NOTICE: Past Exams

Past exams are provided as-is for reference only.

- The content of this exam reflects course content and learning outcomes as they were at the time this exam was administered.
- This course's content, structure, and/or focus may have changed since this past exam was administered.
- As such, **the content, structure and types of questions** contained herein **may differ from those in your final exam**.
- If you plan on using this as a study resource, be sure to do so in conjuction with the current course's syllabus and resources to ensure that your study covers the correct content.

这张图片的内容是一个考试通知说明，标题为 **“NOTICE: Past Exams”**（“注意：往年试题”）。下面我为你详细解释每一条内容的含义，并在最后附上一个口语化总结。

🔹 逐条详细解释

1. **“Past exams are provided as-is for reference only.”**
   往年考试题仅供参考使用，不会做任何修改或保证内容的准确性。
2. **“The content of this exam reflects course content and learning outcomes as they were at the time this exam was administered.”**
   这份考试题反映的是考试当年课程的内容和学习目标，也就是说，它展示的是当时老师讲课的重点和考察方向。
3. **“This course’s content, structure, and/or focus may have changed since this past exam was administered.”**
   课程的内容、结构或重点可能在之后发生了变化，也就是说现在的课程可能和当时不一样了。
4. **“As such, the content, structure and types of questions contained herein may differ from those in your final exam.”**
   因此，这份往年试题里的题目内容、结构以及题型，可能与你现在要参加的期末考试不一样。
   ⚠️ 重点提醒：不能完全依赖往年试题复习。
5. **“If you plan on using this as a study resource, be sure to do so in conjunction with the current course’s syllabus and resources to ensure that your study covers the correct content.”**
   如果你打算用这份往年试题来复习，一定要结合现在课程的**教学大纲（syllabus）**和**教材、讲义等资源**，确保复习内容是正确的，不要复习错方向。

🔹 总结归纳

这整张图的意思是提醒学生：
往年试题只是一个**学习参考**，不能代表现在考试的内容。课程可能已经改动过，所以题目类型、结构和重点可能完全不同。如果要用旧试题复习，必须同时对照最新的课程大纲与教材，否则容易复习偏了。

🗣️ [口语化表达]

这张图其实就是说：“嘿，别太信旧试题啦！它只是以前的版本，老师和课程可能早就改了。现在考试的题型、内容都可能不一样。要是你想用它复习，就一定得对照现在的课程大纲和资料，不然复习错方向就糟了。”

第2页

THE UNIVERSITY of ADELAIDE

Primary Examination, Semester 1, 2017

Computer Systems
COMP SCI 2000, 7081

Official Reading Time:	10 mins
Writing Time:	120 mins
Total Duration:	130 mins

Questions	**Time**	**Marks**
Answer all 12 questions	120 mins	120 marks 120 Total

Instructions for Candidates

- This is a Closed-book examination.
- Begin each answer on a new page.
- Examination material must not be removed from the examination room.

Materials

- Foreign Language Dictionaries are Permitted

DO NOT COMMENCE WRITING UNTIL INSTRUCTED TO DO SO

这张图是 **澳大利亚阿德莱德大学（The University of Adelaide）** 的一份考试说明页，是2017年第一学期的**期末考试（Primary Examination, Semester 1, 2017）**，课程名称是 **Computer Systems（计算机系统）**，课程代码 **COMP SCI 2000, 7081**。下面我来为你逐条详细解释。

🔹 详细解释部分

🕑 考试时间安排

- **Official Reading Time: 10 mins**
  正式阅读时间为10分钟，也就是说考试开始后前10分钟你只能看题，不能动笔答题。
- **Writing Time: 120 mins**
  正式答题时间为120分钟，也就是2个小时。
- **Total Duration: 130 mins**
  整个考试总时长为130分钟（包括10分钟的阅读时间）。

📝 考试内容说明

- **Questions: Answer all 12 questions**
  你必须回答全部12道题，没有选择题或跳题的选项。
- **Time: 120 mins, Marks: 120 marks (Total)**
  总分是120分，时间是120分钟，也就是说平均每题大约10分钟、10分。

📋 考生须知（Instructions for Candidates）

1. **This is a Closed-book examination.**
   这是一场**闭卷考试**，不能带书、笔记或电子资料进入考场。
2. **Begin each answer on a new page.**
   每个题目的答案要从新的一页开始写，不能混在一页上。
   👉 这是为了方便阅卷老师批改时能清楚分题、评分。
3. **Examination material must not be removed from the examination room.**
   考试材料不得带出考场。
   包括试卷、草稿纸、答题纸等，全部都必须留在教室内。

📚允许携带物品（Materials）

- **Foreign Language Dictionaries are Permitted**
  允许携带外语词典（例如英汉词典）。
  ⚠️ 注意：只能是纸质词典，不能带电子设备。

⚠️特别提醒

- **DO NOT COMMENCE WRITING UNTIL INSTRUCTED TO DO SO**
  在监考老师宣布可以开始之前，**不得动笔写字！**
  如果提前书写，有可能被认为违规。

🗣️[口语化表达]

这页其实就是说："这场考试一共130分钟，前10分钟只能看题不能写，之后有两个小时答12道题，总共120分。是闭卷考试，不能带书，答题时每道题都要从新的一页写，考试材料不能带出教室。你可以带纸质词典，但不能带电子设备。最重要的是——别在老师说开始之前动笔，不然会被警告哦！"

Basic Gates and Boolean Logic

Question 1

(a) Consider the expression:

$$x \neq y$$

which is *true* if the boolean values x and y are not equal. The expression is *false* otherwise. Answer the following two questions:

i. Draw the truth table for the $\neq$ operator in terms of x and y.

[3 marks]

ii. Draw an implementaton of the $\neq$ operator solely in terms of And, Or and Not gates.

[6 marks]

[Total for Question 1: 9 marks]

Boolean Arithmetic and ALU design

Question 2

For the following questions you may find the information in Figure 2 useful.

(a) Look at the Hack ALU truth-table shown in Figure 2 in the Appendix. Note that it is possible to design the functionality of the ALU a few operations at a time. For this question, we will consider the last line of the table, which describes the Or operation. Answer the following:

i. In the ALU, the Or operation is implemented using an And chip and some other chips and wires. Give the logical expression for the Or operation implemented by the ALU. Your answer must include an And operation.

[4 marks]

ii. Using a truth table, show that the expression you gave in part i above is equivalent to the Or operation.

[3 marks]

iii. Draw an implementation of a 16-bit Or chip that uses a 16-bit And chip.

[3 marks]

[Total for Question 2: 10 marks]

下面是这页试题（第2页）的逐题中文讲解与做法示范。

基本门电路与布尔逻辑（Basic Gates and Boolean Logic）

Question 1（共9分）

题干给出表达式：**x ≠ y**
当布尔值 x 与 y 不相等时为真，否则为假。这其实就是 **异或 XOR**。

(a) i. 画出“≠”的真值表（3分）

x	y	x ≠ y
0	0	0
0	1	1
1	0	1
1	1	0

说明：只有两者不同（01 或 10）时输出 1，其余为 0。

(a) ii. 只用 And、Or、Not 门实现“≠”（6分）

标准实现（等价于 XOR）：

$$x \neq y \;=\; (x \wedge \neg y) \;\vee\; (\neg x \wedge y) \tag{1}$$

实现思路：

1. 先各自取反得到 **¬x**、**¬y**（用 Not 门）。
2. 计算 **x ∧ ¬y** 与 **¬x ∧ y**（用 And 门）。

3. 再对这两个结果做 **Or**（用 Or 门）得到最终输出。
   文本电路示意：

```
x ──┐         ┌─ AND ──┐
    ├─ NOT y ─┤        │
y ──┘         └────────┤
    ├─ OR ── out = x≠y
x ── NOT ─┐            │
    ├─ AND ─────────┘
y ──────────┘
```

布尔算术与 ALU 设计（Boolean Arithmetic and ALU design）

Question 2（共10分）

提示会参考附录中 Hack ALU 的真值表（图2），这里关注 **Or 运算** 那一行。

(a) i. 用 And 芯片和一些连线实现 Or 的逻辑表达式（4分）

根据德摩根律**x ∨ y ≡ ¬(¬x ∧ ¬y)**，Or 可用 And 与 Not 表示：

$$\mathrm{Or}(x, y) \;=\; \neg\big(\,\neg x \;\wedge\; \neg y\,\big) \tag{2}$$

这里显式包含了 **And** 运算：先把 x、y 各自取反做一次 And，再把结果取反。

> **用"开关"直觉理解**
>
> 想象两个开关 x、y 控制一盏灯：

场景	逻辑	德摩根说法
灯**不亮**当且仅当**两个开关都合上**	¬(x ∧ y)	等价于**至少有一个开关断开** ¬x ∨ ¬y

场景	逻辑	德摩根说法
灯**不亮**当且仅当**至少一个开关合上**	¬(x ∨ y)	等价于**两个开关都断开** ¬x ∧ ¬y

(a) ii. 用真值表证明上式与 Or 等价（3分）

x	y	¬x	¬y	(¬x ∧ ¬y)	¬(¬x ∧ ¬y)	x ∨ y
0	0	1	1	1	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	0	1	1

