



北京大学

第七讲 RISC-V指令和单周期处理器

RISC-V & SingleCycle CPU

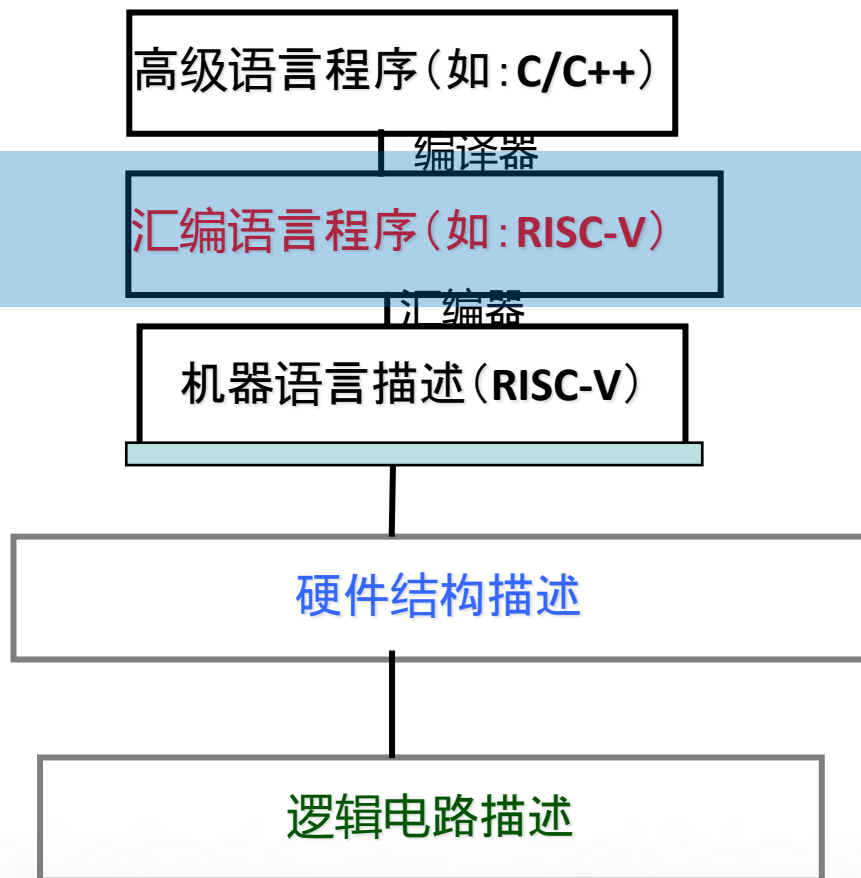
佟冬

tongdong@pku.edu.cn

微处理器研究开发中心 (MPRC)
计算机科学技术系

北京大学

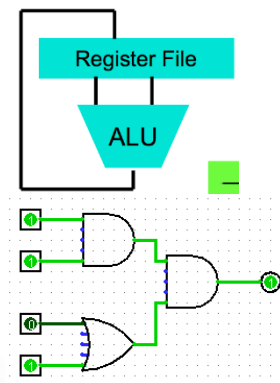
设计准则1：层次化（抽象）



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
□ lw      x3, 0(x10)
□ lw      x4, 4(x10)
□ sw      x4, 0(x10)
□ sw      x3, 4(x10)
```

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```



RISC-V

- 新的开源的、免费的指令系统，逐渐壮大的共享软件生态环境支持。
 - 32位、64位和128位多种形态（本课程用32位RV32I）

- 为什么用RISC-V而不用Intel 80x86？
 - RISC-V 简单、优雅，不需要更重繁琐的细节。
 - RISC-V 已经在世界范围内被爆发式的使用，尤其在加速器和物联网领域。

RISC-V汇编中的变量：寄存器

- RV32I共有32个32位寄存器
- RISC-V的寄存器编号：x0 – x31
- x0 特殊寄存器，长为0值
 - 因此实际上只有31个32位寄存器可以用来保存变量值
- 每个寄存器可以用其他的名字来访问
 - C语言的ABI定义

RISC-V汇编中的注释

- 注释是一种提高代码可读性的方法
- RISC-V中用“井号” (#) 作为注释标志
 - 所有处在井号后面的字母是注释，会被汇编器忽略
 - 就像C99中的“//”
- 注意：与C语言不同
 - C语言的注释形式
/* 注释 */
可以扩展多行注释

RISC-V的汇编格式

□ 指令语法:

One two, three, four **add x1, x2, x3**

其中:

One = 操作名称

two = 操作目的结果(“destination”)

three = 第一源操作数 (“source1”)

four = 第二源操作数 (“source2”)

□ 语法是刚性的:

- 1个操作, 3个操作数

□ 可以通过规整性简化硬件实现。

RISC-V汇编中的加法和减法

□ 汇编中的加法

- 例: `add x1,x2,x3` (RISC-V)
- 等效于: $a = b + c$ (C语言)
- 其中C变量 \Leftrightarrow RISC-V寄存器:
 $a \Leftrightarrow x1, b \Leftrightarrow x2, c \Leftrightarrow x3$

□ 汇编中的减法

- 例: `sub x3,x4,x5` (RISC-V)
- 等效于: $d = e - f$ (C语言)
- 其中C变量 \Leftrightarrow RISC-V寄存器:
 $d \Leftrightarrow x3, e \Leftrightarrow x4, f \Leftrightarrow x5$

□ 注意: C语言中的一行可能对应多行的汇编

RISC-V的立即数

- 立即数是常量数
- 立即数经常用到，需要特别指令
- 立即数加法Add Immediate:
 - `addi x3,x4,10 (RISC-V)`
 - `f = g + 10 (C语言)`
- 负立即数
 - `addi x3,x4,-10 (RISC-V)`
 - `f = g - 10 (C语言)`
- 为什么不需要立即数减法指令？

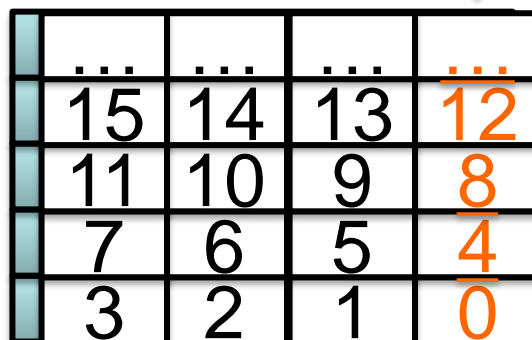
Zero寄存器

- 特殊的立即数，数零 (0)，经常出现在代码中
- 所以寄存器0 (x0)被硬连线成值0，例如
 - `add x3,x4,x0` (RISC-V)
 - `f = g` (C语言)
- 因为实在硬件中定义的，所以下面的指令
 - `add x0,x3,x4` 将不做任何事！

RISC-V的内存地址

- 字节为单位编址
- RV32I中的字word包括4个字节
- 采用Little_endian小端法的地址排布

字中的最低字节Least-significant byte



...
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

31 24 23 16 15 8 7 0

最低字节最小地址

“小端法”表示地址0

从内存到寄存器的数据读load

□ C代码

```
int A[100];  
g = h + A[3];
```

□ 使用RISC-V的Load Word (lw) 指令

```
lw x10,12(x15) # Reg x10 gets A[3]  
add x11,x12,x10 # g = h + A[3]
```

□ 注意： x15 – 基址寄存器 (指到A[0])

12 – 字节单位的偏移offset

□ 在汇编时刻，偏移必须是常值。

从寄存器到内存的数据写store

□ C代码

