# 第八章 冯·诺依曼（Von Neumann）模型

# 存储程序计算机模型

- 第七章，判定元件和存储元件
- 基于约翰·冯·诺依曼（John Von Neumann）提出的存储程序计算机模型
  ——现代计算机的构建思想

# 存储程序计算机模型

- 1943: ENIAC
  - 十进制，电子管……
  - 硬连线程序 - 设置开关
- 1944: EDVAC开始研制
  - Electronic Discrete Variable Automatic Computer,电子离散变量自动计算机
  - 程序存储于memory之中
- 1945: John von Neumann
  - First Draft of a Report on EDVAC, 关于EDVAC的报告草案

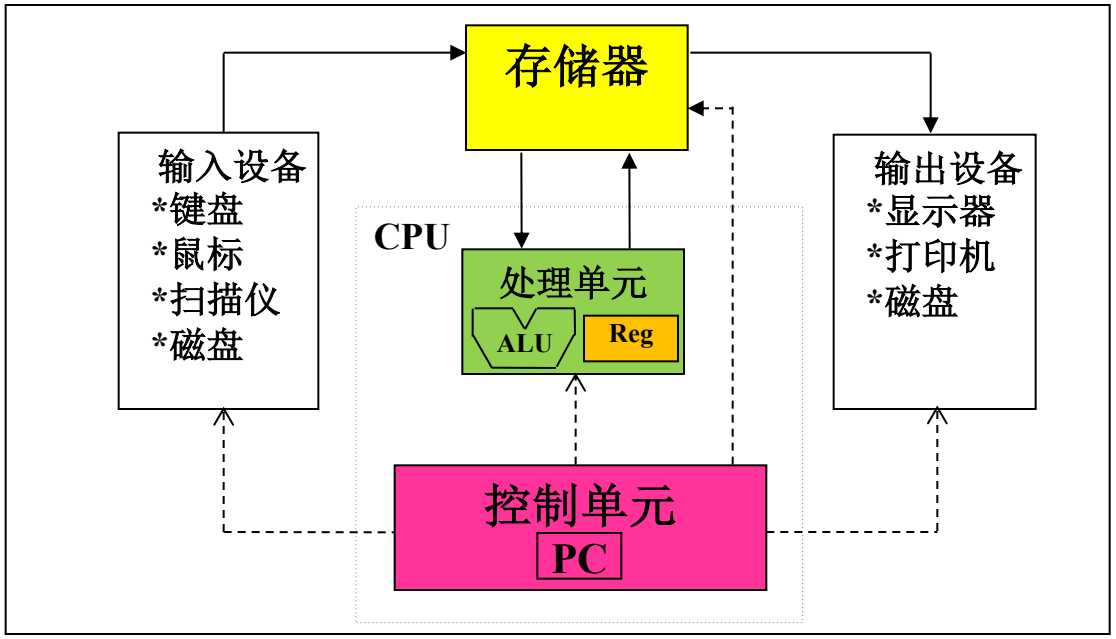
# John von Neumann

- (1903~1957),出生于匈牙利，20世纪最杰出的数学家之一
- 在计算机科学、经济、物理学中的量子力学及几乎所有数学领域都作过重大贡献

# 基本组件

# 冯·诺依曼模型

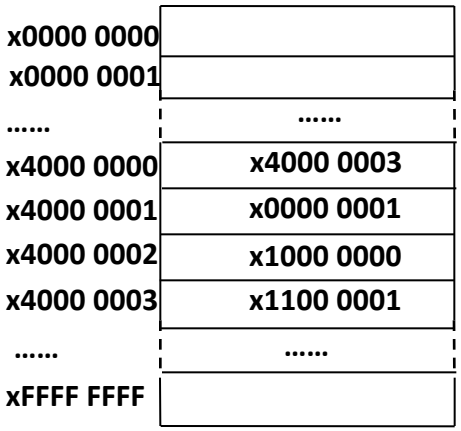
- 由指令组成的程序和程序所需的数据位于存储器中
- 指令的执行由处理单元完成
- 指令执行的顺序由控制单元来控制
- 输入设备将程序和所需的数据送入计算机之中
- 输出设备将执行结果送出计算机之外
- 注意：处理单元和控制单元是CPU的主要组成部分



# 存储器

- 能够存储信息的二维阵列
- 每一行，存储单元（Memory Location）
  - 可使用一个唯一的标识符进行识别，地址
  - 包含一定大小的内容
    - 指令
    - 数据

# 地址和内容


x0000 0000
x0000 0001
……
x4000 0000
x4000 0001
x4000 0002
x4000 0003
……
xFFFF FFFF
……
x4000 0003
x0000 0001
x1000 0000
x1100 0001
……

# 存储器

- $2^{32}$×8位，一共有$2^{32}$个存储单元，每个存储单元可以存储8个比特
- 有$2^{32}$个不同存储单元的地址空间和8位的寻址能力，可称为4G字节（缩写为GB）的存储器
- 要确定$2^{32}$个地址，就必须使用32位二进制数表示

# 处理单元

- 计算机里信息的处理是由处理单元执行的
- 现代计算机的处理单元可以包含许多复杂的功能单元，每个都能够执行一个特定的运算（除法，平方根等）
  - 最简单的单元ALU（Arithmetic and Logic Unit，算术和逻辑单元）

# 寄存器堆/文件

- Reg
- ALU附近，临时存取数据
  - 例如，计算(A+B)×C，先在存储器中存储A+B的结果，随后读取出来，再和C相乘
    - 访问存储器的时间远长于执行加法或乘法的时间
    - 使用临时存储空间存储A+B的结果

# 字

- ALU正常处理的信息量的大小通常被称为计算机的字长(word length)，每一个元素被称为一个字（word）
- 取决于计算机的不同用途，每一个指令集结构都拥有自己的字长
  - Intel的Pentium Ⅳ处理器，32位
  - Sun的SPARC-V9和Intel的Itanium处理器，64位

# 寄存器堆/文件

- 典型的寄存器的大小是和ALU处理的值的大小一样
  - 每个寄存器都包含一个字

# 控制单元

- 处理单元负责“执行信息的实际处理”，而控制单元则是“指挥信息的处理”
- 其具体工作包括：
  - 在执行程序的过程中，跟踪存储器中的指令
  - 在处理指令的过程中，跟踪指令的处理阶段

# PC

- 跟踪存储器中的指令，确切地说是跟踪要处理的下一条指令
  - 为了跟踪哪一条指令是下一步要运行的，控制单元有一个用来容纳下一条指令所在地址的寄存器——“程序计数器”（Program Counter，简称PC）/ “指令指针”

# 控制器

- 控制单元可以是多个控制器，分别从属于各个部件
  - ALU控制器用于控制ALU执行何种运算
  - 对于输入和输出则有专门的I/O控制器
  - ……

# 输入/输出设备

- 要使计算机处理信息，信息必须被送入计算机中
- 为了能够使用处理后的结果，它必须能以某种形式显示在计算机以外
- 为输入和输出的目的而出现的设备在计算机术语中被称为外围设备（peripherals）