观察最后两列完全一致，因此 **¬(¬x ∧ ¬y) ≡ x ∨ y**。

(a) iii. 画出一个使用 16 位 And 芯片实现的 16 位 Or 芯片（3分）

按位实现思路（对16个比特并行）：

$$\mathrm{Or16}(A,B) \;=\; \mathrm{Not16}\Big(\;\mathrm{And16}\big(\mathrm{Not16}(A),\;\mathrm{Not16}(B)\big)\;\Big) \tag{3}$$

文本连线示意（每个方块表示对16位总线的并行操作）：

```
A ── Not16 ┐
           ├┐
B ── Not16 ┘ |
             ├ And16 ── Not16 ── OUT   // OUT 即 Or16(A,B)
             ┘
```

```
          ┌---------┐
A[0..15]──┤         |
          | 16-bit  | Y[0..15]
B[0..15]──┤   OR    ├--->
          | (built  |
          |from DM) |
          └---------┘
```

实现要点：

- 输入 A、B 各先过 **Not16**（逐位取反）。
- 两路取反后的 16 位总线进入 **And16**。
- And16 的输出再接 **Not16**，得到 16 位 Or 的结果。
- 若需要画门级图，只需标注三块芯片：Not16、And16、Not16，并画清 16 位总线。

小提示（答题策略）

- 第1题把 “≠” 识别为 XOR 是关键；若不会画电路，写出等价表达式即可得分。
- 第2题核心是用 **德摩根律** 把 Or 改写为 Not-And-Not，并能给出逐位的 16 位实现。

🗣️ **[口语化表达]**

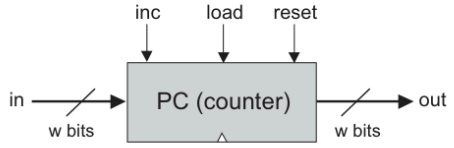
这页主要考两件事：第一题就是让你把 “x≠y” 当成 XOR 来做，先填个真值表（只有不同才为1），然后用基本门搭出来：`(x∧¬y) ∨ (¬x∧y)`。第二题围绕 ALU 的 Or 运算，要求你用 And 来实现 Or。套路就是德摩根律：`Or(x,y) = ¬(¬x ∧ ¬y)`，再用真值表验证一下就行。最后把它扩展成 16 位：先把 A、B 各自 Not16，再 And16，最后再 Not16，一进一出全是16位总线。整体就这么个思路，按这个写基本都能拿满分。

Primary Examination, Semester 1, 2017

Sequential Logic

Question 3

(a) Look at the following diagram and text description for a program counter (PC) from figure 3.5 of the textbook:




```
Chip name: PC  // 16-bit counter
Inputs:    in[16], inc, load, reset
Outputs:   out[16]
Function:  If reset(t-1) then out(t)=0
              else if load(t-1) then out(t)=in(t-1)
                   else if inc(t-1) then out(t)=out(t-1)+1
                        else out(t)=out(t-1)
Comment:   "=" is 16-bit assignment.
           "+" is 16-bit arithmetic addition.
```

Draw an implementation of the PC chip *without the reset wire*. That is, draw an implementation of the PC that can handle signals from the *inc* and the *load* wires but *doesn't* provide a reset function.

Note, you do not have to express your solution in terms of primitive gates such as Nand. You can use large scale chips such as Inc16.

[10 marks]

[Total for Question 3: 10 marks]

下面是本页（第3页）题目的中文解读与解题步骤。

时序逻辑（Sequential Logic）

Question 3（共10分）

题目给了教材图 3.5 的 **程序计数器 PC（16 位计数器）** 定义与时序语义（右侧灰框）：

- **Chip name**: `PC`（16-bit counter）
- **Inputs**: `in[16]`，`inc`，`load`，`reset`
- **Outputs**: `out[16]`
- **时序功能**（按上个周期的控制信号决定本周期输出）
- 若 `reset(t-1)=1`，则 `out(t)=0`
- 否则若 `load(t-1)=1`，则 `out(t)=in(t-1)`
- 否则若 `inc(t-1)=1`，则 `out(t)=out(t-1)+1`
- 否则 `out(t)=out(t-1)`
- 说明：`=` 是 16 位赋值，`+` 是 16 位算术加法。

> **要求**：画出一个 **没有 reset 线** 的 PC（即去掉复位功能），但仍需正确响应 `inc` 和 `load`。可以直接使用大规模芯片（如 `Inc16`、`Register16`、`Mux16`），不用原始门级（如 `Nand`）。

解题思路与实现（给可直接画图的文本连接说明）

去掉 `reset` 后，功能简化为：

$$\text{next} = \begin{cases} \text{in} & \text{if } load = 1 \\ out + 1 & \text{else if } inc = 1 \\ out & \text{else} \end{cases} \tag{4}$$

把“分支”改写成两级多路选择即可：

1. **寄存器**
   使用 `Register16` 保存 PC 值：

- `Register16.in = next`
- `Register16.out = out`（同时作为反馈参与下一拍的计算）
- 时钟沿到来时把 `next` 装入，输出在下一个时刻生效。

2. **加一器**
   用 `Inc16` 计算 `out+1`：

- `Inc16.in = out`
- `Inc16.out = out_plus_1`

3. **第一级选择（处理 inc）**
   用 `Mux16` 在 “保持原值” 与 “加一” 间选择：

- `Mux16_inc.a = out`
- `Mux16_inc.b = out_plus_1`
- `Mux16_inc.sel = inc`
- `Mux16_inc.out = inc_result`

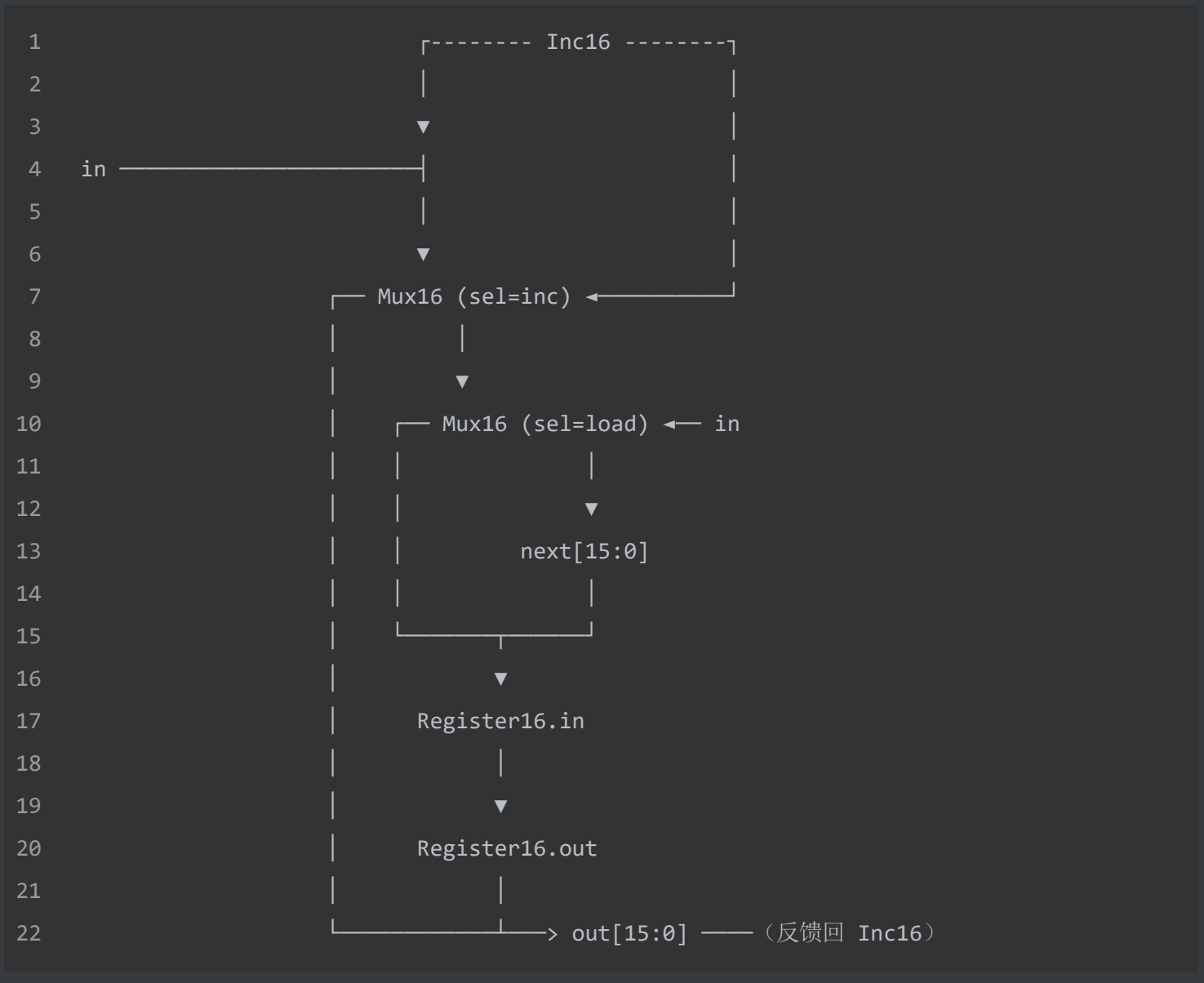
> 语义：`inc=1` 时选择 `out+1`，否则保持 `out`。

4. **第二级选择（处理 load 优先级高于 inc）**
   再用一个 `Mux16` 在 “载入 in” 与 “上一步结果” 间选择：

- `Mux16_load.a = inc_result`
- `Mux16_load.b = in`
- `Mux16_load.sel = load`
- `Mux16_load.out = next`

