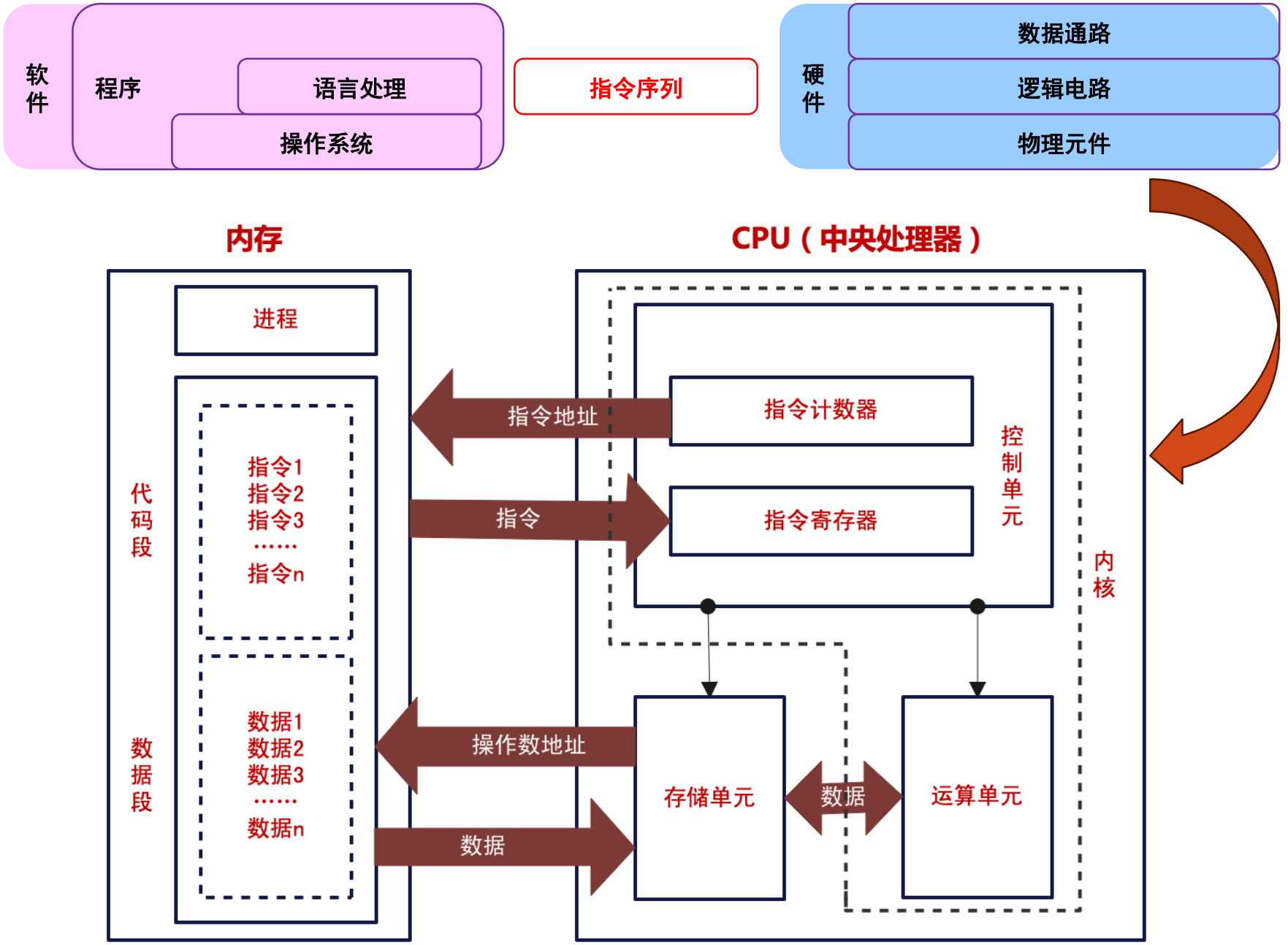
第九章 指令集结构

本章重点

- 指令集结构ISA
  - 概述
  - 算术/逻辑运算指令
  - 数据传送指令
  - 控制指令
- DLX指令处理
- C语言的数据类型与计算机的ISA

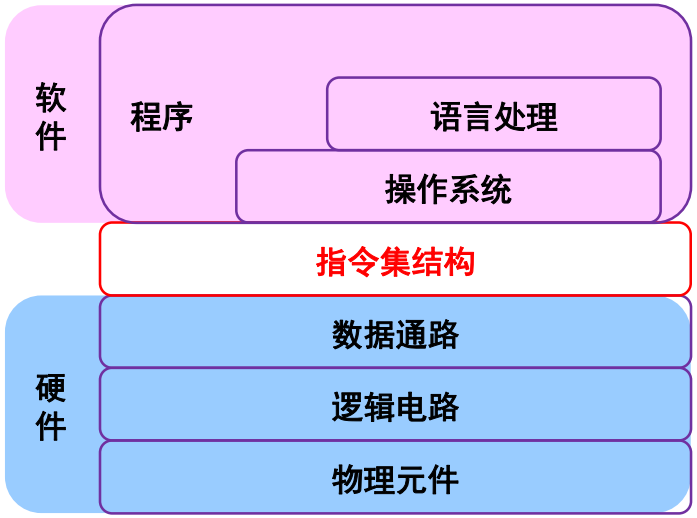
指令集结构ISA

- 概述
- 算术/逻辑运算指令
- 数据传送指令
- 控制指令


软件
程序
语言处理
操作系统
指令序列
硬件
数据通路
逻辑电路
物理元件
内存
CPU（中央处理器）
进程
代码段
指令1
指令2
指令3
……
指令n
数据段
数据1
数据2
数据3
……
数据n
指令地址
指令
操作数地址
数据
指令计数器
指令寄存器
控制单元
内核
存储单元
数据
运算单元

指令集结构

- Instruction Set Architecture, ISA
- 定义计算机能执行的指令集合
- 计算机硬件和软件之间的接口
- 处理器（CPU）设计的第一步



ISA

DLX
算术/逻辑运算指令
数据传送指令
控制指令
浮点指令

DLX
二进制补码整数
（8/16/32位）
单/双精度浮点数
（32/64位）

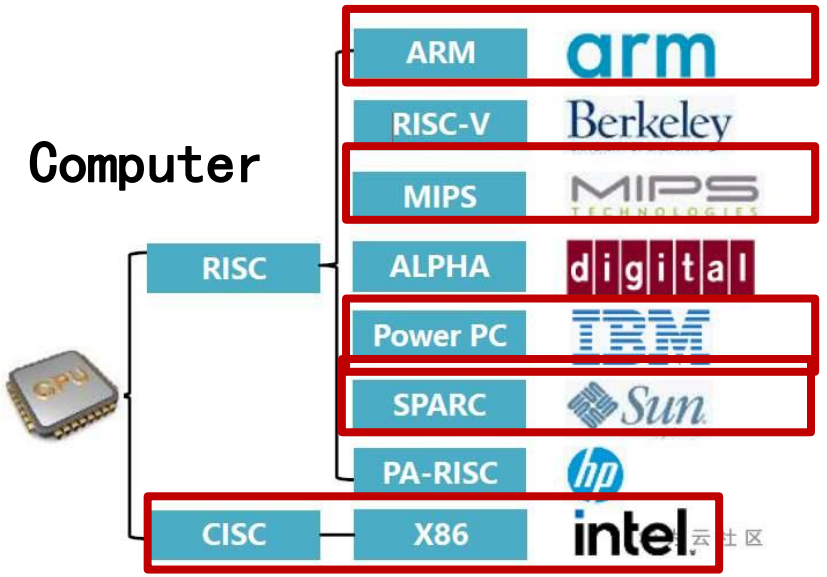
DLX
基址+偏移量

- 计算机能够执行的指令集合
  - 操作码：让计算机执行的**操作**
  - 操作数：每一步操作所需的**数据**
    - “**数据类型**”：操作数在计算机中的表示方式
    - “**寻址模式**”：如何计算操作数在存储器中的地址
  - 存储器
    - **地址空间**：计算机存储单元的数量（2^n）
    - **寻址能力**：每个存储单元存储信息的能力(m位)
  - 寄存器集

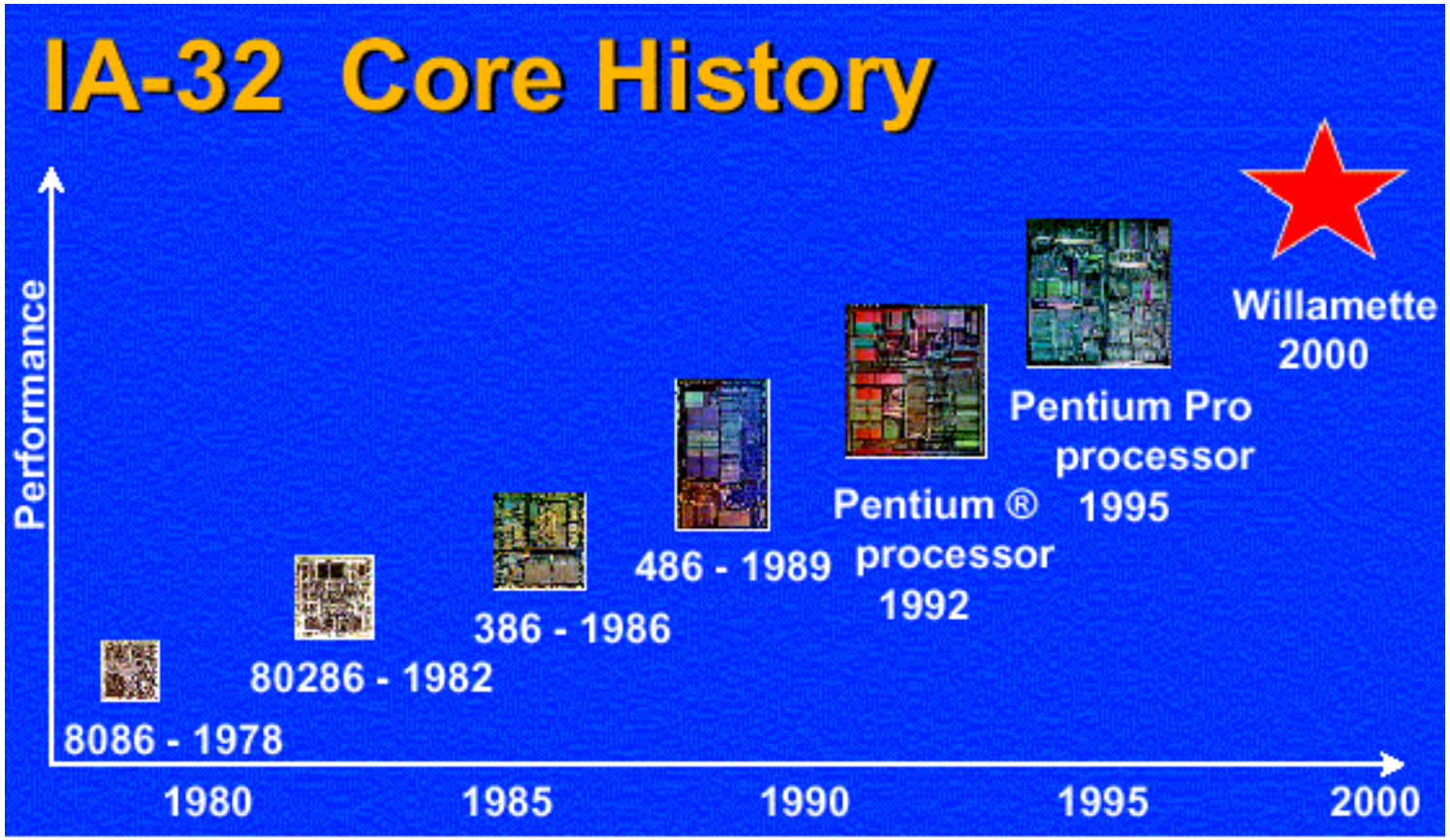
不同的指令集结构规定的操作、操作数数据类型和寻址模式等是不同的。

ISA分类

- CISC
  - Complex Instruction Set Computer
  - 复杂指令集计算机
  - 功能强大的复杂指令
  - 开发程序比较容易
  - 指令执行效率较低
- RISC
  - Reduced Instruction Set Computer
  - 精简指令集计算机
  - 指令集较小
  - 开发程序有所欠缺
  - 指令执行效率比CISC高



ISA示例： IA-32

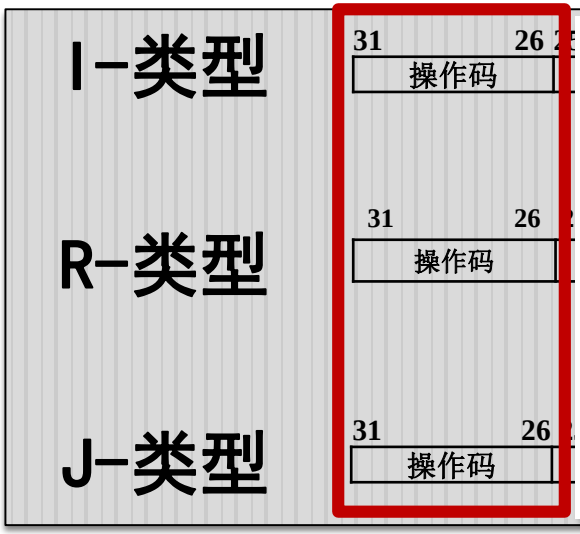

IA-32 Core History
Performance
8086 - 1978
80286 - 1982
386 - 1986
486 - 1989
Pentium ® processor 1992
Pentium Pro processor 1995
Willamette 2000
1980
1985
1990
1995
2000

ISA示例：MIPS

- MIPS指令集
  - 1986年，斯坦福大学Hennessy教授
  - 操作、数据类型和寻址模式要少得多
  - 主要应用领域：工作站等计算机平台，如龙芯
- 简化版本—DLX指令集
  - MIPS指令集为基础
  - 《计算机系统结构：一种定量的方法（第二版）》
  - DLX子集： MIPS进一步简化版（裁剪和扩充）

DLX指令操作类型

- 由指令的[31:26]位定义，64种指令类型
- R类型
  - 指令的[31:26]位为000000
  - [5:0]位定义了函数，有64种可能的函数
- 只定义了91条指令



- 按照功能分为四种类型
  - 算术/逻辑运算指令
    - 处理整数信息
  - 数据传送指令
    - 在存储器和寄存器之间传送数据
    - 在寄存器/存储器和输入/输出设备之间传送数据
  - 控制指令
    - 改变指令被执行的顺序
  - 浮点指令
    - 处理浮点数信息

DLX 指令子集格式

	31 26	25 21	20 16	15 11	10 6	5 0
ADD	000000	SR1	SR2	DR	未用	000001
ADDI	000001	SR1	DR	Imm16		
SUB	000000	SR1	SR2	DR	未用	000011
SUBI	000011	SR1	DR	Imm16		
AND	000000	SR1	SR2	DR	未用	001001
ANDI	001001	SR1	DR	Imm16		
OR	000000	SR1	SR2	DR	未用	001010
ORI	001010	SR1	DR	Imm16		
XOR	000000	SR1	SR2	DR	未用	001011
XORI	001011	SR1	DR	Imm16		
LHI	001100	未用	DR	Imm16		
SLL	000000	SR1	SR2	DR	未用	001101
SLLI	001101	SR1	DR	Imm16		
SRL	000000	SR1	SR2	DR	未用	001110
SRLI	001110	SR1	DR	Imm16		
SRA	000000	SR1	SR2	DR	未用	001111
SRAI	001111	SR1	DR	Imm16		
SLT	000000	SR1	SR2	DR	未用	010000
SLTI	010000	SR1	DR	Imm16		
SLE	000000	SR1	SR2	DR	未用	010010
SLEI	010010	SR1	DR	Imm16		
SEQ	000000	SR1	SR2	DR	未用	010100
SEQI	010100	SR1	DR	Imm16		
LB	010110	SR1	DR	Imm16		
SB	010111	SR1	DR	Imm16		
LW	011100	SR1	DR	Imm16		
SW	011101	SR1	DR	Imm16		
BEQZ	101000	SR1	未用	Imm16		
BNEZ	101001	SR1	未用	Imm16		
J	101100	PCOffset26				
JR	101101	SR1	未用	未用		
JAL	101110	PCOffset26				
JALR	101111	SR1	未用	未用		
TRAP	110000	Vector26				

指令集结构ISA

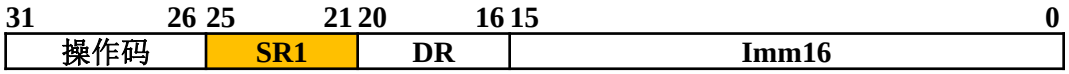
- 概述
- 算术/逻辑运算指令
- 数据传送指令
- 控制指令

算术/逻辑运算指令

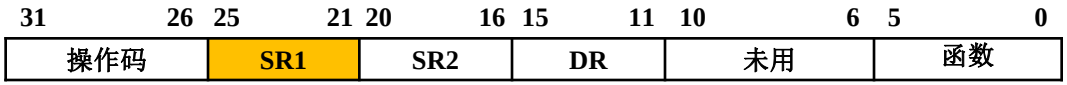
- 对整数进行处理
- 37个算术逻辑运算指令：加、减、乘、除、与、或、异或、移位、比较、加载高位立即数等
- 除加载高位立即数指令（LHI）外，其他运算指令执行的都是二元运算
  - 两个源操作数（即待运算的数据）
    - 来自通用寄存器或从指令中直接获得
  - 一个目标操作数（运算执行后的结果）
    - 存储于通用寄存器中

第一个源操作数

- 来自寄存器
  - 32个整数寄存器，5位编码标识
  - [25:21]，SR1
- I-类型

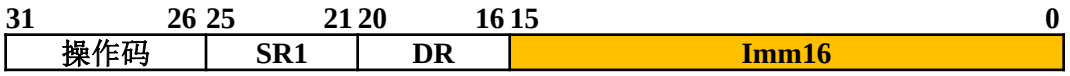


- R-类型

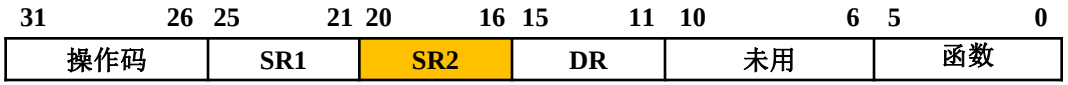


第二个源操作数

- I-类型,Immediate
  - [15:0]，直接获得
  - 立即数

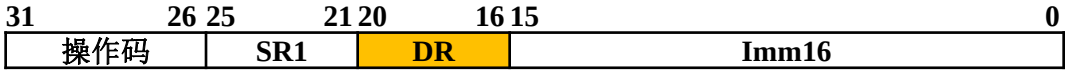


- R-类型，Register
  - [25:21]，SR2

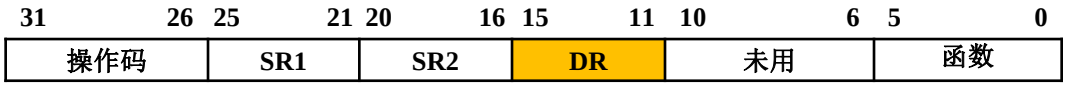


目标操作数

- I-类型,Immediate
  - [20:16], DR

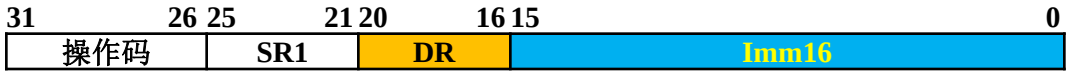


- R-类型, Register
  - [15:11], DR

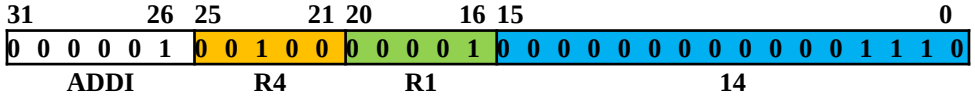


I-类型运算指令

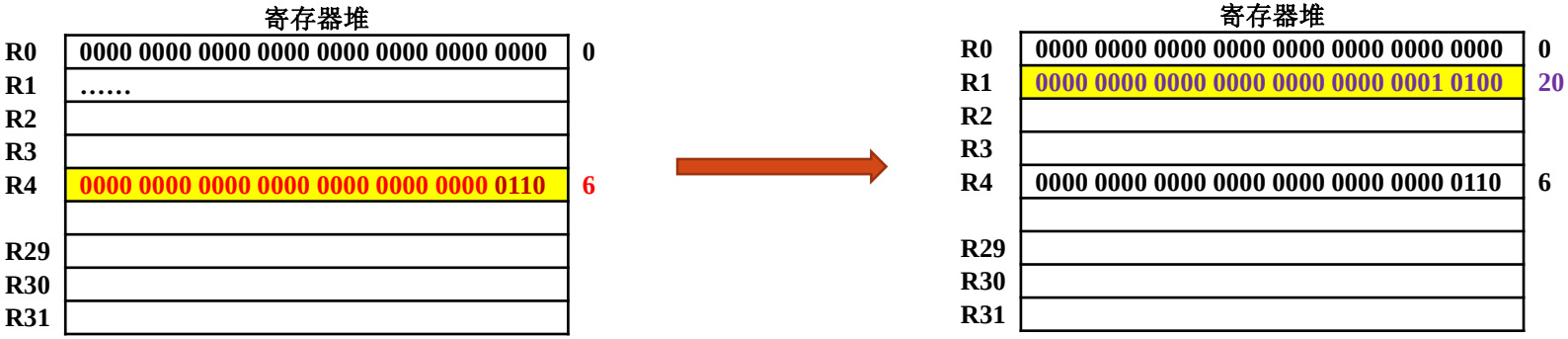
- 第二个源操作数
  - 来自于指令[15:0]进行符号扩展得到的32位整数，即立即数
- 目标操作数
  - 来自于指令[20:16]所标识的寄存器中



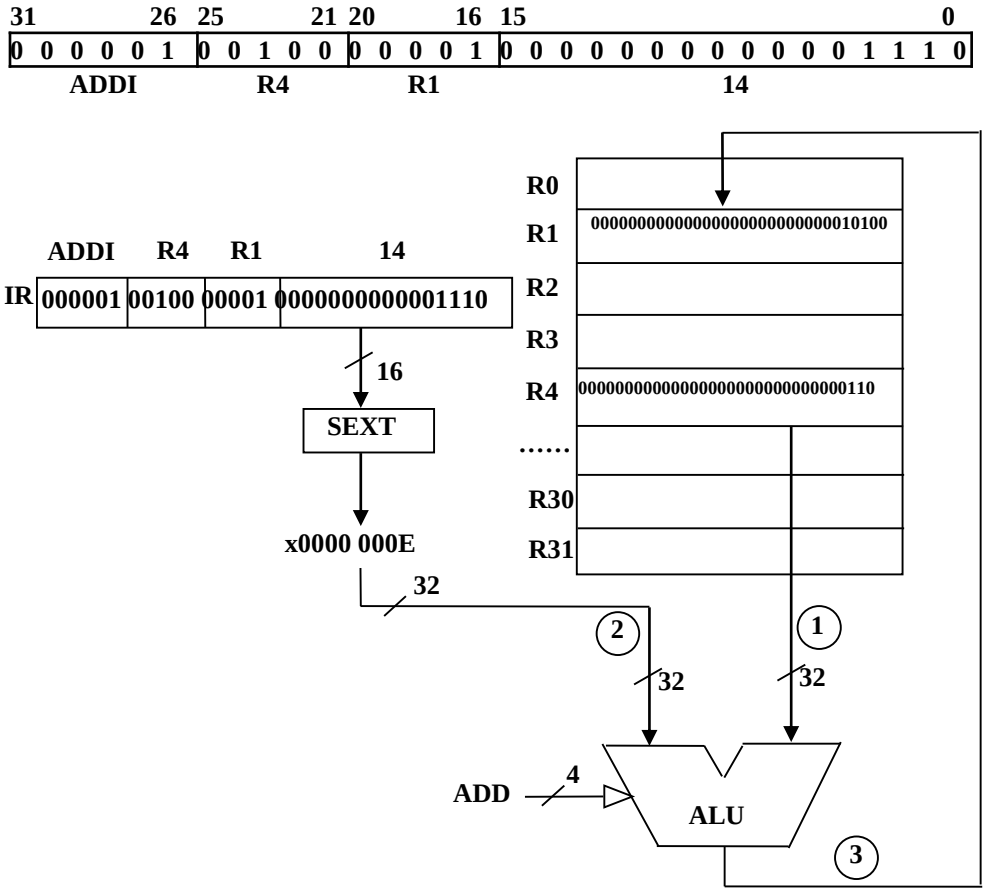
ADDI



- ADD代表加，I代表立即数（Immediate）
  - 第一个操作数R4
  - 第二个源操作数在指令中
    - [15:0]位符号扩展（SEXT）
  - 目标操作数写入R1

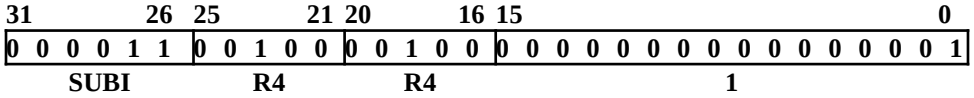


ADDI



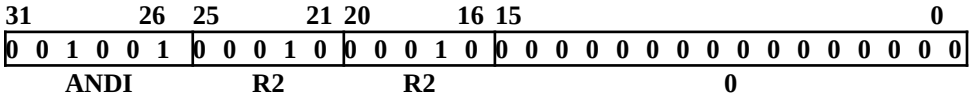
- 问题：哪些整数可以用作立即数？

SUBI

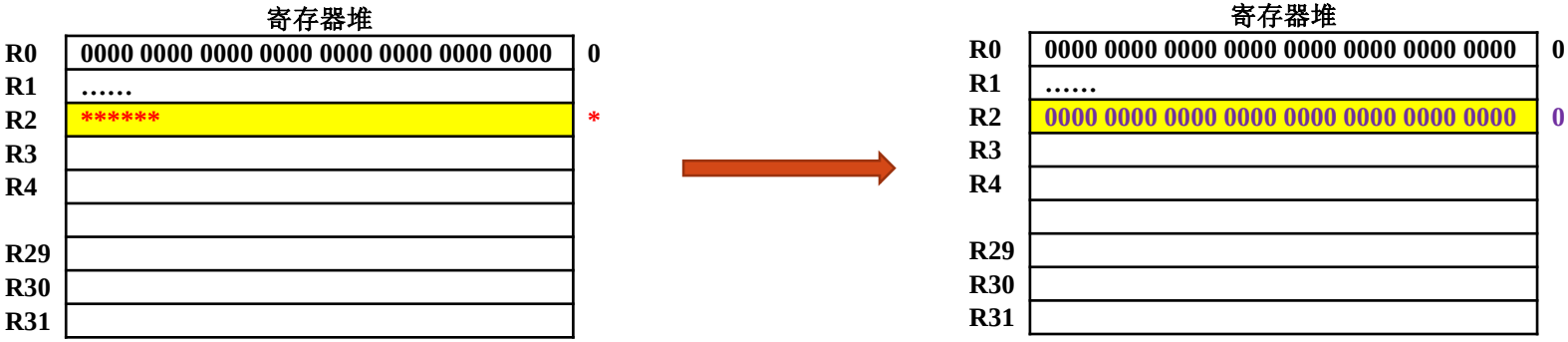


- 减（Subtract）
- 指令执行结果
  - 寄存器R4存储的数据减1
  - R4←(R4)-1
- 在同一条指令中一个寄存器既可以作为源操作数也可以作为目标操作数
  - 对DLX的所有运算指令都是适用的