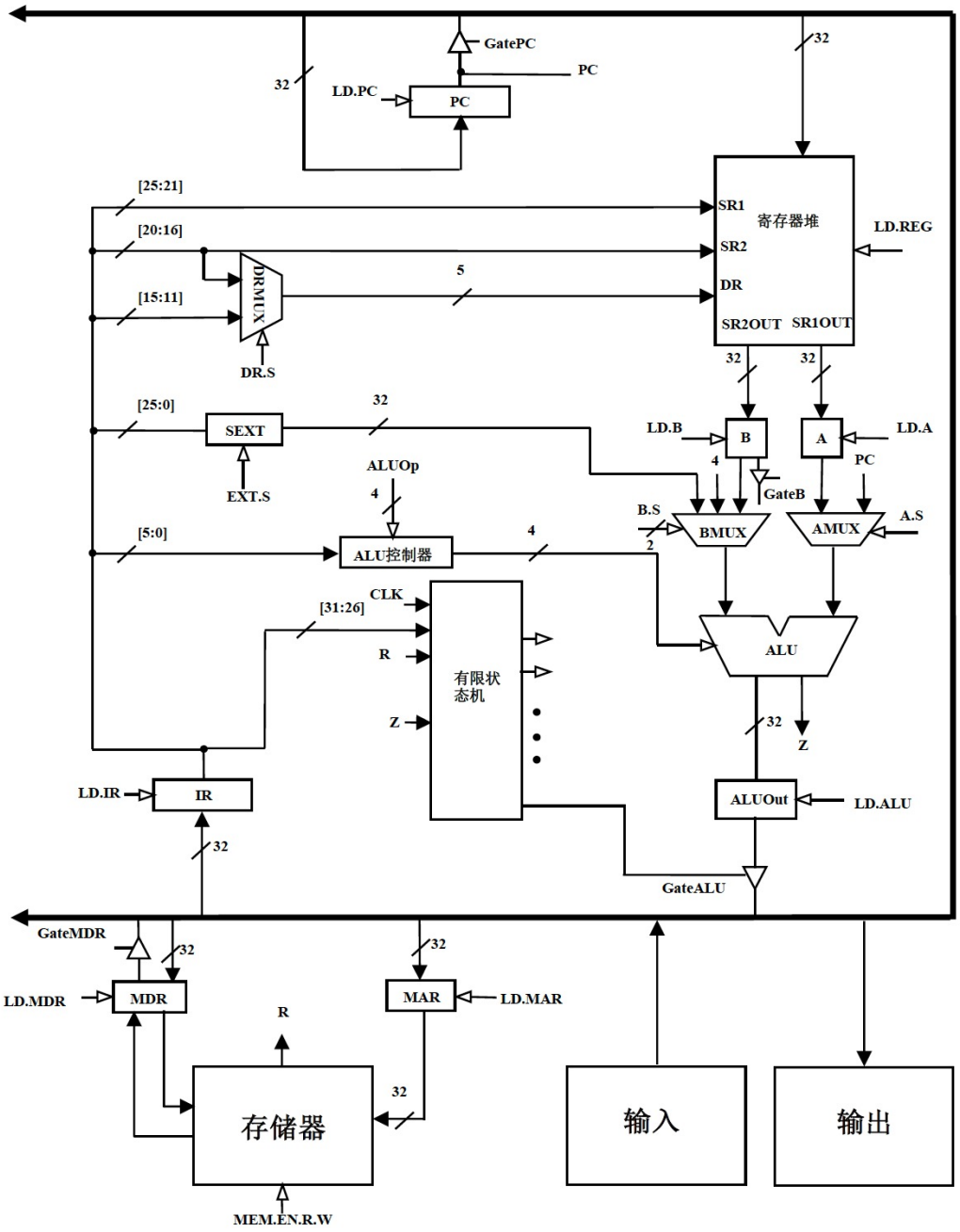
# 输入/输出设备

- 外围设备（peripherals）
  - 键盘——输入；监视器（显示器）——输出
  - 输入：鼠标，数字扫描仪、磁盘
  - 输出：打印机，磁盘

# DLX——冯·诺依曼示例

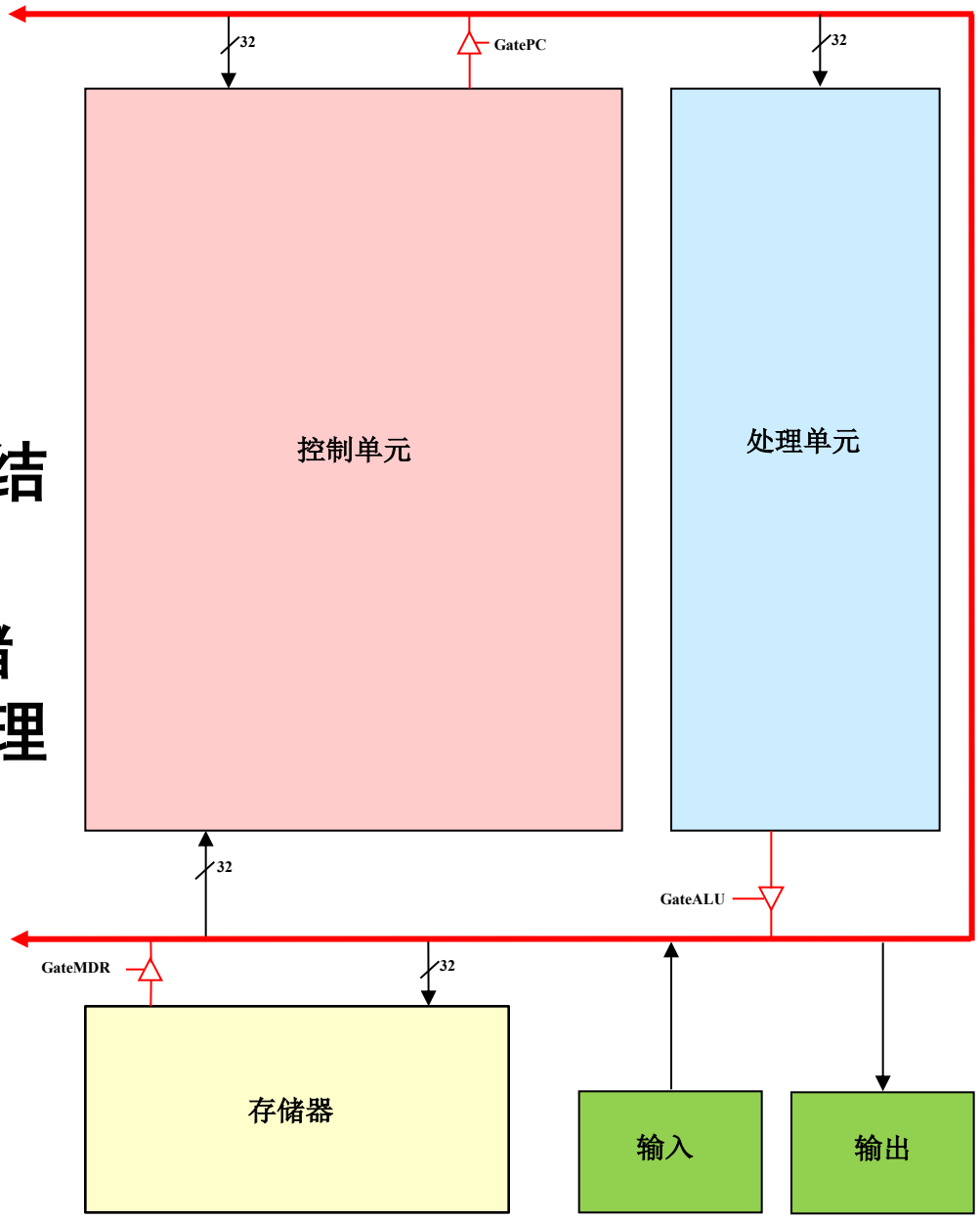
# DLX数据通路

- 一个采用**总线结构、多时钟周期**的实现方案
- “数据通路”，在计算机内部用于处理信息的所有元件的总和



# DLX总线

- 两端都有箭头的粗黑线结构代表数据通路的**总线**
- 组件之间的通信，如存储器与处理器之间、或处理器与I/O设备之间
- 优点：功能多、成本低
- 缺点：通信瓶颈



# DLX总线

- DLX的总线由32根线和相关的电子元件组成
  - 允许将32位信息从一个组件传输到另一个组件
- 在总线上一次只可传输一个值
  - 每一个提供数据给总线的组件在它的输入箭头后都有一个三角形（称为三态设备），使计算机的控制逻辑一次只允许一个提供者能提供信息给总线
  - 从总线获得数据的组件通过将LD.x（加载使能）信号设为1（回忆门控锁存器），从而得到信息

# 存储器：MAR和MDR

- 如果要读出某个存储单元中的内容，首先把它的地址存入地址寄存器（MAR），然后查询存储器，该地址所对应的存储单元的内容将会输出到数据寄存器（MDR）。
- 如果要写一个值到存储单元中，首先要把目的地址存入MAR，把值存入MDR中，然后设“写使能”信号为1，查询存储器，MDR里的信息就会被写到MAR中的地址所对应的存储单元里。

# MAR和MDR

- MAR：32位，反映了DLX的存储器的地址空间是$2^{32}$个存储单元
- MDR：32位，而DLX是字节可寻址的，即每个单元包含8位，因此在大多数情况下，MDR包含了从MAR中的地址开始的4个连续单元的数据，有时，则包含的是MAR所指的单元中的数据（8位）符号扩展的结果（32位）

# 处理单元

- ALU和寄存器堆
- ALU可以做加法、减法、乘法、除法、与、或、异或、比较、移位等运算
- 32个整数寄存器、32个浮点寄存器
- DLX子集
  - 未包括整数乘法、除法及浮点数运算等操作，也未包括浮点寄存器

# 控制单元

- 所有用来管理计算机信息处理的组件
- 最重要的组件是有限状态机，它指挥所有行为
  - 有限状态机的一个输入是CLK，它说明了每个时钟周期持续的时间
  - 为了跟踪指令的处理阶段，控制单元还需要一个指令寄存器（Instruction Register，简称IR），用来保存正在处理的指令。IR也是有限状态机的一个输入，因为要处理的DLX指令决定了计算机要执行的行为。
- 程序计数器（PC）
  - 记录了在当前的指令完成后，下一条要被执行的指令所在的地址

# 空心箭头

- 实心箭头表示沿着相应的通路流动的是数据元素
- 空心箭头表示控制数据元素处理的控制信号
- 有限状态机的所有输出都是空心箭头
  - 控制了计算机的处理
  - LD.IR（1位），控制了当前时钟周期内，指令寄存器（IR）是否要从总线上加载新的指令
  - GateALU，决定ALUOut的值在当前时钟周期内是否被提供给总线

# 输入/输出设备

- 由键盘和显示器组成
  - 最简单的键盘需要两个寄存器，一个数据寄存器（KBDR），用来保存由键盘键入字符的ASCII码，和一个状态寄存器（KBSR），用来提供键盘键入字符的状态信息
  - 最简单的显示器同样需要两个寄存器，一个用来保存那些将被显示在显示器上的内容的ASCII码（DDR），另一个用来提供相关的状态信息（DSR）
- 第12章

# 指令处理

# 指令处理

- 冯·诺伊曼模型的主要思想
  - 把程序和数据都作为一个二进制序列存储在计算机的存储器里，在控制单元的引导下一次执行一条指令
- 指令在控制单元的指挥下以一种系统的方式被逐步处理，根据指令处理所需进行的操作，可以将一条指令的执行分解为一系列步骤

# 指令处理

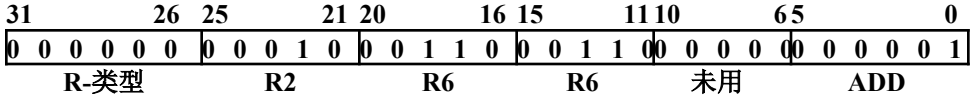
- 多时钟周期的实现方案
  - 指令的每一步将占用一个时钟周期，不同的指令可能被分解为不同的步骤，占用不同的时钟周期，“多周期”因此得名
- 在现代计算机中，时钟周期以纳秒（或称毫微秒，十亿分之一秒）为单位
  - 比如，一个3.3GHz的处理器在1秒内有33亿个时钟周期，即一个时钟周期只需0.303纳秒

# 指令

- 计算机处理的最基本单位
- 计算机程序包含了一组指令，每条指令都是由一个位序列表示，并且整个程序被存储在计算机的存储器中
- 由两个部分组成：操作码（指令执行的内容）和操作数（要操作的对象）
- DLX子集指令，由32 位（一个字）组成，从左向右依次编号为bit[31]…bit[0]

# DLX ADD指令

- 需要三个操作数：两个源操作数（待加的数据）和一个目标操作数（在加法执行后要存储的和）
- 格式：

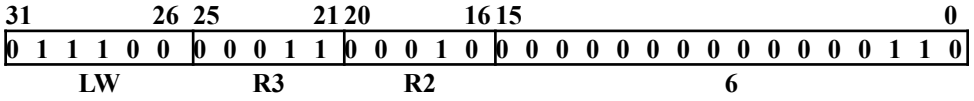



  - 两个源操作数：包含在寄存器中的值
  - 加法的结果：放入32个寄存器的某一个
  - 32个寄存器：需要5位确定每一个
  - 教材订正：R6错了

- bit[31:26]：000000——R类型
- bit[5:0]：要执行的函数，000001，是加法操作
- bit[15:11]：存储的结果所在的位置，R6
- bit[25:21]和bit[20:16]：存储两个源操作数的寄存器，R2和R6
- “将R2（寄存器2）和R6里的内容相加，结果存回R6里。”

# DLX LW指令

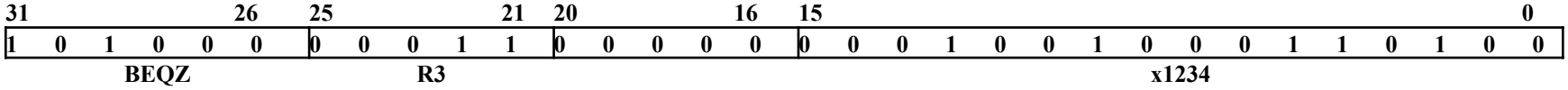
- L代表加载（Load，读存储器中的值，写到寄存器中），W代表字
- “进入指定的存储单元，从该单元开始，读取连续4个存储单元里面包含的值，结果存入一个寄存器里”
- 需要两个操作数：
  - 从存储器读出的值和目标寄存器



- bit[31:26]：011100，是I类型的LW指令
- bit[20:16]：指令执行结束后从存储器里读出的值将存入的寄存器
- bit[25:21]和bit[15:0]：用来计算出要读取位置的地址
  - 将bit[15:0]里的二进制补码整数符号扩展为32位，再与bit[25:21]表示的寄存器里的数值相加，结果即为地址
  - “基址+偏移量”寻址模式
- “将R3里的内容同数值6相加，构成一个存储单元的地址，从该单元开始，读取连续4个存储单元里面包含的值，并加载到R2里。”
- 寻址模式：计算将要读取的存储单元的地址的机制

# DLX BEQZ指令（第九章）

- B代表分支（Branch），EQ代表“相等”，Z代表“零”
- “判断某个寄存器的值是否为零，如果等于零，PC就被某条指令的地址加载”
- 需要两个操作数：
  - 寄存器，某条指令的地址



