Verilog 单周期处理器设计

一、 CPU 设计方案综述

（一） 总体设计概述

本 CPU 为 Verilog 实现的流水线 MIPS-CPU，支持的指令集包括{addu、subu、ori、lw、sw、beq、lui、nop、j}。为了实现这些功能 CPU 主要包括这些模块：GRF(寄存器堆，32 位*32)、ALU(算术逻辑运算单元)、IFU(取指单元)、DM(数据存储器，32 位*1024 字)、Ext(位扩展器，16->32)、Controller(指令译码器)等，这些模块按照自底向上的设计逐层展开。

（二） 关键模块定义

1. IFU(取指单元)。主要由 PC(程序计数器，5 位)，IM(指令存储器，32 位*32 字)两个子模块构成。该模块在主模块(main)中实现，没有单独建立子电路。

表 1 IFU 端口表

端口	方向	描述
reset	Input	异步复位信号。为 1 时指令地址保持 0。
Inst	Ouput	取出的指令

2. ALU(算术逻辑单元，4 位选择 16 种运算)

表 2 ALU 端口表

端口	方向	描述
op1[31:0]	Input	操作数 1
op2[31:0]	Input	操作数 2
ALUCtrl[3:0]	Input	功能选择
result[31:0]	Output	计算结果
Zero	Output	计算结果是否为 0

表 3 ALU 功能表

功能码	描述	功能码	描述
0000	非负	1000	逻辑右移
0001	负数	1001	算术右移
0010	加法	1010	算术/逻辑左移
0011	减法	1011	判断相等
0100	按位与	1100	有符号小于判断
0101	按位或	1101	无符号小于判断
0110	按位异或	1110	大于 0 判断
0111	按位或非	1111	小于等于 0 判断

3. GRF(寄存器堆，32 位*32)。其中 0 号寄存器始终为 0。

表 4 GRF 端口表

端口	方向	描述
clk	Input	时钟信号
reset	Input	复位信号，将 32 个寄存器中的值全部清零
we	Input	写使能信号，高电平为写有效
RAddr1	Input	5 位地址输入信号，指定 32 个寄存器中的一个，将其中的数据读出到 RData1
RAddr2	Input	5 位地址输入信号，指定 32 个寄存器中的一个，将其中的数据读出到 RData2
WAddr	Input	5 位地址输入信号，指定 32 个寄存器中的一个作为写入的目标寄存器
WData	Input	32 位数据输入信号
RData1	Output	输出 RAddr1 指定的寄存器中的 32 位寄存器
RData2	Output	输出 RAddr2 指定的寄存器中的 32 位寄存器

4. DM(数据存储器，32 位*1024 字)。用内置 RAM 实现，采用 Separate load and store ports 属性。地址 10 位，题目中要求输出后 5 位，取后 5 位。RAM 自带时钟信号、写使能、地址端和数据端，与 DM 要求完全相同。
5. Ext(位扩展器，16->32 位，4 选择)

表 5 EXT 端口表

端口	方向	描述
Imm[15:0]	Input	待扩展的 16 位数
Extop[1:0]	Input	扩展方法选择： 00：符号扩展 01：无符号扩展 10：加载至高 16 位 11：符号扩展之后，左移两位
Out[31:0]	Output	扩展后的 32 位数

6. Controller(控制器)

表 6 Controller 端口表

端口	方向	描述
Opcode[5:0]	Input	指令操作码
Func[5:0]	Input	指令功能码
Jump	Output	跳转信号
ExtOp[1:0]	Output	位扩展方式，见表 3
MemtoReg	Output	读内存信号
MemWrite	Output	内存写使能信号
Branch	Output	分支信号
ALUCrtl[3:0]	Output	ALU 控制信号，见 ALU 模块
ALUSrc	Output	ALU 操作数 2 的来源 0：寄存器 1：立即数
RegDst	Output	寄存器写地址选择 0：Instr[20:16] 1：Instr[15:11]
RegWrite	Output	寄存器写使能信号

