实验6

简单计算机系统

系统设计D

主要内容

1、简单计算机系统实验任务简介

2、完善数据通路

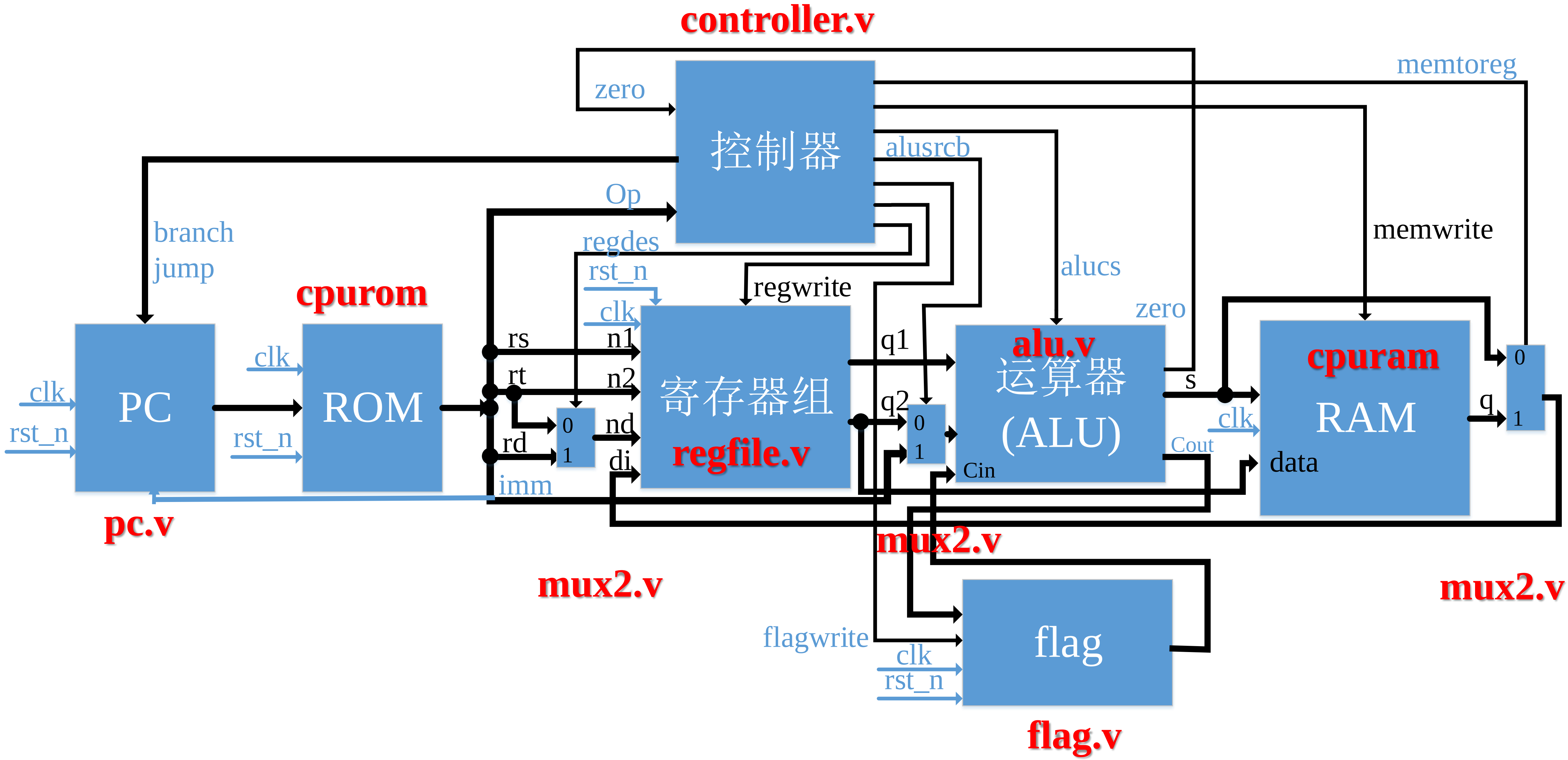
3、动手练习：仿真验证功能

实验任务简介

- ■ 实现一种简单计算机系统的设计.
  - ✓ 精简的MIPS指令集
  - ✓ EDA仿真
- ■ 编写程序，仿真验证所设计系统的功能
  - ✓ 用汇编格式编写程序，并翻译成机器码.
  - ✓ 将机器码程序放入ROM，通过仿真验证简单计算机系统的功能.

