ICS期末复习习题课

2025/1/3

数据结构

- **期末考**试中，一般会有一道关于数据结构的大题，使用LC3汇编语言考察**链表、队列、栈、字符串等常**见数据结构及其基本操作。
- **做**这类题目，需要知道各种数据结构的基本特征（比如栈是后进**先出**，队列是先进先出），以及各种基本操作在课本的什么地方**可以抄**。

栈

- **主要特点**：LIFO (Last In First Out) 后进先出。
- **关**键变量：栈顶指针
- **两种基本操作**：
  - POP：进栈，将元素压入栈顶。代码见课本P276（**无溢出**检查版）
  - PUSH：**出**栈，将元素从栈顶弹出。代码见课本P275（**无溢出**检查版）
- **异常情形**：课本中有参考代码，但具体情况要看题目的规定
  - **上溢出**：**在**栈已满的情形试图进栈。代码见课本P278
  - **下溢出**：**在**栈已空的情形试图出栈。代码见课本P277

队列

- **主要特点**：FIFO (First In First Out) **先**进先出。
- **关**键变量：队头指针，队尾指针
- **两种基本操作**：
  - Insert：**入**队，将元素放入队尾。代码见课本P295（**无溢出**检查版）
  - Remove：**出**队，将元素从队头取出。代码见课本P295（**无溢出**检查版）
- **异常情形**：课本中有参考代码，但具体情况要看题目的规定
  - 队空时试图出队。只需检查队头与队尾指针是否相同。代码见课本P297。
- **循**环队列：充分利用空间的特殊队列。操作代码见课本P296。

字符串

- 主要特点：以一维数组的形式，将一串字符存放在地址连续的内存里。
- 结束位置用x0000表示

例题1

- 我们用LC3汇编语言写了一个程序来判断字符串是否回文。一些指令需要补全。
- 程序接受用户键入的单词作为输入，一次读取一个字符，以回车作为结束标志，输入结束后判断是否回文。
- 若单词是回文字符串，标签OUTPUT标记的内存处会存入1，否则存入0。
- 程序用到了一个栈和一个队列，用到了子函数PUSH，POP，INSERT和REMOVE。R6是栈指针，R3和R4指向队头和队尾。
  - 对于所有的4个子函数，若执行成功R5会包含0，否则会包含1
  - PUSH：将R0的值存入栈顶
  - POP：将栈顶元素弹出，放入R0
  - INSERT：将R0的值放到队尾
  - REMOVE：将队头的元素取出，放入R0
- 假设栈和队列初始为空，内存空间足够，不会发生溢出

例题1

1. 写出POP和INSERT子函数代码
2. 补全程序中的空

观察题目代码第一空，可以推断这一行是在判断刚输入的字符是不是回车来决定是否要结束输入过程，开始判断回文字符串。判断的基本方法一定是检查输入字符的ASCII码。我们观察到代码结尾处有一个NEG_ENTER标签，其内容刚好是ENTER字符的相反数，且程序开头处将它的值加载到了R1，因此容易判断这一行指令的内容是将输入字符R0与回车的ASCII码相减，判断计算结果是不是零。

需要注意的是，这里的计算结果不能存入R0，否则后续的PUSH和INSERT子函数无法将正确的输入值入栈和入队；不能存入R1，因为在输入过程中，R1的值不会反复被加载，必须始终保持ENTER的相反数；不能存入有特殊用途的R3、R4、R6。假如我们令计算结果被存入R2，则第一空答案是ADD R2, R0, R1。

```
    .ORIG x3000
    LD R6, STACK
    LD R3, QUEUE
    ADD R4, R3, #0

    LD R1, NEG_ENTER
INPUT
    TRAP x23
    _____________
    BRz CHECK
    JSR PUSH
    JSR INSERT
    BRnzp INPUT

CHECK
    JSR POP
    _____________
    BRnp A
    ADD R1, R0, #0
    JSR REMOVE
    _____________
    _____________
    _____________
    BRz CHECK
    AND R0, R0, #0
    BRnzp DONE

A
    AND R0, R0, #0
    ADD R0, R0, #1
DONE
    ST R0, OUTPUT
    HALT

NEG_ENTER   .FILL #-10
STACK       .FILL xFE00
QUEUE       .FILL x8000
OUTPUT      .BLKW #1
    .END
```

例题1

1. 写出POP和INSERT子函数代码
2. 补全程序中的空

代码第二空发生在判断阶段出栈以后，并且其结果会直接影响程序是否结束。我们发现如果该指令的结果非零，程序会结束并且OUTPUT处存入1，指示程序检验到了回文字符串，说明检验已经全部完成。由于前文说过当栈空时调用POP子函数，会发生异常并且R5的值会是1，可以推断第二空后的BR指令应该基于R5的结果决定跳转。而且这一空无需对寄存器的值做出任何改动，答案应该是ADD R5, R5, #0.

```
    .ORIG x3000
    LD R6, STACK
    LD R3, QUEUE
    ADD R4, R3, #0

    LD R1, NEG_ENTER
INPUT
    TRAP x23
    _____________
    BRz CHECK
    JSR PUSH
    JSR INSERT
    BRnzp INPUT

CHECK
    JSR POP
    _____________
    BRnp A
    ADD R1, R0, #0
    JSR REMOVE
    _____________
    _____________
    _____________
    BRz CHECK
    AND R0, R0, #0
    BRnzp DONE

A
    AND R0, R0, #0
    ADD R0, R0, #1
DONE
    ST R0, OUTPUT
    HALT

NEG_ENTER   .FILL #-10
STACK       .FILL xFE00
QUEUE       .FILL x8000
OUTPUT      .BLKW #1
    .END
```

例题1

1．写出POP和INSERT子函数代码

2．补全程序中的空

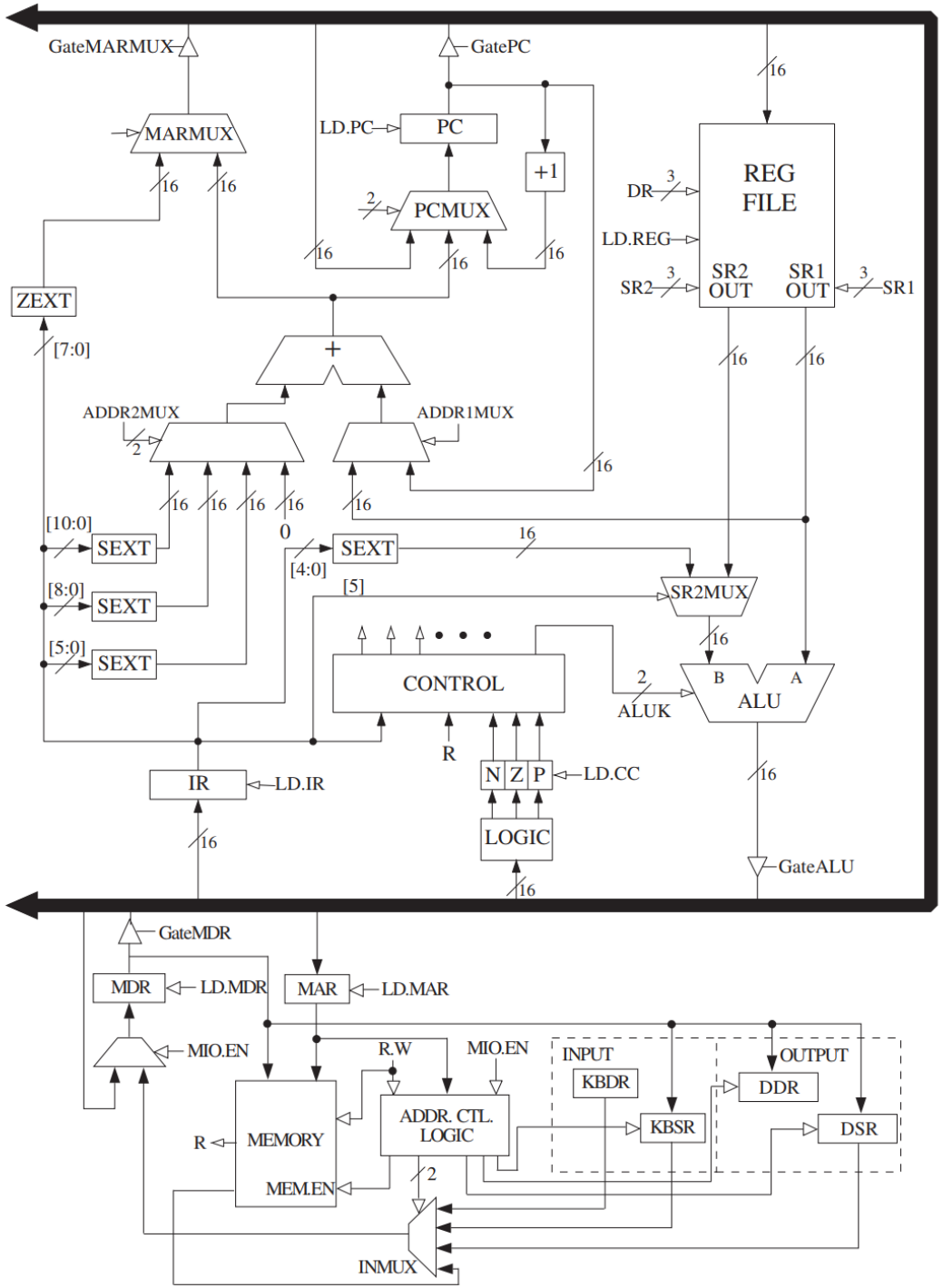
代码第3、4、5空的执行结果会直接影响程序是否结束。若执行结果是0，则继续循环检验字符串；若不是0，则**R0被置零，表示字符串不是回文字符串，然后程序直接**结束。由此我们推断这三行代码的作用是将栈和队列中取出的两个值相减。观察代码发现，此时栈中弹出的元素在**R1内**，队列中取出的元素在**R0内，将两者相减即可**。计算结果可以放在两个寄存器的任意一个中。**因此第3、4、5空的答案**为：

```
NOT R0, R0
ADD R0, R0, #1
ADD R0, R0, R1
```

```
    .ORIG x3000
    LD R6, STACK
    LD R3, QUEUE
    ADD R4, R3, #0

    LD R1, NEG_ENTER
INPUT
    TRAP x23
    ____________
    BRz CHECK
    JSR PUSH
    JSR INSERT
    BRnzp INPUT

CHECK
    JSR POP
    ____________
    BRnp A
    ADD R1, R0, #0
    JSR REMOVE
    ____________
    ____________
    ____________
    BRz CHECK
    AND R0, R0, #0
    BRnzp DONE

A
    AND R0, R0, #0
    ADD R0, R0, #1
DONE
    ST R0, OUTPUT
    HALT

NEG_ENTER   .FILL #-10
STACK       .FILL xFE00
QUEUE       .FILL x8000
OUTPUT      .BLKW #1
    .END
```

数据通路

- 期末考试中，一定会有一道围绕LC3数据通路的大题，类似上次课上做的习题中的后两道以及hw5最后一题。
- 做这些题目，重点是把第三版课本702页的状态图和704页的数据通路弄懂。（一定要去看第三版课本，二、三两版课本有很大的区别）
- 弄懂的意思是，对每一条指令，能对着图想清楚它的执行过程。
- 课本P714还有一个加上了特权模式、中断等功能的数据通路，那个过于复杂没必要看