> 语义：`load=1` 时直接装入 `in`，否则采用 inc 分支的结果。这样自然实现了 **load 优先，inc 次之** 的优先级。

5. **最终连线**



> 无 `reset` 引脚；其余保持与标准 PC 一致。若需要上电初值，可在系统层面由外部写一次 `load` 实现，而不是靠 `reset`。

6. **时序说明（与题面 t / t−1 一致）**

- 在时刻 `t-1` 采样控制信号 `inc`，`load` 和数据 `in`，组合逻辑产生 `next`；
- 时钟上升沿到来后，`Register16` 在时刻 `t` 输出 `out(t)=next(t-1)`；
- 满足题面“若 `load` 则装入 `in`，否则若 `inc` 则 `out+1`，否则保持”。

> 评分点：去掉 reset、保持 `load` 高优先级、 `inc` 次优先级、用寄存器反馈实现状态保持、16 位并行连线清楚。

可选进一步优化（非必须）

- 如果需要显示优先级更直观的单级表达式：

$$next = \mathrm{Mux16}\big(\mathrm{Mux16}(out,\ out+1,\ inc),\ in,\ load\big) \tag{5}$$

外层 `sel=load`，内层 `sel=inc`。

[口语化表达]

这题就是要你把“PC 计数器”做出来，但**不要复位线**。逻辑很简单：`load=1` 就把 `in` 直接装进寄存器；否则如果 `inc=1` 就把当前 `out` 加一；都不是就原地不动。画图时用两个 `Mux16` 搭优先级：先用一个 `Mux16` 在 `out` 和 `out+1`（`Inc16` 出来）之间选，选择信号是 `inc`；再用另一个 `Mux16` 在“上一步结果”和 `in` 之间选，选择信号是 `load`（所以 load 优先）。选出来的 `next` 喂给 `Register16`，寄存器的输出就是 `out` 并回馈。这样就满足题意了。

第5页

Hack Assembler and Machine Code

Question 4

For the following questions you may find the information in Figures 3, 4, 5, 6 and 7 in the appendix of this paper useful.

(a) Look at the following Hack machine code:

```
0000000000010000
1111110000010000
0000000000010001
1111000111001000
1111110000010000
0000000000000000
1110001100000001
```

Answer the following:

i. Using the instruction formats in Figures 3, 4, 5, 6, and 7 as a guide, write down the Hack assember instructions that are equivalent to this code.

[7 marks]

ii. Describe what the machine code above does.

[3 marks]

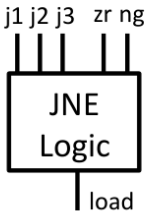
[Total for Question 4: 10 marks]

Computer Architecture

Question 5

For the following questions you may find the information in Figures 1, 2, 3, 4, 5, 6, 7 and 8 in the appendix of this paper useful.

(a) Draw an implementation of the logic that implements the JNE (jump not-equal-to) part of the C-instruction. The interface for the JNE is:



where the inputs are the jump wires from the C instruction and the zr and the ng wires from the ALU and the ouput is the load wire for the PC register. Note that you do not have to implement the logic for every type of jump – just JNE. Hint, in answering your question you may find the information in Figure 7 useful.

[6 marks]

下面是这页（第4页）两道题的中文解读与做法步骤，一次性说清。

Hack 汇编与机器码（Hack Assembler and Machine Code）

Question 4（共10分）

给出的 7 行 **Hack 机器码**（每行 16 位）：

```
0000000000010000
1111110000010000
0000000000010001
1111000111001000
1111110000010000
0000000000000000
1110001100000001
```

(a) i. 译成 Hack 汇编（7分）

按指令格式（A 指令以 0 开头；C 指令以 111 开头）逐行解码：

1. `0000000000010000` → `@16`
2. `1111110000010000` → `D=M`
3. `0000000000010001` → `@17`
4. `1111000111001000` → `M=M-D`
5. `1111110000010000` → `D=M`
6. `0000000000000000` → `@0`
7. `1110001100000001` → `D;JGT`

(a) ii. 机器码做了什么（3分）

- `@16; D=M`：把 `RAM[16]` 读到 D。
- `@17; M=M-D`：执行 `RAM[17] = RAM[17] - RAM[16]`。
- `D=M`：D 取 `RAM[17]`（差值）。
- `@0; D;JGT`：若 D>0（即 `RAM[17] > RAM[16]`）则跳到地址 0（通常回到程序开始/死循环），否则顺序执行。
  结论：比较 `RAM[17]` 与 `RAM[16]`，若前者更大则跳回 0。

计算机体系结构（Computer Architecture）

Question 5（6分）

实现 C 指令中的 **JNE**（jump not equal，即“非零跳转”）逻辑盒。输入：`j1 j2 j3`（C 指令的跳转位）、`zr`（ALU 零标志）、`ng`（ALU 负标志）。输出：`load`（PC 的装载信号）。

- Hack 约定：`JNE` 的条件是 **!zr**，与 `ng` 无关。JNE 的跳转编码是 `j1 j2 j3 = 1 0 1`。
- 逻辑表达式：

$$\text{isJNE} = (j1 \wedge \neg j2 \wedge j3), \quad load = \text{isJNE} \wedge \neg zr \tag{6}$$

电路画法（文字示意）：

```
j1 ──┐
j2 ─ NOT ─┼─ AND3 ── isJNE ─┐
j3 ──┘                      AND ── load
zr ─ NOT ──────────────────┘
```

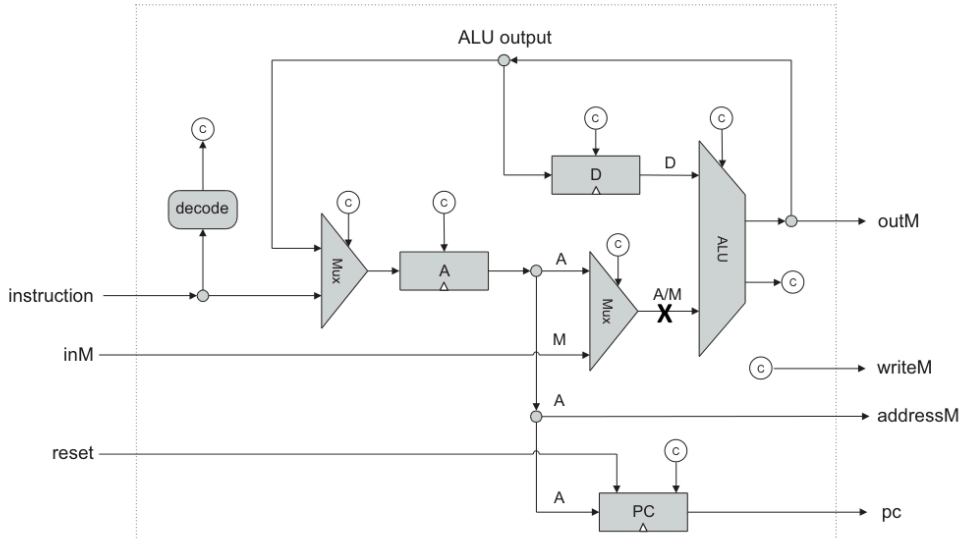
不使用 `ng`。

[口语化表达]

这页就两件事：第一，把 7 条 16 位机器码翻成汇编，结果是 `@16，D=M，@17，M=M-D，D=M，@0，D;JGT`，作用是比较 `RAM[17]` 和 `RAM[16]`，要是前者大就跳回地址 0。第二，实现 JNE 的跳转逻辑：当跳转位是 `101` 且 ALU 的 `zr=0` 时给 PC 的 `load` 置 1，所以 `load = (j1 & ~j2 & j3) & (~zr)`，跟 `ng` 没关系。

第6页

(b) Consider the following diagram of the Hack CPU.



The A/M wire, marked with an **X**, selects either the A-register or RAM as one of the input values for a C-instruction. In the Hack machine a C-instruction cannot access *both* the A-register and RAM as input in the same instruction. So, for example, the following instruction:

```
D=M+A
```

Is not a valid Hack instruction since it has both M and A as input.

Briefly describe why such access is unlikely to be useful even if it were permitted by the Hack machine.