# BEQZ

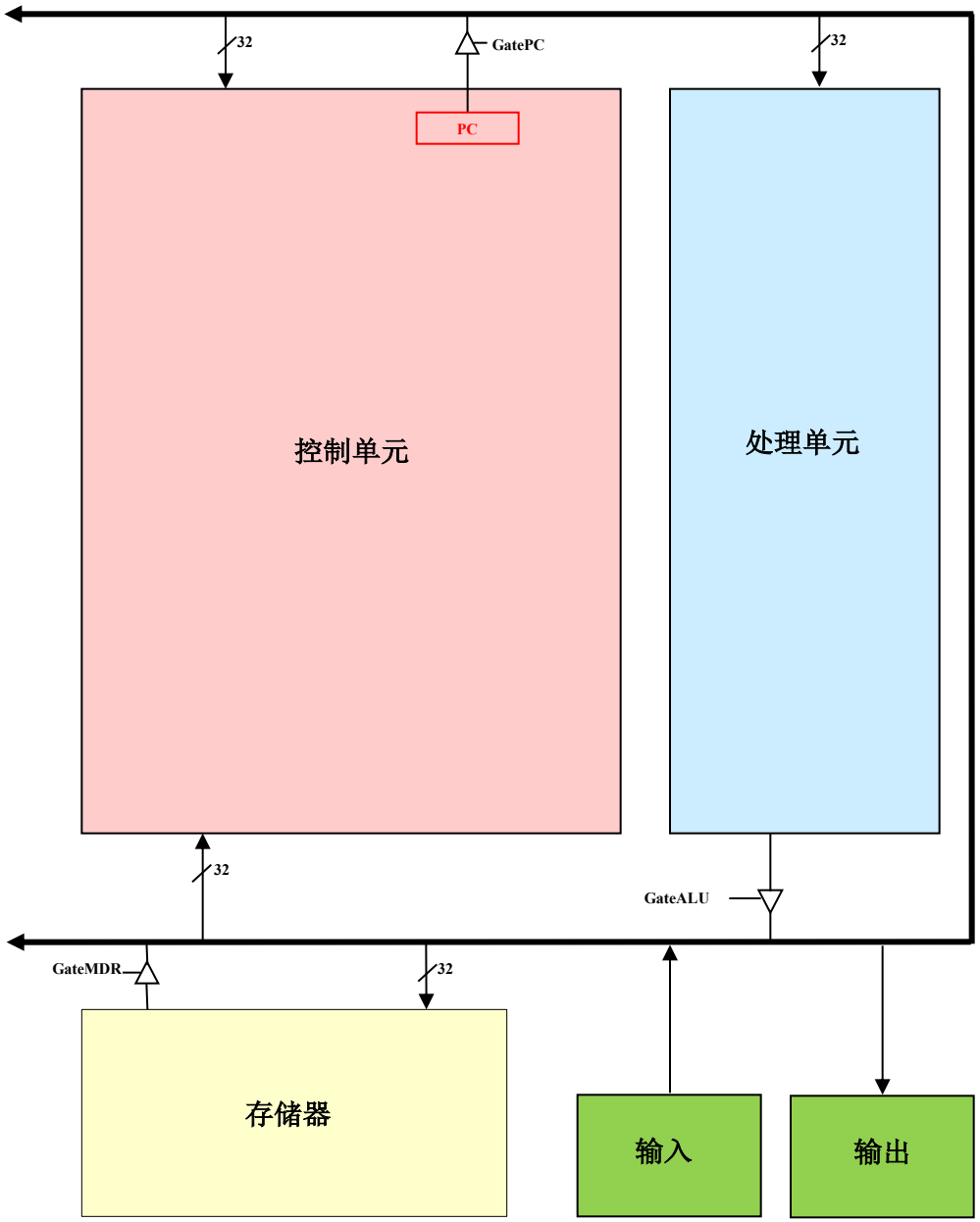
- bit[31:26]：101000，是I类型的BEQZ指令
- bit[25:21]：存储源操作数的寄存器
- bit[15:0]：用来计算出指令地址的值
  - 将bit[15:0]里的二进制补码整数符号扩展为32位，再与PC里的数值相加，结果即为地址
    - PC+SEXT(IR[15:0])
- “判断R3里的值是否为0？如果R3的值为0，PC就被计算得到的地址加载；如果R3的值不为0，PC保持不变”

# 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（每条DLX指令需要其中的3到5个阶段）：
  - 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）
  - 写回（Write-back）

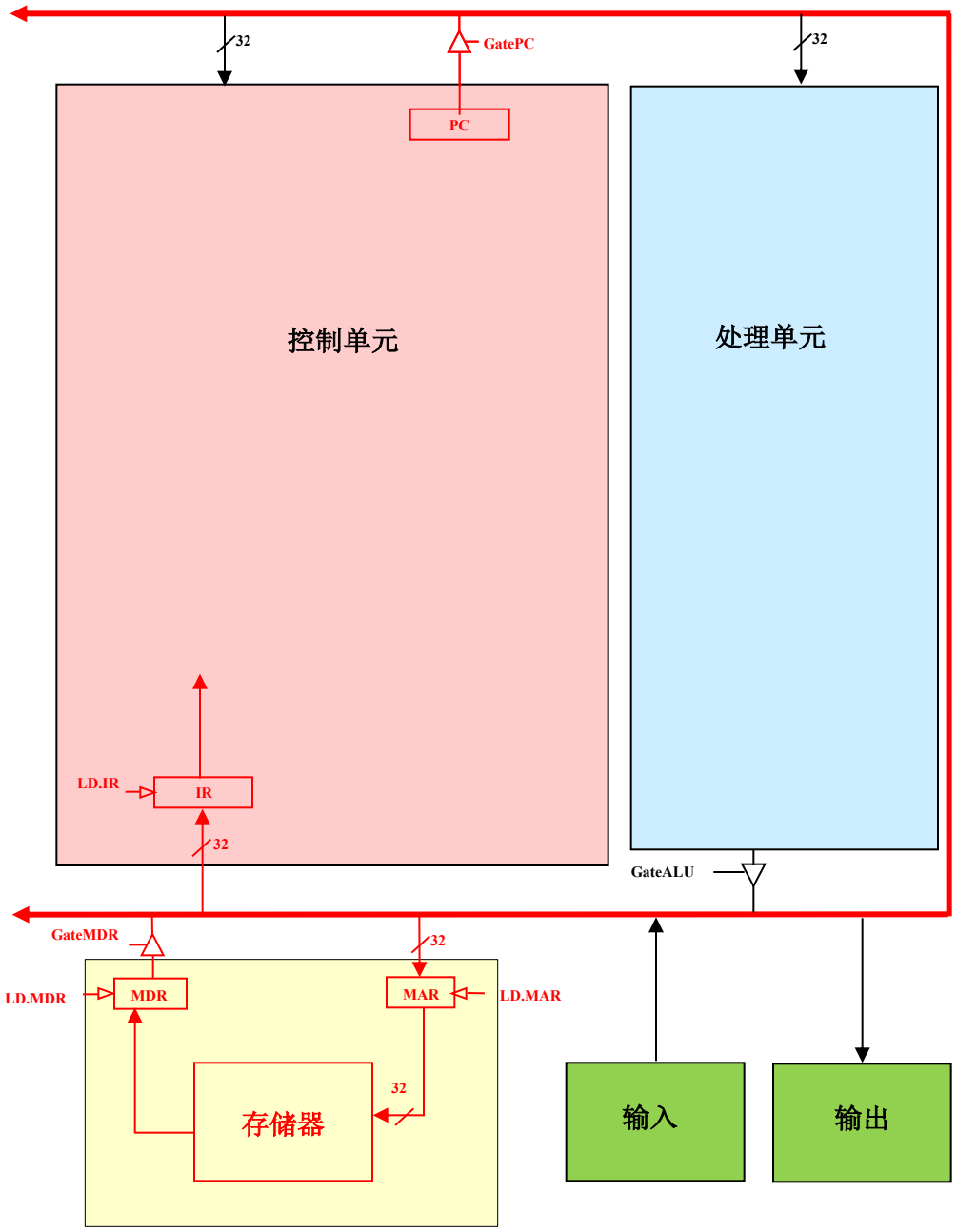
# 1、取指令阶段

- 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）



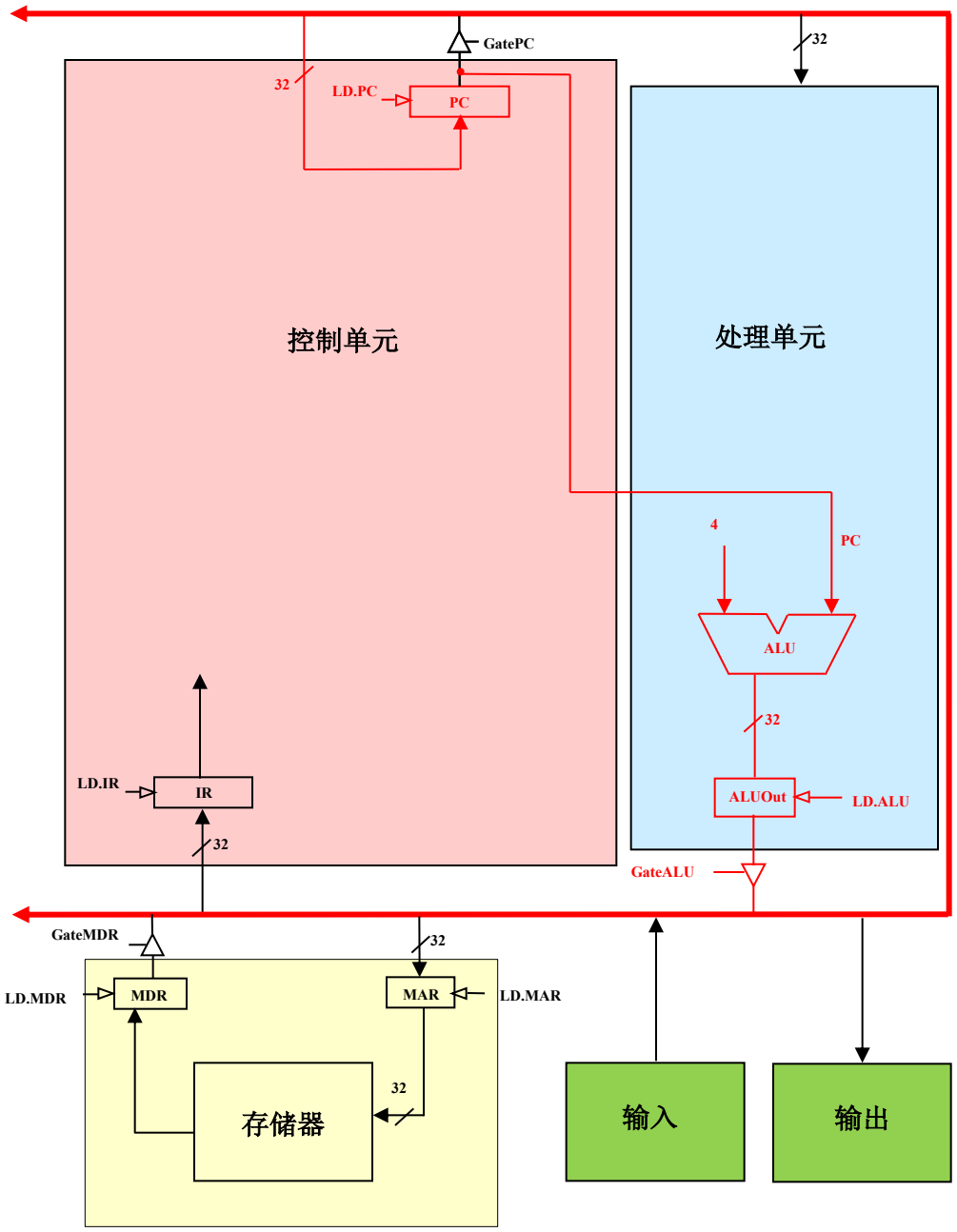
# 取指令阶段步骤(1)