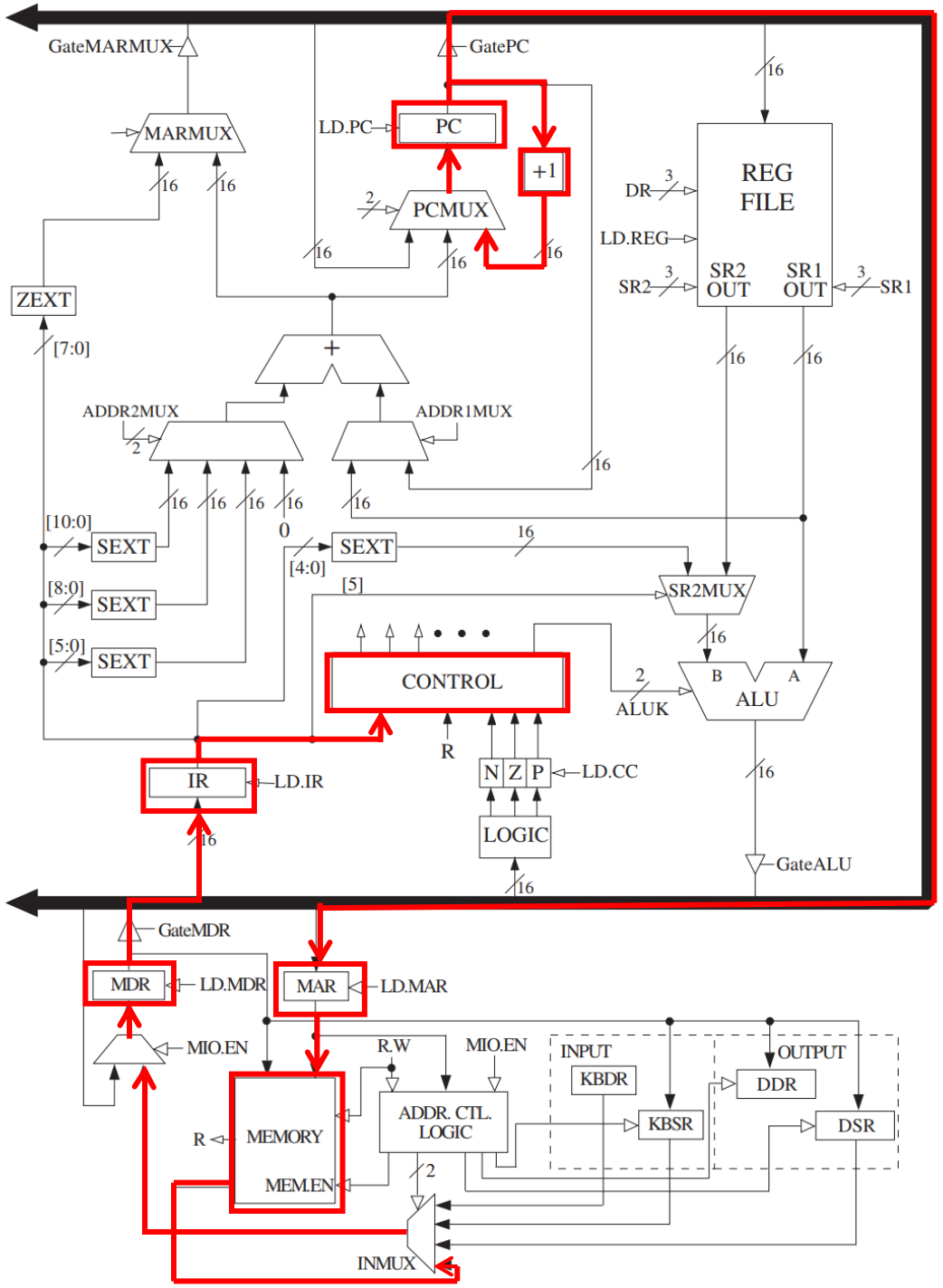


数据通路中的各类基本元件：

1. **Gate**：门控元件，控制数据是否通向**总线的开关。同**时最多只**有一个**Gate**可以**处于开启状态，否则会造成总线数据冲突。
2. **LD.XX**：**控制信号，代表某个部件是否需要写入新的**值。当信**号取**值为NOT时，部件的值不会更新；当取值为LOAD，**部件的**值会更新为输入的值。
3. **MUX**：**数据**选择器。它的基本功能是从若干个输入中选择一路输出。通常是梯形，长边上连接的是可供选择的若干路输入，侧边上连接的是控制信号，可以控制MUX**具体**选择某一路进行输出。
4. **ZEXT和SEXT**：**可以按一定的**规则对输入的数据进行位扩展。ZEXT**代表零**扩展，即在最前面补零。SEXT**代表符号位**扩展，在**最前面**补符号位，即负数补1，**非**负数补0.
5. **ALU**：计算单元，可以对两个输入进行一定规则的运算，输出结果。ALUK**可以控制**ALU进行具体某一种计算。LC3**中的**ALUK**是2位，ALU支持4种运算，分**别是ADD, AND, NOT**和**PASSA。**其中，ALU的两个**输入分别是A**和**B，ADD输出A+B，AND输出A&B，NOT输出~A，PASSA输出A。
6. **REG GILE**：**寄存器堆。支持**读和写功能。当LD.REG**有效**时写入，REG[DR] <= input。读功能则始终有效，SR1OUT = REG[SR1], SR2OUT = REG[SR2]。
7. **MEMORY**：**内存。支持**读和写功能。当MEM.EN**有效**时工作。R.W**控制内存具体**进行读或是写。MEMORY**有两个**输入，分别**是**输入数据和地址。

取指和译码

- 所有指令在取指和译码阶段是相同的
- MAR <- PC, PC <- PC + 1
- MDR <- M[MAR]
- IR <- MDR
- CONTROL模块根据IR的内容生成各种控制信号

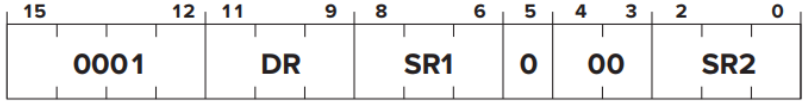


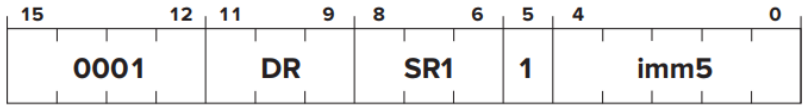
ADD

Assembler Formats

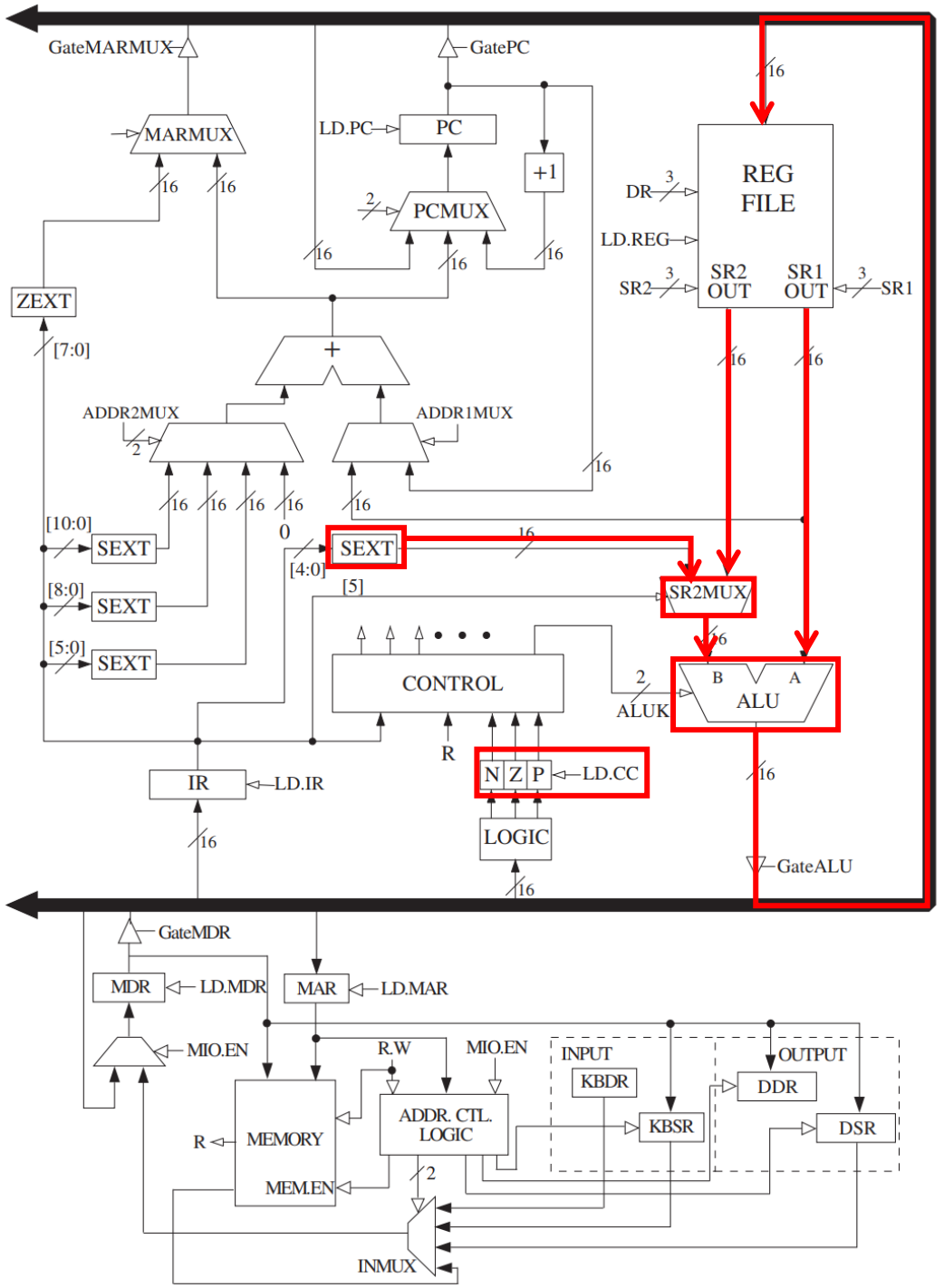
```
ADD   DR, SR1, SR2
ADD   DR, SR1, imm5
```