[3 marks]

[Total for Question 5: 9 marks]

1. 下面把图中**每个器件**（或功能块）在 Hack CPU 里**真正用到的控制信号**一次性列清。
（按数据流向从左到右、从上到下，控制位全部来自当前 16 位 instruction 的各字段。）

---

1. A/M 复用器（A-or-M mux，16-bit 2→1）

- 选择信号：`instruction[12]` （comp 字段最高位，a-bit）
- 0 → 选 A 寄存器输出
- 1 → 选 inM（内存值）

---

2. ALU

- `zx` `nx` `zy` `ny` `f` `no` 共 6 位
  直接接 `instruction[11..6]` （comp 字段低 6 位）
  决定“对 x、y 两输入做什么运算”。

---

3. ALU 输出复用器（目的地选择，3 个并行 2→1）

- `loadA` = `instruction[5]` （dest[2]）
  1 → 把 ALU 结果写回 A 寄存器
- `loadD` = `instruction[4]` （dest[1]）
  1 → 把 ALU 结果写回 D 寄存器
- `writeM` = `instruction[3]` （dest[0]）
  1 → 把 ALU 结果作为 outM 写入内存

---

4. PC（程序计数器）

- `loadPC` = `jump 条件满足`
  条件由 `instruction[2..0]` （jump 字段）与 ALU 输出零/负标志共同计算：

```
jlt  : jump if ng
jeq  : jump if zr
jgt  : jump if (¬zr ∧ ¬ng)
任意组合用简单组合逻辑生成 loadPC
```

- `reset` = 外部系统信号
  1 → 强制 PC 置 0

---

5. Register A（ARegister）
- 时钟输入：系统 clk（图中未画出）
- 数据输入选择：已含在 `loadA` 逻辑里（见第 3 项）
- 输出恒驱动 `addressM` 与 A 复用器

---

6. Register D（DRegister）
- 时钟：系统 clk
- 加载使能：`loadD`（见第 3 项）

---

7. 内存接口（RAM）
- 地址：`addressM`（来自 ARegister，无时钟）
- 写使能：`writeM`（见第 3 项）
- 写数据：`outM`（ALU 输出）
- 读数据：`inM`（送回 CPU）

一句话总结
整颗 CPU 的全部控制信号就是 instruction 的 15 位（去掉首位类型位）拆成的 6+3+3 字段，外加一个异步 reset；每个字段直接连到对应器件的选择/使能端，再无额外神秘信号。

Q5(b) 题意与解法

题目给出 Hack CPU 的结构图，并指出**A/M 选择线（图中有 X 标记）**：`inM` 就是 **从内存（RAM）读出的 16 位数据**

- 该选择线决定 C-instruction 送入 ALU 的一个输入到底来自 **A 寄存器** 还是 **RAM[A]（记为 M）**。
- **Hack 机器在同一条 C 指令里不能同时把 A 和 M 都作为输入。**
    - **功能上也不太需要**
        - 把“地址值”和“该地址里的数据”同时参与运算几乎没有常见用途。
        - 真要用，也能**用两条 C 指令完成**，如要计算 `A + M`：

```
1  D=A         // 先把 A 备份到 D
2  D=D+M       // 选 M 作为 y，ALU 计算 D+M（即 A+RAM[A]）
3
```

        - 代价只是多一个周期，换来更简单的硬件与指令编码。
- 例如 `D = M + A` **不是合法**的 Hack 指令。

问题：即便硬件允许在同一条指令里同时访问 A 和 M，这样的访问为什么“也不太有用”？给出简要说明。

核心解释（如何作答）

1. **功能冗余：两条指令就能等价完成**

- 任何“同时用 A 和 M 做运算”的需求，都可以用**两条**合法 C 指令完成，而且不改变语义：

```
1  D = M         // 先把 RAM[A] 读到 D
2  D = D + A     // 再与 A 运算（A 未改变）
```

例如非法的 `D = M + A`，完全可由上面两句实现。

- 因为读取 `M` 不会修改 `A`，所以第二步仍能使用原来的 `A` 值。**因此允许“一条指令同时用 A 和 M”并不会带来表达能力上的新东西，只是最多省一条指令。**

2. **语义上也很少需要**

- 在 Hack 中，**A 通常用作地址寄存器**，而 **M 是该地址处的内存数据**。把“地址值”和“该地址的数据”混在同一条指令里做算术，语义上不常见（地址不是常规数据）。

- 真正常见的是“用内存数据和寄存器数据（如 D）”运算，或“用两个寄存器（D、A）”运算；这些在现有指令集下都能直接或两步完成。

3. **硬件代价与复杂度不值得**

- 同时把 A 与 M 都喂给 ALU，意味着**一次指令要两路来源（其中一路还是内存读取）**。这会增加硬件：需要额外的存储器读端口或暂存/时序，破坏 Hack 的**极简单周期**设计目标。
- 付出更高硬件复杂度，只换来极少的“省一条指令”的收益，**性价比很低**。

一句话答案要点：

> 因为任何“同时使用 A 和 M”的计算都可用两条现有合法指令轻松实现（先 `D=M` 再与 `A` 运算），而且把“地址(A)”和“数据(M)”混算并不常见；若硬件支持反而要付出更高复杂度但几乎没有实际收益。

🗣️ **[口语化表达]**

这题想说的是：A 是地址、M 是这个地址里的数据；你要把它俩一起算，其实用两条正常指令就能搞定——先把 M 读进 D，再拿 D 跟 A 算就行了。硬件上要想一条指令同时喂 A 和 M 给 ALU，会增加内存读口/连线，复杂度上去了，但只换来“偶尔省一条指令”，不值当；而且地址和数据混着算本来也不常见。所以就算允许这么干，也没啥实际用处。

第7页

Assembler

Question 6

(a) Look at the following Hack assembler code:

```
@x
M=0
@y
M=0
(LOOP)
@x
MD=M+1
@y
M=M+D
@3
D=D-A
@LOOP
D;JLE
```

Hand-assemble this code by writing out the binary machine code the assembler would produce. For this question you may find the information in Figures 3, 4, 5, 6, and 7 useful.

[12 marks]

[Total for Question 6: 12 marks]

Virtual Machine - Expressions

Question 7

(a) Translate the following Jack let statement into Hack Virtual Machine language:

```
let d = (2 - x) * (y + 5)
```

The variables d, x and y are in memory segment *local* at indexes 2,5 and 7 respectively. Assume there is a function named *multiply* that will take two arguments and return the result of multiplying the two numbers together.

[8 marks]

[Total for Question 7: 8 marks]

下面是本页（第6页）两道题的逐条中文解析与标准写法。

Assembler

Question 6（12分）

(a) 把给定 Hack 汇编手工汇编成二进制机器码

给定汇编：

```
@x
M=0
@y
M=0
(LOOP)
@x
MD=M+1
@y
M=M+D
@3
D=D-A
@LOOP
D;JLE
```

符号与地址（两遍扫描要点）

- 变量从 RAM16 起：`x=16`，`y=17`。
- 标签 `(LOOP)` 绑定到其后一条指令的 ROM 地址。数指令（不数标签）：

```
addr: 0  @x
1  M=0
2  @y
3  M=0
4  @x     <- LOOP 指向这里
5  MD=M+1
6  @y
7  M=M+D
8  @3
9  D=D-A
10  @LOOP
11  D;JLE
```

因此 **LOOP = 4**。

逐条编码结果（A 指令：0vvvvvvvvvvvvvv；C 指令：111 a c1..c6 d1 d2 d3 j1 j2 j3）

```
@x        -> @16 -> 0000000000010000
M=0             -> 1110101010001000   // comp=0, dest=M
@y        -> @17 -> 0000000000010001
M=0             -> 1110101010001000
@x        -> @16 -> 0000000000010000
MD=M+1          -> 1111110111011000   // comp=M+1, dest=MD
@y        -> @17 -> 0000000000010001
M=M+D           -> 1111000010001000   // comp=D+M, dest=M
@3        -> @3  -> 0000000000000011
D=D-A           -> 1110010011010000   // comp=D-A, dest=D
@LOOP     -> @4  -> 0000000000000100
D;JLE           -> 1110001100000110   // comp=D, jump=JLE
```

> 以上 12 行即该程序的**二进制机器码**。

Virtual Machine – Expressions

Question 7（8分）

(a) 将 Jack 语句翻译为 Hack VM 代码

Jack：`let d = (2 - x) * (y + 5)`

已知 `d,x,y` 在段 `local` 的索引分别为 `2,5,7`；有函数 `multiply` 接收 2 个实参并返回乘积。

正确 VM 代码（注意 sub 的操作数顺序是“次顶 − 顶”）：

```
// (2 - x)
push constant 2
push local 5
sub

// (y + 5)
push local 7
push constant 5
add

// 乘法并赋给 d
call multiply 2
pop local 2
```

速记与得分点

- 第6题先做**符号表**：`x=16，y=17，LOOP=4`；C 指令域：`dest` 顺序是 ADM，`JLE` 的跳码是 110。
- 第7题 `2 - x` 必须先 push 2 再 push x 再 `sub`，否则会变成 `x - 2`。

[口语化表达]

这一页其实就两件事：先把带变量和标签的那段汇编手动翻成 12 行二进制。变量从 16 开始，标签 `(LOOP)` 指到地址 4，然后按 comp/dest/jump 表把每条指令逐个编码就行；接着把 `let d = (2 - x) * (y + 5)` 写成 VM：先算 `2 - x`（`push 2; push local 5; sub`），再算 `y + 5`，然后 `call multiply`

`2`，最后 `pop local 2` 存回 d。照这个写基本稳拿分。

第8页

Virtual Machine - Subroutines

Question 8

(a) The Hack Virtual Machine language provides three function related commands:

- call f m
- function f n
- return

i. Briefly describe the arguments to the `function` command.

[3 marks]

ii. If the `function` command did not have the second argument, what alternate virtual machine code would need to be generated to implement: `function c.x 2` ?

[4 marks]

iii. Why does the second argument to the `call` command need to be provided?

