Verilog 单周期 CPU 设计文档

设计者：忽骁

一 CPU 设计文档

1 数据通路设计

1）IFU

a.端口定义

```
module IFU(
 input clk,
 input reset,
 input [1:0] pcsrc,
 input [31:0] sign_ext,
 input [31:0] Rdata1,
 output [31:0] instr,
 output [31:0] pcvalue, //for grf print
 output [31:0] pcadd4
 );
```

b.主要信号说明

PC（程序计数器）端口说明

序号	信号名称	位宽	方向	功能描述
1	Clk	1	I	时钟信号，控制 PC 新值的读入
2	Reset	1	I	复位信号，有效时将 PC 内的值置为 0x00000000

3	Data	32	I	运算出的新 PC 值，时钟上升沿时写入 PC 寄存器中
4	Output	32	O	当前 PC 值，包含当前指令的地址信息

IM(指令存储器)端口说明

序	信	位	方	功能描述
1	Addr	32	I	地址信号，根据此信号从 IM 中读出数据作为当前指令
2	Data	32	O	从 IM 中取出的指令

2）ALU

a.端口定义

```
module ALU(
  input [31:0] A,
  input [31:0] B,
  input [3:0] operation,
  output reg [31:0] result,
  output reg equal
  );
```

b.主要信号说明

ALU（算术逻辑单元）端口说明

序号	信号名称	位宽	方向	功能描述
1	ALU operation	4	I	ALU 功能选择信号，根据此信号进行相应的运算
2	A	32	I	第一个运算数
3	B	32	I	第二个运算数
4	Result	32	O	运算结果
5	Equal	1	O	相等判定信号，如果 A=B，则该信号为 1，否则为 0

3）GRF

a.端口定义

```
module GRF(
    input [31:0] pcvalue,
    input clk,
    input reset,
    input regWrite,
    input [4:0] reg1,
    input [4:0] reg2,
    input [4:0] wreg,
    input [31:0] wdata,
    output [31:0] rdata1,
    output [31:0] rdata2
    );
```

b.主要信号说明

GRF（通用寄存器组）端口说明

序号	信号名称	位宽	方向	功能描述
1	RegWrite	1	I	寄存器组写入信号，该信号有效时，寄存器组进行写操作
2	Clk	1	I	时钟信号，控制寄存器组的写入
3	Reset	1	I	复位信号，该信号有效时，寄存器组内所有寄存器被置 0
4	Reg1	5	I	寄存器读取端地址 1，根据此信号从相应寄存器中读取数据到 Rdata1
5	Reg2	5	I	寄存器读取端地址 2，根据此信号从相应寄存器中读取数据到 Rdata2
6	Wreg	5	I	寄存器写入端地址，时钟上升沿到来且 RegWrite 有效时，根据此信号将 Wdata 中数据在写入相应寄存器中
7	Wdata	32	I	寄存器写入数据，当时钟上升沿到来且 RegWrite 有效时，将 Wdata 中数据在写入相应寄存器中

8	Rdata1	32	O	寄存器读出的数据 1
9	Rdata2	32	O	寄存器读出的数据 2

4）DM

a 端口定义

```
module DM(
    input clk,
    input reset,
    input memRead,
    input memWrite,
    input [9:0] addr,
    input [31:0] wdata,
    input [31:0] pcvalue,//for print
    input [31:0] realaddr,
    output [31:0] rdata
    );
```

b 主要信号说明

DM（数据存储器）端口说明

序号	信号名称	位宽	方向	功能描述
1	Clk	1	I	时钟信号，控制 DM 的数据写入
2	Reset	1	I	复位信号，该信号有效时将 DM 内所有数据置为 0x00000000
3	MemRead	1	I	读取信号，该信号有效时 DM 读数据
4	MemWrite	1	I	写入信号，该信号有效时 DM 写数据
5	Addr	5	I	地址信号，根据此信号在相应位置执行写入或读取操作
6	Wdata	32	I	写入数据，写入操作时将其写入相应位置
7	Rdata	32	O	读出数据，读取操作时读出相应位置的数据

5）EXT

a 端口定义

```
module sign_ext(
    input [15:0] in,
    output reg [31:0] out
    );
module unsign_ext(
    input [15:0] in,
    output reg [31:0] out
    );
```

b 主要信号说明

EXT 端口说明

序号	信号名称	位宽	方向	功能描述
1	Imm16	16	I	16 位立即数
2	extended	32	O	将 16 位立即数符号扩展或无符号扩展至 32 位后的结果

