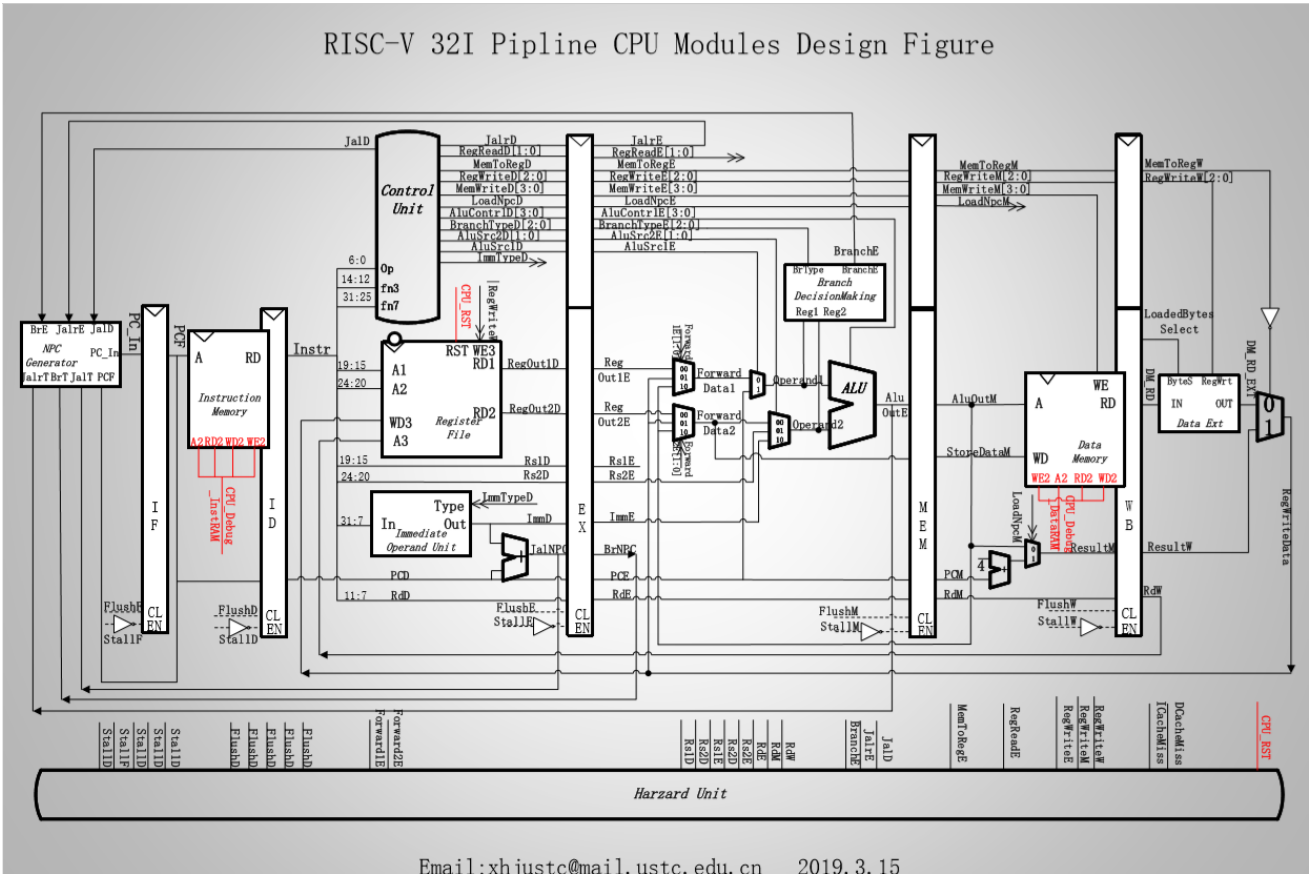
lab1 RISC-V32位流水线CPU设计报告

一.设计图



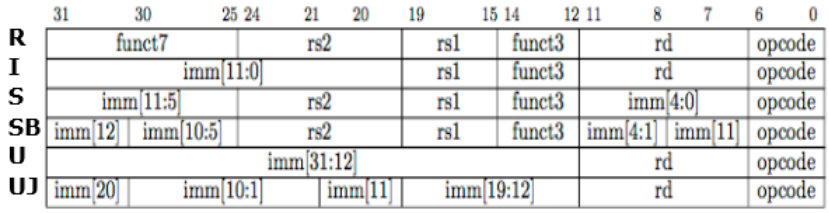
二.RISC-V架构分析

1.寄存器

- 32个32位寄存器
- x0为零寄存器，x1-x31为31个通用寄存器
- 返回地址: 一般用x1
- pc: 额外的用户可见寄存器

2.基本指令格式

- 四种基础指令格式 R/I/S/U
- 字段解释
  - imm：立即数，opcode：操作码
  - rs1：源寄存器1，rs2：源寄存器2，rd：目标寄存器
- 指令格式图示:

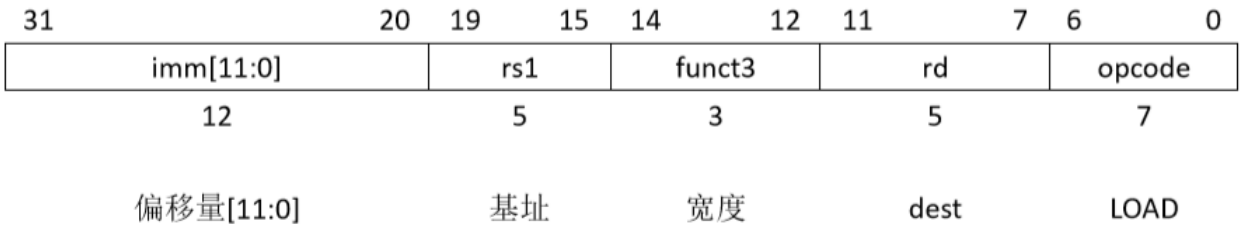


三.指令分析

> 共47条指令，按功能分为：loads(5条), stores(3条), shifts, qrithmetic, logical, compare, branches, jump&link, synch, system, counters，实验需实现前37条

1.Loads类（5条）

指令	指令语义	类型	指令格式
LB	加载8位有符号数	I	LB rd,rs1,imm
LH	加载16位有符号数	I	LH rd,rs1,imm
LW	加载32位有符号数	I	LW rd,rs1,imm
LBU	加载8位无符号数	I	LBU rd,rs1,imm
LHU	加载16位无符号数	I	LHU rd,rs1,imm



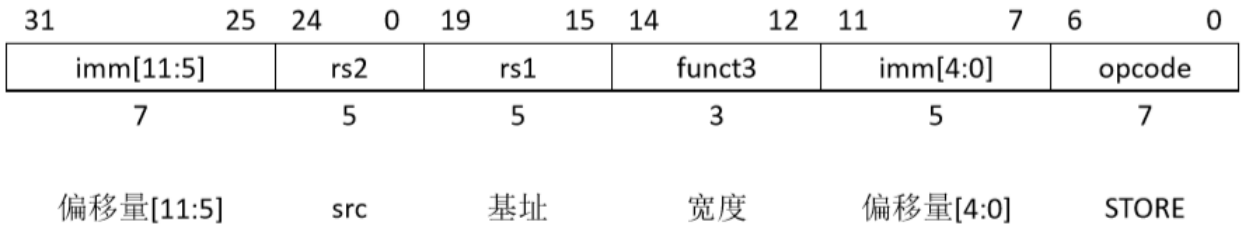
数据通路分析：

- 将地址为rs1+imm存储单元的内容，读入rd寄存器中
- 以LB为例，指令译码后得到的各个字段译码输出：

```
JalD==0
JalrD==0
RegReadD[1]==1
RegReadD[0]==0
MemToRegD==1
RegWriteD[2:0]==3'd1
MemWriteD[3:0]=0000
LoadNpcD==1
AluContrlD[3:0]=ADD
BranchTypeD=NOBRANCH
AluSrc2D[1:0]=10
AluSrc1D=0
ImmType=ITYPE
```

2.Stores类（3条）

指令	指令语义	类型	指令格式
SB	将寄存器rs2的值(低8位)，存入地址为rs1+imm的存储单元	S	SB rs1,rs2,imm
SH	将寄存器rs2的值(低16位)，存入地址为rs1+imm的存储单元	S	SH rs1,rs2,imm
SW	将寄存器rs2的值(32位)，存入地址为rs1+imm的存储单元	S	SW rs1,rs2,imm