Encodings





- SR1MUX: 8:6
- SR2MUX: SEXT或SR2OUT，取决于指令的第二个操作数是立即数还是寄存器
- ALUK: ADD

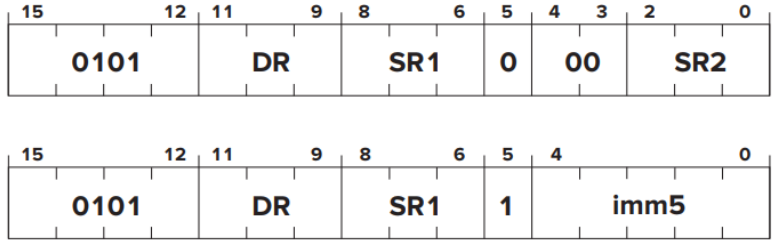


AND

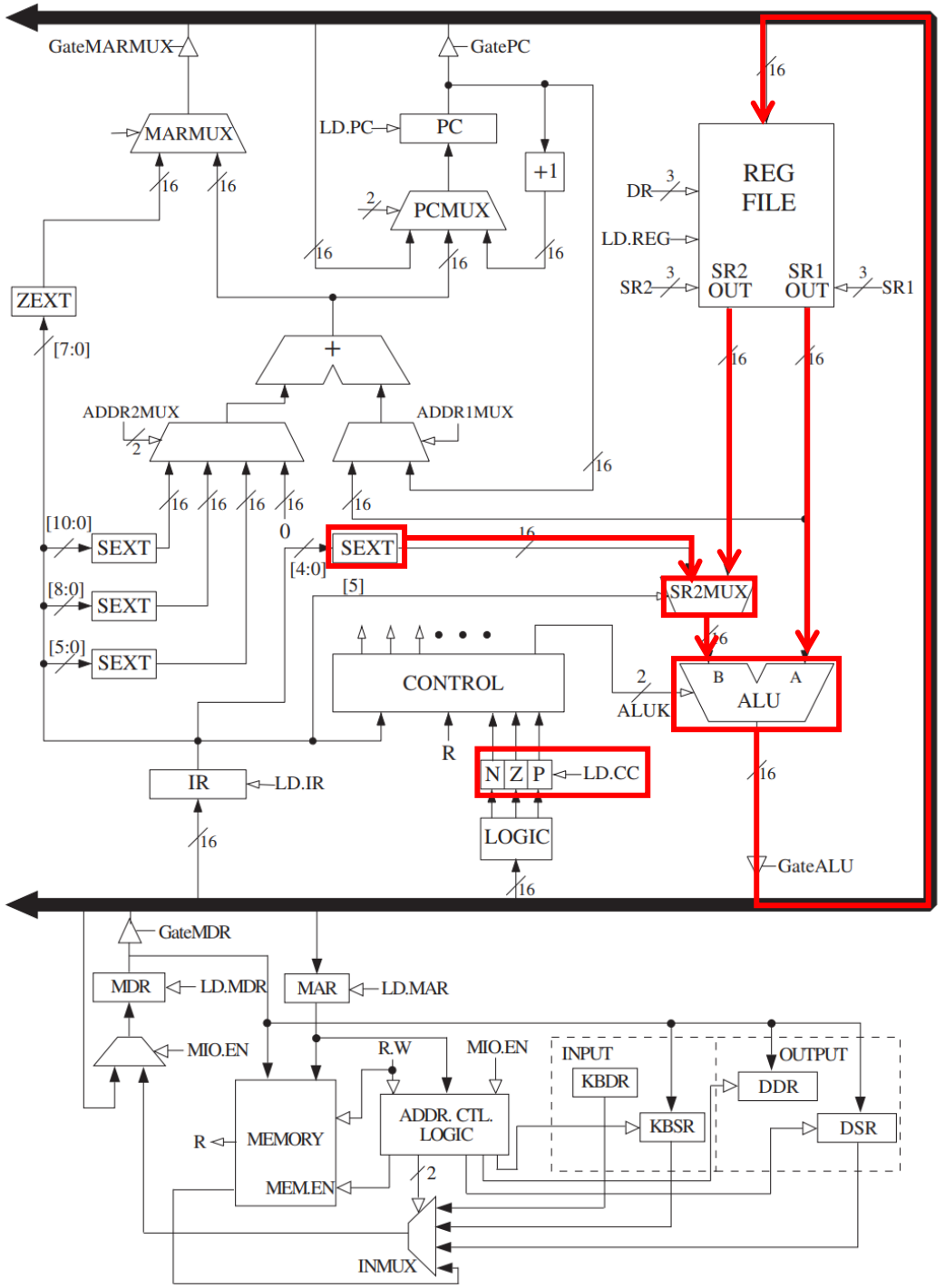
Assembler Formats

AND DR, SR1, SR2
AND DR, SR1, imm5

Encodings



- SR1MUX: 8:6
- SR2MUX: SEXT或SR2OUT，取决于指令的第二个操作数是立即数还是寄存器
- ALUK: AND

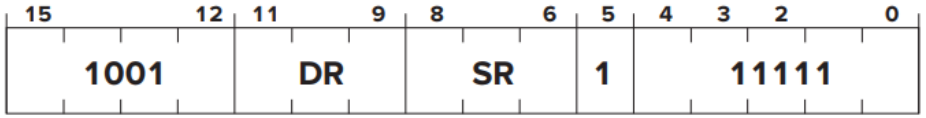


NOT

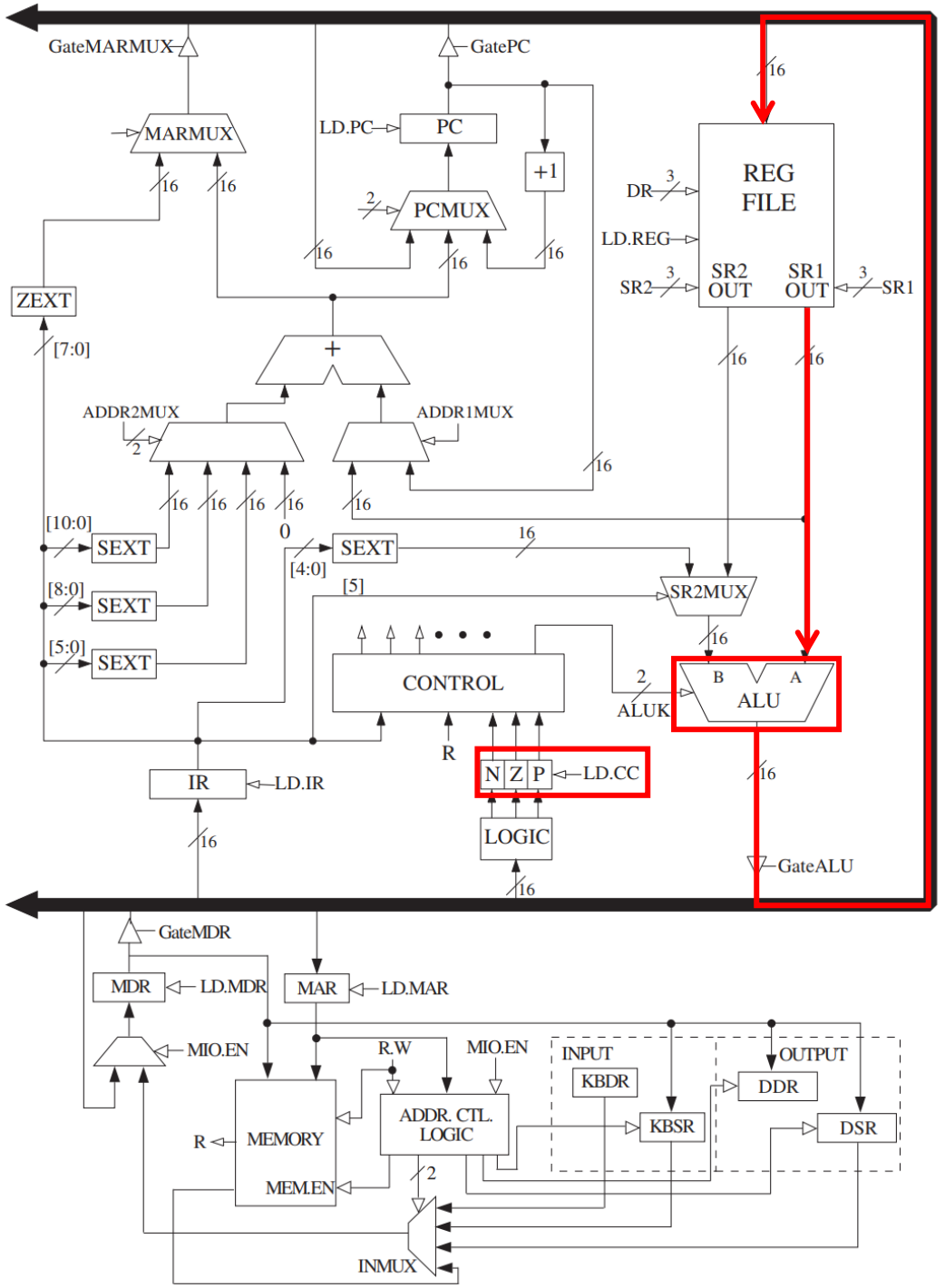
Assembler Format

NOT DR, SR

Encoding



- SR1MUX: 8:6
- ALUK: NOT

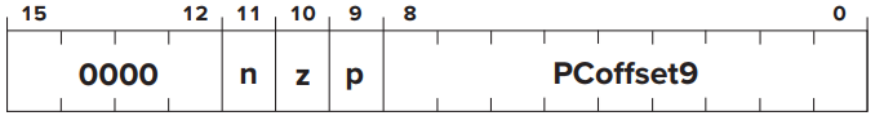


BR

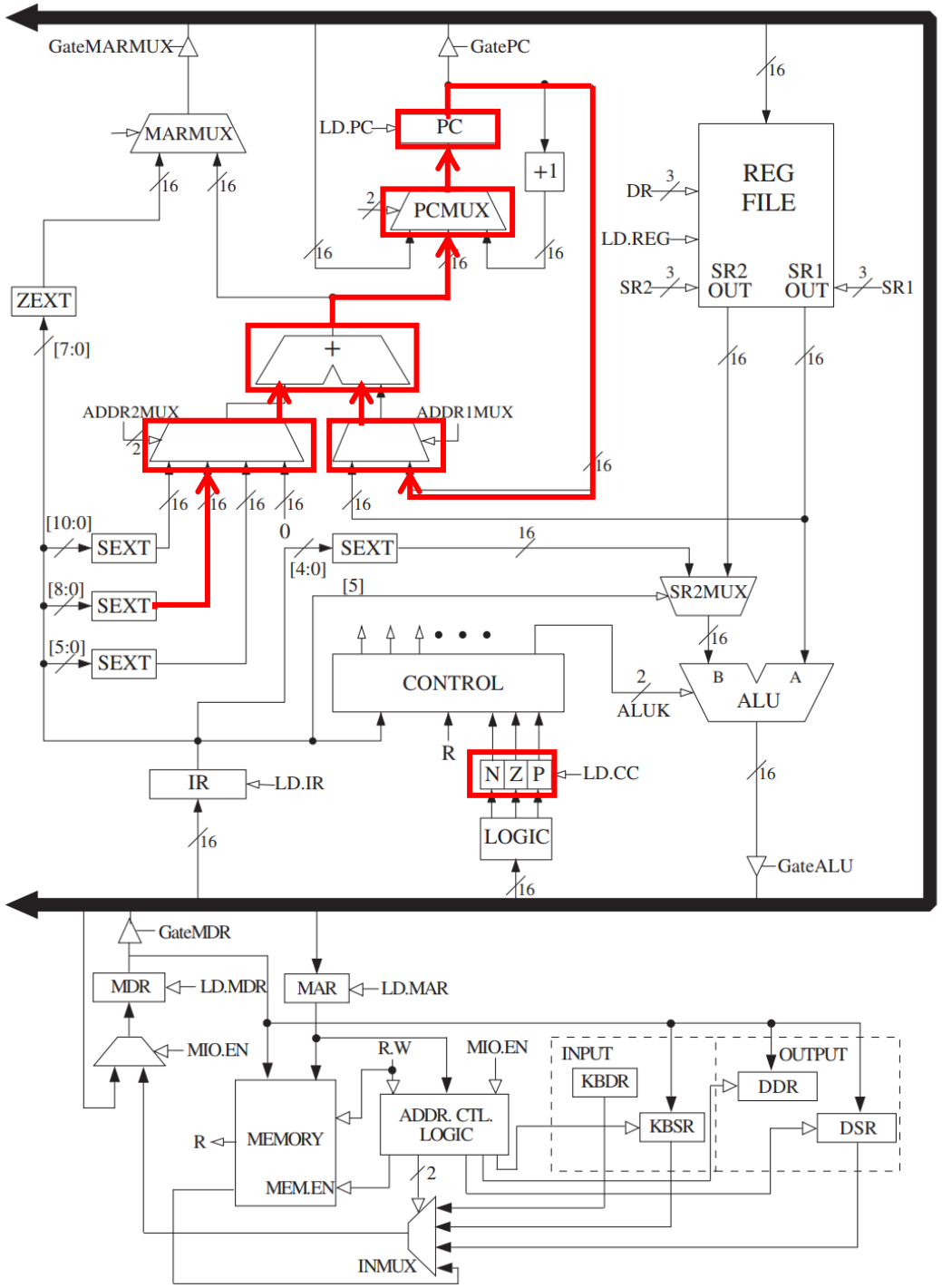
Assembler Formats

BRn	LABEL	BRzp	LABEL
BRz	LABEL	BRnp	LABEL
BRp	LABEL	BRnz	LABEL