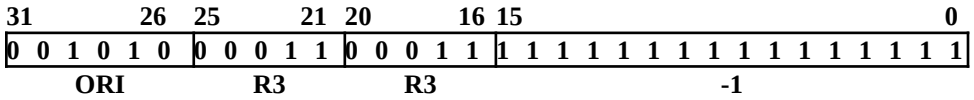
ANDI



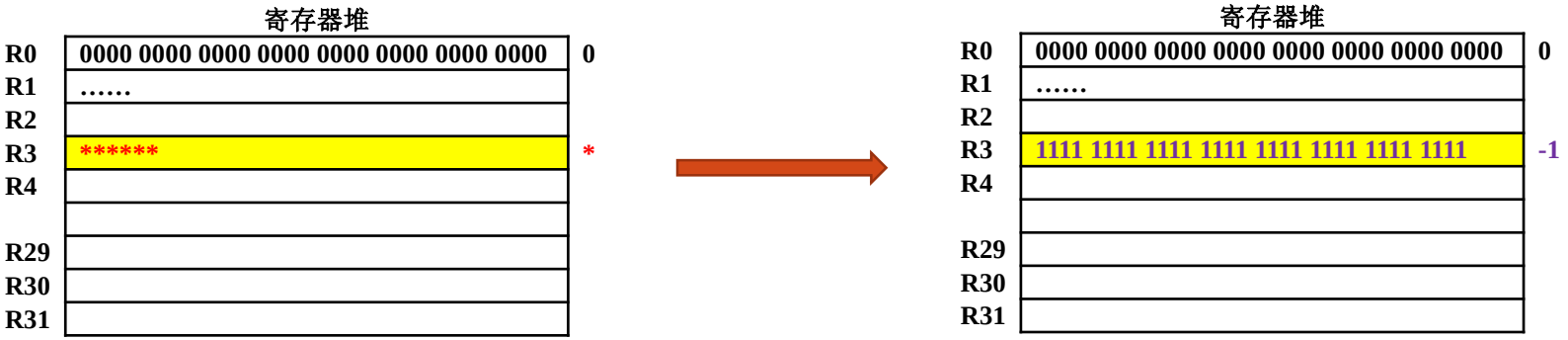
- 指令执行结果：寄存器R2被清空
  - R2←(R2) AND 0
  - 结果，R2的32位全部为0



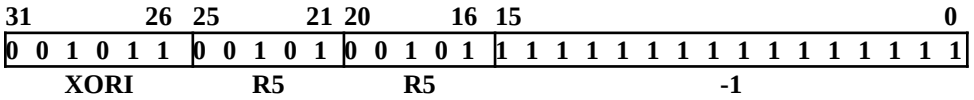
ORI



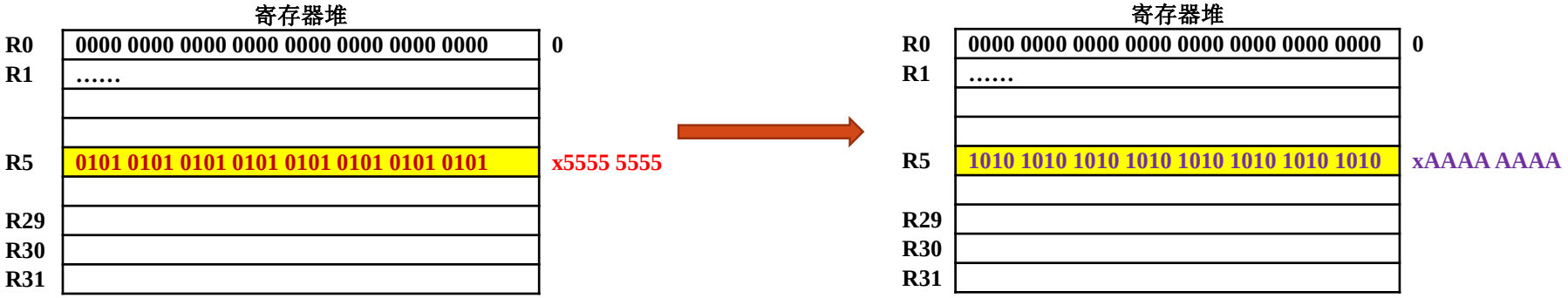
- 指令执行结果：寄存器R3被设为-1
  - R3的32位全部为1



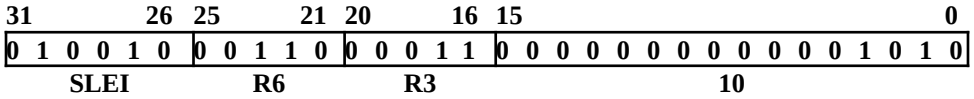
XORI



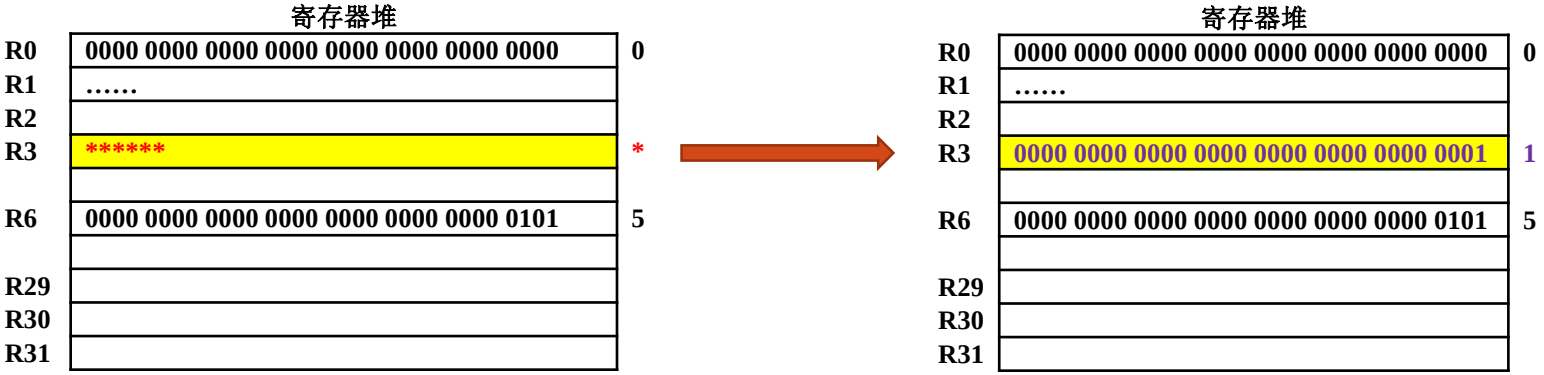
- 指令执行结果：寄存器R5被按位取反



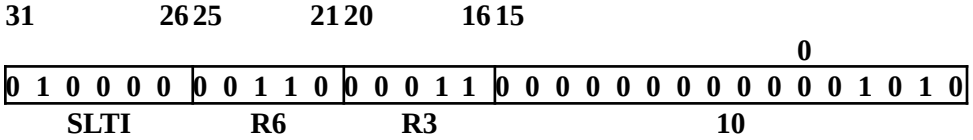
SLEI



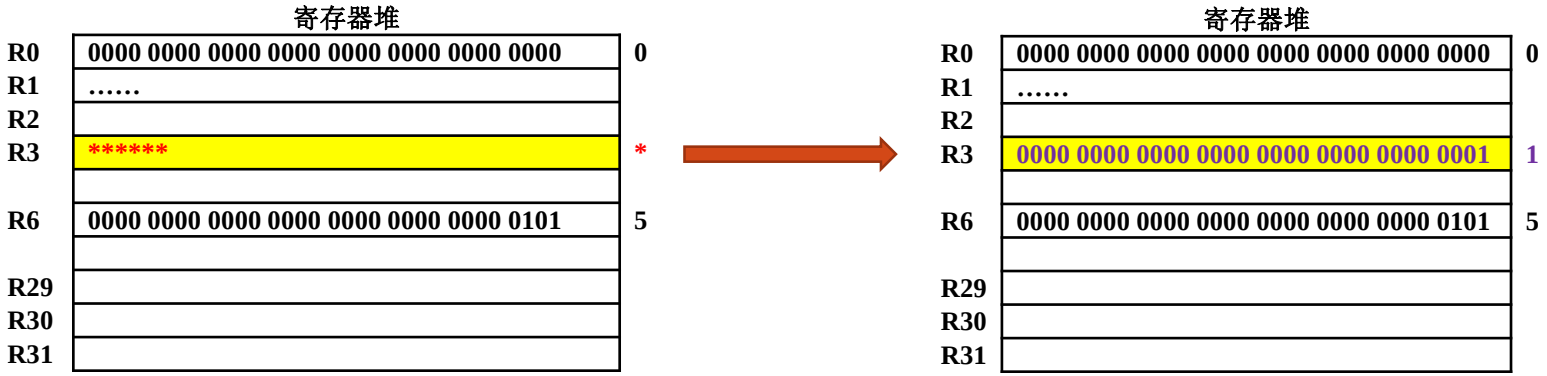
- 设置是否小于等于条件操作（Set on Less than or Equal to）
- 当指令[25:21]表示的寄存器中的值小于等于[15:0]表示的立即数时，[20:16]表示的寄存器中的值被设为1（真），否则设为0（假）
  - 如果R6==5，指令执行结果为：寄存器R3被设为1（因为R6的值小于10，条件为真）



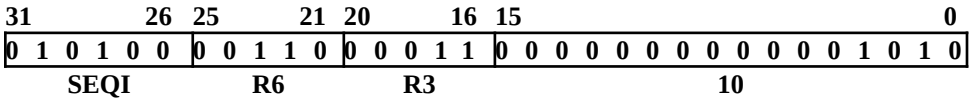
SLTI



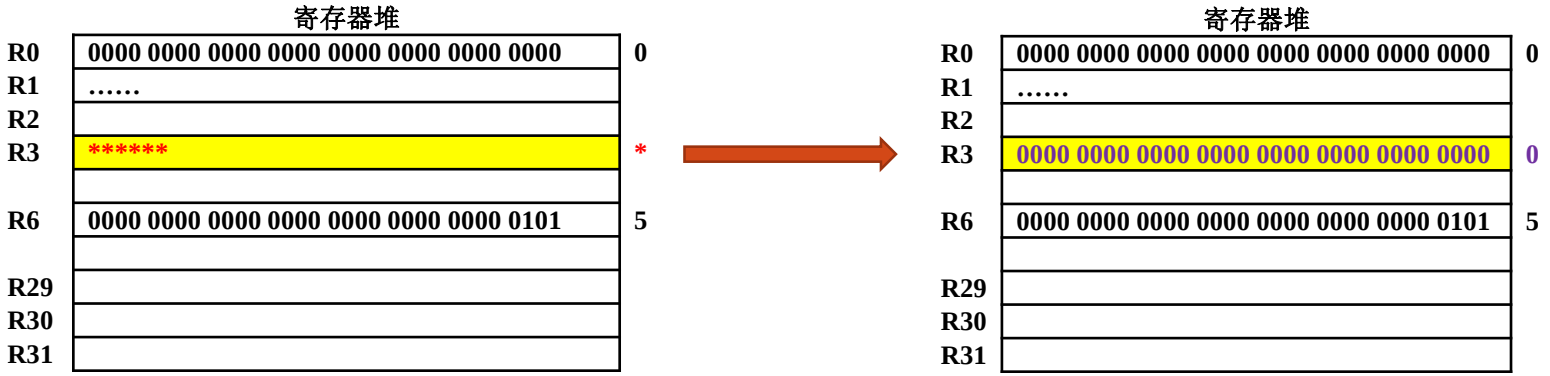
- 设置是否小于条件操作（Set on Less than）
  - 如果R6==5，指令执行结果为：寄存器R3被设为1（因为R6的值小于10，条件为真）



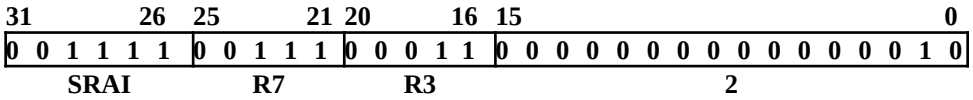
SEQI



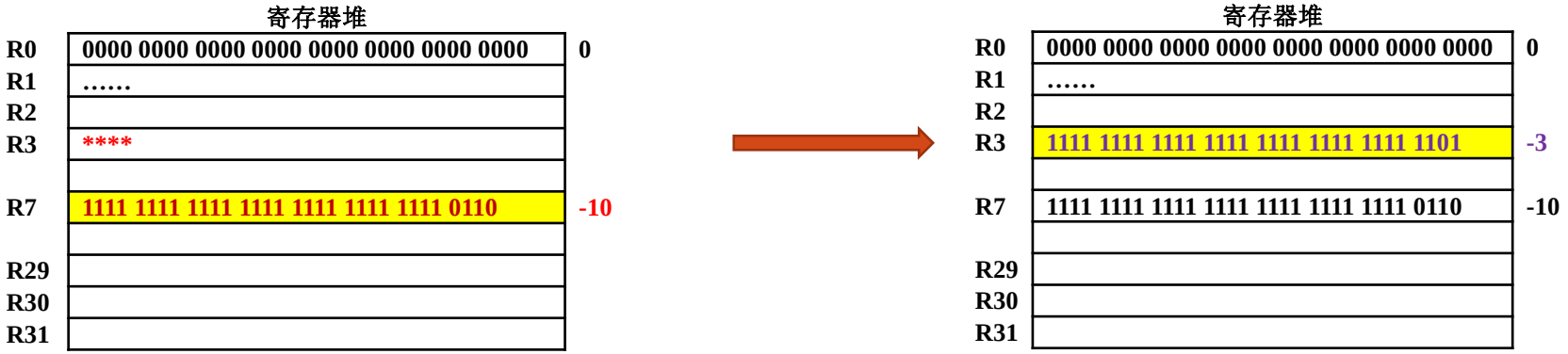
- 设置是否相等条件操作（Set on Equal to）
  - 如果R6==5，指令执行结果为：寄存器R3被设为0（因为R6的值不等于10，条件为假）



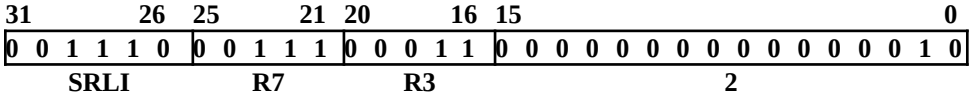
SRAI



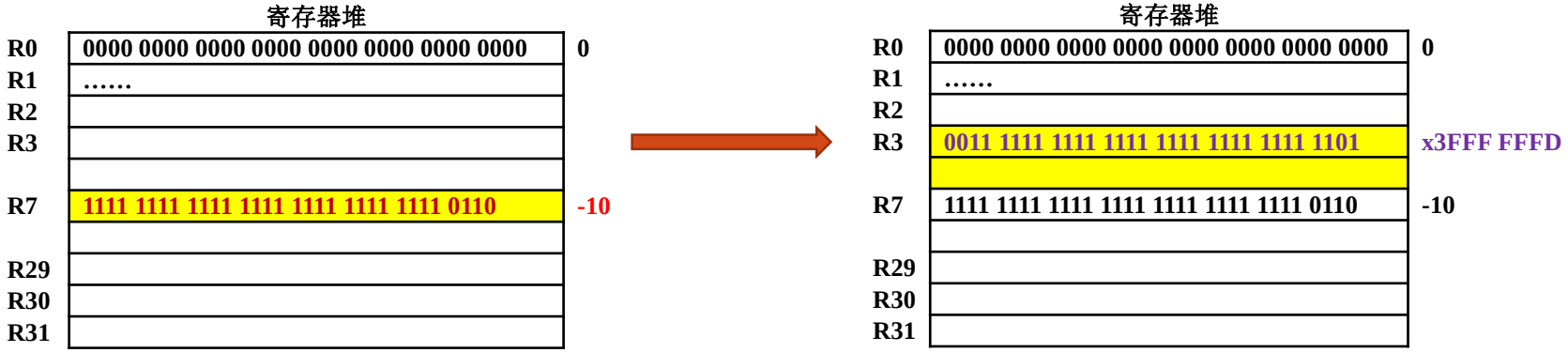
- 算术右移立即数操作（Shift Right Arithmetic）
  - 对指令[25:21]表示的寄存器中的值进行算术右移操作，所移位数为[15:0]表示的立即数
  - 如果R7==-10，指令执行结果为：寄存器R3的值为xFFFF FFFD，即-3（注意，R7是负数，左边补1）
    - R7的值除以4的结果，即算术右移表示作除法



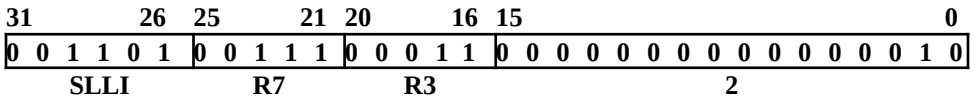
SRLI



- 逻辑右移立即数操作（Shift Right Logical）
  - 对指令[25:21]表示的寄存器中的值进行逻辑右移操作，所移位数为[15:0]表示的立即数
  - 逻辑右移的含义是：移位后，左边补0
    - 如果R7==-10，指令执行结果为：寄存器R3的值为x3FFF FFFD



SLLI



- 左移立即数操作（Shift Left Logical）
  - 对指令[25:21]表示的寄存器中的值进行左移操作，所移位数为[15:0]表示的立即数，右边补0
  - 如果R7==-10，指令执行结果为：寄存器R3的值为xFFFF FFD8，即-40
    - R7的值乘4的结果，即向左移表示作乘法，左移1位就是乘一次2

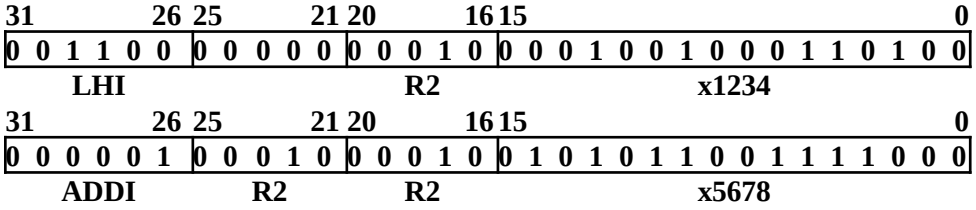
寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	……	
R2		
R3	****	
R7	1111 1111 1111 1111 1111 1111 1111 0110	-10
R29		
R30		
R31		

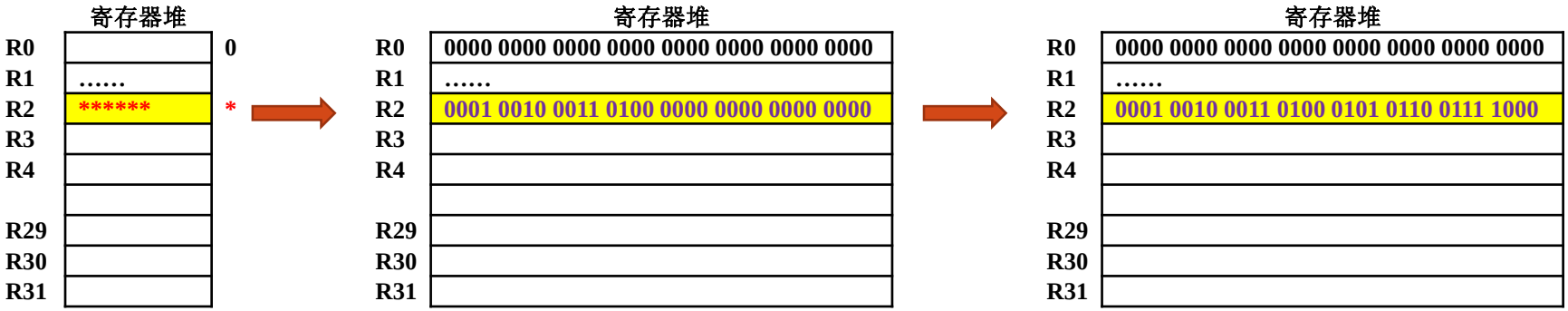
寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	……	
R2		
R3	1111 1111 1111 1111 1111 1111 1101 1000	-40
R7	1111 1111 1111 1111 1111 1111 1111 0110	-10
R29		
R30		
R31		

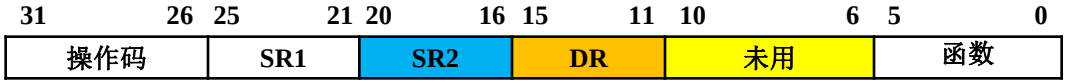
LHI



- LHI指令：加载高位立即数操作，将立即数左移16位后，加载到目标操作数中
  - R2← x12340000
  - R2← x12340000+x00005678，R2← x12345678
  - 将某个较大的常数赋值给某个寄存器

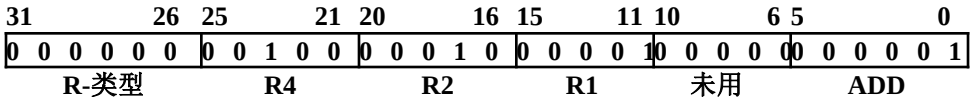


R-类型运算指令



- 第二个源操作数
  - 来自于指令[20:16]所标识的寄存器中
- 目标操作数
  - 来自于指令[15:11]所标识的寄存器中

ADD



- 操作码000000，R-类型
  - [5:0]为000001，ADD函数

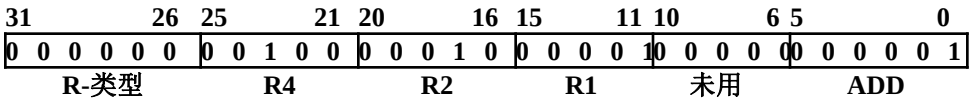
寄存器堆

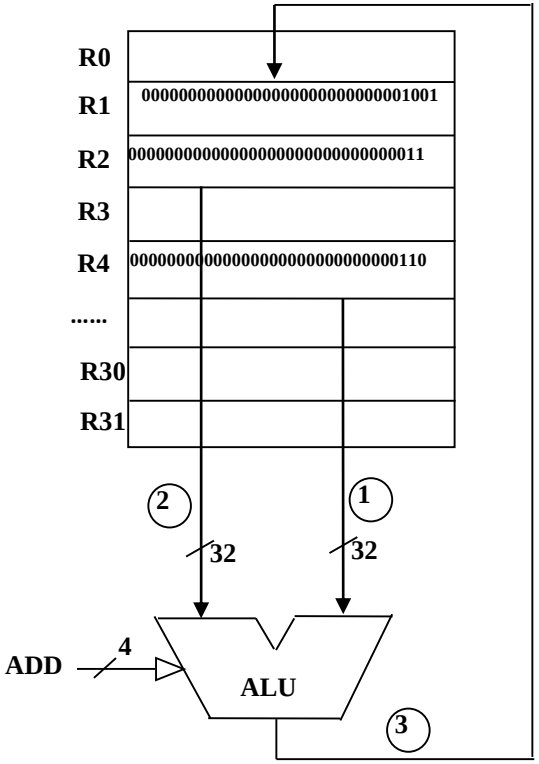
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	******	*
R2	0000 0000 0000 0000 0000 0000 0000 0011	3
R3		
R4	0000 0000 0000 0000 0000 0000 0000 0110	6
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0000 0000 0000 0000 0000 0000 0000 1001	9
R2	0000 0000 0000 0000 0000 0000 0000 0011	3
R3		
R4	0000 0000 0000 0000 0000 0000 0000 0110	6
R29		
R30		
R31		

ADD





- 除LHI指令外，其他运算指令均有I-类型和R-类型指令，其解释均与之类似

指令集结构ISA

- 概述
- 算术/逻辑运算指令
- 数据传送指令
- 控制指令

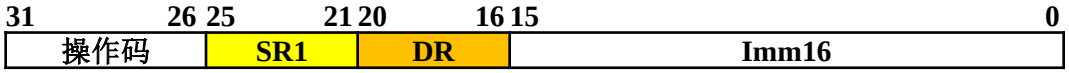
数据传送指令

- 存储器和通用寄存器之间：本章
- 寄存器和输入/输出设备之间：第12章
- 整数寄存器与特殊寄存器之间：第13章
- 还包括整数寄存器与浮点数寄存器之间，存储器和输入/输出设备之间传送数据

加载/存储

- 加载（load）：将数据从存储器移动到寄存器的过程
- 存储（store）：将数据从寄存器移动到存储器的过程
- LB和SB：加载和存储一个8位的字节，在一个存储单元和一个寄存器之间传送数据
- LW和SW：加载和存储一个32位的字，在4个连续的存储单元和一个寄存器之间传送数据

I-类型

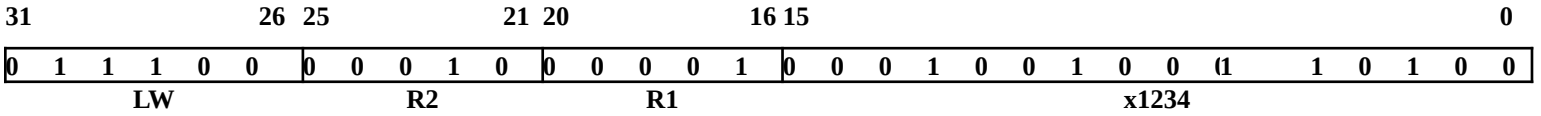


- 指令的[25:21]位指明了源操作数SR1
- [20:16]位指明了目标操作数DR
- 加载：DR寄存器将在从存储器读取数据之后，包含该数值（指令完成时）
- 存储：DR寄存器则包含了要被写到存储器中的数值
- 如何在32位的指令中声明一个32位的存储单元地址？
  - “基址寄存器+偏移量”的寻址模式

基址寄存器+偏移量

- 存储单元的地址：将16位的偏移量进行符号扩展后，与一个基址寄存器相加得到
  - 16位的偏移量是从指令中得到的，是[15:0]位
    - 偏移量值：在-2^{15}到$2^{15}-1$之间的二进制补码整数
  - 基址寄存器则使用指令的[25:21]位来说明，即SR1

LW



地址	
x5678 1234	0000 1111
x5678 1235	0000 1111
x5678 1236	0000 1111
x5678 1237	0000 1111

