「计算机系统概论」TA Session 3

2022-2023 短学期计算系统概论课程

By @HobbitQia

More about Session 2

Address Space

Address Space 地址空间是指内存中，最多能表示的地址个数。如果我们用 n 位表示地址，那么地址空间就是 2^n。不一定所有地址空间里的地址都会用做内存，存储数据，还有的地址会用来映射 I/O 设备。

Address Space

Address Space 地址空间是指内存中，最多能表示的地址个数。如果我们用 n 位表示地址，那么地址空间就是 2^n。不一定所有地址空间里的地址都会用做内存，存储数据，还有的地址会用来映射 I/O 设备。

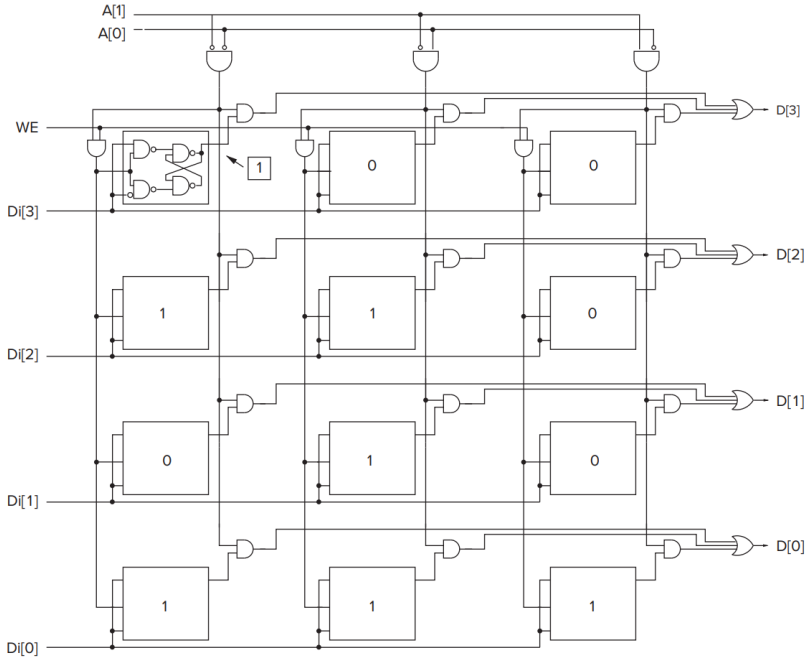


Figure 3.45 Diagram for Exercise 3.40.

因此作业这道题的地址空间应该是 4，而不是 3。

~~（做错子记得及时改）~~

Chapter 4: The von Neumann Model

Basic Components of Computer

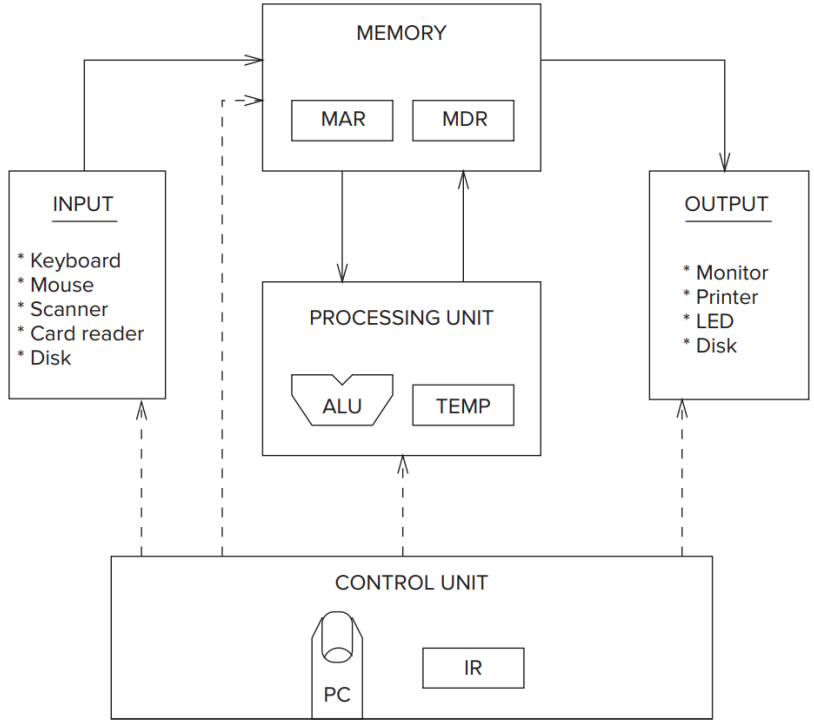


Figure 4.1 The von Neumann model, overall block diagram.