BR†	LABEL	BRnzp	LABEL

Encoding



- ADDR1MUX: PC
- ADDR2MUX: PCoffset9
- PCMUX: ADDER

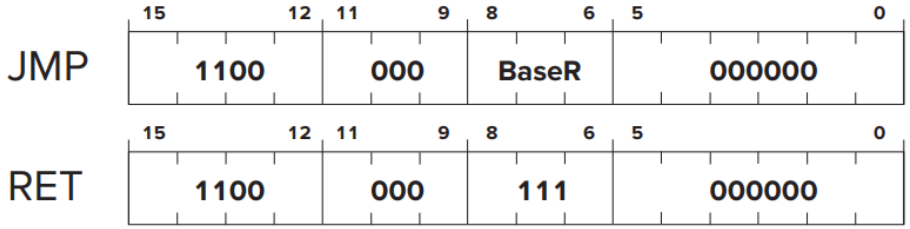


JMP RET

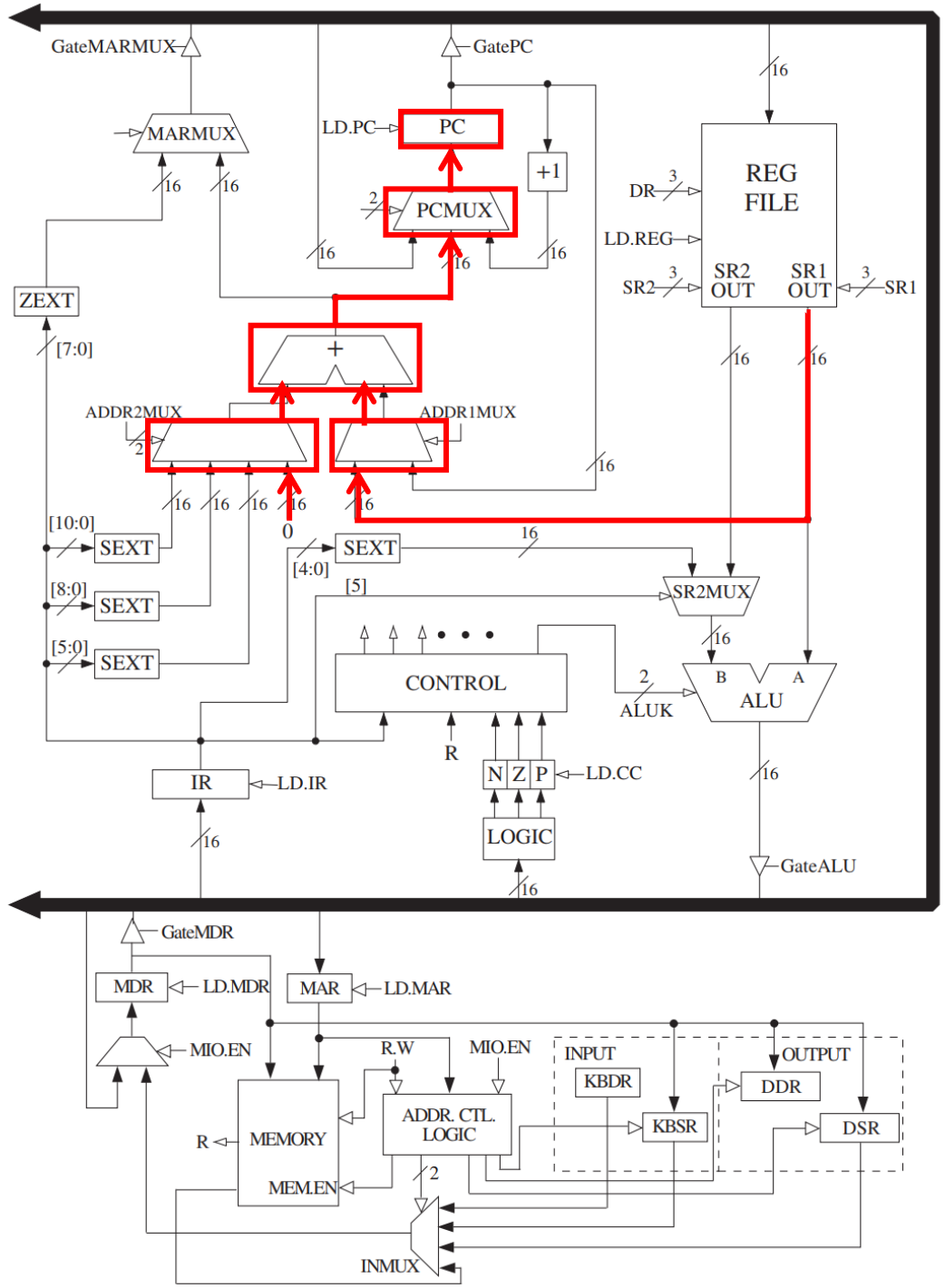
Assembler Formats

JMP BaseR
RET

Encoding



- ADDR1MUX: BaseR
- ADDR2MUX: ZERO
- PCMUX: ADDER

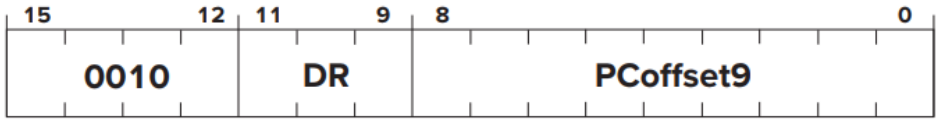


LD

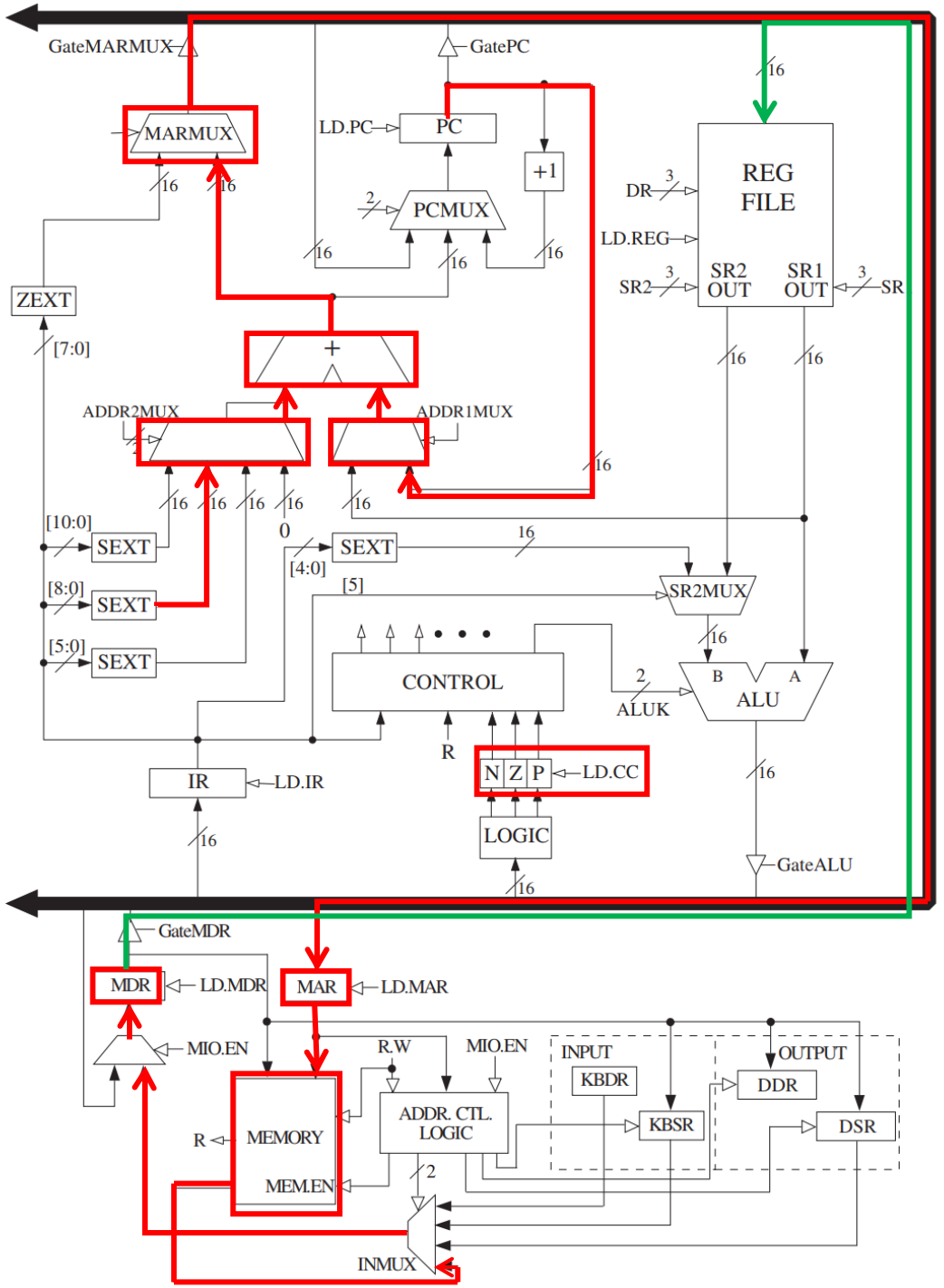
Assembler Format

LD DR, LABEL

Encoding



- ADDR1MUX: PC
- ADDR2MUX: PCoffset9
- MARMUX: ADDER
- DRMUX: 11.9

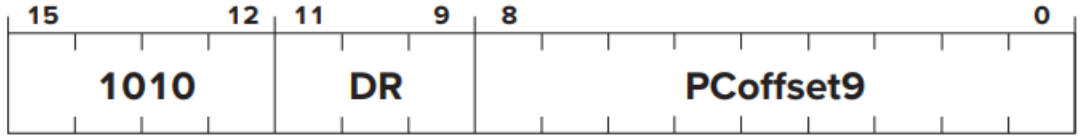


LDI

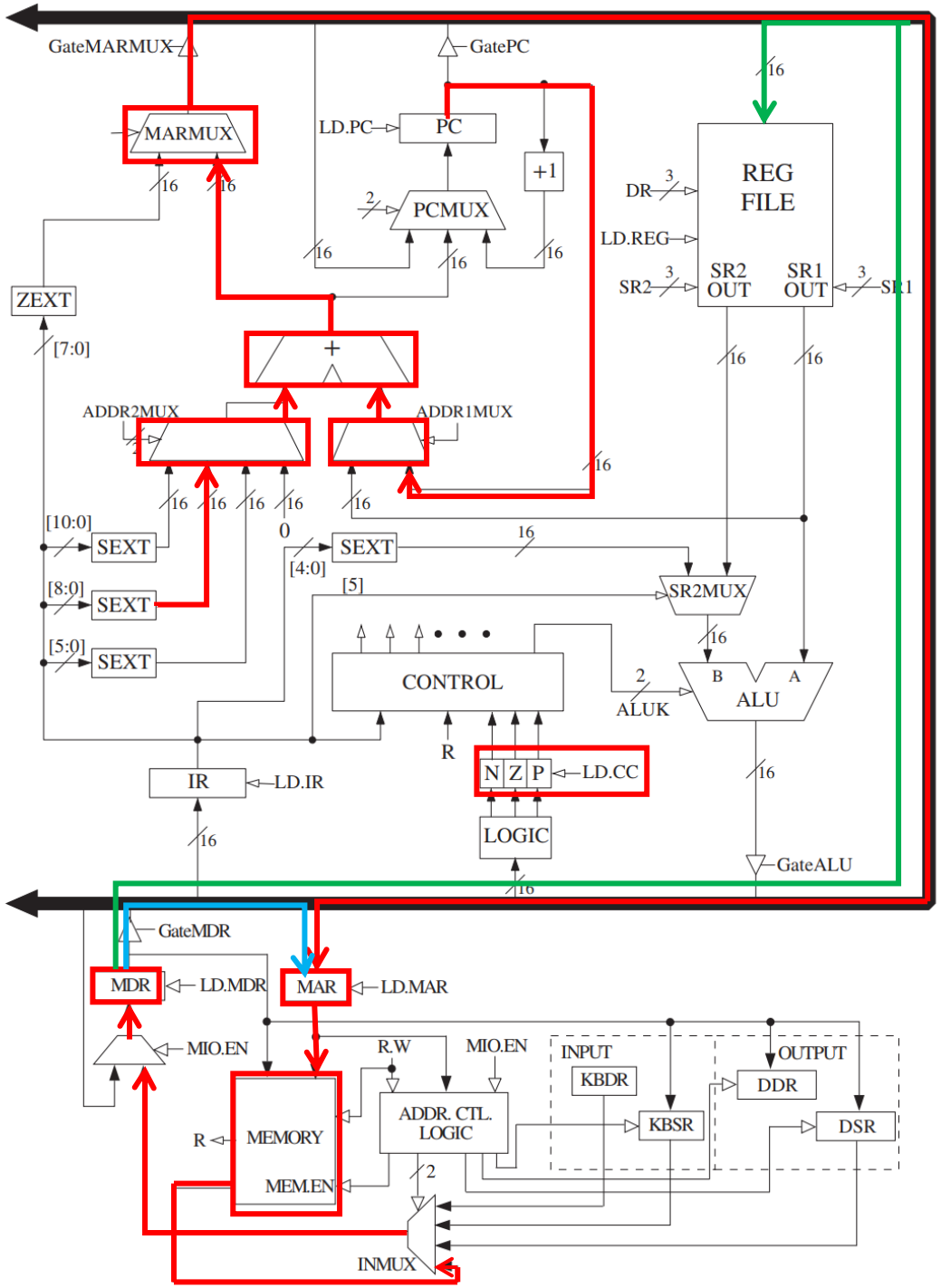
Assembler Format

LDI DR, LABEL

Encoding



- ADDR1MUX: PC
- ADDR2MUX: PCoffset9