6）MUX

a 端口定义

```
module mux4(
```

```
        input [1:0] select,
        input [31:0] m0,
        input [31:0] m1,
        input [31:0] m2,
        input [31:0] m3,
        output reg [31:0] out
        );
module bit5mux4(
    input [1:0] select,
        input [4:0] m0,
        input [4:0] m1,
        input [4:0] m2,
        input [4:0] m3,
        output reg [4:0] out
        );
module mux2(
        input s0,
        input [31:0] m0,
        input [31:0] m1,
        output reg [31:0] out
        );
```

b 主要信号说明

EXT 端口说明

序号	信号名称	方向	功能描述
1	Select	I	选择信号
2	m0	I	第一个输入源
3	M1	I	第二个输入源
4	M2	I	第三个输入源
5	M3	I	第四个输入源
6	out	O	根据选择信号选择相应的输入源来输出

7）Controler

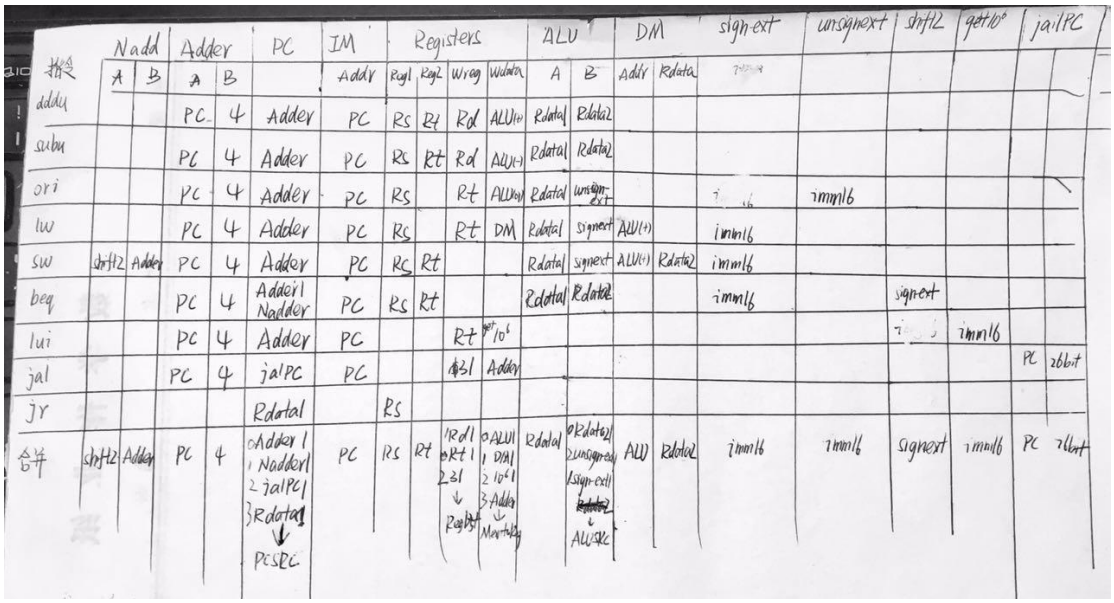
a 端口定义

```
module controler(
    input [5:0] op,
    input [5:0] func,
    input equal,
    output reg [1:0] RegDst,
    output reg [1:0] ALUSrc,
    output reg [1:0] MemtoReg,
    output reg [1:0] pcsrc,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg [3:0] operation
    );
```

b 主要信号说明

序号	信号名称	位宽	方向	功能描述
1	Op	6	I	指令的操作码字段（Op）
2	Func	6	I	指令的功能码字段（Func）
3	RegDst	2	O	寄存器写入端地址选择信号
4	ALUSrc	1	O	ALU 输入端 B 选择信号
5	MemtoReg	2	O	寄存器堆写入端数据选择信号
6	Pcsrc	2	O	Pc 输入源选择信号
7	RegWrite	1	O	寄存器堆写入信号
8	MemRead	1	O	DM 数据读取信号
9	MemWrite	1	O	DM 数据写入信号
10	ALUOperation	4	O	ALU 功能选择信号