数据通路分析：

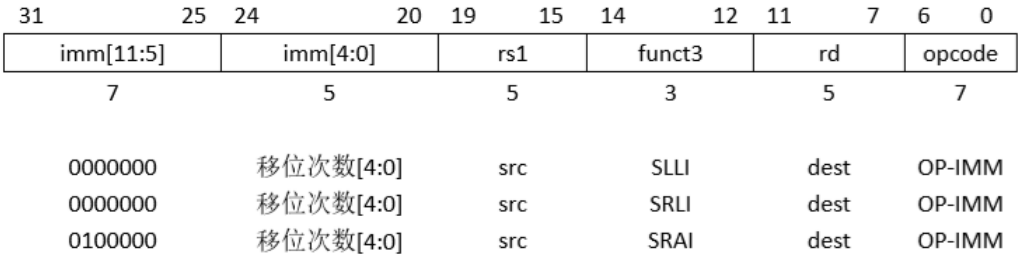
- SB指令译码输出

```
JalD==0,JalrD==0,RegReadD[1]==1,RegReadD[0]==0,
MemToRegD==0,RegWriteD[2:0]==3'd0 ,MemWriteD[3:0]=0001, LoadNpcD==1
,AluContrlD[3:0]=ADD,BranchTypeD=NOBRANCH, AluSrc2D[1:0]=10, AluSrc1D=0,
ImmType=STYPE
```

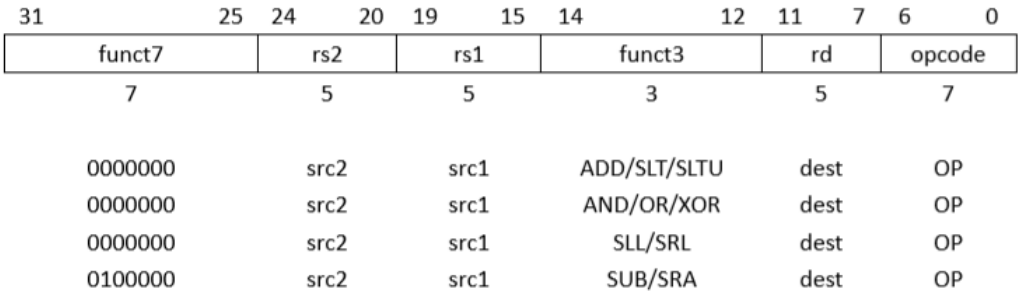
3.Shifts类（6条）

指令	指令语义	类型	指令格式
SLL	逻辑左移	R	SLL rd,rs1,rs2
SLLI	逻辑立即数左移	I	SLLI rd,rs1,shamt
SRL	逻辑右移	R	SRL rd,rs1,rs2
SRLI	逻辑立即数右移	I	SRLI rd,rs1,shamt
SRA	算数右移	R	SRA rd,rs1,rs2
SRAI	算数立即数右移	I	SRAI rd,rs1,shamt

- I型移位：



- R型移位：



4.Arithmetic类（5条）

指令	指令语义	类型	指令格式
ADD	加	R	ADD rd,rs1,rs2
ADDI	立即数加	I	ADDI rd,rs1,imm
SUB	减	R	SUB rd,rs1,rs2
LUI	将立即数放到目标寄存器rd的高20位，将rd的低12位填0	U	LUI rd,imm
AUIPC	从20 位立即数构建一个32位偏移量，将其低12位填0，然后将这个偏移量加到pc上，最后将结 果写入寄存器rd	U	AUIPC rd,imm

5.Logical类（6条）

指令	类型	指令格式
XOR	R	XOR rd,rs1,rs2
XORI	I	XORI rd,rs1,imm
OR	R	OR rd,rs1,rs2
ORI	I	ORI rd,rs1,imm
AND	R	AND rd,rs1,rs2
ANDI	I	ANDI rd,rs1,imm

6.Compare类（4条）

指令	指令语义	类型	指令格式
SLT	<	R	SLT rd,rs1,rs2
SLTI	<(立即数)	I	SLTI rd,rs1,imm
SLTU	<(无符号数)	R	SLTU rd,rs1,rs2
SLTIU	<(无符号立即数)	I	SLTIU rd,rs1,imm

7.Branches类（6条）

指令	指令语义	类型	指令格式
BEQ	=	SB	BEQ rs1,rs2,imm
BNE	!=	SB	BNE rs1,rs2,imm
BLT	<	SB	BLT rs1,rs2,imm
BGE	>=	SB	BGE rs1,rs2,imm
BLTU	<(无符号)	SB	BLTU rs1,rs2,imm
BGEU	>=(无符号)	SB	BGEU rs1,rs2,imm

8.Jump&Link类（2条）

指令	指令语义	类型	指令格式
JAL	20位imm+pc得到跳转地址,将跳转指令后面指令的地址（pc+4）保存到寄存器rd中	UJ	JAL rd,imm
JALR	12位imm+(rs1)再结果最低位设置为0，作为目标地址，(pc+4)保存到rd中	UJ	JALR rd,rs1,imm