```
int A[100];
```

```
A[10] = h + A[3];
```

□ 使用RISC-V的Store Word (sw)指令

```
lw x10,12(x15)           # Temp reg x10 gets A[3]
```

```
add x10,x12,x10           # Temp reg x10 gets h + A[3]
```

```
sw x10,40(x15)           # A[10] = h + A[3]
```

□ 注：x15 – 基址寄存器（指针）

12,40 – 字节为单位的偏移offset

x15+12和x15+40 必须是4的倍数

RISC-V分支指令

- 基于计算，做些不同的事
- 在高级编程语言中if-语句
- 在RISC-V: if-语句指令是

`beq register1,register2,L1`

- 意义：跳转到语句标号L1
如果(`register1`中的值) == (`register2`中的值)
...否则，进行下一条语句
- `beq` 代表相等则转移
- 定一条指令: `bne` 代表不相等则转移

分支指令类型

- 分支指令：改变控制流
- 条件分支指令Conditional Branch – 依赖比较的结果改变控制流
 - branch if equal (beq) or branch if not equal (bne)
 - branch if less than (blt) and branch if greater than or equal (bge)
- 非条件分支指令Unconditional Branch
 - RISC-V指令：跳转(jal), jal label
 - 伪代码：j label

If-else程序实例

□ 假设编译后变量分配

$f \rightarrow x10$

$g \rightarrow x11$

$h \rightarrow x12$

$i \rightarrow x13$

$j \rightarrow x14$

if (i == j)

$f = g + h;$

else

$f = g - h;$

bne x13,x14,Else

add x10,x11,x12

j Exit

Else: sub x10,x11,x12

Exit:

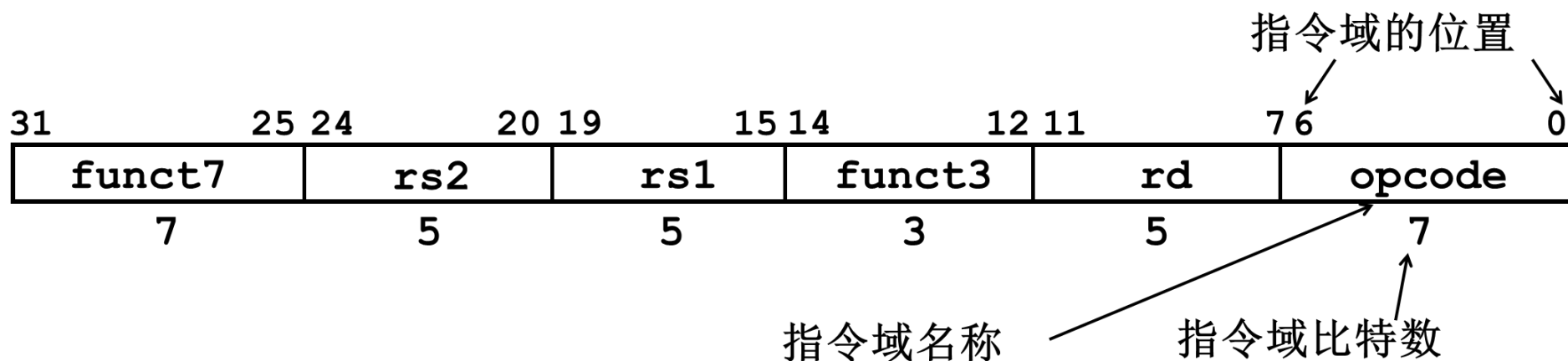
RISC-V逻辑运算指令

□ 两种指令

- 寄存器: `and x5, x6, x7` # $x5 = x6 \& x7$
- 立即数: `andi x5, x6, 3` # $x5 = x6 \& 3$

逻辑操作	C 操作符	Java 操作符	RISC-V 指令
按位与AND	<code>&</code>	<code>&</code>	<code>and</code>
按位或OR	<code> </code>	<code> </code>	<code>or</code>
按位异或XOR	<code>^</code>	<code>^</code>	<code>xor</code>
逻辑左移	<code><<</code>	<code><<</code>	<code>sll</code>
逻辑右移	<code>>></code>	<code>>></code>	<code>srl</code>

R型指令格式

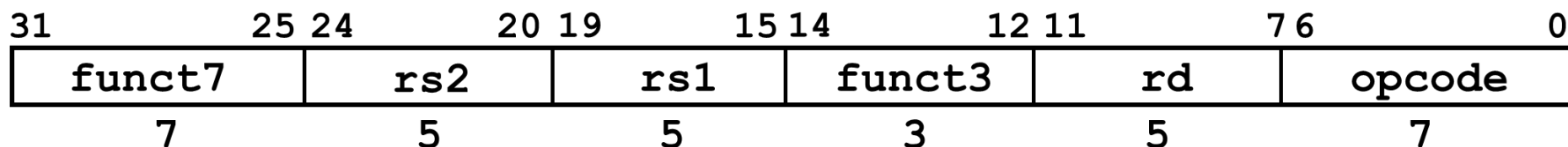


□ 32位指令分为6个指令域

□ 例如

- opcode: 7比特指令域，指令的6-0位
- rs2: 5比特指令域，指令的24-20位

R型指令格式

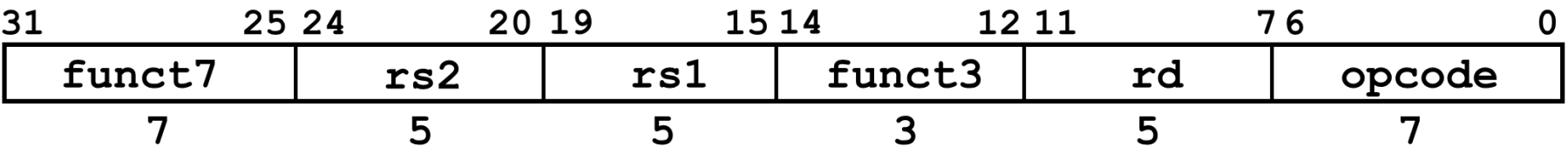


- ❑ **opcode**: 部分表明指令类型
 - 如R型指令, `opcode=0110011`
- ❑ **funct7+funct3**: 表明某opcode指令类型下的具体指令操作功能
- ❑ **rd**目标寄存器, **rs1**第一源寄存器, **rs2**第二源寄存器
 - 分别为5比特, 表示32个寄存器地址 (x0-x31)

R型指令

□ RISC-V的汇编指令

add x18, x19, x10



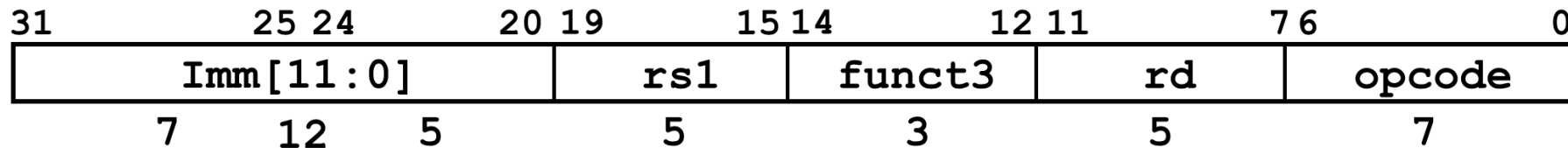
add rs2=10 rs1=19 add rd=18 Reg-Reg OP

RV32I的所有R型指令

- 注意观察funct7的编码
- add和sub指令的区别， srl和sra的区别

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

I型指令格式

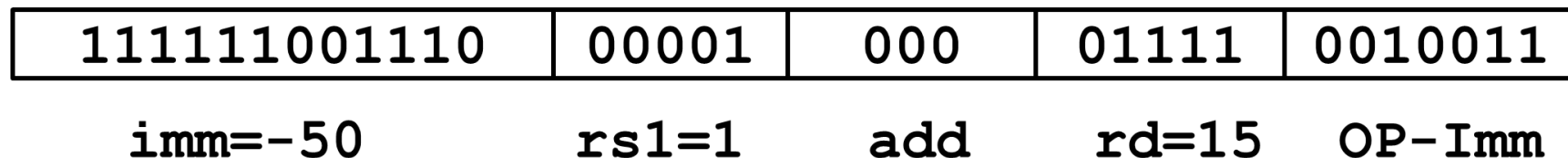
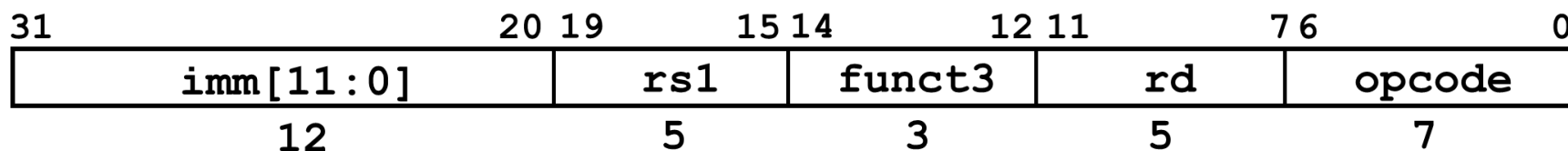


- ❑ R型指令的rs2和funct7合并成立即数Imm[11:0]
- ❑ 12位有符号整型数
- ❑ 运算时符号扩展成32位整型数
- ❑ 大于12位的立即数如何构造？

I型指令示例

□ RISC-V汇编指令

Add x15, x1, -50

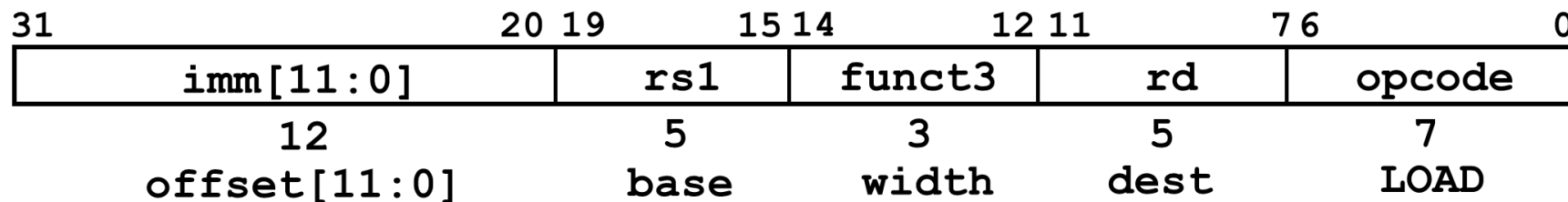


RV32I的所有I型指令

- 注意观察立即数移位指令的立即数指令域
- 5比特移位立即数，7比特funct7

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

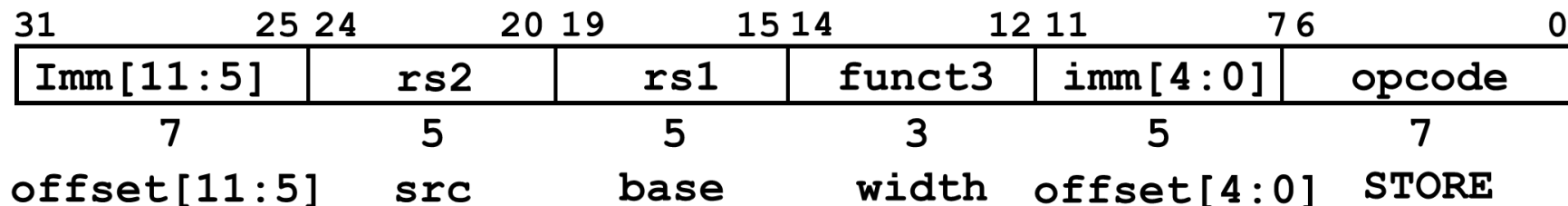
load指令的指令格式



- ❑ I型指令
- ❑ 12比特立即数加上基址寄存器rs1的值，作为内存地址访问内存
- ❑ 内存读的结果写回rd寄存器中
- ❑ funct3表示数据类型

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	110	rd	0000011	lhu

S型指令Store

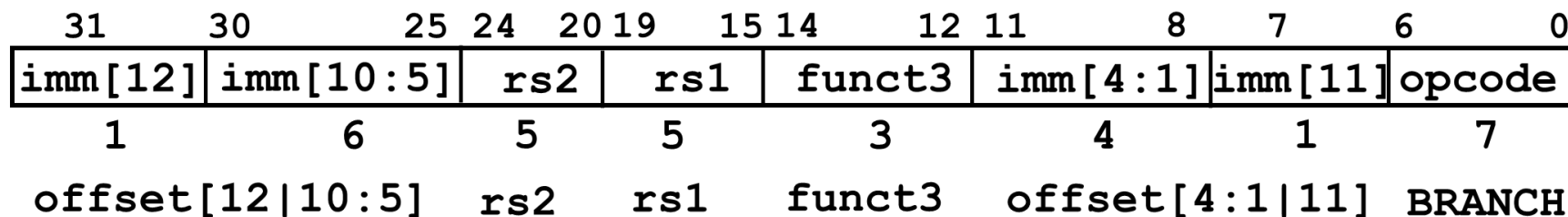


- ❑ Store用两个寄存器，rs1和rs2
- ❑ 12比特立即数由两部分拼接而成，加上基址寄存器rs1的值，形成内存访问地址
- ❑ 将rs2寄存器的值写入内存
- ❑ 没有rd指令位域

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

width

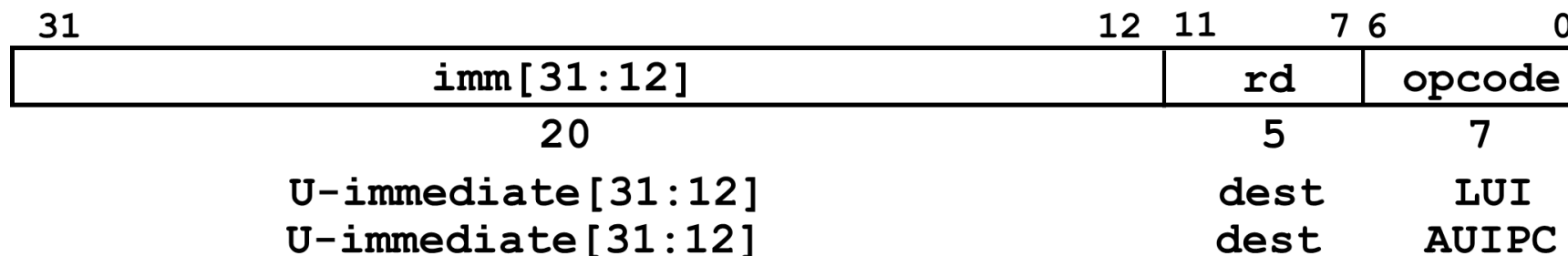
B型指令条件转移指令Branch



- ❑ 比较rs1和rs2寄存器的值
- ❑ funct3定义转移条件
- ❑ 立即数Imm由多个不同位域拼接而成
- ❑ 如果条件满足: $pc = pc + imm[12:1]*2$
- ❑ 如果条件不满足: $pc = pc + 4$
- ❑ RISC-V的pc值最低位恒为0, 所以不用表示在立即数中。

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

U型指令 “高位立即数” 指令

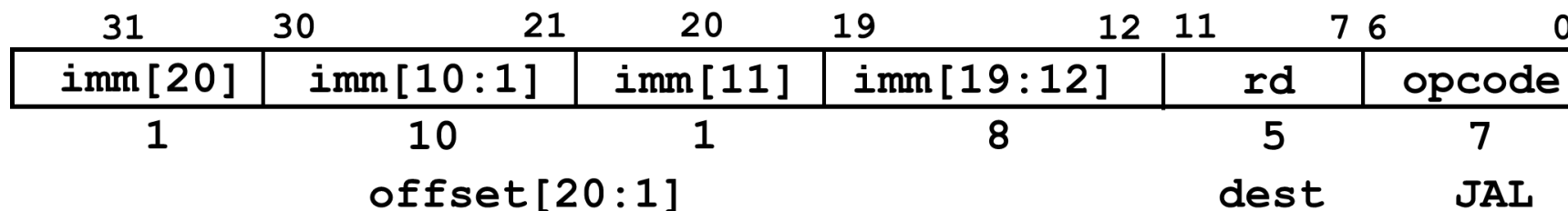


- ❑ 20比特立即数放置到目标寄存器rd的高20位，低12位置0
- ❑ 用lui指令生成32比特长立即数

- ❑ LUI x10, 0x87654 # x10 = 0x87654000
- ❑ ADDI x10, x10, 0x321 # x10 = 0x87654321

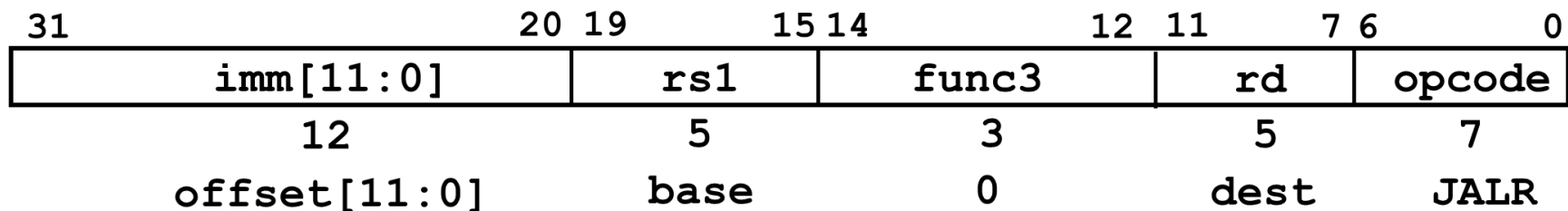
- ❑ 汇编伪代码
- ❑ `li x10, 0xDEADBEEF` # Creates two instructions

J型指令JAL



- ❑ PC相对寻址
- ❑ jal指令将 $pc+4$ 保存在rd寄存器中
- ❑ jal指令跳转到 $pc+imm[20:1]*2$
- ❑ 汇编伪代码j lab: jal x0, lab

JALR跳转指令call



- ❑ JALR rd, rs, immediate
 - PC+4写入rd (return address)
- ❑ 设置PC = rs + immediate
- ❑ 立即数模式和算术指令和load指令相同
 - 立即数不乘以2
 - 对比branches 和 JAL 指令

RISC-V的指令格式汇总

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7				rs2		rs1		funct3		rd			opcode		R-type
imm[11:0]						rs1		funct3		rd			opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]			opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]			opcode		B-type
imm[31:12]										rd			opcode		U-type
imm[20 10:1 11]]						imm[19:12]				rd			opcode		J-type

RV32I指令列表 (1)

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20 10:1 11 19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

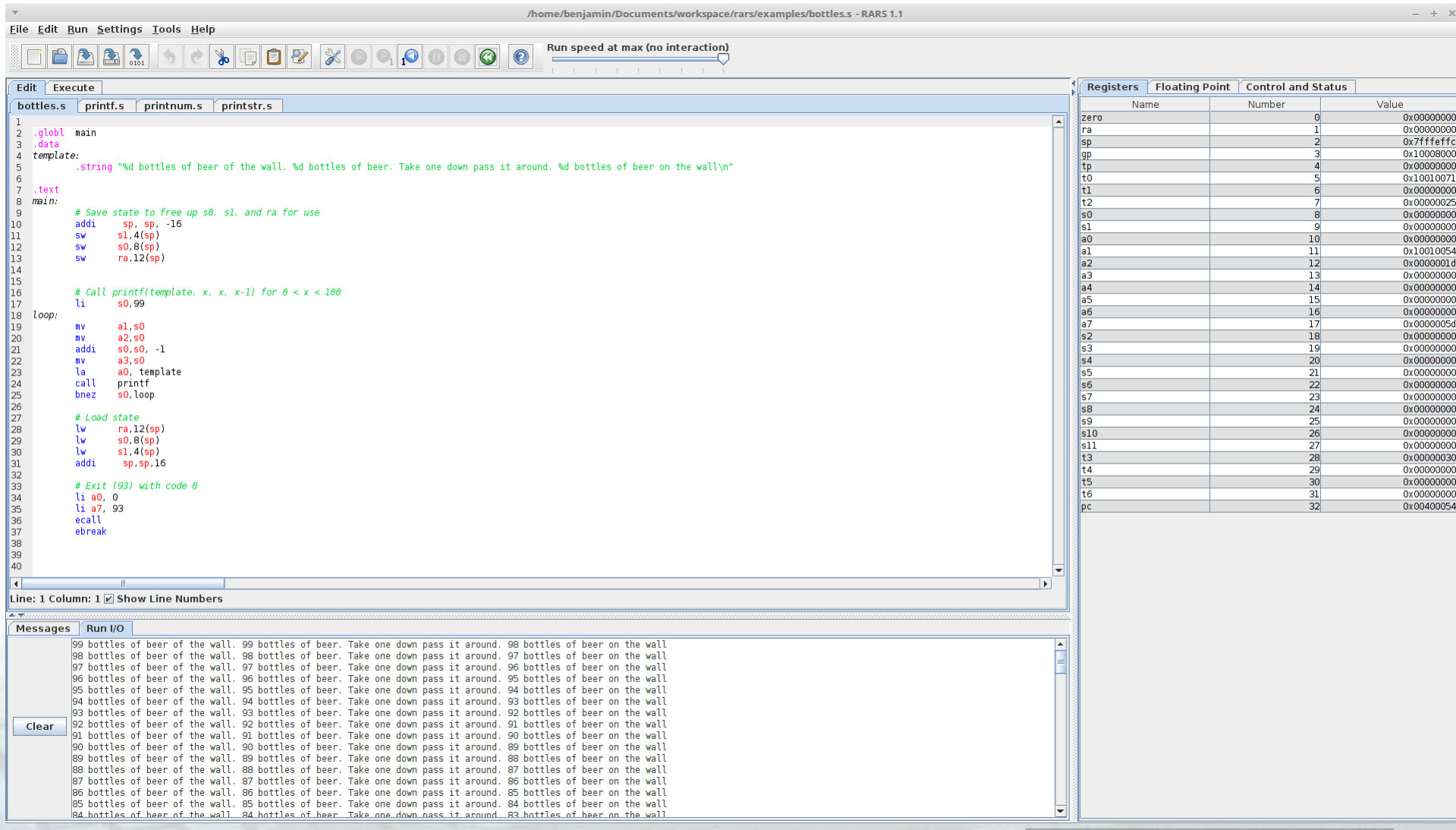
RV32I指令列表 (2)

0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
0000000	rs2	rs1	000	rd	0110011	ADD	
0100000	rs2	rs1	000	rd	0110011	SUB	
0000000	rs2	rs1	001	rd	0110011	SLL	
0000000	rs2	rs1	010	rd	0110011	SLT	
0000000	rs2	rs1	011	rd	0110011	SLTU	
0000000	rs2	rs1	100	rd	0110011	XOR	
0000000	rs2	rs1	101	rd	0110011	SRL	
0100000	rs2	rs1	101	rd	0110011	SRA	
0000000	rs2	rs1	110	rd	0110011	OR	
0000000	rs2	rs1	111	rd	0110011	AND	
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW	
csr		rs1	010	rd	1110011	CSRRS	
csr		rs1	011	rd	1110011	CSRRC	
csr		zimm	101	rd	1110011	CSRRWI	
csr		zimm	110	rd	1110011	CSRRSI	
csr		zimm	111	rd	1110011	CSRRCI	

本课程不涉及这些指令

RISC-V汇编模拟器RARS

□ <https://github.com/TheThirdOne/rars>





北京大学

第八讲 RISC-V单周期处理器

RISC-V Single-Cycle Processor

佟冬

tongdong@pku.edu.cn

微处理器研究开发中心 (MPRC)
计算机科学技术系

北京大学

RV32I指令列表 (1)

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20 10:1 11 19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

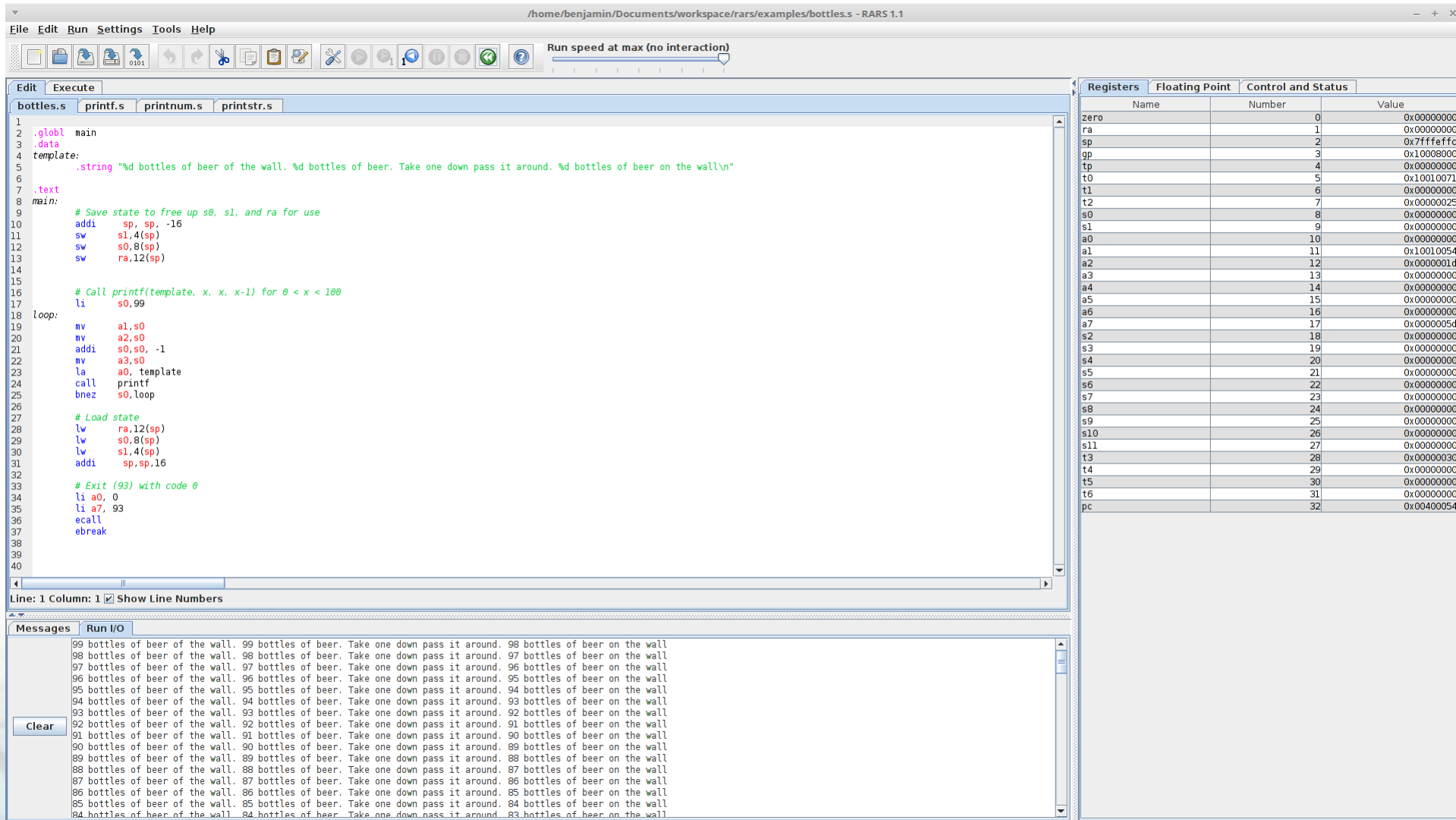
RV32I指令列表 (2)

0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
0000000	rs2	rs1	000	rd	0110011	ADD	
0100000	rs2	rs1	000	rd	0110011	SUB	
0000000	rs2	rs1	001	rd	0110011	SLL	
0000000	rs2	rs1	010	rd	0110011	SLT	
0000000	rs2	rs1	011	rd	0110011	SLTU	
0000000	rs2	rs1	100	rd	0110011	XOR	
0000000	rs2	rs1	101	rd	0110011	SRL	
0100000	rs2	rs1	101	rd	0110011	SRA	
0000000	rs2	rs1	110	rd	0110011	OR	
0000000	rs2	rs1	111	rd	0110011	AND	
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW	
csr		rs1	010	rd	1110011	CSRRS	
csr		rs1	011	rd	1110011	CSRRC	
csr		zimm	101	rd	1110011	CSRRWI	
csr		zimm	110	rd	1110011	CSRRSI	
csr		zimm	111	rd	1110011	CSRRCI	

本课程不涉及这些指令

RISC-V汇编模拟器RARS

□ <https://github.com/TheThirdOne/rars>



The screenshot displays the RARS (RISC-V Assembler and Simulator) interface. The main window shows the assembly code for a program named 'bottles.s'. The code implements a loop that prints a message about beer bottles, decrements a counter, and loops back until the counter reaches zero. The code is as follows:

```

1  .globl main
2  .data
3  template:
4      .string "%d bottles of beer of the wall. %d bottles of beer. Take one down pass it around. %d bottles of beer on the wall\n"
5
6  .text
7  main:
8      # Save state to free up s0, s1, and ra for use
9
10     addi    sp, sp, -16
11     sw      s1, 4(sp)
12     sw      s0, 8(sp)
13     sw      ra, 12(sp)
14
15     # Call printf(template, x, x, x-1) for 0 < x < 100
16     li      s0, 99
17
18 loop:
19     mv      a1, s0
20     mv      a2, s0
21     addi    s0, s0, -1
22     mv      a3, s0
23     la      a0, template
24     call    printf
25     bnez    s0, loop
26
27     # Load state
28     lw      ra, 12(sp)
29     lw      s0, 8(sp)
30     lw      s1, 4(sp)
31     addi    sp, sp, 16
32
33     # Exit (93) with code 0
34     li      a0, 0
35     li      a7, 93
36     ecall
37     ebreak
38
39
40

```

The right panel shows the Register File with the following registers and values:

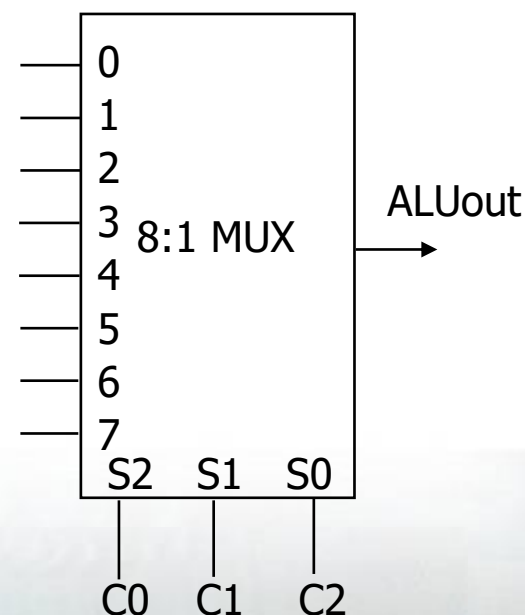
Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7ffffefc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x10010071
t1	6	0x00000000
t2	7	0x00000025
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x10010054
a2	12	0x0000001d
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x0000005d
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000030
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc	32	0x00400054

The bottom panel shows the output of the program, which is a repeating sequence of the message: "99 bottles of beer of the wall. 99 bottles of beer. Take one down pass it around. 98 bottles of beer on the wall".

1. 算术逻辑单元ALU

- 多功能函数模块
 - ❑ 3个控制信号表明所需进行的操作, funct3
 - ❑ 2 组数据输入作为操作数, rs1, rs2/SignExtImm
 - ❑ 1 组输出信号作为结果, ALUout

C0	C1	C2	Function	Comments
0	0	0	ADD/SUB	Addition/Subtraction
0	0	1	SLL	Shift Left Logic
0	1	0	SLT	Set Little
0	1	1	SLTU	Set Little Unsigned
1	0	0	A XOR B	logical XOR
1	0	1	SRL/SRA	Shift Right Logic/Arithmetic
1	1	0	A + B	logical OR
1	1	1	A • B	logical AND



RISCVfpga ALU Verilog描述

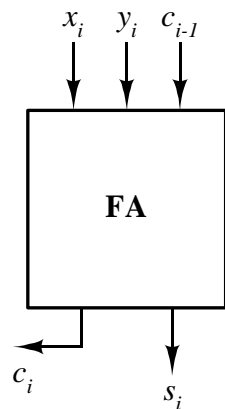
// In lab4, you must redesign this module using logic gate!!!
assign addresult = alucontrol[3] ? (srca - srcb) : (srca + srcb);