- MARMUX: ADDER
- DRMUX: 11.9

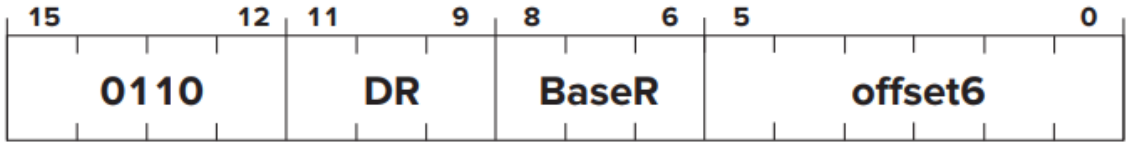


LDR

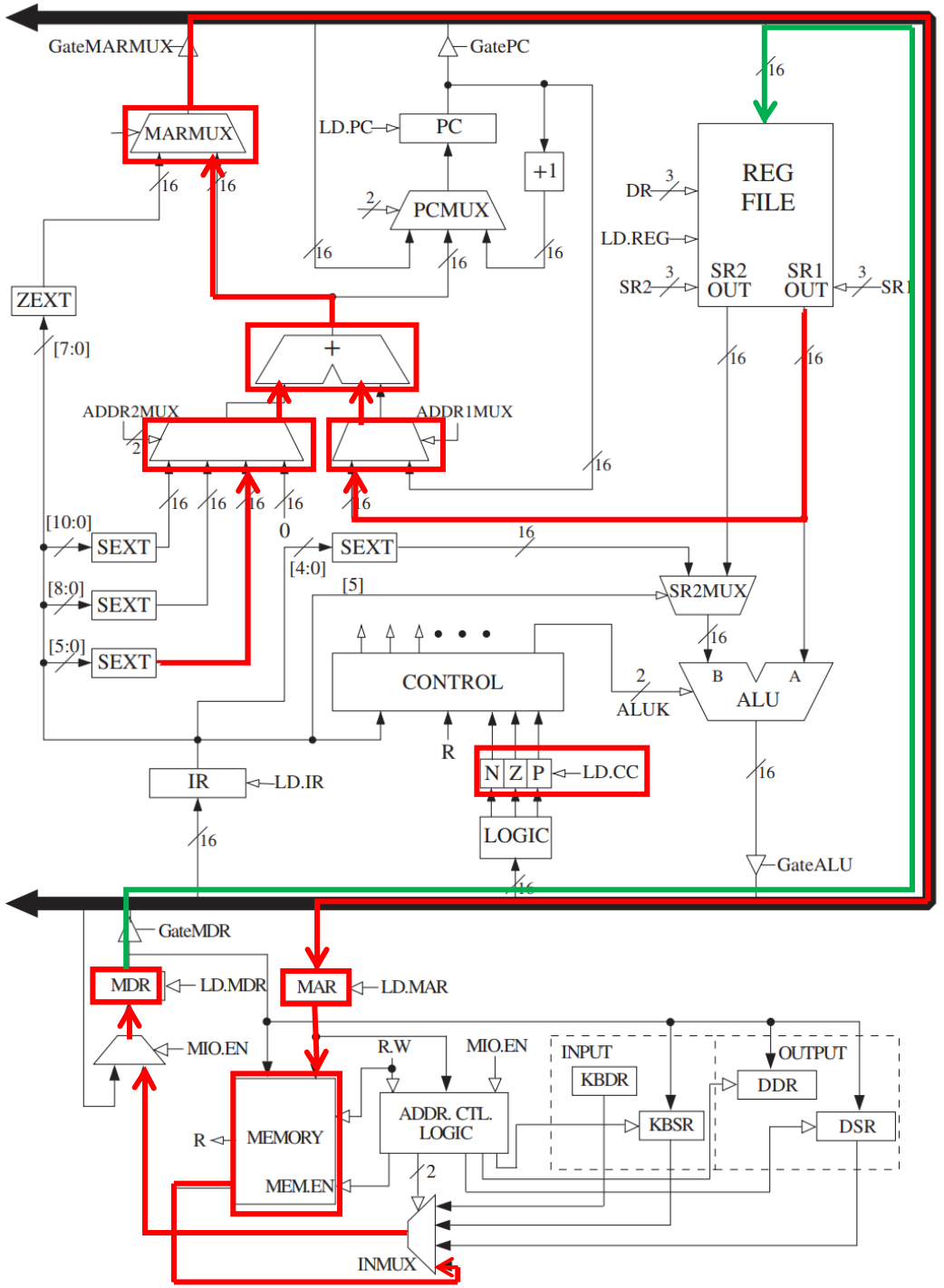
Assembler Format

LDR DR, BaseR, offset6

Encoding



- ADDR1MUX: BaseR
- ADDR2MUX: offset6
- MARMUX: ADDER
- SR1MUX: 8.6
- DRMUX: 11.9

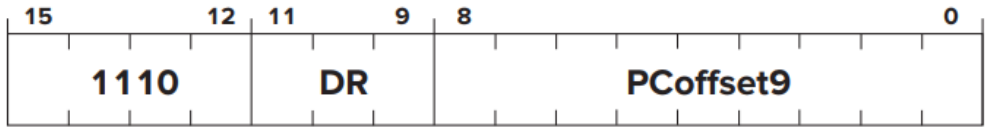


LEA

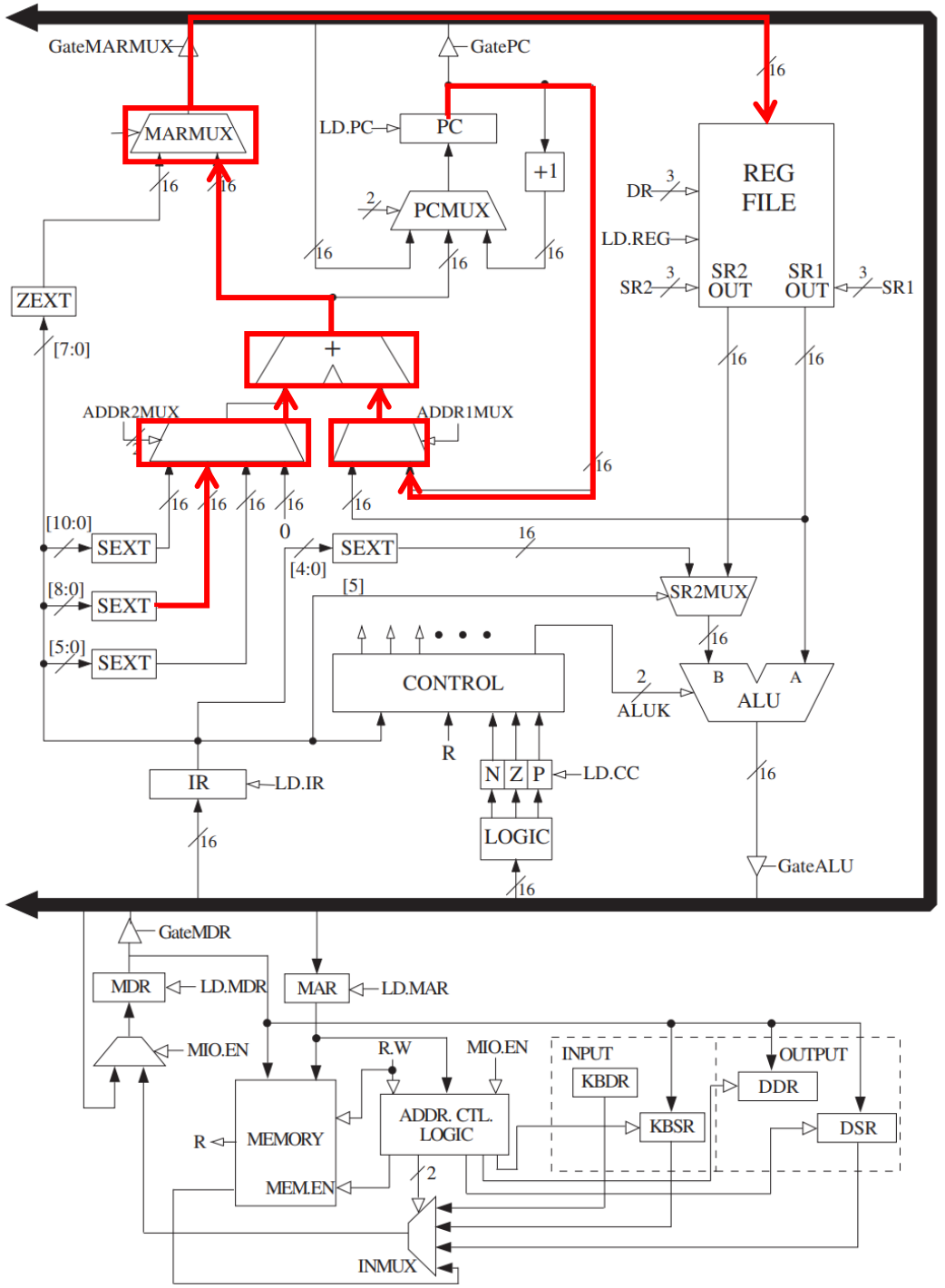
Assembler Format

LEA DR, LABEL

Encoding



- ADDR1MUX: PC
- ADDR2MUX: PCoffset9
- MARMUX: ADDER
- DRMUX: 11.9

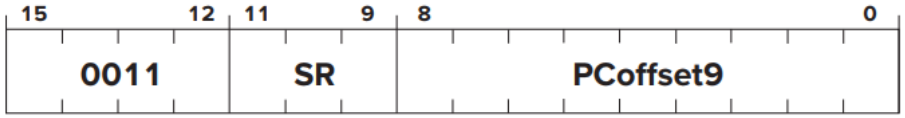


ST

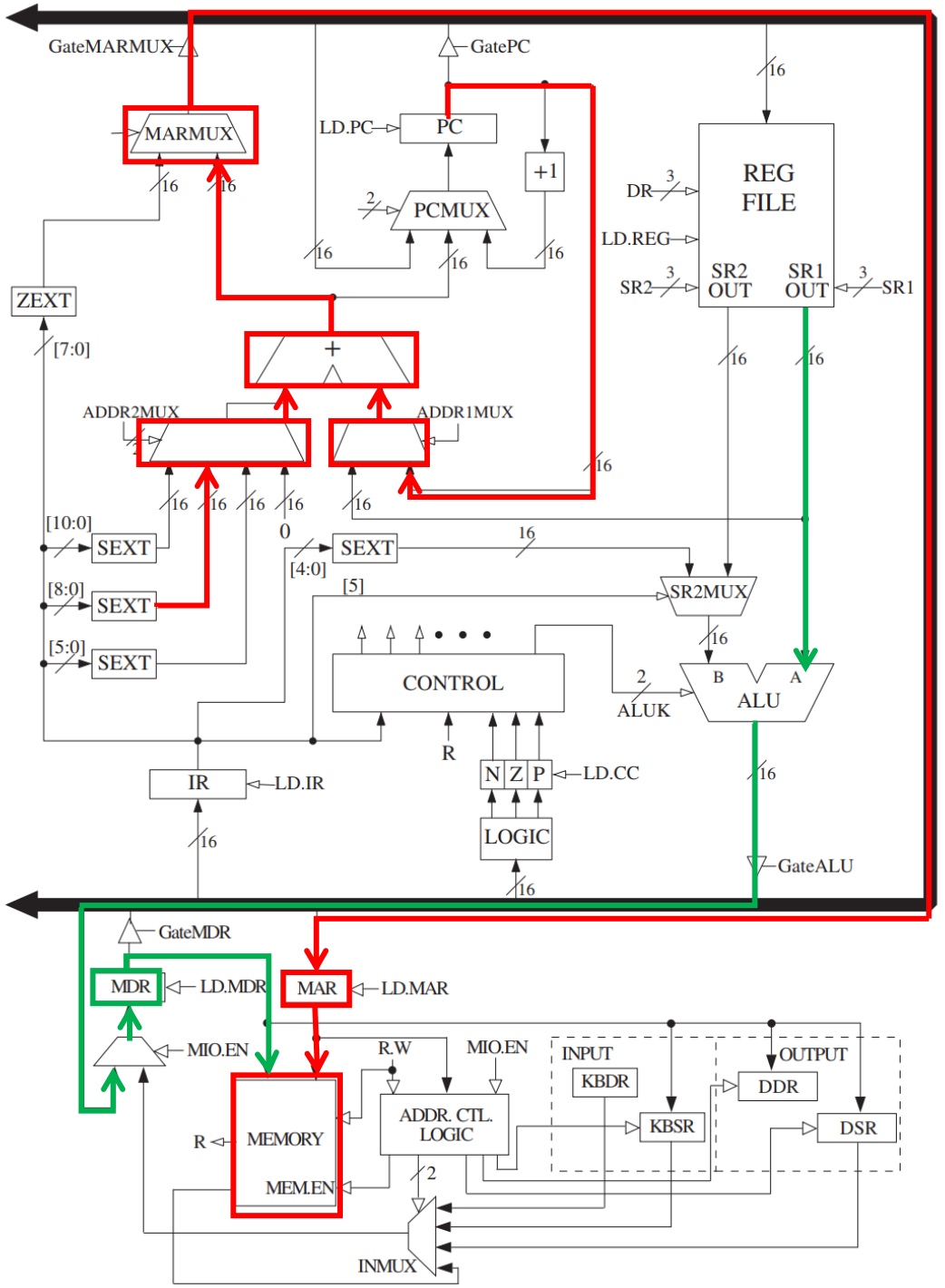
Assembler Format

ST SR, LABEL

Encoding



- ADDR1MUX: PC
- ADDR2MUX: PCoffset9
- MARMUX: ADDER
