综合性课程设计 1：MIPS 汇编器设计

要求：

用算法语言（C 或 C++）设计 MIPS 汇编与反汇编器工具，支持 MIPS 指令子集（附件）。支持不小于 10K 指令的汇编容量。最终提交必须把所有动态链接编译进去(静态连接)，可独立执行程序和源代码及工程。

基本功能：

汇编：

1. 支持附录 A.10（第 3 版）或附录子集所有整数指令和伪指令。生成 2 进制目标文件.bin 和.coe 格式文件；

2. 汇编指令以“;”为结束符，”//”和“#”为注释符；

3. 寄存器符号支持 R0~R31(大小写均可)和 MIPS 使用约定(第 3 版图 2-18)

反汇编：支持附录 A.10（第 3 版）所有整数指令或附录子集。支持.bin 和.coe 文件加载。

文件操作功能：能加载保存 ASM、bin 和. coe 文件

File:	新建(N)	Ctrl+N
	打开(O)	Ctrl+O→（3 个子项：ASM、bin 和 coe）
	保存(S)	Ctrl+S
	保存为(A)	Ctrl+Shift+S→（3 个子项：ASM、bin 和 coe）
	打印(P)	Ctrl+P（选做）
	Exit(X)	

以上为必做内容

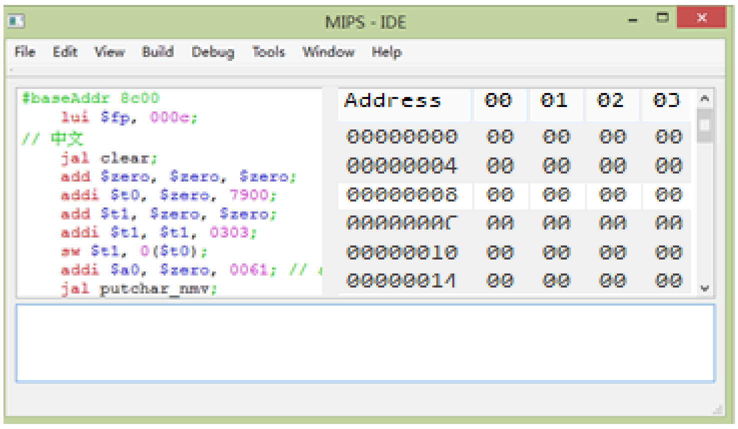
扩展功能（选做）： **编辑：**可以编辑文本和 2、16 进制

模拟器功能

调试（Debug）：支持单步执行、单步(跳过,不进入函数)、运行到。内存 1MB，支持运行结果显示格式如下：

```
寄存器：IP=12345678;
R0($zero) = 00000000;  R1($at)   =12345678;      ……     R7（$a3）  =12345678
R8（$t0）  =00000000; R9（$t1）=12345678;        ……     R15（$t7）=12345678
R16($s0)   =00000000;  R17($s0)=12345678;        ……     R23（$7）  =12345678
R24($t8)   =00000000;  R25($t9)=12345678;        ……     R31($ra)=12345678
内存
00000000: 12345678 12345678 12345678 12345678 12345678 12345678 12345678 12345678
00001000: 12345678 12345678 12345678 12345678 12345678 12345678 12345678 12345678
00002000: 12345678 12345678 12345678 12345678 12345678 12345678 12345678 12345678
……
```

窗口参考格式：



```
#baseAddr 8c00
    lui $fp, 000c;
// 中文
    jal clear;
    add $zero, $zero, $zero;
    addi $t0, $zero, 7900;
    add $t1, $zero, $zero;
    addi $t1, $t1, 0303;
    sw $t1, 0($t0);
    addi $a0, $zero, 0061; //
    jal putchar_nmv;
```

```
Address    00  01  02  03
00000000   00  00  00  00
00000004   00  00  00  00
00000008   00  00  00  00
0000000C   00  00  00  00
00000010   00  00  00  00
00000014   00  00  00  00
```