```
always @*  
begin  
    case (alucontrol[2:0])  
        3'b000: aluresult = addresult;           //Add/SUB  
        3'b001: aluresult = srca << shamt;       //SLL  
        3'b010: aluresult = (srca < srcb) ? 1 : 0; //SLT  
        3'b011: aluresult = {(WIDTH){1'bx}};    //SLTU  
        3'b100: aluresult = srca ^ srcb;         //XOR  
        3'b101: aluresult = {{1'b0}, srca[WIDTH-1:1]}; //SRL/SRA, only shift by 1 in this lab  
// 3'b101: aluresult = alucontrol[3] ? (srca >>> shamt) : (srca >> shamt); // ok in 40MHz  
        3'b110: aluresult = srca | srcb;         //OR  
        3'b111: aluresult = srca & srcb;         //AND  
        default: aluresult = {(WIDTH){1'bx}};  
    endcase  
end
```

加法器设计

$$s_i = x_i \oplus y_i \oplus c_{i-1}$$

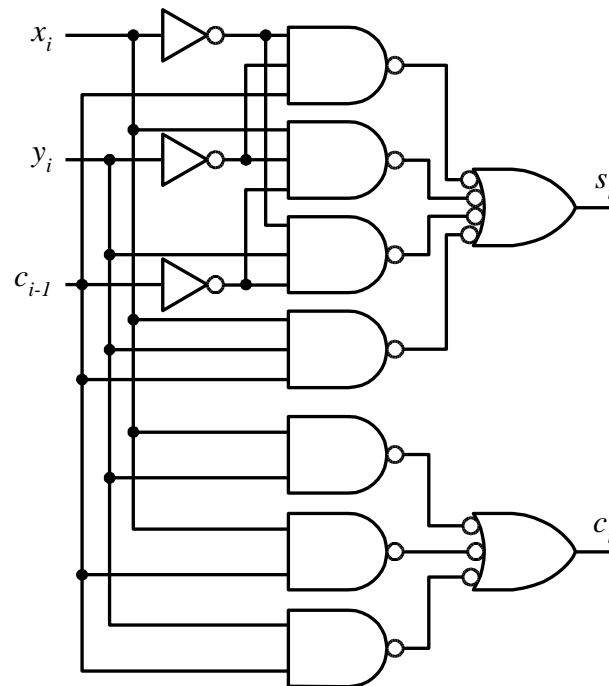
$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$



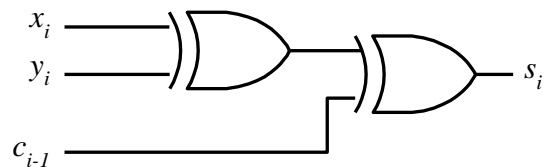
(d)

x_i	y_i	c_{i-1}	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(e)



(f)

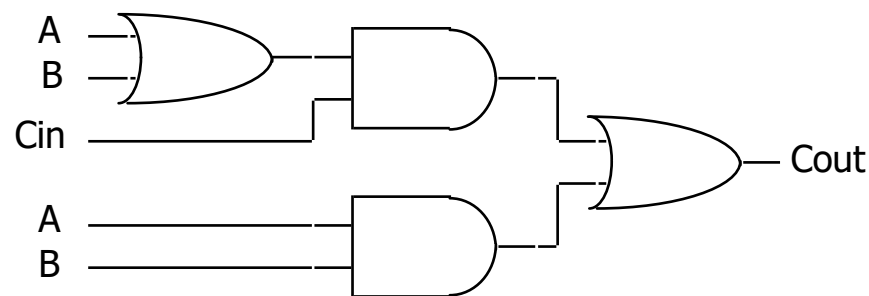
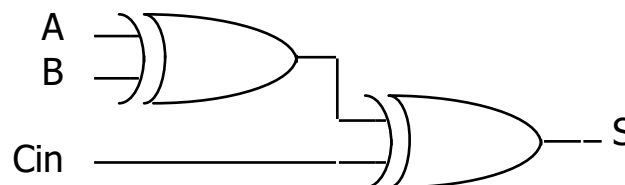


(g)

全加器实现

□ 标准实现

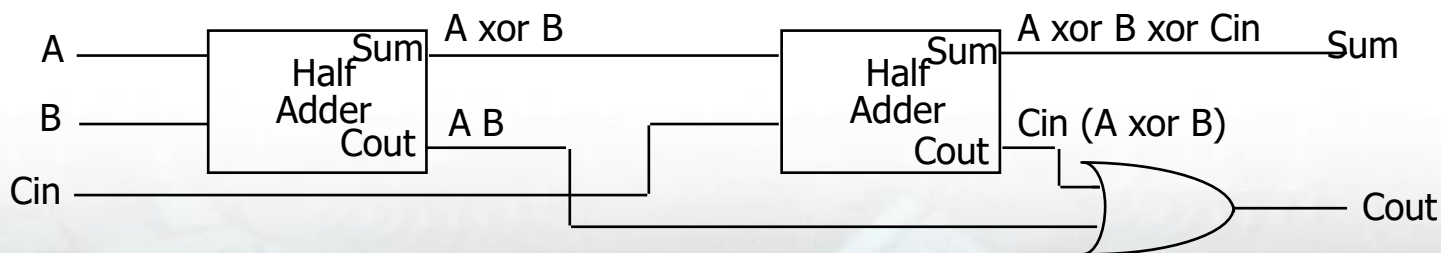
- 6 个门
- 2 XORs, 2 ANDs, 2 ORs



□ 另外一种实现

- 5 gates
- 半加器是 1 XOR 和 1 AND
- 2 XORs, 2 ANDs, 1 OR

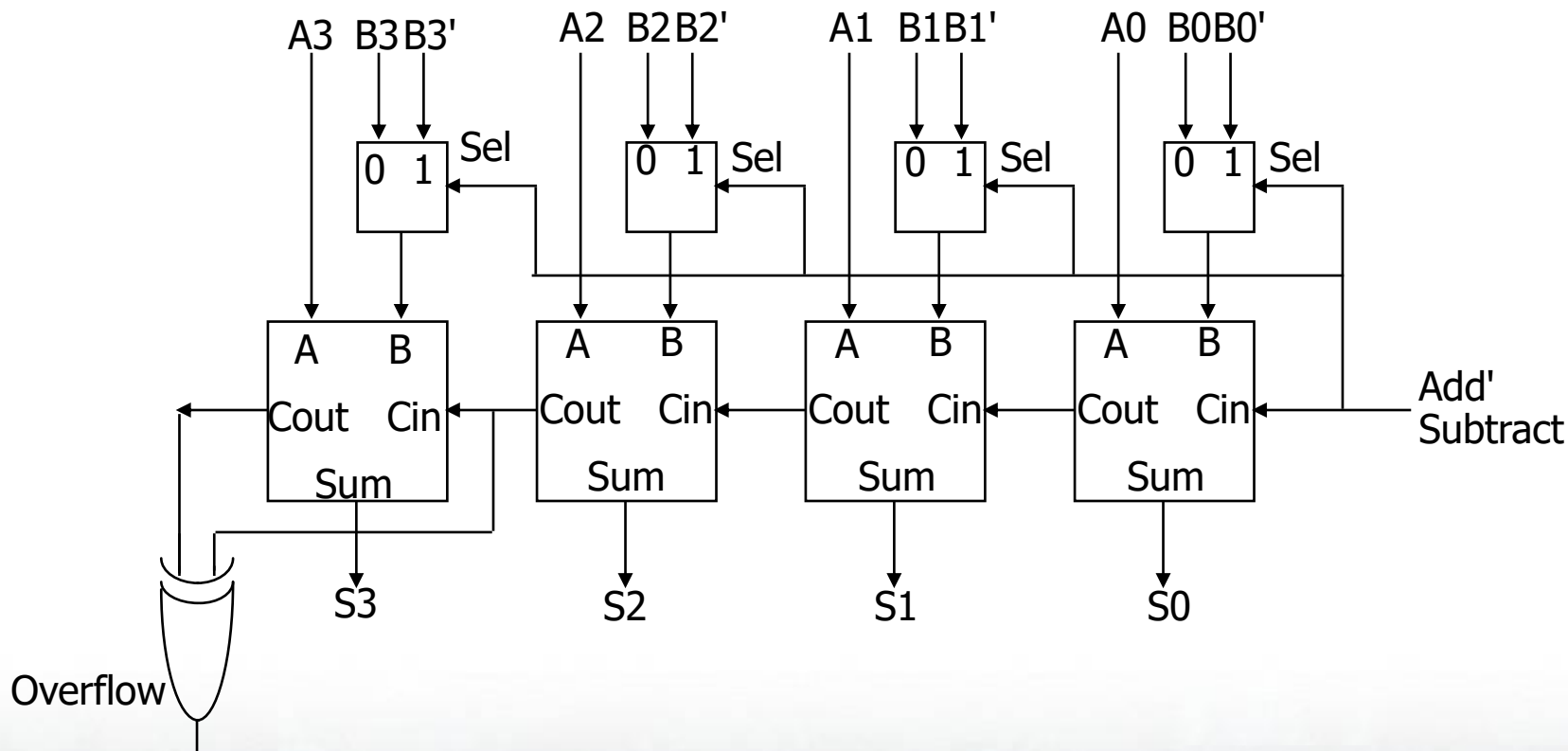
$$\text{Cout} = A B + \text{Cin} (A \text{ xor } B) = A B + B \text{Cin} + A \text{Cin}$$



加法器/减法器Adder/subtractor

□ 使用加法器实现减法器（补码表示）

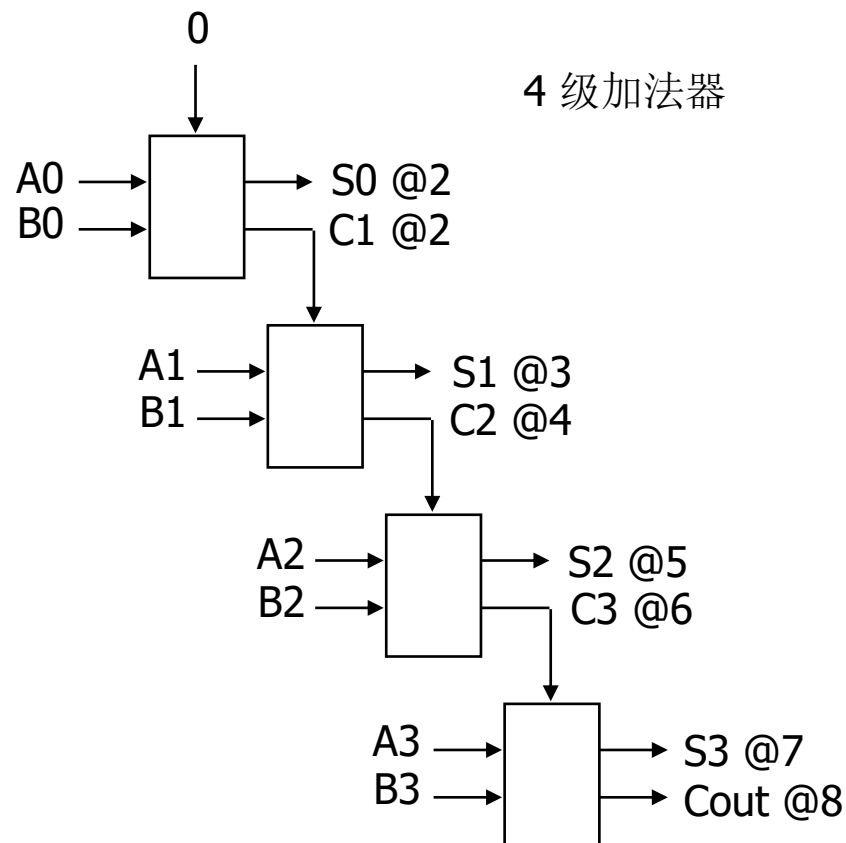
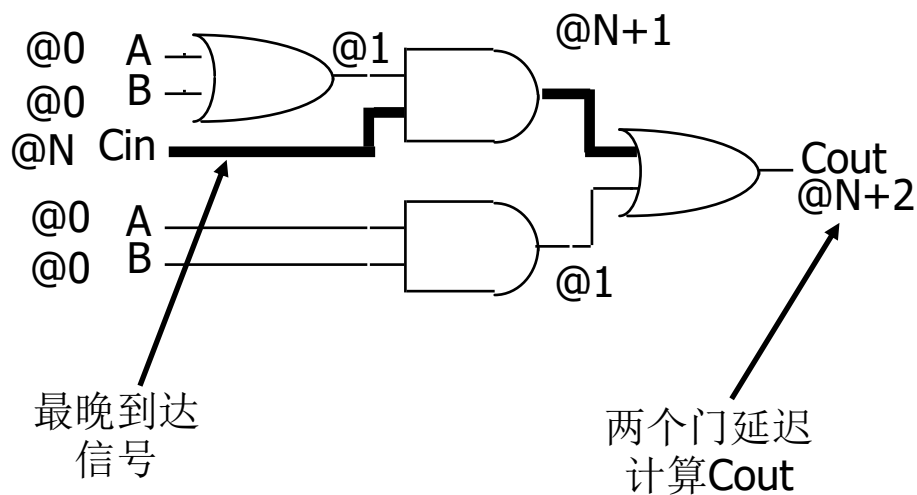
- $A - B = A + (-B) = A + B' + 1$
- 控制信号选择加B 或 B的补



行波进位加法 (Ripple-carry adders)

关键路径延时(Critical delay)

- 从低位到高位传播进位



超前进位加法器(Carry-lookahead logic)

- 进位产生因子Carry generate: $G_i = A_i B_i$
 - 当 $A = B = 1$, 必然产生进位
- 进位传播因子Carry propagate: $P_i = A_i \text{ xor } B_i$
 - 当 $P_i = 1$ 时, carry-in 等于 carry-out
- Sum and Cout 可以用产生因子和传播因子表示:
 - $S_i = A_i \text{ xor } B_i \text{ xor } C_i$
 $= P_i \text{ xor } C_i$
 - $C_{i+1} = A_i B_i + A_i C_i + B_i C_i$
 $= A_i B_i + C_i (A_i + B_i)$
 $= A_i B_i + C_i (A_i \text{ xor } B_i)$
 $= G_i + C_i P_i$

超前进位逻辑（续）

□ 每一个进位逻辑重新用进位因子和传播因子表示：

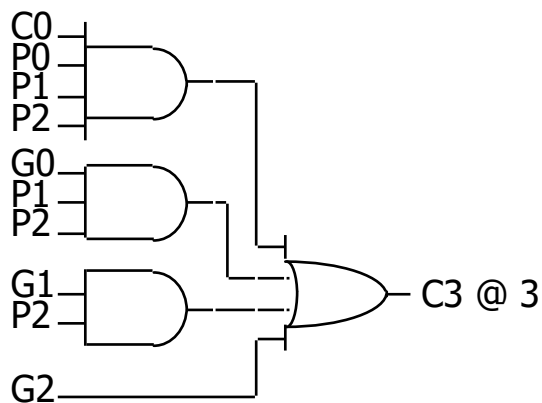
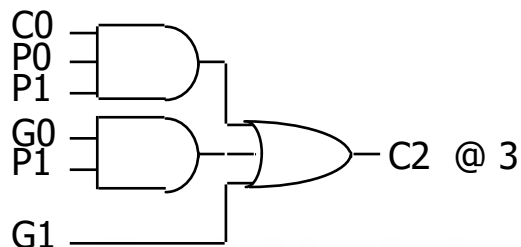
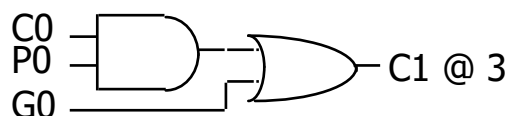
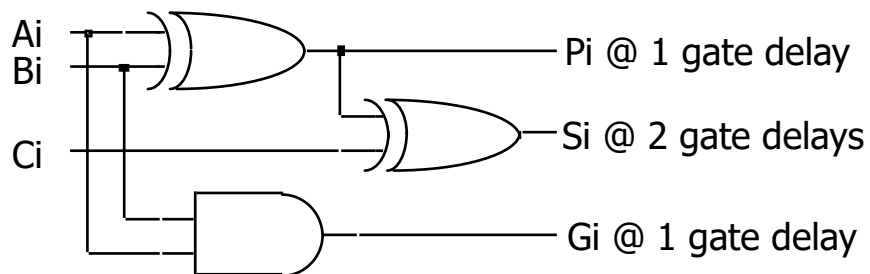
- $C_1 = G_0 + P_0 C_0$
- $C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$
- $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$
- $C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$

□ 每一个进位的布尔函数可用两级逻辑实现：

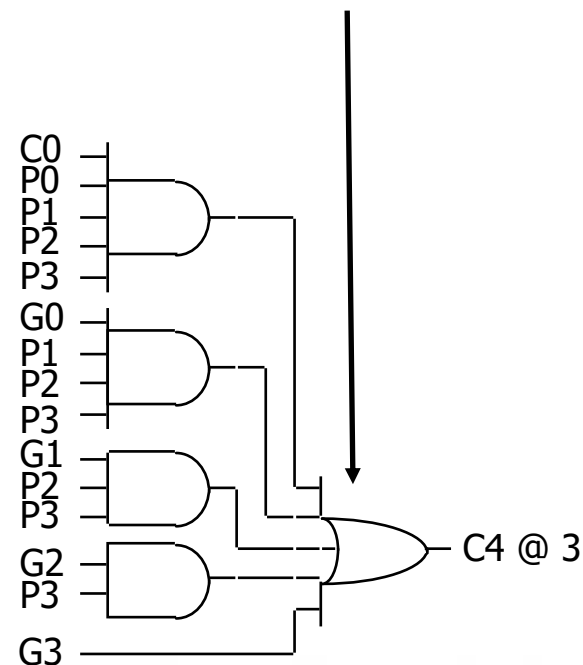
- 所有输入信号都直接起源于原始的数据输入信号，而不是起源于中间进位
- 并行的计算每一位结果sum

先行进位链实现

包含传播因子和产生因子的加法器



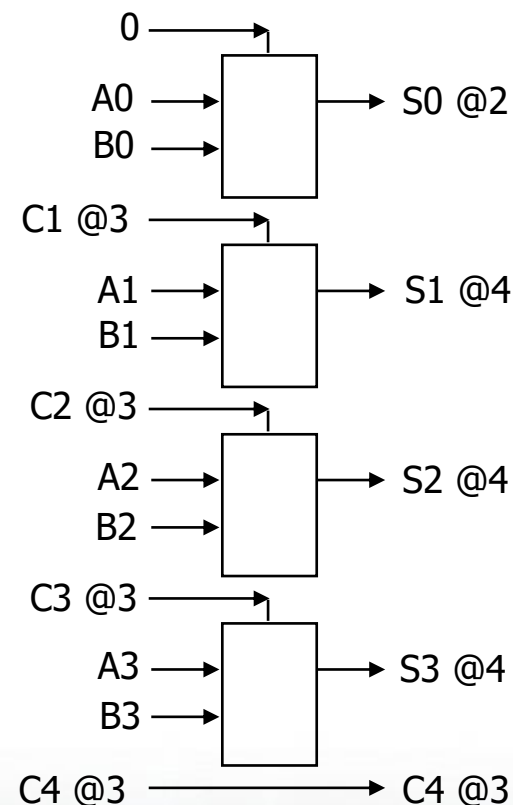
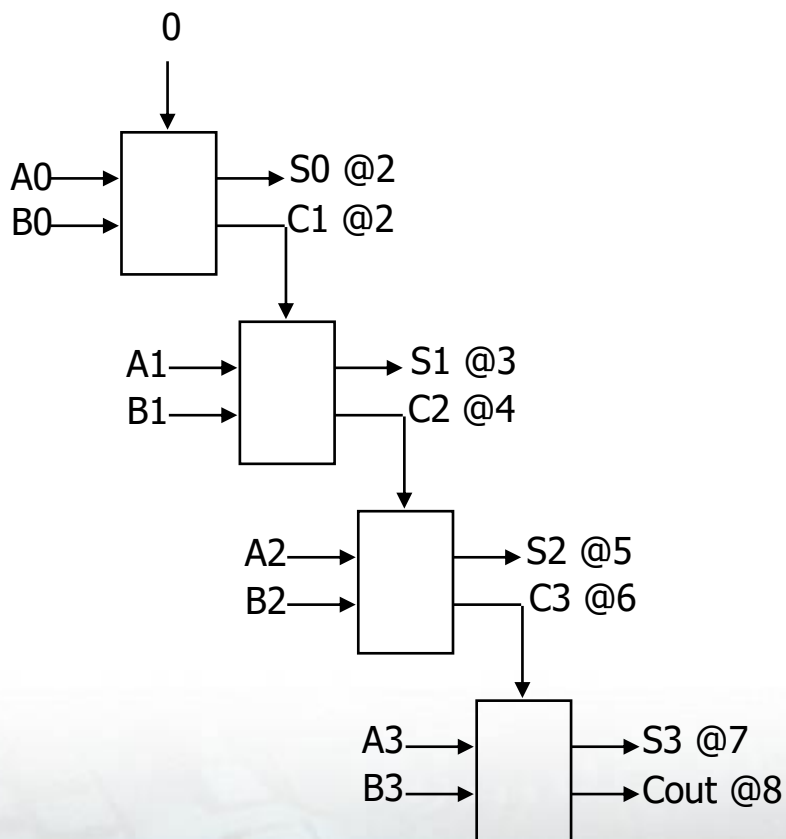
进位逻辑的复杂度增加了



先行进位链的实现（续）

□ 先行进位链逻辑独立的产生所有进位

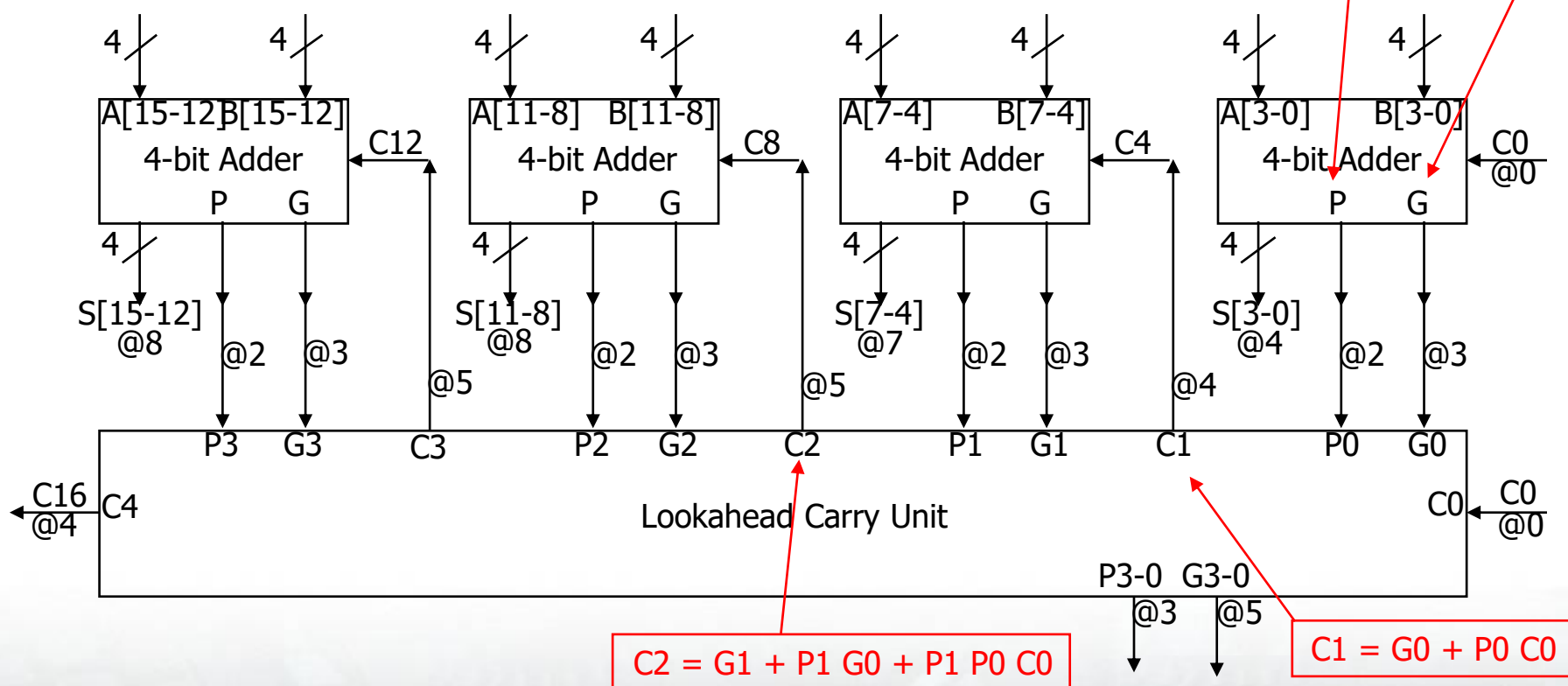
- Sum可以并行的快速计算出来
- 当然也增加了逻辑门数量



级联先行进位链的先行进位加法器

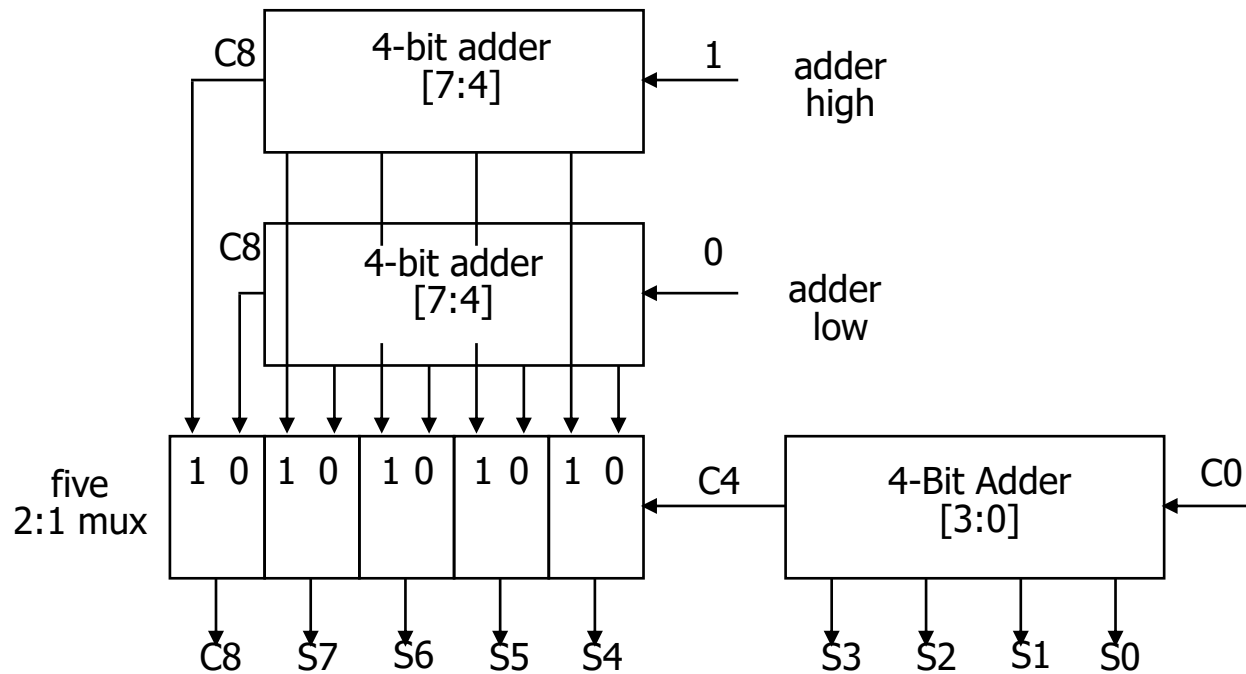
16位先行进位加法器

- 4个4位加法器出生内部先行进位
- 第二级先行进位模块扩展到16位先行进位逻辑

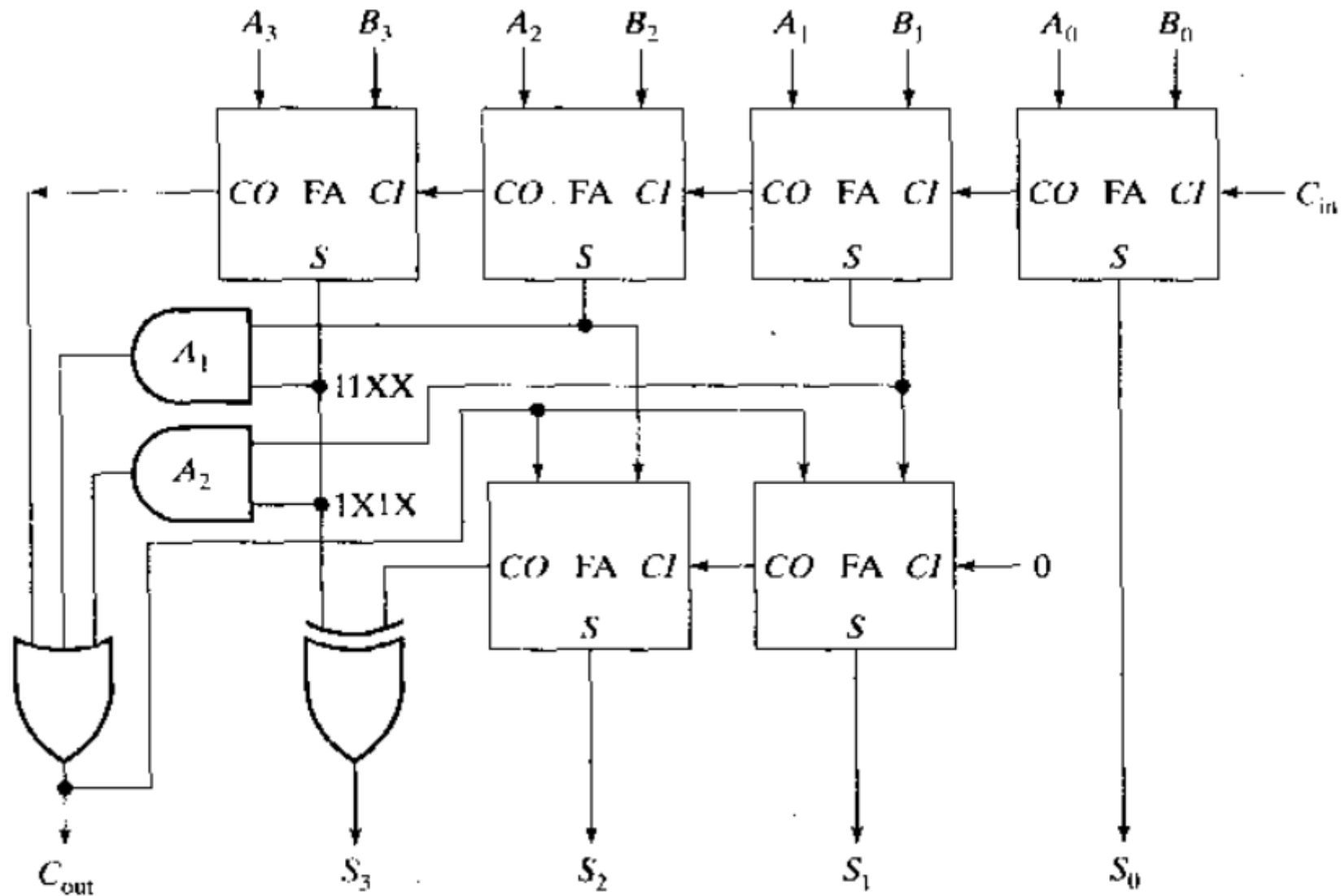


进位选择加法器 Carry-select adder

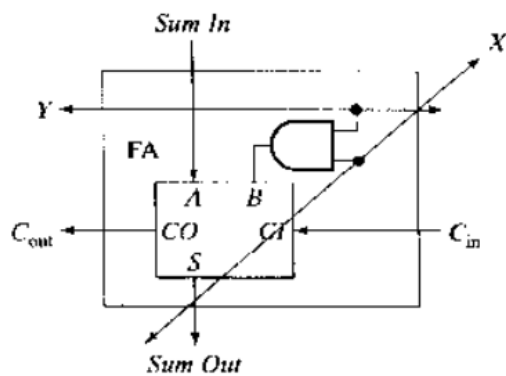
- 采用冗余的硬件来加快进位产生



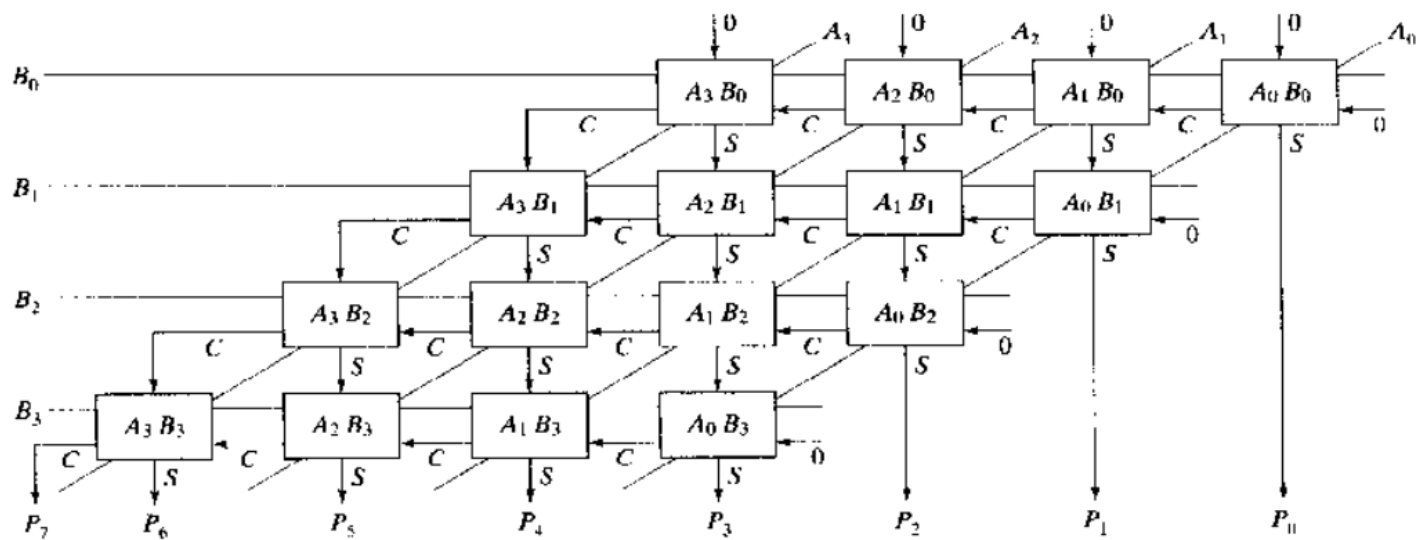
BCD加法器



组合乘法器



(a) 基本构建模块



(b) 4×4组合乘法器

2. 存储部件

□ 寄存器堆 Register File

- 32个通用寄存器x0~x31
- x0硬连线为0，写无效
- 寄存器堆端口：2读，1写

□ 指令存储器

- 只读存储器ROM
- 端口：地址和读数据

□ 数据存储器

- 随机访问存储器RAM
- 端口：地址，读数据，写数据，写使能信号

指令只读存储器imem.v

