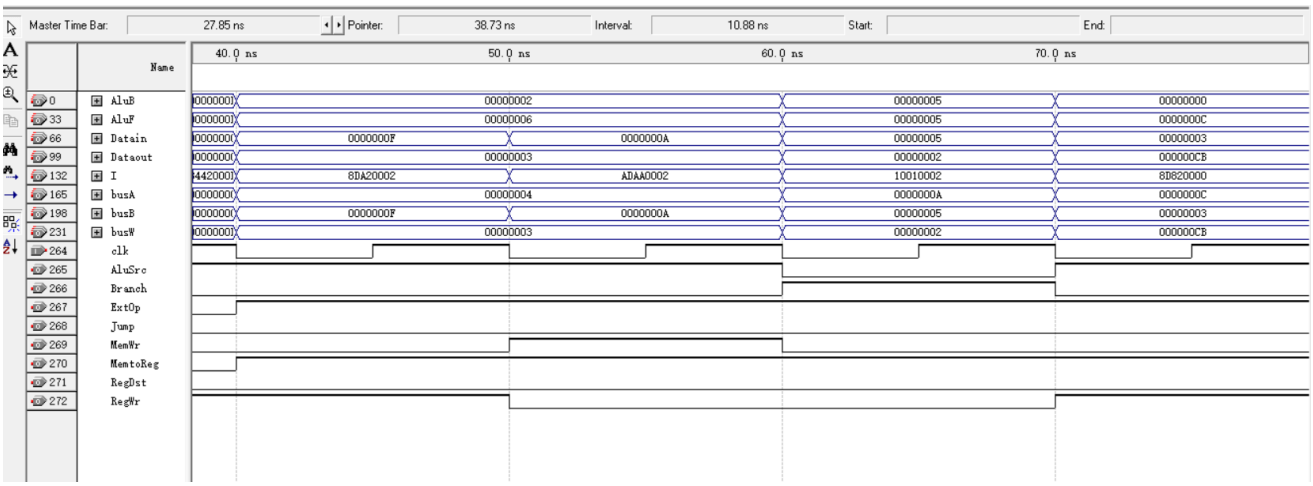


PC的初值为0，所以从Mem[0]指令开始,此指令为add $2,$0,$1,即$2=$0+$1=10+5=15。从图上可知，busA=10,AluB=5,AluF=15,符合要求。

接下来是Mem[4]指令，此指令是sub $2,$2,$3,即$2=$2-$3=15-3=12。从图上得到：busA=15,AluB=3,AluF=12,符合要求。

接下来是Mem[8]指令，此指令是slt $2,$0,$1,即$0<$1?$2=1:$2=0。从图上得到：busA=10,AluB=5,AluF=0,符合要求。

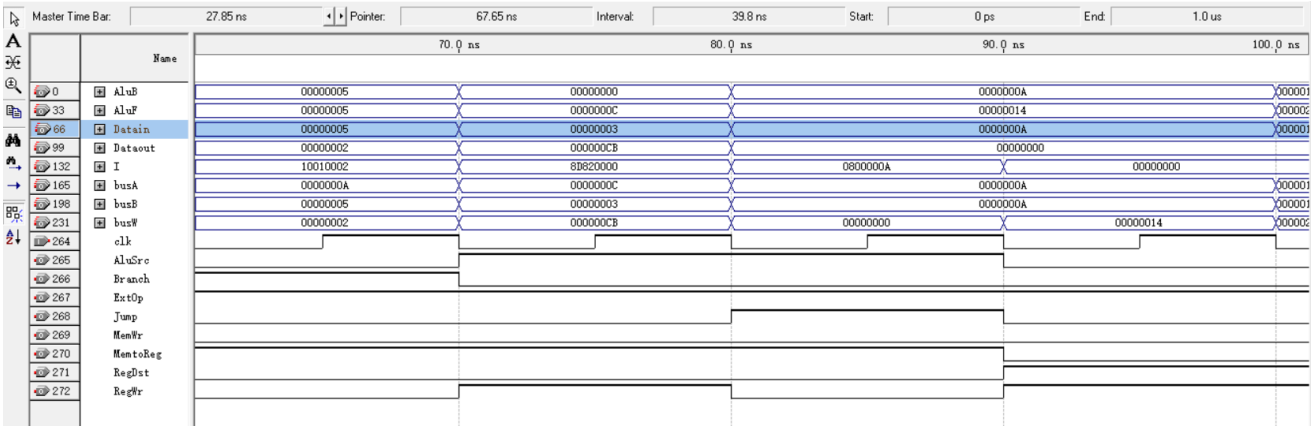
接下来是Mem[12]指令，此指令是ori $2 $2 0x0F,即$2=$2|0x0F=0|15=15。从图上得到：busA=0,AluB=15,AluF=15,符合要求。



接下来是Mem[16]指令，是lw $2=Memory[$13+2]，即$2=Memory[4+2]=3，从图上得到AluF=6,Dataout=busW=3,符合要求。

接下来是Mem[20]指令，是sw Memory[$13+2]=$10 即 Memory[6]=10。从图上得到：busB=Datain=10,AluF=6,符合要求。

接下来是Mem[24]指令，是beq $0(10),$1(5),2,此时$0和$1的内容相等，应该顺序执行，PC=28。



此时指令为{6'b100011, 5'b01100, 5'b00010, 16'b0000000000000000}符合要求，PC再加4Mem[32]<={6'b000010, 26'b00000000000000000000001010}为jump指令，因此需要继续跳转，跳转到PC=4*10=40,此时Mem[40]<=0，指令结束。

到此，七条指令全部测试成功。