EE3450 Project #1: MIPS Assembly Programming

Chi-Ming Lee

1 Objective

In this project, you will implement several classical algorithms that determine the Fibonacci number[1] with MIPS assembly. In addition to coding, we will use MARS[2] as the simulation tool to perform analysis and comparison on these algorithms. The definition of Fibonacci number is shown below:

$$F_n = \begin{cases} 0 & (n = 0) \\ 1 & (n = 1) \\ F_{n-1} + F_{n-2} & (n > 1) \end{cases}$$

We will enumerate five algorithms for you. For each of them, please do the following:

a. Use the described method to solve the Fibonacci number with C.

b. Use the described method to solve the Fibonacci number with MIPS assembly and verify your result with the C version.

c. Organize statistics from MARS and use your favorite tool (Excel, Matlab, gnuplot, etc.) to **plot a graph** that illustrates relationship between input size and total instruction count.

All your programs need to be explained explicitly and carefully with **comments in source codes** because they are the important part of grading. **Do not copy your codes into the report** but instead discuss your insight into algorithms or the key idea of your implementation.

After you are done with these algorithms and plots, you must perform analysis and comparison on them with various metrics in your report. For examples:

- Complexity
- Code size
- Instruction type distribution

You might further discuss how an algorithm trade off among these (or more) metrics, what is the distribution of instruction types in an algorithm (this affects instruction per cycle), and so on. There is no standard template for this report, but you need to **make a conclusion** (e.g., what you have learned or accomplished) in the end of it. Note that **coding (including comments) and the report will take identical proportion** in this project.

2 Problem Statements

2.1 Problem A: Iterative Method

The iterative method begins with the determined part of the problem (i.e. F_0 and F_1), and approaches the final result step by step as follows:

$$
\begin{aligned}
& F_0 = 0, F_1 = 1 \\
\rightarrow\ & F_2 = F_1 + F_0 = 1 + 0 = 1 \\
\rightarrow\ & F_3 = F_2 + F_1 = 1 + 1 = 2 \\
\rightarrow\ & F_4 = F_3 + F_2 = 2 + 1 = 3 \\
& \ldots.
\end{aligned}
$$

You will need a loop to implement such iterations in your program. To illustrate this method more concretely, let's take an example from another problem, Factorial number, which is defined below:

$$
n! = \begin{cases} 1 & (n = 0) \\ \prod_{k=1}^{n} k & (n > 0) \end{cases}
$$

The iterative-method sample codes solving Factorial are shown below:

```c
#include<stdio.h>
int fac_ite(int n){
    if( n == 0 )
        return 1;
    int i;
    for(i = n-1; i > 1; i--)
        n = n * i;
    return n;
}
int main(){
    int n, fac;
    scanf("%d", &n);
    fac = fac_ite(n);
    printf("%d\n",fac);
    return 0;
}
```

fac_ite.c

```
.data
msg_str:        .asciiz "Enter some number: "

.text

.globl main

main:
    la      $a0, msg_str        # load the address of the message "Enter some Number: "
    li      $v0, 4              # prepare for syscall 4, printing string to the user
    syscall                     # syscall 4
    li      $v0, 5              # prepare for syscall 5, reading integer from the user
    syscall                     # syscall 5
    move    $a0, $v0            # Now input integer is in $v0. We cocy it to $a0 for function call
    jal     fac                 # call "fac" function and jump to fac tag
    move    $a0, $v0            # Now the result of fac(n) is in $v0. Copy it to $a0 for syscall 1
    li      $v0, 1              # prepare for syscall 1, printing an integer
```

```
    syscall                     # syscall 1
    li      $v0, 10             # prepare for syscall 10, finish
    syscall

fac:
    bne     $a0, $zero, loophead# If input is 0 zero, return 1. If not, go to the loop
    li      $v0, 1
    jr      $ra                 # return value 1

loophead:
    move    $v0, $a0            # copy input to $v0 (prod). we will keep multiplying it until loop ends
    li      $t0, 1              # load value 1 to $t0 as loop end condition
loopbody:
    addi    $a0, $a0, -1        # i--
    mul     $v0, $v0, $a0       # n = n * i
    bne     $a0, $t0, loopbody  # if $a0 != 1, iteration keep going
    jr      $ra                 # return final result
```

fac_ite.asm

You might reference the samples to implement your Fibonacci programs.