- SR1MUX: 11.9
- ALUK: PASSA

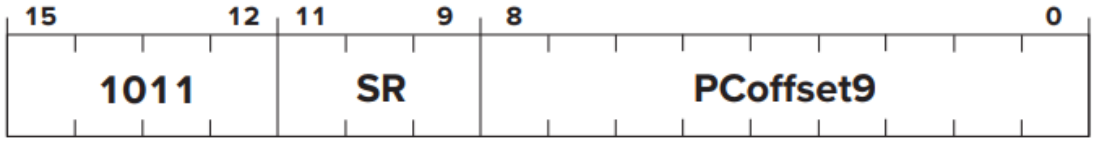


STI

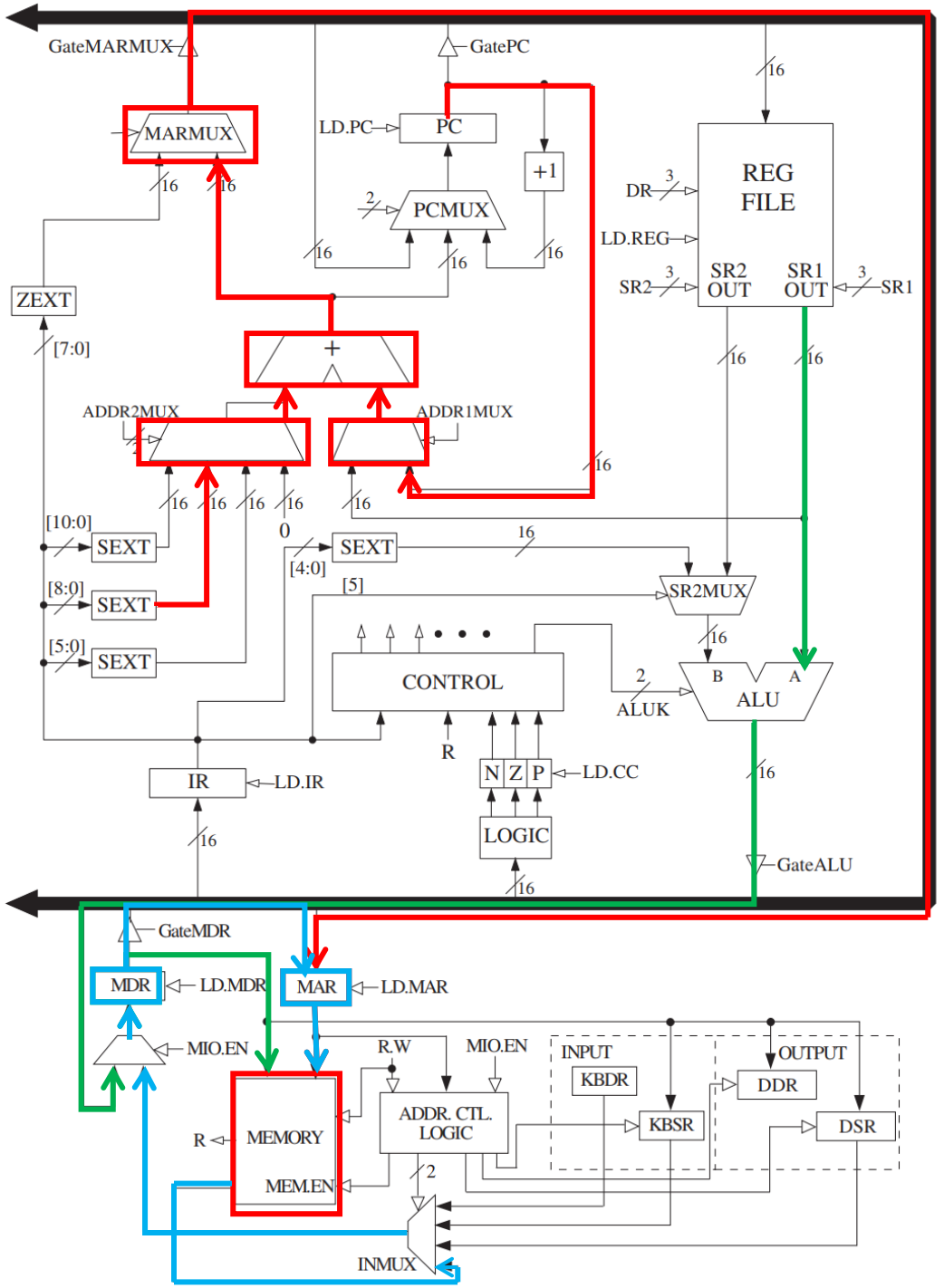
Assembler Format

STI SR, LABEL

Encoding



- ADDR1MUX: PC
- ADDR2MUX: PCoffset9
- MARMUX: ADDER
- SR1MUX: 11.9
- ALUK: PASSA

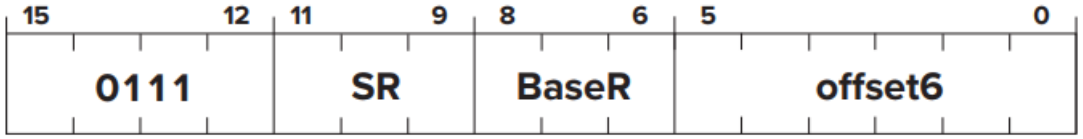


STR

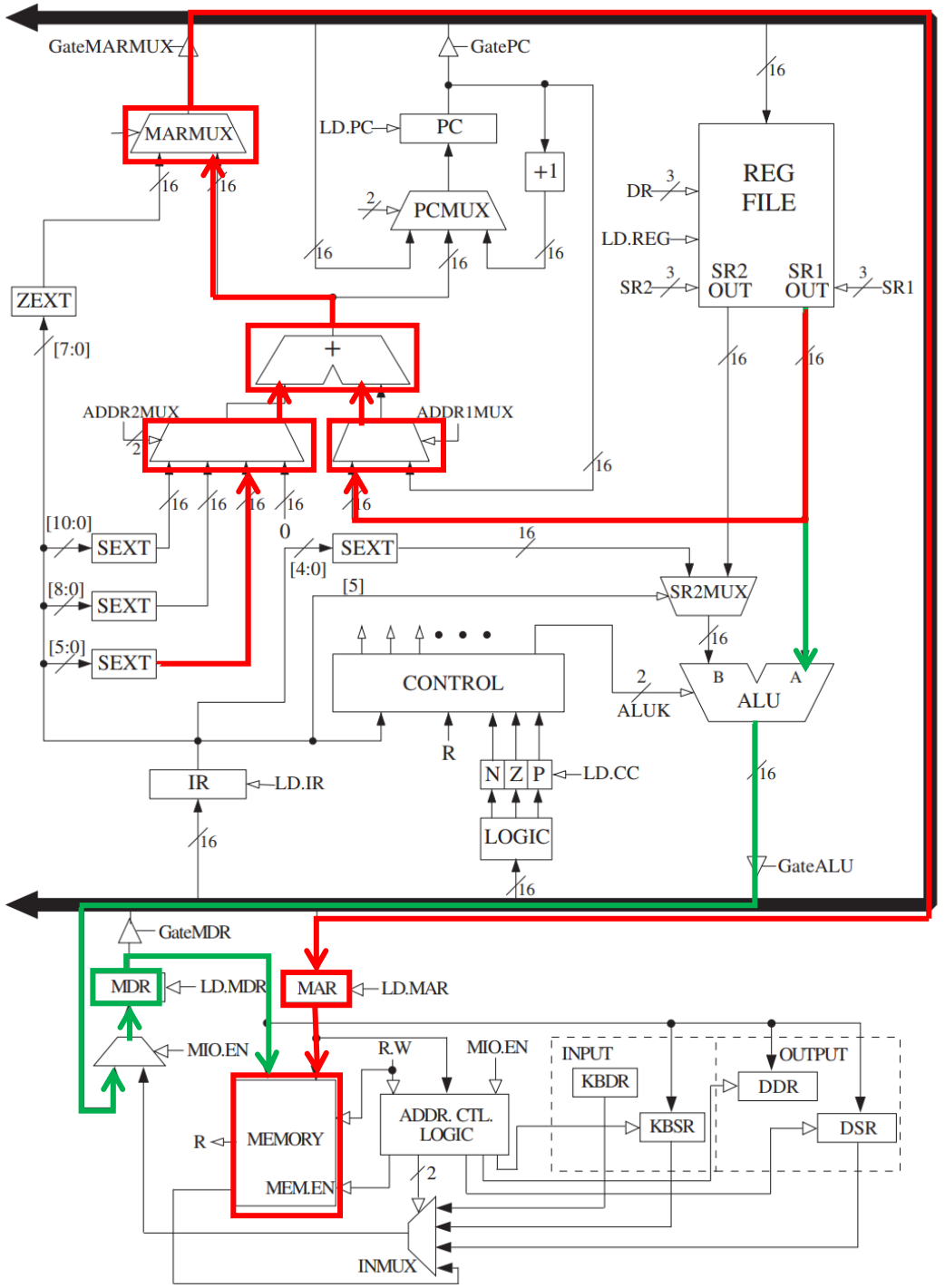
Assembler Format

STR SR, BaseR, offset6

Encoding



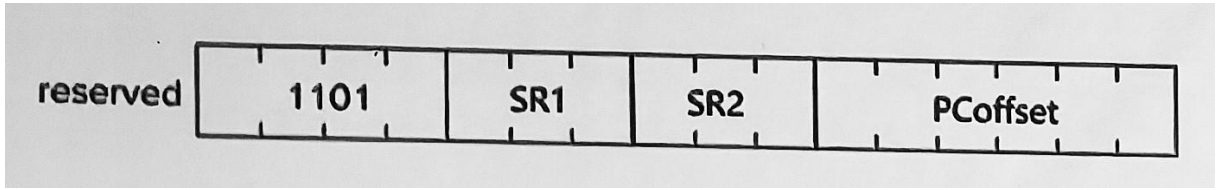
- ADDR1MUX: BaseR
- ADDR2MUX: offset6
- MARMUX: ADDER
- SR1MUX: 8.6, 11.9
- ALUK: PASSA



例题2

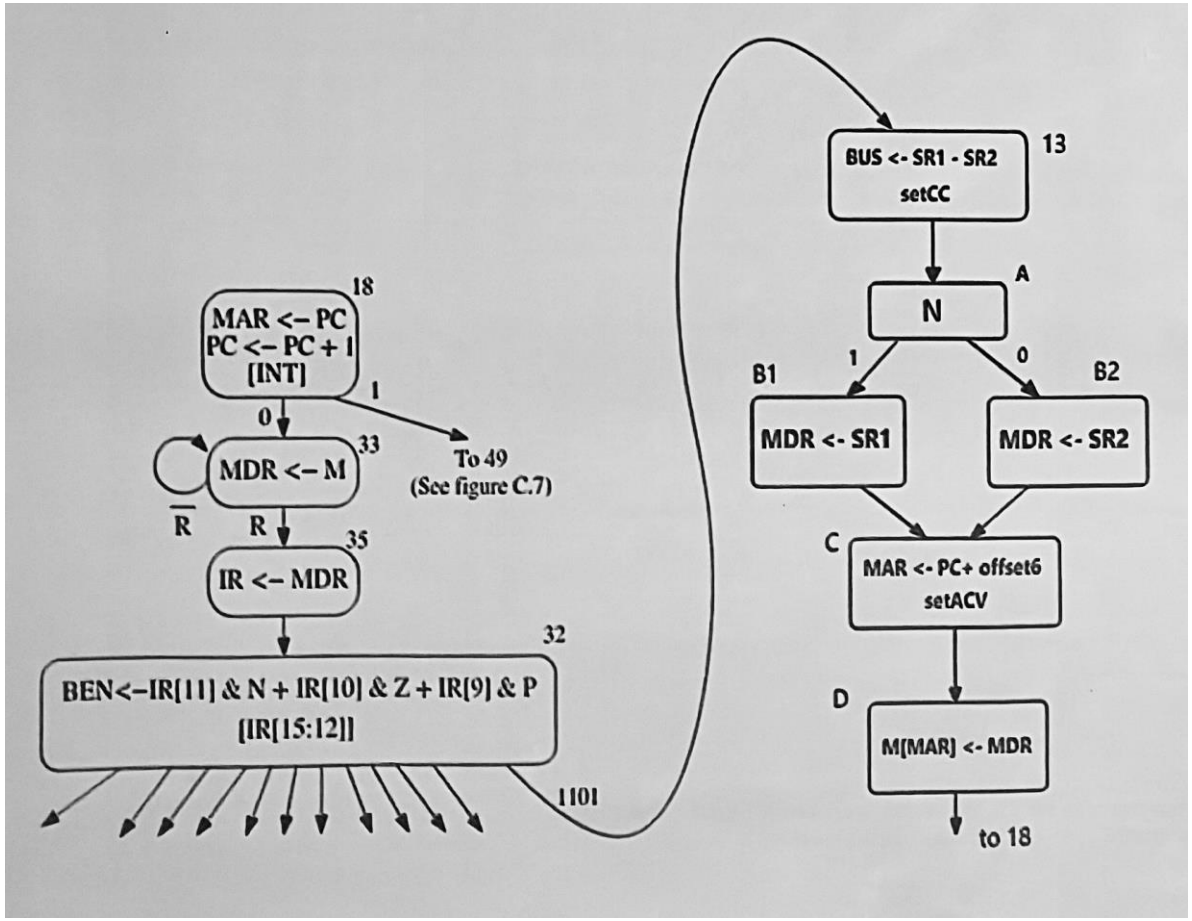
我们用闲置的指令码1101来实现一种新操作。

1. 为了实现我们需要设计一个SR2MUXnew（不同于SR2MUX）。画出你的设计