菜单

File：	新建(N)	Ctrl+N
	打开(O)	Ctrl+O
	保存(S)	Ctrl+S
	保存为(A)	Ctrl+Shift+S
	打印(P)	Ctrl+P
	Exit(X)	
Edit：	撤消(U)	Ctrl+Z
	剪切(T)	Ctrl+X
	复制(C)	Ctrl+C
	粘贴(P)	Ctrl+V
	删除(L)	Del
	查找(F)	Ctrl+F
	替换(R)...	Ctrl+H
	转到(G)	Ctrl+G
	全选(A)	Ctrl+A
Bulid：	汇编(Asm)	
	Coe	
	反汇编	
Debug：	单步(Step)	
	单步（跳过 Jal）	
	运行到…	
	停止	

汇编器的一些补充说明(参考后面的例子)：

汇编器语法参考附录 A.10.2，并增加下列伪指令：

1. 指定代码起始地址：BaseAddre

   BaseAddre: 00008c00;　　　　　　//用”,”分隔，”;”结束。可以多个定义，但后继地址

   ……;　　　　　　　　　　　　//必须大于前面定义的地址。中间不确定区域填“0”

2. 指定数据起始地址：DataAddre

DataAddre: 00008d00;　　　　//用”,”分隔，”;”结束。可以多个定义，但后继地址

DB:　0x12，0x34，……;　　　　//必须大于前面定义的地址。中间不确定区域填“0”

DW:　0x1234，0x4567，……;

二个地址申明之间的空余处填“00000000”，后继申明地址必须大于“前次申明地址+已使用空间”，否则结出错误指示。代码空间超出范围也结出错误指示

3. 初始化数据伪指令

关键字后定义指定，每个数据由逗号分隔：

DB：　定义字节数据	例：	DB	0x12，0x34，……;
DW：定义 16 位字数据	例：	DW	0x1234，0x4567，……;
DD：　定义 32 位字数据		DD	0x12345678，0x87654321，……;

参数定义：

EQU：定义常量	例：	X	EQU 12;　　//定义 X 为常数 12

4. 所有标号包括数据区基地址都支持伪指令：la rdest, label_address

以上为必须

声明未初始化的数据（定义数据空间）：

RESB：　定义字节空间

L1:　RESB 16;　　　　//定义 16 个字节空间，L1 是标号

RESW：定义 16 位字空间

L2:　RESW 16;　　　　//定义 16 个 16 位字空间，L2 是标号

RESD：　定义 32 位字节空间

L3:　RESD 16　　　　//定义 16 个 32 位字空间，L3 是标号

数据定义例子，逗号隔离(默认为 16 进制):

```
db    0x55 ;               //ust the byte 0x55
db    0x55,0x56,0x57;      // three bytes in succession
db    'a',0x55;            //character constants are OK
db    'hello',13,10,'$' ;  // so are string constants
dw    0x1234 ;             // 0x34 0x12
dw    'a' ;                // 0x41 0x00 (it's just a number)
dw    'ab' ;               //0x41 0x42 (character constant)
dw    'abc' ;              //0x41 0x42 0x43 0x00 (string)
dd    0x12345678 ;         //0x78 0x56 0x34 0x12
```

定义空间的例子：

```
buffer:     resb    64;    // reserve 64 bytes，buffer 是标号
wordvar:    resw     1;    // reserve a word，wordvar 是标号
```