[3 marks]

[Total for Question 8: 10 marks]

Jack

Question 9

(a) How does the **Jack** compiler provided with the nand2Tetris tools ensure that a constructor, function or method from another class is being called correctly? Why does it do this?

[3 marks]

(b) List the syntax errors in the following **Jack** class definition:

```
01 class x
02 {
03     function int xx(var int n)
04     {
05         if ( n <= 2 ) return 17 ;
06         return y.xxx(n--) ;
07     }
08 }
```

[5 marks]

[Total for Question 9: 8 marks]

下面是本页（第7页）的详细中文讲解，包括 Question 8 和 Question 9 的逐项分析与解答。

Virtual Machine - Subroutines

Question 8（共 10 分）

(a) i. `function f n` 的参数含义（3 分）

Hack VM 中 `function f n` 指令的两个参数分别代表：

- **f**：函数名（包含类名的全称，如 `ClassName.FunctionName`）；
- **n**：该函数的局部变量（local variables）数量。
  执行这条指令时，虚拟机会创建一个新的栈帧，并在函数开头自动为这些局部变量分配空间，每个都初始化为 0。

(a) ii. 如果没有第二个参数 n，要如何实现 `function c.x 2`？（4 分）

如果 `function` 没有告诉编译器局部变量数，就必须手动生成初始化局部变量的 VM 指令。

等价代码为：

```
function c.x
push constant 0
pop local 0
push constant 0
pop local 1
```

解释：

- 原本 `function c.x 2` 会自动生成两次 `push constant 0` 初始化 local 段；
- 现在因为没有 n 参数，只能手动添加；

- 这保证了 local[0] 和 local[1] 都存在并被清零。

(a) iii. 为什么 `call` 指令的第二个参数必须提供？（3 分）

因为：

- `call f m` 的第二个参数 **m** 表示要传递给函数的参数数量；
- 翻译器需要用它来计算 **ARG 指针的位置**，即：

```
1  ARG = SP - m - 5
```

（其中 5 表示返回地址、LCL、ARG、THIS、THAT 五个保存项）

- 如果省略 m，编译器无法正确移动栈指针，也就不能让被调用函数正确取到参数；
- 因此，这个参数是必须的。

Jack

Question 9（共 8 分）

(a) nand2tetris 的 Jack 编译器如何确保"从其他类调用构造函数/函数/方法"是正确的？（3 分）

- Jack 编译器**不会跨类全局检查目标函数是否真实存在**，但会根据语法结构判断调用类型。例如：
- `ClassName.func()` → 编译为**函数或构造函数调用**；
- `objectName.method()` → 编译为**方法调用**（并自动在参数前压入对象引用 `this`）。
- 编译器通过符号表（Symbol Table）识别 `ClassName` 与变量名的区别，生成正确的调用格式（带或不带隐式参数）。
- 这么做的原因是：项目教学目标是让学生理解虚拟机调用约定，而非构建完整链接器，所以编译器只保证语法合法，不检查跨文件定义。

(b) 找出以下 Jack 类定义中的语法错误（5 分）

```
01 class x
02 {
03   function int xx(var int n)
04   {
05     if ( n <= 2 ) return 17 ;
06     return y.xxx(n--) ;
07   }
08 }
```

逐项分析错误：

1. **参数声明错误**：

- `var` 不能用于函数参数，只能用于函数体内局部变量声明。

> - 因为 JavaScript 的 `var` **没有块级作用域**，只有**函数作用域**。
> 如果把 `var` 允许放在参数列表里，它会被“提升”到整个函数顶部**只要在函数体内出现 `var x`，无论写在哪一行，引擎都会在进入函数时把它当成在函数顶部声明的局部变量**。，导致：
>
> 1. 同一函数体内还可以再 `var x` 声明一次 x，产生**重复声明**却不报错；
> 2. 参数与提升后的变量实际上是**同一个绑定**，修改其一就改了另一个，极易造成**隐蔽 bug**；
>
> ES6 以后引入的 `let` / `const` 具备块级作用域且禁止重复声明，才解决了这些问题。
> 因此标准干脆**禁止在参数列表中使用 `var`**，只让 `let` / `const` 出现，保持语义清晰、安全。

- 正确写法：`function int xx(int n)`

2. **if 语句语法错误**：

- Jack 语言中 `if` 语句的语法是
  `if (condition) { statements } [else { statements }]`
- 必须加花括号 `{}` 包围语句块。

- 正确：`if (n < 3) { return 17; }`

3. **比较运算符错误**：

- Jack 只支持 `<`，`>`，`=`，没有 `<=`。
- 可以改为：`if (n < 3)`。

4. **自减运算符错误**：

- Jack 不支持 `n--` 或 `++`。
- 应改成：`n = n - 1; return y.xxx(n);`

> 以上为语法错误。`y` 未声明属**语义错误**（符号未定义），非语法错误，因此不列入答案。

[口语化表达]

这页主要讲函数与子程序的 VM 机制，还有 Jack 的语法。

首先，`function f n` 里 f 是函数名，n 是局部变量数；如果没有 n，就得自己手动写几条 `push constant 0` 给 local 段占位。`call f m` 里的 m 也不能省，因为编译器要靠它算出 ARG 指针，否则参数会乱套。

接着是 Jack 部分：编译器不会真去查另一个类的函数存不存在，只靠符号表区分类名和对象名，按规则生成调用。最后那段 Jack 代码错了一堆：参数不能加 `var`，`if` 要有花括号 `{}`，不能写 `<=`，也不支持 `n--`。改好这几个就没问题啦。

Parsing

Question 10

(a) Turn the following **Jack** code fragment into XML with one node for each non-terminal in the grammar.

```
let x[ix] = y ;
```

You should start with a node for a let statement and you may omit nodes for any keywords, identifiers or symbols. The grammar can be found in Figure 9 in the appendix.

[8 marks]

[Total for Question 10: 8 marks]

下面是本页（第8页）**Question 10** 的中文讲解与参考答案。

Parsing

Question 10（共8分）

题意

把下面这句 **Jack** 代码按照语法（只能为**每个非终结符**建一个 XML 节点）改写成 XML。
代码：`let x[ix] = y ;`
要求：从 `letStatement` 作为根节点开始；**关键词、标识符、符号**（如 `let`、`x`、`[`、`]`、`=`、`;` 等）可以省略节点。

Jack（nand2tetris）相关产生式（只取本题用到的）：

```
letStatement → 'let' varName ('[' expression ']')? '=' expression ';'
expression   → term (op term)*
term         → integerConstant | stringConstant | keywordConstant
| varName | varName '[' expression ']' | '(' expression ')' | unaryOp term
```

本题中：

- `varName` 是 **x**（终结符，不建节点）；
- 左侧是 **数组形式**，因此有一个 **index 的 expression**（即 `ix`，本质是一个 `term` / `varName`）；
- 右侧 **RHS** 是一个 **expression**（即 `y`，同样是一个 `term` / `varName`）。

参考XML（仅包含非终结符，符合“每个非终结符一个节点”的要求）

```
<letStatement>
<!-- array index: x[ ix ] -->
<expression>        <!-- index expression -->
<term/>           <!-- varName: ix -->
</expression>

<!-- right-hand side: y -->
<expression>        <!-- RHS expression -->
<term/>           <!-- varName: y -->
</expression>
</letStatement>
```

> 说明：上面**不为** `let`、`x`、`[`、`]`、`=`、`;`等终结符建节点；`ix`与`y`作为标识符也不单独建节点，只以`<term/>`占位，满足“仅非终结符建节点”的要求。如果你想更直观，可在`<term>`里加上文字注释或属性标明是`ix`、`y`（题目允许省略终结符节点）。

作答要点/评分点

- 根节点必须是 `letStatement`；
- 包含 **两个** `expression`：一个代表数组 **索引**（来自 `x[ ... ]`），一个代表 **右值**（`= ...`）；
- 每个 `expression` 下至少有一个 `term`；
- 不为关键词/符号/标识符建立节点（或即便建了也不加分，题干允许省略，重点在**非终结符**齐全与结构正确）。

[口语化表达]

这题就是把 `let x[ix] = y;` 画成**语法树的 XML**，而且只给**非终结符**建节点。根是 `<letStatement>`，里面要有两个 `<expression>`：第一个是数组下标 `ix` 的表达式，第二个是右边 `y` 的表达式；每个表达式里面放一个 `<term>` 占位就行。像 `let`、`x`、`[`、`]`、`=`、`;`、以及标识符本身都算终结符，可以不建节点。按我给的那个 XML 抄就对了。

第10页

Code Generation

Question 11

(a) Consider the following **Jack** method:

```
// class Complex contains 4 instance variables
// declared in this order: aa, bb, cc and dd, aa is an Array
method Complex useful(Complex a, Complex b)
{
}
```

What Hack Virtual Machine language code would implement the following Jack program fragments if they were in the body of the method `useful`?

i.
```
let b = Complex.new(3,2) ;
```
[4 marks]

ii.
```
aa[7] = a ;
```
[6 marks]

iii.
```
return b ;
```
[2 marks]

iv.
```
let bb = dd ;
```
[3 marks]

(b) Show the two symbol tables for the following code just after the variable declaration in the method getSerial has been parsed.

```
class SerialNums
{
        static int id ;
        field int myid ;

        constructor SerialsNums new(int key)
        {
                let myid = id ;
                return this ;
        }
        method int getSerial(int password)
        {
                var int ignore_me ;
                return myid ;
        }
}
```

