Lab Report: Learn from the past

Zhu Yunqin, PB20061372

Task

Use a high-level programming language to implement all the code that has been written before. The algorithm needs to be consistent with what was used before.

Solution

1) lab1 L-version: *lab0l*

- **Algorithm**
  When multiplying two 16-bit integers a and b, we actually calculate the b-th power of a in the group $(\mathbb{Z}_{65536}, +)$. Consider exponentiating by squaring. Write the exponent b as

  $$b = \sum_{i=0}^{15} b_i \cdot 2^i.$$

  where b^i represents the i-th binary digit of b.Then we have

  $$a \times b = \sum_{i=0}^{15} b_i \cdot 2^i a.$$

  The algorithm runs in $O(w)$ time, where w is the number of bits for an integer.
- **LC3 assembly code**
  Input: At the beginning, `R0` and `R1` store a and b respectively.
  Output: `R7` stores $a \times b$ as the result.

```
      ADD R2, R2, #1
LOOP  AND R3, R0, R2
      BRz SKIP
      ADD R7, R7, R1
SKIP  ADD R1, R1, R1
      ADD R2, R2, R2
      BRnp LOOP
      HALT
```

- **C++ code**
  Input: At the beginning, global variants `a` and `b` store a and b respectively.
  Output: `c` stores the result $a \times b$.

```cpp
int main() {
  for (int i = 0; i < 16; i++)
    if (b & 1 << i) c += a << i;
  return 0;
}
```

2) lab1 P-version: *lab0p*

- **Algorithm**
  The same as lab1 L-version, but the loop structure is unrolled.
- **LC3 assembly code**
  Input & Output: Refer to lab1 L-version.

```
        ADD R2, R2, #8
        AND R3, R0, #1
        BRz NEXT1
```

```
        ADD R7, R7, R1
NEXT1   ADD R1, R1, R1
        AND R3, R0, #2
        BRz NEXT2
        ADD R7, R7, R1
NEXT2   ADD R1, R1, R1
        AND R3, R0, #4
        BRz NEXT3
        ADD R7, R7, R1
NEXT3   ADD R1, R1, R1
        AND R3, R0, R2
        BRz NEXT4
        ADD R7, R7, R1
NEXT4   ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT5
        ADD R7, R7, R1
NEXT5   ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT6
        ADD R7, R7, R1
NEXT6   ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT7
        ADD R7, R7, R1
NEXT7   ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT8
        ADD R7, R7, R1
NEXT8   ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT9
        ADD R7, R7, R1
NEXT9   ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT10
        ADD R7, R7, R1
NEXT10  ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT11
        ADD R7, R7, R1
NEXT11  ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT12
        ADD R7, R7, R1
NEXT12  ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT13
        ADD R7, R7, R1
NEXT13  ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT14
        ADD R7, R7, R1
NEXT14  ADD R1, R1, R1
        ADD R2, R2, R2
        AND R3, R0, R2
        BRz NEXT15
```

```
        ADD R7, R7, R1
NEXT15  ADD R1, R1, R1
        AND R3, R0, #-1
        BRzp NEXT16
        ADD R7, R7, R1
NEXT16  HALT
```

- **C++ code**

  Input & Output: Refer to lab1 L-version.

```cpp
int main() {
  if (b & 1 << 0) c += a << 0;
  if (b & 1 << 1) c += a << 1;
  if (b & 1 << 2) c += a << 2;
  if (b & 1 << 3) c += a << 3;
  if (b & 1 << 4) c += a << 4;
  if (b & 1 << 5) c += a << 5;
  if (b & 1 << 6) c += a << 6;
  if (b & 1 << 7) c += a << 7;
  if (b & 1 << 8) c += a << 8;
  if (b & 1 << 9) c += a << 9;
  if (b & 1 << 10) c += a << 10;
  if (b & 1 << 11) c += a << 11;
  if (b & 1 << 12) c += a << 12;
  if (b & 1 << 13) c += a << 13;
  if (b & 1 << 14) c += a << 14;
  if (b & 1 << 15) c += a << 15;
  return 0;
}
```

3) lab2: *fib*

- **Algorithm**

  Repeatedly calculate $F(n)$ using the recurrence relation

$$F(n) = [F(n-1) + 2F(n-3)] \mod 1024,$$

  To implement the modulo operation, consider the dividend in binary radix

$$b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0.$$

  Clearly, the remainder equals

$$000000b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0,$$

  for we only need the part less than 2^{10}, which can be obtained by anding the dividend with `0x3ff`.