四.待完成模块设计思路

> 按照给定设计图的模块划分，总计9个待完成模块:

1. **生成新PC模块** `NPC_Generator.v`
    - 用来生成Next PC值得模块，根据不同的跳转信号选择不同的新PC值
    - 设计思路：无跳转信号时，PC=PC+4。有跳转信号时，PC=跳转地址
    - 输入

```
[31:0]PCF           //旧的PC值
[31:0]JalrTarget    //jalr指令的对应的跳转目标
[31:0]BranchTarget  //branch指令的对应的跳转目标
[31:0]JalTarget     //jal指令的对应的跳转目标
BranchE==1        //Ex阶段的Branch指令确定跳转
JalD==1           //ID阶段的Jal指令确定跳转
JalrE==1          //Ex阶段的Jalr指令确定跳转
```

    - 输出

```
[31:0]PC_In             //NPC的值
```

2. **指令存储器+ID级寄存器模块** `IDSegReg.v`

- 同步读memory 相当于 异步读memory 的输出外接D触发器，需要时钟上升沿才能读取数据。此时如果再通过段寄存器缓存，那么需要两个时钟上升沿才能将数据传递到Ex段。因此在段寄存器模块中调用该同步memory，直接将输出传递到ID段组合逻辑
- 设计思路：在ID级寄存器中例化指令存储器，实现接口连接，实现段寄存器stall和clear功能
- 输入

```
clk
clear,
en,
//Instrution Memory Access
[31:0] A
//Instruction Memory Debug
[31:0] A2
[31:0] WD2
[3:0] WE2
//
[31:0] PCF
```

- 输出

```
[31:0] RD
[31:0] RD2
[31:0] PCD
```

3. **译码模块** `ControlUnit.v`
   - 本CPU的指令译码器，组合逻辑电路
   - 设计思路：根据指令的字段，输出相应控制信号
   - 输入

```
[6:0]Op           //是指令的操作码部分
[2:0]Fn3          //是指令的func3部分
[6:0]Fn7          //是指令的func7部分
```

   - 输出

```
JalD==0          //不为Jal指令
JalrD==0         //不为Jalr指令
RegReadD[1]==1   //表示A1对应的寄存器值被使用到了，
RegReadD[0]==0   //表示A2对应的寄存器值没被使用到，用于forward的处理
MemToRegD==1     //表示需要将data memory读取的值写入寄存器
RegWriteD[2:0]==3'd1  //不写寄存器为0,LB为1,LH为2,LW为3,LBU为4,LHU为5
MemWriteD[3:0]=0000  //仅store指令使用
//共4bit，采用独热码格式，对于data memory的32bit字按byte进行写入,MemWriteD=0001表示只写入最低1个byte，和xilinx bram的接口类似
LoadNpcD==0          //表示将NextPC输出到ResultM
AluContrlD[3:0]=ADD  //表示不同的ALU计算功能，
BranchTypeD=NOBRANCH //表示不同的分支类型，所有类型定义在Parameters.v中
AluSrc2D[1:0]=10     //表示Alu输入源2的选择，00来自操作数2，01来自寄存器2，10来自立即数
```

```
AluSrc1D=0          //表示Alu输入源1的选择，0代表来自操作数1，1代表来自PC
ImmType=ITYPE       //表示imm格式，与指令类型有关
```

4. **立即数扩展** `ImmOperandUnit.v`
   - 利用正在被译码的指令的部分编码值，生成不同类型的32bit立即数
   - 设计思路：根据不同的指令类型，将指令中立即数的各个字段拼接和扩展，得到一个可使用的32位立即数
   - 输入

```
[31:7] In
[2:0] Type
```

   - 输出

```
[31:0] Out
```

5. **ALU模块** `ALU.v`
6.
   - ALU接受两个操作数，根据AluContrl的不同，进行不同的计算操作，将计算结果输出到AluOut
   - 设计思路：根据ALU功能码，实现算术运算
   - 输入

```
[31:0] Operand1
[31:0] Operand2
[3:0] AluContrl
```

   - 输出

```
[31:0] AluOut
```

7. **branch分支指令判定模块** `BranchDecisionMaking.v`
   - 设计思路:ranchDecisionMaking接受两个操作数，根据BranchTypeE的不同，进行不同的判断，当分支应该taken时，令BranchE=1'b1
   - 输入：

```
[2:0] BranchTypeE
[31:0] Operand1
[31:0] Operand2
```