二、 指令分析

1、addu 指令

当功能码为 100001 时，表示 addu 指令，加法运算。

指令用法为：addu rd, rs, rt。

指令作用为：rd <- rs + rt, 将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行加法运算，结果保存到地址为 rd 的通用寄存器中。与 add 指令的不同之处在于 addu 指令不进行溢出检查，总是保存到目的寄存器。

指令码：`000000 rs[25:21] rt[20:16] rd[15:11] 100001`

控制信号：

表 7　addu 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	xx	0	0	0	0010	0	1	1

2、subu 指令

指令用法为：subu rd, rs, rt。

指令作用为：rd <- rs – rt, 将地址为 rs 的通用寄存器的值用地址为 rt 的通用寄存器的值进行减法运算，结果保存到地址为 rd 的通用寄存器中。与 sub 指令的不同之处在于：subu 指令不进行溢出检查，总是将结果保存到目的寄存器。

指令码：`000000 rs[25:21] rs[20:16] rd[15:11] 100011`

控制信号：

表 8 subu 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	xx	0	0	0	0011	0	1	1

3、ori 指令

这是一个 I 类型的指令，ori 指令的指令码是 001101，当处理器发现当前处理的指令的高 6bit 是 001101 时，就知道当前正在处理的是 ori 指令。

指令用法为：ori rs, rt, immediate，作用是将指令中的 16 位立即数 immediate 进行无符号扩展至 32 位，然后与索引为 rs 的通用寄存器的值进行逻辑”或”运算，运算结果保存到索引为 rt 的通用寄存器中。

指令码：`001101 rs[25:21] rt[20:16] immediate[15:0]`

控制信号：　　表 9 ori 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	01	0	0	0	0101	1	0	1

4、lw 指令

指令用法为：lw rt, offset(base)；

指令作用为：从内存中指定的加载地址处，读取一个字，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低两位为 00。

指令码：`100011 base[25:21] rt[20:16] offset[15:0]`

控制信号：　　表 10　lw 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	00	1	0	0	0010	1	0	1

5、sw 指令

这是一个 I 类型的指令，sw 指令的指令码是 101011，当处理器发现当前处理的指令的高 6bit 是 101011 时，就知道当前正在处理的是 sw 指令。

指令用法为：sw rt, offset(base);

指令作用为：将地址为 rt 的通用寄存器的值存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低两位为 00.。

指令码：`101011 base[25:21] rt[20:16] offset[15:0];`

控制信号：　　表 11 sw 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	00	0	1	0	0010	1	x	0

6、beq 指令

指令用法为：beq rs, rt, offset

指令作用为：if rs == rt then branch, 将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行比较，如果相等，那么发生跳转。如果发生转移，那么将 offset 左移 2 位，并符号扩展至 32 位，然后与分支指令后面指令地址相加，加法结果就是转移的目标地址，从该地址取指令。

转移目标地址 = (signed_extend)(offset || 00) + (pc + 4);

指令码：`000100 rs[25:21] rt[20:16] offset[15:0]`

控制信号：　　表 12 beq 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	11	0	0	1	0011（或 0110）	0	x	0

7、lui 指令

指令用法为：lui rt, immediate;

指令作用为：rt <- immediate || 0x0000，将指令中的 16bit 立即数保存到地址为 rt 的通用寄存器的高 16 位。另外，地址为 rt 的通用寄存器的低 16 位使用 0 填充。

指令码：`001111 00000 rt[20:16] immediate[15:0];`

在 `lui` 指令实现过程中，采用将 `lui` 指令转化为 `ori` 指令来执行，语句如下：

`lui rt, immediate = ori rt, $0, (immediate || 0x0000);` 也就是将指令中的立即数左移 `16bit`，然后与 `0` 寄存器进行逻辑”或”运算。

控制信号：　　表 13 lui 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	10	0	0	0	0010	0	0	1