  The algorithm runs in $O(n)$ time.

- **LC3 assembly code**

  Input: At the beginning, `R0` stores n.

  Output: `R7` stores $F(n)$ as the result.

```
IF      ADD R7, R7, #1
        ADD R0, R0, #-2
        BRn DONE
        BRz IF
        ADD R5, R5, #1
        ADD R6, R6, #1
        ADD R7, R7, #1
        LD R1, BITMASK
LOOP    ADD R2, R5, R5
        ADD R5, R6, #0
        ADD R6, R7, #0
        ADD R7, R7, R2
        AND R7, R7, R1
        ADD R0, R0, #-1
        BRp LOOP
DONE    HALT
BITMASK .FILL x3ff
```

- **C++ code**

  Input: At the beginning, a global variant `n` stores n.

  Output: `f` stores the result $F(n)$.

```cpp
int main() {
  f = n <= 1 ? 1 : 2;
  uint16_t g = 1, h = 1;
  while (n-- > 2) {
    uint16_t temp = f;
    f = (f + 2 * h) & 1023;
    h = g;
    g = temp;
  }
  return 0;
}
```

4) lab3: *fib-opt*

- **Algorithm**

  With Floyd's tortoise and hare algorithm, we find a minimal $\mu = 19$ and a minimal $\lambda = 128$ such that

$$\forall i > 0,\ \ F(\mu + i) = F(\lambda + i).$$

  Calculate the first $\mu + \lambda + 1 = 148$ elements of the sequence to form a table, and then

$$F(n) = \begin{cases} F[(n - 20) \mod 128] + 20, & n \geq 20, \\ F(n), & n < 20, \end{cases}$$

  which can get the result in $O(1)$ time.

- **LC3 assembly code**

  Input & Output: Refer to lab2 fibonacci.

```
        LEA R1, TABLE
        ADD R2, R0, #-10
        ADD R2, R2, #-10
        BRn OK
        LD R3, BITMASK
        AND R0, R2, R3
        ADD R0, R0, #10
        ADD R0, R0, #10
OK      ADD R1, R1, R0
        LDR R7, R1, #0
        HALT
BITMASK .FILL #127
TABLE   .FILL #1
        .FILL #1
        .FILL #2
        .FILL #4
        .FILL #6
        .FILL #10
        .FILL #18
        .FILL #30
        .FILL #50
        .FILL #86
        .FILL #146
        .FILL #246
        .FILL #418
        .FILL #710
        .FILL #178
        .FILL #1014
        .FILL #386
        .FILL #742
        .FILL #722
        .FILL #470
        .FILL #930
        .FILL #326
        .FILL #242
```

```
        .FILL #54
        .FILL #706
        .FILL #166
        .FILL #274
        .FILL #662
        .FILL #994
        .FILL #518
        .FILL #818
        .FILL #758
        .FILL #770
        .FILL #358
        .FILL #850
        .FILL #342
        .FILL #34
        .FILL #710
        .FILL #370
        .FILL #438
        .FILL #834
        .FILL #550
        .FILL #402
        .FILL #22
        .FILL #98
        .FILL #902
        .FILL #946
        .FILL #118
        .FILL #898
        .FILL #742
        .FILL #978
        .FILL #726
        .FILL #162
        .FILL #70
        .FILL #498
        .FILL #822
        .FILL #962
        .FILL #934
        .FILL #530
        .FILL #406
        .FILL #226
        .FILL #262
        .FILL #50
        .FILL #502
        .FILL #2
        .FILL #102
        .FILL #82
        .FILL #86
        .FILL #290
        .FILL #454
        .FILL #626
        .FILL #182
        .FILL #66
        .FILL #294
        .FILL #658
        .FILL #790
        .FILL #354
        .FILL #646
        .FILL #178
        .FILL #886
        .FILL #130
        .FILL #486
        .FILL #210
        .FILL #470
        .FILL #418
        .FILL #838
        .FILL #754
        .FILL #566
        .FILL #194
        .FILL #678
```

```
        .FILL #786
        .FILL #150
        .FILL #482
        .FILL #6
        .FILL #306
        .FILL #246
        .FILL #258
        .FILL #870
        .FILL #338
        .FILL #854
        .FILL #546
        .FILL #198
        .FILL #882
        .FILL #950
        .FILL #322
        .FILL #38
        .FILL #914
        .FILL #534
        .FILL #610
        .FILL #390
        .FILL #434
        .FILL #630
        .FILL #386
        .FILL #230
        .FILL #466
        .FILL #214
        .FILL #674
        .FILL #582
        .FILL #1010
        .FILL #310
        .FILL #450
        .FILL #422
        .FILL #18
        .FILL #918
        .FILL #738
        .FILL #774
        .FILL #562
        .FILL #1014
        .FILL #514
        .FILL #614
        .FILL #594
        .FILL #598
        .FILL #802
        .FILL #966
        .FILL #114
        .FILL #694
        .FILL #578
        .FILL #806
        .FILL #146
        .FILL #278
        .FILL #866
        .FILL #134
        .FILL #690
        .FILL #374
        .FILL #642
        .FILL #998
        .FILL #722
        .FILL #982
```