```
module imem(input  [5:0]  a,
            output [31:0] rd);

    reg [31:0] RAM[63:0];

    initial
        begin
            $readmemh("memfile.dat",RAM);
        end

    assign rd = RAM[a]; // word aligned
endmodule
```

数据随机访问存储器dmem.v

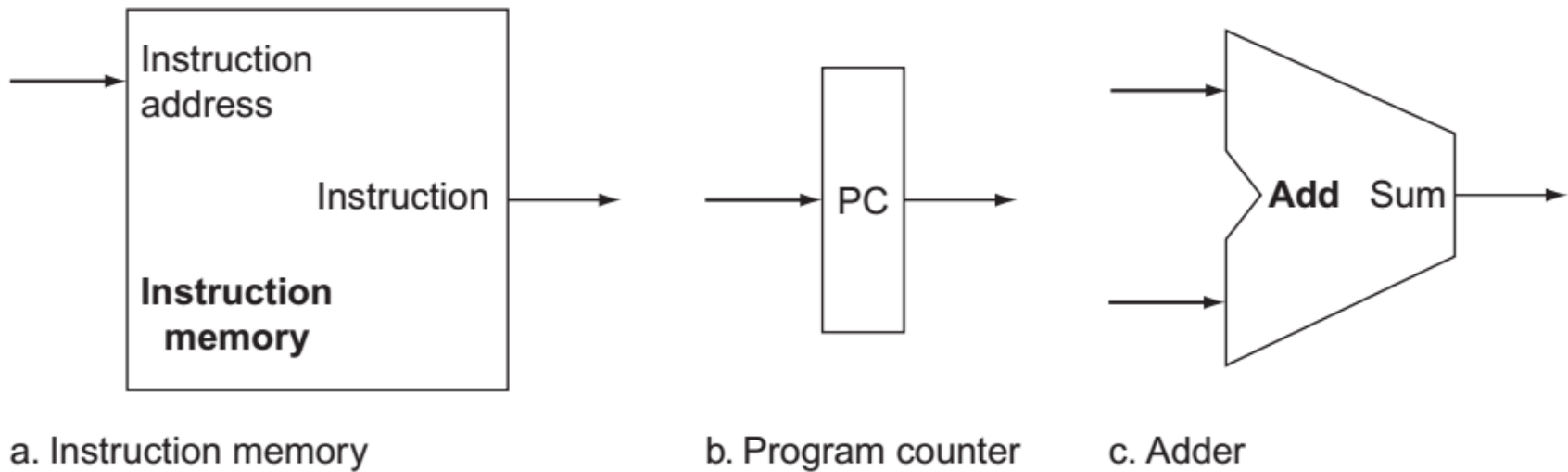
```
module dmem(input          clk, we,
            input  [31:0] a, wd,
            output [31:0] rd);

    reg [31:0] RAM[63:0];

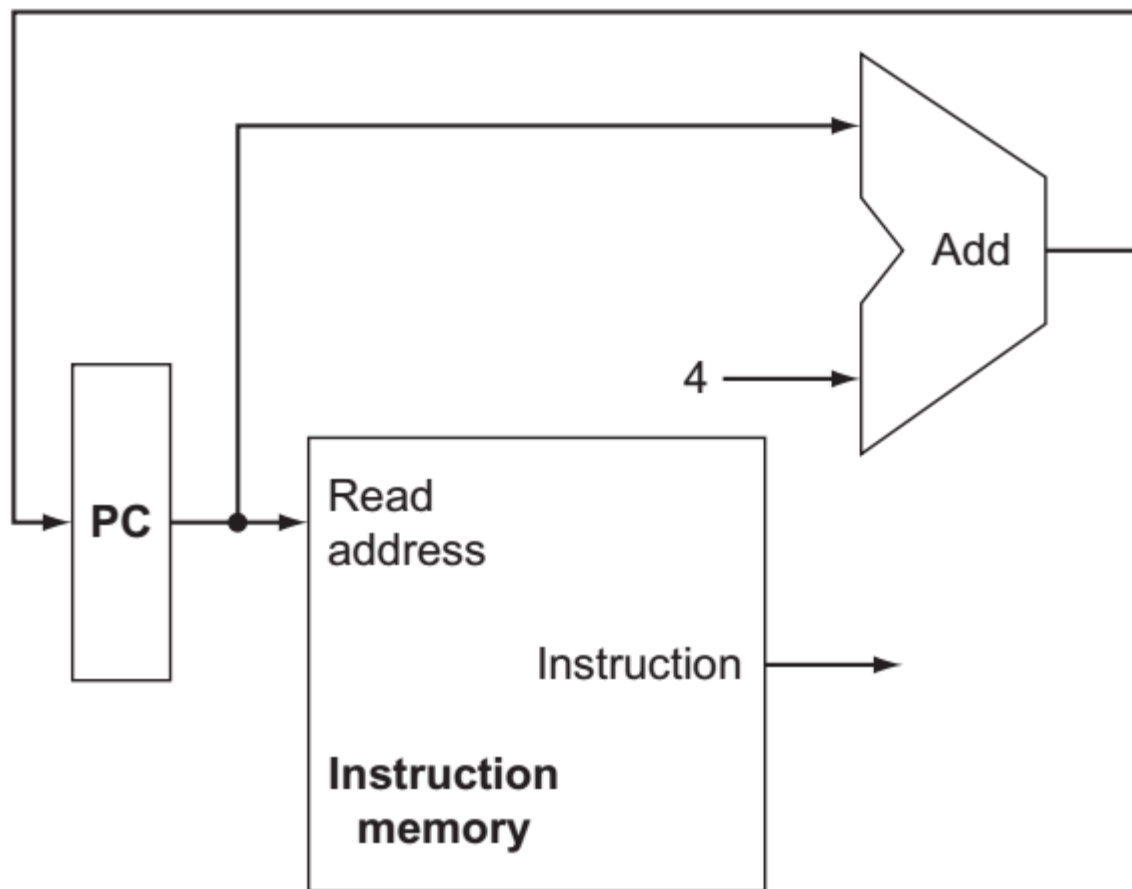
    assign rd = RAM[a[31:2]]; // word aligned