2.2 Problem B: Recursive Method

The recursive method[3] solves the problem by partitioning it into sub-problems level by level, and then combines sub-solutions into the final result. Procedure call in programming language provides us an intuitive way to implement such a recursive method. The programmer can partition the problem by calling the same procedure with smaller input, and then combine values into the final solution. Please make sure you are familiar with MIPS calling convention [4] before doing this problem. Again, we take factorial number as an example with its recursive form:

$$n! = \begin{cases} 1 & (n = 0) \\ n \times (n-1)! & (n > 0) \end{cases}$$

Sample codes are shown below:

```c
#include <stdio.h>
int fac_rec(int n){
    if( n == 0)
        return 1;
    else
        return n * fac_rec(n-1);
}
int main(){
    int n, fac;
    scanf("%d", &n);
    fac = fac_rec(n);
    printf("%d\n", fac);
    return 0;
}
```

fac_rec.c

```
.data
msg_str:        .asciiz "Enter some Number: "

.text

.globl main

```

```
main:
    la      $a0, msg_str        # load the address of the message "Enter some Number: "
    li      $v0, 4              # prepare for syscall 4, printing string to the user
    syscall                     # syscall 4
    li      $v0, 5              # prepare for syscall 5, reading integer from the user
    syscall                     # syscall 5
    move    $a0, $v0            # Now input integer is in $v0. We cocy it to $a0 for function call
    jal     fac                 # call "fac" function and jump to fac tag
    move    $a0,$v0             # Now the result of fac(n) is in $v0. Copy it to $a0 for syscall 1
    li      $v0,1               # prepare for syscall 1, printing an integer
    syscall                     # syscall 1
    li      $v0,10              # prepare for syscall 10, finish
    syscall
fac:
    beq     $a0,$zero,ret_one   # branch to "ret_one" if input is 0

    addi    $sp,$sp,-8          # make room for stack push. we must do this before recursive call
    sw      $a0,0($sp)          # push input n to the stack
    sw      $ra,4($sp)          # push return address to the stack
    addi    $a0,$a0,-1
    jal     fac                 # recursive call
    lw      $t0,0($sp)          # pop input n back from the stack
    lw      $ra,4($sp)          # pop return address from the stack
    addi    $sp,$sp,8           # restore the stack

    mul     $v0,$v0,$t0         # do n! = (n-1)! * n, and prepare value n! for return
    j       ret                 # exit procedure
ret_one:
    li      $v0,1               # prepare value 1 for return
ret:
    jr      $ra                 # return
```

fac_rec.asm

You might reference the samples to implement your Fibonacci programs.

2.3 Problem C: Tail Recursion

Suppose you have completed problem B, you might notice its instruction count grows significantly with input size. To address this issue, programmers might use "tail recursion" to improve the efficiency of the original recursion. Please refer to [5] and discuss how the tail recursion out-performs the original one in Problem B. The pseudo code of tail recursion that solves Fibonacci number is shown in the Algorithm 1.

Algorithm 1 Tail Recursion Solving Fibonacci

Input: n, a (set as 0 when first call), b (set as 1 when first call)
Output: Fibonacci Number of n

if $n == 0$ **then**
    return a
else
    return Tail Recursion Solving Fibonacci($n-1$, b, $a+b$)
end if

2.4 Problem D: Q Matrix

The Q matrix method[6] accelerates finding Fibonacci number by manipulating matrix multiplication as below:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n + F_{n-1} & F_{n-1} + F_{n-2} \\ F_n & F_{n-1} \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \tag{2}$$

Repeat the above operation k times, (1) becomes:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k \times \begin{bmatrix} F_{(n+1)-k} & F_{n-k} \\ F_{n-k} & F_{(n-1)-k} \end{bmatrix} \tag{3}$$

In (3) let:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = Q \tag{4}$$

$$k = n - 1 \tag{5}$$

We have:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = Q^{n-1} \times \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} \tag{6}$$

$$= Q^{n-1} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \tag{7}$$

$$= Q^n \tag{8}$$

By this formula, we can get F_n directly by finding Q^n. Nevertheless, multiplying with Q n times is not an efficient approach. Instead, the Q matrix algorithm solves Q^n by finding square root of it recursively. Let's take an example of $n = 16$

$$Q^{16} = Q^8Q^8 = (Q^4Q^4)(Q^4Q^4) = ...$$

However, the power of Q will be an odd number very likely. In such a scenario, you will need some trivial work to partition it carefully. Let's take another example of $n = 5$

$$Q^5 = Q^3Q^2 = (Q^2Q^1)(Q^1Q^1) = [(Q^1Q^1)Q^1](Q^1Q^1)$$

The pseudo code of this method is organized in Algorithm 2.