- **C++ code**

  Input & Output: Refer to lab2 fibonacci.

```cpp
const uint16_t data[148] = {
    1,   1,    2,   4,   6,   10,  18,   30,  50,  86,  146, 246, 418, 710,
    178, 1014, 386, 742, 722, 470, 930,  326, 242, 54,  706, 166, 274, 662,
    994, 518,  818, 758, 770, 358, 850,  342, 34,  710, 370, 438, 834, 550,
    402, 22,   98,  902, 946, 118, 898,  742, 978, 726, 162, 70,  498, 822,
    962, 934,  530, 406, 226, 262, 50,   502, 2,   102, 82,  86,  290, 454,
```

```
    626, 182,  66,  294, 658, 790, 354,  646, 178, 886, 130, 486, 210, 470,
    418, 838,  754, 566, 194, 678, 786,  150, 482, 6,   306, 246, 258, 870,
    338, 854,  546, 198, 882, 950, 322,  38,  914, 534, 610, 390, 434, 630,
    386, 230,  466, 214, 674, 582, 1010, 310, 450, 422, 18,  918, 738, 774,
    562, 1014, 514, 614, 594, 598, 802,  966, 114, 694, 578, 806, 146, 278,
    866, 134,  690, 374, 642, 998, 722,  982};
int main() {
  f = n < 20 ? data[n] : data[(n - 20 & 127) + 20];
  return 0;
}
```

5) lab4 task1: *rec*

- **Algorithm**
  Increment an integer from 0 to 5 by means of recursion.
- **LC3 assembly code**
  Output: The integer is in `R0`.

```
        LEA R2, STACK
        AND R0, R0, #0
        JSR SUB
        HALT
SUB     STR R7, R2, #0
        ADD R2, R2, #1
        ADD R0, R0, #1
        LD R1, COUNT
        ADD R1, R1, #-1
        ST R1, COUNT
        BRz RESTORE
        JSR SUB0
RESTORE ADD R2, R2, #-1
        LDR R7, R2, #0
        RET
STACK   .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
        .FILL x0
COUNT   .FILL x5
```

- **C++ code**
  Output: The integer is in a global variant `r0`.

```
uint16_t count = 5;
void sub() {
  r0++;
  if (--count) {
    sub();
  } else
    return;
}
int main() {
  r0 = 0;
  sub();
  return 0;
}
```

6) lab4 task2: *mod*

- **Algorithm**
  Modulo operation for a divisor 7 can be implemented with the following steps:

1. Divide the dividend N by 8. Let the quotient be Q and the remainder be R such that

$$N = 8Q + R \equiv Q + R \pmod 7.$$

2. Set N to $Q + R$.
3. Repeat the above steps until $0 \leq N < 7$, and then the remainder modulo 7 should be N.

In the first step, get the remainder modulo 8 by anding the dividend N with `0x0007`, and the quotient by shifting N right by 3 bits.