    always @(posedge clk)
        if (we)
            RAM[a[31:2]] <= wd;
endmodule
```

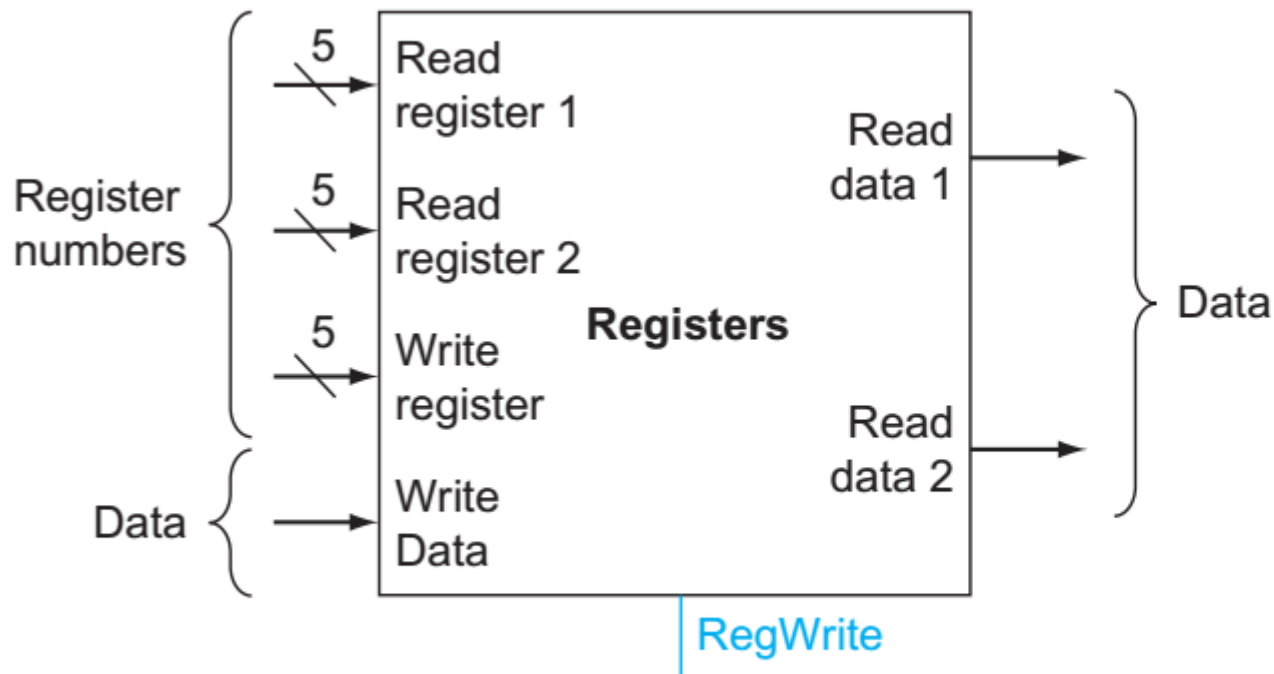
3 RISCVfpga单周期处理器设计



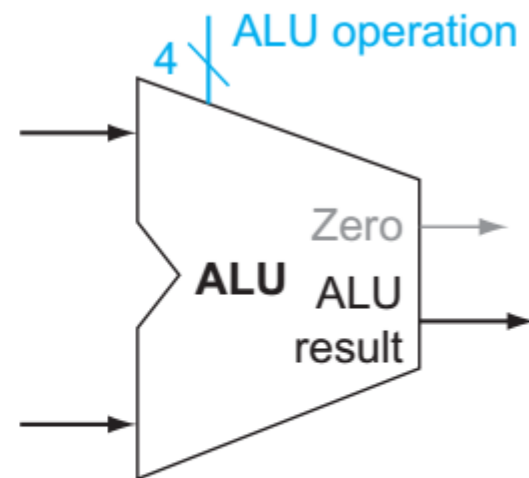
取指过程和后继指令地址生成



寄存器堆和ALU部件



a. Registers



b. ALU

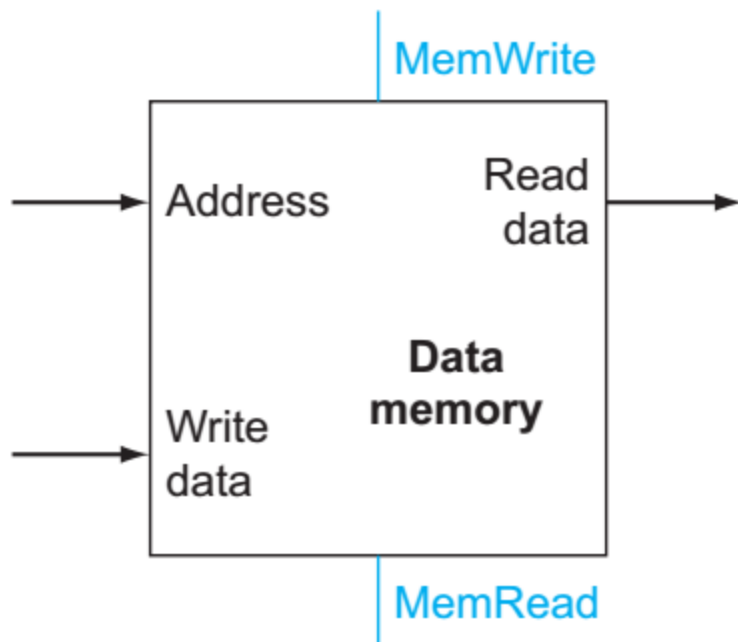
寄存器堆Verilog

```
module regfile(input      clk,
               input      we3,
               input [4:0] ra1,
               input [4:0] ra2,
               input [4:0] wa3,
               input [31:0] wd3,
               output [31:0] rd1,
               output [31:0] rd2);
    reg [31:0] rf[31:0];

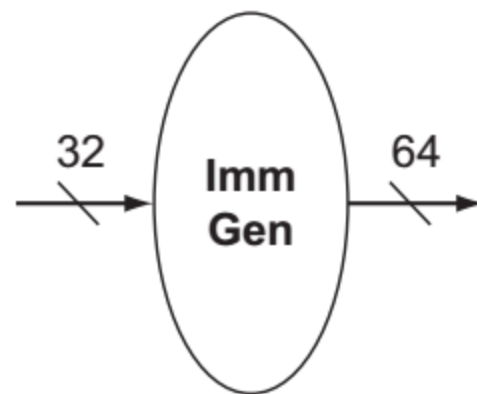
    always @(posedge clk)
        if (we3) rf[wa3] <= wd3;