8）数据通路总体



数据通路及其合并结果

2 控制器设计

1）Controller（控制器）端口说明

Controller（控制器）端口说明

序号	信号名称	位宽	方向	功能描述
1	Op	6	I	指令的操作码字段（Op）
2	Func	6	I	指令的功能码字段（Func）
3	RegDst	2	O	寄存器写入端地址选择信号
4	ALUSrc	1	O	ALU 输入端 B 选择信号
5	MemtoReg	2	O	寄存器堆写入端数据选择信号
6	PcSrc	2	O	Pc 选择信号信号
7	RegWrite	1	O	寄存器堆写入信号
8	MemRead	1	O	DM 数据读取信号
9	MemWrite	1	O	DM 数据写入信号
10	ALUOperation	4	O	ALU 功能选择信号

2）主控单元信号

控制信号	失效时作用	有效时作用
RegDst	寄存器堆写入端地址来选择 Rt 字段	寄存器堆写入端地址选择 Rd 字段
RegWrite	无	把数据写入寄存器堆中对应寄存器
MemRead	无	数据存储器 DM 读数据（输出）
MemWrite	无	数据存储器 DM 写数据（输入）

3）ALU 控制信号分析

ALU（算术逻辑单元）端口说明

序号	ALU operation	运算描述	具体运算
1	0000	加法运算	A+B
2	0001	减法运算	A-B
3	0010	或运算	A|B
4	0011	乘法运算	A*B

由数据通路我们可以得到：

①lw，sw 指令：ALU operation=0000，ALU 执行加法运算

②beq，subu 指令：ALU operation=0001，ALU 执行减法运算

③ori 指令：ALU operation=0010，ALU 执行或运算

4）主控单元信号分析

由此我们可以得到主控单元控制信号的分析如下：

①RegDst

R 型指令：RegDst=01，选择 Rd

ori,Lw,lui 指令： RegDst=00，选择 Rt

Jal 指令：RegDst=10，选择 Rdata1（$31 的值）

其他指令：不关心

②ALUSrc

R 型指令：ALUSrc=00，选择寄存器堆的 Read data2 输出

Beq 指令（减法运算）： ALUSrc=00，选择 Read data2 输出

Lw 指令，Sw 指令：ALUSrc=01，选择 Signext 的输出

Sw 指令： ALUSrc=01，选择 Signext 的输出

Ori 指令：ALUSrc=10，选择 unsignext 的输出

③MemtoReg

R 型指令，Ori 指令：MemtoReg=00，选择 ALU 输出

Lw 指令： MemtoReg=01，选择数据存储器 DM 输出

Lui 指令：MemtoReg=10，选择 get10^6 移位扩展器输出

Jal 指令：MemtoReg=11，选择 Adder 的输出（PC+4）

其他指令： 不关心 （不管选择哪个输出，RegWrite=0 就不会输出）

④Pcsrc 指令

Beq 指令：此时若 equal=1，Pcsrc=01，PC 输入选择加法器 Nadd 输出（分支指令目的地址），否则选择加法器 Add 输出（PC+4）

Jal 指令，pcsrc=10，PC 输入选择 jalPc 输出

Jr 指令，pcsrc=11，PC 输入选择 Rdata1（$31 的值）

其他指令：pcsrc=00，PC 输入选择加法器 Adder 输出（PC+4）

5）读写控制信号分析

①RegWrite

R 类指令，Ori、Lw、Lui，jal 指令：Regwrite=1，寄存器堆执行写操作

其他指令：RegWrite=0，寄存器堆不执行写操作

②MemRead

Lw 指令：MemRead=1，DM 执行读出操作

其他指令：MemRead=0，DM 不执行读出操作

③MemWrite

Sw 指令：MemWrite=1，DM 执行写入操作

其他指令，MemWrite=0，Dm 不执行写入操作

6）控制信号真值表

控制信号真值表

Func:	100001	100011	None				
Op:	00000	00000	001101	100011	101011	000100	001111
	addu	subu	ori	lw	sw	beq	lui
RegDst	**1**	**1**	**0**	**0**	**X**	**X**	**0**
RegWrite	**1**	**1**	**1**	**1**	**0**	**0**	**1**
ALUSrc<1:0>	**00**	**00**	**10**	**01**	**01**	**00**	**X**
Pcsrc<1:0>	**00**	**00**	**00**	**00**	**00**	**01**	**00**
MemRead	**0**	**0**	**0**	**1**	**0**	**0**	**0**
MemWrite	**0**	**0**	**0**	**0**	**1**	**0**	**0**
MemtoReg<1:0>	**00**	**00**	**00**	**01**	**XX**	**XX**	**10**
ALU operation<3:0>	**0000**	**0001**	**0010**	**0000**	**0000**	**0001**	**XXXX**