例，代码和数据，以字地址计算：

```
#baseAddr 0000
        j       start;                          //0
        add $zero, $zero, $zero;                //4
        add $zero, $zero, $zero;                //8
        add $zero, $zero, $zero;                //C
        add $zero, $zero, $zero;                //10
        add $zero, $zero, $zero;                //14
        add $zero, $zero, $zero;                //18
        add $zero, $zero, $zero;                //1C

start:                                          //标号，可以不换行，后面直接跟代码
        nor  $at, $zero, $zero;                 //r1=FFFFFFFF
        add $v1, $at, $at;                      //r3=FFFFFFFE
        add $v1, $v1, $v1;                      //r3=FFFFFFFC
        add $v1, $v1, $v1;                      //r3=FFFFFFF8
        add $v1, $v1, $v1;                      //r3=FFFFFFF0

        add $v1, $v1, $v1;                      //r3=FFFFFFE0
        add $v1, $v1, $v1;                      //r3=FFFFFFC0
        nor  $s4, $v1, $zero;                   //r20=0000003F
        add $v1, $v1, $v1;                      //r3=FFFFFF80
        add $v1, $v1, $v1;                      //r3=FFFFFF00

        add $v1, $v1, $v1;                      //r3=FFFFFE00
        add $v1, $v1, $v1;                      //r3=FFFFFC00
        add $v1, $v1, $v1;                      //r3=FFFFF800
        add $v1, $v1, $v1;                      //r3=FFFFF000

        add $v1, $v1, $v1;                      //r3=FFFFE000
        add $v1, $v1, $v1;                      //r3=FFFFC000
        add $v1, $v1, $v1;                      //r3=FFFF8000
        add $v1, $v1, $v1;                      //r3=FFFF0000

        add $v1, $v1, $v1;                      //r3=FFFE0000
        add $v1, $v1, $v1;                      //r3=FFFC0000
        add $v1, $v1, $v1;                      //r3=FFF80000
        add $v1, $v1, $v1;                      //r3=FFF00000

        add $v1, $v1, $v1;                      //r3=FFE00000
        add $v1, $v1, $v1;                      //r3=FFC00000
        add $v1, $v1, $v1;              //r3=FF800000
        add $v1, $v1, $v1;              //r3=FF000000

        add $v1, $v1, $v1;              //r3=FE000000
        add $v1, $v1, $v1;              //r3=FC000000
        add $a2, $v1, $v1;              //r6=F8000000
        add $v1, $a2, $a2;              //r3=F0000000

        add $a0, $v1, $v1;              //r4=E0000000

        add $t5, $a0, $a0;              //r13=C0000000
        add $t0, $t5, $t5;              //r8=80000000

loop:                                   //标号，可以不换行，后面直接跟代码
        slt   $v0, $zero,$at;           //r2=00000001
        add $t6, $v0, $v0;
        add $t6, $t6, $t6;              //r14=4
        nor  $t2, $zero, $zero;         //r10=FFFFFFFF
        add $t2, $t2, $t2;              //r10=FFFFFFFE

loop1:
        sw   $a2, 4($v1);               //计数器端口:F0000004，送计数常数 r6=F8000000
        lw   $a1, 0($v1);
        add $a1, $a1, $a1;              //左移
        add $a1, $a1, $a1;
        sw   $a1, 0($v1);

        add $t1, $t1, $v0;              //r9=r9+1
        sw   $t1, 0($a0);               //r9 送 r4=E0000000 七段码端口
        lw   $t5, 14($zero);            //取存储器 20 单元预存数据至 r13,程序计数延时常数

loop2:                                  //标号，可以不换行，后面直接跟代码
        lw   $a1, 0($v1);               //读 GPIO 端口 F0000000 状态
        add $a1, $a1, $a1;
        add $a1, $a1, $a1;              //左移 2 位将 SW 与 LED 对齐，同时 D1D0 置 00，选择计数器
通道 0
        sw   $a1, 0($v1);               //r5 输出到 GPIO 端口 F0000000，计数器通道
counter_set=00

        lw   $a1, 0($v1);               //再读 GPIO 端口 F0000000 状态
        and $t3,$a1,$t0;                //取最高位=out0，屏蔽其余位送 r11
//      beq        $t3,$t0,C_init;      //out0=0,Counter 通道 0 溢出,转计数器初始化,修改 7 段码显
示:C_init
        add $t5, $t5, $v0;              //程序计数延时
        beq $t5, $zero,C_init;              //程序计数 r13=0,转计数器初始化,修改 7 段码显