- MAR←PC
- MDR←Mem[MAR]
- IR←MDR



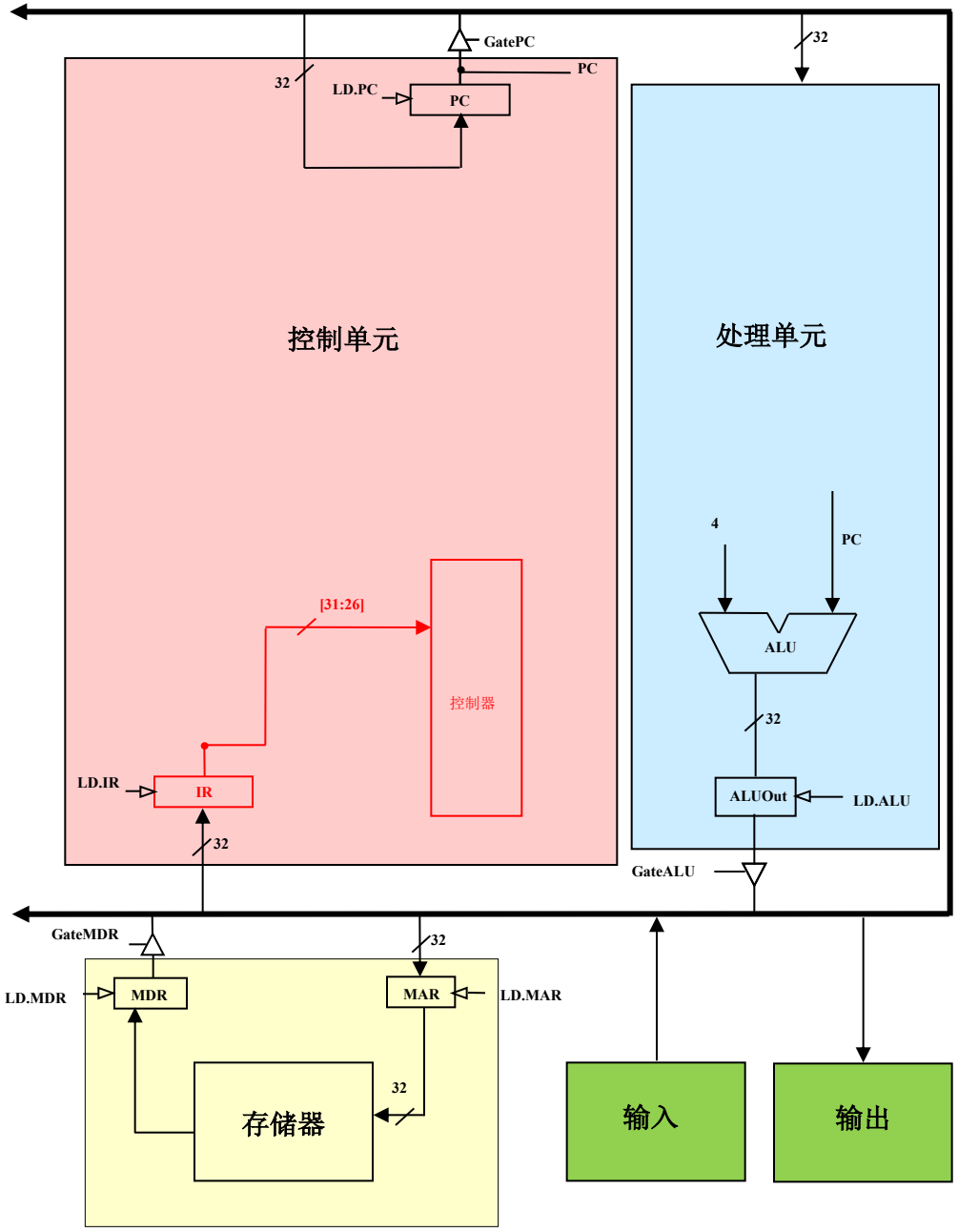
# 取指令阶段步骤(2)

- MAR←PC
  - ALUOut←PC+4
- MDR←Mem[MAR]
  - PC←ALUOut
- IR←MDR



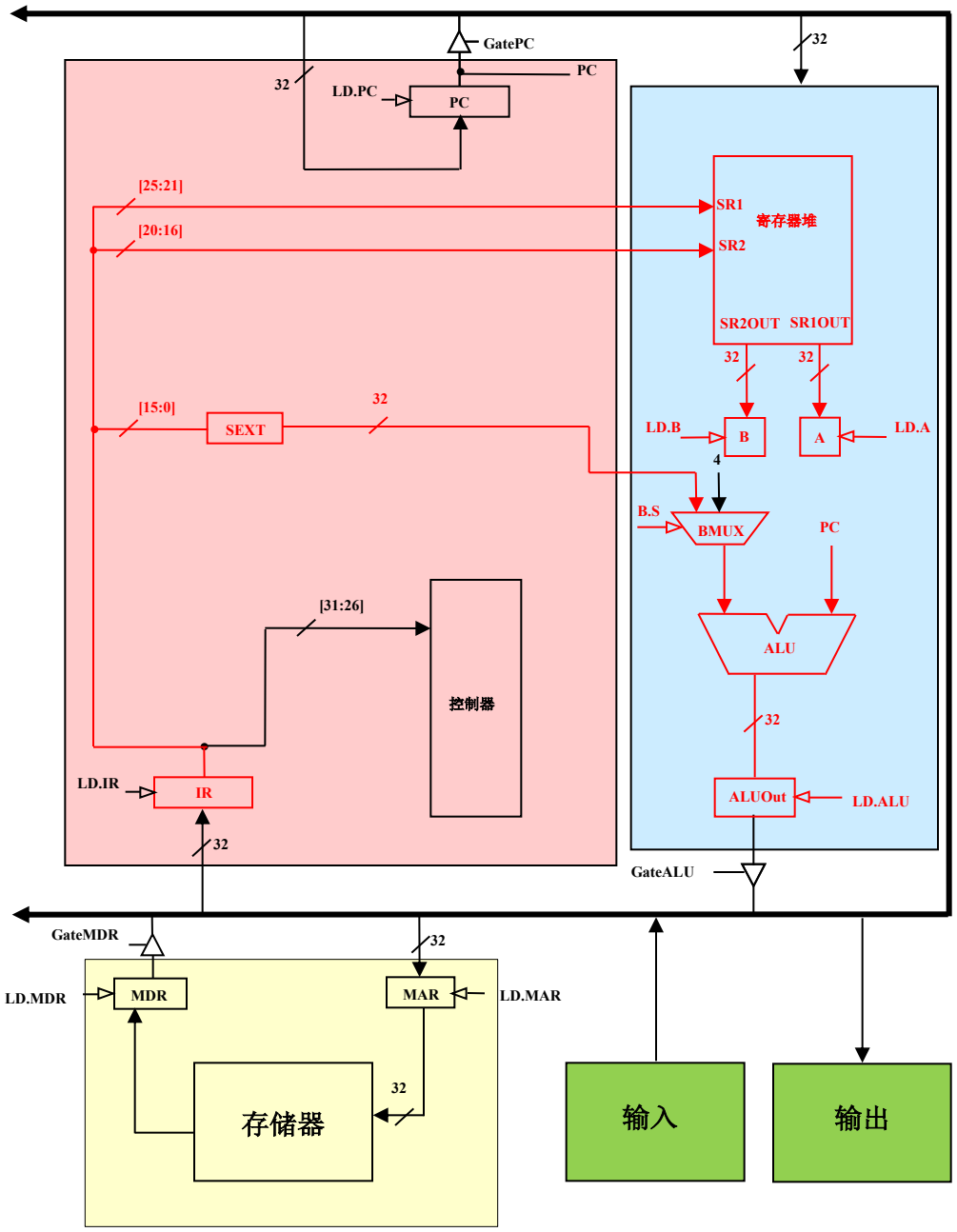
# 2、译码/取寄存器阶段(1)

- 译码：
  - 识别指令进而确定下一步要去做什么
  - 6-64 译码器：IR[31:26]
    - 依据译码器的输出为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
  - 为后面阶段获取操作数
  - A←(IR[25:21]), B←(IR[20:16]), ALUOut←PC+SEXT(IR[15:0])

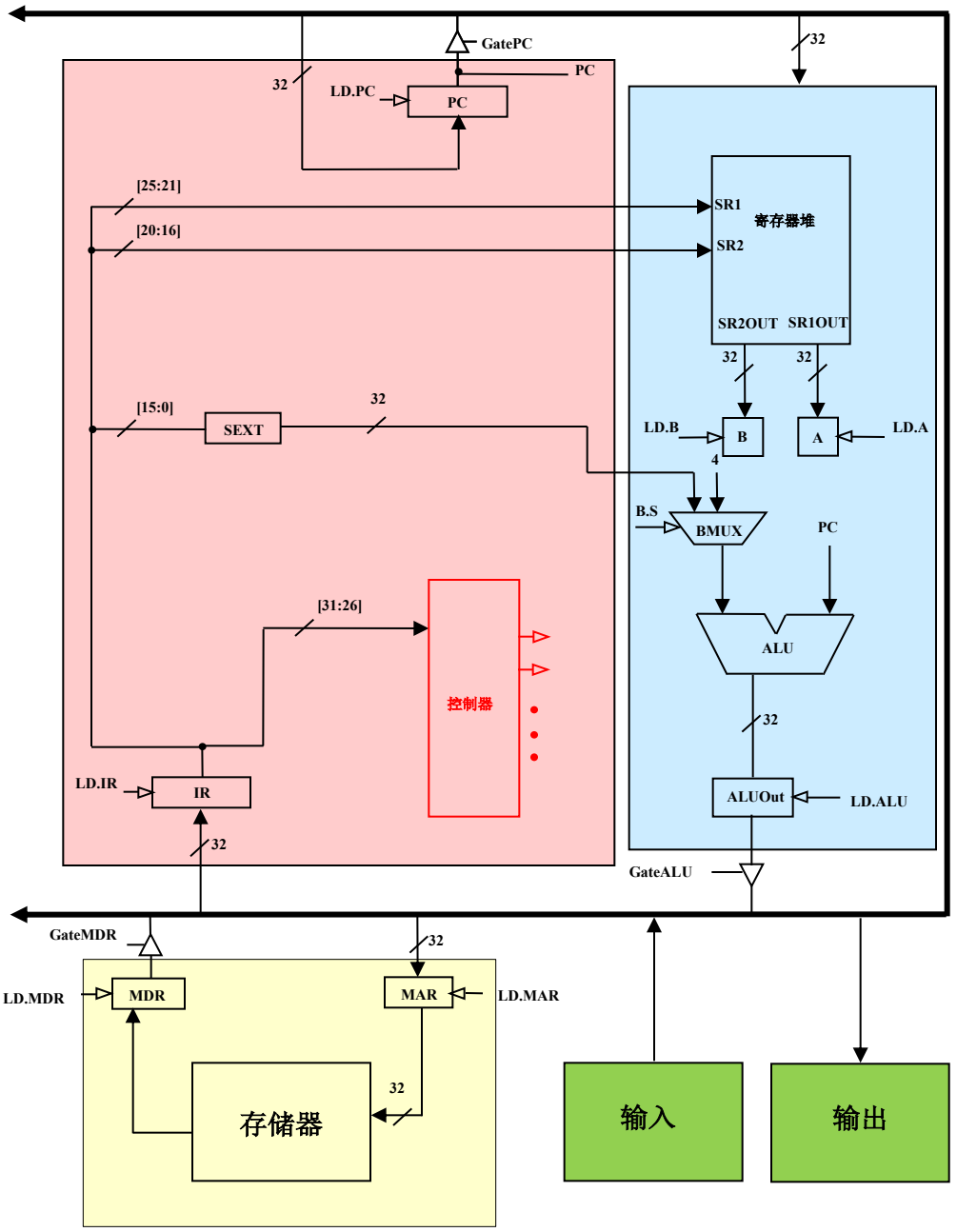


# 取寄存器—示例

- ADD指令
  - 将从IR[25:21]和IR[20:16]所指示的R2和R6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R3和R2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