- **LC3 assembly code**

  Output: The remainder N is in `R1`.

```
          LD R1, DIVIDEND
LOOP      JSR DIV_8
          AND R2, R1, #7
          ADD R1, R2, R4
          ADD R0, R1, #-7
          BRp LOOP
          ADD R0, R1, #-7
          BRn DONE
          ADD R1, R1, #-7
DONE      HALT
DIV_8     AND R2, R2, #0
          AND R3, R3, #0
          AND R4, R4, #0
          ADD R2, R2, #1
          ADD R3, R3, #8
NEXT_BIT  AND R5, R3, R1
          BRz SKIP
          ADD R4, R2, R4
SKIP      ADD R2, R2, R2
          ADD R3, R3, R3
          BRnp NEXT_BIT
          RET
DIVIDEND  .FILL x120
```

- **C++ code**

  Output: The remainder N is in a global variant `n`.

```
uint16_t q;
void div_8() {
  q = 0;
  for (int i = 3; i < 16; i++)
    if (d & (1 << i)) q += 1 << (i - 3);
  return;
}
int main() {
  n = 288;
  do {
    div_8();
  } while ((n = q + (n & 7)) > 7);
  if (n == 7) n -= 7;
  return 0;
}
```

7) lab5: *prime*

- **Algorithm**

  To judge if a number is prime, check if it is divisible by any number from 2 to its square root. Due to the difficulty of calculating the square root, stop numerating the possible divisors when the square of the current one is greater than the dividend. We implement functions for multiplication, division, etc., to make the code more readable.

  For multiplication algorithm, we refer to lab1 L-version.

  For division algorithm, inspired by the pen-and-paper division of multi-digit decimal numbers, we have the algorithm for a binary radix. Suppose we are to divide N by D, placing the quotient in Q and the remainder in R. The following is the pseudo-code.

```
Q := 0                  -- Initialize quotient and remainder to zero
R := 0
for i := w - 1 .. 0 do  -- Where w is number of bits in N
  R := R << 1           -- Left-shift R by 1 bit
```

```
  R(0) := N(i)          -- Set the least-significant bit of R equal to bit i of the numerator
  if R ≥ D then
    R := R − D
    Q(i) := 1
  end
end
```

Multiplication and division both execute in $O(w)$ time, where w is the number of bits in an integer. Given $w = 16$, the time complexity for the whole algorithm will be $O(\sqrt{n})$, where n is the number to judge for.

- **LC3 assembly code**

  Input: The number is in `R0`.