5 parts: Memory, Processing Unit, Control Unit, Input, Output

其中我们常说的 CPU，就包括上面的处理单元和控制单元。

Memory

内存是用来存储信息的，内存的每个位置都有唯一的地址，我们可以把数据存放在内存中。***e.g.*** 程序代码

在 LC-3 中，地址是 16 bits 的，因此地址空间为 2^{16}。寻址能力为 2 bytes，即 16 bits。

为了访问内存中的数据，我们有两个寄存器 MAR（Memory's Address Register）和 MDR（Memory's Data Register）

- 从内存中读数据
  - 将要读的地址放入 MAR
  - 若干个时钟周期后，MAR 地址上的数据会被送到 MDR 中
- 从内存中写数据
  - 将要写的地址放入 MAR，要写的数据放入 MDR
  - 若干个时钟周期后，内存中 MAR 地址对应的数据被修改为 MDR

Processing Unit

处理单元是计算机中实际处理信息的单元。处理单元可以很复杂，但在 LC-3 中我们只考虑最简单的情况，即处理单元只有两个部分：ALU (Arithmetic Logic Unit) 和 Register Files 寄存器堆。

- ALU 是用来执行算术运算和逻辑运算的。 ***e.g.*** ADD, AND
  ALU 对固定大小的数据进行操作，这样的数据称为 **word** 字，这个大小称为 **word length** 字长。
  LC-3 的字长为 16 bits。
- 寄存器堆里放了若干个寄存器，寄存器的作用是临时存储值。(每次都从内存访问数据非常慢，因此我们常用寄存器来临时存储数据以便运算)
  在 LC-3 中，我们有 8 个 General Purpose Registers 通用寄存器 $(R0, R1, \ldots R7)$

Input and Output

我们可以通过外部设备向计算机输入数据（键盘、鼠标），计算机也可以向外部设备输出数据（显示屏）。I/O 设备与计算机通信的方式会在 Chapter 9 中讲到。

Control Unit

控制单元会控制计算机的运行（当前执行到哪条指令，指令进行到了哪个阶段），控制单元主要包括 3 个部分

- **program counter (PC)** 程序计数器
  PC 存放的是下一条指令的地址。
- **instruction register (IR)** 指令寄存器
  IR 存放的是当前正在执行的指令。
- FSM
  有限状态机根据 IR，决定控制信号并把控制信号传到其他部件。

e.g. 如果当前指令是 ADD 指令，那么 FSM 需要告诉 ALU 执行加法运算。

Instruction

Instruction 指令是计算机中最小操作单位。指令包括两个部分：

- opcode 操作码：规定指令的作用。 ***e.g.*** 在 LC-3 中，规定 0001 的 opcode 表示 ADD 运算。
- operand 操作数：指令作用在谁上。 ***e.g.*** 在 LC-3 中，ADD 运算可以是两个寄存器的值相加，也可以是一个寄存器的值和一个立即数相加。

在 LC-3 中，主要有三种指令：

- operation 操作指令，进行算术运算和逻辑运算。
- data movement 与内存交互，读写内存中的数据。
- control 控制指令，可能会改变原本指令的执行顺序。

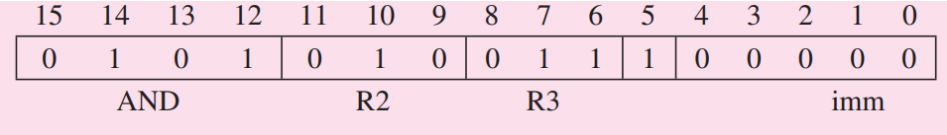
ADD

DR ← SR1 ADD SR2

DR ← SR1 ADD imm5

- bit[15:12]: opcode (0001 表示 ADD 的操作码)
- bit[11:9]: DR (Destination Register 目的寄存器)
- bit[8:6]: SR1 (Source Register 1 源寄存器)
- bit[5]:
  - bit[5]=0, 说明 SR1 + SR2 → DR, 此时 bit[4:3] 默认为 0, bit[3:0] 表示 SR2。
  - bit[5]=1, 说明 SR1 + imm5 → DR, 此时 bit[4:0] 会被符号扩充为 16 bits 并作为第二个加数。

