

به نام او



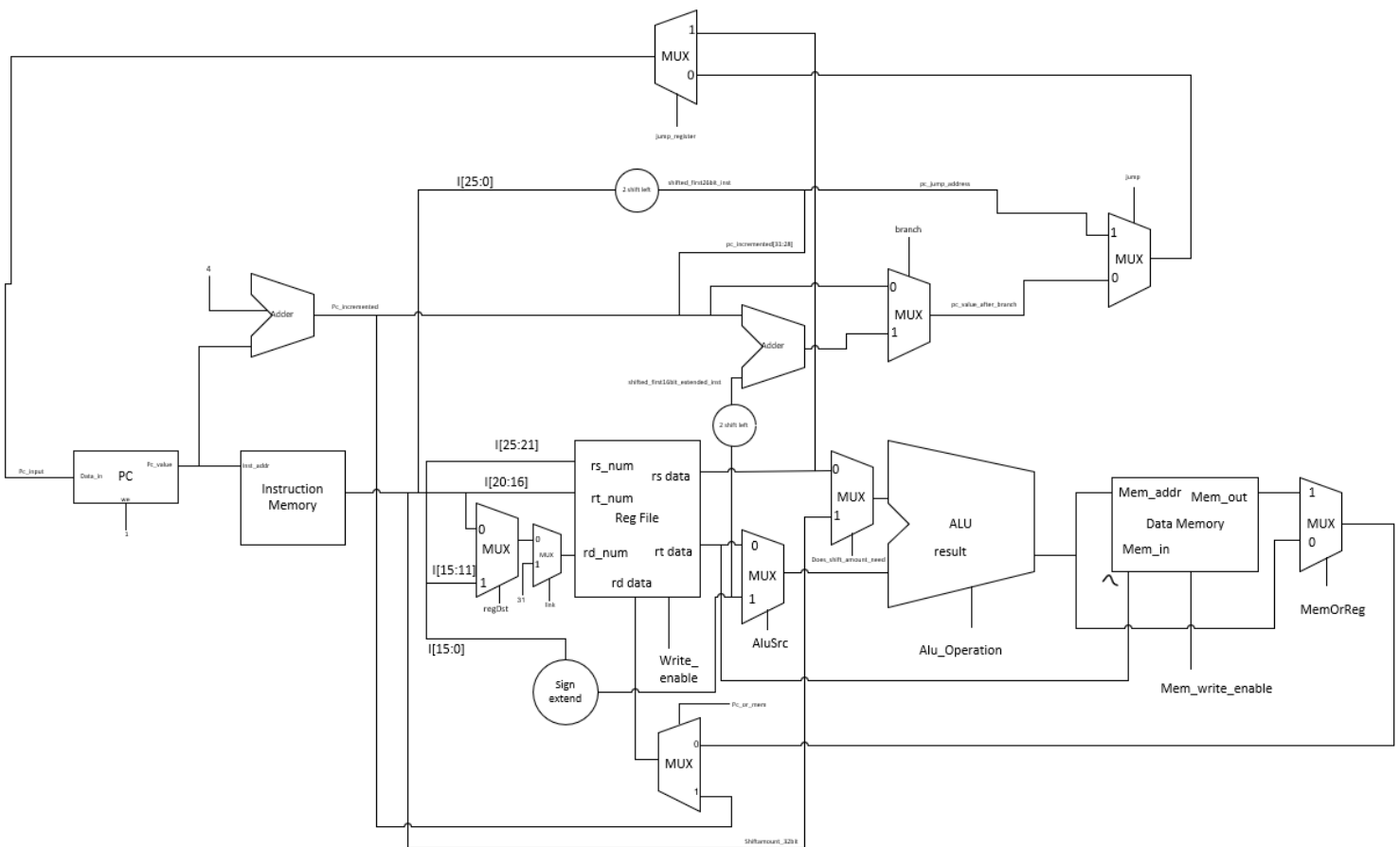
گزارش فاز اول پروژه معماری کامپیوتر

استاد سربازی

تیم: حسین گلی ، عرفان صدرائیه، امیرحسین براتی، علیرضا فرودنیا

طراحی پردازنده:

برای طراحی پردازنده آن را به 2 بخش **ControlUnit** و **DataPath** تقسیم میکنیم که عملاً **DataPath** وظیفه انجام عملیات های منطقی و حسابی بر روی مقادیر رجیستر ها را دارد و وظیفه **ControlUnit** مشخص کردن سیگنال های کنترلی مورد نیاز برای **DataPath** است . برای طراحی **DataPath** از نرم افزار **Visio** استفاده کردیم. **DataPath** استفاده شده در پروژه ما به شکل زیر است:



بخش های اصلی مسیر داده:

- 1- ALU
- 2- Register File
- 3- Branching Controller
- 4- Program Counter and Instruction Memory
- 5- Data Memory

بخش مربوط به Register File و Instruction/Data Memory به صورت آماده به ما داده شده است و ما باید صرفاً ورودی های آن را set کنیم که در DataPath داده شده ست شده اند.