- 基址+偏移量
  - （R2）+ x0000 1234
  - x56781234

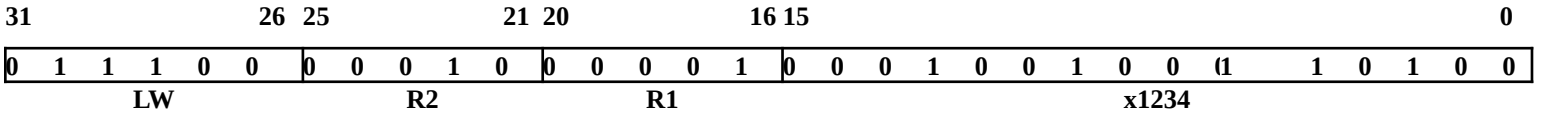
寄存器堆

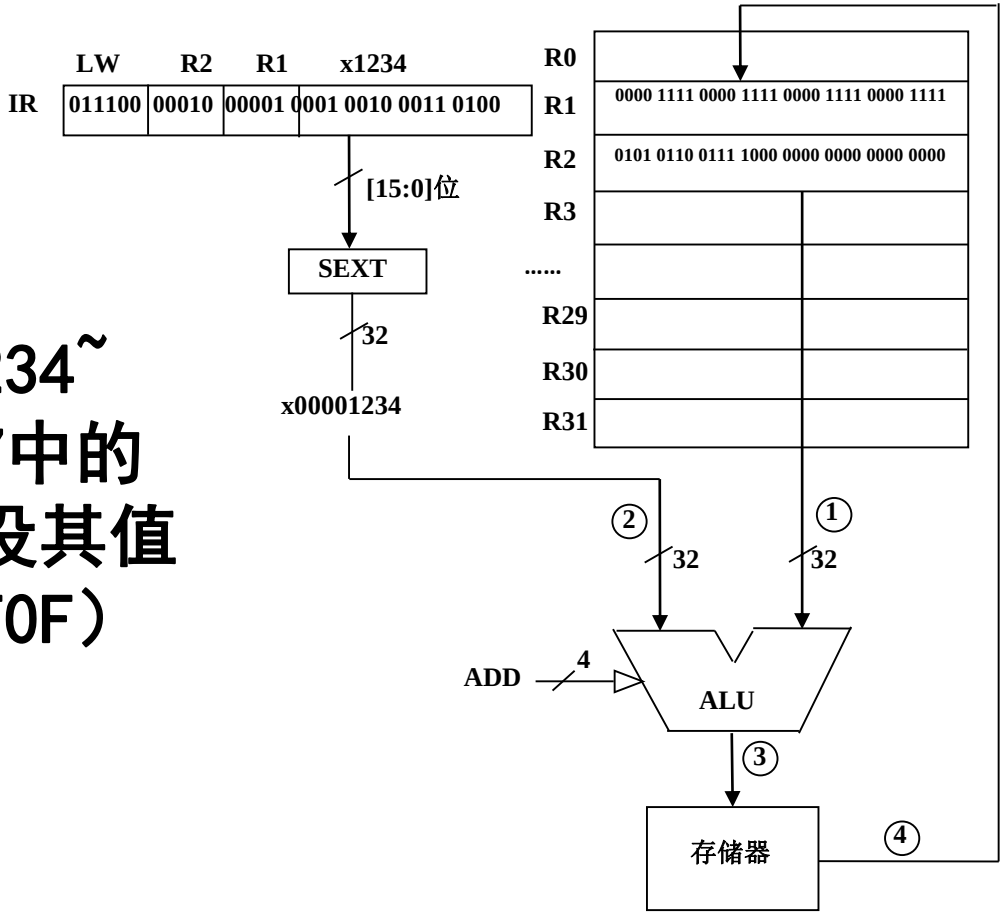
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	******	*
R2	0101 0110 0111 1000 0000 0000 0000 0000	x5678 0000
R3		
R4		
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0000 1111 0000 1111 0000 1111 0000 1111	x0F0F 0F0F
R2	0101 0110 0111 1000 0000 0000 0000 0000	
R3		
R4		
R29		
R30		
R31		

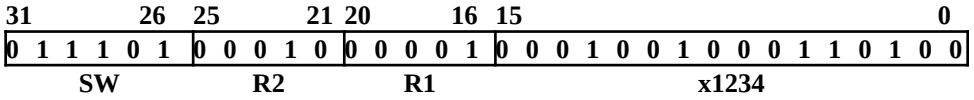
LW





- 将x56781234~x56781237中的内容（假设其值为x0F0F0F0F）加载到R1

SW



- “基址（R2）+偏移量（x0000 1234）”计算的结果是x56781234
- 取出R1中的数值（如x0F0F0F0F），存储于x56781234~x56781237单元中

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0000 1111 0000 1111 0000 1111 0000 1111	x0F0F 0F0F
R2	0101 0110 0111 1000 0000 0000 0000 0000	x5678 0000
R3		
R4		
R29		
R30		
R31		

地址

x5678 1234	0000 1111
x5678 1235	0000 1111
x5678 1236	0000 1111
x5678 1237	0000 1111

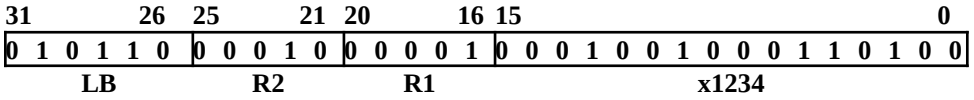
R0

- 绝对地址
  - 如果SR1为R0，由于R0=零，“基址+偏移量”的计算结果就是IR[15:0]经过符号扩展得到的值，该值就是访问存储器的地址
- 加载指令不可使用R0作为目标寄存器
- 存储指令使用R0作为DR
  - 表示存储到存储单元的是数值0

边界对齐

- LW/SW指令：加载/存储一个32位的字
  - “基址+偏移量”的计算结果是4个连续的存储单元的低地址，必须是4的倍数

LB



- 基址+偏移量
  - （R2）+ x0000 1234
  - x56781234
- 符号扩展到32位

地址	
x5678 1234	0000 1111
x5678 1235	******
x5678 1236	******
x5678 1237	******

寄存器堆

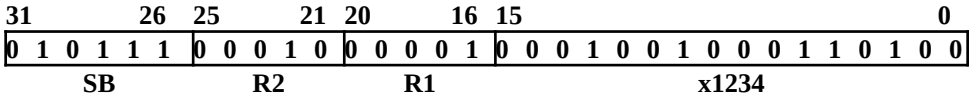
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	******	*
R2	0101 0110 0111 1000 0000 0000 0000 0000	x5678 0000
R3		
R4		
R29		
R30		
R31		

寄存器堆

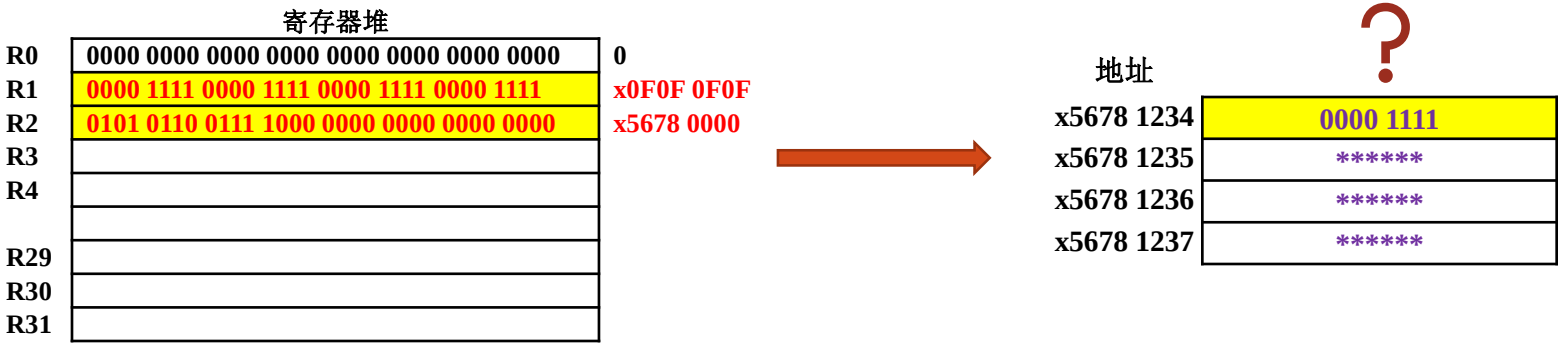
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0000 0000 0000 0000 0000 0000 0000 1111	15
R2	0101 0110 0111 1000 0000 0000 0000 0000	
R3		
R4		
R29		
R30		
R31		

?

SB



- 基址+偏移量
  - （R2）+ x0000 1234
  - x56781234
- R1中数值低8位（最低有效字节）



示例

地址	31 26	25 21	20 16	15 11	10 6	5 0	
x4000 0000	001100	00000	00001	0100 0000 0000 0000			LHI R1, x4000
x4000 0004	000001	00000	00010	0000 0000 0000 0101			ADDI R2, R0, 5
x4000 0008	011101	00001	00010	0000 0000 0001 0000			SW 16(R1), R2
x4000 000C	011100	00001	00011	0000 0000 0001 0000			LW R3, 16(R1)
x4000 0010							

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	……	
R2		
R3		
R4		
R29		
R30		
R31		

LHI

地址	31 26	25 21	20 16	15 11	10 6	5 0	
x4000 0000	001100	00000	00001	0100 0000 0000 0000			LHI R1, x4000
x4000 0004	000001	00000	00010	0000 0000 0000 0101			ADDI R2, R0, 5
x4000 0008	011101	00001	00010	0000 0000 0001 0000			SW 16(R1), R2
x4000 000C	011100	00001	00011	0000 0000 0001 0000			LW R3, 16(R1)
x4000 0010							

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	……	
R2		
R3		
R4		
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0100 0000 0000 0000 0000 0000 0000 0000	x4000 0000
R2	……	
R3		
R4		
R29		
R30		
R31		

- R1 ← x4000 0000

ADDI

地址	31 26	25 21	20 16	15 11	10 6	5 0	
x4000 0000	001100	00000	00001	0100 0000 0000 0000			LHI R1, x4000
x4000 0004	000001	00000	00010	0000 0000 0000 0101			ADDI R2, R0, 5
x4000 0008	011101	00001	00010	0000 0000 0001 0000			SW 16(R1), R2
x4000 000C	011100	00001	00011	0000 0000 0001 0000			LW R3, 16(R1)
x4000 0010							

- R2 ← （R0）+ 5

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0100 0000 0000 0000 0000 0000 0000 0000	x4000 0000
R2	……	
R3		
R4		
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0100 0000 0000 0000 0000 0000 0000 0000	x4000 0000
R2	0000 0000 0000 0000 0000 0000 0000 0101	5
R3		
R4		
R29		
R30		
R31		

SW

地址	31 26	25 21	20 16	15 11	10 6	5 0	
x4000 0000	001100	00000	00001	0100 0000 0000 0000			LHI R1, x4000
x4000 0004	000001	00000	00010	0000 0000 0000 0101			ADDI R2, R0, 5
x4000 0008	011101	00001	00010	0000 0000 0001 0000			SW 16(R1), R2
x4000 000C	011100	00001	00011	0000 0000 0001 0000			LW R3, 16(R1)
x4000 0010	0000 0000 0000 0000 0000 0000 0000 0101						

- 基址+偏移量： x4000 0000 + x0000 0010

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0100 0000 0000 0000 0000 0000 0000 0000	x4000 0000
R2	0000 0000 0000 0000 0000 0000 0000 0101	5
R3		
R4		
R29		
R30		
R31		

地址	
x4000 0010	0000 0000
x4000 0011	0000 0000
x4000 0012	0000 0000
x4000 0013	0000 0101

LW

地址	31 26	25 21	20 16	15 11	10 6	5 0	
x4000 0000	001100	00000	00001	0100 0000 0000 0000			LHI R1, x4000
x4000 0004	000001	00000	00010	0000 0000 0000 0101			ADDI R2, R0, 5
x4000 0008	011101	00001	00010	0000 0000 0001 0000			SW 16(R1), R2
x4000 000C	011100	00001	00011	0000 0000 0001 0000			LW R3, 16(R1)
x4000 0010	0000 0000 0000 0000 0000 0000 0000 0101						

- 基址+偏移量：x4000 0000 + x0000 0010

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0100 0000 0000 0000 0000 0000 0000 0000	x4000 0000
R2	0000 0000 0000 0000 0000 0000 0000 0101	5
R3	……	
R4		
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0100 0000 0000 0000 0000 0000 0000 0000	x4000 0000
R2	0000 0000 0000 0000 0000 0000 0000 0101	5
R3	0000 0000 0000 0000 0000 0000 0000 0101	5
R4		
R29		
R30		
R31		

运算指令和数据传送指令

- 不改变指令执行的顺序
- 控制器中的程序计数器PC
  - 记录下一条要执行的指令的地址
  - PC ← PC+4
    - 一条指令占用连续的4个存储单元

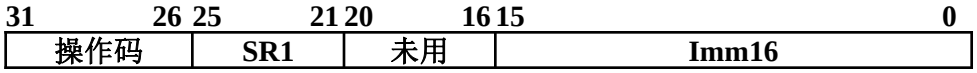
指令集结构ISA

- 概述
- 算术/逻辑运算指令
- 数据传送指令
- 控制指令

控制指令

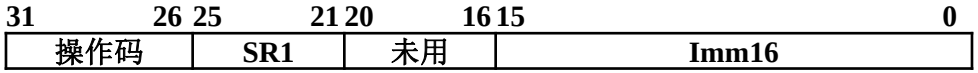
- 改变被执行的指令的顺序
- DLX有10条指令能使顺序流被打破
  - 条件分支
  - 无条件跳转
  - 子例程（有时称为函数）调用
  - TRAP
  - 从异常/中断返回

条件分支



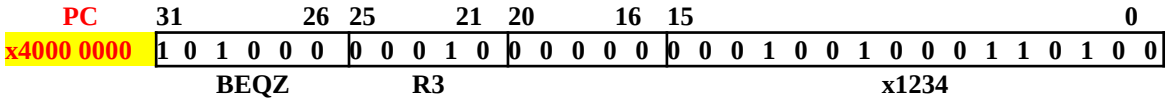
- 采用I-类型格式
- 使用[25:21]位的寄存器，决定是否改变指令流，即是否改变正常执行指令的顺序
  - BEQZ，等于零时分支（Branch on Equal to Zero）
  - BNEZ，不等于零时分支（Branch on Not Equal to Zero）

BEQZ指令

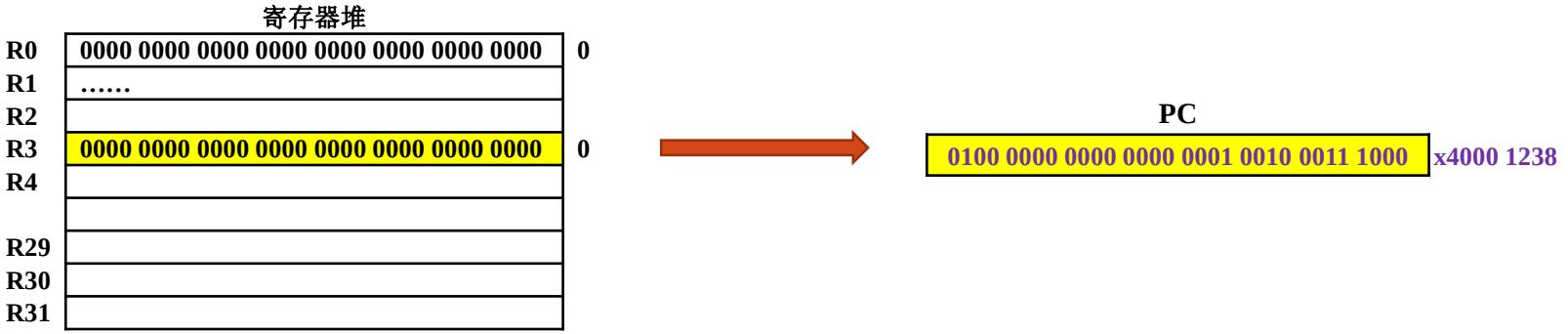


- 取指令阶段：PC加4；
- 译码/取寄存器阶段：计算（加过4的）PC加SEXT(IR[15:0])；
- 完成分支阶段：判断[25:21]位的SR1的值是否为0？
  - 如果SR1的值为0，PC就被上一阶段得到的地址加载；
    - PC ← PC + 4 + SEXT(Imm16)
  - 如果SR1的值不为0，那么，直接进入下一取指令阶段，PC保持不变
    - PC ← PC + 4

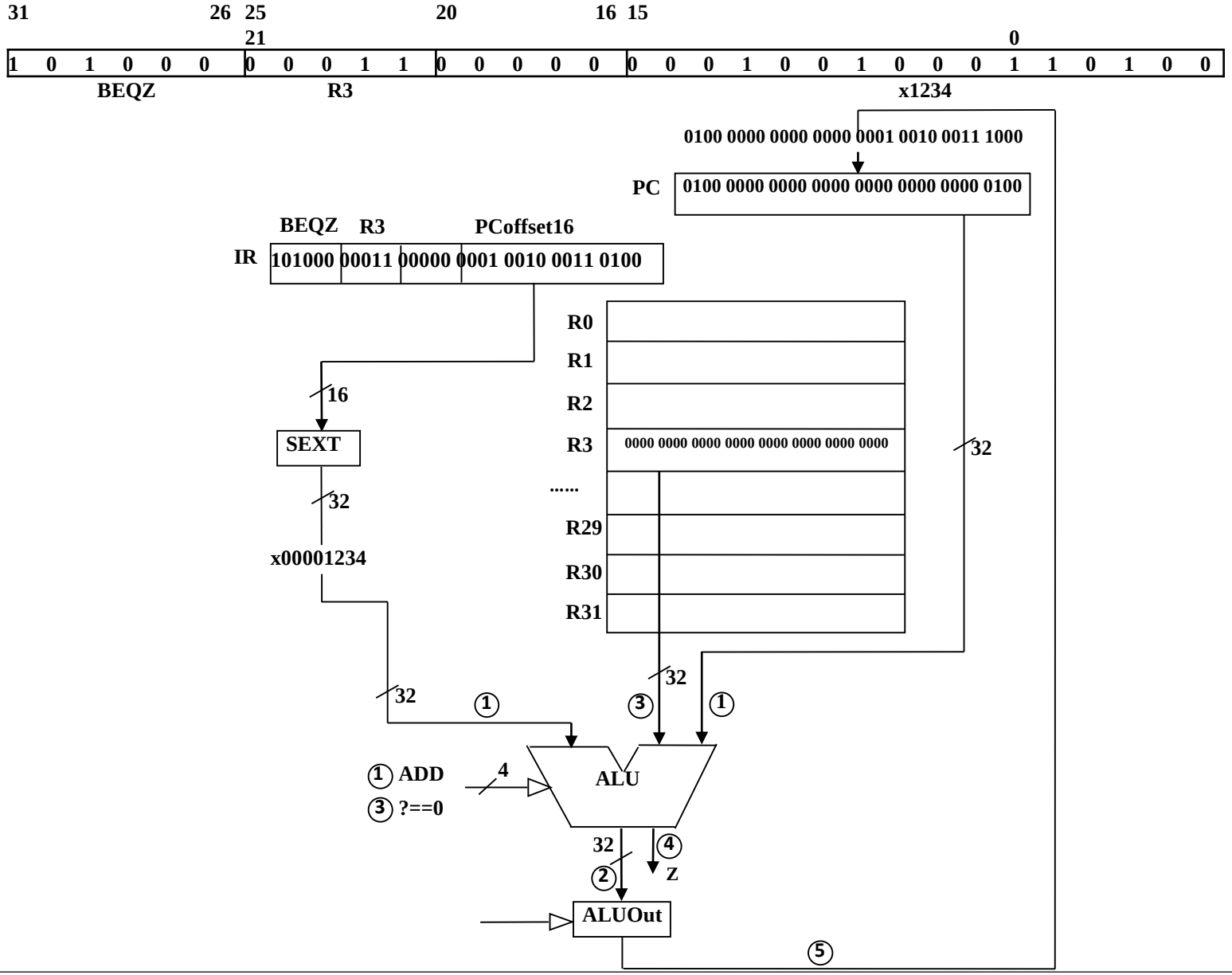
BEQZ指令



- PC ← PC + 4 + SEXT(Imm16)
  - 实际PC ← PC + x0000 0004 + x0000 1234



BEQZ


31
26
25
21
20
16
15
0
1 0 1 0 0 0
0 0 0 1 1
0 0 0 0 0
0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0
BEQZ
R3
x1234
0100 0000 0000 0000 0001 0010 0011 1000
PC
0100 0000 0000 0000 0000 0000 0000 0100
BEQZ
R3
PCoffset16
IR
101000
00011
00000
0001 0010 0011 0100
R0
R1
R2
R3
0000 0000 0000 0000 0000 0000 0000 0000
.....
R29
R30
R31
16
SEXT
32
x00001234
32
32
32
1 ADD
3 ?==0
4
ALU
32
2
4
Z
ALUOut
5

地址限制

- 在加上偏移量之前，PC已经被增加4
- 计算出来的地址
  - 只能在BEQZ或BNEZ这条指令的PC+4+(2^{15}–1)或PC+4–2^{15}的单元范围之中
    - 即[PC+4–2^{15}, PC+3+2^{15}]
  - 指令的[15:0]位经过符号扩展后，能够提供的范围[–2^{15}, 2^{15}–1]

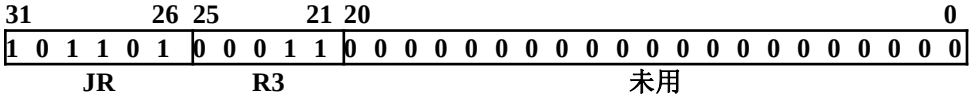
无条件分支

- 如果[25:21]位全部是0，表明要判断的寄存器是R0
- R0=0，PC被计算得到的地址加载
- 指令流无条件被改变

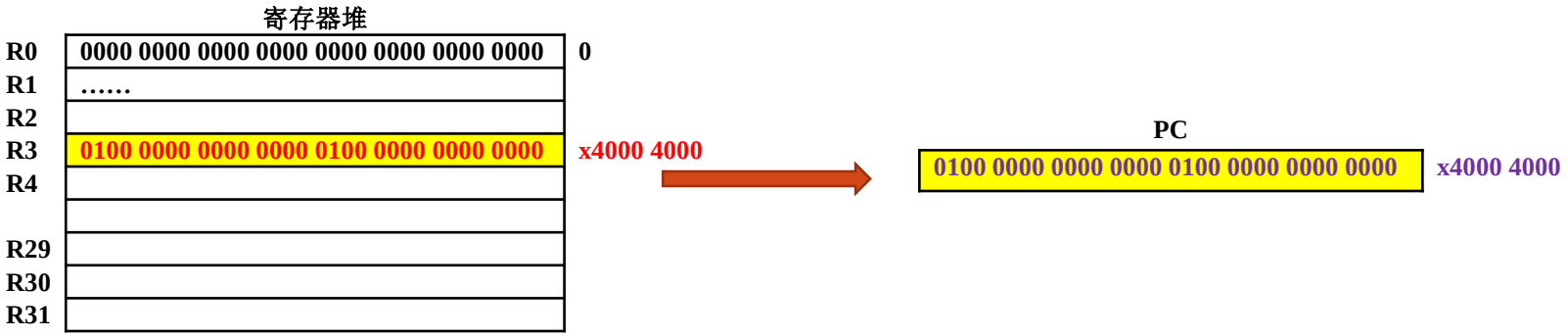
无条件跳转指令

- JR指令和J指令

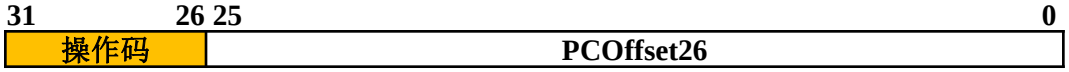
JR指令



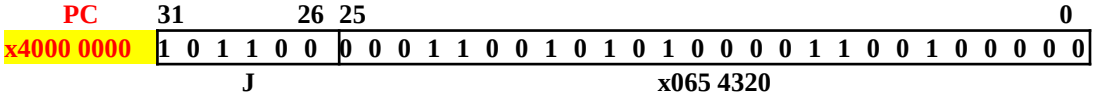
- 寄存器跳转（Jump Register）
- I-类型
- [20：0]位未用，设为0
- [25:21]位的寄存器
  - 包含下一条将要被执行的指令地址
- PC ← (R3)



J指令



- 跳转（Jump）
- J-类型
- PC ← PC+4+SEXT(PCOffset26)
- 订正：边界对齐，x4320



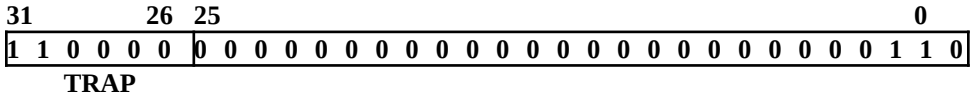
地址限制

- 因为在加上偏移量之前，PC已经被增加4
  - 计算出来的地址在J指令PC+4+$(2^{25}-1)$或PC+4-2^{25}的单元范围之中
    - 即[PC+4-2^{25}, PC+3+2^{25}]
- 若执行的下一条指令位于距当前指令2^{26}的位置上，如何实现？
  - JR指令

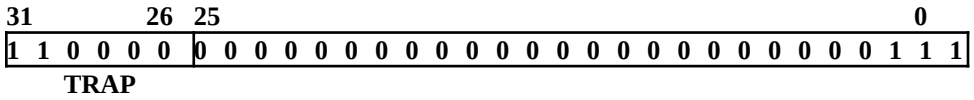
TRAP指令

- 用于输入和输出
- TRAP指令（操作码是110000）
  - 改变PC，使其指向属于操作系统的某部分的存储器地址
  - 作用是为了让操作系统代表正在执行的程序执行一些任务
  - TRAP调用了一个操作系统的“服务例程”
  - TRAP指令的[25:0]位为“TRAP向量”，标明程序希望操作系统执行哪一个服务调用

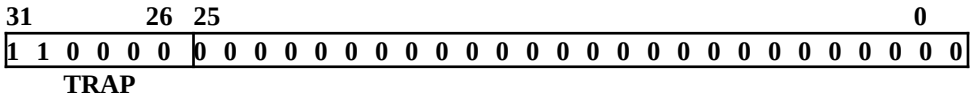
示例



- TRAP向量＝x0006
  - 从键盘输入一个字符，并存储于R4中



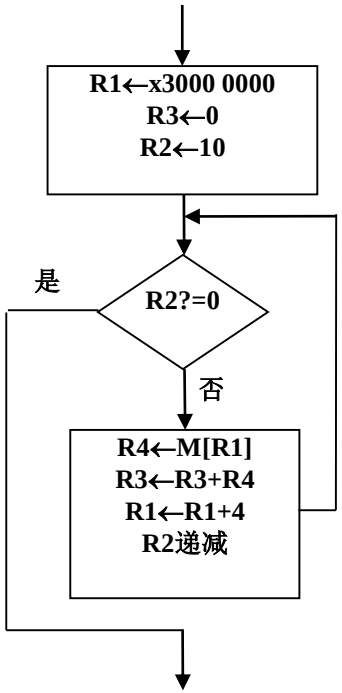
- TRAP向量＝x0007
  - 向显示器输出存储于R4中的一个字符



- TRAP向量＝x0000
  - 停止程序

示例：计算10个整数的和

- 假设已知从x3000 0000到x3000 0027的地址中存储了10个整数，希望计算这些整数的和
- 计数器控制的循环(for)
  - 初始化变量——寄存器
    - R1：存储整数的起始地址
    - R3：累加和
    - R2：循环计数器（等于0跳出循环）
    - R4：整数临时存放
  - 注：M[xx]表示在存储器中xx地址开始的连续4个存储单元的总共32位值