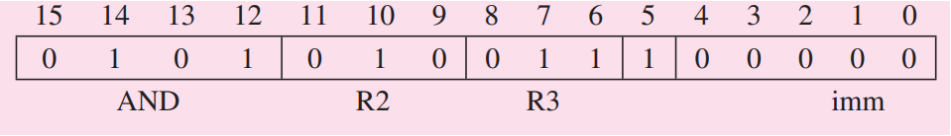
AND




```
DR ← SR1 AND SR2
DR ← SR1 AND imm5
```

- AND 的格式和 ADD 相同（除了 AND 的 opcode 是 0101）
- 我们可以设 bit[5]=1, bit[4:0]=00000, SR1=DR。这样这条指令就变为了 reg AND 0 → reg。相当于把寄存器初始化为 0。

AND

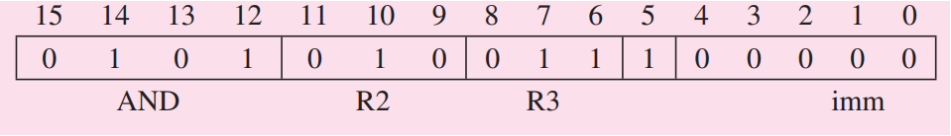



```
DR ← SR1 AND SR2
DR ← SR1 AND imm5
```

- AND 的格式和 ADD 相同（除了 AND 的 opcode 是 0101）
- 我们可以设 bit[5]=1, bit[4:0]=00000, SR1=DR。这样这条指令就变为了 reg AND 0 → reg。相当于把寄存器初始化为 0。

Question: ADD, AND 如果采用寄存器与立即数运算的方式，立即数的范围是多少？

AND



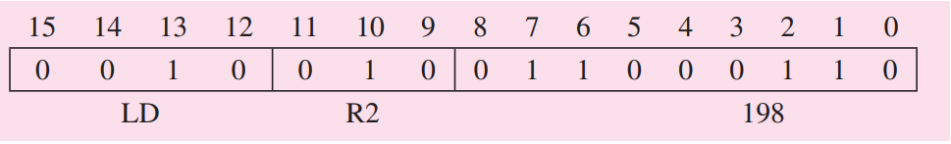

```
DR ← SR1 AND SR2
DR ← SR1 AND imm5
```

- AND 的格式和 ADD 相同（除了 AND 的 opcode 是 0101）
- 我们可以设 bit[5]=1, bit[4:0]=00000, SR1=DR。这样这条指令就变为了 reg AND 0 → reg。相当于把寄存器初始化为 0。

Question: ADD, AND 如果采用寄存器与立即数运算的方式，立即数的范围是多少？

$$[-2^4, 2^4 - 1]$$

LD




```
DR ← M[PC+offset]
```

LD 表示 load，即到内存地址里取出数据并加载到寄存器中。

LD 采用 **PC+offset** 的方式寻址。

- bit[15:12]: opcode（0010 表示 LD 的操作码）
- bit[11:9]: DR 我们从内存中读出的数据就放在 DR 目的寄存器中。
- bit[8:0]: offset 偏移量，同样需要被符号扩充到 16 bits。这样 PC+offset 就是我们要读取的内存地址。

The Instruction Cycle

完整执行一条指令的过程称为 **instruction cycle** 指令周期。在 LC-3 中指令周期包括这 6 个阶段：

fetch 取指，decode 译码，evaluate address 计算地址，fetch operands 取操作数，execute 执行计算，store result 写回结果。

指令周期不等于时钟周期（往往需要多个时钟周期），也不是所有的指令都需要经过 6 个阶段。

The Instruction Cycle (Cont.)

The Instruction Cycle (Cont.)