示:C_init

l_next:                                     // 判断 7 段码显示模式：SW[4:3]控制
        lw    $a1, 0($v1);                  //再读 GPIO 端口 F0000000 开关 SW 状态
        add $s2, $t6, $t6;                  //r14=4,r18=00000008
        add $s6, $s2, $s2;                  //r22=00000010
        add $s2, $s2, $s6;                  //r18=00000018(00011000)
        and $t3, $a1, $s2;                  //取 SW[4:3]
        beq $t3, $zero, L20;                //SW[4:3]=00,7 段显示"点"循环移位：L20，SW0=0
        beq $t3, $s2, L21;                  //SW[4:3]=11，显示七段图形，L21，SW0=0
        add $s2, $t6, $t6;                  //r18=8
        beq $t3, $s2, L22;                  //SW[4:3]=01,七段显示预置数字，L22，SW0=1
        sw    $t1, 0($a0);                  //SW[4:3]=10，显示 r9，SW0=1
        j       loop2;

L20:
        beq $t2, $at, L4;                   //r10=ffffffff,转移 L4
        j       L3;

L4:
        nor  $t2, $zero, $zero;             //r10=ffffffff
        add $t2, $t2, $t2;                  //r10=fffffffe

L3:
        sw    $t2, 0($a0);                  //SW[4:3]=00,7 段显示点移位后显示
        j       loop2;

L21:
        lw    $t1, 60($s1);                 //SW[4:3]=11，从内存取预存七段图形
        sw    $t1, 0($a0);                  //SW[4:3]=11，显示七段图形
        j       loop2;

L22:
        lw    $t1, 20($s1);                 //SW[4:3]=01，从内存取预存数字
        sw    $t1, 0($a0);                  //SW[4:3]=01,七段显示预置数字
        j       loop2;

C_init:
        lw    $t5, 14($zero);               //取程序计数延时初始化常数
        add $t2, $t2, $t2;                  //r10=fffffffc，7 段图形点左移
        or    $t2, $t2, $v0;                //r10 末位置 1，对应右上角不显示
        add $s1, $s1, $t6;                  //r17=00000004，LED 图形访存地址+4
        and $s1, $s1, $s4;                  //r17=000000XX，屏蔽地址高位，只取 6 位
        add $t1, $t1, $v0;                  //r9+1
        beq $t1, $at, L6;                   //若 r9=ffffffff,重置 r9=5
        j      L7;

L6:
        add $t1, $zero, $t6;                //r9=4
        add $t1, $t1, $v0;                  //重置 r9=5

L7:
        lw    $a1, 0($v1);                  //读 GPIO 端口 F0000000 状态
        add $t3, $a1, $a1;
        add $t3, $t3, $t3;                  //左移 2 位将 SW 与 LED 对齐，同时 D1D0 置 00，选择计数器
通道 0
        sw   $t3, 0($v1);                   //r5 输出到 GPIO 端口 F0000000，计数器通道
counter_set=00
        sw   $a2,4($v1);                    //计数器端口:F0000004，送计数常数 r6=F8000000

        j      l_next;

        ......                                  //从此处-0000FFFC 填“00000000”。

#DataAddre: 00001000;                          //数据地址，此处 00001000H 开始定义数据
Data1:                                         //数据区 1，标号
dd FFFFFF00, 000002AB, 80000000, 0000003F, 00000001, FFFF0000, 0000FFFF,   80000000,
00000000, 11111111, 22222222, 33333333, 44444444, 55555555,   66666666, 77777777,
88888888, 99999999, AAAAAAAA, BBBBBBBB, CCCCCCCC, DDDDDDDD, EEEEEEEE, FFFFFFFF;
db 0x55,0x56,0x57,0x58;                        //db伪指令,定义数据0x55…在00001060
dw 'ab',0x1234;                                /dw伪指令,定义数据0x41,0x42,0x1234在
000010604
dd 0x12345678;                                 //dd伪指令，定义数据0x12345678在00001068
data2:RESB16;                                  //data2是标号，从0000106C开始定义16个字节空
间

                ......              //此处 00001070-00001FFC 填“00000000”。

#DataAddre: 00002000;                          //数据地址，此处 00002000H 开始定义数据
Data2:                                         //数据区 2，标号
557EF7E0,  D7BDFBD9,  D7DBFDB9,  DFCFFCFB,  DFCFBFFF,  F7F3DFFF,  FFFFDF3D,
FFFF9DB9,
FFFFBCFB,  DFCFFCFB,  DFCFBFFF,  D7DB9FFF,  D7DBFDB9,  D7BDFBD9,  FFFF07E0,
007E0FFF,
03bdf020, 03def820, 08002300;

                ......          //此处从0000204C-0000BFFC填“00000000”。
```

```
#DataAddre: 0000C000;                //数据地址，此处0000C000H开始定义数据
buffer:RESD32;                       //数据区3，buffer标号=0000C000，定义32个32
位字空间
                                     //到 0000C080 结束
```