[4 marks]

[Total for Question 11: 19 marks]

下面是本页（第9页）**Question 11：Code Generation** 的逐题中文解析与标准答案。

Question 11（总分 19）

(a) 把给定 Jack 代码片段翻译成 Hack VM 代码

> 已知：`class Complex` 有 4 个实例字段，声明顺序 **aa, bb, cc, dd**（其中 **aa 为 Array**）。
> 当前方法头：`method Complex useful(Complex a, Complex b) { ... }`
> 方法调用约定：`argument 0` 为隐式 `this`，`a → argument 1`，`b → argument 2`；字段映射到 `this` 段：`aa→this 0`，`bb→this 1`，`cc→this 2`，`dd→this 3`。

i. `let b = Complex.new(3,2);` （4分）

```
push constant 3
push constant 2
call Complex.new 2
pop argument 2        // b ← new(3,2)
```

ii. `aa[7] = a;` （6分）

```
push this 0           // base = aa
push constant 7
add                   // base + 7
pop pointer 1         // THAT = base+7
push argument 1       // a
pop that 0            // *THAT = a
```

iii. `return b;` （2分）

```
push argument 2
return
```

iv. `let bb = dd;` （3 分）

```
1  push this 3           // dd
2  pop this 1            // bb = dd
```

(b) 画出符号表（4 分）

> 位置：在 **解析完 getSerial 方法里的变量声明** `var int ignore_me;` 之后。

给定代码（节选）：

```
1  class SerialNums {
2  static int id ;
3  field  int myid ;
4
5  constructor SerialsNums new(int key) {
6  let myid = id ;
7  return this ;
8  }
9
10 method int getSerial(int password) {
11 var int ignore_me ;
12 return myid ;
13 }
14 }
```

类级符号表（Class Scope）

name	type	kind	index
id	int	static	0
myid	int	field	0

子程序级符号表（Subroutine Scope；当前位于 `getSerial`）

name	type	kind	index
password	int	argument	0
ignore_me	int	var	0

> 说明：方法的隐式 `this` 不作为符号表条目列出；构造器 `new(int key)` 的参数与本方法无关，属于另一个子程序作用域。

得分小贴士

- 方法参数与字段的段映射要写对： `argument` 与 `this` 。
- 数组写入套路： `base+index → THAT` ，再 `pop that 0` 。
- 符号表分两张：类级（ `static/field` ）与子程序级（ `argument/var` ），索引从 0 递增。

[口语化表达]

这页要把几句 Jack 代码翻成 VM 并列出符号表。方法里 `a` 在 `argument 1` 、 `b` 在 `argument 2` ，字段 `aa/bb/cc/dd` 对应 `this 0..3` 。

- `b = Complex.new(3,2)` ： `push 3; push 2; call Complex.new 2; pop argument 2` 。
- `aa[7] = a` ：先把 `aa` 的基址取出来加 7，写到 `THAT` ，再把 `a` 存到 `that 0` 。
- `return b` ： `push argument 2; return` 。
- `bb = dd` ： `push this 3; pop this 1` 。
  符号表方面，类表有 `id(static0)` 、 `myid(field0)` ；到了 `getSerial` ，子程序表就是 `password(argument0)` 和 `ignore_me(var0)` 。照这个写就稳了。

Jack OS, Optimisation

Question 12

(a) Why might implementing a 16-bit multiply operation in the ALU of the Hack machine significantly increase the time it takes to execute an instruction that sets the A and D registers to the value 0?

[3 marks]

(b) What three aspects of a processor's physical implementation determine the power consumption and how is this calculated?

[4 marks]

[Total for Question 12: 7 marks]

下面是本页（第10页）**Question 12** 的中文解析与标准答题要点。

Jack OS, Optimisation

Question 12（共7分）

(a) 为什么在 Hack 机器的 ALU 中加入16位乘法器，会显著增加"把 A 和 D 置 0"这类简单指令的执行时间？（3分）

核心原因：临界路径（critical path）变长 → 时钟周期被迫拉长。

- Hack 的 ALU是**组合电路**，同一个数据通路要同时支持所有运算。
- 一旦把**乘法器**并入 ALU，为了在一拍内完成乘法，**乘法器的传播延迟**（远大于加法/常量输出）就会出现在可能的选择路径/多路复用器之后，使**最坏情况延迟**变大。
- 处理器时钟必须按**最慢的操作**设定（满足乘法的延迟），因此**即便执行"输出 0"这类本来很快的操作**，也要用更长的时钟周期。
- 结论：**添加乘法器会拖慢所有指令**（包括 `A=0`、`D=0`），因为它提高了全 ALU 的最坏路径延迟。

> 可顺带指出：若 ALU 通过更大的多路复用器统一选择结果，乘法器还会让**多路复用和扇出**更重，进一步加大延迟。

(b) 处理器物理实现的三个方面决定功耗？如何计算？（4分）

聚焦 CMOS 数字电路的主功耗来源（动态功耗为主，辅以静态）：

1. **供电电压 V**
2. **时钟频率 f**
3. **等效开关电容 C_eff**（由晶体管尺寸/连线负载/门数及**切换活动因子 α** 决定）

动态功耗（开关充放电）：

$$P_{dyn} \approx \alpha\, C_{\text{eff}}\, V^2\, f \tag{7}$$

- 降低 V、f、C_eff 或活动因子 α 都能降功耗。

静态/泄漏功耗（器件不切换也耗电）：

$$P_{leak} \approx I_{leak} \cdot V \tag{8}$$

- 由工艺、阈值电压、温度与晶体管数目决定。

总功耗：$P_{total} \approx \alpha C_{\text{eff}} V^2 f + I_{leak} V$。

🗣️ **[口语化表达]**

这页就两问：第一，往 ALU 里塞了个16位乘法器，会把整条数据通路的**最慢延迟**拉长，时钟周期被迫加大，所以连“置0”这种本来很快的指令也要等更久。第二，功耗主要看三件事：**电压V、频率f、等效电容C_eff（含切换率α）**，公式就是 **P≈α·C_eff·V²·f**，再加上泄漏 **P_leak=I_leak·V**，两者相加就是总功耗。

Primary Examination, Semester 1, 2017

APPENDICES

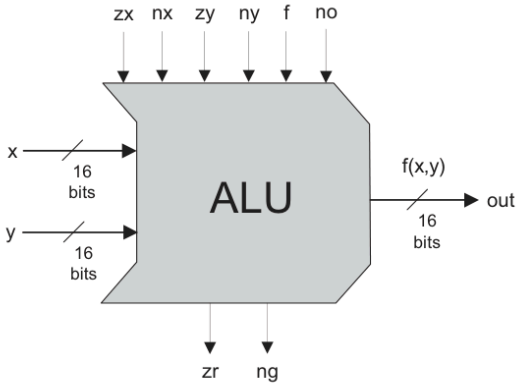


Figure 1: An interface diagram for the ALU. From figure 2.5 of the textbook.

These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Figure 2: The Hack ALU truth table. From figure 2.6 of the textbook.

PLEASE SEE NEXT PAGE

下面是本页（第11页，附录）内容的中文解读与使用方法说明。

这页在讲什么？

本页给了 **Hack ALU 的接口图（Figure 1）** 和 **ALU 真值表（Figure 2）**。

- **接口图**告诉你 ALU 的输入/输出：
- 数据输入：`x[16]`、`y[16]`
- 控制位输入：`zx nx zy ny f no`（6 个 1bit）
- 数据输出：`out[16]`
- 状态标志输出：`zr`（out 是否为 0），`ng`（out 的符号位是否为 1，即负数）
- **真值表**告诉你：给定 6 个控制位如何把 `x`、`y` 变换并组合成 `out`，以及能实现哪些常见操作（0、1、-1、x、y、!x、!y、-x、-y、x+1、y+1、x-1、y-1、x+y、x-y、y-x、x&y、x|y）。

---

6 个控制位各自的作用

按硬件实现的顺序可以理解为“先预处理输入，再选择运算，再可能取反输出”：

1. **预置/改造 x**

- `zx`：若为1，则把 x 清零（送 0）；若为0，保留 x。
- `nx`：对“预置后的 x”按位取反（1→0，0→1）。

2. **预置/改造 y**

- `zy`：若为1，则把 y 清零；若为0，保留 y。
- `ny`：对“预置后的 y”按位取反。

3. **选择运算**

- `f`：为1用**加法**（`x + y`），为0用**按位与**（`x & y`）。

4. **后置处理**

- `no`：对上一步的结果 `out` 取反（为1时取反，为0时原样）。

> **标志位**：
>
> - `zr = 1` 当且仅当 `out == 0`；
> - `ng = 1` 当且仅当 `out` 的最高位（符号位）为 1（即负数）。

如何使用真值表（读表+例子）

真值表的每一行对应一组控制位，会标出得到的函数 `f(x,y)= ...`。使用时：

1. 先确定想要的运算（如 `x+y` / `x&y` / `0` / `-1` / `!x` 等）；
2. 在表里找到这一行；
3. 读出该行的 6 个控制位，即可驱动 ALU。