Algorithm 2 Q Matrix Solving Fibonacci

Input: n (Integer), Q^1 (Matrix)
Output: Fibonacci Number of n

$Q^n \leftarrow$ Find_Q_Matrix(n)
$F_n \leftarrow Q^n[0][1]$
return F_n

function Find_Q_Matrix(k)
  if $k == 1$ **then return** Q^1
  else if $k \in$ Even number **then**
    $Q^{\frac{k}{2}} \leftarrow$ Find_Q_Matrix($\frac{k}{2}$)
    return $Q^{\frac{k}{2}} \times Q^{\frac{k}{2}}$
  else if $k \in$ Odd number **then**
    $Q^{\lfloor\frac{k}{2}\rfloor} \leftarrow$ Find_Q_Matrix($\lfloor\frac{k}{2}\rfloor$)
    $Q^{\lfloor\frac{k}{2}\rfloor+1} \leftarrow$ Find_Q_Matrix($\lfloor\frac{k}{2}\rfloor + 1$)
    return $Q^{\lfloor\frac{k}{2}\rfloor} \times Q^{\lfloor\frac{k}{2}\rfloor+1}$
  end if
end function

Here are two hints for this problem:

a. We usually need 3-level loops to implement matrix multiplication [7]. In this case, however, unrolling loops (i.e., writing every single addition and multiplication explicitly instead of using loops) might makes it easier because only 2 by 2 matrices are involved.

b. Before doing matrix multiplication, we must allocate memory for the result matrix. This can be done by the function "malloc"[8] in C and "syscall 9"[9] in MIPS.

Provided template C and MIPS codes with mmul (matrix multiplication) functions that can be used directly might be helpful in this problem, but **they are not golden implementation.** You might improve it too get bonus.

```c
#include <stdio.h>
#include <stdlib.h>
//=========================================
// We use an array to represent 2x2 matrix
// Q[0][0] <-> Q[0]  , Q[0][1] <-> Q[1]
// Q[1][0] <-> Q[2]  , Q[1][1] <-> Q[3]
//=========================================

int Q1[4] = {1, 1, 1, 0};        //  Constant matrix Q1

int* mmul(int* m1, int* m2){
//=========================================
//     **mmul implemented for you**
// Please use arrays to represent matrices
// You can use this function as follows:
// int* C = mmul(A, B), to calculate C = AB
// where A, B, C are 2 by 2 matrices
//=========================================
    int* r = (int*)malloc(sizeof(int) * 4);
    r[0] = m1[0] * m2[0] + m1[1] * m2[2];
    r[1] = m1[0] * m2[1] + m1[1] * m2[3];
    r[2] = m1[2] * m2[0] + m1[3] * m2[2];
    r[3] = m1[2] * m2[1] + m1[3] * m2[3];
    return r;
```

```c
}

int main(){
    //=====================================
    // Sample program demonstrating
    // how to determine Q3 with mmul
    // ===================================
    int* Q2 = mmul(Q1, Q1);
    int* Q3 = mmul(Q1, Q2);
    printf("Q3 = \n");
    printf("%d %d\n", Q3[0], Q3[1]);
    printf("%d %d\n", Q3[2], Q3[3]);
    return 0;
}
```