8、nop 空指令

指令码为 000000。

指令格式为：000000 00000[25:21] 00000[20:16] 00000[15:11] 000000，nop 指令的二进制码与 sll $0, 0, 0 的二进制码一样，nop 指令不用特意实现，完全可以当作特殊的逻辑左移指令 sll，不执行任何操作，所有控制信号均为 0。

9、 j 指令

指令用法为：j target;

指令作用为：pc <- (pc + 4)[31 : 28] || target || 00，转移到新的指令地址，其中新指令地址的低 28 位是指令中的 target 左移两位的值，新指令地址的高四位是跳转指令后面指令的地址高 4 位。

指令码：000010 target[25:0];

控制信号：

表 14　j 指令控制信号

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
1	00	x	0	x	xxxx	x	x	0

三、 数据通路

单周期 CPU 执行指令主要过程：

① 根据程序计数器(PC)从内存中取指令，PC 的值为该指令在内存中存放的地址。

② 计算 PC 的值，能自动计算 PC 的值以确定下一条指令的地址。

③ 对指令操作码进行解析，产生控制信号来控制指令进行相应的操作。

④ 取操作数，根据指令字段的内容选择从存储器读取数据或直接从寄存器取数。

⑤ 算术/逻辑运算，根据译码结果进行算术/逻辑运算或者计算操作数的地址。

⑥ 根据指令的要求对运算后的结果进行写回操作，如写存储器或写寄存器。

PC：输入端接 next 信号，从存储器中取出 32 位指令，输出端接指令存储器，将 PC 递增，PC = PC + 4；

IM：输出端接指令译码器。其中 31:26 接 Opcode，5:0 接 funct。15:0 接立即数扩展 ext。25:21，20:16，15:11 接寄存器文件地址。

DM：写数据端接寄存器文件输出，读数据端接寄存器文件写数据。写地址端接 ALU 结果。MemWrite 接写使能端。

ALU：操作数接寄存器和立即数，输出端接寄存器写数据和内存写地址。还有一个零端，与 Branch 信号接在与门上，当二者均为真时分支有效。

next 信号（下一个 PC 值）：当 j 为真时，为立即数。否则当 branch 和 zero 均为真时，为 branch 计算所得地址；二者有一个为假时，为 pc + 4（在 Logisim 中不允许跨地址存放数据，因此 32 位算作一个地址，地址应该加 1，但是 PC 为保持和 Mars 一致，设计成字对齐，即 4 的倍数，因此应该加 4。取地址时右移两位（或直接取[6:2]）。

四、 控制器设计

控制器的功能是识别指令，并对其它模块发出信号。控制器通常要读取 Opcode(指令码)、function(功能码)部分，有时也需要 rt,rs 等部分，当 Opcode(和 function 等部分)的组合符合某一条指令时，控制器识别这条指令，并发出各个模块执行这条指令所需要的信号。

下面给出 9 条指令对应的 Opcode(指令码)以及 function(功能码)：

表 15 指令对应的 Opcode 及 function

指令	Opcode	function	指令	Opcode
addu	000000	100001	sw	101011
subu	000000	100011	beq	000100
ori	001101	X	lui	001111
lw	100011	X	j	000010

然后，把通过指令与 Opcode 及 function 的对应关系，建立与或逻辑电路，例如：

```
sw = Opcode[5] & ~Opcode[4] & Opcode[3] & ~Opcode[2] & Opcode[1] & Opcode[0];
```