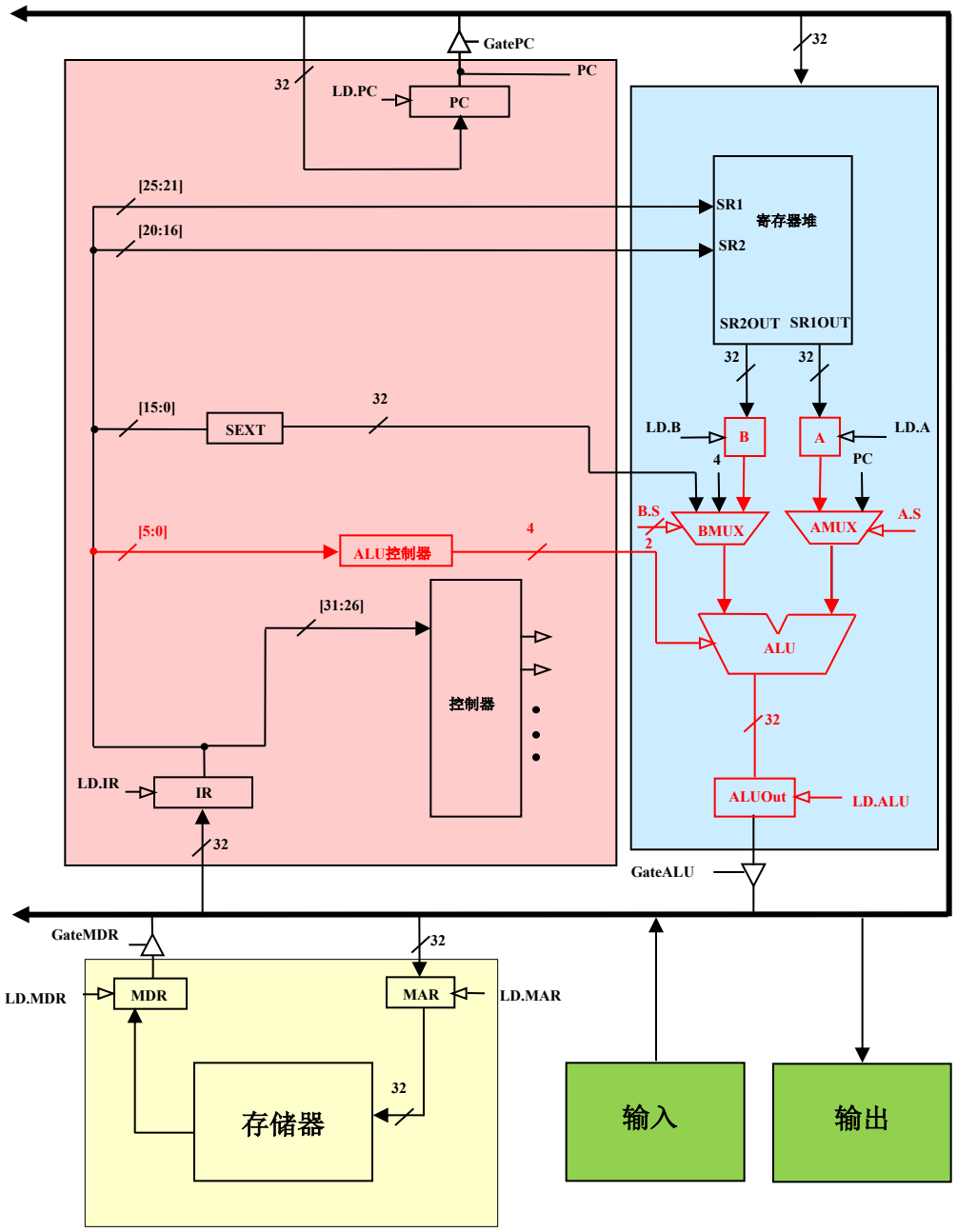
# 3、执行/有效地址/完成分支

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



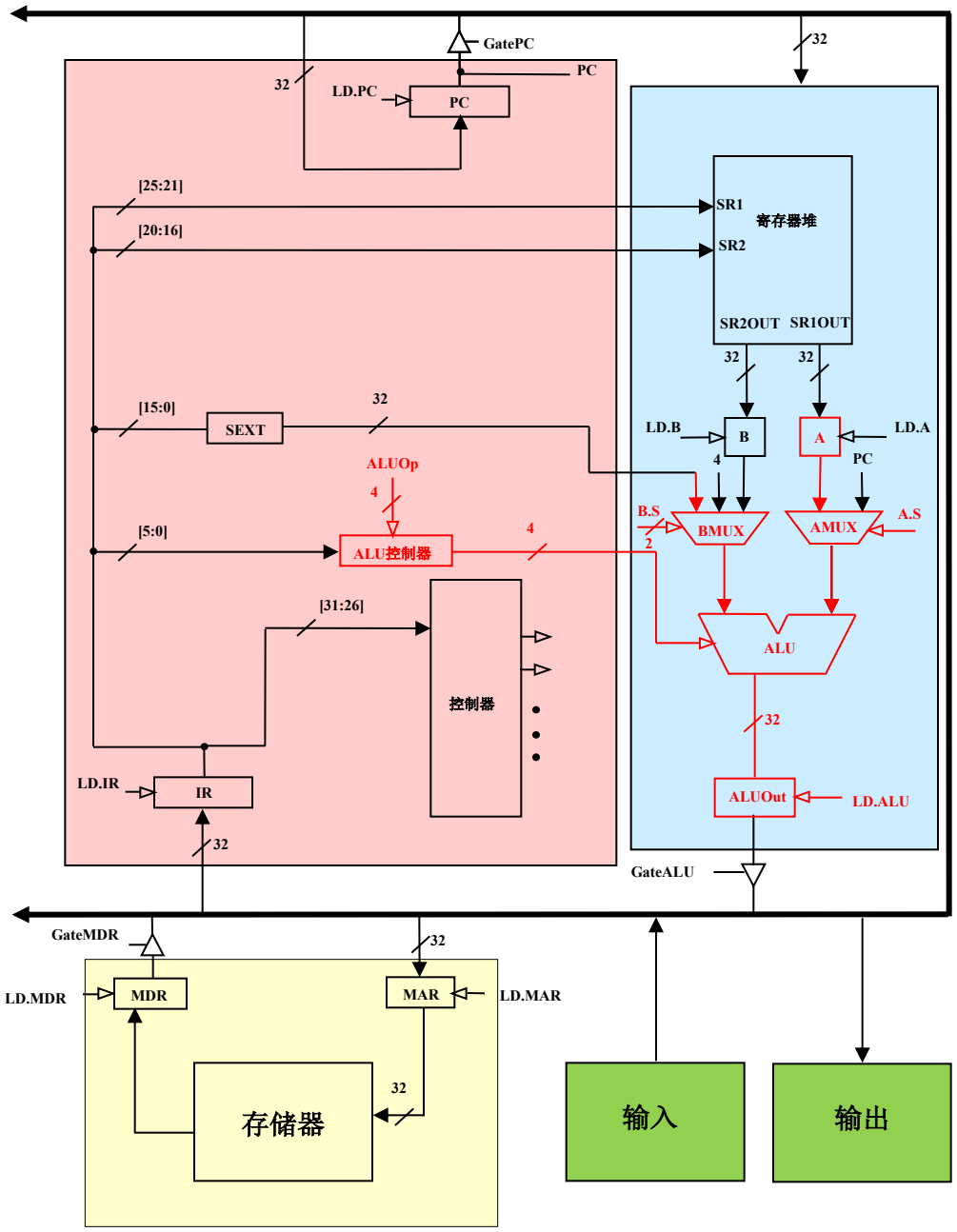
# ADD指令

- ALUOut ← A + B



# LW指令

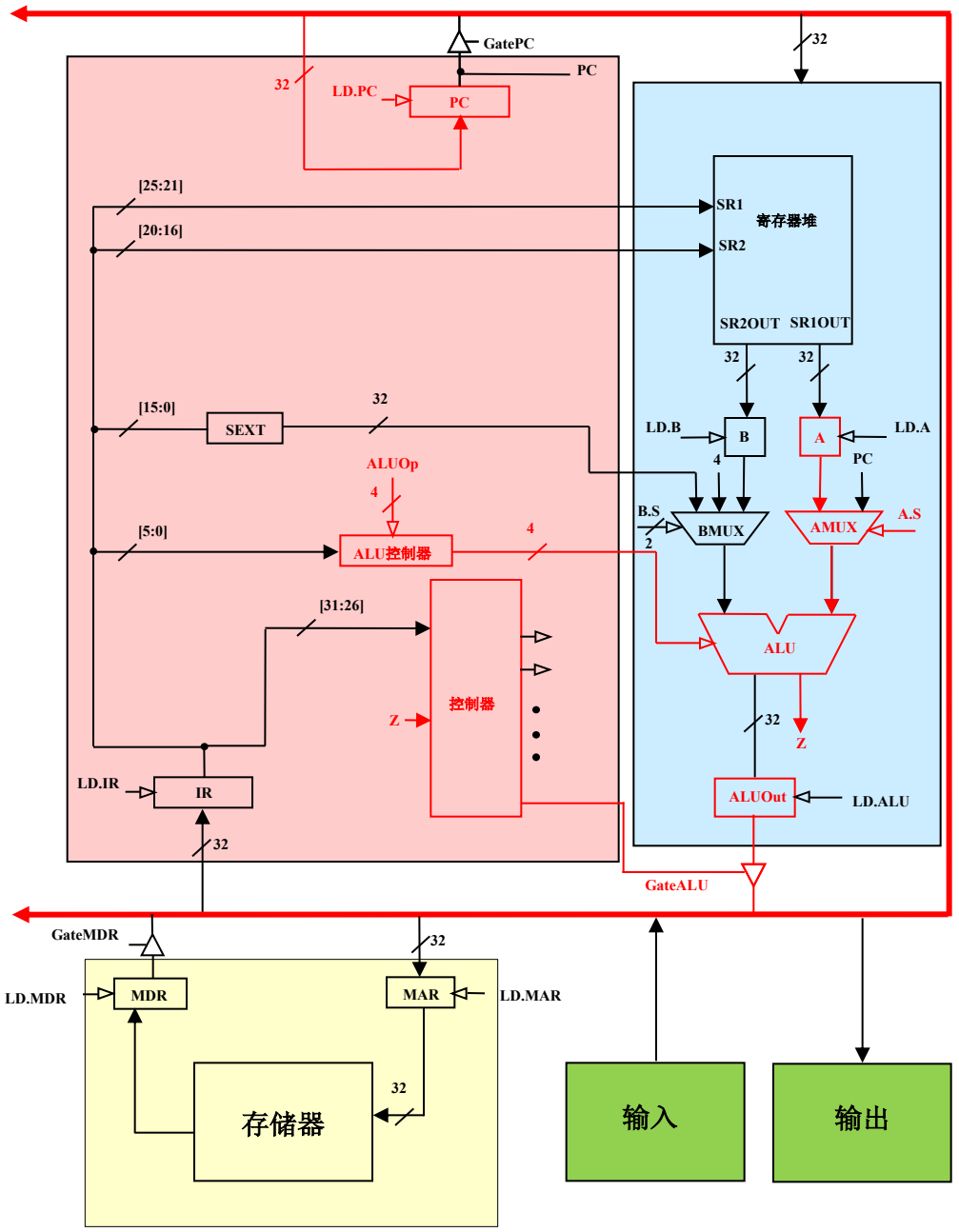
- ALUOut ← A + SEXT(IR[15:0])



# BEQZ指令

- Z ← A == 0
- if(Z)

  PC ← ALUOut



# 执行/有效地址/完成分支

- 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需的存储单元的地址，即有效地址；
- 或者完成分支跳转