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

数据存储单元和立即数生成单元

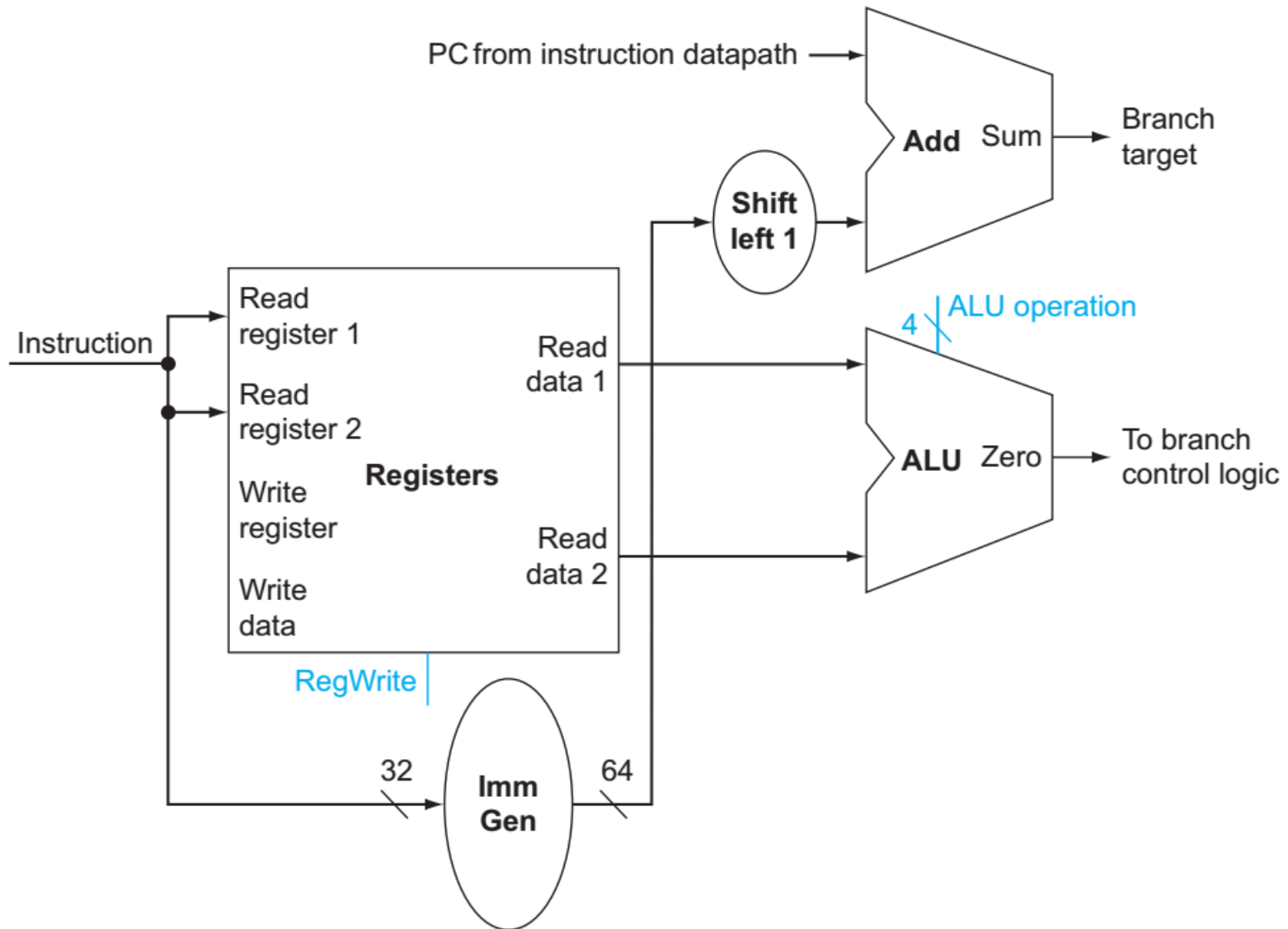


a. Data memory unit

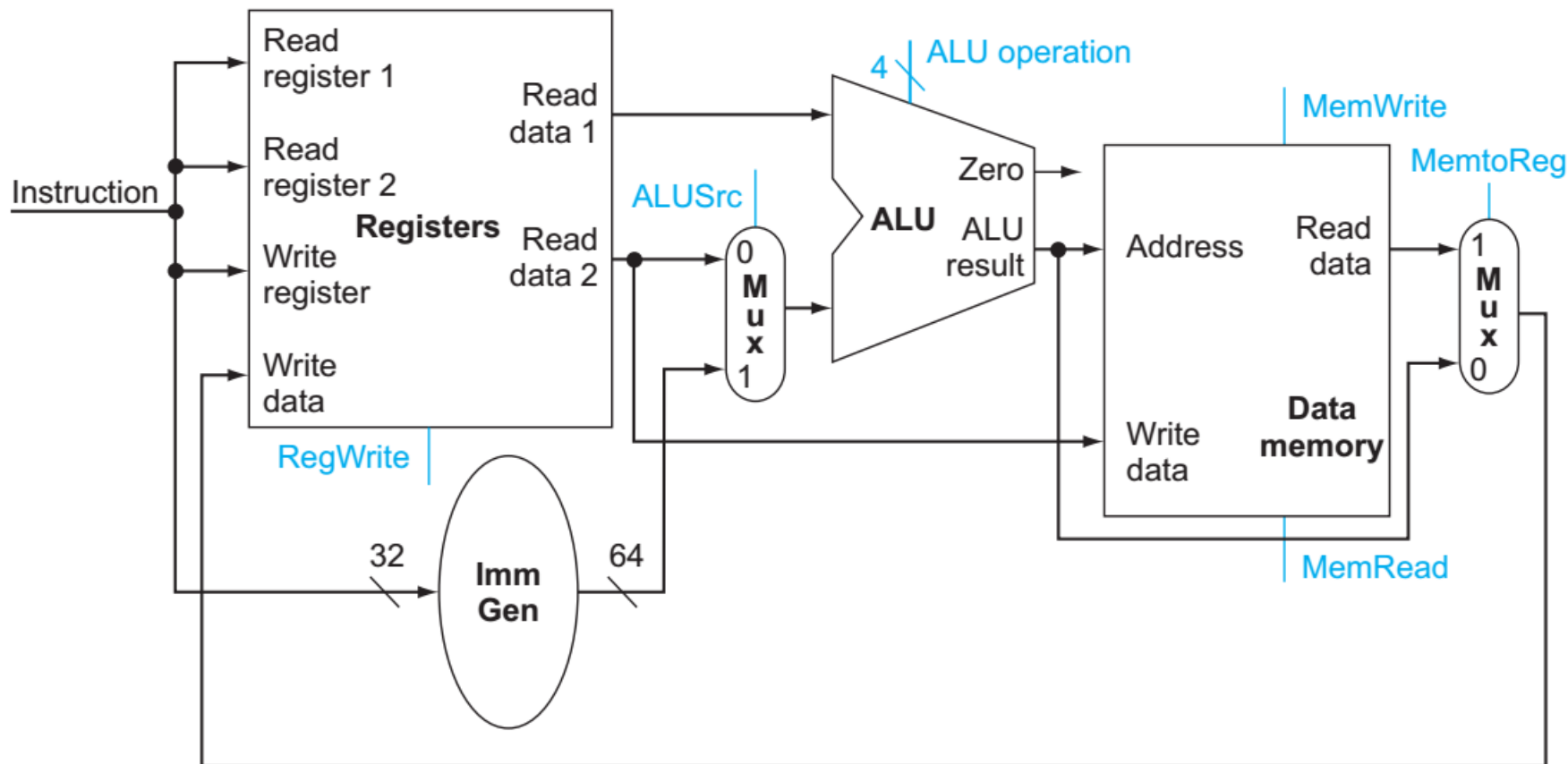


b. Immediate generation unit

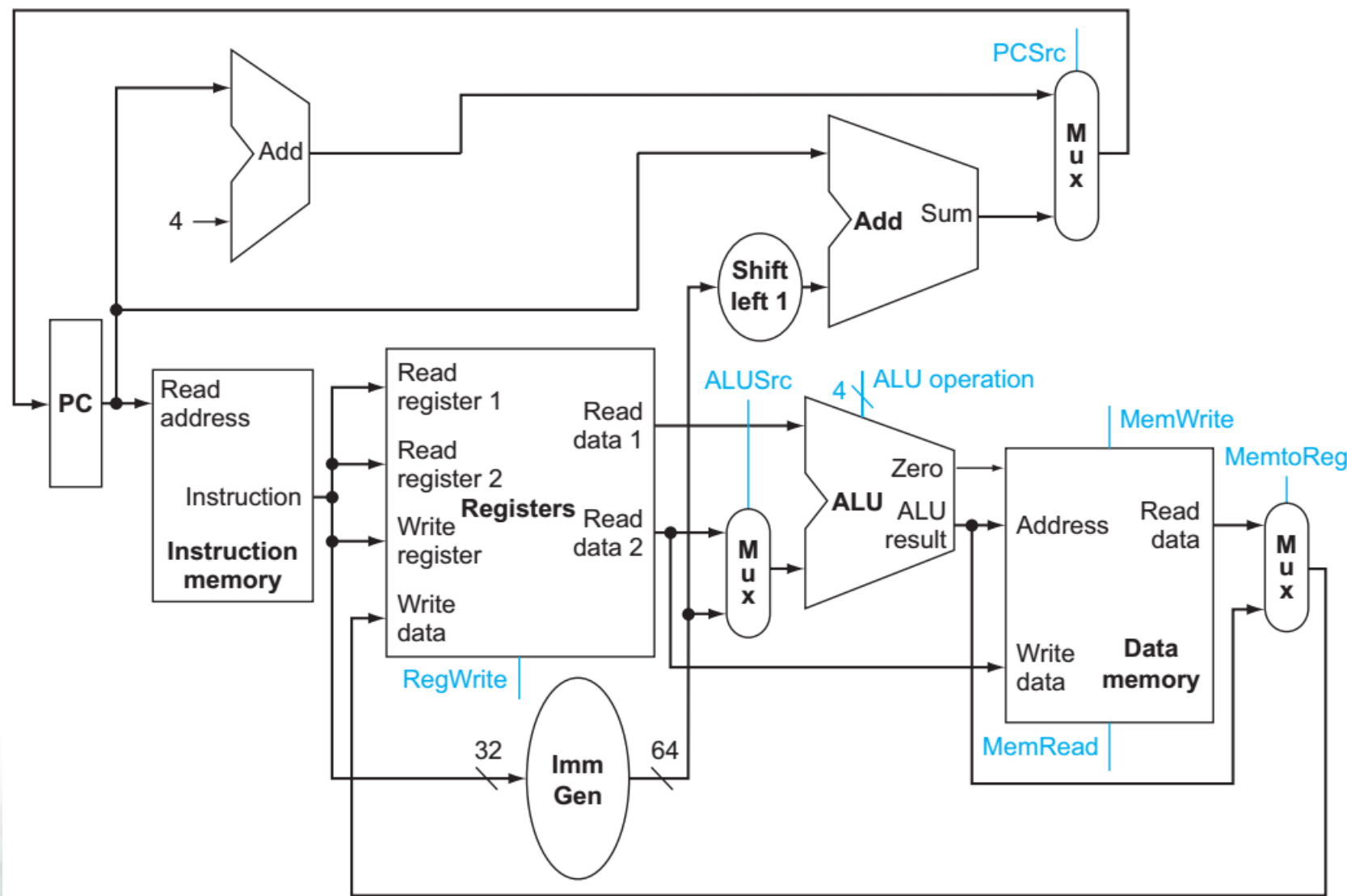
数据通路：branch指令



数据通路 —— 访存指令和R-type指令



数据通路



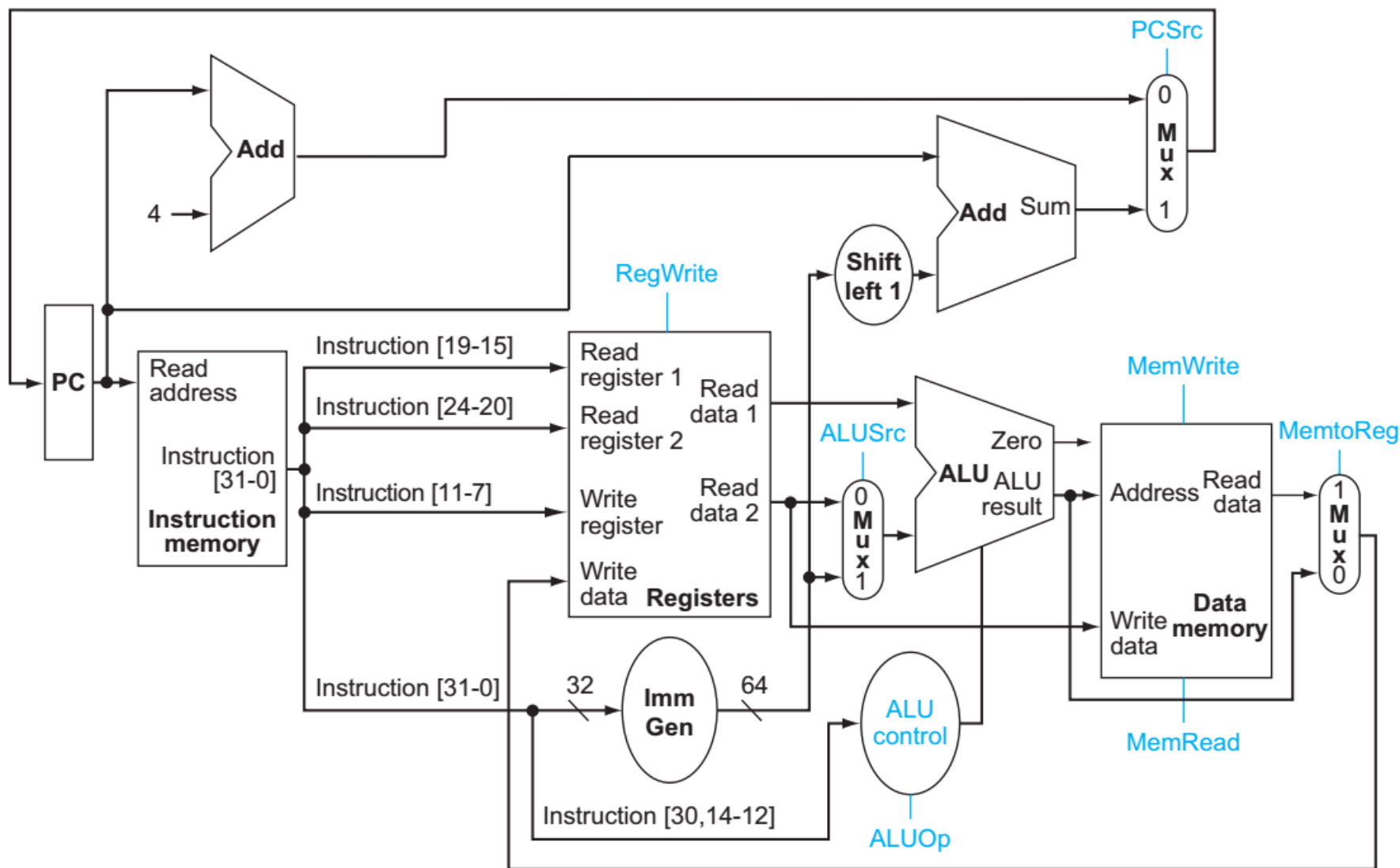
ALU控制信号设计

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action
ld	00	load doubleword	XXXXXXX	XXX	add
sd	00	store doubleword	XXXXXXX	XXX	add
beq	01	branch if equal	XXXXXXX	XXX	subtract
R-type	10	add	0000000	000	add
R-type	10	sub	0100000	000	subtract
R-type	10	and	0000000	111	AND
R-type	10	or	0000000	110	OR

立即数生成: ImmGen

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2			rs1		funct3		rd			opcode		R-type	
imm[11:0]						rs1		funct3		rd			opcode		I-type		
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type	
imm[12]		imm[10:5]		rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]										rd			opcode		U-type		
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type		

RISCV单周期处理器数据通路

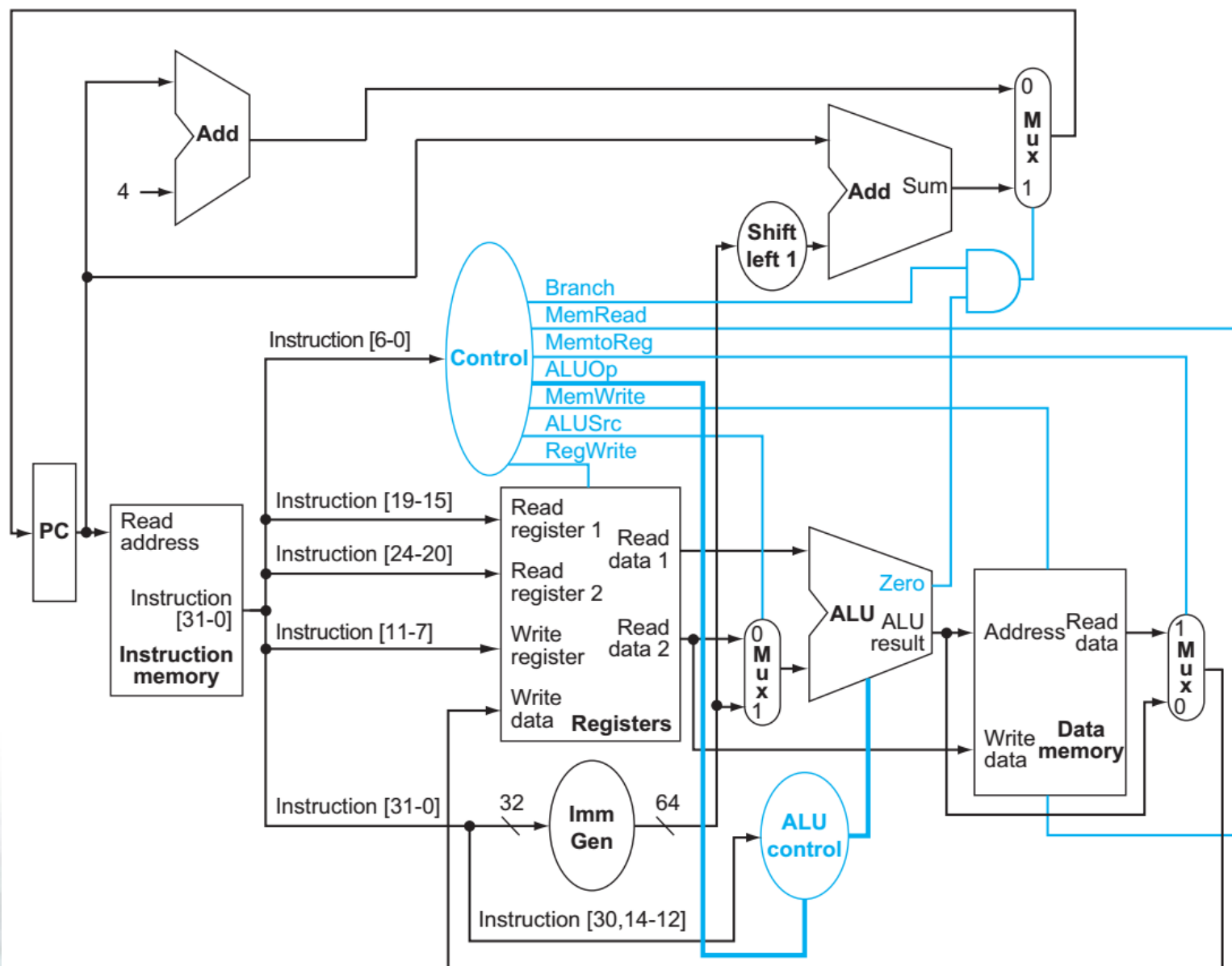


控制器：控制信号

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of $PC + 4$.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.



单周期处理器：数据通路+控制器



主控制器

Instruction	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	0	0	1	0	0	0	1	0
I-type	0	0	1	0	0	0	1	1
lw	1	1	1	0	1	0	0	0
sw	1	X	0	1	0	0	0	0
beq	0	X	0	0	0	1	0	1

```
assign {regwrite, luitoreg, alusrc, branch, memwrite, memtoreg, jump, aluop} = controls;
```

```
always @( * )
```

```
case(op)
```

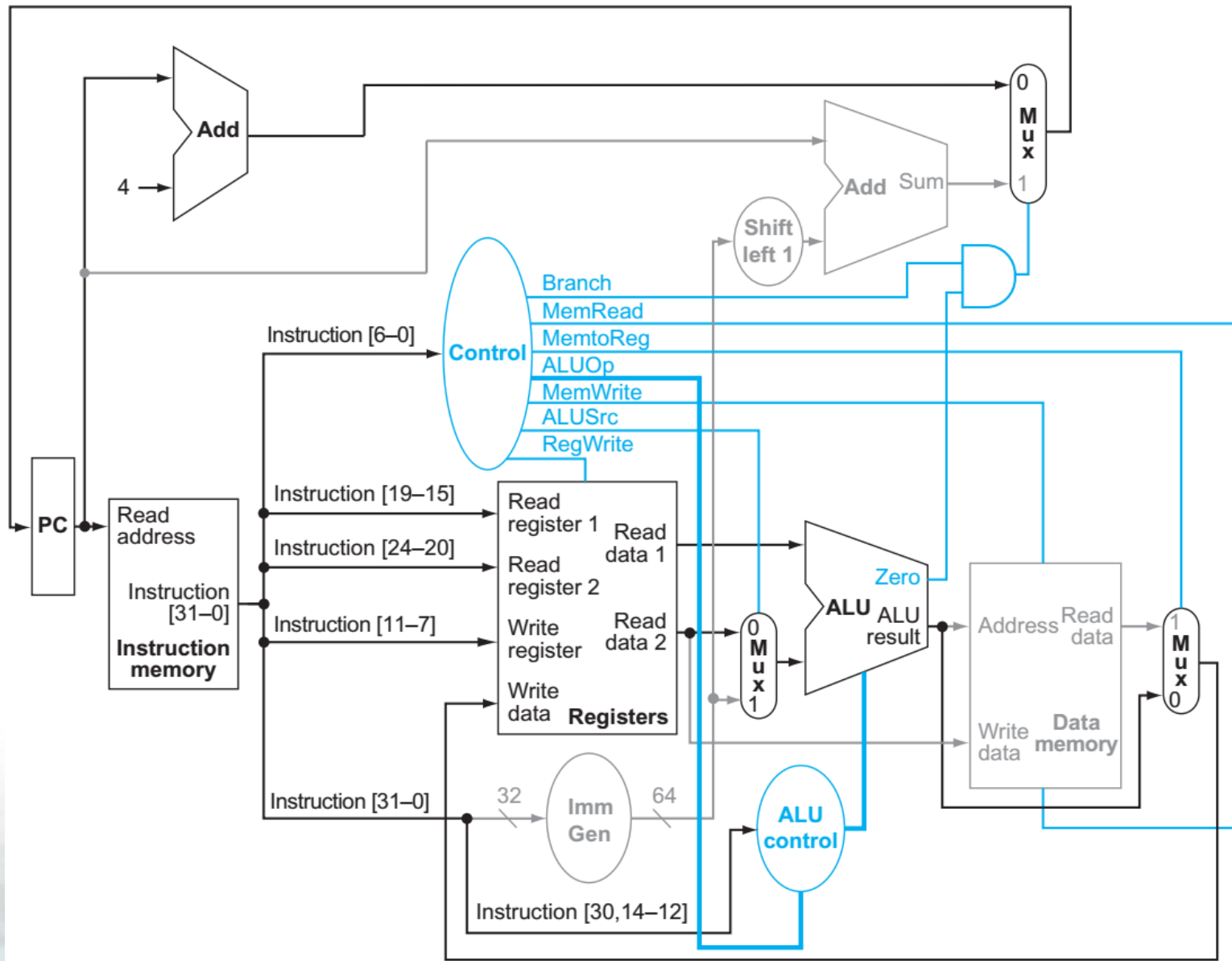
```
7'b0110111: controls <= 9'b1100000_00;           //LUI
7'b0110011: controls <= 9'b1000000_10;           //R-TYP
7'b0000011: controls <= 9'b1010010_00;           //LW
7'b0100011: controls <= 9'b0010100_00;           //SW
7'b1100011: controls <= 9'b0001000_01;           //BEQ
7'b0010011: controls <= 9'b1010000_11;           //I-TYPE
7'b1101111: controls <= 9'b0000001_00;           //JAL
default:    controls <= 9'bxxxxxx_xxx;           //Undefined
```

```
endcase
```