تمامی سیگنال های کنترلی (branch- MuxControllers- ...) توسط واحد کنترل ست میشوند و وظیفه تنظیم آن ها با control unit است.

شرح ALU :

ALU که طراحی کردیم می تواند عملیات های زیر را پشتیبانی کند . وظیفه ALU انجام عملیات های محاسباتی بر دو ورودی خود است. ما برای توصیف ALU از توصیف behavioural استفاده کردیم.

```
XOR :  
OR :  
AND :  
NOR :  
SLL :  
SRL :  
ADD :  
ADDU :  
SUB :  
SUBU :  
MULT :  
DIV :  
SLT :  
SRA :  
LUI :  
NOP :
```

همچنین در طراحی ALU این که ALU چه عملیاتی انجام دهد با یک ورودی کنترلی ALU operation تولید میشود که وظیفه ALU controller تولید کردن این سیگنال است.

```

case (opcode)
    // R instructions
    6'b000000 :begin
        case (func)
            6'b000000 : alu_operation = SLL; //SLLV
            6'b000100 : alu_operation = SLL; //SLL
            6'b000010 : alu_operation = SRL; //SRL
            6'b000110 : alu_operation = SRL; //SRLV
            6'b000011 : alu_operation = SRA; //SRA
            6'b100110 : alu_operation = XOR; //XOR
            6'b100010 : alu_operation = SUB; // sub
            6'b101010 : alu_operation = SLT; //SLT
            6'b100011 : alu_operation = SUBU; //sub unsigned
            6'b100101 : alu_operation = OR; // OR
            6'b100111 : alu_operation = NOR; //NOR
            6'b100001 : alu_operation = ADDU; //add unsigned
            6'b011000 : alu_operation = MULT; //mult
            6'b011010 : alu_operation = DIV; //div
            6'b100100 : alu_operation = AND; //AND
            6'b100000 : alu_operation = ADD; //add
            default : alu_operation = NOP;
        endcase
    end
    // I instructions
    6'b001110 : alu_operation = XOR; //XORi
    6'b001010 : alu_operation = SLT; //SLTi
    6'b001000 : alu_operation = ADD; //ADDi
    6'b001100 : alu_operation = AND; //ANDi
    6'b001101 : alu_operation = OR; //ORi
    6'b001001 : alu_operation = ADDU; //ADDiu (unsigned)
    6'b000100 : alu_operation = SUB; //BEQ
    6'b000101 : alu_operation = SUB; //BNE
    6'b001111 : alu_operation = LUI; //LUI
    6'b000110 : alu_operation = NOP; //BLEZ
    6'b000111 : alu_operation = NOP; //BGTZ
    6'b000001 : alu_operation = NOP; //BGEZ
    6'b101011 : alu_operation = ADD; //SW
    6'b100011 : alu_operation = ADD; //LW
    default : alu_operation = NOP;

```

همانطور که در بخش snip شده از ALU Controller مشخص شده است وظیفه ALU controller تولید ALU operation در هر سیکل از CPU با توجه به opcode ورودی است. (همچنین opcode همانطور که در data path مشخص شده است decode شده و به ALU controller داده میشود).

شرح Branching Mechanism :

در پردازنده ما باید در هر سیکل دستوری که آدرس آن در رجیستر pc است را بخوانیم و سپس pc را با 4 جمع کنیم. اما اگر دستورات از نوع branch/jump/jr باشند لازم است pc را با مقدار جدید آپدیت کنیم در نتیجه چند ماکس در مسیر pc bus قرار دهیم که مقدار آدرس تولید branch که توسط ALU حساب می شود را در ورودی دوم خود دارند و سیگنال های آن توسط Control Unit ست میشوند.