# 示例

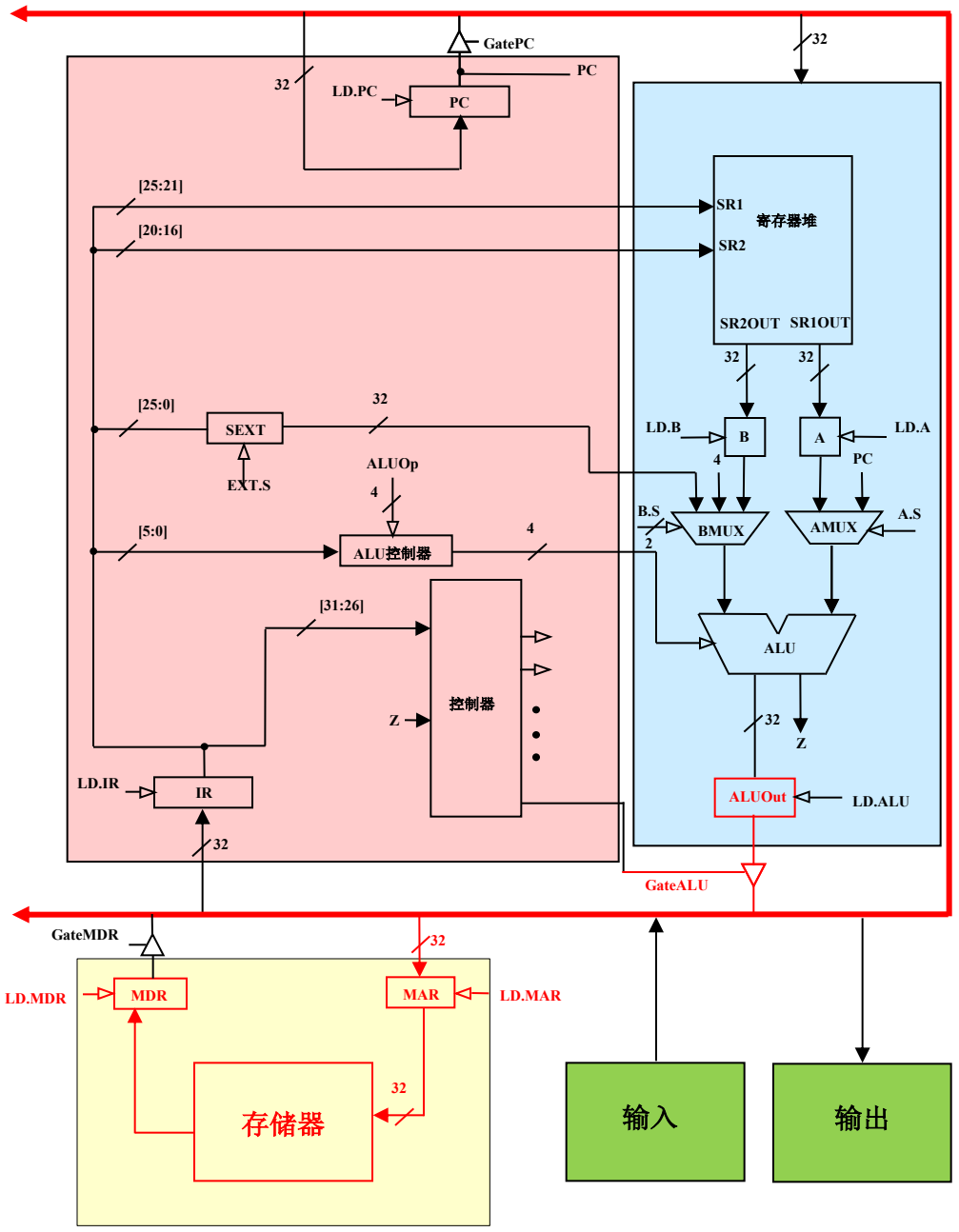
- ADD指令
  - 在ALU中进行加法运算，得到加法运算结果，存储于ALUOut寄存器中
- LW指令
  - 选择IR[15:0]符号扩展的结果，在ALU中与A寄存器进行加法运算，得到一个有效地址，存储于ALUOut寄存器中
- BEQZ指令
  - 如果A寄存器中的值为零，PC被ALUOut寄存器中的值（上一阶段计算所得）加载，否则保持不变

# 4、访问内存

- 获取内存中的数据
- LW指令
  - 把在上一阶段计算得到的ALUOut寄存器中的地址加载到地址寄存器MAR
  - 读取存储器，一个32位的数据被放进数据寄存器MDR

# LW指令

- 访问内存
- LW指令
  - MAR ← ALUOut
  - MDR←Mem[MAR]

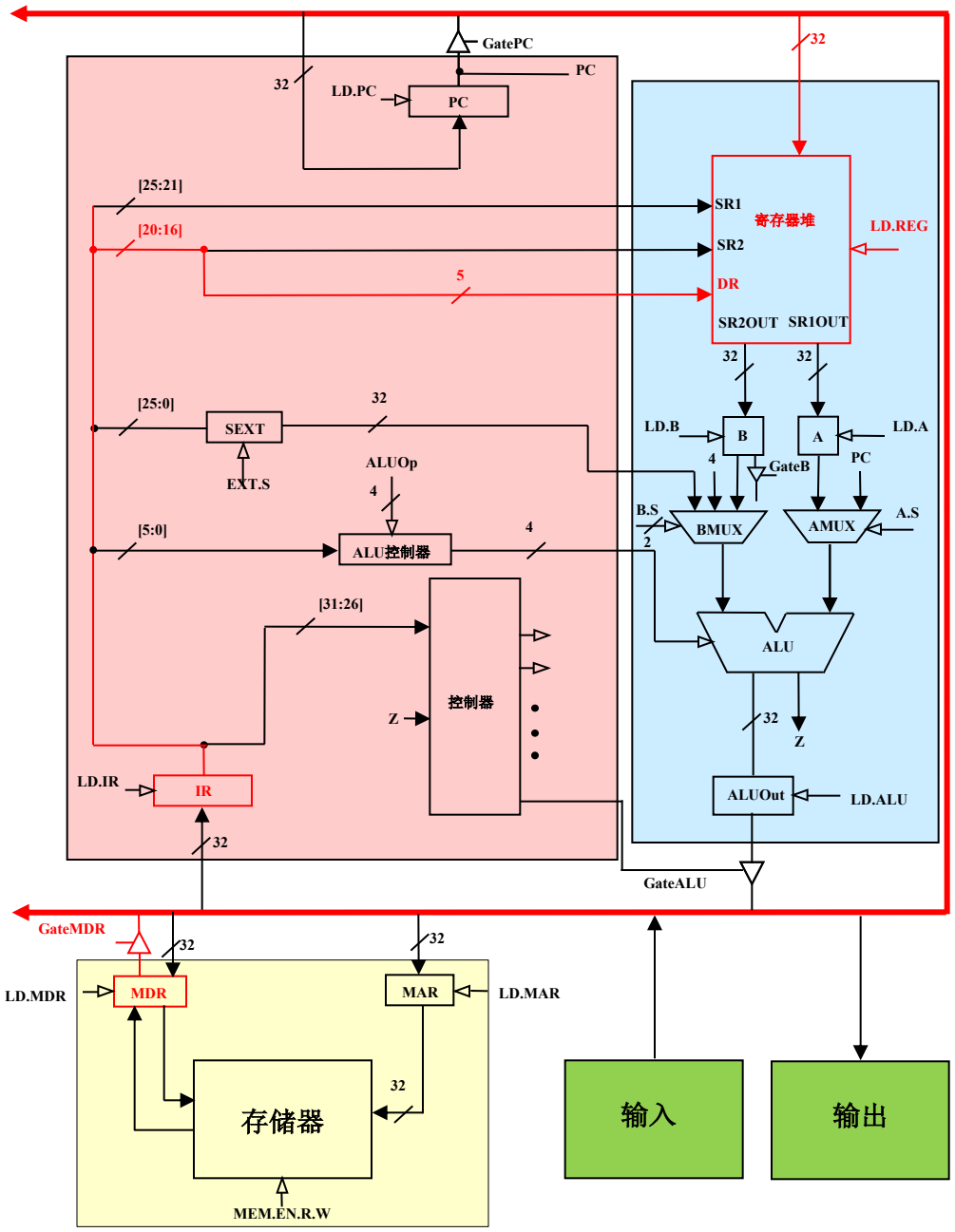


# 5、写回

- 结果被写到指定的目标中
- LW指令
  - MDR中的值被写入IR[20:16]所指示的R2中
- ADD指令
  - 加法运算的ALUOut寄存器中的结果被写入IR[15:11]所指示的R6中

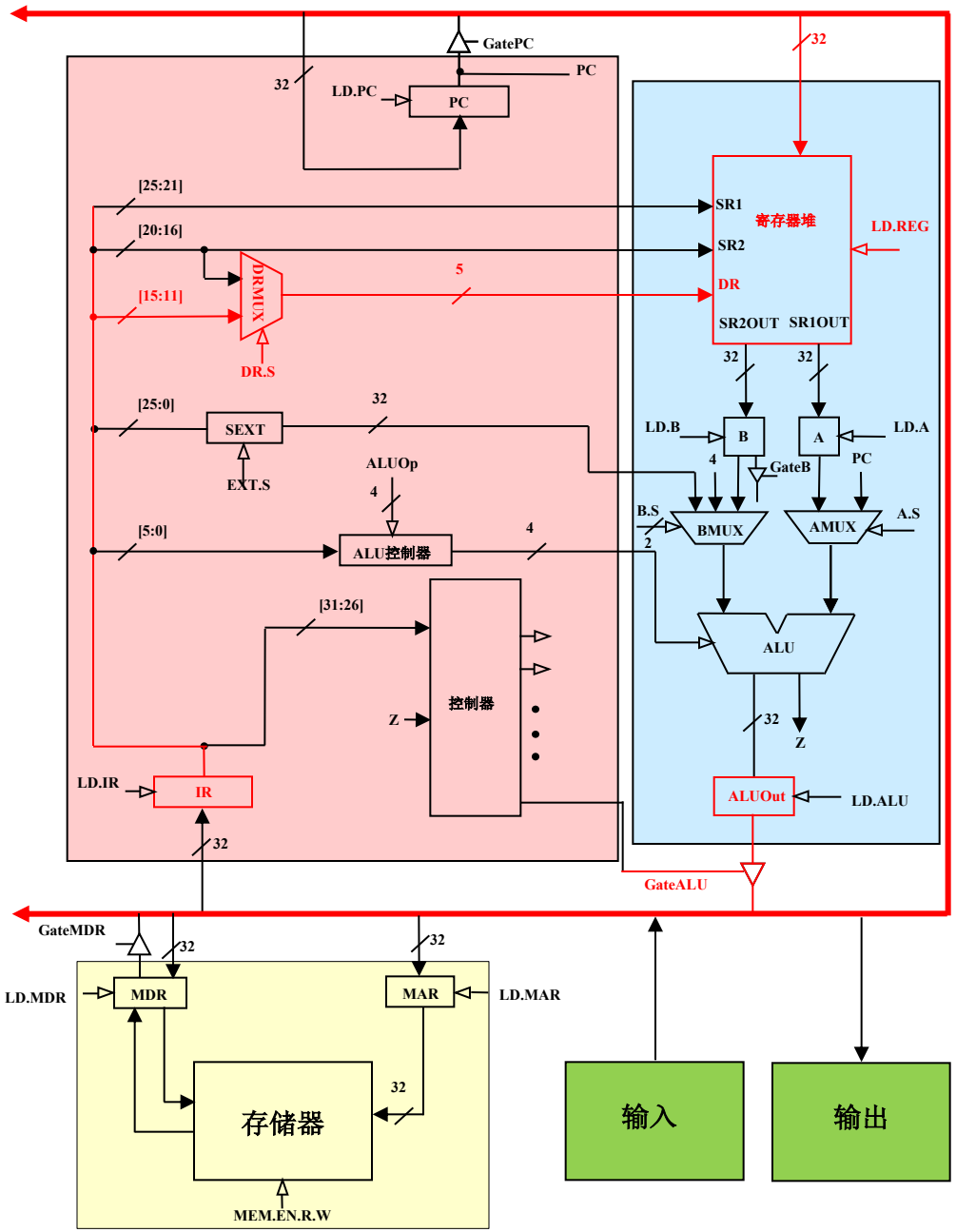
# LW指令

- 结果写到指定目标
- LW指令
  - (IR[20:16])←MDR



# ADD指令

- (IR[15:11]) ← ALUOut



# 下一阶段

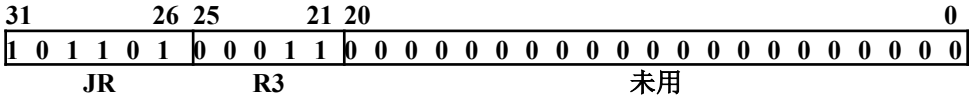
- 这五个阶段完成之后，控制单元就会从取指令阶段开始执行下一条指令
- 由于在取指令阶段PC被更新，包含了存储在存储单元中的下一条指令的地址
- 这样下一条指令接下来就会被读取。处理就这样持续下去直到被打断
- 不是所有的DLX指令都包括上述五个阶段。但是所有指令均需要取指令阶段和译码/取寄存器阶段
  - ADD指令，不需要访问内存阶段