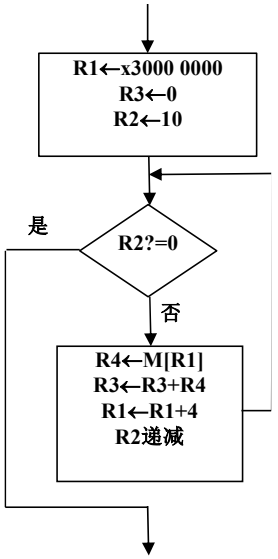


指令序列

地址 PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	……	
R2		
R3		
R4		
R29		
R30		
R31		

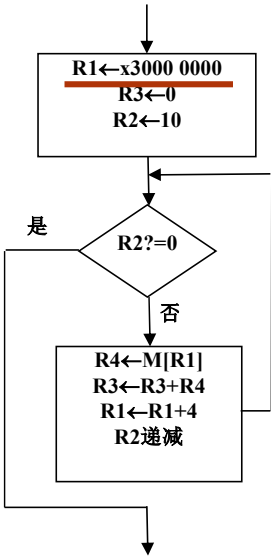


- BEQZ采用R0作为条件判断，实现了无条件分支
- 更好的做法是使用J指令：

101100	11 1111 1111 1111 1111 1110 1000
J	-24

LHI指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	……	
R2		
R3		
R4		
R29		
R30		
R31		

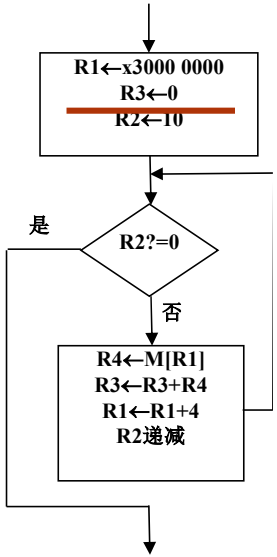
寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	……	
R3		
R4		
R29		
R30		
R31		

- R1 ← x3000 0000

ANDI指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- R3 ← (R3) AND 0

寄存器堆

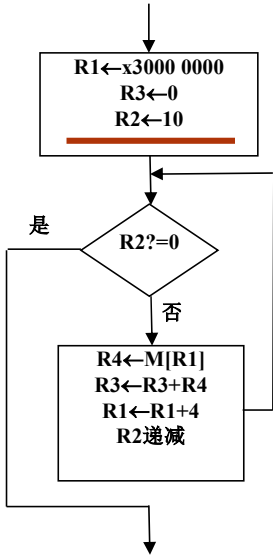
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2		
R3	******	
R4		
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	……	
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4		
R29		
R30		
R31		

ADDI指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- R2 ← (R0) + 10

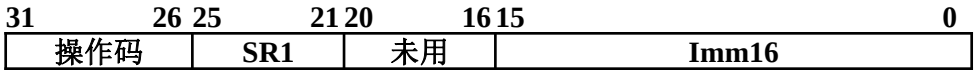
寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	******	
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4		
R29		
R30		
R31		

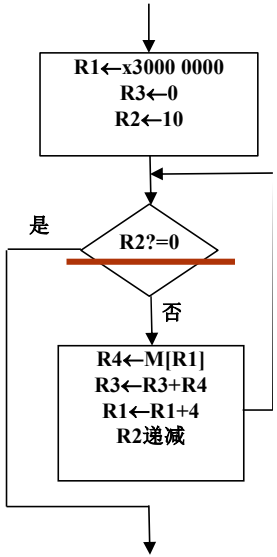
寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4		
R29		
R30		
R31		

BEQZ指令



PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- R2!=0, PC ← PC + 4

寄存器堆

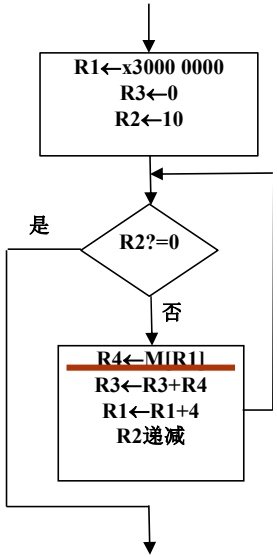
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4		
R29		
R30		
R31		

PC

0100 0000 0000 0000 0000 0000 0001 0000 x4000 0010

LW指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- 基址+偏移量： x3000 0000 + x0000 0000

寄存器堆

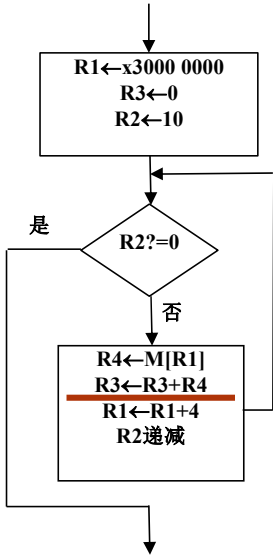
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4	******	
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4	******	M[x3000 0000]
R29		
R30		
R31		

ADD指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- R3 ← (R3) + (R4)

寄存器堆

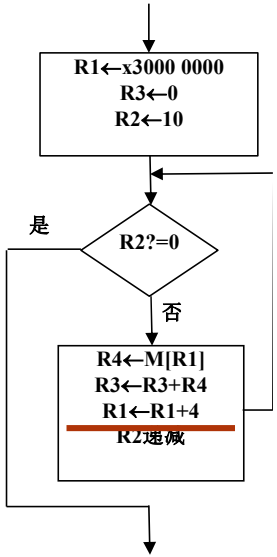
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	0000 0000 0000 0000 0000 0000 0000 0000	0
R4	******	M[x3000 0000]
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	******	M[x3000 0000]
R4	******	M[x3000 0000]
R29		
R30		
R31		

ADDI指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- R1 ← (R1) + 4

寄存器堆

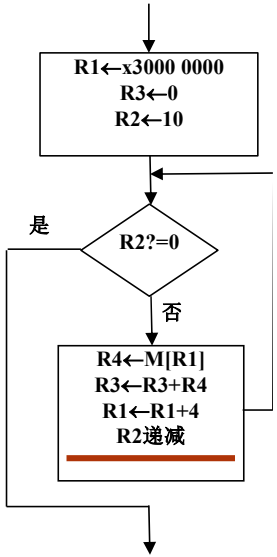
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0000	x3000 0000
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	******	M[x3000 0000]
R4	******	M[x3000 0000]
R29		
R30		
R31		

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0100	x3000 0004
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	******	M[x3000 0000]
R4	******	M[x3000 0000]
R29		
R30		
R31		

SUBI指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							



- R2 ← (R2) – 1

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0100	x3000 0004
R2	0000 0000 0000 0000 0000 0000 0000 1010	10
R3	******	M[x3000 0000]
R4	******	Mx3000 0000]
R29		
R30		
R31		

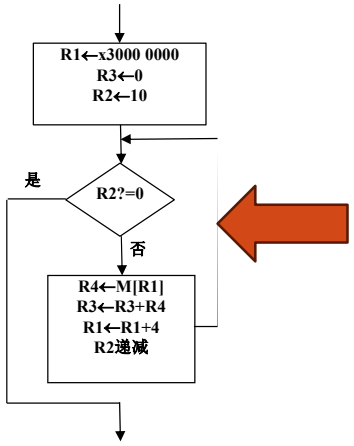
寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0100	x3000 0004
R2	0000 0000 0000 0000 0000 0000 0000 1001	9
R3	******	M[x3000 0000]
R4	******	M[x3000 0000]
R29		
R30		
R31		

J指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							

- PC ← PC + 4 + SEXT(xFFE8)
- X4000 0020 + x0000 0004 + SEXT(Imm16)
  = x4000 000C



PC
0100 0000 0000 0000 0000 0000 0010 0000 x4000 0020

PC
0100 0000 0000 0000 0000 0000 0000 1100 x4000 000C

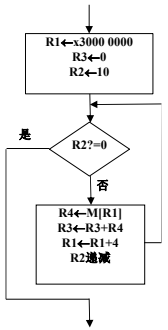
BEQZ指令

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							

- R2!=0, PC ← PC + 4

寄存器堆

R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0000 0100	x3000 0004
R2	0000 0000 0000 0000 0000 0000 0000 1001	9
R3	******	M[x3000 0000]
R4	******	M[x3000 0000]
R29		
R30		
R31		



PC

0100 0000 0000 0000 0000 0000 0001 0000	x4000 0010

重复执行

PC	31 26	25 21	20 16	15 11	10 6	5 0	解释
x4000 0000	001100	00000	00001	0011 0000 0000 0000			LHI R1,x3000
x4000 0004	001001	00011	00011	0000 0000 0000 0000			ANDI R3, R3, #0
x4000 0008	000001	00000	00010	0000 0000 0000 1010			ADDI R2,R0,#10
x4000 000C	101000	00010	00000	0000 0000 0001 0100			BEQZ R2, x14
x4000 0010	011100	00001	00100	0000 0000 0000 0000			LW R4,0(R1)
x4000 0014	000000	00011	00100	00011	00000	000001	ADD R3,R3,R4
x4000 0018	000001	00001	00001	0000 0000 0000 0100			ADDI R1,R1,#4
x4000 001C	000011	00010	00010	0000 0000 0000 0001			SUBI R2,R2,#1
x4000 0020	101100	11 1111 1111 1111 1111 1110 1000					J, #-24
x4000 0024							

- R2==0, PC ← PC + 4 + SEXT(x0014)

寄存器堆

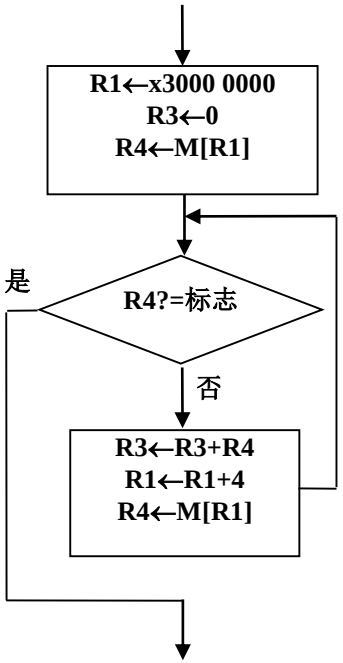
R0	0000 0000 0000 0000 0000 0000 0000 0000	0
R1	0011 0000 0000 0000 0000 0000 0010 1000	x3000 0028
R2	0000 0000 0000 0000 0000 0000 0000 0000	0
R3	******	Sum
R4	******	M[x3000 0024]
R29		
R30		
R31		

PC

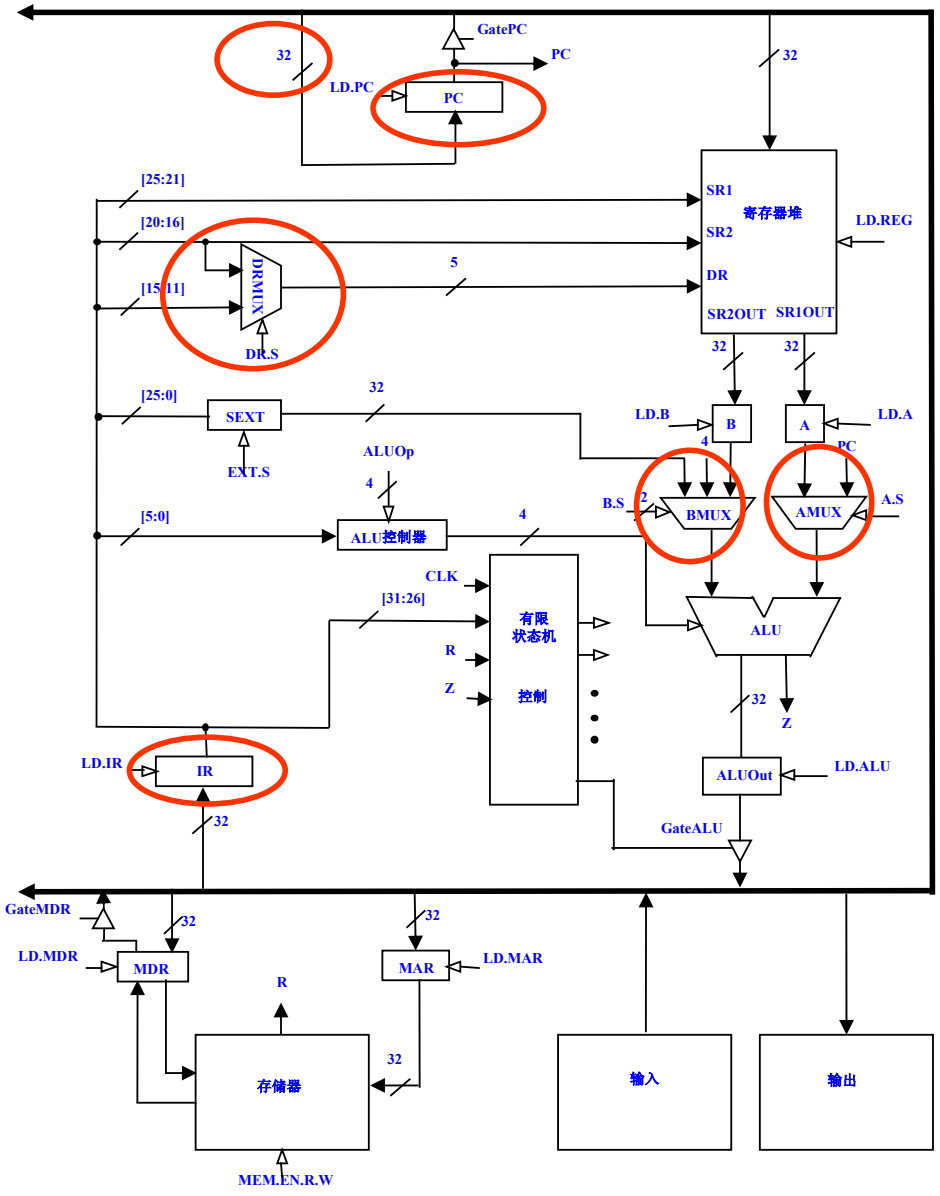
0100 0000 0000 0000 0000 0000 0010 0100 x4000 0024

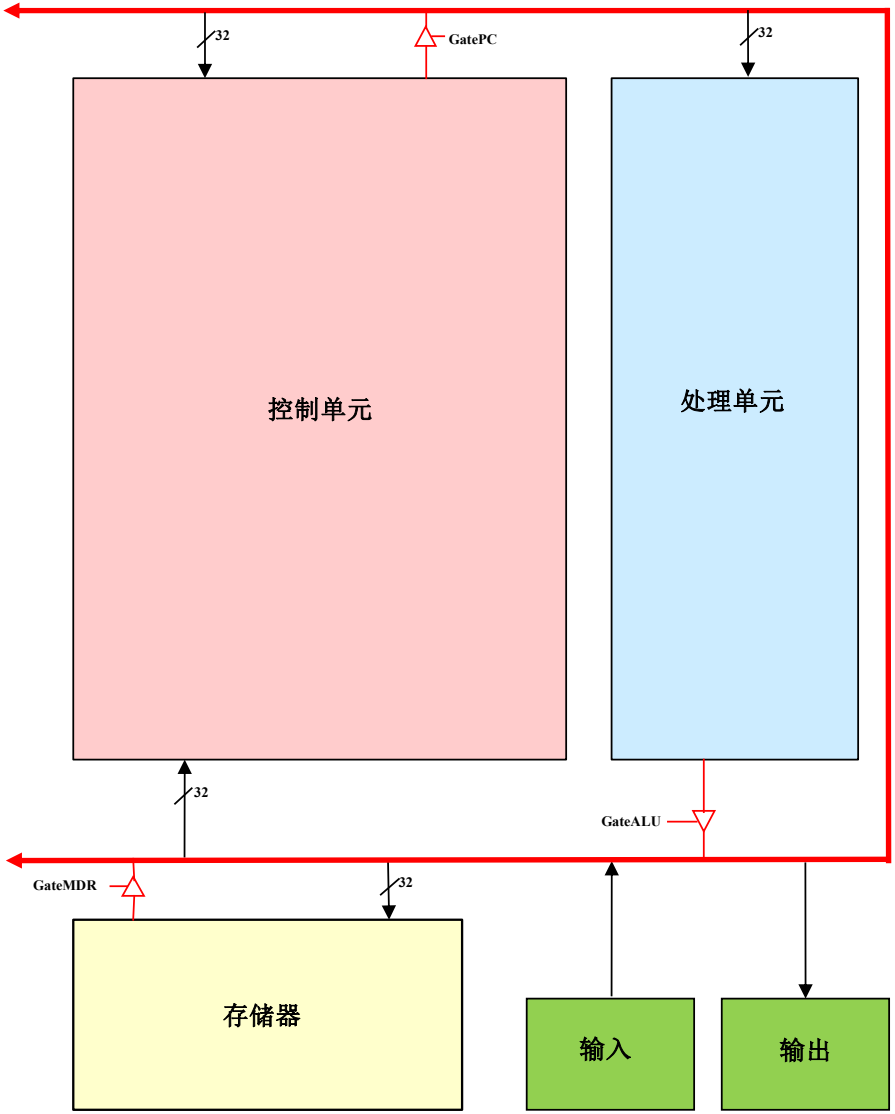
标志控制的循环

- 已知从x3000 0000开始存储了一列正整数，以0结束



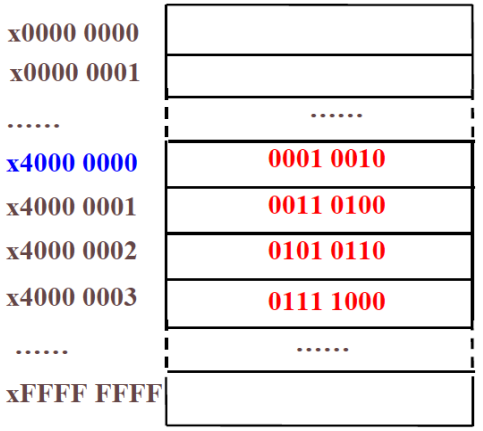
DLX指令处理


32
GatePC
PC
LD.PC
PC
32
[25:21]
[20:16]
[15:11]
DRMUX
DR.S
5
SR1
SR2
DR
寄存器堆
SR2OUT
SR1OUT
LD.REG
32
32
[25:0]
SEXT
32
EXT.S
LD.B
B
A
LD.A
4
ALUOp
4
PC
B.S
2
BMUX
AMUX
A.S
[5:0]
ALU控制器
4
CLK
[31:26]
有限
状态机
R
Z
控制
ALU
32
Z
LD.IR
IR
ALUOut
LD.ALU
32
GateALU
GateMDR
32
32
LD.MDR
MDR
MAR
LD.MAR
R
32
存储器
输入
输出
MEM.EN.R.W


32
GatePC
32
控制单元
处理单元
32
GateALU
GateMDR
32
存储器
输入
输出

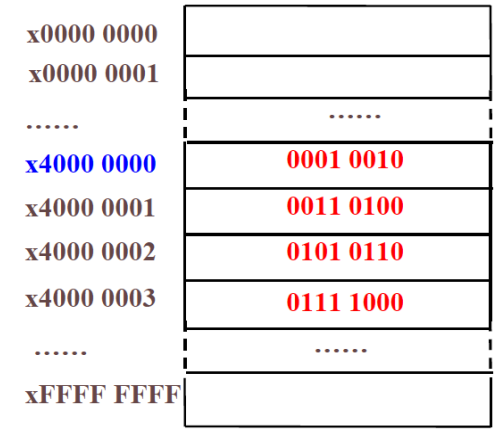
存储器——DLX

- 地址空间：2^{32}（即4G）个单元
  - 地址标识符是32位二进制数，4字节
  - 或8位十六进制数
- 寻址能力：8位，字节可寻址
  - 每个存储单元中的数据位数
  - ASCII码，只需访问1个存储单元
  - 访问字只需访问起始地址
  - 起始地址必须是4的倍数，即边界对齐



高位优先——DLX

- 数据的存储和排列顺序
  - 大端，Big Endian,高位优先
  - 字的高位字节存放在内存的低地址端
  - 低位字节存放在高地址端
    - x4000 0000~x4000 0003的4个存储单元存一个字
    - 当访问这个字时，只需使用其起始地址x4000 0000即可
    - 这个字是x1234 5678

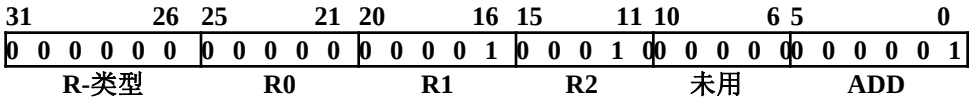


寄存器——DLX

- 在一个时钟周期内访问的附加临时存储空间
- DLX使用通用寄存器集（GPR）
- 每一个寄存器中的存储位数通常是32位的字
- 被唯一识别，同存储器的地址
  - 32个通用寄存器
  - 5位编码来识别
  - 分别被标记为R0、R1……R31
- 注意，R0寄存器中的数据必须为零

寄存器堆

R0	00000000000000000000000000000000
R1	00000000000000000000000000000011
R2	00000000000000000000000000000101
R3	00000000000000000000000000000111
R4	11111111111111111111111111111110
	……
R29	11111111111111111111111111111100
R30	11111111111111111111111111111010
R31	11111111111111111111111111111000



R0	00000000000000000000000000000000
R1	00000000000000000000000000000011
R2	00000000000000000000000000000011
R3	00000000000000000000000000000111
R4	11111111111111111111111111111110
R29	11111111111111111111111111111100
R30	11111111111111111111111111111010
R31	11111111111111111111111111111000

浮点寄存器

- 32个浮点寄存器，用于单精度或双精度计算
  - 使用5位编码来识别
  - 分别被标记为F0、F1……F31
- 每个寄存器也是32位存储能力
  - 单精度数只需1个浮点寄存器
  - 双精度数则需要2个浮点寄存器

指令处理

- 多时钟周期的实现方案
  - 指令的每一步将占用一个时钟周期
  - 不同的指令可能被分解为不同的步骤
  - 占用不同的时钟周期，“多周期”因此得名
- 在现代计算机中，时钟周期以纳秒（或称毫微秒，十亿分之一秒）为单位
  - 一个3.3GHz的处理器在1秒内有33亿个时钟周期
  - 即一个时钟周期只需0.303纳秒

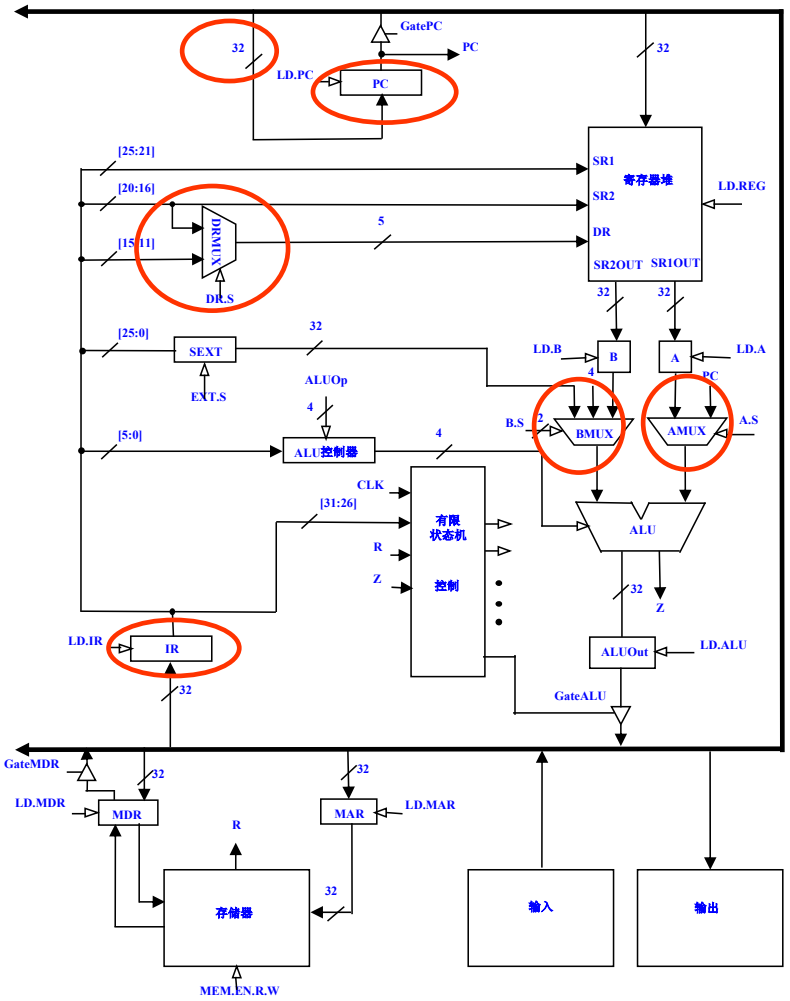
DLX指令执行阶段

- 按照DLX指令执行的步骤，将处理指令所需的操作划分为以下阶段：
  - 取指令（Instruction fetch）
  - 译码/取寄存器（Instruction decode/Register fetch）
  - 执行/有效地址/完成分支(Execution/Effective address/Branch completion)
  - 访问内存（Memory access）
  - 存储结果（Write-back）