3 测试程序

1）a 测试程序

```
    .text
#test ori
    ori $a0,$0,456
    ori $a1,$a0,788
#test lui
    lui $a2,455
    lui $a3,0xffff #sign
    ori $a3,$a3,0xffff #set $a3 to -1
```

```
#twst addu

   addu $s0,$a0,$0

   addu $s1,$a0,$a1

   addu $s2,$0,$0

#twst subu

   subu $s3,$0,$0

   subu $s4,$0,$a0

   subu $s5,$a0,$0

   subu $s6,$a1,$a0

#test sw

   ori $t0,0x0000

   sw $a0,0($t0)

   sw $a1,4($t0)

   sw $a2,8($t0)

   sw $a3,12($t0)

   sw $s4,16($t0)

   sw $s5,20($t0)

#test lw
```

```
    lw $a1,0($t0)
    lw $a0,4($t0)
    lw $a3,8($t0)
    lw $a2,12($t0)
    lw $s5,16($t0)
    lw $s4,20($t0)
#test beq
    ori $a0,$0,1
    ori $a1,$0,2
    ori $a2,$0,1
    jal add1
    beq $a0,$a1,loop1
    beq $a0,$a2,loop2
loop1:
    sw $a0,24($t0)
add1:
    ori  $t0,1234
    jr $31
```

```
loop2:

   sw $a1,28($t1)
```

b 测试期望

如果 CPU 运行正确，则运行完所有指令后我们会得到以下输出

```
@00003000: $ 4 <= 000001c8

@00003004: $ 5 <= 000003dc

@00003008: $ 6 <= 01c70000

@0000300c: $ 7 <= ffff0000

@00003010: $ 7 <= ffffffff

@00003014: $16 <= 000001c8

@00003018: $17 <= 000005a4

@0000301c: $18 <= 00000000

@00003020: $19 <= 00000000

@00003024: $20 <= fffffe38

@00003028: $21 <= 000001c8

@0000302c: $22 <= 00000214

@00003030: $ 8 <= 00000000
```

```
@00003034: *00000000 <= 000001c8

@00003038: *00000004 <= 000003dc

@0000303c: *00000008 <= 01c70000

@00003040: *0000000c <= ffffffff

@00003044: *00000010 <= fffffe38

@00003048: *00000014 <= 000001c8

@0000304c: $ 5 <= 000001c8

@00003050: $ 4 <= 000003dc

@00003054: $ 7 <= 01c70000

@00003058: $ 6 <= ffffffff

@0000305c: $21 <= fffffe38

@00003060: $20 <= 000001c8

@00003064: $ 4 <= 00000001

@00003068: $ 5 <= 00000002

@0000306c: $ 6 <= 00000001

@00003070: $31 <= 00003074

@00003080: $ 8 <= 000004d2

@00003088: *0000001c <= 00000002
```

2） a 测试程序

```
.text

ori $4,$0,123

ori $5,$0,125

addu $6,$4,$5

subu $7,$5,$4

jal first

beq $4,$5,fake_end

beq $7,$11,end

first:

ori $11,$0,2

sw $6,2($7)

sw $7,1($4)

ori $10,$0,4

lw $8,0($10)

jr $31

fake_end: lui $9,1998

end:lui $10,1998
```

b 测试期望

```
@00003000: $ 4 <= 0000007b

@00003004: $ 5 <= 0000007d

@00003008: $ 6 <= 000000f8

@0000300c: $ 7 <= 00000002

@00003010: $31 <= 00003014

@0000301c: $11 <= 00000002

@00003020: *00000002 <= 000000f8

@00003024: *0000007b <= 00000002

@00003028: $10 <= 00000004

@0000302c: $ 8 <= 00000000

@00003038: $10 <= 07ce0000

@00003000: $ 4 <= 0000007b

@00003004: $ 5 <= 0000007d

@00003008: $ 6 <= 000000f8

@0000300c: $ 7 <= 00000002

@00003010: $31 <= 00003014

@0000301c: $11 <= 00000002
```

```
@00003020: *00000004 <= 000000f8

@00003024: *0000007c <= 00000002

@00003028: $10 <= 00000004

@0000302c: $ 8 <= 000000f8

@00003038: $10 <= 07ce0000
```