  Output: The result is in `R1`, 1 for prime, 0 for not prime.

```
                JSR PUSH_R0
                JSR JUDGE
                JSR POP_R1
                HALT

JUDGE           ST R0, JUDGE_SAVE_R0
                ST R2, JUDGE_SAVE_R2
                ST R3, JUDGE_SAVE_R3
                ST R7, JUDGE_SAVE_R7

                JSR POP_R0
                AND R1, R1, #0
                ADD R1, R1, #1
                ADD R2, R1, #1
JUDGE_LOOP      JSR PUSH_R2
                JSR PUSH_R2
                JSR MUL
                JSR PUSH_R0
                JSR CMP
                JSR POP_R3
                BRp JUDGE_BREAK
                JSR PUSH_R0
                JSR PUSH_R2
                JSR MOD
                JSR POP_R3
                BRnp JUDGE_IF
                AND R1, R1, #0
                BR JUDGE_BREAK
JUDGE_IF        ADD R2, R2, #1
                BR JUDGE_LOOP
JUDGE_BREAK     JSR PUSH_R1

                LD R0, JUDGE_SAVE_R0
                LD R2, JUDGE_SAVE_R2
                LD R3, JUDGE_SAVE_R3
                LD R7, JUDGE_SAVE_R7
                RET

                JUDGE_SAVE_R0 .BLKW #1
                JUDGE_SAVE_R1 .BLKW #1
                JUDGE_SAVE_R2 .BLKW #1
                JUDGE_SAVE_R3 .BLKW #1
                JUDGE_SAVE_R7 .BLKW #1

; Subtraction
SUB             ST R0, SUB_SAVE_R0
                ST R1, SUB_SAVE_R1
                ST R7, SUB_SAVE_R7

                JSR POP_R1
                JSR POP_R0
                NOT R1, R1
                ADD R1, R1, #1
                ADD R0, R0, R1
```

```
                JSR PUSH_R0

                LD R0, SUB_SAVE_R0
                LD R1, SUB_SAVE_R1
                LD R7, SUB_SAVE_R7
                RET

SUB_SAVE_R0     .BLKW #1
SUB_SAVE_R1     .BLKW #1
SUB_SAVE_R7     .BLKW #1

; Multiplication
MUL             ST R0, MUL_SAVE_R0
                ST R1, MUL_SAVE_R1
                ST R2, MUL_SAVE_R2
                ST R3, MUL_SAVE_R3
                ST R7, MUL_SAVE_R7

                JSR POP_R1
                JSR POP_R0
                AND R2, R2, #0
                AND R3, R3, #0
                ADD R3, R3, #1
MUL_LOOP        AND R7, R0, R3
                BRz MUL_SKIP
                ADD R2, R2, R1
MUL_SKIP        ADD R1, R1, R1
                ADD R3, R3, R3
                BRnp MUL_LOOP
                JSR PUSH_R2

                LD R0, MUL_SAVE_R0
                LD R1, MUL_SAVE_R1
                LD R2, MUL_SAVE_R2
                LD R3, MUL_SAVE_R3
                LD R7, MUL_SAVE_R7
                RET

MUL_SAVE_R0     .BLKW #1
MUL_SAVE_R1     .BLKW #1
MUL_SAVE_R2     .BLKW #1
MUL_SAVE_R3     .BLKW #1
MUL_SAVE_R7     .BLKW #1

; Modulo
MOD             ST R0, MOD_SAVE_R0
                ST R1, MOD_SAVE_R1
                ST R2, MOD_SAVE_R2
                ST R3, MOD_SAVE_R3
                ST R4, MOD_SAVE_R4
                ST R7, MOD_SAVE_R7
                JSR POP_R1
                JSR POP_R0

                NOT R3, R1
                ADD R3, R3, #1
                AND R4, R4, #0
                ADD R4, R4, #-16
MOD_INIT_LOOP   AND R7, R0, #-1
                BRn MOD_INIT_BREAK
                ADD R0, R0, R0
                ADD R4, R4, #1
                BR MOD_INIT_LOOP
MOD_INIT_BREAK  AND R2, R2, #0
MOD_MAIN_LOOP   ADD R4, R4, #1
                BRp MOD_MAIN_BREAK
                ADD R2, R2, R2
```

```
                AND R7, R0, #-1
                BRzp MOD_SKIP
                ADD R2, R2, #1
MOD_SKIP        JSR PUSH_R1
                JSR PUSH_R2
                JSR CMP
                JSR POP_R5
                BRp MOD_IF
                ADD R2, R2, R3
MOD_IF          ADD R0, R0, R0
                BR MOD_MAIN_LOOP

MOD_MAIN_BREAK  JSR PUSH_R2
                LD R0, MOD_SAVE_R0
                LD R1, MOD_SAVE_R1
                LD R2, MOD_SAVE_R2
                LD R3, MOD_SAVE_R3
                LD R4, MOD_SAVE_R4
                LD R7, MOD_SAVE_R7
                RET

MOD_SAVE_R0     .BLKW #1
MOD_SAVE_R1     .BLKW #1
MOD_SAVE_R2     .BLKW #1
MOD_SAVE_R3     .BLKW #1
MOD_SAVE_R4     .BLKW #1
MOD_SAVE_R7     .BLKW #1

; Comparison
CMP             ST R0, CMP_SAVE_R0
                ST R1, CMP_SAVE_R1
                ST R2, CMP_SAVE_R2
                ST R7, CMP_SAVE_R7

                AND R2, R2, #0
                ADD R2, R2, #-1
                JSR POP_R1