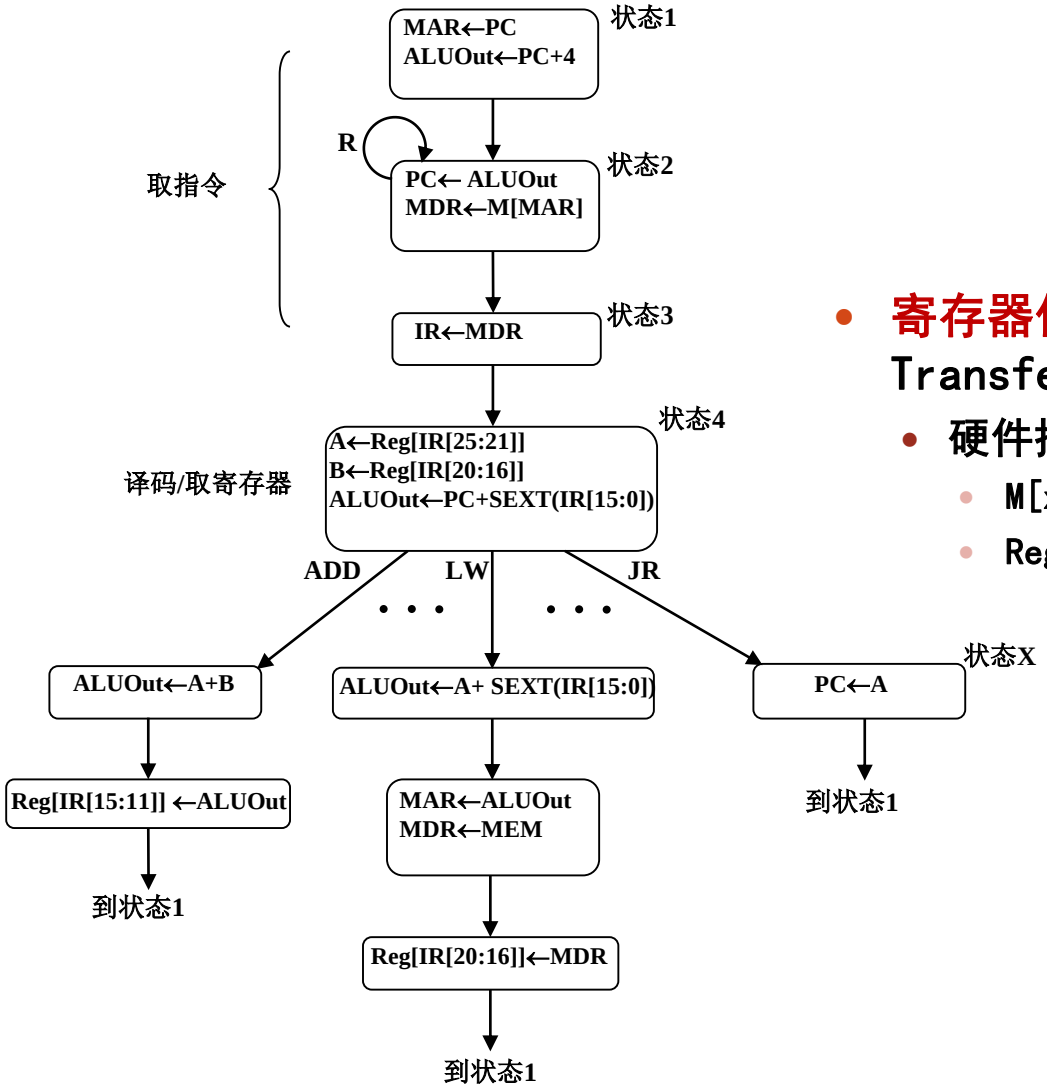
每条DLX指令需要其中的3到5个阶段

DLX的有限状态机

- 一条指令的执行可能包含3~5个阶段，每一个阶段还由一些步骤组成
- 执行执行每一个阶段的每一步都是由控制单元的有限状态机控制的
- 状态在时钟控制下发生转换



简化的状态图



- **寄存器传送语言** RTL（Register Transfer Language）
  - 硬件描述语言
    - M[xx]表示在存储器中xx地址的值
    - Reg[xx]表示寄存器xx的值

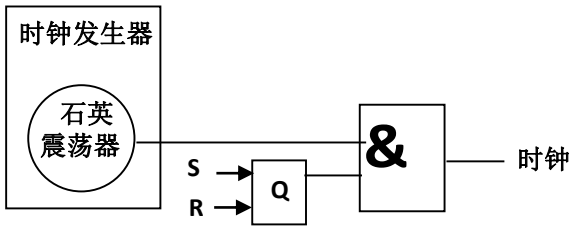
- 接下来，针对不同的指令，将进入不同的状态，直到该指令执行完毕，下一个状态就是返回到有限状态机的状态1
- 有限状态机一个时钟周期接一个时钟周期，控制每条指令的执行
- 既然每条指令的执行都是以返回到状态1结束，有限状态机就可以一个周期接一个周期的，控制整个计算机程序的执行

停止时钟

- 计算机以**时钟周期**为单位**持续**运行指令
- 用户程序通常在操作系统的控制下运行
  - 操作系统本身也是计算机程序
- 当一个用户程序执行完毕，计算机可以再次开始运行操作系统
- 也可以在操作系统控制下，运行另一个用户程序
- 一直持续下去直到断掉计算机的电源吗？
- **停止时钟**，停止指令的运行

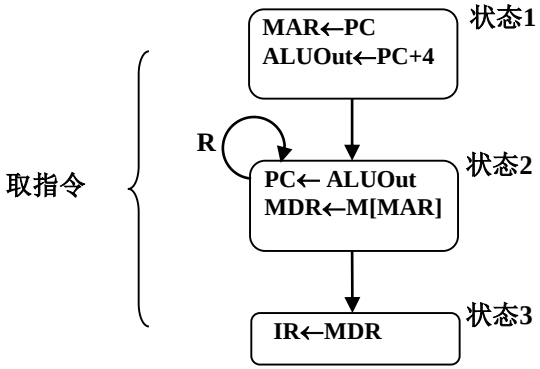
停止时钟

- 如果运行锁在状态1（即Q=1），时钟电路的输出和时钟发生器的输出是一样的
- 如果运行锁在状态0（即Q=0），时钟电路的输出就为0
- 只需要把运行锁清0，就可以停止指令运行



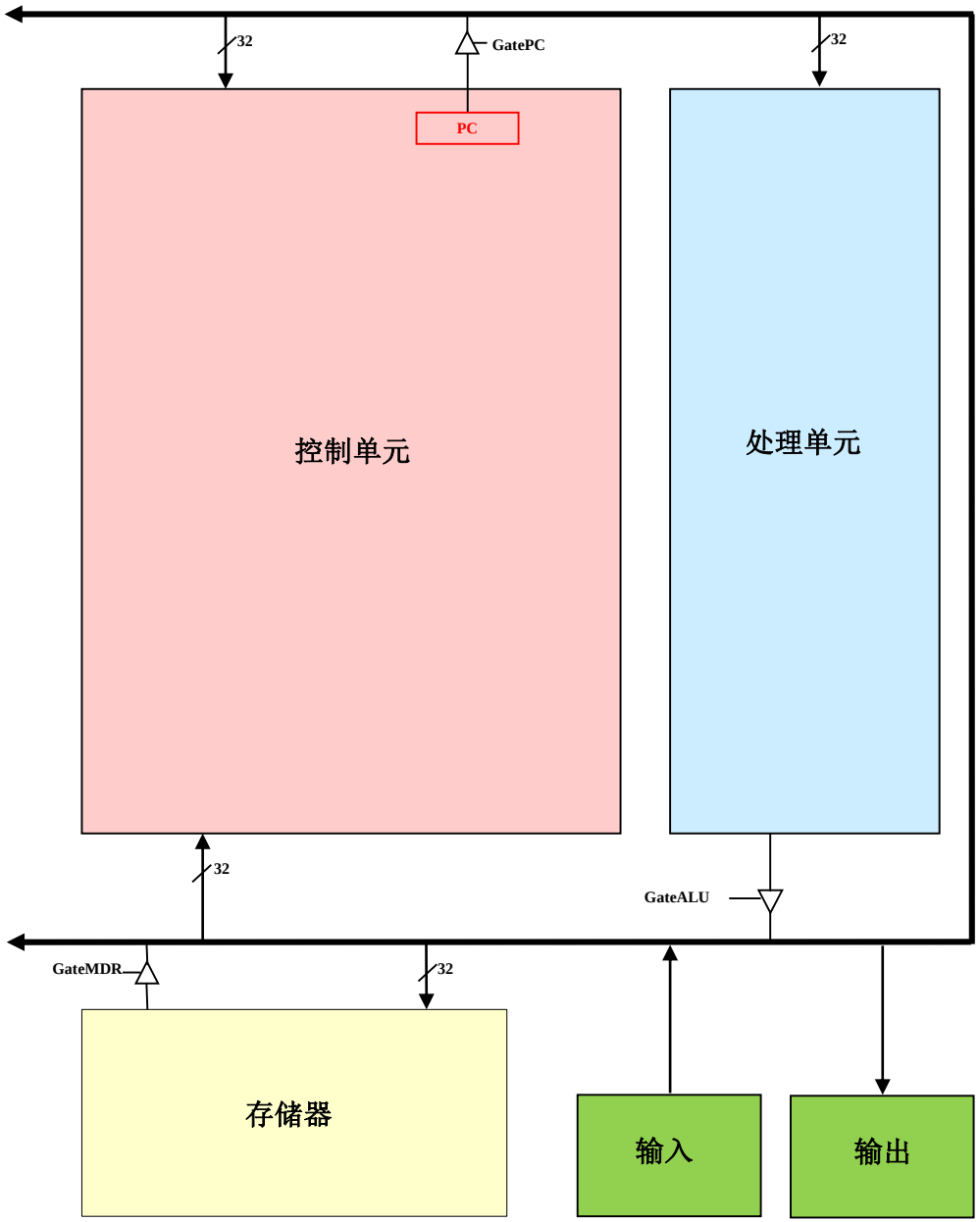
1、取指令阶段

- 需要多个时钟周期

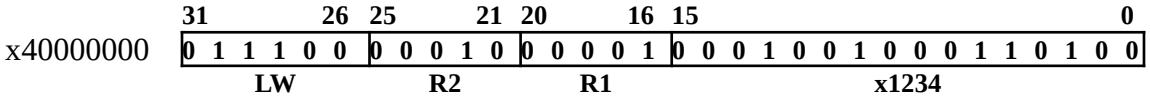


1、取指令阶段

- 从存储器中获得下一条指令，放在控制单元的指令寄存器中
- 为了执行下一条指令的任务，必须先确定它位于哪里
- 程序计数器（PC）包含着下一条指令的起始地址（DLX指令由32位组成，需要4个连续的存储单元）

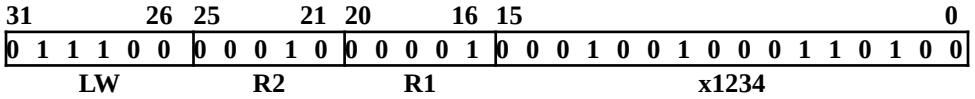


1、取指令阶段



- 示例：
  - 假设PC包含的是x40000000
  - 一条指令32位二进制数据，4个字节
  - 即下一条指令的存储单元x40000000~x40000003
  - 指令内容011100 00010 00001 0001 0010 0011 0100

1、取指令阶段

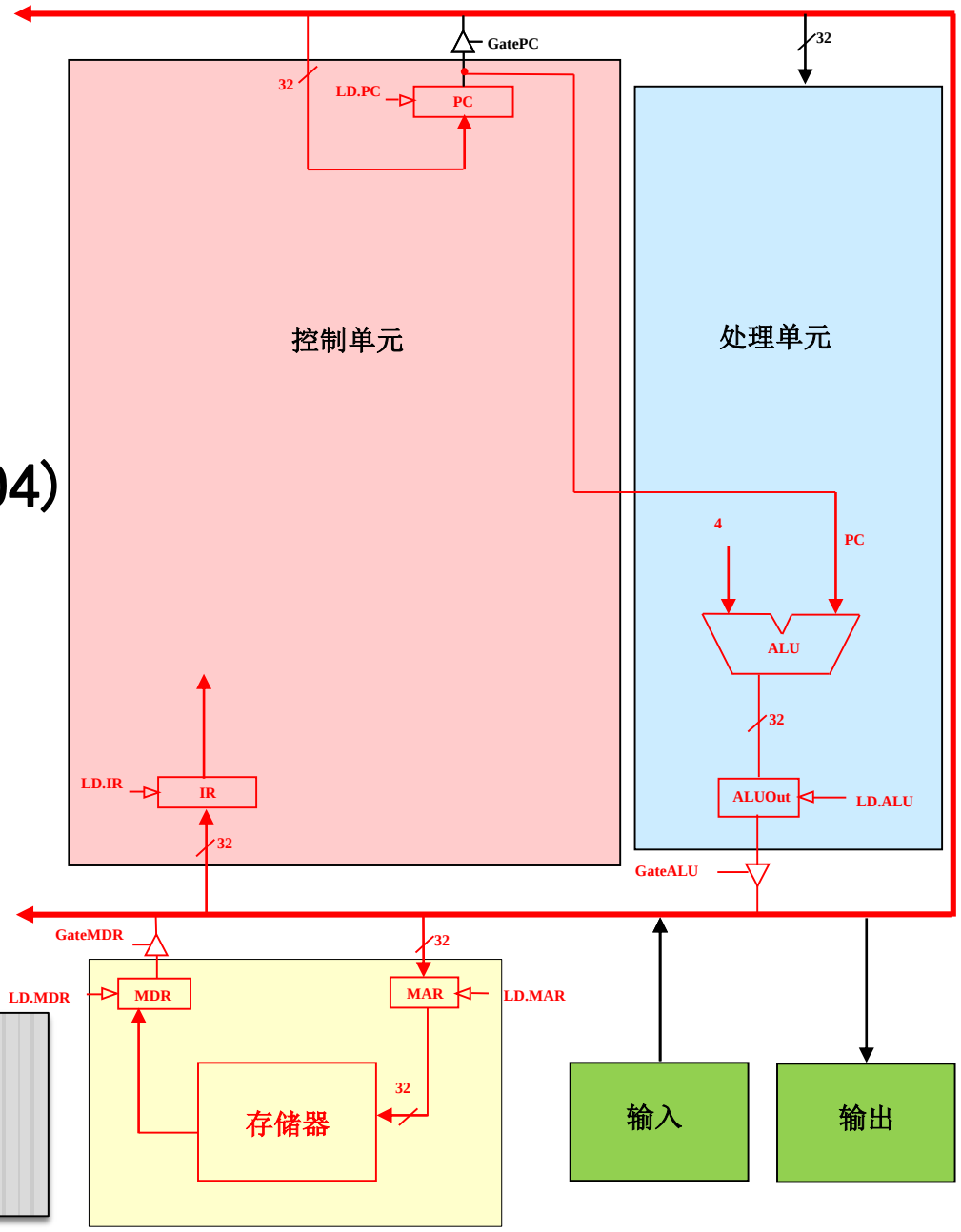


- **第一个时钟周期（状态1）**
  - PC中的内容通过总线被加载到MAR中
  - 在ALU中执行PC+4的运算
- **下一个时钟周期（状态2）**（如果存储器可以在一个时钟周期里提供信息）
  - 存储器被读取，指令内容011100 00010 00001 0001 0010 0011 0100被加载到MDR
  - PC+4的结果加载到PC（x4000 0004）
- **再下一个时钟周期（状态3）**
  - MDR中的值被加载到指令寄存器（IR）

1、取指令阶段

- **时钟周期1（状态1）**
  - MAR←PC(x40000000)
  - ALUOut←PC+4(x40000004)
- **时钟周期2（状态2）**
  - MDR←Mem[MAR]
  - PC←ALUOut(x40000004)
- **时钟周期3（状态3）**
  - IR←MDR(LW指令内容)

从总线获得数据的组件，需将LD.X信号设为1，才能得到信息，如LD.MAR、LD.IR。



状态1

- 第一个时钟周期

  MAR←PC

  ALUOut←PC+4

  - 控制信号

    GatePC=1

    LD. MAR=1

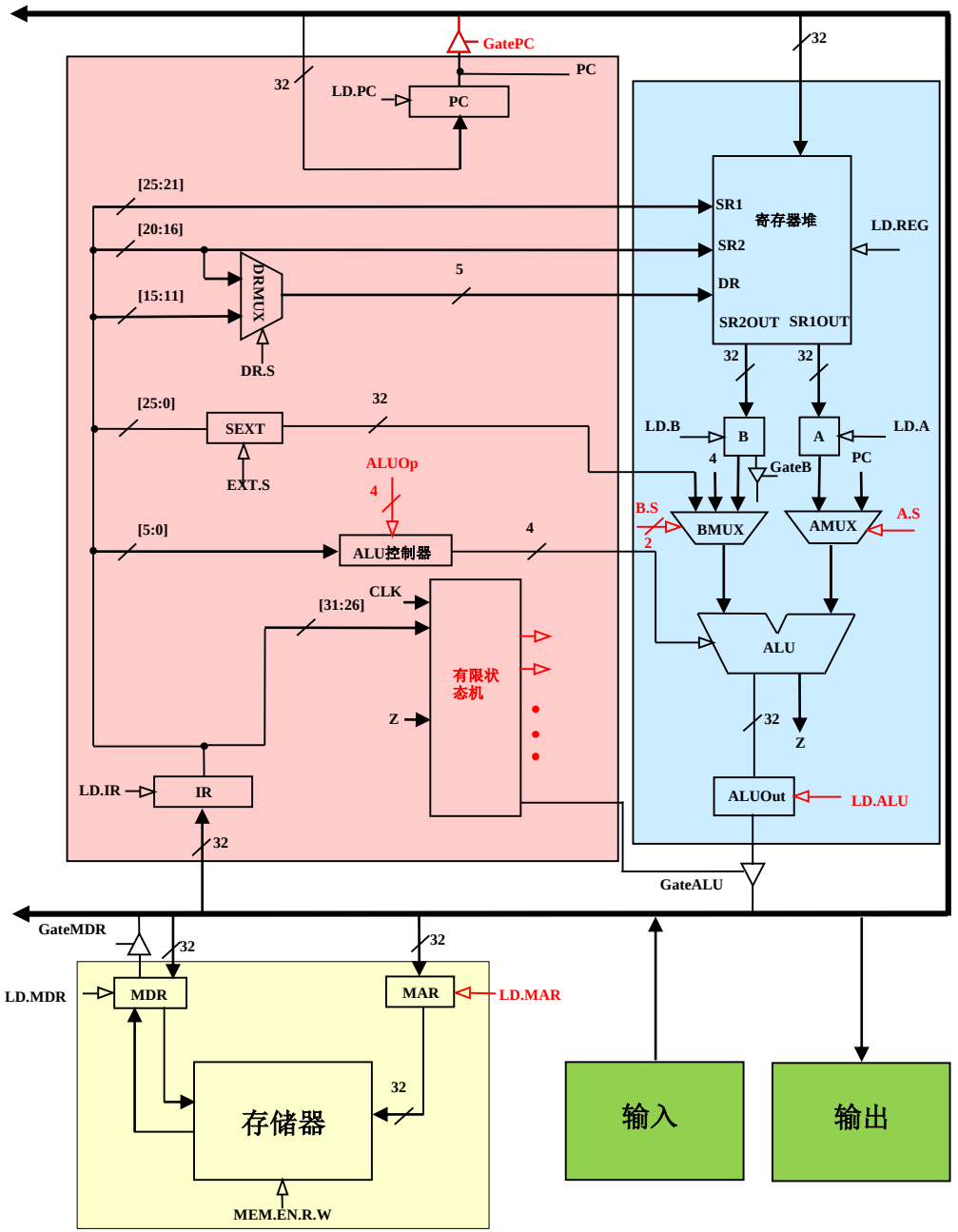
    A. S=1

    B. S=01

    ALUOp=0001

    LD. ALU=1

    ……



状态2

- 第二个时钟周期

PC ← ALUOut

MDR ← M[MAR]