2. ALU也要做一些改动，新的ALU需要支持减法，使用位宽为3的控制信号（共有与运算，加法，非运算，减法，PASSA和PASSB几种运算）。并且SR2MUX的控制信号现在来自控制模块而不是来自IR[5]。为了控制数据通路执行指令，状态机需要增加图中所示的几个状态。
    1. 这条指令运行过程中要访问几次寄存器和ALU？
    2. 这条新指令的功能是什么？



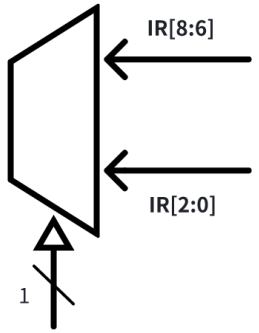
例题2

- 2.2很简单并且与具体数据通路无关，只需要观察状态图就能看出来。新指令的作用是把SR1和SR2中较小的一个存入内存，采用PC+偏移值寻址。即M[PC+offset] <= min(SR1, SR2)。
- 2.1也不需要看数据通路。观察状态图可以发现，状态13会读寄存器以及用ALU算一次减法，状态B会读寄存器以及用ALU传递数据。因此一共需要访问2次寄存器和2次ALU。

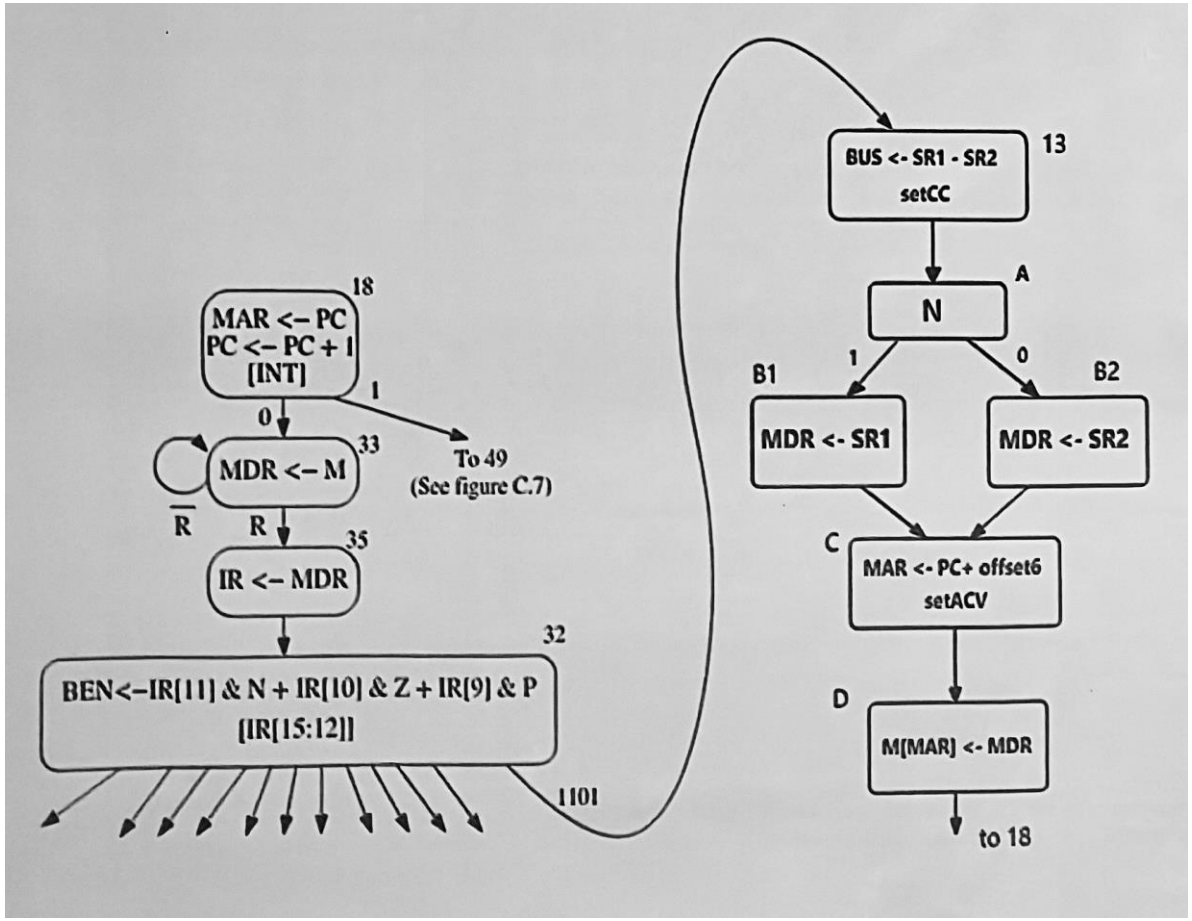


例题2

- **寄存器的SR2的**输入原本是不需要MUX**来**选择的，一定是**IR的2到0位。但是在**这道题新增的指令中，我们可以发现寄存器输出**SR2的来源**变成了**IR的8到6位**，这个**新增的SR2MUXnew就是用来控制**这个的。**由于**这个数据选择器只有两种需要选择**的**输入，它的控制信号只需要**1位。所以2.1答案如下**图。