# 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

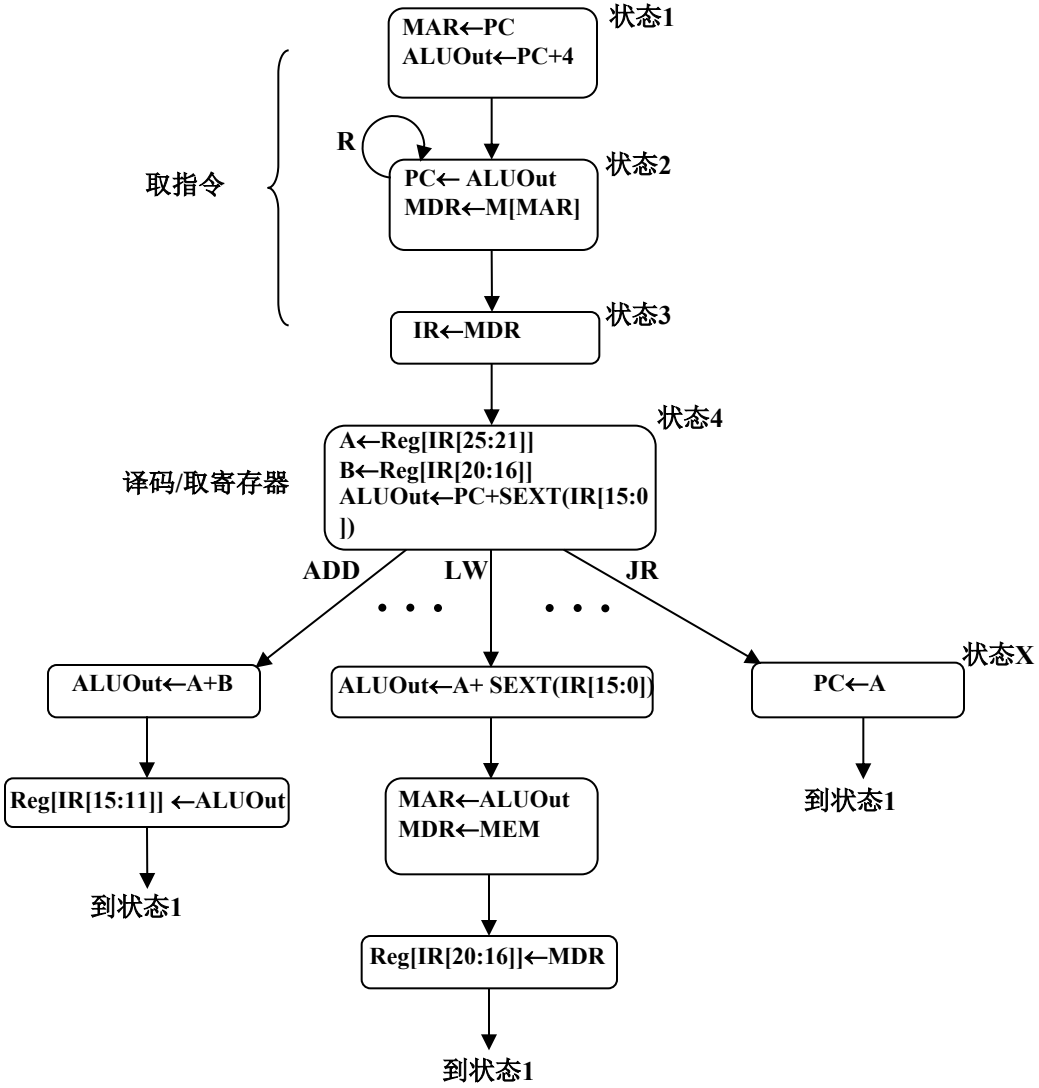
- 从PC=x80008000开始
- 取指令阶段：IR被加载为JR指令，PC更新为地址x80008004
- 译码/取寄存器阶段：译码，并取出R3和R0中的值，计算PC与SEXT(IR[15:0])的和
- 完成分支阶段：PC被加载为x80004000（假设在指令开始处R3的内容为x80004000）
- 指令执行完毕（只需要3个阶段）
- 要处理的下一条指令将位于地址x80004000而不是在地址x80008004

# DLX的有限状态机

# DLX的有限状态机

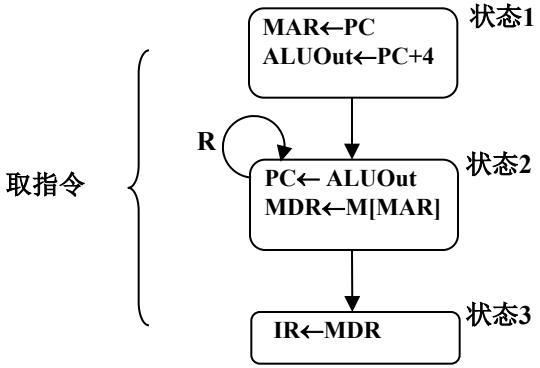
- 一条指令的执行可能包含3~5个阶段，每一个阶段还由一些步骤组成，而每一个阶段的每一步都是由控制单元的有限状态机控制的
- 状态在时钟控制下发生转换
- 寄存器传送语言 RTL（Register Transfer Language）
  - 硬件描述语言
    - M[xx]表示在存储器中xx地址的值
    - Reg[xx]表示寄存器xx的值

# 简化的状态图


MAR←PC
ALUOut←PC+4
状态1
R
取指令
PC← ALUOut
MDR←M[MAR]
状态2
IR←MDR
状态3
状态4
A←Reg[IR[25:21]]
B←Reg[IR[20:16]]
ALUOut←PC+SEXT(IR[15:0])
译码/取寄存器
ADD
LW
JR
ALUOut←A+B
ALUOut←A+ SEXT(IR[15:0])
PC←A
状态X
Reg[IR[15:11]] ←ALUOut
MAR←ALUOut
MDR←MEM
到状态1
到状态1
Reg[IR[20:16]]←MDR
到状态1

# 取指令阶段

- 需要多个时钟周期



# 状态1

- 第一个时钟周期

  MAR←PC

  ALUOut←PC+4

  - 控制信号

    GatePC=1

    LD.MAR=1

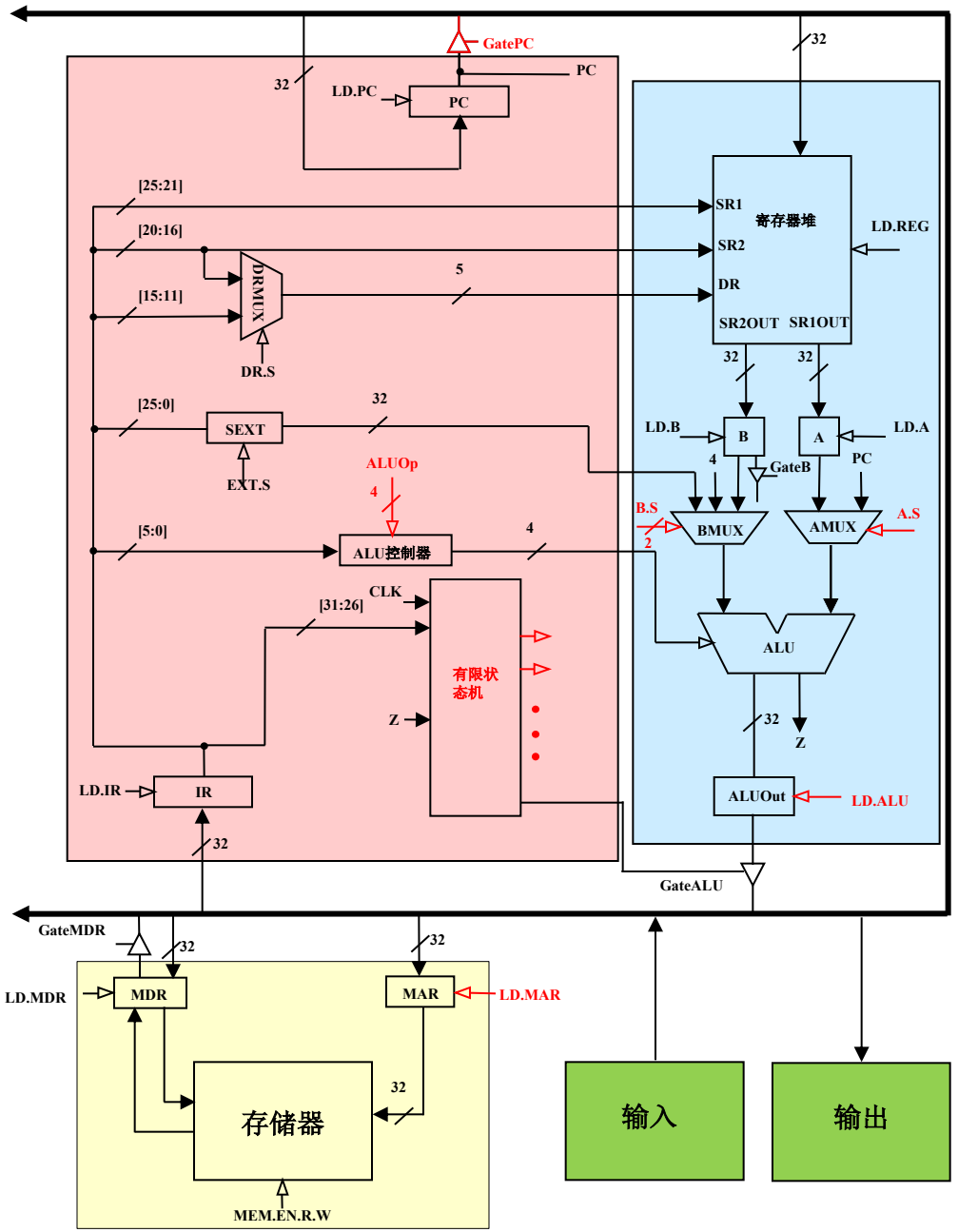
    A.S=1

    B.S=01

    ALUOp=0001

    LD.ALU=1

    ……



# 状态2

PC ← ALUOut

MDR ← M[MAR]

