بسمه تعالی

گزارش معماری کامپیوتر

تمرین هشتم : طراحی پردازنده پایپ لاین MIPS

استاد: دکتر اسدی

نویسنده: سپهر کیانیان گل افشانی(98102154)

دانشگاه صنعتی شریف بهار ۰۰-۹۹

فهرست

مقدمه و هدف

هدف از انجام این تمرین طراحی یک پردازنده پایپ لاین MIPS است.

شرح آزمایش

در این آزمایش، باید همان مدار آزمایش قبلی را پایپ لاین کنیم.

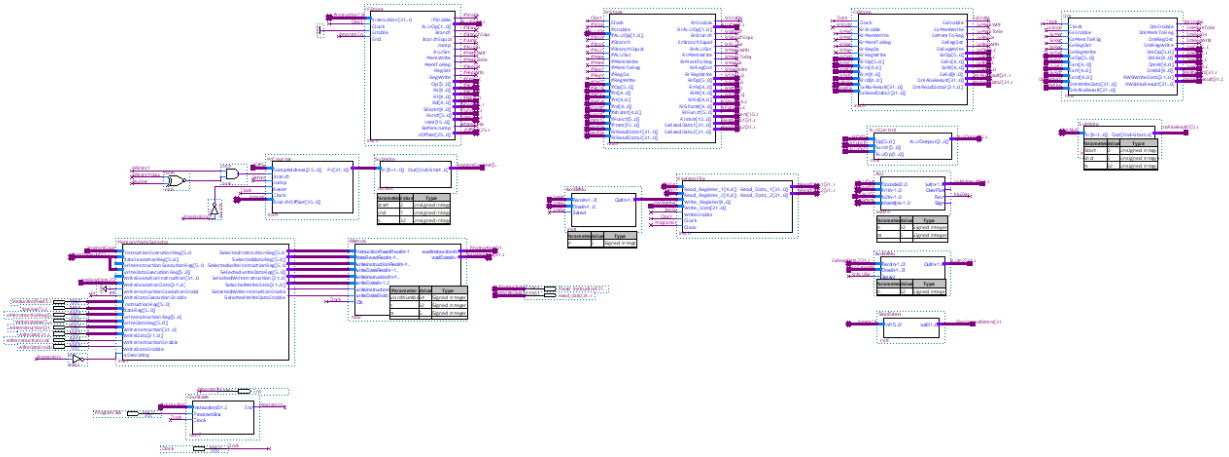
برای پایپ لاین کردن مدار، باید بین مراحل مختلف رجیستر های میانی قرار دهیم، و همچنین control Unit های برای تشخیص Hazard های مختلف قرار دهیم.

در این آزمایش ما DataHazard را به روش DataForwarding و همچنین Control Hazard را با ۲ کلاک Stall رفع می کنیم.

لازم به ذکر است تمام Component های مراحل مختلف و همچنین ControlUnit های مربوط به این بخش در پوشه Pipeline واقع در پوشه آزمایش قرار دارند.

روش کار:

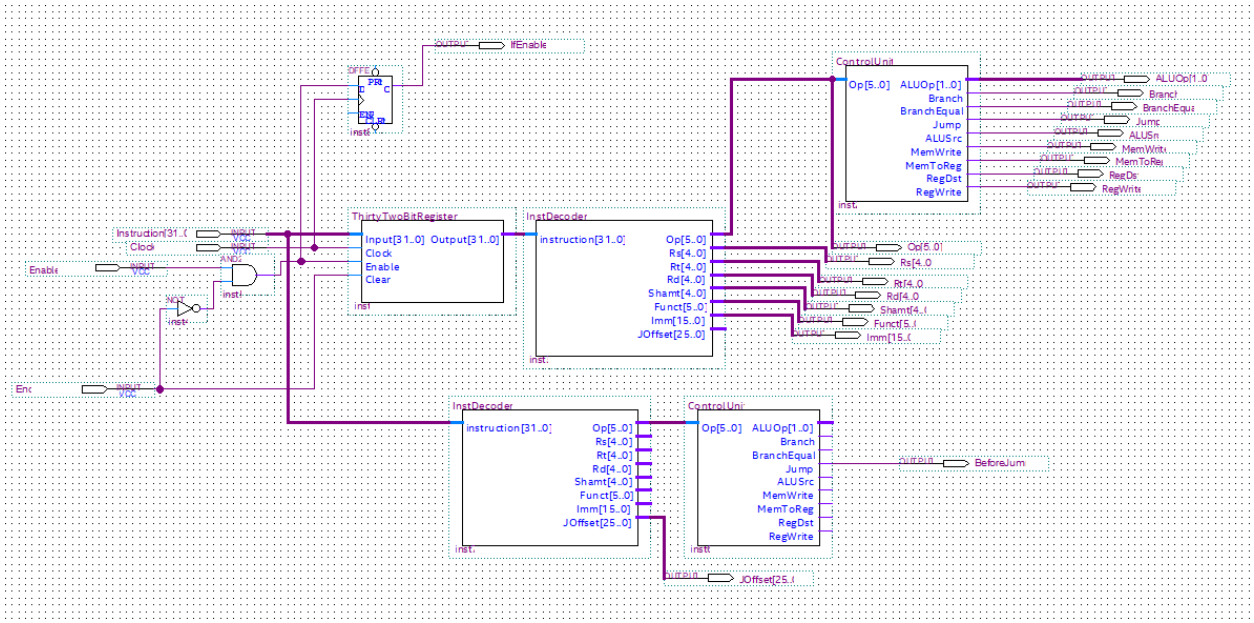
در ابتدا بین مراحل مختلف مدار، رجیستر های میانی قرار می دهیم، و همچنین لازم به ذکر است مدار اصلی ما در DataPath واقع است. شکل زیر شماتیک بعد قرار دادن رجیستر های میانی است و فایل آن DataPath.bdf است.