   - 输出：

```
BranchE
```

8. **数据存储器+WB级寄存器模块** `WBSegReg.v`
   - 设计思路: 配合DataExt模块，实现字节，半字，字的load和store指令

- 输入&输出：

```
input wire clk,
input wire en,
input wire clear,
//Data Memory Access
input wire [31:0] A,
input wire [31:0] WD,
input wire [3:0] WE,
output wire [31:0] RD,
output reg [1:0] LoadedBytesSelect,
//Data Memory Debug
 input wire [31:0] A2,
 input wire [31:0] WD2,
 input wire [3:0] WE2,
 output wire [31:0] RD2,
 //input control signals
 input wire [31:0] ResultM,
 output reg [31:0] ResultW,
 input wire [4:0] RdM,
 output reg [4:0] RdW,
 //output constrol signals
 input wire [2:0] RegWriteM,
 output reg [2:0] RegWriteW,
 input wire MemToRegM,
 output reg MemToRegW
```

9. **数据扩展** `DataExt.v`
   - 实现思路：DataExt是用来处理非字对齐load的情形，同时根据load的不同模式对Data Mem中load的数进行符号或者无符号拓展，组合逻辑电路
   - 输入：

```
[31:0] IN//存储器的原始读取结果
[1:0] LoadedBytesSelect //字节选择
[2:0] RegWriteW //选择lb,lh,lw等
```

   - 输出：

```
[31:0] OUT //OUT表示要写入寄存器的最终值
```

10. **冒险处理** `HarzardUnit.v`
    - 实现转发和flush
    - 实现思路：通过转发机制处理一阶数据相关(EX冒险)，二阶数据相关(MEM冒险)，三阶数据相关(同一周期内同时读写同一个寄存器)。通过冒险+流水线阻塞处理处理数据冒险。通过提前分支指令实现控制冒险
    - 输入&输出：

```
    input wire CpuRst, ICacheMiss, DCacheMiss,
    input wire BranchE, JalrE, JalD,
    input wire [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    input wire [1:0] RegReadE,
    input wire [2:0] MemToRegE, RegWriteM, RegWriteW,
    output reg StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW,
    output reg [1:0] Forward1E, Forward2E
```

五.问答题

1.为什么将 DataMemory 和 InstructionMemory 嵌入在段寄存器中?

- 这样可同步读，时钟上升沿读新值。

2.DataMemory 和 InstructionMemory 输入地址是字（32bit）地址，如何将访存地址转化为字地址输入进去?

- 物理访存地址为4的倍数，对于非字对齐的地址，将最低两位设置为0，其他位数不变，即可得到用于输入的字地址。

3.如何实现 DataMemory 的非字对齐的 Load?

- 将原访存地址的最后两位记录在LoadedBytesSelect信号中，用于选择字节地址，译码阶段的RegWriteW信号用来区别lh,lw,lb。根据这两个信号进行选择和扩展，得到最后存入寄存器的值。

4.如何实现 DataMemory 的非字对齐的 Store?

- 通过MemWrite[3:0]信号,这是一个4位独热码，每位代表一个字节，该位为1及代表对一个字中对应字节进行写操作。比如SW就是1111，SB就是0001/0010/0100/1000(具体是哪个根据字节地址确定)。

5.为什么 RegFile 的时钟要取反?

- 相当于不采用同步读，比较方便五级流水，不用在内部转发。

6.NPC_Generator 中对于不同跳转 target 的选择有没有优先级?

- 执行越靠后越优先。

7.ALU 模块中，默认 wire 变量是有符号数还是无符号数?

- 无符号数。

8.AluSrc1E 执行哪些指令时等于 1'b1?

- ALU，PC指令

9.AluSrc2E 执行哪些指令时等于 2'b01?

- 左移/右移

10.哪条指令执行过程中会使得 LoadNpcD==1?

- jalr, jal

11.DataExt 模块中，LoadedBytesSelect 的意义是什么?

- 从取到的32bits选择相应的位

12.Harzard 模块中，有哪几类冲突需要插入气泡?

- Load相关

13.Hazard 模块中采用默认不跳转的策略，遇到 branch 指令时，如何控制 flush 和 stall 信 号?

- 将流水线中的IF、ID和EX级的指令都清除掉（flush）

14.Harzard 模块中，RegReadE 信号有什么用?

- 判断EX级有没有读寄存器的操作，用来判断会不会与之前的写操作矛盾，用于转发

15.0 号寄存器值始终为 0，是否会对 forward 的处理产生影响?

- 有影响。需要判断是否为x0,如果是则不需要转发