常见操作（给出**正确的一组**控制位，便于记忆和考试使用）：

- **`x + y`**： `zx=0 nx=0 zy=0 ny=0 f=1 no=0`
- **`x & y`**： `zx=0 nx=0 zy=0 ny=0 f=0 no=0`
- **`0`**： `zx=1 nx=0 zy=1 ny=0 f=1 no=0`
- **`1`**： `zx=1 nx=1 zy=1 ny=1 f=1 no=1`
- **`-1`**： `zx=1 nx=1 zy=1 ny=0 f=1 no=0`
- **`x`**： `zx=0 nx=0 zy=1 ny=1 f=0 no=0`（y 被清零并取反两次，相当于0）
- **`y`**： `zx=1 nx=1 zy=0 ny=0 f=0 no=0`
- **`!x`**： `zx=0 nx=1 zy=1 ny=1 f=0 no=1`（等价于把 x 取反输出）

- **`x-1`**、**`x+1`** 等自增/自减、取负 `-x` 等也都能在表中直接查到对应控制位。

> 小技巧：
>
> - 需要“把某一侧置 0”时，用 `z*` =1；需要“按位取反”时，用 `n*` =1。
> - `f=1` 走加法，`f=0` 走按位与；最后 `no` 再决定是否把结果整体取反。

标志位的用法（和跳转相关）

- 执行完 ALU，硬件会据 `out` 自动置位：
- `zr=1` ⇔ `out==0`；
- `ng=1` ⇔ `out` 为负（最高位=1）。
- Hack 的跳转（如 `JGT/JLT/JNE` 等）正是根据 **`zr`**、**`ng`** 判断的，例如 `JNE` 只看 `!zr`。

🗣️ **[口语化表达]**
这页就是 ALU 的“使用手册”：6 个小开关（`zx nx zy ny f no`）先把 x、y 清零/取反，再选是加法还是与，最后再决定要不要把结果取反；输出除了 16 位结果外，还送出 `zr`（是否为0）和 `ng`（是否为负）。你要啥功能就去真值表找那一行，比如 `x+y` 就是 `0 0 0 0 1 0`，`x&y` 是 `0 0 0 0 0 0`，要常量 0 就把 x、y 都清零再选加法（`1 0 1 0 1 0`）。查到控制位，塞进 ALU，标志位也会自动给你算好，后面的跳转就能用了。

A-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

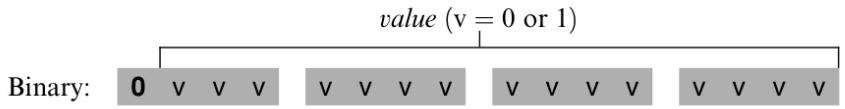


Figure 3: The format of an A-instruction. From page 64 of the text book.

C-instruction: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.

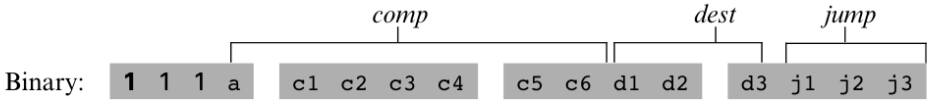


Figure 4: The format of an C-instruction. From page 66 of the text book.

(when a=0) ***comp*** *mnemonic*	c1	c2	c3	c4	c5	c6	(when a=1) ***comp*** *mnemonic*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

Figure 5: The meaning of C-instruction Fields. From figure 4.3 of the textbook.

这页（第12页）主要是 Hack 机器指令编码的**格式与二进制结构说明**，没有题目，但它是前几页汇编题（例如第6题）的关键参考资料。下面详细讲每一部分的意思和使用方法。

一、A-指令（@value）

格式说明

A 指令形如：

```
@value
```

其中 `value` 可以是：

- 非负整数（例如 `@21`）
- 或一个符号（例如 `@LOOP`、`@i`），汇编器会在符号表查到其对应地址。

二进制格式

A 指令的机器码是 **16位**，以 **0** 开头：

```
0 v v v v v v v v v v v v v v v
```

- 第一位永远是 0（表示这是 A 指令）。
- 后 15 位是地址或常数的二进制值。

例如：

```
@21  →  0000000000010101
```

二、C-指令（dest = comp ; jump）

C 指令用来描述计算、存储与条件跳转。
一般形式如下：

```
dest = comp ; jump
```

三部分可任选：

- **comp**：计算部分（必须有）
- **dest**：可选，用来指定把结果存到哪个寄存器/内存（如 D、M、A、MD、AM、AD、AMD）
- **jump**：可选，用于控制跳转（如 JGT、JEQ、JMP 等）

例子：

```
D=M+1       // comp=M+1, dest=D
D;JGT       // comp=D, jump=JGT
M=D+1;JMP   // dest=M, comp=D+1, jump=JMP
```

三、C 指令的二进制格式

16 位格式如下：

```
1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3
```

- **前三位固定为 111**（表明是 C 指令）
- **a c1~c6**：共 7 位表示 comp（计算操作）
- **d1~d3**：3 位表示 dest（目标寄存器）
- **j1~j3**：3 位表示 jump（跳转条件）

四、comp 字段（Figure 5）

`comp` 字段（a, c1~c6）决定 ALU 做什么计算。

- 当 `a=0` → 计算中使用 **A 寄存器**；
- 当 `a=1` → 使用 **M（内存[A]）**。

comp	对应 c1~c6（二进制）	含义		
0	101010	常数 0		
1	111111	常数 1		
-1	111010	常数 -1		
D	001100	D		
A	110000	A		
M	110000（a=1）	M		
D+1	011111	D+1		
A+1	110111	A+1		
D-1	001110	D-1		
A-1	110010	A-1		
D+A	000010	D+A		
D-A	010011	D-A		
A-D	000111	A-D		
D&A	000000	D&A		
D	A	010101	D	A

五、dest 字段（存储目标）

三位二进制分别对应：

位	含义	1表示存储到
d1	A 位	A 寄存器
d2	D 位	D 寄存器
d3	M 位	内存地址 [A]

例如：

dest	d1 d2 d3
null	0 0 0
M	0 0 1
D	0 1 0
MD	0 1 1
A	1 0 0
AM	1 0 1
AD	1 1 0
AMD	1 1 1

六、jump 字段（跳转条件）

跳转条件	j1 j2 j3	含义
null	000	不跳转
JGT	001	若 out>0
JEQ	010	若 out=0
JGE	011	若 out≥0
JLT	100	若 out<0
JNE	101	若 out≠0
JLE	110	若 out≤0
JMP	111	无条件跳转

七、完整例子

指令：

```
D=M+1;JGT
```

拆解：

- comp = M+1 → `a=1 c1~c6=110111`
- dest 无（=空）→ `d1d2d3=000`
- jump = JGT → `j1j2j3=001`

拼接：

```
111 1 110111 000 001
```

即：

```
1111110111000001
```

[口语化表达]

这一页其实是 Hack 机器指令的“说明书”。A 指令超简单，就是 `@` 开头，后面数值转成二进制放到 15 位里；C 指令稍复杂，是“111 + 运算 + 目的地 + 跳转”。要看运算怎么写，就查那个大表：如果用 A 就 a=0，用 M 就 a=1；D、A、M、+1、-1、|、& 全都有编码。存哪儿就改 dest 位，跳不跳就看 jump 位。比如 `D=M+1;JGT` 就是 `1111110111000001`。这页主要就是帮你在写 Hack 汇编题（像第6题那种）时查表转机器码的。

---

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

Figure 6: The meaning of the destination bits of the C-instruction From figure 4.4 of the textbook.

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

Figure 4.5 The *jump* field of the *C*-instruction. *Out* refers to the ALU output (resulting from the instruction's *comp* part), and *jump* implies "continue execution with the instruction addressed by the A register."

Figure 7: The meaning of the jump bits of the C-instruction From figure 4.5 of the textbook.

Label	*RAM address*
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
R0-R15	0-15
SCREEN	16384
KBD	24576

Figure 8: The predefined symbols in Hack Assembly language. From page 110 of the text book.

下面是本页（第13页，附录）的中文解读与“怎么用”的说明。它包含三张表：**C 指令的目的地位（dest）**、**跳转位（jump）**、以及 **Hack 预定义符号**。

Figure 6： C-instruction 的目的地位（dest）

- 三个位 **d1 d2 d3** 分别对应 **A、D、M** 三个目的寄存器/内存：
- **d1=1** → 结果写入 **A**
- **d2=1** → 写入 **D**
- **d3=1** → 写入 **M**（即 RAM[A]）
- 组合即可同时写多处（如 AD、AM、MD、AMD）。
- 表中助记符（Mnemonic）就是我们写汇编时的 `dest=` 部分；右侧文字描述是“结果被存到哪里”。

快速记忆：dest 的位顺序固定是 **A D M**（从高到低：d1 d2 d3）。

例子

- `D=M+1` → dest=D → **d1d2d3 = 010**
- `AMD=D-1` → dest=AMD → **111**
- `M=D` → dest=M → **001**

Figure 7： C-instruction 的跳转位（jump）

- 三个位 **j1 j2 j3** 描述跳转条件；它们和 ALU 输出的符号/零标志（ `out<0` ， `out=0` ， `out>0` ）对应。
- 助记符与含义：
- **000 null**：不跳转

- **001 JGT**：若 `out > 0` 跳
- **010 JEQ**：`out == 0`
- **011 JGE**：`out ≥ 0`
- **100 JLT**：`out < 0`
- **101 JNE**：`out ≠ 0`
- **110 JLE**：`out ≤ 0`
- **111 JMP**：无条件跳

例子

- `D;JLE` → comp=D，jump=JLE → **j1j2j3 = 110**
- `0;JMP` → 跳转恒真（常写成死循环）

实操要点：是否跳转由 ALU 的 `zr`、`ng` 决定；编译成机器码时把 jump 三位放到尾部。

Figure 8： Hack 预定义符号（符号名 → RAM 地址）

符号	含义/段	地址
SP	栈顶指针	0
LCL	当前函数的 Local 段基址	1
ARG	Argument 段基址	2
THIS	THIS 段基址	3
THAT	THAT 段基址	4
R0–R15	通用寄存器别名	0–15
SCREEN	屏幕内存起始	16384
KBD	键盘内存起始	24576

使用规则

- 在 A 指令中可直接写这些符号：如 `@SP`、`@SCREEN`，汇编器替换为相应地址。
- **变量符号**（非预定义、非标签）从 **RAM[16]** 起自动分配：第一次出现的变量取 16，下一个 17，以此类推（这条不在本页，但与本表配套使用）。

汇编到机器码时怎么用这三张表?