سیگنال برنچ ورودی ماکس در مسیر داده است (در عکس اول داک نشان داده شده است)

```
always @(should_branch , zero , negative)begin
    branch = 0;
    if (should_branch)begin
        case(opcode)
            BEQ: if(zero) branch = 1;
            BNE: if(~zero) branch = 1;
            BLEZ: if(zero || negative) branch = 1;
            BGTZ: if(~negative && ~zero) branch = 1;
            BGEZ: if(~negative) branch = 1;
            default:begin end
        endcase
    end
end
```

شرح Control Unit :

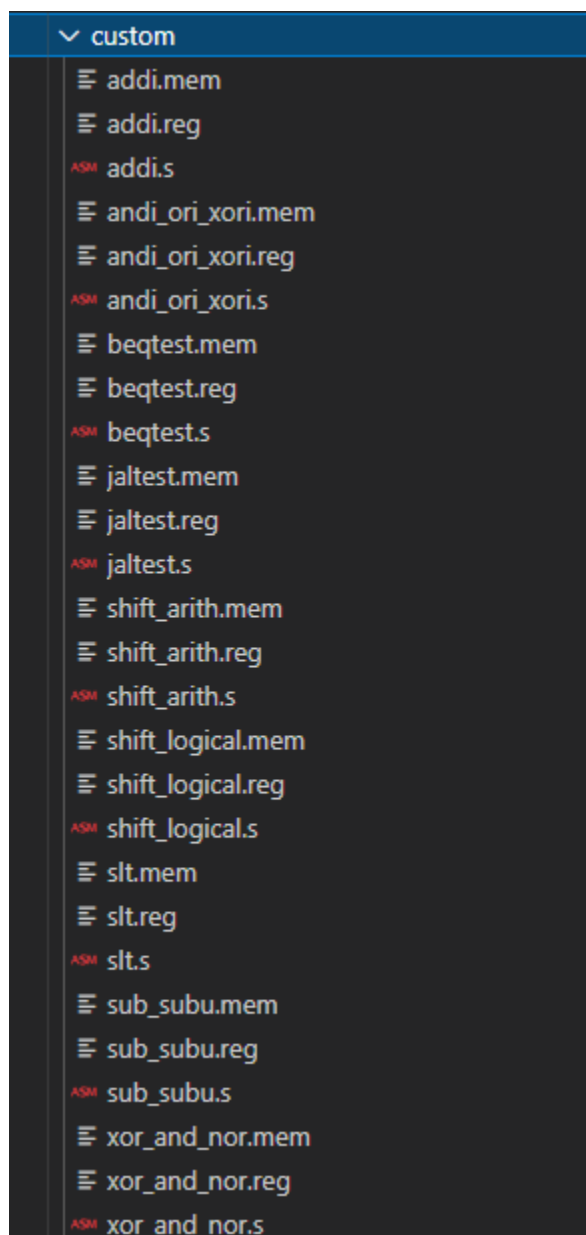
در Control Unit ما با استفاده از opcode و 6 بیت آخر دستور که func می شوند باید تصمیم بگیریم که سیگنال های کنترلی مسیر داده چه میشود .

همانطور که در یک نمونه از کد control unit نشان داده شده است این بخش عملیات switch case است که به MUX سنتز میشوند و در هر case سیگنال های مورد نیاز data path با استفاده از نوع

دستور ست میشوند.

```
reg_write_enable = 1;
alu_src = 1;
end
ANDi: begin
    reg_write_enable = 1;
    alu_src = 1;
    is_unsigned = 1;
end
XORi: begin
    reg_write_enable = 1;
    alu_src = 1;
    is_unsigned = 1;
end
ORi: begin
    reg_write_enable = 1;
    alu_src = 1;
    is_unsigned = 1;
end
SLTi : begin
    reg_write_enable = 1;
    alu_src = 1;
end
JAL : begin
```

همچنین در این پروژه علاوه بر تست های عادی تست های دیگری نوشتیم و با استفاده از gnu mips assembler آن ها را assemble کردیم و با استفاده از نرم افزار MARS کد را شبیه سازی و reg file را ساختیم . تست های کاستوم اضافه شده در دایرکتوری custom :



همانطور که در نتیجه شبیه سازی نشان داده شده است تمامی 7 تست و تست های custom پاس میشوند و پردازنده ما به درستی کار میکند:

```
diff -u test/custom/xor_and_nor.reg output/regdump.reg 1>&2
make[1]: Leaving directory '/home/hogo/CA-Project'
All tests passed! (16 tests)
hogo@LAPTOP-4QJ4TM83:~/CA-Project$
```