MIPS 指令子集

本表根据 MIPS32 指令集和教材 Computer Organization and Design-The Hardware/Software Interface(第 3 版)附录 A10,选择了常用整数指令构成了 MIPS 指令子集。指令子集包括 CP0、异常处理等指令，可以支持简单的操作系统的运行。指令二进制编码以附录 A10 为标准。

1. Instruction format

- Typical format of R type instruction

Field	**opcode**	**rs**	**rt**	**rd**	**shamt**	**func**
Length	**6**	**5**	**5**	**5**	**5**	**6**

- Typical format of I type instruction

Field	**opcode**	**rs**	**rt**	**imme**
Length	**6**	**5**	**5**	**16**

- Table 1 Typical format of J type instruction

Field	**opcode**	**target**
Length	**6**	**26**

2. Instruction Table：

No	类型	功能	汇编符	OPCODE/FUNCT	描述(Displ.=Displacement*)
1	访存/取	取字	lw	23H	GPR [rt] <= Mem[GPR[rs]+sign_ext(Displacement)]
2		取字节	lb	20H	GPR [rt] <= {24{Mem[GPR[rs]+sign_ext(Displ.)][7]}, Mem[GPR[rs]+sign_ext(Displ.)][7:0]}
3		取字节	lbu	24H	GPR [rt] <={24’b0, Mem[GPR[rs]+ sign_ext(Displ.)][7:0]}
4		取半字	lh	21H	R[rt] <= {16{Mem[GPR[rs]+sign_ext(Displ.)][15]}, Mem[GPR[rs]+sign_ext(Displ.)][15:0]}
5		取半字	lhu	25H	R[rt] <= {16’b0, Mem[GPR[rs] + sign_ext(Displ.)][15:0]}
6	访存/存	存字	sw	2BH	Mem[GPR[rs]+sign_ext(Displacement)] <= R[rt]
7		存字节	sb	28H	Mem[GPR[rs]+sign_ext(Displ.)][7:0] <= R[rt][7:0]
8		存半字	sh	29H	Mem[GPR[rs]+sign_ext(Displ.)][15:0] <= R[rt][15:0]
9	ALU-R	加	add	0/20H	GPR[rd] <= GPR[rs] + GPR[rt]
10		无符号加	addu	0/21H	GPR[rd] <= GPR[rs] + GPR[rt]
11		减	sub	0/22H	GPR[rd] <= GPR[rs] - GPR[rt]
12		无符号减	subu	0/23H	GPR[rd] <= GPR[rs] - GPR[rt]
13		小于置 1	slt	0/2AH	GPR[rd] <= (GPR[rs] < GPR[rt]) ? 1:0
14		小于置 1	sltu	0/2BH	GPR[rd] <= (GPR[rs] < GPR[rt]) ? 1:0
15		与	and	0/24H	GPR[rd] <= GPR[rs] & GPR[rt
16		或	or	0/25H	GPR[rd] <= GPR[rs] | GPR[rt]
17		异或	xor	0/26H	GPR[rd] <= GPR[rs] ^ GPR[rt]
18		或非	nor	0/27	GPR[rd] <= ~(GPR[rs] | GPR[rt])
19		逻辑左移	sll	0/0H	GPR[rd] <= GPR[rt] << sa, 符号不变
20		逻辑右移	srl	0/2H	GPR[rd]<= GPR[rt] >> sa，符号不变