- Fetch
  根据 PC 取出下一条指令，并将指令内容放到 IR 内。具体流程如下：
  - PC 通过总线送到 MAR 中，同时执行 PC ← PC+1 (1 clock cycle)
  - 内存中取出 MAR 地址对应的数据，放到 MDR 中。(≥1 clock cycle)
  - MDR 的结果送到 IR 寄存器中去。(1 clock cycle)

The Instruction Cycle (Cont.)

- Fetch
  根据 PC 取出下一条指令，并将指令内容放到 IR 内。具体流程如下：
  - PC 通过总线送到 MAR 中，同时执行 PC ← PC+1 (1 clock cycle)
  - 内存中取出 MAR 地址对应的数据，放到 MDR 中。(≥1 clock cycle)
  - MDR 的结果送到 IR 寄存器中去。(1 clock cycle)
- Decode
  FSM 内，根据 IR 内存储的指令内容进行解码。(通过 opcode 决定指令类型)

The Instruction Cycle (Cont.)

- Fetch
  根据 PC 取出下一条指令，并将指令内容放到 IR 内。具体流程如下：
  - PC 通过总线送到 MAR 中，同时执行 PC ← PC+1 (1 clock cycle)
  - 内存中取出 MAR 地址对应的数据，放到 MDR 中。(≥1 clock cycle)
  - MDR 的结果送到 IR 寄存器中去。(1 clock cycle)
- Decode
  FSM 内，根据 IR 内存储的指令内容进行解码。(通过 opcode 决定指令类型)
- Evaluate Address
  计算我们要访问内存的地址。(这个阶段只会在我们需要访问内存时才出现) ***e.g.*** LD

The Instruction Cycle (Cont.)

The Instruction Cycle (Cont.)

- Fetch Operands
  取出源操作数（包括内存和寄存器堆中的数据）***e.g.*** LD 在这个阶段会从内存中取出数据，ADD 在这个阶段会从寄存器堆中取出数据。

The Instruction Cycle (Cont.)

- Fetch Operands
  取出源操作数（包括内存和寄存器堆中的数据）***e.g.*** LD 在这个阶段会从内存中取出数据，ADD 在这个阶段会从寄存器堆中取出数据。
- Execute
  执行计算。 ***e.g.*** ADD 指令在这个阶段我们会加上源操作数。

The Instruction Cycle (Cont.)

- Fetch Operands
  取出源操作数（包括内存和寄存器堆中的数据）***e.g.*** LD 在这个阶段会从内存中取出数据，ADD 在这个阶段会从寄存器堆中取出数据。
- Execute
  执行计算。 ***e.g.*** ADD 指令在这个阶段我们会加上源操作数。
- Store Result
  把结果写回到目的地。 ***e.g.*** 比如 ADD 会把计算的结果写回到寄存器堆中，ST（store）会把数据写回到内存中。

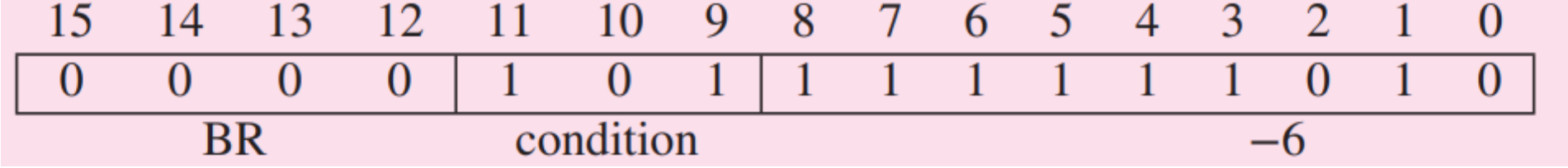
The Instruction Cycle (Cont.)

- Fetch Operands
  取出源操作数（包括内存和寄存器堆中的数据）***e.g.*** LD 在这个阶段会从内存中取出数据，ADD 在这个阶段会从寄存器堆中取出数据。
- Execute
  执行计算。 ***e.g.*** ADD 指令在这个阶段我们会加上源操作数。
- Store Result
  把结果写回到目的地。 ***e.g.*** 比如 ADD 会把计算的结果写回到寄存器堆中，ST（store）会把数据写回到内存中。