رجیستر های میانی هر مرحله، به صورت یک Component در آمده، و به جز IfStage بقیه Component ها فقط شامل رجیستر های میانی اند.

هر Component هم یک ورودی و خروجی Enable دارد که با استفاده از آن ها غیرفعال شدن و فعال شدن این Component ها اتفاق می افتد.

IfStage در هر کلاک دستور خوانده شده از Inst.Memory را Decode می کند و همچنین با استفاده از واحد Control Unit سیگنال های مربوط به دستور تولید می شود. شکل زیر شماتیک IfStage است و فایل آن در پوشه Pipeline قرار دارد و IfStage.bdf است.



سپس، مدار DataForwarding را درست می کنیم و وارد مدار اصلی می کنیم.

۱. بخش DataForwarding

با توجه به این که به ما اجازه داده شده که بخش های کنترلی را با Verilog بزنیم، این بخش با وریلاگ زده شده.

در این بخش، ما برای رفع DataHazard مرحله Dm را با مراحل Rr, Ex مقایسه می کنیم. چون در مرحله Dm نتایجی که می خواهد در رجیستر Write شود استفاده می گردد (برای Rtype, Itype اول مرحله و برای LW مقدار خوانده شده از Memory) و در مرحله ۵ هم نیز چون از latch استفاده کرده ایم، دیگر Hazard ای نداریم.

حال، آن رجیستری که می خواهد در Register ریخته شود را با رجیستر های Rs, Rt مراحل Rr, Ex چک می کنیم. اگر هرکدام از آن ها یکی بود، در آن مرحله به جای آن ها از مقدار مرحله Dm استفاده می کنیم. برای این بخش، در DataPath از مالتی پلکسر های مختلف استفاده کرده ایم، و در DataForwardingUnit فقط سیگنال تولید می کنیم.

کد این بخش به صورت زیر است و فایل آن در پوشه Pipeline و DataForwardingUnit.v است.

```
module DataForwardingUnit(Enable, DmOp, DmDs, IfRs, IfRt, RrRs, RrRt,
    SelectBitForDmCalcs,
    SelectBitForDmRrRead,
    SelectBitForDmRrRead2,
    SelectBitForDmCalcsRead,
    SelectBitForDmCalcsRead2);

    input Enable;
    input [5 : 0] DmOp;
    input [4 : 0] DmDs;
    input [4 : 0] IfRs, IfRt, RrRs, RrRt;
    output SelectBitForDmCalcs,
           SelectBitForDmRrRead,
           SelectBitForDmRrRead2,
           SelectBitForDmCalcsRead,
           SelectBitForDmCalcsRead2;

    wire ifItsForDataHazard;
    assign ifItsForDataHazard = DmOp == 34 | DmOp == 0 | DmOp == 8 | DmOp == 9 | DmOp == 10 | DmOp == 12 | DmOp == 14;

    assign SelectBitForDmCalcs = Enable & ifItsForDataHazard & (DmOp == 34);
    assign SelectBitForDmRrRead = Enable & ifItsForDataHazard & (DmDs == IfRs);
    assign SelectBitForDmRrRead2 = Enable & ifItsForDataHazard & (DmDs == IfRt);
    assign SelectBitForDmCalcsRead = Enable & ifItsForDataHazard & (DmDs == RrRs);
    assign SelectBitForDmCalcsRead2 = Enable & ifItsForDataHazard & (DmDs == RrRt);
endmodule
```

حال، مدار BranchHazard را درست می کنیم.

۲. بخش ControlHazardDetectionUnit :

در این بخش، ما Op دریافت شده از مرحله If را بررسی می کنیم، و اگر Bne یا Beq بود، مدار رو دو حباب Stall می کنیم، و ProgramCounter را اصلا تغییر نمی دهیم. سپس بعد دو حباب Stall سیگنال Branch مشخص می کند که باید Branch کنیم یا نه، و از آن به بعد مدار به روال عادی خود باز می گردد.

برای دو بار Stall کردن مدار، سیگنال Enable برای بخش IfUnit را دو کلاک 0 می کنیم، و چون این سیگنال های Enable از هر مرحله به مرحله دیگر منتقل می شود، با این کار عملا انگار دو nopInstruction در اینجا قرار دادیم.بعد آن دو کلاک دوباره سیگنال را 1 می کنیم.