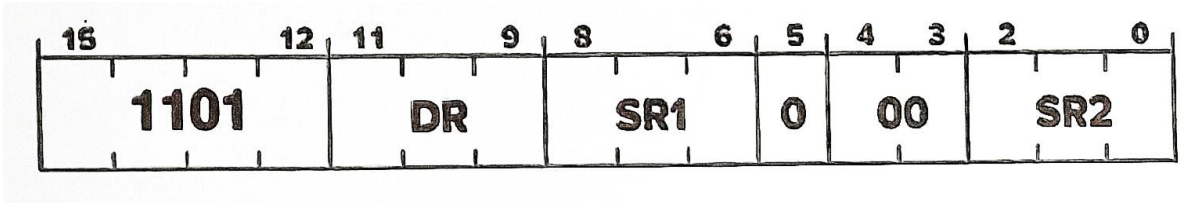


SR2MUXnew

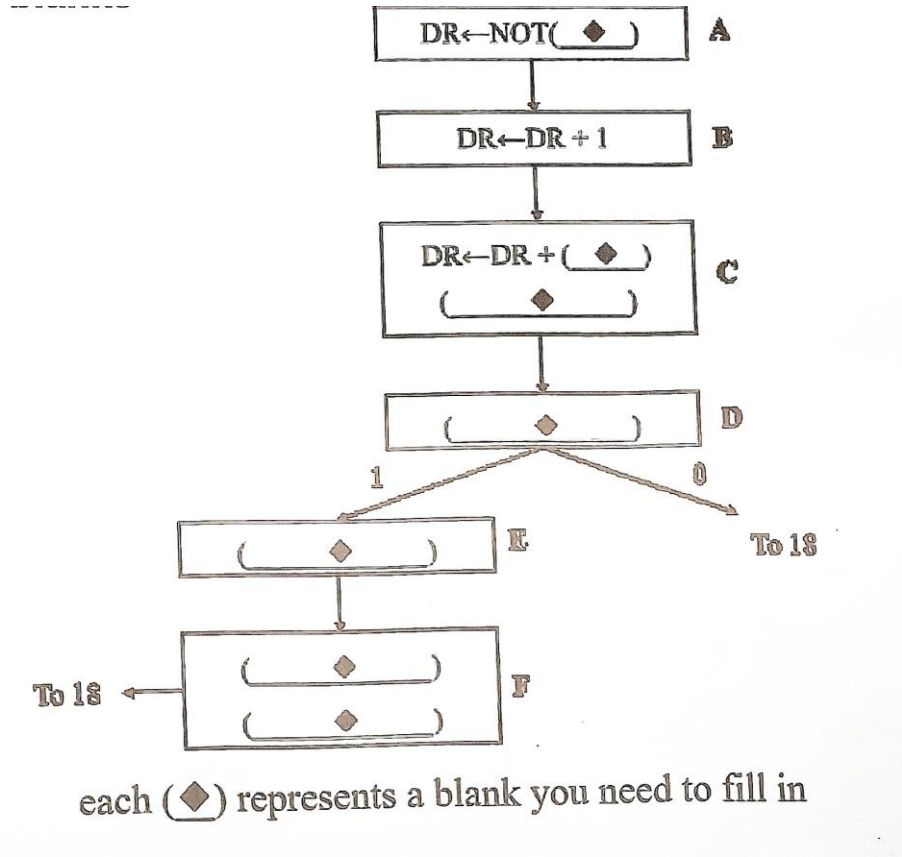


例题3

- 我们用闲置的指令码1101来实现一种新指令ABS如下图所示。

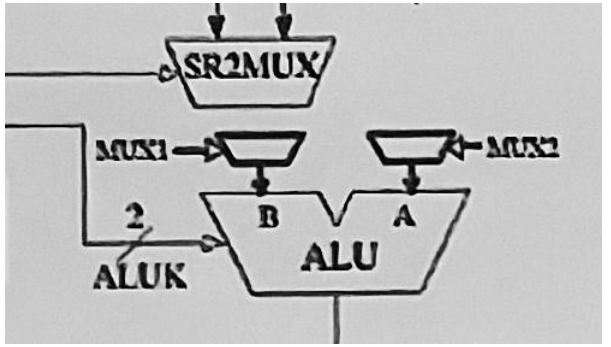


- 这个ABS指令计算SR1和SR2差的绝对值并存入DR，也就是DR = |SR1 - SR2|。条件码根据DR的值设置。
- 我们假设DR与SR1和SR2均不同。



例题3

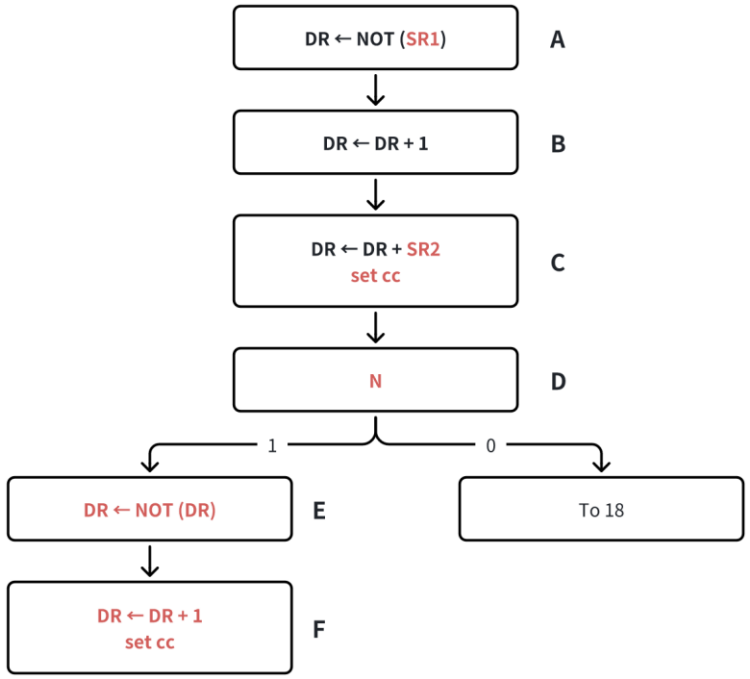
1. 假设数据通路已经实现好了，填空
2. 为了实现指令，数据通路需要加两个数据选择器。MUX1和MUX2是两个数据选择器的控制信号。请列出它们的值。
3. 假设下一条要执行的指令是ABS R2, R1, R0，同时R1的值是8，R0的值是6，P寄存器的值是1，填表。无所谓的填X。



State	LD.CC	GateALU	MARMUX	SR1MUX	ALUK	ADDR1MUX	ADDR2MUX	P	MUX1	MUX2
A										
C										
D										
E										

例题3

- 先看第一小题。首先我们观察到图中有一个条件判断。由于ABS指令计算出的结果是一个绝对值，我们可以猜到这个指令执行过程中需要先算出两个数的差，再根据这个差的正负来决定是否需要取反。所以D那个条件判断的依据就是差值的正负，若差的结果为负则需要取反一次。另外也可以猜出ABC三步是在算差值，EF两步是在取反。这个ABS指令会改变条件码，所以第一次算出差值之后需要更新条件码，以进行条件判断；算出最终结果后需要再更新一次条件码。
- ABC三步做的显然是经典的取反加一再相加，EF是取反加一。需要注意的是，我认为A只能填DR <= SR1。老师上课的时候说这里也可以填DR <= SR2，但是我们观察NOT指令的结构可以发现，NOT指令唯一的源操作数位置是在IR[8:6]，是SR1的位置。也就是说，ALU在执行NOT运算的时候，是输出NOT(SR1)，而不是NOT(SR2)。因此答案如图。



例题3

第二小题跟上面那个设计MUX类似。我们观察第一小题做出来的微指令，发现相比ALU能支持的一般的计算，我们还需要它能做以下的事情：

1. DR <= DR + 1、DR <= DR + SR2和DR <= NOT(DR)中，ALU的SR1需要取DR
2. DR <= DR + 1中，ALU的SR2需要取1

MUX2的输出是ALU的SR1，MUX1的输出是ALU的SR2。所以把上面这两个要求分别加到两个数据选择器上即可。答案是：

- MUX1:
- SR2MUX
- #1
- MUX2:
- SR1OUT
- DR

例题3

做到这里第三小题已经显得没什么难度了。

State	LD.CC	GateALU	MARMUX	SR1MUX	ALUK	ADDR1MUX	ADDR2MUX	P	MUX1	MUX2
A	NO	YES	X	8.6	NOT	X	X	1	X	SR1OUT
C	LOAD	YES	X	X	ADD	X	X	0	SR2MUX	DR
D	NO	NO	X	X	X	X	X	0	X	X
E	NO	YES	X	X	NOT	X	X	0	X	DR

I/O & Interrupt

Xuehan Zhang
University of Science and Technology of China

Important

- Figure C.7 状态 45 有误，改为
  - Saved_USP <– SP
  - SP <- Saved_SSP
- 所谓 Stack Pointer (SP)，在 LC-3 的实现中就是 R6。

Chapter 9

- Processor status register (PSR): P315, Figure 9.1 处理器状态寄存器
  - PSR[15]: Privilege bit. PSR[15] = 0 代表处理器处于特权模式。
  - PSR[14] (P351, Section 9.5.2): Interrupt enable bit. PSR[14] = 1 表示接受中断。
  - PSR[10:8]: Priority bits. 可表示 PL0 至 PL7 的优先级，数字越大越优先。
  - PSR[2:0]: Condition codes. PSR[2:0] = NZP, respectively. 分别是 NZP。
- Three exceptions: P717, Section 7.3 三个异常:
  - Privilege Mode Exception: 用户模式下执行 RTI 触发
  - Illegal Opcode Exception: 使用未定义的 1101 指令码触发
  - Access Control Violation (ACV) Exception: 用户模式下访问[x3000, xFE00)之外
- 内存地址分配 P316 Figure 9.2
  - Supervisor Stack Pointer, 特权栈指针，使用 R6。TRAP 和中断只会存系统栈。
  - User Stack Pointer, 用户栈指针，同样使用 R6，但 R6 也可以用于其他用途。
  - 从数据通路可以看出，SSP 必须用 R6，但 USP 也可用别的。

Chapter 9

- 内存映射 I/O (Memory-Mapped I/O): P654, Table A.1
  - KBSR[15]: Ready bit, KBSR[15]=1 表示有键盘输入要读，同时拒绝之后的输入
  - KBDR[7:0]: 数据位，用于读（作为输入）
  - KBSR[14]: interrupt enable (IE) bit, KBSR[14]=1 表示设备可以请求中断
  - DSR[15]: Ready bit, DDR[15]=0 表示屏幕还在处理，同时拒绝后续写入
  - DDR[7:0]: 数据位，用于写（作为输出）
  - DSR[14]: interrupt enable (IE) bit, DSR[14]=1 表示设备可以请求中断
- TRAP 与 Interrupt
  - x0000 至 x00FF: Trap Vector Table: 每行对应一个服务，其内容是服务在内存中的起始地址。一共可以有 256 个服务。
  - x0100 至 x01FF: Interrupt Vector Table: 同样每行对应一个服务的地址
  - TRAP 和 Interrupt 都需要保存程序状态 (SP, PSR 和 PC)，并都通过 RTI 返回
  - TRAP 是程序执行的指令（可控），Interrupt 是外界产生的打断（不可控）

一起努力

打造国产基础软硬件体系！

2025/1/3