- 控制信号

GateALU=1

LD.PC=1

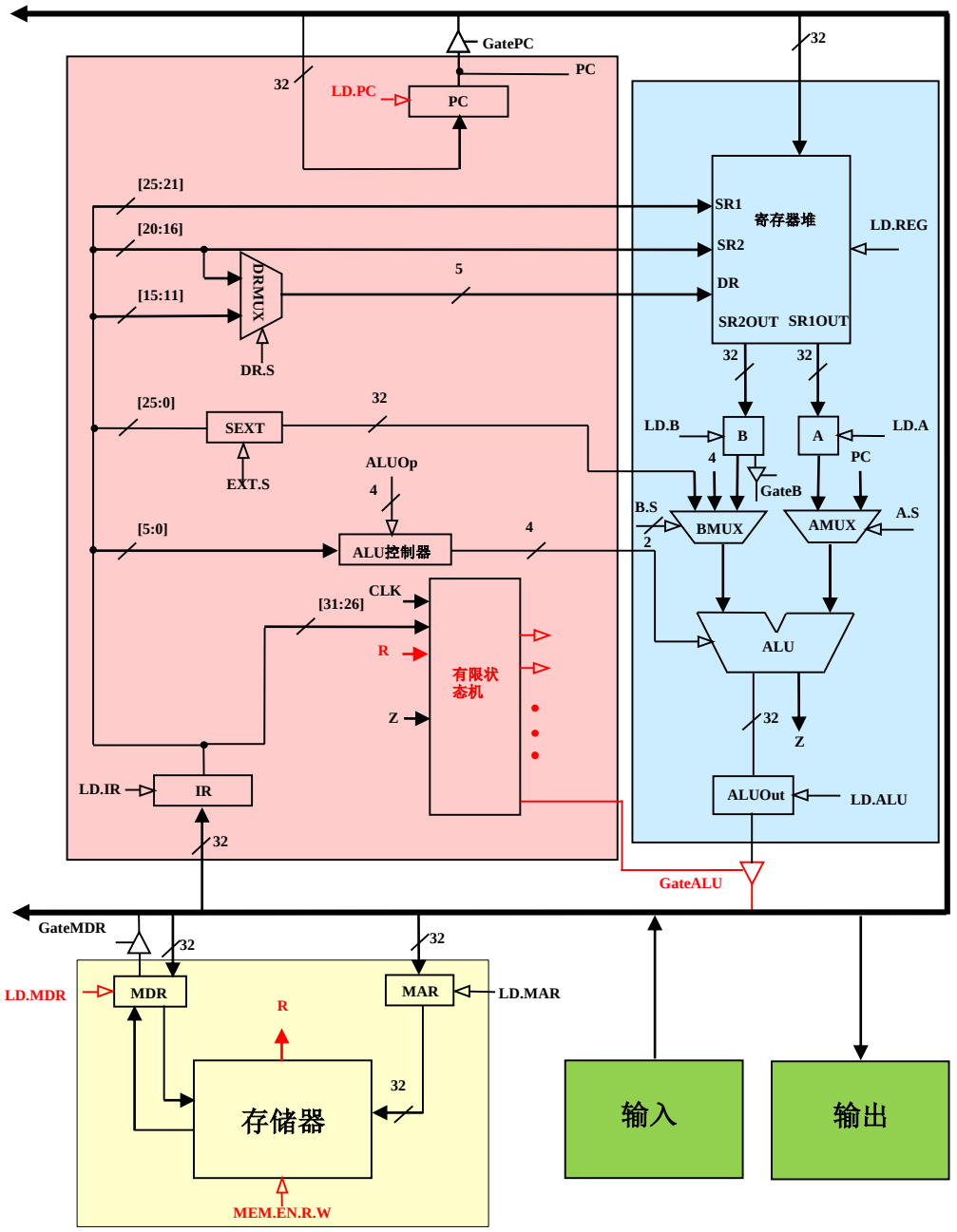
MEM.EN.R.W=0

LD.MDR=1

就绪信号（Ready）

读完设为1

……



状态3

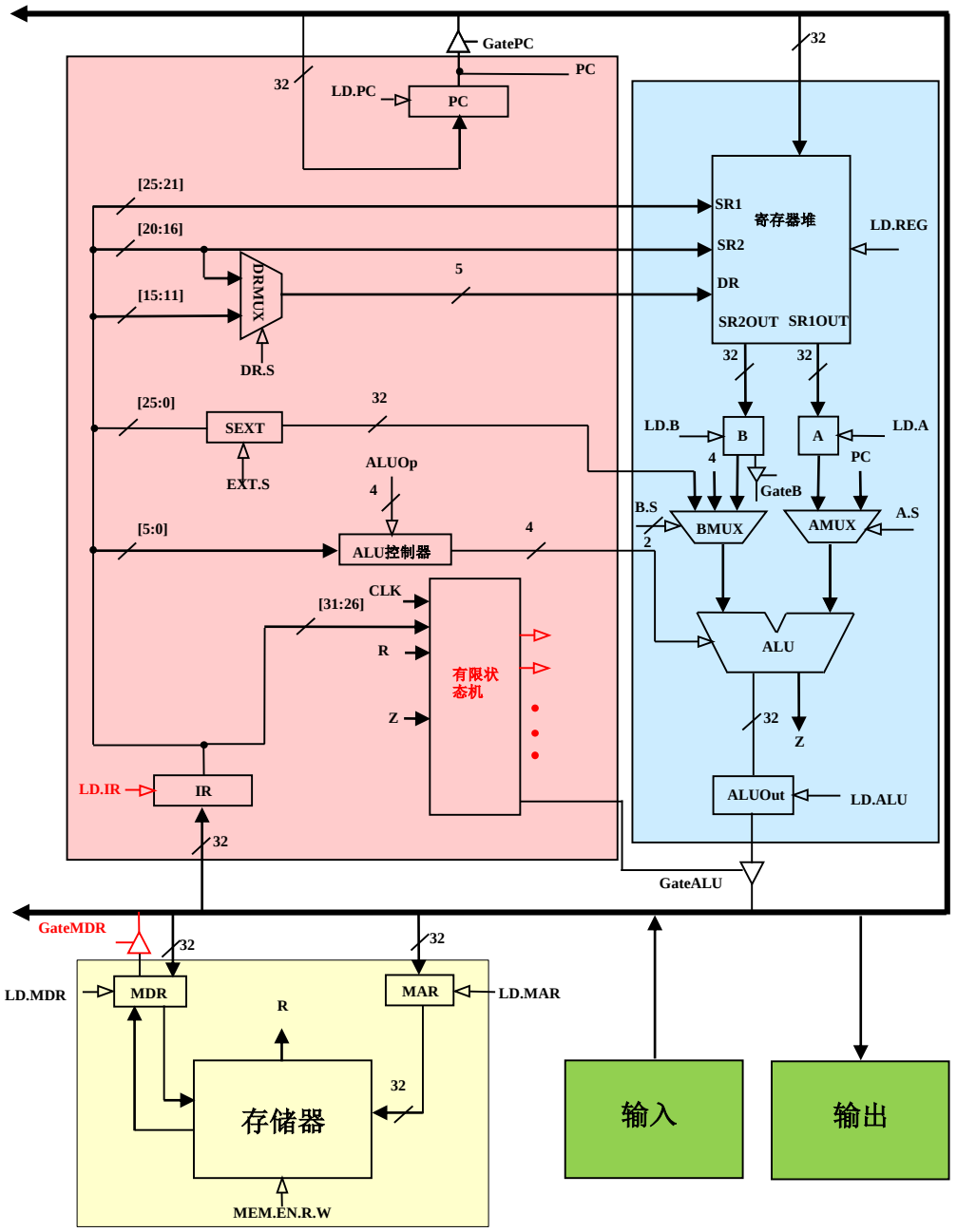
IR←MDR

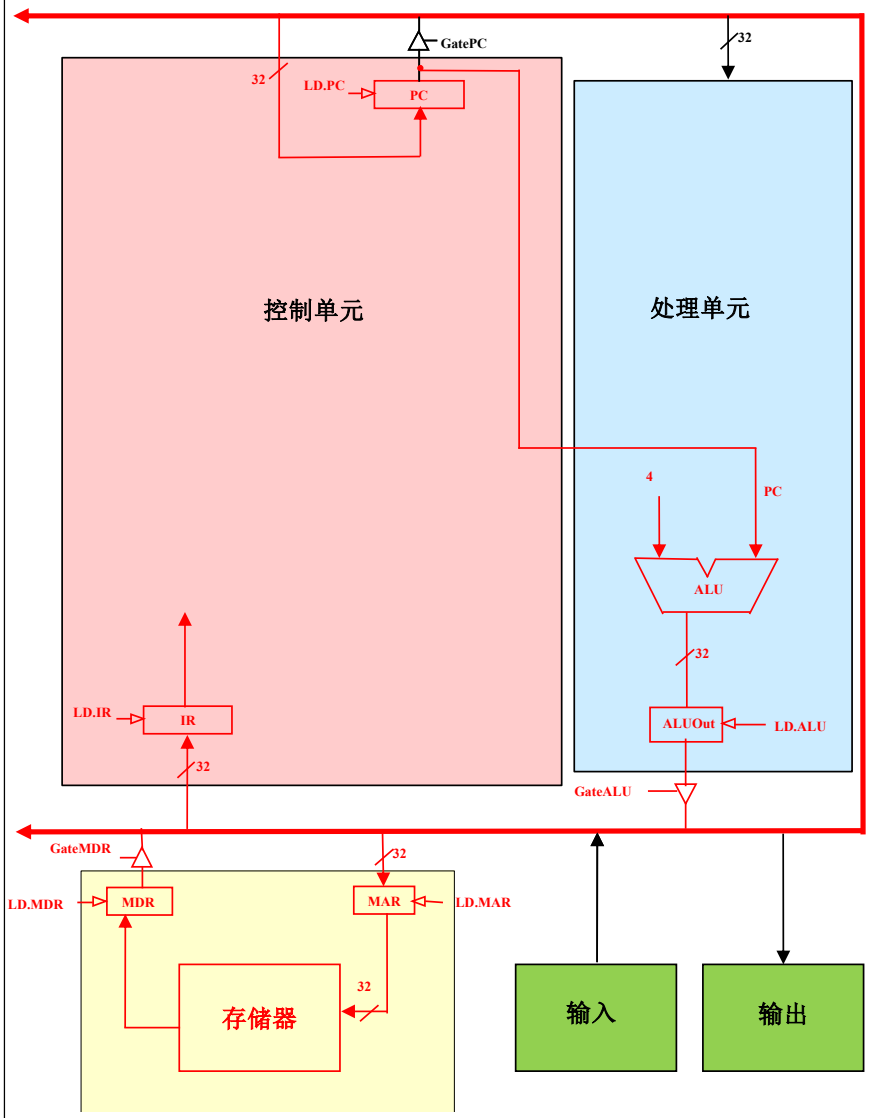
- 控制信号

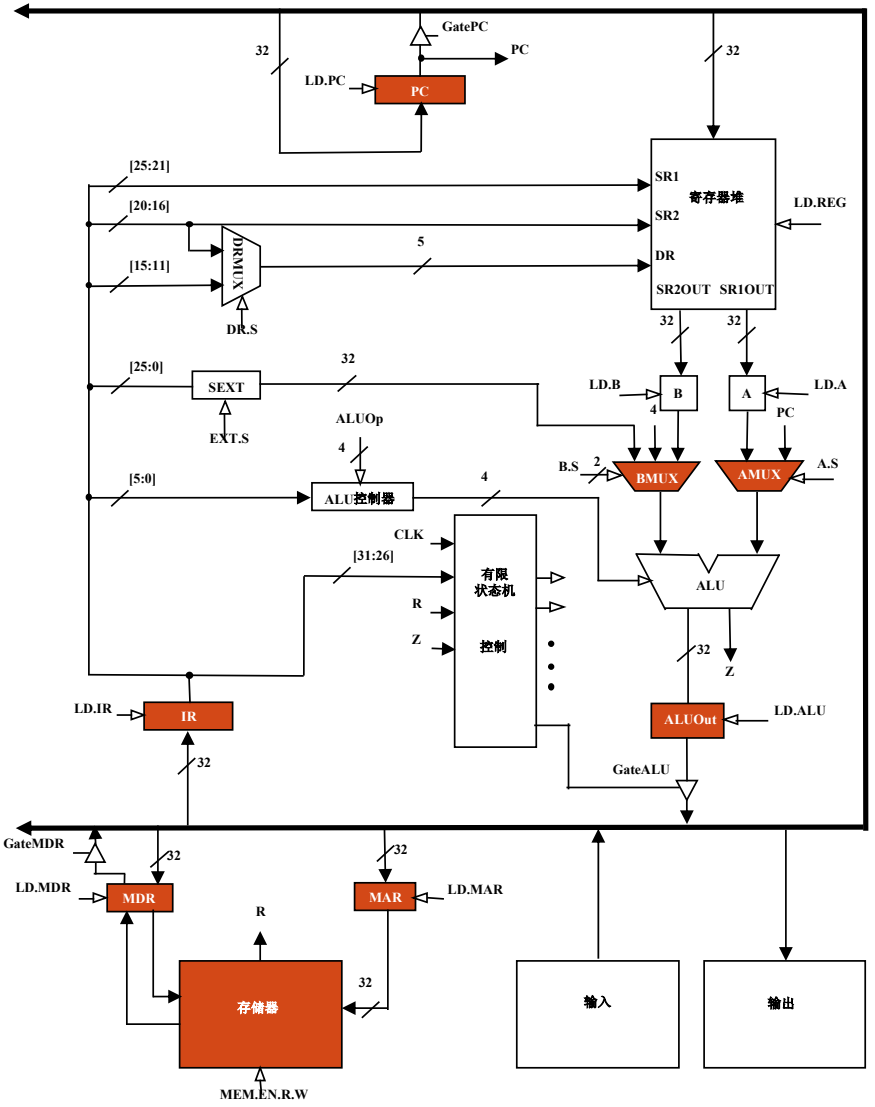
  GateMDR=1

  LD.IR=1

  ……

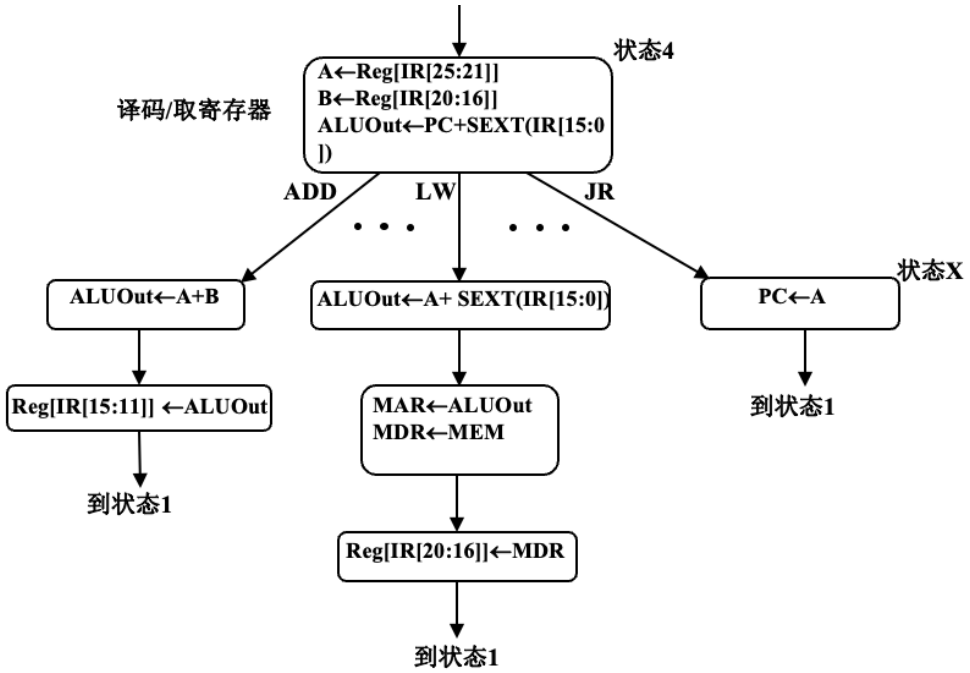



控制单元
处理单元
GatePC
32
LD.PC
PC
4
ALU
ALUOut
LD.ALU
GateALU
LD.IR
IR
GateMDR
LD.MDR
MDR
MAR
LD.MAR
存储器
输入
输出

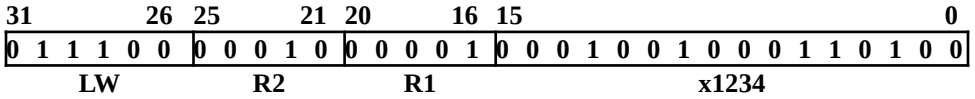

GatePC
PC
LD.PC
[25:21]
[20:16]
[15:11]
DRMUX
DR.S
寄存器堆
SR1
SR2
DR
SR2OUT
SR1OUT
LD.REG
[25:0]
SEXT
EXT.S
LD.B
B
A
LD.A
ALUOp
BMUX
AMUX
B.S
A.S
[5:0]
ALU控制器
CLK
[31:26]
有限状态机
控制
R
Z
ALU
IR
LD.IR
ALUOut
LD.ALU
GateALU
GateMDR
LD.MDR
MDR
MAR
LD.MAR
存储器
MEM.EN.R.W
输入
输出

2、译码/取寄存器阶段

- 使用指令的操作码（IR[31:26]），有限状态机就能够到达下一个适当的状态



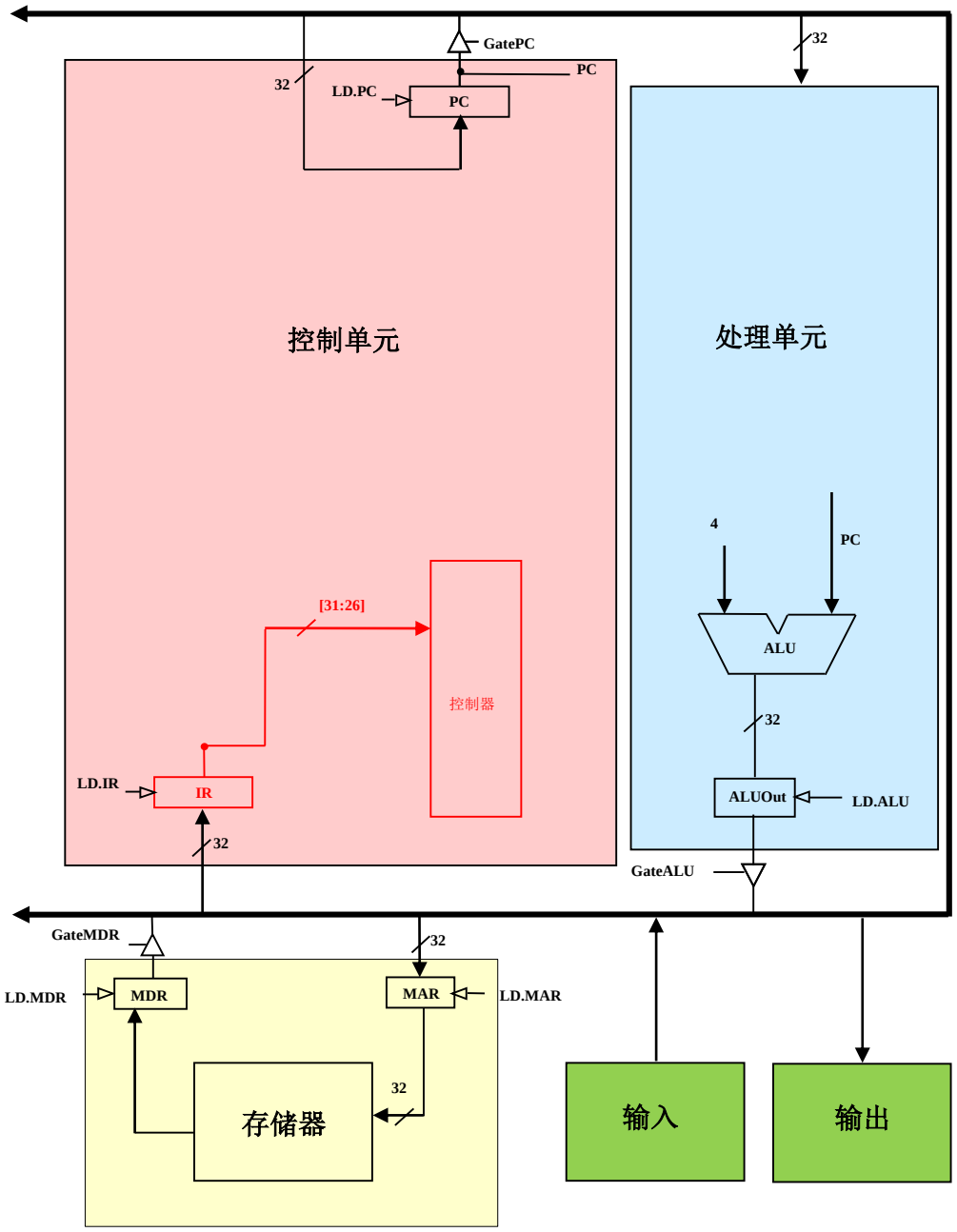
2、译码/取寄存器



- 下一周期（状态4）
  - IR中指令操作码被译码：确定下一步要去做什么
    - IR[31:26]对应的操作码是011100，为LW指令
    - 控制逻辑发出正确的控制信号（空心箭头），从而控制指令的执行
  - 取寄存器：为后面阶段获取操作数
    - 读取IR[25:21]的内容(即R2)，写到寄存器A中
    - 读取IR[20:16]的内容(即R1)，写到寄存器B中
    - 在ALU中执行PC+SEXT(IR[15:0])，结果存储于ALUOut中

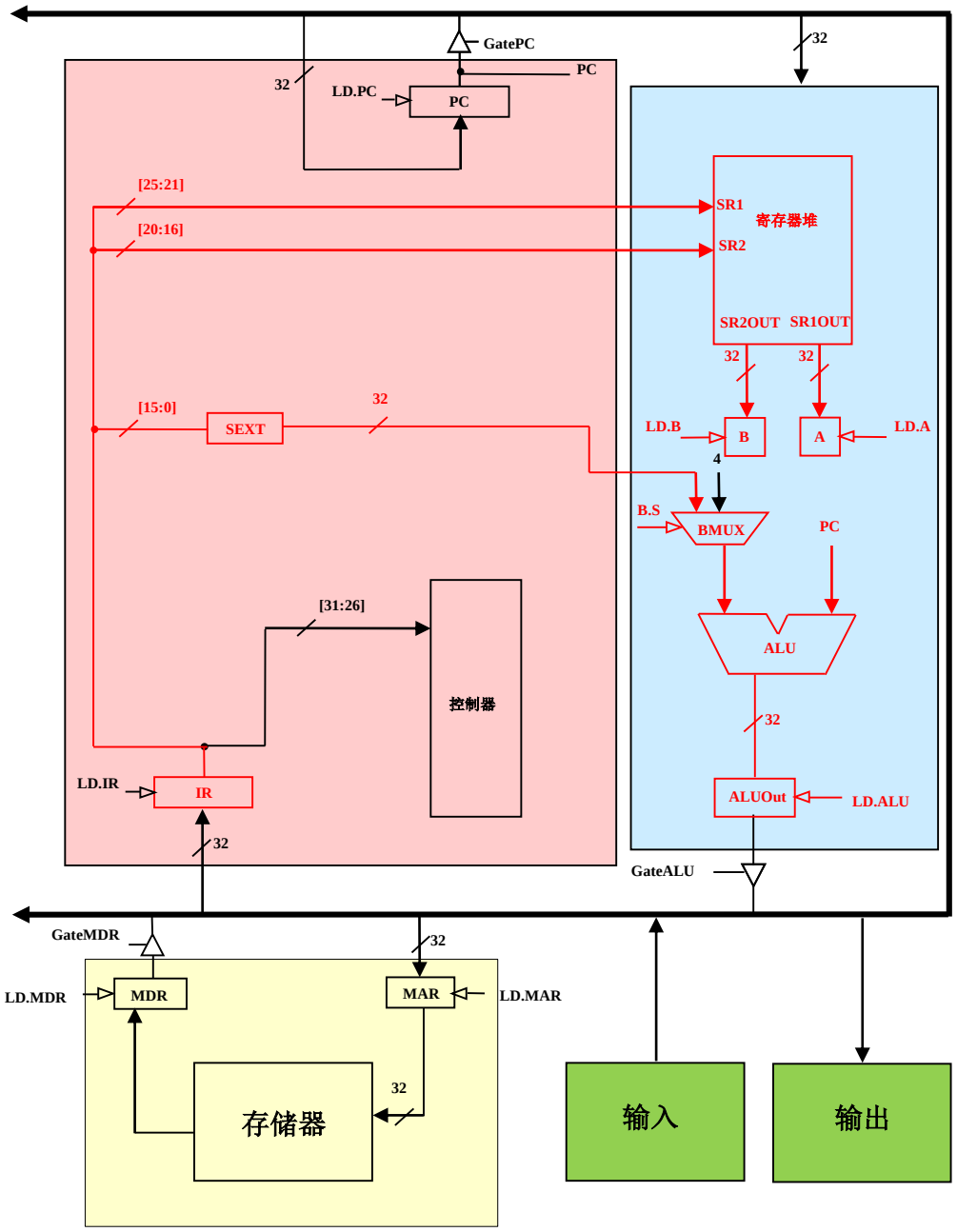
2、译码/取寄存器阶段(1)

- 译码：
  - IR[31:26]
  - 6-64译码器
  - 依据译码器输出为1的线，决定了余下的26位需要做哪些工作



2、译码/取寄存器阶段(2)

- 取寄存器
  - A←(IR[25:21])
  - B←(IR[20:16])
  - ALUOut←PC+SEXT(IR[15:0])



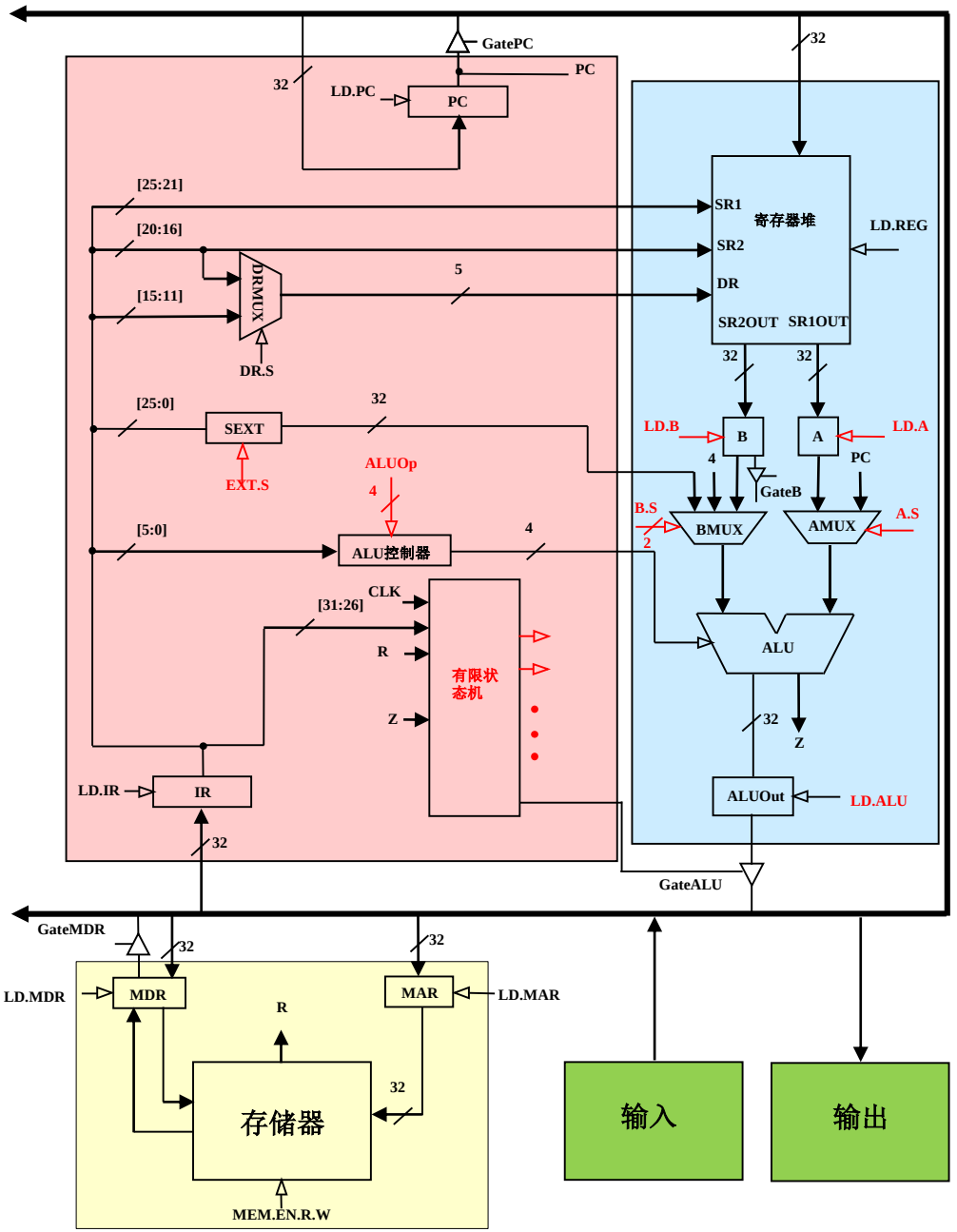
2、译码/取寄存器阶段

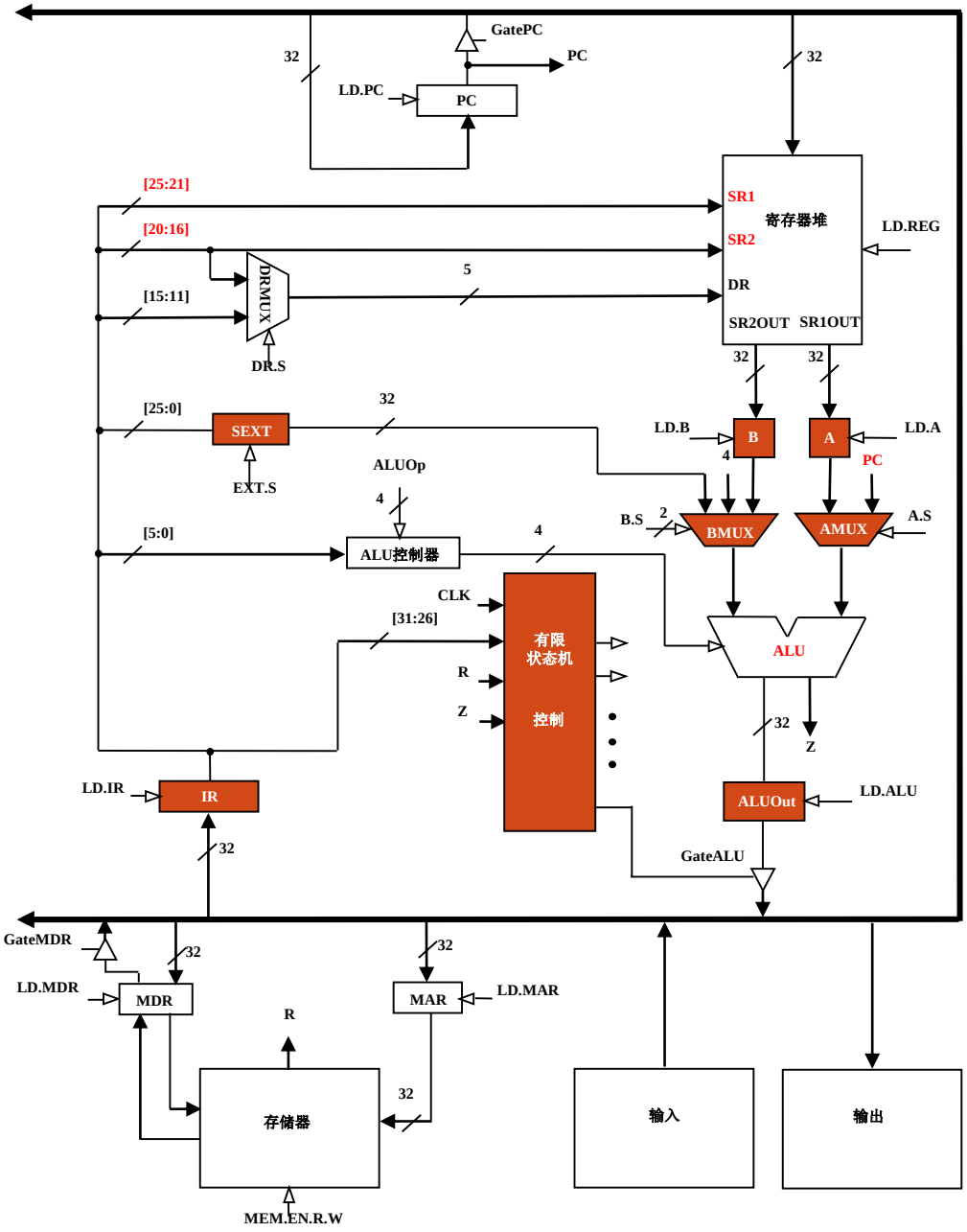
- 状态4

A←Reg[IR[25:21]]
B←Reg[IR[20:16]]
ALUOut←PC+SEXT(IR[15:0])

- 控制信号

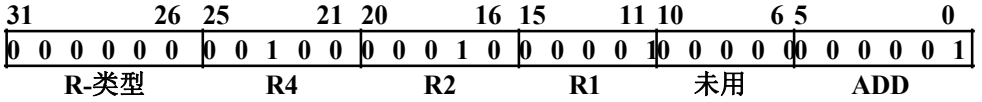
LD. A=1
LD. B=1
A. S=1
B. S=00
EXT. S=0
ALUOp=0001
LD. ALU=1
……



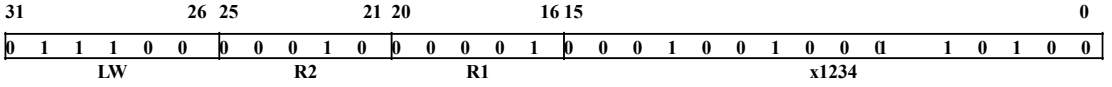

GatePC
PC
32
LD.PC
PC
32
[25:21]
[20:16]
[15:11]
DRMUX
DR.S
5
SR1
SR2
DR
寄存器堆
LD.REG
SR2OUT
SR1OUT
32
32
[25:0]
SEXT
EXT.S
32
LD.B
B
4
A
LD.A
PC
ALUOp
4
[5:0]
ALU控制器
4
B.S
2
BMUX
AMUX
A.S
CLK
[31:26]
R
Z
有限
状态机
控制
ALU
32
Z
LD.IR
IR
32
ALUOut
LD.ALU
GateALU
GateMDR
32
LD.MDR
MDR
32
MAR
LD.MAR
R
存储器
32
MEM.EN.R.W
输入
输出