                JSR POP_R0
                AND R7, R0, R1
                BRn CMP_SUB
                AND R7, R0, #-1
                BRn CMP_GREATER
                AND R7, R1, #-1
                BRn CMP_LESS
CMP_SUB         JSR PUSH_R0
                JSR PUSH_R1
                JSR SUB
                JSR POP_R0
                BRn CMP_LESS
                BRz CMP_ZERO
CMP_GREATER     ADD R2, R2, #1
CMP_ZERO        ADD R2, R2, #1
CMP_LESS        JSR PUSH_R0

                LD R0, CMP_SAVE_R0
                LD R1, CMP_SAVE_R1
                LD R2, CMP_SAVE_R2
                LD R7, CMP_SAVE_R7
                RET
CMP_SAVE_R0     .BLKW #1
CMP_SAVE_R1     .BLKW #1
CMP_SAVE_R2     .BLKW #1
CMP_SAVE_R7     .BLKW #1

; Stack
POP             ST R1, STACK_SAVE_R1
                ST R2, STACK_SAVE_R2
```

```
                ST R5, STACK_SAVE_R5
                ST R6, STACK_SAVE_R6

                LD R1, STACK_EMPTY
                LD R6, STACK_POINTER
                AND R5, R5, #0
                ADD R2, R6, R1
                BRnp POP_SUCCESS
                ADD R5, R5, #1
                BR POP_OK
POP_SUCCESS     LDR R0, R6, #0
                ADD R6, R6, #1
POP_OK          ST R5, STACK_FAIL_FLAG
                ST R6, STACK_POINTER

                LD R1, STACK_SAVE_R1
                LD R2, STACK_SAVE_R2
                LD R5, STACK_SAVE_R5
                LD R6, STACK_SAVE_R6
                RET

PUSH            ST R1, STACK_SAVE_R1
                ST R2, STACK_SAVE_R2
                ST R5, STACK_SAVE_R5
                ST R6, STACK_SAVE_R6

                LD R1, STACK_FULL
                LD R6, STACK_POINTER
                AND R5, R5, #0
                ADD R2, R6, R1
                BRnp PUSH_SUCCESS
                ADD R5, R5, #1
                BR PUSH_OK
PUSH_SUCCESS    ADD R6, R6, #-1
                STR R0, R6, #0
PUSH_OK         ST R5, STACK_FAIL_FLAG
                ST R6, STACK_POINTER

                LD R1, STACK_SAVE_R1
                LD R2, STACK_SAVE_R2
                LD R5, STACK_SAVE_R5
                LD R6, STACK_SAVE_R6
                RET

STACK_FAIL_FLAG .FILL #0
STACK_POINTER   .FILL x4000
STACK_EMPTY     .FILL x-4000
STACK_FULL      .FILL x-3FF0
STACK_SAVE_R1   .BLKW #1
STACK_SAVE_R2   .BLKW #1
STACK_SAVE_R5   .BLKW #1
STACK_SAVE_R6   .BLKW #1

POP_R0          ST R7, STACK_R_SAVE_R7
                JSR POP
                LD R7, STACK_R_SAVE_R7
                ADD R0, R0, #0
                RET

POP_R1          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R1, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R1, R1, #0
                RET
```

```
POP_R2          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R2, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R2, R2, #0
                RET

POP_R3          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R3, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R3, R3, #0
                RET

POP_R4          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R4, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R4, R4, #0
                RET

POP_R5          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R5, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R5, R5, #0
                RET

POP_R6          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R6, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R6, R6, #0
                RET

PUSH_R0         ST R7, STACK_R_SAVE_R7
                JSR PUSH
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R1         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R1, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R2         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R2, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET
```

```
PUSH_R3         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R3, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R4         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R4, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R5         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R5, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R6         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R6, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

STACK_R_SAVE_R0 .BLKW #1
STACK_R_SAVE_R7 .BLKW #1
```