21		算术右移	sra	0/3H	GPR[rd]<= GPR[rt] << sa
22	乘除-R*	乘	mult	0/18H	{HI, LO} <= GPR[rs] × GPR[rt]
23		无符号乘	multu	0/19H	{HI, LO} <= GPR[rs] × GPR[rt]
24		除	div	0/1A	{HI, LO} <= GPR[rs] / GPR[rt]
25		无符号除	divu	0/1BH	{HI, LO} <= GPR[rs] / GPR[rt]
26	ALU-I	加立即数	addi	8H	GPR[rt] <= GPR[rs] + SignExt(Imm)
27		无符加立	addiu	9H	GPR[rt] <= GPR[rs] + SignExt(Imm)
28		与立即数	andi	CH	GPR[rt] <= GPR[rs] & ZeroExt(Imm)
29		或立即数	ori	DH	GPR[rt] <= GPR[rs] | ZeroExt(Imm)
30		异或常数	xori	EH	GPR[rt] <= GPR[rs] ^ ZeroExt(Imm)
31		高位取常	lui	FH	GPR[rt] <= {imm, 16’b0}
32		小于置 1	slti	AH	GPR[rt] <= (GPR[rs] < SignExt(Imm)) ? 1 : 0
33		小于置 1 无符号	sltiu	BH	GPR[rt] <= (GPR[rs] < SignExt(Imm)) ? 1 : 0
34	分支	相等转移	beq	4H	if (GPR[rs] == GPR[rt]) PC <= PC + 4 + Branch offset
35		不等转移	bne	5H	if (GPR[rs] != GPR[rt]) PC <= PC + 4 + Branch offset
36		<=0 转移	blez	6H	if (GPR[rs] <= 0) PC <= PC + 4 + Branch offset
37		>0 转移	bgtz	7H	if (GPR[rs] > 0) PC <= PC + 4 + Branch offset
38		<0 转移	bltz	01/*	if (GPR[rs] <0) PC <= PC + 4 + Branch offset
39		>=0 转移	bgez	01/*	if (GPR[rs] >= 0) PC <= PC + 4 + Branch offset
40	转移	无条件	j	2H	PC <= JumpAddr
41		直接调用	jal	3H	PC <= JumpAddr; GPR[ra] <= PC + 4
42		间址调用	jalr	0/9H	PC <= GPR[rs]; GPR[rd] <= PC + 4
43		间址转移	jr	0/8H	PC <= GPR[rs]
44	传输*	读 HI	mfhi	0/10H	GPR[rd] <= HI
45		读 LO	mflo	0/12H	GPR[rd] <= LO
46		写 HI	mthi	0/11H	HI <= GPR[rs]
47		写 LO	mtlo	0/13H	LO <= GPR[rs
48	特权	异常返回	eret	10/18H	PC <= EPC;(IR25==1, CP0 的 Cause 和 Status 寄存器有变化)
49		读 CP0	mfco	10H/*	GPR[rt] <= CP0[rd]
50		写 CP0	mtco	10H/*	
51	陷阱	断点异常	break	0/DH	EPC <= PC+4; PC <= 异常处理地址; (CP0 的 Cause 和 Status 寄存器有变化)
52		系统调用	syscall	0/CH	EPC = PC+4; PC <= 异常处理地址; (CP0 的 Cause 和 Status 寄存器有变化)
53	伪指令	取地址	la rdest, address		将地址值(而不是地址中的内容)保存到寄存器 rdest，rdest 是通用寄存器。(Rdest←&Table, Table 是标号)
54		传输指令	move rdest, rsrc		Rdest 和 rsrc 是通用寄存器
55		取立即数	li rdest, imm		Rdest 是通用寄存器
		清寄存器			

注：*Displacement 和 Offset 均为 16 立即数 Immediate(Imm)，应根据需要按符号位扩展或逻辑扩展。用于数据存储器偏移量(相对指针)时用 Displacement(Displ)表示，用于指令偏移（相对转移）时用 Offset 表示，需要除 4 转换为字地址。

①BLTZ：IR31..26/IR20..16＝01/00H

②BGEZ：IR31..26/IR20..16＝01/01H
③MFC0：IR31..26/IR25..21＝10/00H
④MTC0：IR31..26/IR25..21＝10/04H

3. Instruction description(历史版本)

R-Type

add

Format: add rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] + GPR[rt]

Exceptions:

Integer Overflow

addu

Format: addu rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] + GPR[rt]

Exceptions:

None

sub

Format: sub rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] - GPR[rt]

Exceptions:

Integer Overflow

subu

Format: subu rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] - GPR[rt]

Exceptions:

None

and

Format: and rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] AND GPR[rt]

or

Format: or rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] OR GPR[rt]

xor

Format: xor rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] XOR GPR[rt]

nor

Format: nor rd, rs, rt

Description: GPR[rd] ⟵ GPR[rs] NOR GPR[rt]

sll

Format: sll rd, rt, sa

Description: GPR[rd] ⟵ GPR[rt] << sa (logical)

Shift word Left Logical

srl

Format: srl rd, rt, sa

Description: GPR[rd] ⟵ GPR[rt] >> sa (logical)

Shift word Right Logical

sra

Format: sra rd, rt, sa

Description: GPR[rd] ⟵ GPR[rt] >> sa (arithmetic)

Shift word Right Arithmetic

sllv

Format: sllv rd, rt, rs

Description: GPR[rd] ⟵ GPR[rt] << GPR[rs]

Shift word Left Logical Variable

srlv

Format: srlv rd, rt, rs

Description: GPR[rd] ⟵ GPR[rt] >> GPR[rs]   (logical)

Shift word Right Logical Variable

srav

Format: srav rd, rt, rs

Description: GPR[rd] ⟵ GPR[rt] >> GPR[rs]   (arithmetic)

Shift word Right Arithmetic Variable

slt

Format: slt rd, rs, rt

Description: GPR[rd] ⟵ (GPR[rs] < GPR[rt]) ? 1 : 0

Set on Less Than.