2、译码/取寄存器—示例

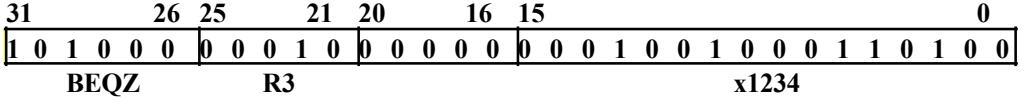
- ADD指令



  - 将从IR[25:21]和IR[20:16]所指示的R4和R2中获得源操作数，传给ALU的A和B寄存器；
  - 计算PC与IR[15:0]符号扩展的和，结果存储于ALUOut寄存器。
- LW指令



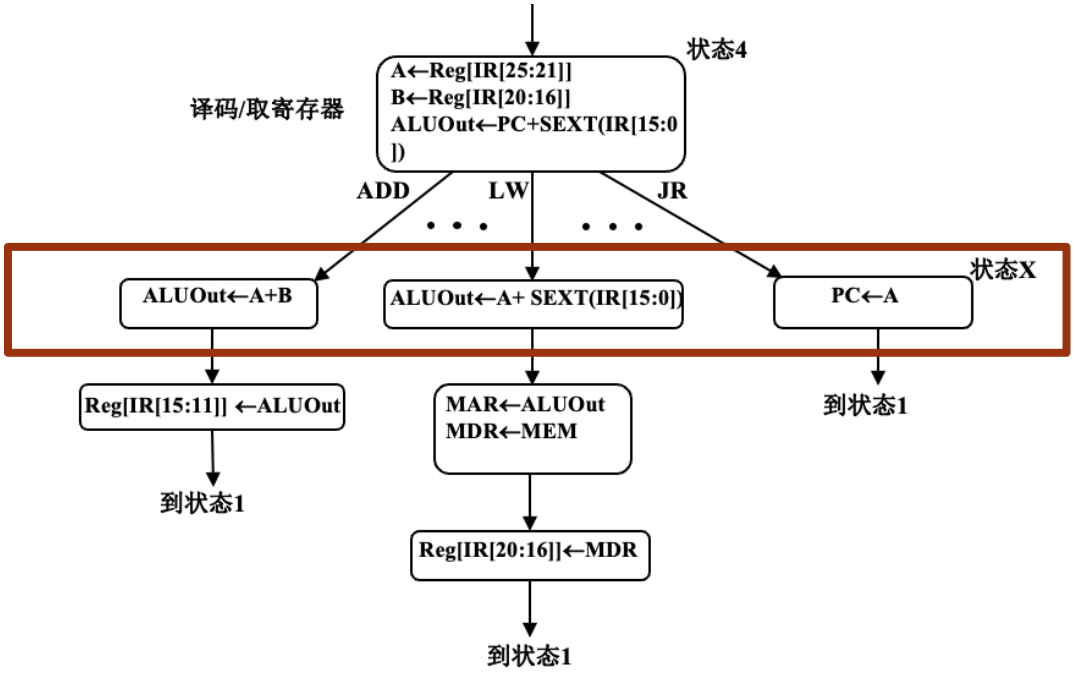
  - 将从IR[25:21]和IR[20:16]所指示的R2和R1中获得源操作数，传给ALU的A寄存器和B寄存器；
  - 计算PC与IR[15:0]符号扩展的和，结果存储于ALUOut寄存器。
- BEQZ指令



  - 将从IR[25:21] 和IR[20:16]所指示的R3和R0中获得源操作数，传给ALU的A寄存器和B寄存器；
  - 计算PC与IR[15:0]符号扩展的和，结果存储于ALUOut寄存器。
- 有的操作的结果在后面的阶段并不会用到，但这并不浪费时间，因为这些操作是同时进行的

3、执行/有效地址/完成分支

- 根据译码产生的控制信号（空心箭头）执行算术或逻辑运算
- 或计算有效地址
- 或完成分支

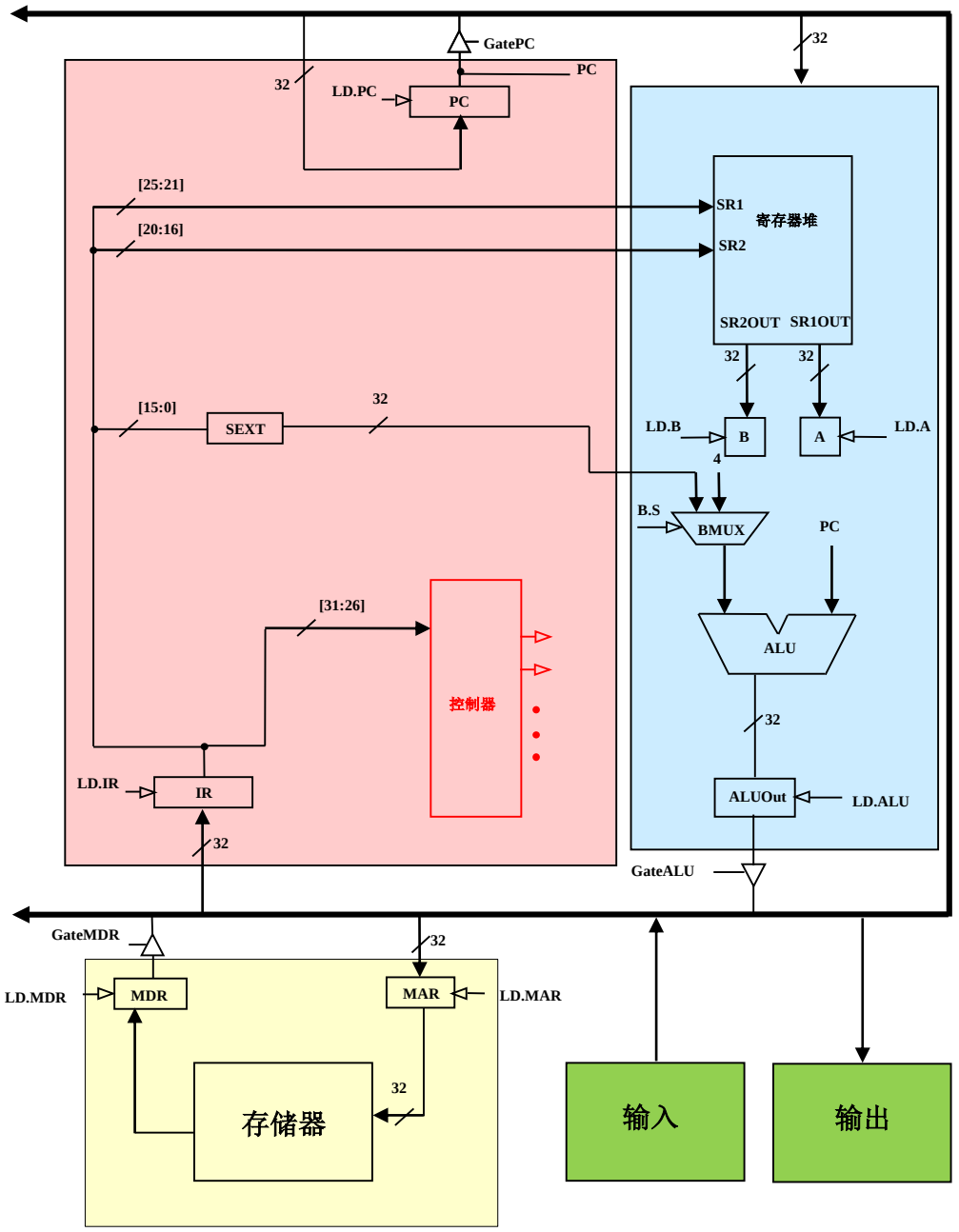


3、执行/有效地址/完成分支

- 下一周期
  - 根据译码产生的控制信号，对上一阶段得到的A寄存器和B寄存器的值执行算术/逻辑运算
  - 或计算出处理指令所在存储单元的地址（有效地址）
    - 有限状态机选择来自寄存器A（即基址寄存器R2）和来自IR[15:0]符号扩展的值
      - 将AMUX和BMUX的选择信号A.S和B.S分别设为0和00，
      - 将EXT.S设为0（SEXT逻辑将执行IR[15:0]的符号扩展操作）
    - 将ALUOp设为0001（4位控制信号），在ALU中进行加法运算，即计算“基址+偏移量”，形成有效地址
    - 将LD.ALU设为1，结果存储于ALUOut寄存器中
  - 或者完成分支跳转

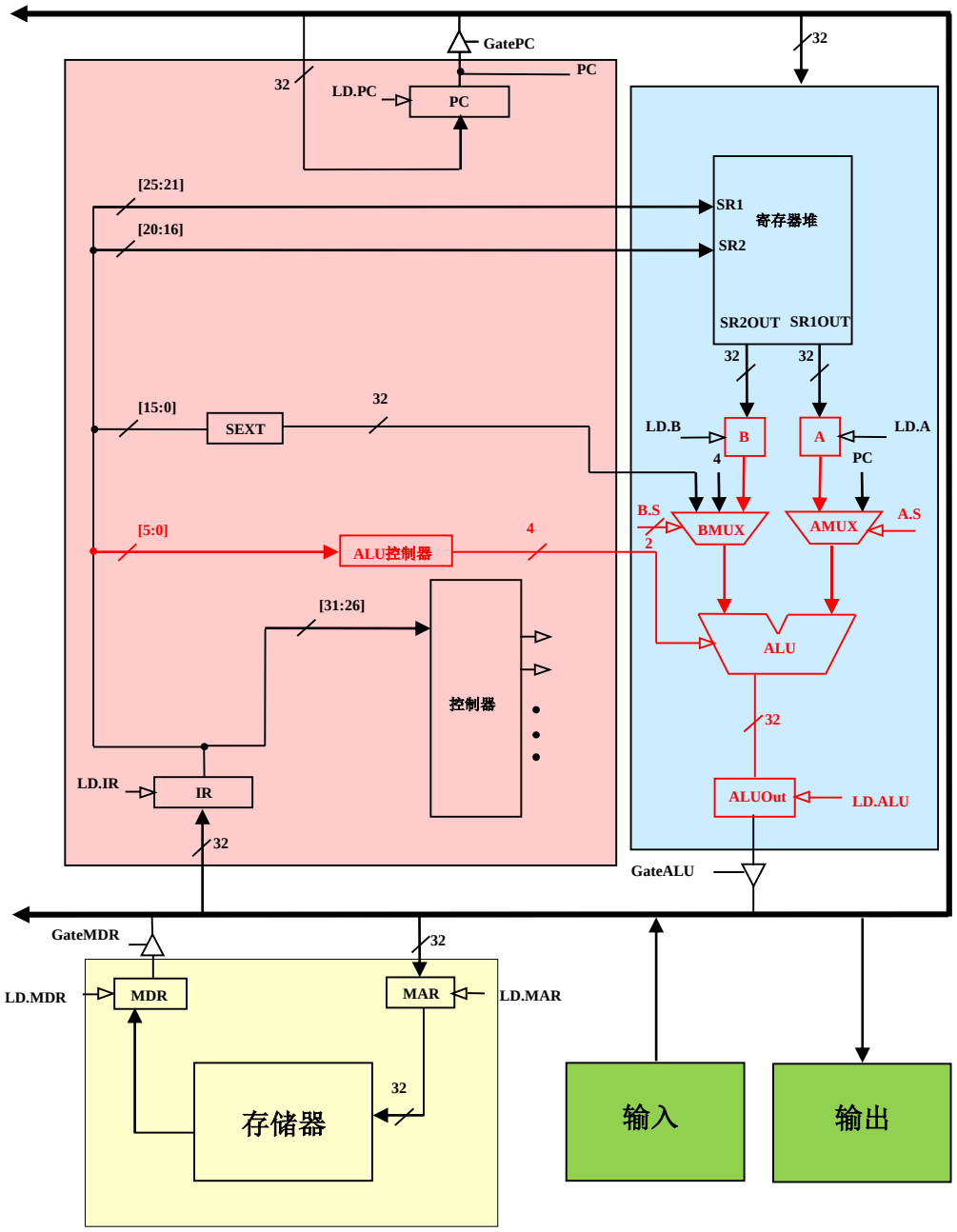
3、执行/有效地址/完成分支

- 根据译码产生的控制信号（空心箭头）
  - ALUOut ← A Op B
  - 或计算有效地址；
  - 或完成分支



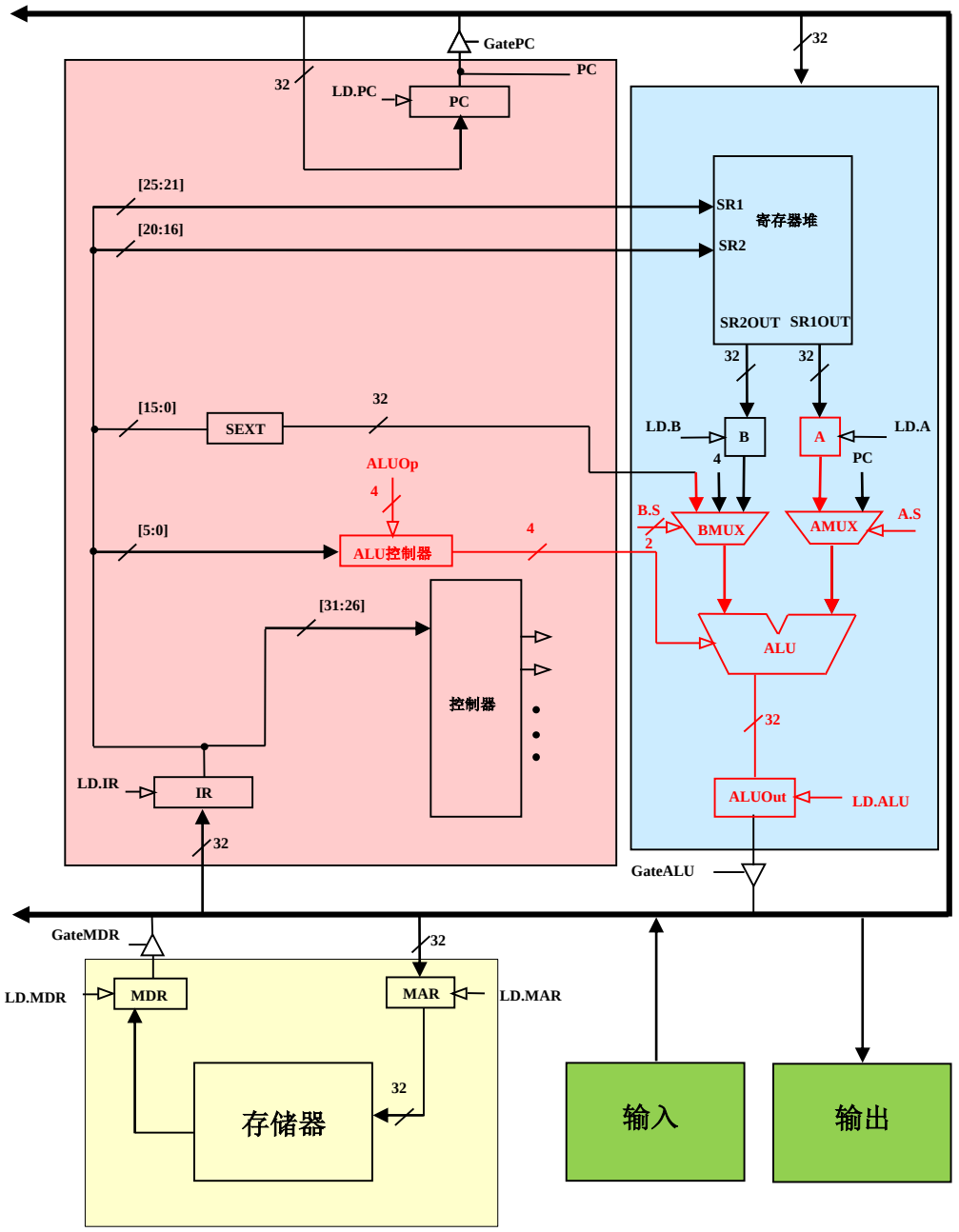
ADD指令

- ALUOut ← A + B
  - A. S=0
  - B. S=10
  - ALUOp=0001
  - LD. ALU=1



LW指令

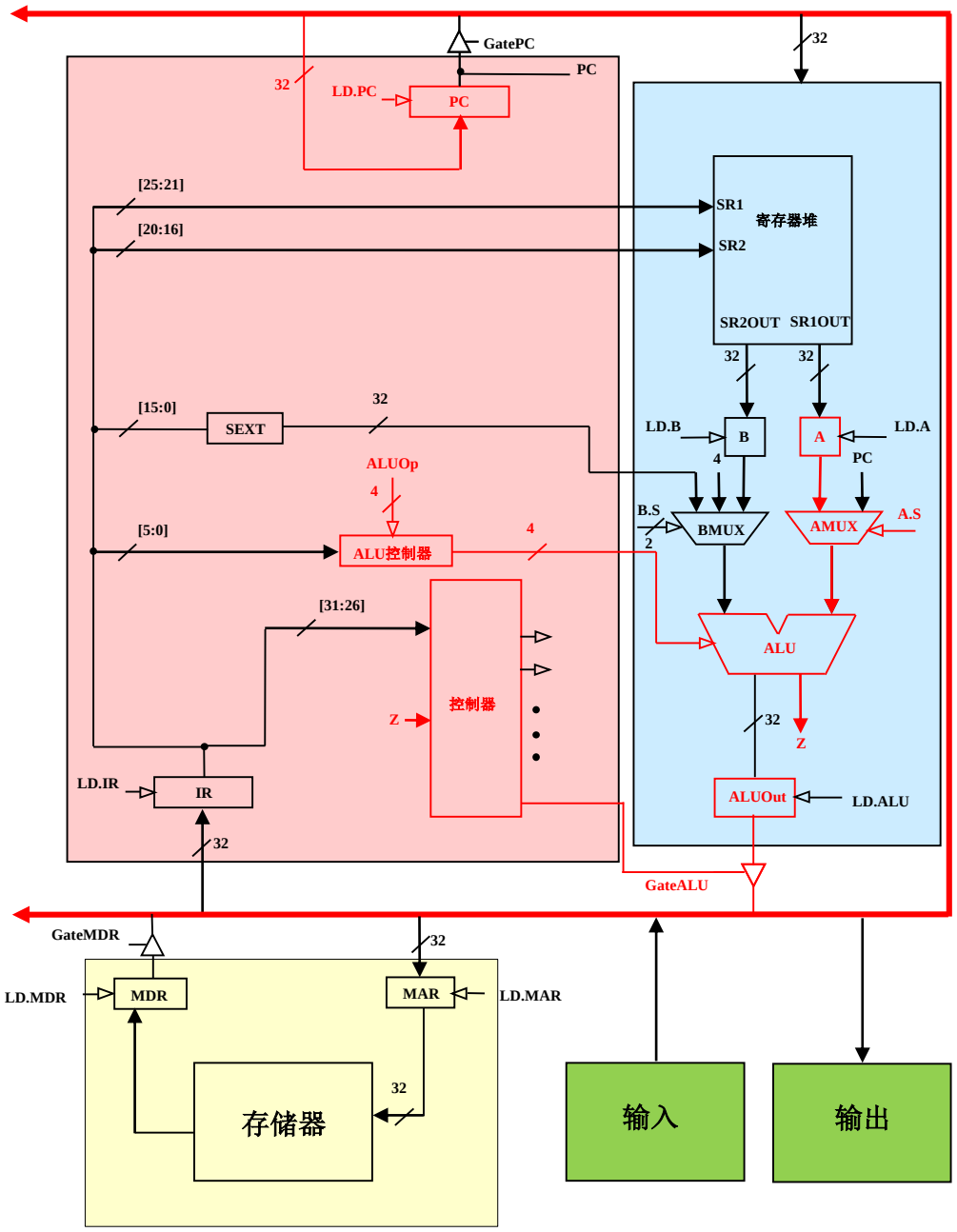
- ALUOut ← A + SEXT(IR[15:0])
  - A.S=0
  - B.S=00
  - EXT.S=0
  - ALUOp=0001
  - LD.ALU=1



BEQZ指令

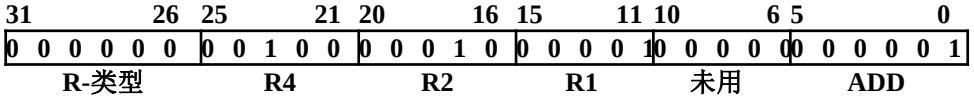
- Z ← A == 0
- if(Z)

  PC ← ALUOut
  - A.S=0
  - ALUOp控制加载上一阶段计算的值给ALUOut
  - GateALU=1
  - LD.PC=1

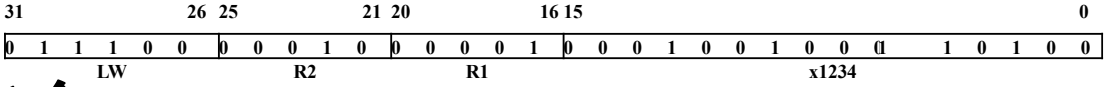


3、执行/有效地址/完成分支—示例

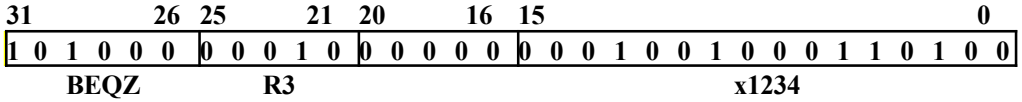
- ADD指令
  - 在ALU中进行加法运算，得到加法运算结果，存储于ALUOut寄存器中

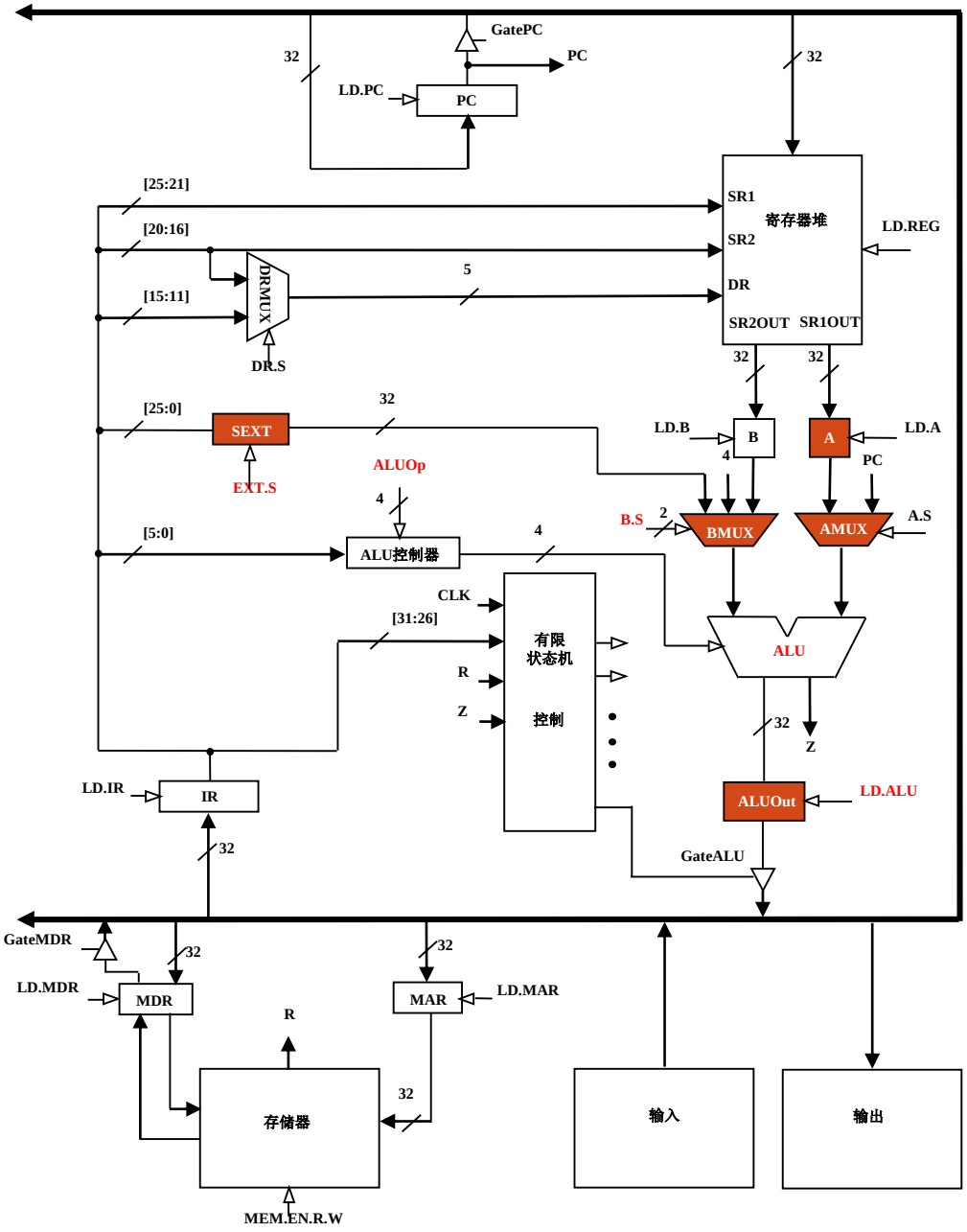


- LW指令
  - 选择IR[15:0]符号扩展的结果，在ALU中与A寄存器进行加法运算，得到一个有效地址，存储于ALUOut寄存器中



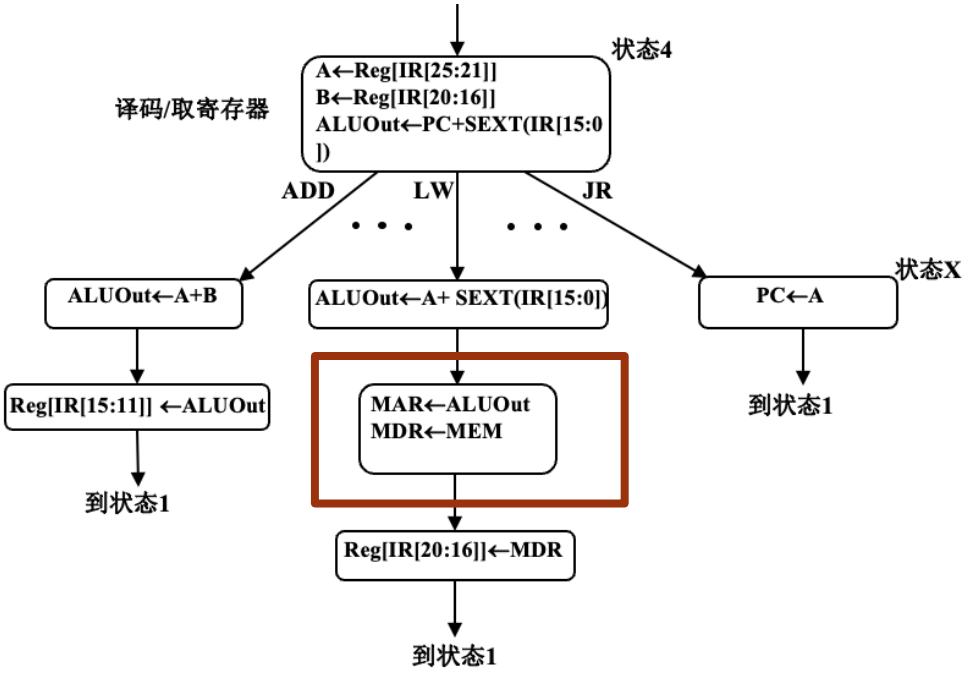
- BEQZ指令
  - 如果A寄存器中的值为零，PC被ALUOut寄存器中的值（上一阶段计算所得）加载，否则保持不变