简单计算机系统指令集

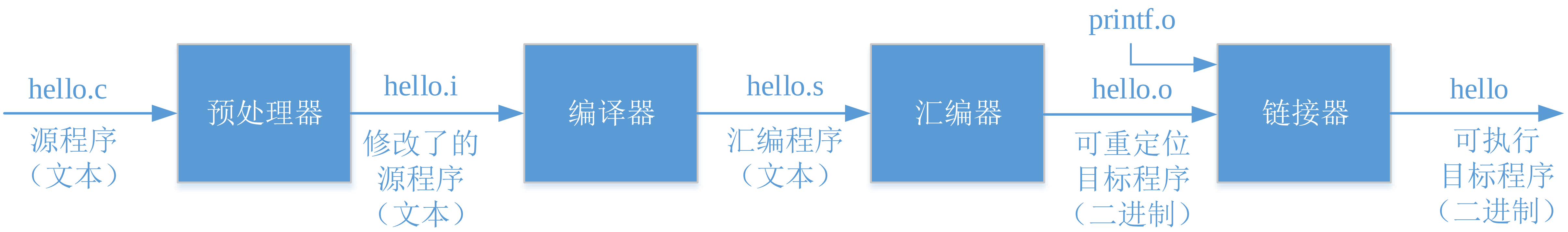
操作名称	操作码	汇编语言格式指令	执行操作
与	0000	AND Rd, Rs, Rt	Rd←Rs and Rt; PC ← PC + 1
或	0001	OR Rd, Rs, Rt	Rd←Rs or Rt; PC ← PC +1
不带进位加	0010	ADD Rd, Rs, Rt	Rd←Rs+ Rt; PC ← PC + 1
不带借位减	0011	SUB Rd, Rs, Rt	Rd←Rs– Rt; PC ← PC + 1
无符号数比较	0100	SLT Rd,Rs,Rt	If Rs<Rt, Rd=1 else Rd=0; PC ← PC + 1
带借位减	0101	SUBC Rd, Rs, Rt	Rd←Rs– Rt-(1-C); PC ← PC + 1
带进位加	0110	ADDC Rd, Rs, Rt	Rd←Rs+ Rt+C; PC ← PC + 1
立即数与	1000	ANDI Rt, Rs, imm	Rt←Rs and imm; PC ← PC +1
立即数或	1001	ORI Rt, Rs, imm	Rt←Rs or imm; PC ←PC +1
立即数加	1010	ADDI Rt, Rs, imm	Rt←Rs+ imm; PC ← PC +1
读存储器	1011	LW Rt, Rs, imm	Rt←MEM[Rs+imm]; PC ←PC +1
写存储器	1100	SW Rt, Rs, imm	MEM[Rs+imm] ←Rt；PC ← PC +1
相等时跳转	1101	BEQ Rs, Rt, imm	If Rt=Rs, PC←PC+imm+1 else PC←PC+1
不等时跳转	1110	BNE Rs, Rt, imm	If Rt!=Rs, PC←PC+imm+1 else PC←PC+1
无条件跳转	0111	JMP imm	PC ← imm



controller.v
控制器
zero
Op
regdes
rst_n
clk
regwrite
alusrcb
alucs
memtoreg
memwrite
zero
branch
jump
cpurom
clk
rst_n
PC
ROM
pc.v
rs
rt
rd
imm
n1
n2
nd
di
0
1
寄存器组
regfile.v
mux2.v
q1
q2
0
1
alu.v
运算器
(ALU)
Cin
s
Cout
mux2.v
cpuram
RAM
clk
data
q
0
1
mux2.v
flagwrite
clk
rst_n
flag
flag.v

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```



高级语言程序

```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

编译器

汇编语言程序

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      lw   x5, 0(x6)
      lw   x7, 4(x6)
      sw   x7, 0(x6)
      sw   x5, 4(x6)
      jalr x0, 0(x1)
```

汇编器

机器语言程序

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

32位标准MIPS指令集

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **3个操作数均来自寄存器（共有32个寄存器，位宽为32）**
  - ✓ 源操作数：rs, rt
  - ✓ 目的操作数：rd
- **其他位域**
  - ✓ 操作码位域op：0
  - ✓ 功能域funct: 决定cpu要执行何种操作，如**add为32，sub为34**
  - ✓ shamt: 移位操作时决定移位量*shift amount*，其他指令取0

add rd, rs, rt

汇编代码

add $s0, $s1,$s2

sub $t0, $t3, $t5

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

机器码

0x02328020

0x016D4022

op	rs	rt	rd	shamt	funct
000000	10001	10010	10000	00000	100000
000000	01011	01101	01000	00000	100010
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

- **3个操作数**
  - ✓ rs, rt：寄存器操作数，rs为源操作数，rt为目的操作数
  - ✓ imm：16-bit立即数

Assembly Code / Field Values

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw   $t2, 32($0)`	35	0	10	32
`sw   $s1,  4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