GPR rs and GPR rt are treated as signed integers.

sltu

Format: sltu rd, rs, rt

Description: GPR[rd] ⟵ (GPR[rs] < GPR[rt]) ? 1 : 0

Set on Less Than Unsigned.

GPR rs and GPR rt are treated as unsigned integers.

jr

Format: jr rs

Description: PC ⟵ GPR[rs]

Jump Register.

Jump to the effective target address stored in GPR rs.

jalr

Format: jalr rs (rd = 31 implied)

Description: GPR[rd] ⟵ return_addr, PC ⟵ GPR[rs]

Jump and Link Register.

syscall

Format: syscall

Description: To cause a system call exception

System Call.

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

➢ Type

addi

Format: addi rt, rs, imme

Description: GPR[rt] ⟵ GPR[rs] + imme

The 16-bit immediate is signed.

Exceptions:

Integer Overflow

addiu

Format: addiu rt, rs, imme

Description: GPR[rt] ⟵GPR[rs] + imme

The 16-bit immediate is signed.

Exceptions:

None

ori

Format: ori rt, rs, imme

Description: GPR[rt] ⟵ GPR[rs] OR imme

The 16-bit immediate is zero-extended.

andi

Format: andi rt, rs, imme

Description: GPR[rt] ⟵ GPR[rs] AND imme

The 16-bit immediate is zero-extended.

xori

Format: xori rt, rs, imme

Description: GPR[rt] ⟵ GPR[rs] XOR imme

The 16-bit immediate is zero-extended.

lui

Format: lui rt, imme

Description: GPR[rt] ⟵ imme || 0^{16}

Load Upper Immediate.

The 16-bit immediate is shifted left 16 bits and concatenated with 16 bits of low-order zeros.

slti

Format: slti rt, rs, imme

Description: GPR[rt] ⟵ (GPR[rs] < imme) ? 1 : 0

Set on Less Than Immediate.

The 16-bit immediate is signed.

sltiu

Format: sltiu rt, rs, imme

Description: GPR[rt] ⟵ (GPR[rs] < imme) ? 1 : 0

Set on Less Than Immediate.Unsigned

The sign-extended 16-bit immediate is treated as an unsigned integer.

lw

Format: lw rt, imme(rs)

Description: GPR[rt] ⟵ memory[GPR[rs] + imme]

Load Word.

The 16-bit imme is signed.

lb

Format: lb rt, imme(rs)

Description: GPR[rt] ⟵ memory[GPR[rs] + imme]

Load Byte.

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

lbu

Format: lbu rt, imme(rs)

Description: GPR[rt] ⟵ memory[GPR[rs] + imme]

Load Byte Unsigned.

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

lh

Format: lh rt, imme(rs)

Description: GPR[rt] ⟵ memory[GPR[rs] + imme]

Load Halfword.

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

lhu

Format: lhu rt, imme(rs)

Description: GPR[rt] ⟵ memory[GPR[rs] + imme]

Load Halfword.

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

sw

Format: sw rt, imme(rs)

Description: memory[GPR[rs] + imme] ⟵ GPR[rt]

Store Word.

The least-significant 32-bit word of GPR rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

sh

Format: sh rt, imme(rs)

Description: memory[GPR[rs] + imme] ⟵ GPR[rt]

Store Halfword.

The least-significant 16-bit halfword of GPR rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

sb

Format: sb rt, imme(rs)

Description: memory[GPR[rs] + imme] ⟵ GPR[rt]

Store Byte.