fib_template.c

```
.data
Q1:             .word 1, 1, 1, 0 # Q1[0][0], Q1[0][1], Q1[1][0], Q1[1][1]
msg_str:        .asciiz "Q3 =\n"
.text
.globl main
main:
#================================
# sample program demonstrating
# how to determine Q3 with mmul
#================================
    la  $a0, Q1                 # Q1 be the first input matrix
    la  $a1, Q1                 # Q1 be the second input matrix
    jal mmul                    # $v0 = Q2 = Q1xQ1
    move    $a0, $v0            # $v0, i.e, Q2, be the first input matrix
    la  $a1, Q1                 # Q1 be the second input matrix again
    jal     mmul                # $v0 = Q3 = Q2xQ1
    move    $t0, $v0            # copy $v0, i.e, Q3 to another register
    la  $a0, msg_str            # load the address of the message "Q3=\n"
    li  $v0, 4                  # print string
    syscall
    lw  $a0, 0($t0)             # load Q3[0][0] from memory to argument register
    li  $v0, 1                  # prepare to print integer
    syscall
    li  $a0, 32                 # prepare to print ascii code 32: white space
    li  $v0, 11                 # print char: white space
    syscall
    lw  $a0, 4($t0)             # load Q3[0][1] from memory to argument register
    li  $v0, 1                  # print integer: Q3[0][1]
    syscall
    li  $a0, 10                 # prepare to print ascii code 10: change line
    li  $v0, 11                 # print char: change line
    syscall
    lw  $a0, 8($t0)             # load Q3[1][0] from memory to argument register
    li  $v0, 1                  # print integer Q3[1][0]
    syscall
    li  $a0, 32                 # prepare to print ascii code 32: white space
    li  $v0, 11                 # print char: white space
    syscall
    lw  $a0, 12($t0)            # load Q3[1][1] from memory to argument register
    li  $v0, 1                  # print integer: Q3[1][1]
    syscall
    li  $v0, 10
    syscall
mmul:
##### mmul implemented for you #####
# you can use this as follows:
# la    $a0, A   ~load address of A
# la    $a1, B   ~load address of B
```

```
# jr     mmul      ~do multiplication
# The return register $v0 will hold
# address of the result matrix C=AxB.
# You can access matrix C by:
# lw $t0, 0($v0) ~load C[0][0] to $t0
# lw $t0, 4($v0) ~load C[0][0] to $t0
# lw $t0, 8($v0) ~load C[0][0] to $t0
# lw $t0,12($v0) ~load C[0][0] to $t0
#==========================================
    move    $t0, $a0            # int* m1
    move    $t1, $a1            # int* m2
    li   $a0, 16                # request for 16 byte location to hold result matrix
    li   $v0, 9                 # malloc system call
    syscall
                                # Calculate C[0]
    lw   $t2, 0($t0)            # load A[0]
    lw   $t3, 4($t0)            # load A[1]
    lw   $t4, 0($t1)            # load B[0]
    lw   $t5, 8($t1)            # load B[2]
    mul $t7, $t2, $t4           #m1[0]*m2[0]
    mul $t8, $t3, $t5           #m1[1]*m2[2]
    add $t7, $t7, $t8           #r[0]
    sw   $t7, 0($v0)

    lw   $t4, 4($t1)            #m2[1]
    lw   $t5, 12($t1)           #m2[3]
    mul $t7, $t2, $t4
    mul $t8, $t3, $t5
    add $t7, $t7, $t8           #r[1]
    sw   $t7, 4($v0)

    #r[3]
    lw   $t2, 8($t0)            #m1[2]
    lw   $t3, 12($t0)           #m1[3]
    mul $t7, $t2, $t4
    mul $t8, $t3, $t5
    add $t7, $t7, $t8           #r[3]
    sw   $t7, 12($v0)

    #r[2]
    lw   $t4, 0($t1)            #m2[0]
    lw   $t5, 8($t1)            #m2[2]
    mul $t7, $t2, $t4
    mul $t8, $t3, $t5
    add $t7, $t7, $t8           #r[2]
    sw   $t7, 8($v0)

    jr   $ra
```

fib_template.asm

2.5 Problem E: Fast Doubling Method

The fast doubling[10] method uses a bottom-up approach to solve A^n rather than a top-down one used in the Q matrix method. The idea of fast doubling comes from Q matrix:

Let:

$$\begin{bmatrix} F_{2n+1} \\ F_{2n} \end{bmatrix} = \begin{bmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{9}$$

Use the equation of Q matrix:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \tag{10}$$

(9) can be written as:

$$\begin{aligned}
\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} \begin{bmatrix} 1 \\ 0 \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} && (11) \\
&= \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} && (12) \\
&= \begin{bmatrix} F_{n+1}^2 + F_n^2 \\ F_n(F_{n+1} + F_{n-1}) \end{bmatrix} && (13)
\end{aligned}$$

In (13) let:

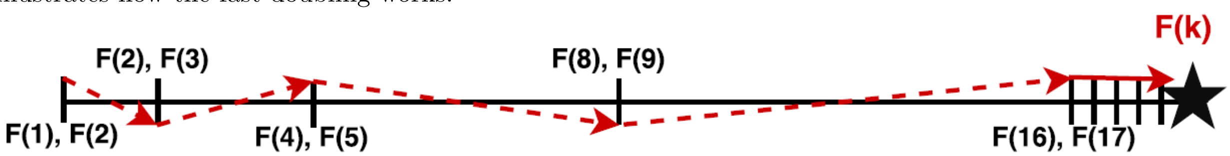
$$F_{n-1} = F_{n+1} - F_n \tag{14}$$

We get the fast doubling formula:

$$F_{2n+1} = F_{n+1}^2 + F_n^2 \tag{15}$$

$$F_{2n} = F_n(2F_{n+1} - F_n) \tag{16}$$

With this formula, we can find F_{2n+1} and F_{2n} simply by manipulating F_{n+1} and F_n. The figure below illustrates how the fast doubling works:



The fast doubling begins with $F_1 = 1$ and $F_2 = 1$, and doubles its index iteration by iteration. As soon as the index closes enough with the target, the algorithm switches to the iterative method and gets the final result.
The pseudo code is organized in the Algorithm 3.

Algorithm 3 Fast Doubling Solving Fibonacci

Input: n (Integer)
Output: Fibonacci Number of n

if $n == 0$ **then return** 0
end if
$i \leftarrow 1$; $F_i \leftarrow 1$; $F_{i+1} \leftarrow 1$
while $i < n$ **do**
  if $i \leq \frac{n}{2}$ **then**
    $F_{2i+1} \leftarrow F_i^2 + F_{i+1}^2$; $F_{2i} \leftarrow F_i \times (2F_{i+1} - F_i)$
    $F_i \leftarrow F_{2i}$; $F_{i+1} \leftarrow F_{2i+1}$
    $i \leftarrow i \times 2$
  else
    $F_{i+2} \leftarrow F_{i+1} + F_i$
    $F_i \leftarrow F_{i+1}$
    $F_{i+1} \leftarrow F_{i+2}$
    $i \leftarrow i + 1$
  end if
end while
return F_i

3 What to submit

This is the most important section. Your codes might fail if you ignore anything in this section.

- Five C codes, fibA.c, fibB.c, fibC.c, fibD.c and fibE.c, described as section 2
  You must make sure all your C codes can be tested in the terminal of EE workstation as follows:
  a. In the folder of your C codes
  b. Enter: gcc fibA.c
  c. Enter: ./a.out
  d. Enter: 10
  e. Terminal shows nothing but ”55”

  Please do not print anything else on the terminal.
- Five MIPS codes, fibA.asm, fibB.asm, fibC.asm, fibD.asm and fibE.asm, described as section 2
  You must make sure all your MIPS codes can be tested in MARS follows:
  a. Load fibA.asm
  b. Run
  c. Enter: 10
  d. The console shows ”55” and the system information ”– program is finished running –”
- A PDF report named ”report.pdf”.

Please **zip these 11 items directly without any folder**, and name your zip file as **”ID+pj1.zip”**. For example, mine is ”106061552pj1.zip”.

References

[1] Wikipedia.org, “Fibonacci number,” http://en.wikipedia.org/wiki/Fibonacci_number.

[2] K. Vollmar, “Mars: Mips assembler and runtime simulator,” http://courses.missouristate.edu/kenvollmar/mars/.

[3] P.-Y. Ting, “Recursive function design,” http://squall.cs.ntou.edu.tw/cprog/materials/recursive.html.

[4] Y. Hsu, “Chapter2: Instructions part b, ee3450,” http://lms.nthu.edu.tw/sys/read_attach.php?id=709260.

[5] stackoverflow.com, “What is tail recursion,” http://stackoverflow.com/questions/33923/what-is-tail-recursion.

[6] Mathworld, “Fibonacci q-matrix,” http://mathworld.wolfram.com/FibonacciQ-Matrix.html.

[7] dailyfreecode.com, “Matrix multiplication function,” http://www.dailyfreecode.com/code/matrix-multiplication-function-2225.aspx.

[8] cppreference.com, “C dynamic memory management: malloc,” http://en.cppreference.com/w/c/memory/malloc.

[9] stackoverflow.com, “How does mips allocate memory for arrays,” http://stackoverflow.com/questions/19612459/mips-how-does-mips-allocate-memory-for-arrays-in-the-stack.

[10] stackexchange.com, “Fast doubling proof,” http://math.stackexchange.com/questions/1124590/need-help-understanding-fibonacci-fast-doubling-proof.