然后将第二部分中指令的控制信号表进行合并，得到每一个控制信号的逻辑表达式：

表 16 控制信号表

指令	Opcode	Funct	Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
addu	000000	100001	0	xx	0	0	0	0010	0	1	1
subu	000000	100011	0	xx	0	0	0	0011	0	1	1
ori	001101	xxxxxx	0	01	0	0	0	0101	1	0	1
lw	100011		0	00	1	0	0	0010	1	0	1
sw	101011		0	00	0	1	0	0010	1	x	0
beq	000100		0	11	0	0	1	0011	0	x	0
lui	001111		0	10	0	0	0	0010	0	0	1
j	000010		1	00	x	0	x	xxxx	x	x	0

```
jump=j
ExtOp[0]=ori||beq ExtOp[1]=lui||beq
MemtoReg=lw
MemWrite=sw
Branch=beq
ALUSrc=ori||lw||sw
RegDst=addu||subu
RegWrite=addu||subu||ori||lw||lui
ALUCtrl[3]=0
ALUCtrl[2]=ori
ALUCtrl[1]=addu||subu||lw||sw||beq||lui
ALUCtrl[0]=subu||ori||beq
```

根据这些便可建立对应的与或逻辑电路。

五、 测试方案

（一） 典型测试样例

1. addu、subu 指令测试

MIPS 汇编指令	机器码
ori $1, $0, 0x1234	34011234
ori $2, $0, 0x5678	34025678
ori $3, $0, 21	34030015
ori $4, $0, 0xffff	3404ffff
ori $31, $0, 5	341f0005
ori $30, $0, 1	341e0001
lui $5, 0xaffc	3c05affc
lui $6, 0xaaac	3c06aaac
addu $7, $5, $6	00a63821
addu $8, $5, $1	00a14021
addu $9, $6, $2	00c24821
addu $10, $7, $3	00e35021
addu $11, $3, $4	00645821
subu $12, $7, $3	00e36023
subu $13, $2, $3	00436823
subu $14, $3, $31	007f7023
subu $15, $1, $31	003f7823
addu $16, $15, $31	01ff8021

Verilog 运行结果：

```
@00003000: $ 1 <= 00001234
@00003004: $ 2 <= 00005678
@00003008: $ 3 <= 00000015
@0000300c: $ 4 <= 0000ffff
@00003010: $31 <= 00000005
@00003014: $30 <= 00000001
@00003018: $ 5 <= affc0000
@0000301c: $ 6 <= aaac0000
@00003020: $ 7 <= 5aa80000
@00003024: $ 8 <= affc1234
@00003028: $ 9 <= aaac5678
@0000302c: $10 <= 5aa80015
@00003030: $11 <= 00010014
@00003034: $12 <= 5aa7ffeb
@00003038: $13 <= 00005663
@0000303c: $14 <= 00000010
@00003040: $15 <= 0000122f
@00003044: $16 <= 00001234
```

2. ori、lui 指令测试

MIPS 汇编指令	机器码
ori $17, $0, 0	34110000
ori $18, $0, 256	34120100
ori $19, $18, 768	36530300
ori $17, $19, 0xffff	3671ffff
ori $28, $0, 1	341c0001
ori $20, $28, 5	37940005
ori $31, $28, 6	379f0006
ori $14, $20, 512	368e0200
ori $28, $14, 0xaffc	35dcaffc
ori $15, $30, 0xcbac	37cfcbac
lui $6, 1	3c060001
lui $7, 0xffff	3c07ffff
lui $8, 0xaffc	3c08affc
lui $9, 256	3c090100
lui $10, 0x1abc	3c0a1abc
lui $11, 0xbbca	3c0bbbca

Verilog 运行结果：

```
@00003000: $17 <= 00000000
@00003004: $18 <= 00000100
@00003008: $19 <= 00000300
@0000300c: $17 <= 0000ffff
@00003010: $28 <= 00000001
@00003014: $20 <= 00000005
@00003018: $31 <= 00000007
@0000301c: $14 <= 00000205
@00003020: $28 <= 0000affd
@00003024: $15 <= 0000cbac
@00003028: $ 6 <= 00010000
@0000302c: $ 7 <= ffff0000
```

```
@00003030: $ 8 <= affc0000
@00003034: $ 9 <= 01000000
@00003038: $10 <= 1abc0000
@0000303c: $11 <= bbca0000
```