The least-significant 8-bit byte of GPR rt is stored in memory at the location specified by the effective address. The 16-bit signed offset is added to the contents of GPR rs to form the effective address.

beq

Format: beq rs, rt, imme

Description: if GPR[rs] == GPR[rt] then branch

Branch on Equal.

bne

Format: bne rs, rt, imme

Description: if GPR[rs] != GPR[rt] then branch

Branch on Not Equal.

bgez

Format: bgez rs, imme

Description: if GPR[rs] >= 0 then branch

Branch on Great than or Equal to Zero.

bgezal

Format: bgezal rs, imme

Description: if GPR[rs] >= 0 then procedure_call

Branch on Great than or Equal to Zero And Link.

bgtz

Format: bgtz rs, imme

Description: if GPR[rs] > 0 then branch

Branch on Great Than Zero

blez

Format: blez rs, imme

Description: if GPR[rs] <= 0 then branch

Branch on Less than or Equal to Zero

bltz

Format: bltz rs, imme

Description: if GPR[rs] < 0 then branch

Branch on Less Than Zero

bltzal

Format: bltzal rs, imme

Description: if GPR[rs] < 0 then procedure_call

Branch on Less Than Zero And Link.

➢ J-Type

j

Format: j target

Description:

Jump.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region.

The low 28 bits of the target address is the 26-bit target field shifted left 2 bits. The remaining upper 4 bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

jal

Format: jal target

Description:

Jump And Link.

The same to j instruction except that the return address is placed in GPR 31.

➢ Coprocessor 0 instructions

mfc0

Format: mfc0 rt, rd, sel

Description: GPR[rt] $\leftarrow$ CPR[0, rd, sel]

Move From Coprocessor 0.

mtc0

Format: mtc0 rt, rd, sel

Description: CPR[0, rd, sel] ← GPR[rt]

Move To Coprocessor 0.

Exception Scheme in QS-I CPU

QS-I CPU has implemented some registers used in the handling of exception defined in MIPS architecture, though with some slight changes. Users can use instructions MFC0 and MTC0 to read or write these registers. Such as,

MFC0 $t0, Cause

MTC0 $t1, Cause

Coprocessor 0 registers

Status (#12):

31 30 29 28 27 9 8 0

IE		Interrupt Mask	Reserved

Cause (#13):

31 30 29 28 27 9 8 7 6 2 1 0

BR		Interrupt Pending		EXC	

IE: Interrupt Enable. If this bit is clear, then all interrupts will be disabled.

Interrupt Mask: this part is 20 bits wide, respectively indicates whether the up to 20 external interrupts should be masked or not. This part will be “and” with the interrupt pending part of the Cause register, and the priority of the 20 external interrupts is handled by software.

BR: Branch. If the instruction executing in the pipeline is a branch, then the BR will be asserted when any exception happens.

Interrupt Pending: this field indicates the pending state of the up to 20 external interrupts. Note that whenever an interrupt is serviced by software, the pending bit of the interrupt pending filed should be cleared.

EXC: Exception Code. When any exception happens, the exception code will be written into the Cause(#13) system register. And software can use the information provided by Cause register to determine which ISR to invoke.

EPC(#14):

Exception PC

When any exception happens, the PC of the victim instruction will be written into the EPC register. And when leaving from the ISR, software needs to read from EPC and then goes back to the interrupted instruction.

Programming guide

What exactly should software do whenever an exception occurs? The steps should be taken are listed below:

1. Note that when an exception starts, PC will jump to 0x4000000, which is the exception entrance. And software takes over the handling task from this point. First, the context of the CPU should be saved. Usually software should save all the registers and the stack.

2. After properly saving the context, software needs to know what the cause is of the occurring exception. The EXC (exception code) part of the Cause register offers the information.

3. If the cause of the occurring exception is an external interrupt, then check the interrupt pending bits of Cause register, and probably there will be more than one external interrupt pending. If that is true, software takes hold the priority management of the pending external interrupts. Attention should be paid that after servicing an external interrupt, the pending bit of Cause register must be cleared. If the occurring exception is not an external interrupt, then it's internal. Like a clock tick timer interrupt, a system call, a range issue and so forth.

4. After finding out the cause of the occurring exception, the according ISR (Interrupt Service Routine) is called and the exception is handled.

5. Before leaving from an exception, software needs to know the original interrupted PC, which is offered by EPC register. And what is very important, the context needs to be restored properly before leaving.