- ✓ *Jump-type*
- ✓ 26-bit address operand (`addr`)
- ✓ Used for jump instructions (`j`)

Table B.1 Instructions, sorted by opcode

Opcode	Name	Description	Operation
000000 (0)	`R-type`	all R-type instructions	see Table B.2
000001 (1) (`rt` = 0/1)	`bltz rs, label` / `bgez rs, label`	branch less than zero/branch greater than or equal to zero	`if ([rs] < 0) PC = BTA/` `if ([rs] ≥ 0) PC = BTA`
000010 (2)	`j label`	jump	`PC = JTA`
000011 (3)	`jal label`	jump and link	`$ra = PC + 4, PC = JTA`
000100 (4)	`beq rs, rt, label`	branch if equal	`if ([rs] == [rt]) PC = BTA`
000101 (5)	`bne rs, rt, label`	branch if not equal	`if ([rs] != [rt]) PC = BTA`
000110 (6)	`blez rs, label`	branch if less than or equal to zero	`if ([rs] ≤ 0) PC = BTA`
000111 (7)	`bgtz rs, label`	branch if greater than zero	`if ([rs] > 0) PC = BTA`
001000 (8)	`addi rt, rs, imm`	add immediate	`[rt] = [rs] + SignImm`
001001 (9)	`addiu rt, rs, imm`	add immediate unsigned	`[rt] = [rs] + SignImm`
001010 (10)	`slti rt, rs, imm`	set less than immediate	`[rs] < SignImm ? [rt] = 1 : [rt] = 0`
001011 (11)	`sltiu rt, rs, imm`	set less than immediate unsigned	`[rs] < SignImm ? [rt] = 1 : [rt] = 0`
001100 (12)	`andi rt, rs, imm`	and immediate	`[rt] = [rs] & ZeroImm`
001101 (13)	`ori rt, rs, imm`	or immediate	`[rt] = [rs] | ZeroImm`
001110 (14)	`xori rt, rs, imm`	xor immediate	`[rt] = [rs] ^ ZeroImm`
001111 (15)	`lui rt, imm`	load upper immediate	`[rt] = {imm, 16'b0}`
010000 (16) (`rs` = 0/4)	`mfc0 rt, rd` / `mtc0 rt, rd`	move from/to coprocessor 0	`[rt] = [rd]/[rd] = [rt]` `(rd is in coprocessor 0)`
010001 (17)	`F-type`	`fop` = 16/17: F-type instructions	see Table B.3

010001 (17)	F-type	fop = 16/17: F-type instructions	see Table B.3
010001 (17) (rt = 0/1)	bc1f label/ bc1t label	fop = 8: branch if fpcond is FALSE/TRUE	if (fpcond == 0) PC = BTA/ if (fpcond == 1) PC = BTA
011100 (28) (func = 2)	mul rd, rs, rt	multiply (32-bit result)	[rd] = [rs] x [rt]
100000 (32)	lb rt, imm(rs)	load byte	[rt] = SignExt ($[\text{Address}]_{7:0}$)
100001 (33)	lh rt, imm(rs)	load halfword	[rt] = SignExt ($[\text{Address}]_{15:0}$)
100011 (35)	lw rt, imm(rs)	load word	[rt] = [Address]
100100 (36)	lbu rt, imm(rs)	load byte unsigned	[rt] = ZeroExt ($[\text{Address}]_{7:0}$)
100101 (37)	lhu rt, imm(rs)	load halfword unsigned	[rt] = ZeroExt ($[\text{Address}]_{15:0}$)

(*continued*)

Table B.1 Instructions, sorted by opcode—Cont'd

Opcode	Name	Description	Operation
101000 (40)	`sb rt, imm(rs)`	store byte	$[Address]_{7:0} = [rt]_{7:0}$
101001 (41)	`sh rt, imm(rs)`	store halfword	$[Address]_{15:0} = [rt]_{15:0}$
101011 (43)	`sw rt, imm(rs)`	store word	[Address] = [rt]
110001 (49)	`lwc1 ft, imm(rs)`	load word to FP coprocessor 1	[ft] = [Address]
111001 (56)	`swc1 ft, imm(rs)`	store word to FP coprocessor 1	[Address] = [ft]