کد مدار در شکل زیر مشخص است و فایل آن در پوشه Pipeline و ControlHazardDetectionUnit.v است.

```
module ControlHazardDetectionUnit(Clock, Jump, Branch, Op, outJump, outBranch, stall);
    input Clock;
    input Jump, Branch;
    input [5 : 0] Op;

    output outJump, outBranch;
    output stall;

    assign outJump = state[1] ? (0) : (state[0] ? (1) : (Jump));
    assign outBranch = state[1] ? (Branch) : (state[0] ? (1) : (Branch));
    assign stall = state != 2'b00;

    reg[1 : 0] state = 2'b00;

    always @(posedge Clock) begin
        if ( state == 2'b00 ) begin
            if ( Op == 4 || Op == 5 ) begin            // Branch
                state <= 2'b01;
            end
        end
        else if ( state == 2'b01 ) begin
            state <= 2'b10;
        end
        else if ( state == 2'b10 ) begin
            state <= 2'b00;
        end
        else begin
            state <= 2'b00;
        end
```

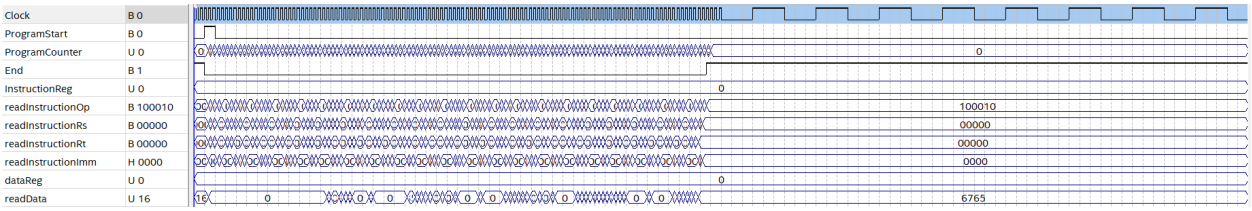
شبیه سازی کلی

در اینجا ما یک کد آماده کردیم برای محاسبه کردن تابع فیبوناچی، که اندیس صفرم حافظه را ورودی می گیرد و یه سری عملیات انجام می دهد، و خروجی را در همان اندیس حافظه قرار می دهد.

کد تابع در واحد Memory قرار گرفته و به صورت زیر است. این کد را در همان ابتدا در واحد حافظه قرار می دهیم.

```
numbers[0] <= 32'b100010_00000_00000_0000000000000000;   // Lw S0, S0(0)          (S0 = Mem[0](n))
numbers[1] <= 32'b001000_00010_00010_0000000000000001;   // Addi S2, S2, 1        (S2 (F1) = 1, S1 (F0) = 0)
numbers[2] <= 32'b001001_00000_00000_0000000000000001;   // L1: Subi S0, S0, 1    (S0 -= 1)
numbers[3] <= 32'b000000_00010_00001_00011_00000_100000; // Add S3, S2, S1        (S3 = S2 + S1 -> Fn+1 = Fn + Fn-1)
numbers[4] <= 32'b001000_00010_00001_0000000000000000;   // Addi S1, S2, 0        (S1 = S2 -> Fn-1 = Fn)
numbers[5] <= 32'b001000_00011_00010_0000000000000000;   // Addi S2, S3, 0        (S2 = S3 -> Fn = Fn+1)
numbers[6] <= 32'b101011_00000_00001_0000000000000000;   // SW S1, S0(0)          (Mem[0] = S1(Fn))
numbers[7] <= 32'b000100_00111_00000_1111111111111010;   // BNE S0, S4, L1        (If S2 != S4 (0), Then Branch L1)
numbers[8] <= 32'b111111_00000_00000_0000000000000000;   // End Of The Program (OP=6'b111111)
numbers[wordNumber - 1] <= 16;                                    // n = 16 (Because Of First Input Increases);
for(i = 9; i < wordNumber - 1; i = i + 1) begin
    numbers[i] <= 0;
end
```

حال، شبیه سازی را اجرا می کنیم و می بینیم مدار درست کار کرده یا خیر.



در ابتدا Programstart را یک کردیم تا مدار شروع به کار کند، و مدار بعد از کلی کلاک (حدود ۱۵۰ کلاک) ۲۰ امین مقدار فیبوناچی را برای ما محاسبه کرد.

یک ورودی dataReg هم برای مدار قرار داده ایم که بتوانیم با استفاده از آن در زمان هایی که End یک است، از حافظه داده بخوانیم. خروجی حافظه هم از readData مشخص می شود. با توجه به آزمایش می بینیم بعد آن که End یک شد، اندیس صفرم داده ها 6765 شد که درست است.

با توجه به شبیه سازی، می بینیم مدار بعد از کلی کلاک نتیجه را در حافظه ذخیره کرده است و ما می توانیم آن را بخوانیم.

فایل شبیه سازی WaveForm8.vwf است.

منابع و مباحث:

Community.intel.com

صحبت های استاد محترم سر کلاس

اسلاید های درس

کلاس حل تمرین

سایت های آموزشی زبان Verilog

Samir Palnitkar. Verilog HDL A Guide to Digital Design and Synthesis.

2nd edition, SunSoft Press, 2003