- **C++ code**

  Input: The number is in a global variant `r0`.

  Output: The result is in `r1`, 1 for prime, 0 for not prime.

```cpp
uint16_t mul(uint16_t a, uint16_t b) {
  uint16_t c = 0;
  for (int i = 0; i < 16; i++)
    if (b & 1 << i) c += a << i;
  return c;
}
uint16_t mod(uint16_t n, uint16_t q) {
  int16_t w = 15;
  uint16_t r = 0;
  while (!(n & 1 << w)) w--;
  while (w >= 0) {
    r <<= 1;
    if (n & 1 << w) r += 1;
    if (r >= q) r -= q;
    w--;
  }
  return r;
}
uint16_t judge(uint16_t r0) {
  for (uint16_t i = 2; mul(i, i) <= r0; i++)
    if (mod(r0, i) == 0) return r1 = 0;
  return r1 = 1;
}
int main() {
  r1 = judge(r0);
  return 0;
}
```

Questions

1) How do you evaluate the performance of your high-level language programs?

- First, analyse the theoretical efficiency of the algorithm from two aspects:
  1. Time complexity. Count the number of instructions involved in the algorithm.
  2. Space complexity. Measure the size of extra memory required by the algorithm.

  Typically, both the worst case or the average case should be considered, and we often use Big-O notation to compare different algorithms.
- However, the runtime performance of a program depends on the data set, the hardware, the compiler, the language and so forth. Hence, it is necessary to choose a fixed environment so that comparison between different programs will be meaningful. To be specific, we may disable compiler optimization, turn off debug options, run the programs on a certian hardware platform and test a sufficient data set that covers almost all cases of output.

  Some typical metrics for runtime performance are:
  1. The correctness.
  2. The time taken.
  3. The memory allocated.
  4. The usage of particular instructions.
  5. The frequency and duration of branches and function calls.
  6. ...

  For example, to measure the time taken by a c++ program, we can call system functions like `clock()` or `time()` to calculate the time in the code itself.

  More often, we apply an existing performance analysis tool, or a profiler to measure the runtime performance of a program. It collects useful data including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, performance counters and so forth.

  For c++ program, Gcov, provided by GCC suite, comes as a good choice for source code coverage analysis. To enable coverage testing, the c++ program should be compiled with options like `-fprofile-arcs`, `-ftest-coverage`. After execution, the coverage data can then be obtained by `gcov` command.

2) Why is a high-level language easier to write than LC3 assembly?

1. When writing in a high-level language, we mainly focus on abstract concepts such as variables, expressions, arrays, objects, functions and are detached from the machine. In LC3 assembly language, however, what we deal with, like registers and memory addresses, is directly related to the microprocesser, which often makes the code less readable and more hard to maintain.
2. LC3 assembly language only provides a very limited set of machine's native instructions and pseudo commands, while a high-level language goes far beyond, providing features like string handling, file input/output, object-oriented programming, etc.
3. High-level programming can amplify instructions and trigger a lot of arithmetic operations and data movements in the background, without programmers' knowledge on the microarchitecture. The responsibility and power of executing instructions have been handed over to the machine. What we concerns about is usability rather than optimal efficiency of basic instructions.

3) What instructions do you think need to be added to LC3?

To simplify programming in LC3, the following instructions can be taken into consideration:

1. More arithmetic instructions, such as

- `SUB`, for subtraction,
- `MUL`, for multiplication,
- `DIV`, for unsigned division,
- `MOD`, for unsigned modulo,
- * `INC`, for increment by 1,
- * `DEC`, for decrement by 1,
- `NEG`, for two's complement negation,
- `CMP`, for comparison between two unsigned integers,
- `SHL`, for left-shift by any bits,
- `SHR`, for right-shift by any bits.

2. More logical instructions, such as

- `OR`, for bitwise or,
- `XOR`, for bitwise exclusive or.

3. More data movement instructions, such as

- `PUSH`, for pushing a register onto the stack,
- `POP`, for popping a register from the stack,
- * `MOVE`, for moving data from one register to another,
- `XCHG`, for exchanging two registers.

Note that the instructions denoted with `*` can be implemented in a single line, but still worth to be added for the readability of code.

4) Is there anything you need to learn from LC3 for the high-level language you use?

1. Most high-level languages implicitly use a call stack to implement function calls and returns. When code in high-level language is compiled into assembly language, there will be a few lines of code at the beginning of a function, called function prologue, which prepare the stack and registers for use within the function. Similarly, function epilogue appears at the end of the function, and restores the stack and registers to the state they were in before the function was called. These hidden operations will cost a great deal of time or cause a lot of extra memory usage (even overflow) if overused.
   Be careful about overuse of function calls. Replace them with macros or inline functions if it is acceptable. Consider using a loop before using a recursive function instead.
2. Memory needs to be managed, taken care of and most importantly, not forgotten. High-level languages hide the details of automatic memory management, but sometimes we need to operate on memory manually (for example, to avoid copying or exchanging large scale data, or to implement a data structure). In such cases, we should be cautious against violations of memory safety or memory leaks.