3. lw、sw 指令测试

MIPS 汇编指令	机器码
ori $1, $0, 12	3401000c
ori $2, $0, 16	34020010
ori $3, $0, 24	34030018
ori $4, $0, 0x0100	34040100
ori $5, $0, 0x0104	34050104
ori $6, $0, 0x0108	34060108
ori $17, $0, 0x1234	34111234
ori $18, $0, 0x5678	34125678
ori $19, $0, 0xbbbb	3413bbbb
ori $20, $0, 0xaffc	3414affc
lui $21, 0x4321	3c154321
lui $22, 0x9876	3c169876
lui $23, 0xcbbc	3c17cbbc
lui $24, 0xddab	3c18ddab
sw $17, 0($1)	ac310000
sw $18, 4($3)	ac720004
sw $19, 0($2)	ac530000
sw $20, 8($3)	ac740008
sw $21, 0($1)	ac350000
sw $22, 12($3)	ac76000c
sw $23, 0($2)	ac570000
sw $24, 16($3)	ac780010
lw $9, 0($1)	8c290000
lw $10, 0($2)	8c4a0000
lw $11, 0($3)	8c6b0000
lw $12, 4($1)	8c2c0004
lw $13, 8($3)	8c6d0008
lw $14, 12($3)	8c6e000c
lw $15, 4($2)	8c4f0004
lw $16, 16($3)	8c700010
sw $17, 0($4)	ac910000
sw $18, 4($6)	acd20004
sw $19, 0($5)	acb30000
sw $20, 8($6)	acd40008
sw $21, 0($4)	ac950000
sw $22, 12($6)	acd6000c
sw $23, 0($5)	acb70000
sw $24, 16($6)	acd80010
lw $9, 0($4)	8c890000
lw $10, 0($5)	8caa0000
lw $11, 0($6)	8ccb0000
lw $12, 4($4)	8c8c0004
lw $13, 8($6)	8ccd0008
lw $14, 12($6)	8cce000c
lw $15, 4($5)	8caf0004
lw $16, 16($6)	8cd00010

Verilog 运行结果：

```
@00003000: $ 1 <= 0000000c
@00003004: $ 2 <= 00000010
@00003008: $ 3 <= 00000018
@0000300c: $ 4 <= 00000100
@00003010: $ 5 <= 00000104
@00003014: $ 6 <= 00000108
```

```
@00003018: $17 <= 00001234
@0000301c: $18 <= 00005678
@00003020: $19 <= 0000bbbb
@00003024: $20 <= 0000affc
@00003028: $21 <= 43210000
@0000302c: $22 <= 98760000
@00003030: $23 <= cbbc0000
@00003034: $24 <= ddab0000
@00003038: *0000000c <= 00001234
@0000303c: *0000001c <= 00005678
@00003040: *00000010 <= 0000bbbb
@00003044: *00000020 <= 0000affc
@00003048: *0000000c <= 43210000
@0000304c: *00000024 <= 98760000
@00003050: *00000010 <= cbbc0000
@00003054: *00000028 <= ddab0000
@00003058: $ 9 <= 43210000
@0000305c: $10 <= cbbc0000
@00003060: $11 <= 00000000
@00003064: $12 <= cbbc0000
@00003068: $13 <= 0000affc
@0000306c: $14 <= 98760000
@00003070: $15 <= 00000000
@00003074: $16 <= ddab0000
@00003078: *00000100 <= 00001234
@0000307c: *0000010c <= 00005678
@00003080: *00000104 <= 0000bbbb
@00003084: *00000110 <= 0000affc
@00003088: *00000100 <= 43210000
@0000308c: *00000114 <= 98760000
@00003090: *00000104 <= cbbc0000
@00003094: *00000118 <= ddab0000
@00003098: $ 9 <= 43210000
@0000309c: $10 <= cbbc0000
@000030a0: $11 <= 00000000
@000030a4: $12 <= cbbc0000
@000030a8: $13 <= 0000affc
@000030ac: $14 <= 98760000
@000030b0: $15 <= 00000000
@000030b4: $16 <= ddab0000
```