在 LC-3 中，ADD 指令可以在一个时钟周期内，完成 fetch operands，execute 和 store result 三个阶段。

Changing the Sequence of Execution -- BR

我们可以用 BR（Branch）改变执行的顺序。



- bit[15:12]: opcode（0000 表示 BR 的操作码）
- bit[11:9]: 跳转的条件。（bit[11:9] 分别对应 NZP，具体含义见下一页）
- bit[8:0]: offset 偏移量，和 LD 一样也需要符号扩充到 16 bits，这样 PC+offset 就是我们要跳转到的地址。

Condition Code

CC (Condition Codes) 是三个 1-bit 的状态（N Z P）N 表示 Negative，Z 表示 Zero，P 表示 Positive。部分指令执行后，会根据执行的结果更新 NZP 三个条件码。 ***e.g.*** 假设我们 ADD 的结果是 3>0，那么 P 就被设为 1，Z 和 N 就被设为 0。
不是所有指令都会修改 CC，见书上 148 页，其中右上角带有 + 的指令是会影响 CC 的，其他不会影响。

而我们 BR 是否跳转就根据 CC 来决定。如果 bit[11:9] 中有一位为 1，同时对应的条件码也为 1，那么我们就执行跳转。如果 bit[11:9] 均为 0，或者 bit[11:9] 中为 1 对应的条件码均为 0，就不执行跳转。

如果执行跳转，就跳到 PC+offset。

Overview of the LC-3 Datapath

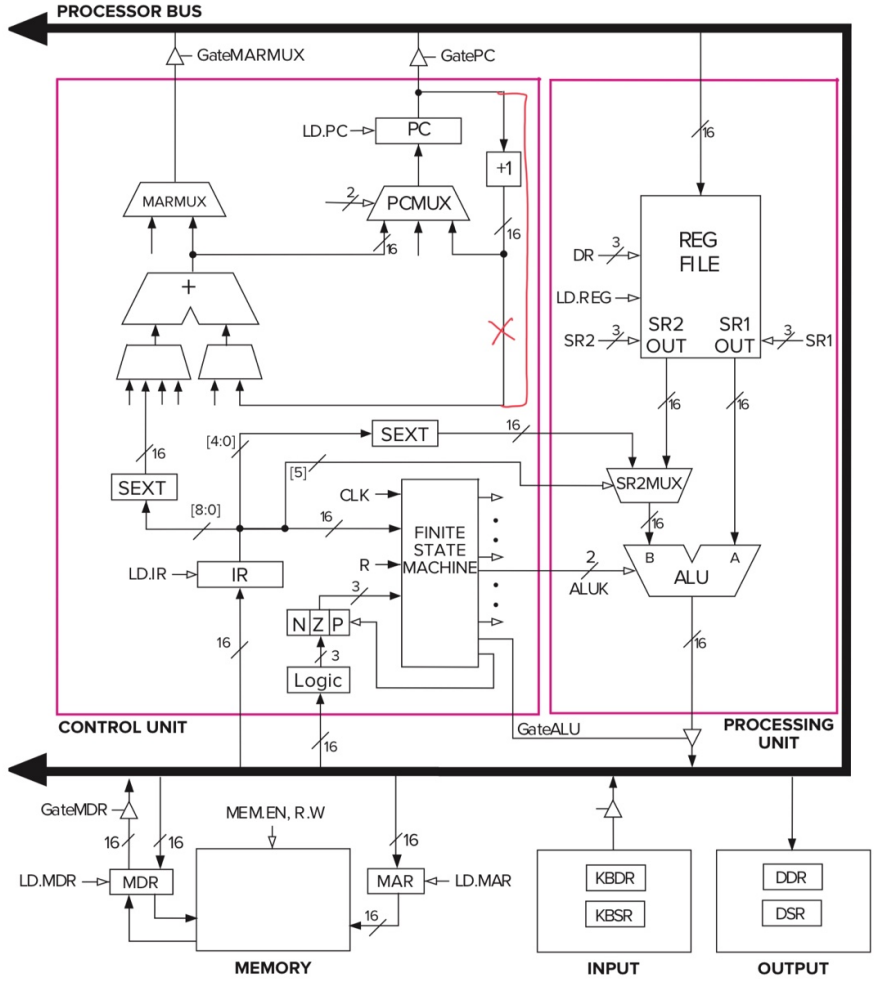


Figure 4.3 The LC-3 as an example of the von Neumann model.

谢谢大家

Question?