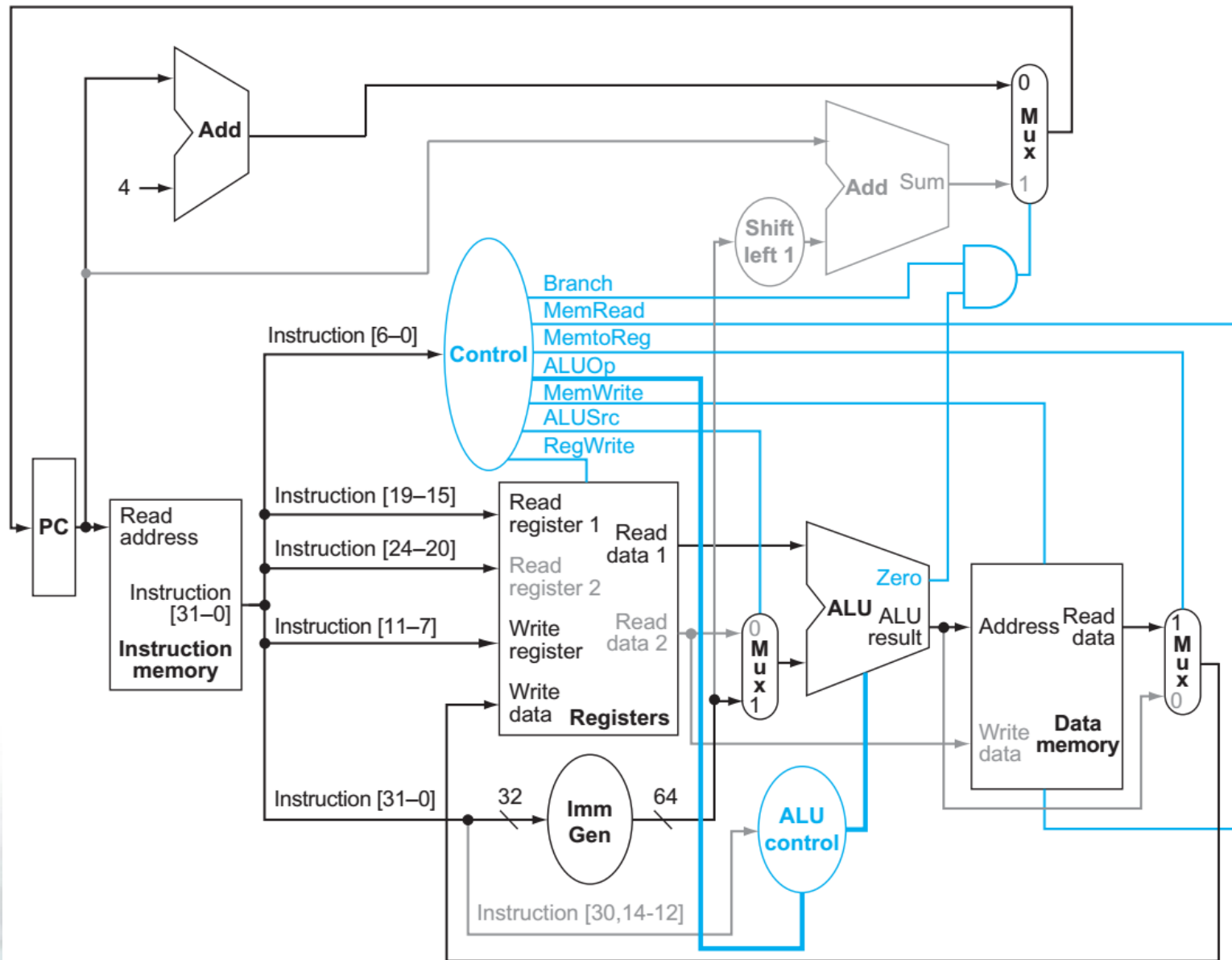
ALU控制器

```
module aludec(input      [6:0] funct7,  
               input      [2:0] funct3,  
               input      [1:0] aluop,  
               output reg [3:0] alucontrol);  
  
always @( * )  
    case(aluop)  
        2'b00: alucontrol <= 4'b0000;           // add  
        2'b01: alucontrol <= 4'b1000;           // sub  
        2'b10: alucontrol <= {funct7[5], funct3}; // R-TYP  
        2'b11: alucontrol <= {{1'b0}, funct3};  // I-TYP  
        default: alucontrol <= 4'hx;  
    endcase  
endmodule
```

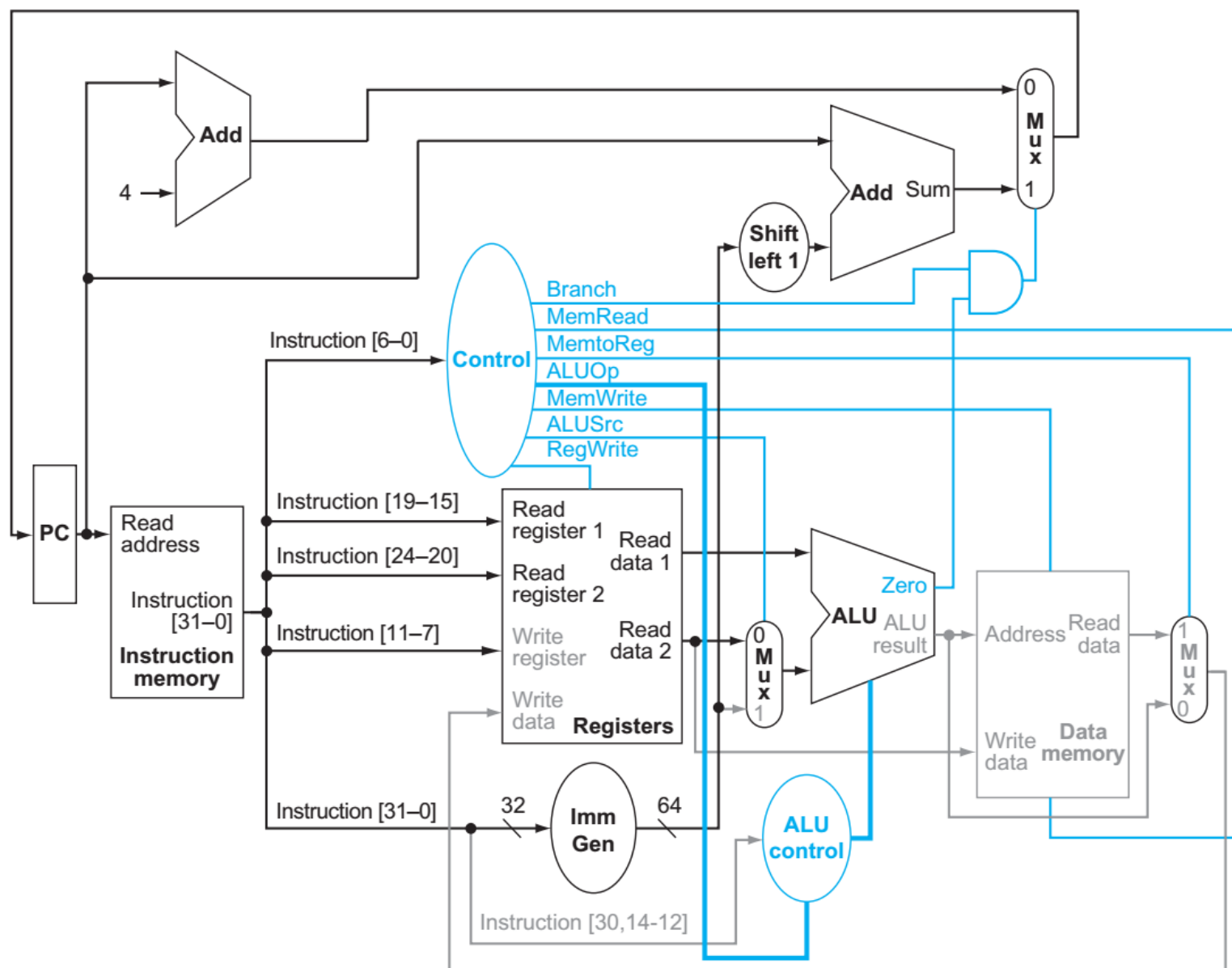
R-type Instruction



Load Instruction



Branch-if-equal (其它转移和跳转指令)



Lab6: 基于SingleRISCVfpga的简单计算器

□ 内存映射输入输出（MMIO）：

- 0x0000-0x0FFF：指令空间
- 0x1000-0x17FF：数据空间
- 0x7F00-0x7FFF：I/O空间

□ MMIO寄存器地址：

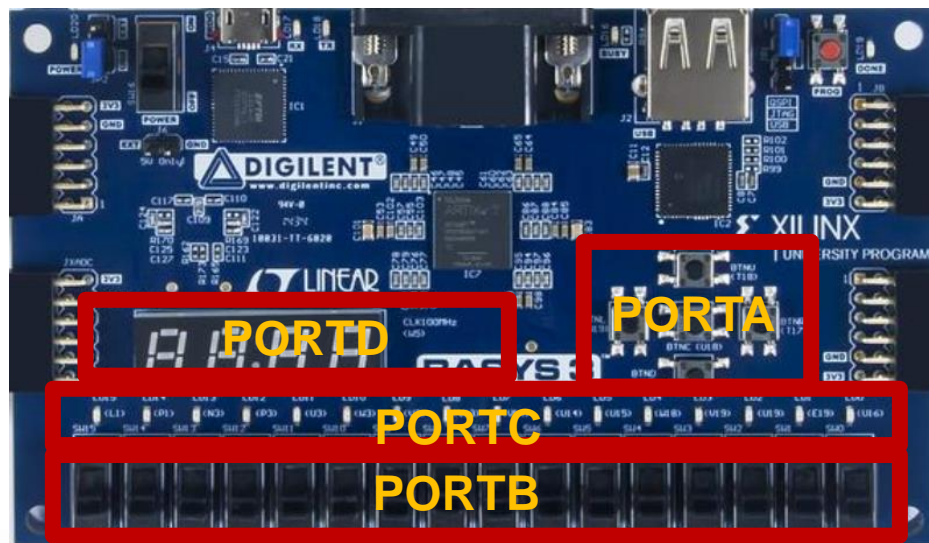
- PORTA: 0x7F00, 只读, 寄存器低4位保存4个按钮通过同步器后的输入值, 从高到低BntD(=), BntL(加), BntC(减), BntD(乘)
- PORTB: 0x7F10, 只读, 寄存器中保存16个开关SW[15:0]经过BCD2Bin转换后的16位二进制。
- PORTC: 0x7F20, 读写, 寄存器中保存16位二进制, 后经过Bin2BCD的值通过LED模块显示。
- PORTD: 0x7FFC, 读写, 寄存器中保存值的低16位二进制输出LD[15:0], 驱动16个发光二极管。

□ Verilog模拟虚拟停机寄存器的I/O地址

- PORTD/HALT: 0x7FFC, 读写, 当数据地址是0x7FFC, 同时写数据为0x1时, Verilog模拟停止。

最终课程项目：基于处理器的简单计算器

- ❑ 十进制输入、十进制输出
- ❑ 加法、减法、乘法功能
- ❑ Verilog设计



4. RISC-V单周期处理器的IO端口dmem_io.v

```
assign we_dmem = (((a >= 32'h00001000) && (a < 32'h00001800)) ? 1 : 0) & we;
assign we_portc = ( a == 32'h00007f20 );
assign we_portd = ( a == 32'h00007ffc );
assign rdata_RAM = RAM[a[5:2]];
// add bcd2bin module here, from input sw to output portb
// add bin2bcd and led7seg module here, from input portc_reg to output seg/an
always @(a, porta, portb, portc_reg, portd_reg, rdata_RAM)
begin
    if ( a == 32'h0000ff00 )
        begin rdata = {{28{1'b0}}, porta}; end
    else if ( a == 32'h0000ff10 )
        begin rdata = {{16{1'b0}}, portb}; end
    else if ( a == 32'h0000ff20 )
        begin rdata = {{16{1'b0}}, portc_reg}; end
    else if ( a == 32'h0000fffc )
        begin rdata = {{16{1'b0}}, portd_reg}; end
    else if ((a >= 32'h00001000) && (a < 32'h00001800))
        begin rdata = rdata_RAM; end // word aligned
    else rdata = 0;
end
```

5. imem_fpga在vivado中的生成

Flow Navigator PROJECT MANAGER - singleriscv

PROJECT MANAGER

- Settings
- Add Sources
- Language Tools
- IP Catalog

IP INTEGRATOR

- Create Block Design
- Open Block Design
- Generate Block Design

SIMULATION

- Run Simulation

RTL ANALYSIS

- Open Elaborated Design

SYNTHESIS

- Run Synthesis
- Open Synthesized Design

IMPLEMENTATION

- Run Implementation
- Open Implemented Design

PROGRAM AND DEBUG

- Generate Bitstream
- Open Hardware Manager

Sources

- Design Sources
- Constraints
- Simulation Sources
 - sim_1
- Utility Sources

Hierarchy Libraries Compile Order

IP Properties

Distributed Memory Generator

Version: 8.0 (Rev. 13)

Description: The LogiCORE Xilinx Distributed Memory Generator creates area and performance optimized ROM blocks, single, dual and simple dual port distributed memories for Xilinx FPGAs. The core supersedes the previously released LogiCORE Distributed Memory core. Use this core in all new designs for supported families wherever a distributed memory is required.

Status: Production

License: Included

Project Summary IP Catalog

Cores Interfaces

Search: Q-rom (10 matches)

Name	AXI4	Status	License	VLNV
Vivado Repository				
Alliance Partners				
OmniTek				
Chroma Down-Sampler	AXI4, AXI4-Stream	Production	Purchase	omnitek.tv:ip:omni_chroma_down:0.0
Chroma Up-Sampler	AXI4, AXI4-Stream	Production	Purchase	omnitek.tv:ip:omni_chroma:0.0
Chroma Up-Sampler 420 to 422	AXI4, AXI4-Stream	Production	Purchase	omnitek.tv:ip:omni_chroma_420_to_422:0.0
Memories & Storage Elements				
RAMs & ROMs				
Distributed Memory Generator		Production	Included	xilinx.com:ip:dist_mem_gen:8.0
RAMs & ROMs & BRAM				
Block Memory Generator	AXI4	Production	Included	xilinx.com:ip:blk_mem_gen:8.4
Video & Image Processing				

Details

Name: Distributed Memory Generator

Version: 8.0 (Rev. 13)

Description: The LogiCORE Xilinx Distributed Memory Generator creates area and performance optimized ROM blocks, single, dual and simple dual FPGAs. The core supersedes the previously released LogiCORE Distributed Memory core. Use this core in all new designs for support memory is required.


Status: Production


License: Included




Tcl Console Messages Log Reports Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2019)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2019)

选择ROM类型

 Customize IP

Distributed Memory Generator (8.0)

 Documentation  IP Location  Switch to Defaults

☒ Show disabled ports

a[5:0]

d[31:0]

dpra[5:0]

clk

we

i_ce

qspo_ce

qdpo_ce

qdpo_clk

qspo_rst

qdpo_rst

qspo_srst

qdpo_srst

spo[31:0]

dpo[31:0]

qspo[31:0]

qdpo[31:0]

Component Nameimem_fpga

memory config

Port config

RST & Initialization

Options

Depth64[16 - 65536]

Data Width32[1 - 1024]

Memory Type

Memory Type

☒ ROM

☐ Single Port RAM

☐ Simple Dual Port RAM

☐ Dual Port RAM

Select the Memory Type

OK

Cancel

输入输出选择无寄存模式

Customize IP

Distributed Memory Generator (8.0)

Documentation IP Location Switch to Defaults

☒ Show disabled ports

Component Name:

memory config Port config RST & Initialization

Input Options

Input Options

☒ Non Registered ☐ Registered

☐ Input Clock Enable ☐ Qualify WE with L_CE

Dual Port Address

Dual Port Address

☒ Non Registered ☐ Registered

Output Options

Output Options

☒ Non Registered ☐ Registered ☐ Both

☐ Common Output CLK ☐ Single Port Output CE

☐ Common Output CE ☐ Dual Port Output CE

Pipelining Options

Pipeline Stages:

OK Cancel

Ports:

- a[5:0]
- d[31:0]
- dpra[5:0]
- clk
- we
- i_ce
- qspo_ce
- qdpo_ce
- qdpo_clk
- qspo_rst
- qdpo_rst
- qspo_srst
- qdpo_srst
- spo[31:0]
- dpo[31:0]
- qspo[31:0]
- qdpo[31:0]

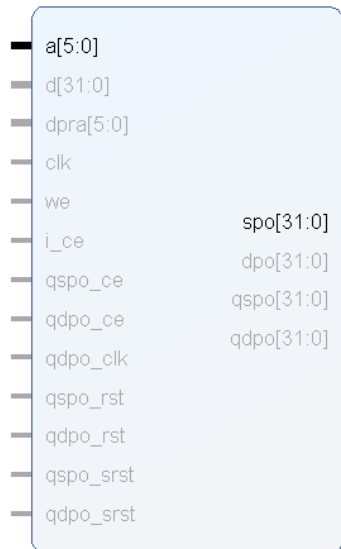
指定ROM中的汇编程序

Customize IP

Distributed Memory Generator (8.0)

[Documentation](#) [IP Location](#) [Switch to Defaults](#)

☒ Show disabled ports



Component Name

memory config | Port config | **RST & Initialization**

Load COE File

The initial memory content can be set by using a COE file. This will be passed to the core as a Memory Initialisation File (MIF).

Coefficients File



COE Options

Default Data :



Radix :



Reset Options

☐ Reset QSP0

☐ Reset QDPO

☐ Synchronous Reset QSP0

☐ Synchronous Reset QDPO

ce overrides

☒ CE Overrides Sync Controls

☐ Sync Controls Overrides CE

OK

Cancel