4. beq、nop 指令测试

MIPS 汇编指令	机器码
ori $1, $0, 0x1111	34011111
ori $2, $0, 0xaffc	3402affc
ori $31, $0, 4	341f0004
lui $3, 0x1234	3c031234

lui $4, 0xaffc	3c04affc
addu $8, $1, $2	00224021
nop	00000000
addu $10, $2, $1	00415021
beq $3, $4, label1	10640004
subu $20, $8, $1	0101a023
sw $20, 0($31)	aff40000
nop	00000000
beq $8, $10, label2	110a0001
label1:	00235821
addu $11, $1, $3	8c1e0004
label2:	3419affc
lw $30, 4($0)	13220002
ori $25, $0, 0xaffc	00000000
beq $25, $2, label3	033fc021
nop	3c171234
addu $24, $25, $31	12e20002
label3:	00000000
lui $23, 0x1234	3c101123
beq $23, $2, label4	00000000
nop	3c1b1000
lui $16, 0x1123	
label4:	
nop	
lui $27, 0x1000	

Verilog 运行结果：

```
@00003000: $ 1 <= 00001111
@00003004: $ 2 <= 0000affc
@00003008: $31 <= 00000004
@0000300c: $ 3 <= 12340000
@00003010: $ 4 <= affc0000
@00003014: $ 8 <= 0000c10d
@0000301c: $10 <= 0000c10d
@00003024: $20 <= 0000affc
@00003028: *00000004 <= 0000affc
@00003038: $30 <= 0000affc
@0000303c: $25 <= 0000affc
@0000304c: $23 <= 12340000
@00003058: $16 <= 11230000
@00003060: $27 <= 10000000
```

5. jal、jr 指令测试

MIPS 汇编指令	机器码
ori $1, $0, 0x3048	34013048
ori $2, $0, 256	34020100
ori $3, $0, 0xffff	3403ffff
ori $4, $0, 1	34040001
addu $8, $2, $4	00444021
addu $10, $8, $8	01085021
subu $11, $10, $2	01425823
addu $12, $8, $4	01046021
jal label	0c000c0d
beq $11, $12, label2	116c0001
ori $21, $0, 145	34150091
label2:	3c191853
lui $25, 0x1853	00200008
jr $1	340f007b
label:	3c101234
ori $15, $0, 123	03e00008

lui $16, 0x1234	36315678
jr $ra	02119821
ori $17, 0x5678	0324d023
addu $19, $16, $17	0044d823
subu $26, $25, $4	0084e021
subu $27, $2, $4	
addu $28, $4, $4	

Verilog 运行结果：

```
@00003000: $ 1 <= 00003048
@00003004: $ 2 <= 00000100
@00003008: $ 3 <= 0000ffff
@0000300c: $ 4 <= 00000001
@00003010: $ 8 <= 00000101
@00003014: $10 <= 00000202
@00003018: $11 <= 00000102
@0000301c: $12 <= 00000102
@00003020: $31 <= 00003024
@00003034: $15 <= 0000007b
@00003038: $16 <= 12340000
@0000302c: $25 <= 18530000
@00003048: $26 <= 1852ffff
@0000304c: $27 <= 000000ff
@00003050: $28 <= 00000002
```

六、 思考题

（一）根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是[11:2]而不是[9:0]？这个 addr 信号又是从哪里来的？

答：在指令中，地址的偏移单位是 4，即每个单位存储一个字节，而 dm 的偏移单位是 1，即每个单位存储一个字，因此指令地址偏移是 dm 中地址的四倍，与此同时，在二进制表示中，四倍相当于左移两位，即 dm 中地址选取[11:2]位。