GatePC
PC
32
LD.PC
PC
32
[25:21]
[20:16]
[15:11]
DRMUX
5
DR.S
SR1
SR2
DR
寄存器堆
LD.REG
SR2OUT
SR1OUT
32
32
[25:0]
SEXT
32
EXT.S
LD.B
B
4
A
LD.A
PC
ALUOp
4
B.S
2
BMUX
AMUX
A.S
[5:0]
ALU控制器
4
CLK
[31:26]
有限
状态机
R
Z
控制
ALU
32
Z
LD.IR
IR
32
ALUOut
LD.ALU
GateALU
GateMDR
32
32
LD.MDR
MDR
MAR
LD.MAR
R
存储器
32
MEM.EN.R.W
输入
输出

4、访问内存

- 获取内存中的数据

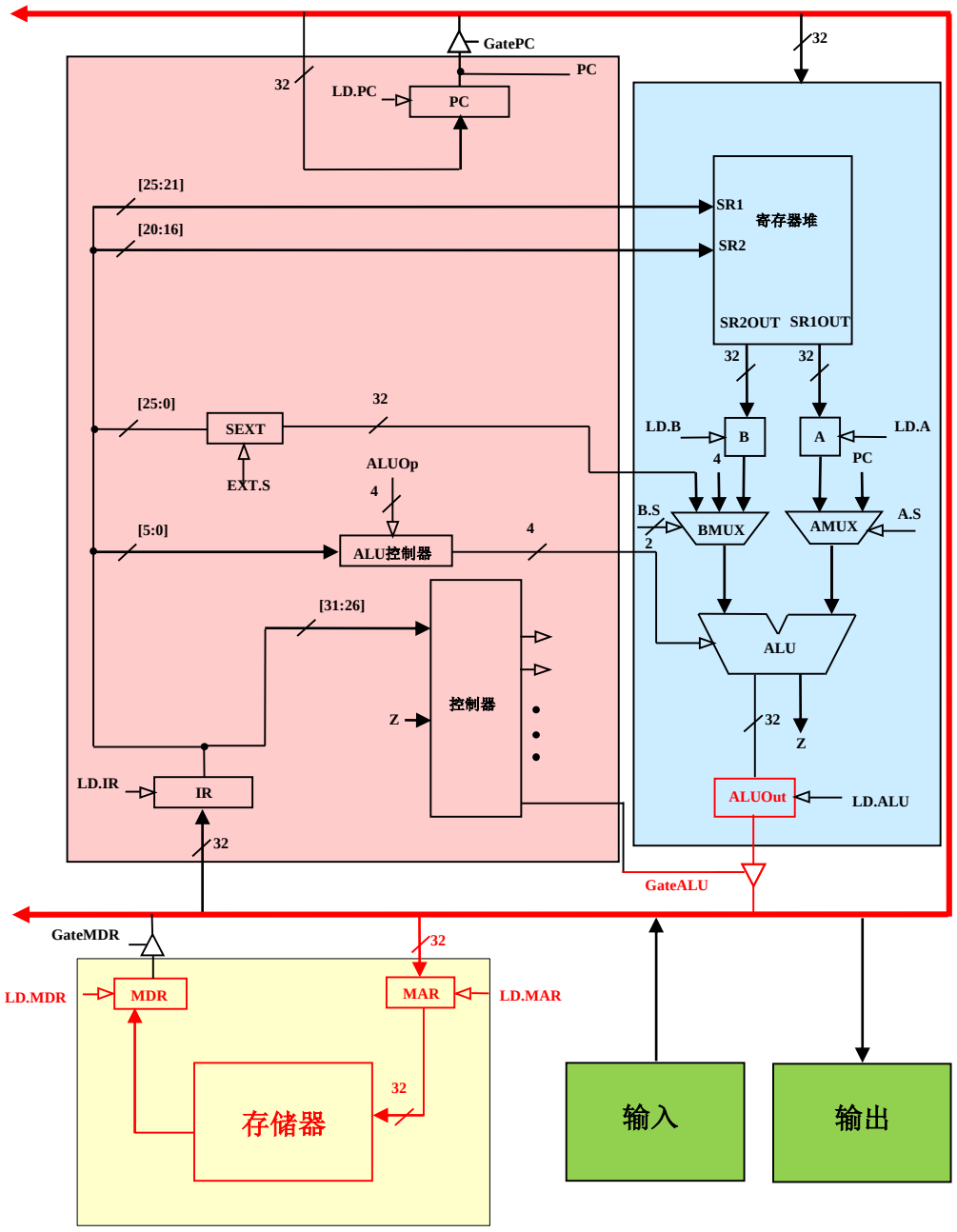


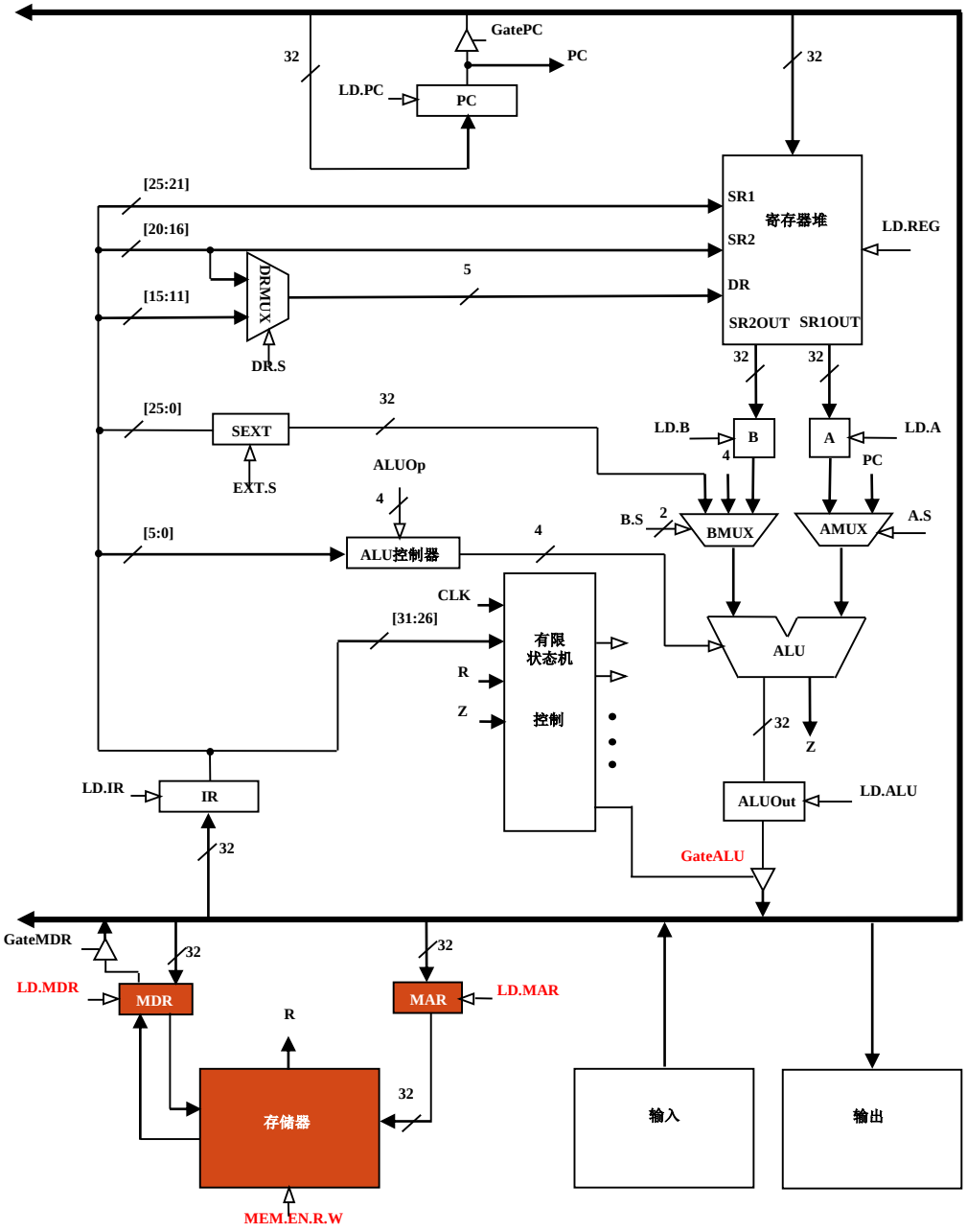
4、访问内存

- 下一周期（或多于一个，如果访问存储器需要多于一个时钟周期的话）
  - 获取内存中的数据
  - 有限状态机将GateALU和LD.MAR设为1，将ALUOut中的值通过总线传给MAR；
  - 将MEM.EN.R.W设为0（即读存储器模式），LD.MDR设为1，读取存储器的数据，将以该地址开头连续4个单元中的内容加载进MDR。

LW指令

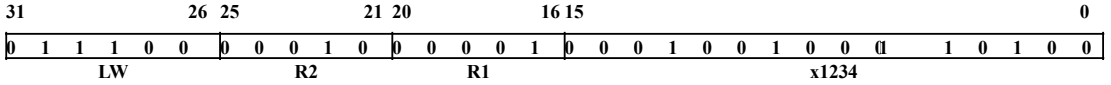
- 访问内存
- LW指令
- MAR ← ALUOut
  - GateALU=1
  - LD.MAR=1
- MDR←Mem[MAR]
  - MEM.EN.R.W=0
  - LD.MDR=1



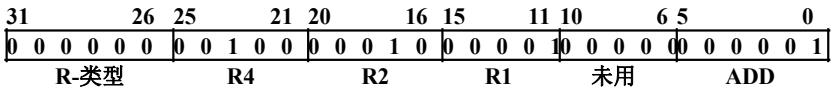

GatePC
PC
32
LD.PC
PC
32
[25:21]
[20:16]
[15:11]
DRMUX
DR.S
5
SR1
SR2
DR
寄存器堆
SR2OUT
SR1OUT
LD.REG
32
32
[25:0]
SEXT
EXT.S
32
LD.B
B
4
A
LD.A
PC
ALUOp
4
[5:0]
ALU控制器
4
B.S
2
BMUX
AMUX
A.S
CLK
[31:26]
R
Z
有限
状态机
控制
ALU
32
Z
LD.IR
IR
32
ALUOut
LD.ALU
GateALU
GateMDR
32
32
LD.MDR
MDR
MAR
LD.MAR
R
存储器
32
MEM.EN.R.W
输入
输出

5、存储结果

- **最后一个周期**
  - 结果被写到指定的目标中
  - 有限状态机将DR.S设为1，选择IR的目标寄存器（DR），即被加载的寄存器；
  - LW指令：



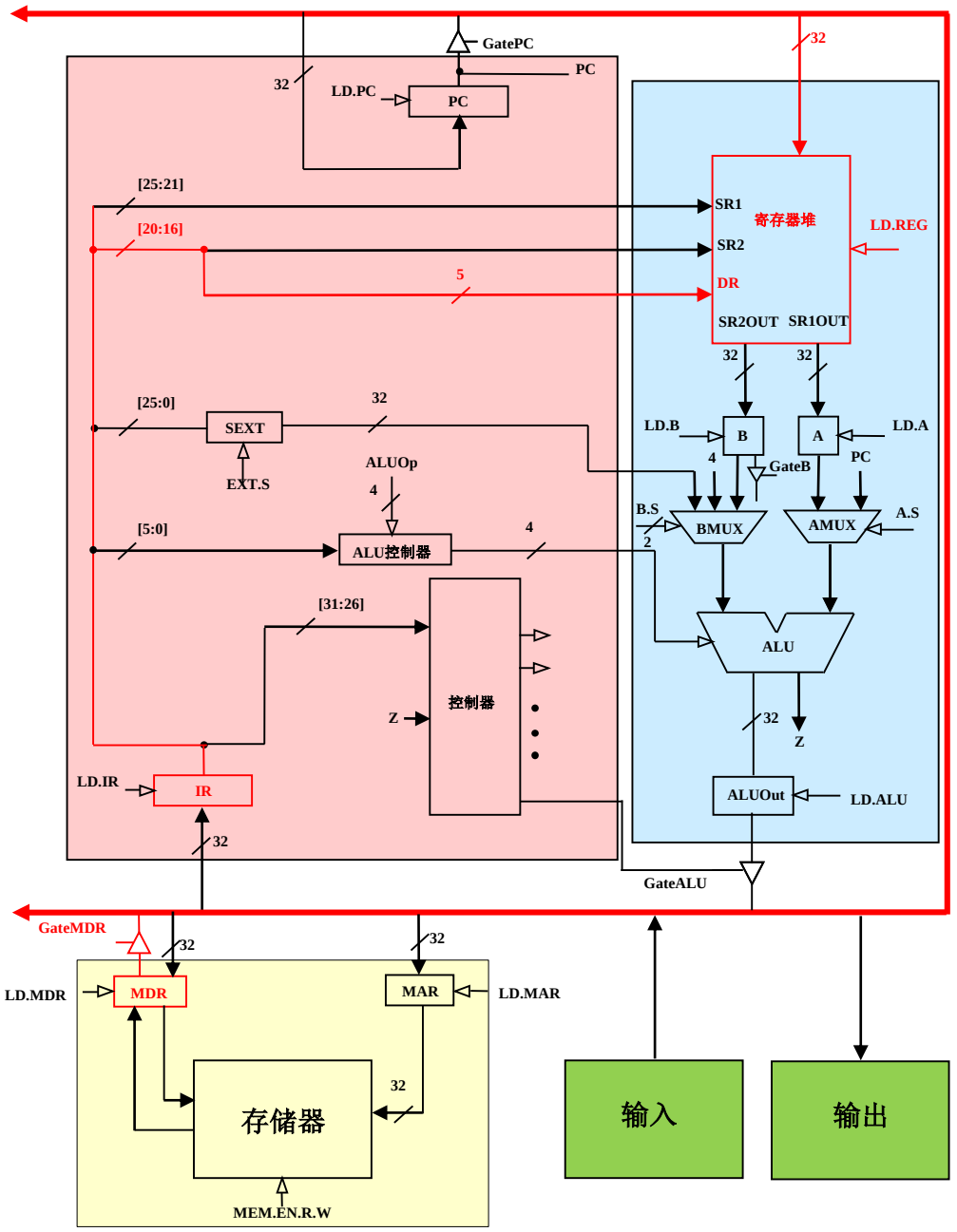
    - 将GateMDR和LD.REG设为1，在时钟周期结束时，MDR中的值被加载到IR[20:16]的DR（R1）中
  - ADD指令

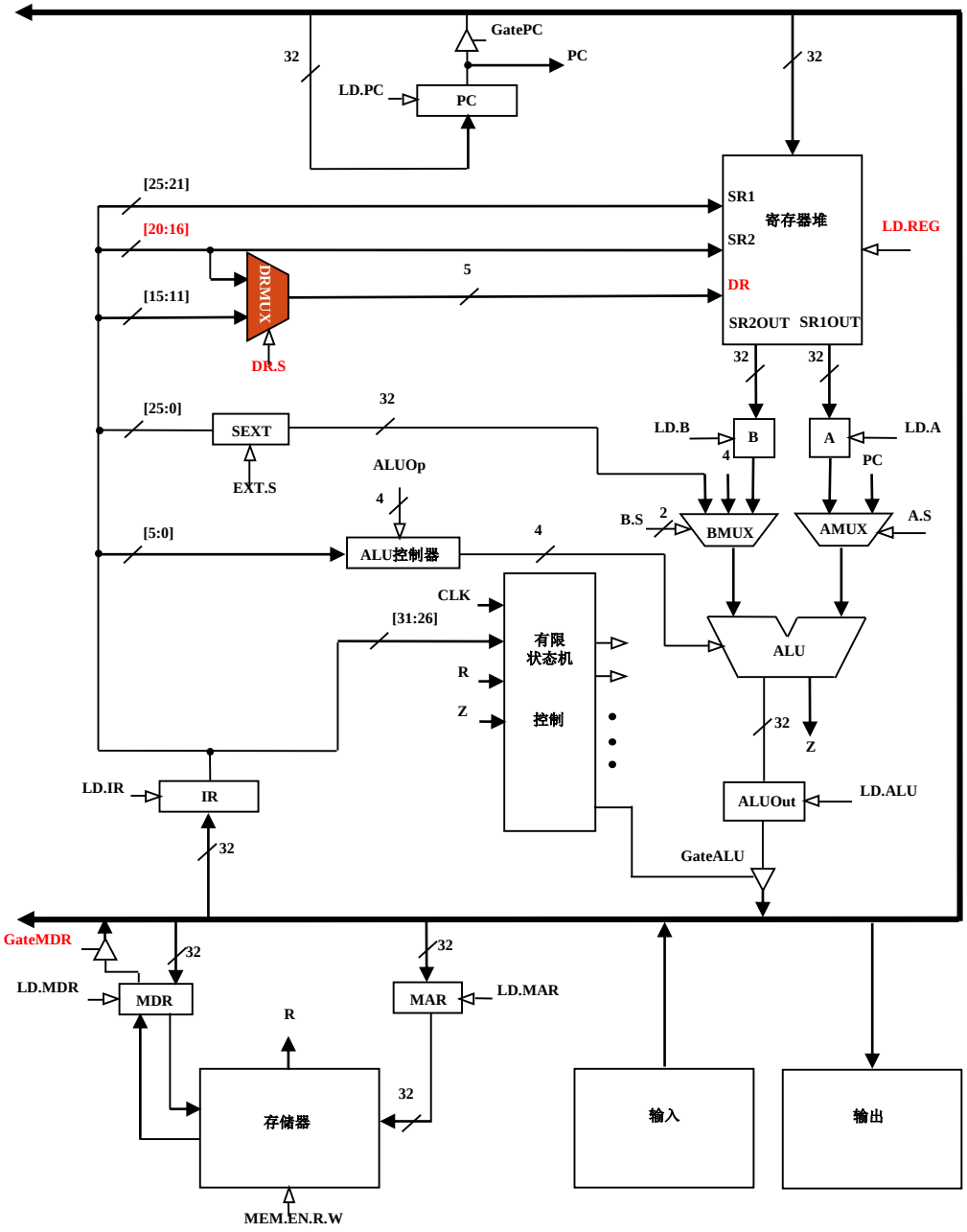


    - 将GateALU和LD.REG设为1，加法运算的ALUOut寄存器中的结果被写入IR[15:11]所指示的R1中

LW指令

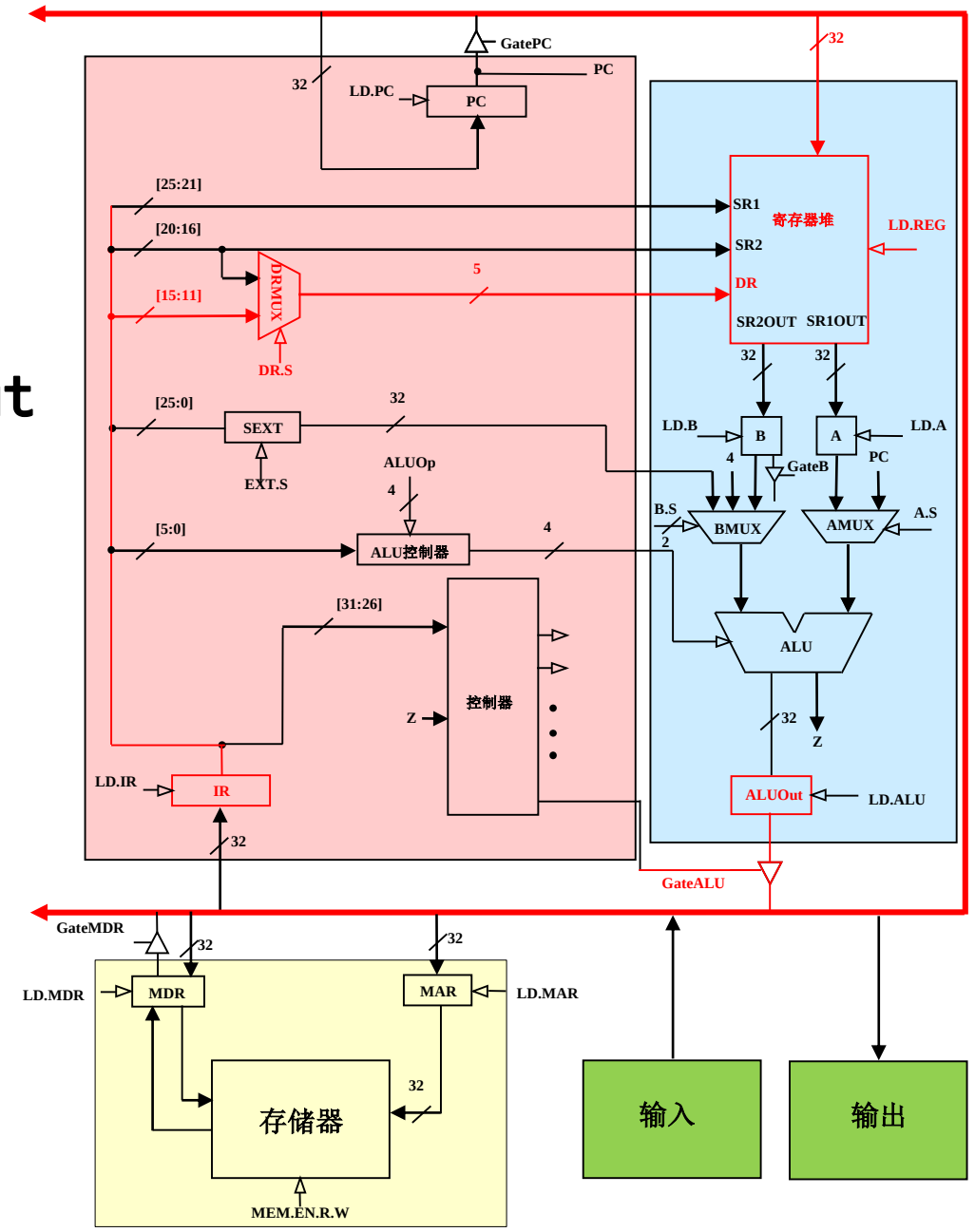
- (IR[20:16])←MDR
  - 选择DR
    - DR.S=1（参考下页数据通路图）
  - MDR数据加载到DR
    - GateMDR=1
    - LD.REG=1




GatePC
PC
32
LD.PC
PC
32
[25:21]
[20:16]
[15:11]
DRMUX
5
DR.S
SR1
寄存器堆
SR2
DR
SR2OUT
SR1OUT
LD.REG
32
32
[25:0]
SEXT
32
EXT.S
LD.B
B
A
LD.A
4
PC
ALUOp
4
[5:0]
ALU控制器
4
B.S
2
BMUX
AMUX
A.S
CLK
[31:26]
有限
状态机
R
Z
控制
ALU
32
Z
LD.IR
IR
ALUOut
LD.ALU
32
GateALU
GateMDR
32
32
LD.MDR
MDR
MAR
LD.MAR
R
32
存储器
输入
输出
MEM.EN.R.W

ADD指令

- DR(IR[15:11])←ALUOut
  - 选择DR
    - DR. S=1
  - MDR数据加载到DR
    - GateALU=1
    - LD. REG=1



下一阶段

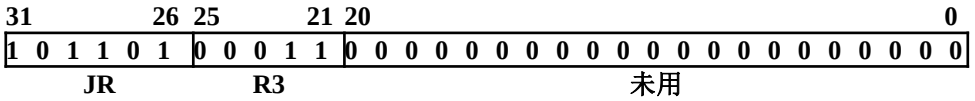
- 这五个阶段完成之后，控制单元就会从取指令阶段开始执行下一条指令
- 由于在取指令阶段PC被更新，包含了存储在存储单元中的下一条指令的地址
- 下一条指令接下来就会被读取，处理就这样持续下去直到被打断
- 不是所有的DLX指令都包括上述五个阶段
- 所有指令均需要取指令阶段和译码/取寄存器阶段
  - ADD指令，不需要访问内存阶段

改变执行顺序

- ADD：处理数据的运算指令
- LW：把数据从一个地方移动到另一个地方的数据传送指令
- BEQZ：改变指令执行的顺序的控制指令
  - 有时会需要先执行第一条指令，接着第二条，第三条，然后又执行第一条，接着第二第三，接着又是第一条……，即循环结构
  - 由于每条指令的执行都是从用PC加载MAR开始的，因此，如果想要改变指令执行的顺序，就需要在PC增加4（即在取指令阶段执行时）后、执行下一指令的取指令阶段之前改变PC（执行阶段改变PC）

DLX JR指令

- bits[31:26]：101101，JR
- bits[25:21]：包含下一条将要被执行的指令地址的寄存器



“把R3的内容加载到PC，这样，下一条将要被执行的指令的地址就是那个包含在R3中的地址”

JR指令

- 从PC=x80008000开始
- 取指令阶段：IR被加载为JR指令，PC更新为地址x80008004
- 译码/取寄存器阶段：译码，并取出R3和R0中的值，计算PC与SEXT(IR[15:0])的和
- 完成分支阶段：PC被加载为x80004000（假设在指令开始处R3的内容为x80004000）
- 指令执行完毕（只需要3个阶段）
- 要处理的下一条指令将位于地址x80004000而不是在地址x80008004

C语言的数据类型与计算机的ISA

C语言的数据类型与计算机的ISA

- 三种基本类型的数值范围
  - int，ISA的字长的二进制补码整数
    - 在DLX上，-2^{31}~$2^{31}-1$
    - 变量声明：为这个变量留出足够的存储空间（DLX：4个存储单元）
  - char，8位
    - DLX，字符型变量只需占用一个存储单元
  - double
    - 通常，一个double是64位长
    - float是依照IEEE754浮点数标准的32位

三种基本类型的变化

- C还为程序员提供了改变三种基本类型表示的数值范围的能力
  - 通过加上修饰符long和short达到扩展或缩短其缺省长度的目的
  - 许多编译器仅当计算机的ISA支持这些长度变化时，才支持修饰符long和short

示例

- 如果系统支持长度变化
  - long int
    - 位数是常规int的2倍的整数
  - long double
    - 范围更广、精度更高的浮点数类型
  - short int
    - 比缺省长度更小的变量
  - unsigned int
    - 无符号整数，非负整数（即正数和零）
    - 当处理那些不需要负数的实际问题
    - 0 — 4294967295($2^{32}-1$)

类型提升

- 混合类型表达式
  - “i + 3.1”
  - “x + 'a'”
  - 转换规则为“类型提升”
- 在C语言中，较短的类型会被转换成较长的类型

书面作业

- 8. 2
- 8. 3
- 8. 4
- 9. 1
- 9. 2
- 9. 3
- 9. 4
- 9. 5
- 9. 9
- 9. 10
- 9. 11