Table B.2 R-type instructions, sorted by funct field

Funct	Name	Description	Operation
000000 (0)	`sll rd, rt, shamt`	shift left logical	`[rd] = [rt] << shamt`
000010 (2)	`srl rd, rt, shamt`	shift right logical	`[rd] = [rt] >> shamt`
000011 (3)	`sra rd, rt, shamt`	shift right arithmetic	`[rd] = [rt] >>> shamt`
000100 (4)	`sllv rd, rt, rs`	shift left logical variable	$[rd] = [rt] \ll [rs]_{4:0}$
000110 (6)	`srlv rd, rt, rs`	shift right logical variable	$[rd] = [rt] \gg [rs]_{4:0}$
000111 (7)	`srav rd, rt, rs`	shift right arithmetic variable	$[rd] = [rt] >\!>\!> [rs]_{4:0}$
001000 (8)	`jr rs`	jump register	`PC = [rs]`
001001 (9)	`jalr rs`	jump and link register	`$ra = PC + 4, PC = [rs]`
001100 (12)	`syscall`	system call	`system call exception`
001101 (13)	`break`	break	`break exception`
010000 (16)	`mfhi rd`	move from hi	`[rd] = [hi]`
010001 (17)	`mthi rs`	move to hi	`[hi] = [rs]`
010010 (18)	`mflo rd`	move from lo	`[rd] = [lo]`
010011 (19)	`mtlo rs`	move to lo	`[lo] = [rs]`
011000 (24)	`mult rs, rt`	multiply	`{[hi], [lo]} = [rs] × [rt]`
011001 (25)	`multu rs, rt`	multiply unsigned	`{[hi], [lo]} = [rs] × [rt]`
011010 (26)	`div rs, rt`	divide	`[lo] = [rs]/[rt],` `[hi] = [rs]%[rt]`
011011 (27)	`divu rs, rt`	divide unsigned	`[lo] = [rs]/[rt],` `[hi] = [rs]%[rt]`

Table B.2 **R-type instructions, sorted by funct field—Cont'd**

Funct	Name	Description	Operation
100000 (32)	add rd, rs, rt	add	[rd] = [rs] + [rt]
100001 (33)	addu rd, rs, rt	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub rd, rs, rt	subtract	[rd] = [rs] - [rt]
100011 (35)	subu rd, rs, rt	subtract unsigned	[rd] = [rs] - [rt]
100100 (36)	and rd, rs, rt	and	[rd] = [rs] & [rt]
100101 (37)	or rd, rs, rt	or	[rd] = [rs] | [rt]
100110 (38)	xor rd, rs, rt	xor	[rd] = [rs] ^ [rt]
100111 (39)	nor rd, rs, rt	nor	[rd] = ~([rs] | [rt])
101010 (42)	slt rd, rs, rt	set less than	[rs] < [rt] ? [rd] = 1 : [rd] = 0
101011 (43)	sltu rd, rs, rt	set less than unsigned	[rs] < [rt] ? [rd] = 1 : [rd] = 0

Table B.3 **F-type instructions (`fop` = 16/17)**

Funct	Name	Description	Operation
000000 (0)	`add.s fd, fs, ft /` `add.d fd, fs, ft`	FP add	`[fd] = [fs] + [ft]`
000001 (1)	`sub.s fd, fs, ft /` `sub.d fd, fs, ft`	FP subtract	`[fd] = [fs] - [ft]`
000010 (2)	`mul.s fd, fs, ft /` `mul.d fd, fs, ft`	FP multiply	`[fd] = [fs] × [ft]`
000011 (3)	`div.s fd, fs, ft /` `div.d fd, fs, ft`	FP divide	`[fd] = [fs]/[ft]`
000101 (5)	`abs.s fd, fs /` `abs.d fd, fs`	FP absolute value	`[fd] = ([fs] < 0) ? [-fs] : [fs]`
000111 (7)	`neg.s fd, fs /` `neg.d fd, fs`	FP negation	`[fd] = [-fs]`
111010 (58)	`c.seq.s fs, ft /` `c.seq.d fs, ft`	FP equality comparison	`fpcond = ([fs] == [ft])`
111100 (60)	`c.lt.s fs, ft /` `c.lt.d fs, ft`	FP less than comparison	`fpcond = ([fs] < [ft])`
111110 (62)	`c.le.s fs, ft /` `c.le.d fs, ft`	FP less than or equal comparison	`fpcond = ([fs] ≤ [ft])`

实验任务6

任务6.1

（1）结合实验1~实验5，以及理论课内容，实现可以执行32位标准MIPS指令集中add和sub的数据通路；给出仿真结果，分析验证功能.

（2）测试代码为如下两条指令（s0/s1/s2的寄存器号分别为16/17/18，t0/t3/t5的寄存器号分别为8/11/13）：

add $s0, $s1,$s2

sub $t0, $t3, $t5

在寄存器组的实现模块中，给s1/s2及t3/t5赋初值；将上述代码段翻译成机器码，写入ROM数据文件中.

（3）分析仿真结果.

实验任务6

任务6.2

（1）用32位标准MIPS指令集完成任务5.2.

（2）增加必要的指令（可仅为完成上述任务所需要的指令），完善数据通路.

（3）给出仿真结果，分析验证功能.

THE END