3）a 测试程序

```
.text

jal one

jal two

jal three

ori $9,321

beq $5,$4,fake_end

beq $3,$9,end

three:

lw $5,0($0) #0xffff0000

lw $4,4($0) #-1

lw $3,4($8) #321

lw $2,0($8) #123

lw $1,8($8) #454

jr $31

two:ori $8,8
```

```
sw $1,0($0)

sw $2,4($0)

sw $3,0($8)

sw $4,4($8)

sw $5,8($8)

jr $31

one:lui $1,0xffff #1 is 0xffff0000

ori $2,$1,0xffff #2 stores -1

ori $3,123

ori $4,321

addu $5,$3,$4

subu $6,$3,$4

subu $7,$4,$3

jr $31

fake_end:addu $10,$4,$3

end:ori $11,$0,1

subu $11,$4,$11
```

b 测试期望

```
@00003000: $31 <= 00003004

@0000304c: $ 1 <= ffff0000

@00003050: $ 2 <= ffffffff

@00003054: $ 3 <= 0000007b
```

```
@00003058: $ 4 <= 00000141

@0000305c: $ 5 <= 000001bc

@00003060: $ 6 <= ffffff3a

@00003064: $ 7 <= 000000c6

@00003004: $31 <= 00003008

@00003030: $ 8 <= 00000008

@00003034: *00000000 <= ffff0000

@00003038: *00000004 <= ffffffff

@0000303c: *00000008 <= 0000007b

@00003040: *0000000c <= 00000141

@00003044: *00000010 <= 000001bc

@00003008: $31 <= 0000300c

@00003018: $ 5 <= ffff0000

@0000301c: $ 4 <= ffffffff

@00003020: $ 3 <= 00000141

@00003024: $ 2 <= 0000007b

@00003028: $ 1 <= 000001bc

@0000300c: $ 9 <= 00000141

@00003070: $11 <= 00000001

@00003074: $11 <= fffffffe
```

二 思考题

1）数据通路设计部分

1 因为 DM 的内存为 4KB，即 2 的 12 次方字节，因此需要 12 位的地址输入，而一个字又是 4 个字节，因此需要 10 位的地址来选择相应的字，因此地址输入选择的是[11:2]，相当于原地址除以 4，Addr 的来源是 ALU 的 result 输出。

2 需要对 PC，DM，GRF 清零，因为它们能够记录状态，将其清零后能够恢复到初始状态，从而进行新一次的运行。

2）控制器设计部分

1 编码方式如下：

a.If-else:

```
if(op==6'b001101)  //ori
        begin
         RegDst<=2'b00;
         ALUSrc<=2;
         MemtoReg<=2'b00;
         RegWrite<=1;
         operation<=2;
         end
```

b.assign:

```
assign AluSrc=(op==ori)||(op==lw)||(op==sw)
```

c.`define:

```
`define ori_control=15'b001000010010000

Case(op)

   001101:signal=ori_control


  Assign{RegDst,ALUSrc,MemtoReg,RegWrite,operation
,pcrc,MemWrite}=signal
```

2

优缺点：

a.逻辑清晰，比较容易理清思路，调试比较容易，但是代码比较长

b.代码比较简洁，但是需要时间理清 op 字段，func 字段与各路控制信号的关系，发生错误不容易看出

c 代码很精练，但是需要花时间写出每个指令的控制信号并将其合并，而合并到一起后，一旦控制信号多了，就很难找出错误的控制信号，不容易调试，且编码时容易发生错误

3）在线测试相关信息部分

1 因为 add 指令是通过将 GPR[Rs]和 GPR[Rt]的最高位拼接到原值的高位后将两者相加得到 temp，然后通过判断 temp 的第 32 位和第 31 位是否相等来判断是否溢出，不溢出则输出计算结果 temp[31:0]，而 addu 指令则直接输出计算结果。如果不考虑溢出，则判断语句失效，因此 add 指令与 addu 指令等效，addi 与 addiu 同理。

2 优点：一个时钟周期内执行只一条指令，逻辑比较简单，容易设计

缺点：在任何时间都只有一个部件在工作，所以处理器性能不高。并且时钟周期由时间最长的那条指令决定，造成了该处理器比较浪费时间，速度比较慢

3 递归调用子程序的时候如果用到了 jal 指令，则需要将当前$ra 的值压入堆栈，否则第二次调用 jal 指令时候，原$ra 的值就会被新的$ra 覆盖，从而无法正确运行递归程序。