- 控制信号

  GateALU=1

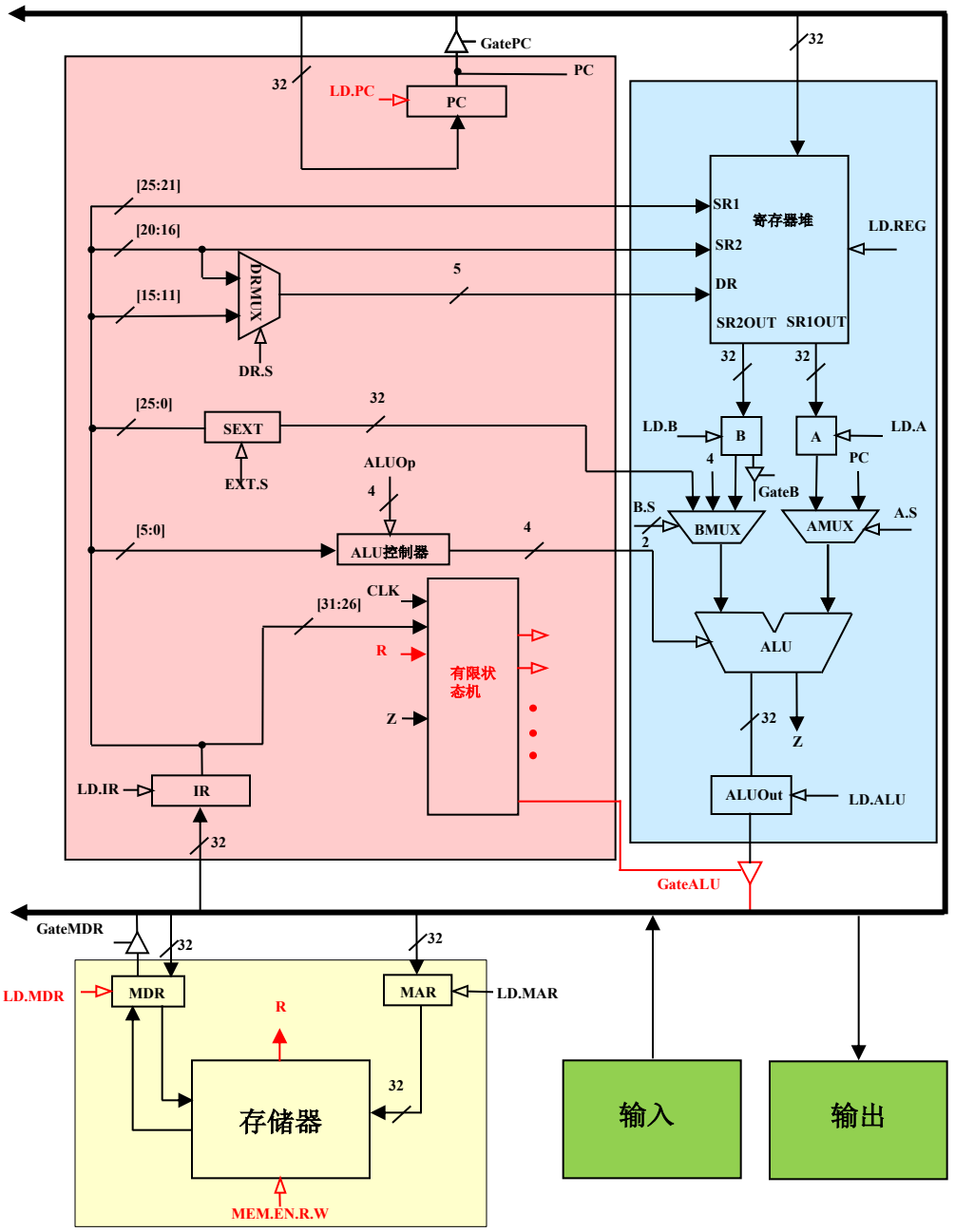
  LD.PC=1

  MEM.EN.R.W=0

  LD.MDR=1

  就绪信号（Ready）

  ……



- 状态2，有限状态机同时将
  - MEM.EN.R.W设为0，LD.MDR设为1
    - 读存储器，MDR被加载从存储器中读取的指令
    - 可能需要多个时钟周期，所以需要一个就绪信号（Ready）表示是否就绪，当读取结束，存储器将R（Ready）信号设为1
  - GateALU和LD.PC信号设为1
    - 将ALUOut的值写入PC

# 状态3

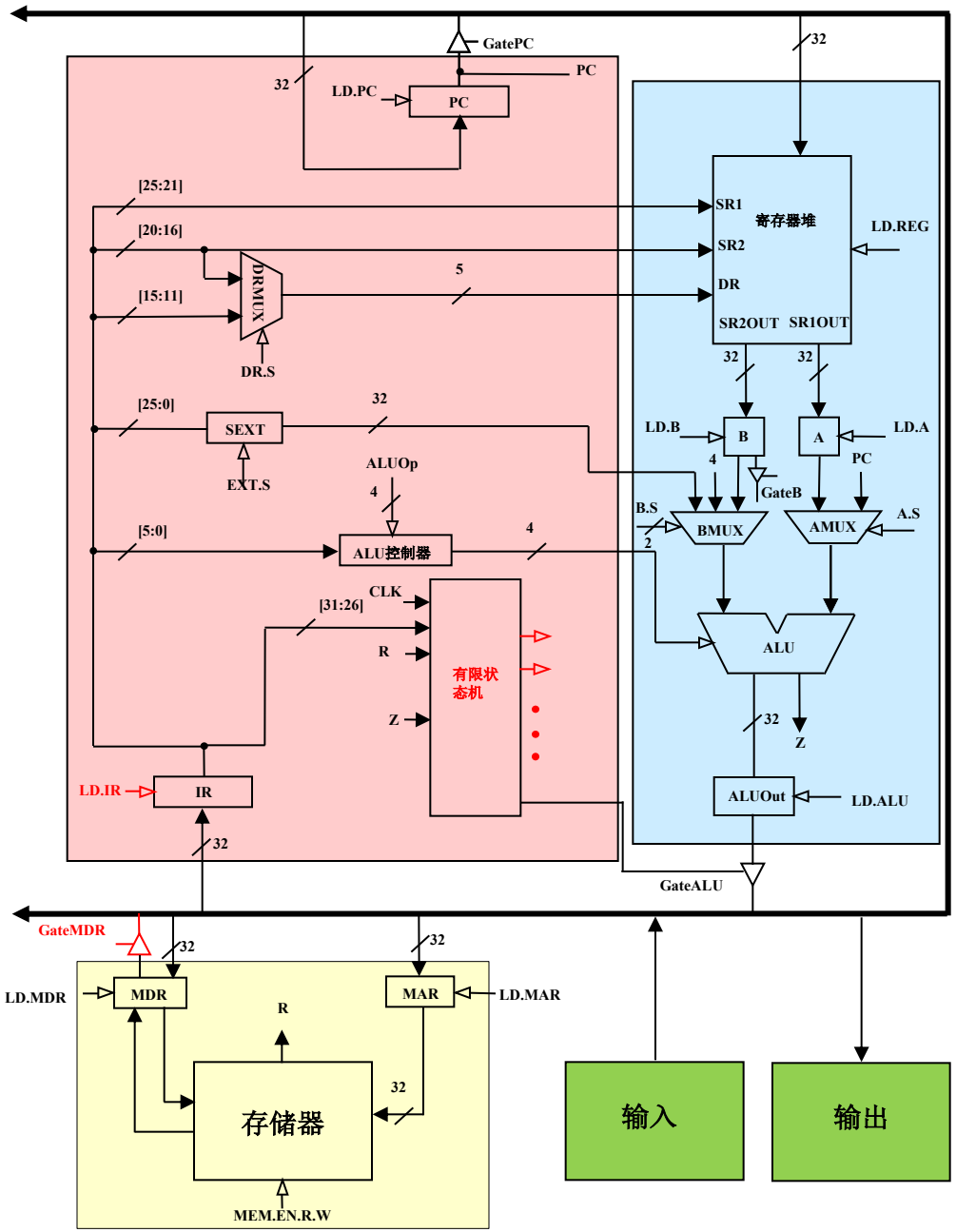
**IR←MDR**

- 控制信号

  GateMDR=1

  LD.IR=1

  ……



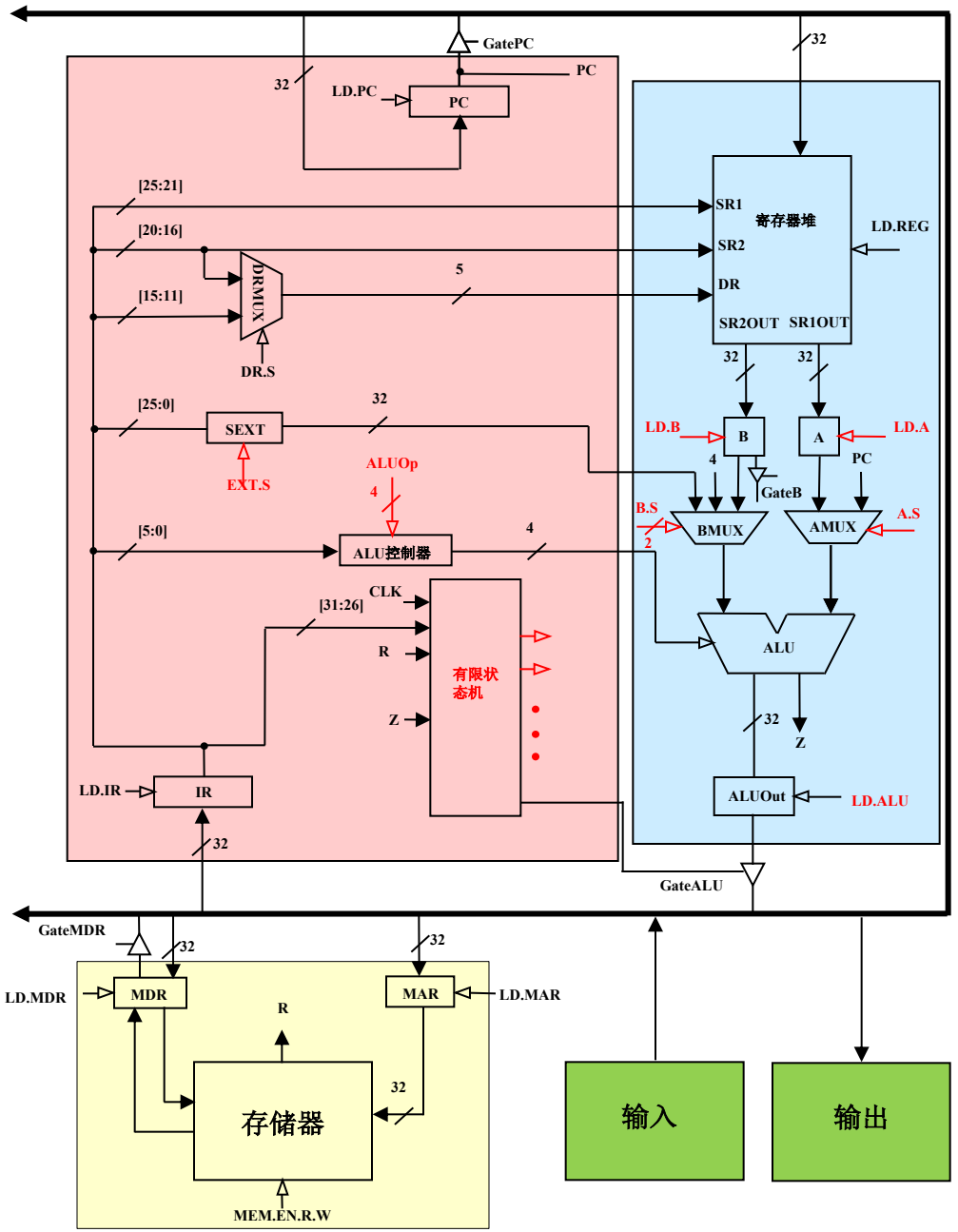
# 译码/取寄存器阶段

- 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
……



# 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

- 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

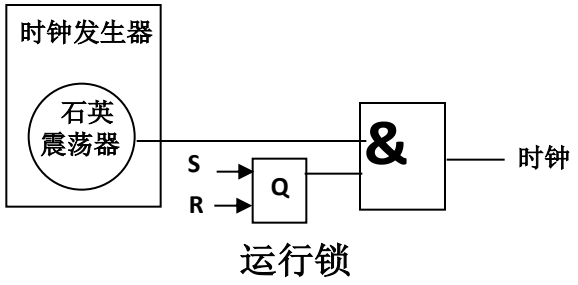
# 停止时钟

- 计算机将一个时钟周期接一个时钟周期，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，计算机就再次开始运行操作系统，也可以在操作系统控制下，开始运行另一个用户程序

  ——计算机总是在持续地运行
- 那么，这种情况就会一直持续下去直到断掉计算机的电源吗？
- 停止时钟，就可以停止指令的运行

# 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



# 习题

- 8. 2