1. **确定 comp**（见上一页 Figure 5），得到 `a c1..c6`。
2. **根据本页 Figure 6** 把目标写入位置转换成 **d1 d2 d3**。
3. **根据本页 Figure 7** 把跳转条件转换成 **j1 j2 j3**。
4. 拼出 16 位 C 指令：`111 a c1..c6 d1 d2 d3 j1 j2 j3`。
5. 若是 A 指令，若写 `@SP/@SCREEN` 等，直接代换地址后编码为 `0 + 15位地址`。

完整例子

`D=M+1;JGT` →

- comp `M+1` → `a=1, c=110111`
- dest 空 → `000`
- jump `JGT` → `001`
  机器码：`111 1 110111 000 001` → **1111110111000001**

[口语化表达]

这页就是三张“速查表”：

- 第一张告诉你 C 指令要把结果存到哪（A/D/M），就把 **dest 的 ADM 三位**设成 1 或 0；
- 第二张告诉你 **jump 的三位**怎么写，对应 `out` 的大小关系（JGT/JEQ/JMP…）；
- 第三张列出了所有 **预定义符号**的内存地址（SP=0、LCL=1、SCREEN=16384、KBD=24576…）。
  写汇编时，A 指令用这些符号很方便；C 指令则用这两张表把 `dest` 和 `jump` 变成二进制，和上一页的 `comp` 一起拼成 16 位机器码，妥妥的。

第15页

Lexical Elements

```
keyword         ::= 'class' | 'constructor' | 'function' | 'method' | \
                    'field' | 'static' | 'var' | 'int' | 'char' | \
                    'boolean' | 'void' | 'true' | 'false' | 'null' | \
                    'this' | 'let' | 'do' | 'if' | 'else' | 'while' | \
                    'return'
symbol          ::= '{' | '}' | '(' | ')' | '[' | ']' | '.' | \
                    ',' | ';' | '+' | '-' | '*' | '/' | '&' | \
                    '|' | '<' | '>' | '=' | '~' | '
integerConstant ::= A decimal number in the range 0 .. 32767
stringConstant  ::= '"' A sequence of Unicode characters not including
                        double quote or newline '"'
identifier      ::= A sequence of letters, digits and underscore ('_')
                    not starting with a digit.
```

Statements

```
statements      ::= statement*
statement       ::= letStatement | ifStatement | whileStatement | \
                    doStatement | returnStatement}
letStatement    ::= 'let' varName ('[' expression ']')? '=' expression ';'
ifStatement     ::= 'if' '(' expression ')' '{' statements '}' \
                   ('else' '{' statements '}')?
whileStatement  ::= 'while' '(' expression ')' '{' statements '}'
doStatement     ::= 'do' subroutineCall ';'
returnStatement ::= 'return' expression? ';'
```

Expressions

```
expression      ::= term (op term)*
term            ::= integerConstant | stringConstant | \
                    keywordConstant | varName | \
                    varName '[' expression ']' | subroutineCall | \
                    '(' expression ')' |  unaryOp term
subroutineCall  ::= subroutineName '(' expressionList ')' |  \
                    (className | varName) '.' subroutineName '(' expressionList ')'
expressionList  ::= (expression (',' expression)*)?
op              ::= '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
unaryOp         ::= '-' | '~'
keywordConstant ::= 'true' | 'false' | 'null' | 'this'
varName         ::= identifier
```

Figure 9: The Jack grammar. From figure 10.5 of the textbook.

END OF EXAMINATION PAPER

这页（第14页）是 **Jack 语言的语法定义（grammar）**，用于说明 Jack 编译器如何解析源代码。它并没有题目，但为前面的题（比如 Parsing、Code Generation、Jack 语法分析相关）提供规则依据。下面是详细解释每一部分的意思，以及在考试或编译器实现中该如何使用。

一、Lexical Elements（词法单元）

这一部分定义了 Jack 语言的基本“单词”，也就是**token 类型**。

项目	含义	举例	
keyword	Jack 保留字（系统关键字）	`class`，`function`，`method`，`let`，`if`，`return`，`do`，`while`，`true`，`false`，`null`，`this` 等	
symbol	特殊符号	`{ } ( ) [ ] . , ; + - * / &`	`< > = ~` 等
integerConstant	整型常数，范围 0–32767	`0`，`100`，`32767`	
stringConstant	字符串常数（双引号包裹，不含换行）	`"Hello World"`	
identifier	标识符，用于变量名、函数名、类名，不以数字开头	`x`，`sum`，`_index`，`Point3D`	

> **用途**：词法分析（Lexical Analysis）时，编译器扫描源文件并根据这些规则分割出 token。

二、Statements（语句）

定义了 Jack 程序由哪些语句组成及其结构。

statements

表示由多个 `statement` 组成的语句块（即 `{ ... }` 里的内容）。

statement

语句类型包括：

- `letStatement` → 赋值语句
- `ifStatement` → 条件语句
- `whileStatement` → 循环语句
- `doStatement` → 调用子程序语句
- `returnStatement` → 返回语句

各语句结构

1. **letStatement**

```
let varName (‘[’ expression ‘]’)? = expression ;
```

说明：赋值语句，可以对数组元素赋值。
例：`let x = 3;` 或 `let arr[i] = y;`

2. **ifStatement**

```
if ( expression ) { statements } (else { statements })?
```

可选 `else` 块。
例：

```
if (x < 5) {
let y = 0;
} else {
let y = 1;
}
```

3. **whileStatement**

```
while ( expression ) { statements }
```

例：`while (i < 10) { let i = i + 1; }`

4. **doStatement**

```
do subroutineCall ;
```

调用方法但不接收返回值。

例：`do Screen.drawLine();`

5. **returnStatement**

```
return expression? ;
```

可带返回值，也可以空返回。

例：`return;` 或 `return x;`

三、Expressions（表达式）

表达式定义如何构成值。

expression

```
term (op term)*
```

表示一个或多个 `term` 由运算符连接组成。

例：`x + 5 * y`

term

```
integerConstant | stringConstant | keywordConstant | varName |
varName [ expression ] | subroutineCall | ( expression ) | unaryOp term
```

说明一个最小的“项”，可能是：

- 常数（整型或字符串）
- 变量
- 数组元素
- 调用
- 括号表达式
- 单目运算（取负、按位非）

subroutineCall（子程序调用）

有两种形式：

1. 直接调用：`subroutineName ( expressionList )`
2. 通过对象/类调用：`(className | varName) . subroutineName ( expressionList )`

例如：

```
do move();
do Math.multiply(x, y);
```

expressionList（参数列表）

```
(expression (',' expression)*)?
```

逗号分隔的表达式序列，可为空。

op（运算符）

`+ - * / & | < > =`

（注意：乘除最终会调用 OS 库函数 `Math.multiply` 与 `Math.divide`）

unaryOp（单目运算）

`-` 或 `~`（负号、按位取反）

keywordConstant（特殊常量）

`true`，`false`，`null`，`this`

四、用途总结

- 在**编译器的语法分析（Parsing）阶段**，Jack 代码按此规则生成 XML 树。
- 例如题目 “Turn `let x[ix] = y;` into XML” 就是用这些规则逐层展开：

```
<letStatement>
<keyword>let</keyword>
<identifier>x</identifier>
<symbol>[</symbol>
<expression>...</expression>
<symbol>]</symbol>
<symbol>=</symbol>
<expression>...</expression>
<symbol>;</symbol>
</letStatement>
```

🗣️ **[口语化表达]**

这页其实就是 Jack 语言的“语法说明书”。上面先告诉你它有哪些关键词（像 class、function、let、if、return 这些），接着列出语句结构，比如 `let` 怎么写、`if` 要带括号和大括号、`do` 是子程序调用、`return` 可有可无表达式。下面那块讲表达式怎么构成，什么是 term、expression、subroutineCall。像 `x + 5`、`arr[i]`、`do Math.multiply(x, y)` 都是根据这套规则来的。

总结一句话：**这页是你写语法分析器或 XML 结构时的标准模板**，按照它的产生式定义去递归解析，整个 Jack 编译器的 Parser 就能跑通。