（二）思考 Verilog 语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

1. 使用 if-else 语句加 case 语句（本 CPU 所用方法）

```
if(cmd == 0) begin
   control_signals = 0; // nop操作
end
else begin
   case(cmd[31 : 26])  // 根据Opcode段进行判断指令
   0 : case( cmd [5:0])  // Opcode为0时，根据功能码进行判断
            0 : begin
                control_signals = 17'b00_1_01_10_00_00_0_01010;     //sl
            end
            2 : begin
                control_signals = 17'b00_1_01_10_00_00_0_01000; //srl
            end
            3 : begin
                control_signals = 17'b00_1_01_10_00_00_0_01001; //sra
            end
            4 : begin
                control_signals = 17'b00_1_01_00_00_00_0_01010;    //sl
            end
            6 : begin
                control_signals = 17'b00_1_01_00_00_00_0_01000; //srlv
            end
            7 : begin
                control_signals = 17'b00_1_01_00_00_00_0_01001; //srav
            end
            8 : begin
                control_signals = 17'b00_0_00_00_00_00_1_00000; //jr
            end
```

2. 使用宏定义加三目运算符加或运算（参考自己动手写 **CPU** 一书）

```
`define EXE_J      6'b000010
`define EXE_JAL    6'b000011
`define EXE_JALR   6'b001001
`define EXE_JR     6'b001000
`define EXE_BEQ    6'b000100
`define EXE_BGEZ   5'b00001
`define EXE_BGEZAL 5'b10001
`define EXE_BGTZ   6'b000111
`define EXE_BLEZ   6'b000110
`define EXE_BLTZ   5'b00000
`define EXE_BLTZAL 5'b10000
`define EXE_BNE    6'b000101
```

```
......
`EXE_JR: begin                    // jr指令
  wreg_o                    <= `WriteDisable;
  aluop_o                   <= `EXE_JR_OP;
  alusel_o                  <= `EXE_RES_JUMP_BRANCH;
  reg1_read_o               <= 1'b1;
  reg2_read_o               <= 1'b0;
  link_addr_o               <= `ZeroWord;
  branch_target_address_o   <= reg1_o;
  branch_flag_o             <= `Branch;
  next_inst_in_delayslot_o  <= `InDelaySlot;
  instvalid                 <= `InstValid;
end
```

比较：我个人喜欢使用 case 语句，可以大大简化代码，并且结构清晰易读，可理解性非常好，并且便于调试，在出现错误时，更容易及时找出问题。

（三）题目：在相应的部件中，reset 的优先级比其他控制信号（不包括 clk 信号）都要高，且相应的设计都是同步复位。清零信号 reset 所驱动的部件具有什么共同特点？

答：reset 驱动的部件本质上都是记忆性部件（logisim 里对应的 Memory Library），且都支持写入和读取，reset 信号为 1 时，在时钟上升沿部件内所有数据返回到初始值（对于 PC 来说是 0x00003000，对于其余部件来说是 0）。

（四）题目：C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理，这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。

答：从相关指令的具体操作来看，add 和 addi 判断溢出的具体方法是在 32 位数前再接一位符号位并将其与加法执行后的第 31 位比较（从 0 位计），若不溢出，计算结果仍取 0-31 位。我的理解是判断溢出的功能只是增添了一位用于判断的数，而并没

有对原数据计算结果的 0-31 造成影响，因此在不考虑溢出即程序员可以保证程序不会有溢出时 add 和 addi 均可正常产生结果且结果分别与 addu 和 addiu 相同。即可以认为在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。

（五）题目：根据自己的设计说明单周期处理器的优缺点。

答：优点：设计简单，各模块内聚程度高，模块间耦合程度小。大部分模块内部是简单的组合逻辑，只需要建立好逻辑清晰的数据通路，进而完成各模块内部的逻辑即可保证较低的错误率，增添指令时只需考虑各模块内部真值表的补充，实现简单。缺点：时钟频率受各部分组合逻辑的制约，为保证在一个时钟周期内完成任一条指令的执行，需设置时钟周期大于等于执行时间最长的一条指令的延迟时间（本设计中该指令是 lw），这将导致在执行其他指令时大部分时间被浪费，CPU 速度较慢，性能不高。