جامعة نيويورك أبوظبي

NYU | ABU DHABI

Computer Organization and Architecture

ENGR-UH 3511

Lab 2

Fall 2024

MIPS assembly, recursion and the SPIM simulator

Yamsine Elsisi ~ yme2013

Matrix Multiplication

A matrix multiplication algorithm for a 2x2 matrix was written in C and then translated to MIPS through several iterations to get the code running in assembly.

Step 1: C-code

C matrixmultiply.c > main()

```c
#include <stdio.h>

void multiplyMatrices(int firstMatrix[2][2], int secondMatrix[2][2], int resultMatrix[2][2]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            resultMatrix[i][j] = 0;
            for (int k = 0; k < 2; k++) {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}

int main() {
    int firstMatrix[2][2] = {{1, 2}, {6, 4}};
    int secondMatrix[2][2] = {{3, 2}, {1, 8}};
    int resultMatrix[2][2];

    multiplyMatrices(firstMatrix, secondMatrix, resultMatrix);

    printf("Result of matrix multiplication:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", resultMatrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Figure 1: matrix multiplication in C

- The first two loops (i and j) iterate over the rows and columns of the result matrix.i is used as the row index of the first matrix, and j is used as to the column index of the second matrix. The inner loop (k) is used to compute the dot product between the corresponding row of the firstMatrix and column of the secondMatrix. Each element resultMatrix[i][j] is computed by summing the product of the corresponding elements in the i-th row of firstMatrix and the j-th column of secondMatrix.

- *Formula : resultMatrix[i][j]=k=0∑1(firstMatrix[i][k]×secondMatrix[k][j])*

- In the main, the matrices are created and two loops go through all the elments of the resultMatrix to print it out.

Step 2: Changing For-loops to while Loops

C matrixpultiply2.c > main()

```c
#include <stdio.h>

void multiplyMatrices(int firstMatrix[2][2], int secondMatrix[2][2], int resultMatrix[2][2]) {

    int i = 0;
    while (i<2)
    {
        int j=0;

        while(j<2)
        {
            resultMatrix[i][j] = 0;

           int k=0;

            while(k<2)
            {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
                k++;
            }

            j++;
        }

        i++;
    }
}
```

Figure 2: Switching to while-loop

- This makes it easier to switch to goto and if-statements.

Step 3: Changing while Loops to goto and if-statements

matrixmultiply3.c > multiplyMatrices(int [2][2], int [2][2], int [2][2])

```c
#include <stdio.h>

void multiplyMatrices(int firstMatrix[2][2], int secondMatrix[2][2], int resultMatrix[2][2]) {

    int i = 0;
    loop1:
    if (i >= 2) goto end1;
    {
        int j=0;
        loop2:
        if (j>=2) goto end2;
        {
            resultMatrix[i][j] = 0;
           int k=0;

            loop3:
            if (k>=2) goto end3;
            {
                resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
                k++;
                goto loop3;

            }
            end3:;
            j++;
            goto loop2;
        }
        end2:;
        i++;
    }
    goto loop1;
end1:;
```

matrixmultiply3.c > main()

```c
void multiplyMatrices(int firstMatrix[2][2], int secondMatrix[2][2], int resultMatrix[2][2]) {
    goto loop1;
end1:;
return;
}

int main() {
    int firstMatrix[2][2] = {{1, 2}, {6, 4}};
    int secondMatrix[2][2] = {{3, 2}, {1, 8}};
    int resultMatrix[2][2];
    multiplyMatrices(firstMatrix, secondMatrix, resultMatrix);
    printf("Result of matrix multiplication:\n");
    int i = 0;
    loop4:
    if (i>=2) goto end4;
    {
        int j = 0;
        loop5:
        if (j>=2) goto end5;
        {
            printf("%d ", resultMatrix[i][j]);
            j++;
            goto loop5;
        }
        end5:;
        printf("\n");
        i++;
        goto loop4;
    }
    end4:;
    return 0;
}
```

Figure 3: Switching to goto and if-statements

- The three loops of the multiplyMatricies function use a goto statement to loop until k, j then i and larger than 2, then they end by using a goto end. The same concept is used to print the result matrix.

Step 4: Writing in assembly

- The instructions are then written in assembly using the variable names as register names to make it easier to understand, then written again using real register names.

Results:

First Matrix:

$$\begin{bmatrix} 1 & 2 \\ 6 & 4 \end{bmatrix}$$

Second Matrix:

$$\begin{bmatrix} 3 & 2 \\ 1 & 8 \end{bmatrix}$$

First Row, First Column:

(1×3) + (2×1) = 3 + 2 = 5
resultMatrix[0][0] = 19

First Row, Second Column:

(1×2) + (2×8) = 2 +1 6 = 18
resultMatrix[0][1] = 22

Second Row, First Column:

(6×3) + (4×1) = 18 + 4 = 22
resultMatrix[1][0] = 43

Second Row, Second Column:

(6×2) + (4×8) = 12 + 32 = 44
resultMatrix[1][1] = 50

Result Matrix:

$$\begin{bmatrix} 5 & 18 \\ 22 & 44 \end{bmatrix}$$

MIPS Code:

```
matrixMultiply.s
.data
firstMatrix: .word 1, 2, 6, 4
secondMatrix: .word 3, 2, 1, 8
resultMatrix: .space 16  # Allocate space for 2x2 matrix

newline: .asciiz "\n"  # Newline character for printing
space: .asciiz " "     # Space charcter for printing


.text
.globl main

main:
    # Initialize base addresses of matrices
    la   $a0, firstMatrix   # Load address of firstMatrix into $a0
    la   $a1, secondMatrix  # Load address of secondMatrix into $a1
    la   $t0, resultMatrix  # Load address of resultMatrix into $t0

    addi $sp, $sp, -8   # Allocate space on the stack
    sw   $ra, 0($sp)    # Save return address on the stack

    li   $t1, 0   # initialize i to 0
    li   $t2, 2   # matrix size is 2

loop1:
    bge  $t1, $t2, end1   # if i >= size, end loop1
    li   $t3, 0           # j = 0, initialize j to 0

loop2:
    bge  $t3, $t2, end2   # if j >= size, end loop2

    mul  $t4, $t1, 2     # rowOffset = i * 2
    add  $t5, $t4, $t3   # index = i * 2 + j
    sll  $t6, $t5, 2     # byteOffset = (i * 2 + j) * 4
    add  $t7, $t0, $t6   # elementAddr = &resultMatrix[i][j]
    ori  $t8, $zero, 0   # temp = 0
    sw   $t8, 0($t7)     # resultMatrix[i][j] = 0

    li   $t9, 0  #initialize k to 0

loop3:
    bge  $t9, $t2, end3  # if k >= size, exit loop3

    # Load firstMatrix[i][k]
    mul  $t4, $t1, 2      # rowOffset = i * 2
    add  $t5, $t4, $t9    # index = i * 2 + k
    sll  $t6, $t5, 2      # byteOffset = (i * 2 + k) * 4
    add  $t7, $a0, $t6    # # elementAddr = &resultMatrix[i][j]
    lw   $s0, 0($t7)      # temp1 = firstMatrix[i][k]

    # Load secondMatrix[k][j]
    mul  $t4, $t9, 2     # rowOffset1 = k * 2
    add  $t5, $t4, $t3   # index = k * 2 + j
    sll  $t6, $t5, 2     # byteOffset = (k * 2 + j) * 4
    add  $t7, $a1, $t6   # elementAddr = &secondMatrix[k][j]
    lw   $s1, 0($t7)     # temp2 = secondMatrix[k][j]

    # Multiply temp1 and temp2 and assign it to temp3
    mul  $s2, $s0, $s1

    # Load resultMatrix[i][j]
    mul  $t4, $t1, 2    # rowOffse = i * 2
    add  $t5, $t4, $t3  # index = i * 2 + j
    sll  $t6, $t5, 2    # byteOffset= (i * 2 + j) * 4
    add  $t7, $t0, $t6  # elementAddr = &resultMatrix[i][j]
    lw   $s3, 0($t7)    # temp4 = resultMatrix[i][j]

    add  $s3, $s3, $s2  # $s3 = resultMatrix[i][j] + (firstMatrix[i][k] * secondMatrix[k][j])


    sw   $s3, 0($t7)  # Store new resultMatrix[i][j] back to memory

    addi $t9, $t9, 1   # k++
    j loop3            # Jump back to loop3

end3:
    addi $t3, $t3, 1    # j++
    j loop2             # Jump back to loop2

end2:
    addi $t1, $t1, 1  # i++
    j loop1           # Jump back to loop1

end1:
    # printing the result matrix
    li   $t1, 0   # initialize i to 0

print_loop1:
    bge  $t1, $t2, end4  # if i >= size, exit loop1
    li   $t3, 0          # initialize j to 0

print_loop2:
    bge  $t3, $t2, newlineRow  # if j >= size, exit loop2

    # Load resultMatrix[i][j]
    mul  $t4, $t1, 2    # rowOffset = i * 2
    add  $t5, $t4, $t3  # index = i * 2 + j
    sll  $t6, $t5, 2    # byteOffset = (i * 2 + j) * 4
    add  $t7, $t0, $t6  # elementAddr = &resultMatrix[i][j]
    lw   $s0, 0($t7)    # Load the value into $s0

    # Print the value in $s0
    li   $v0, 1    # Print integer syscall
    move $a0, $s0  # Move the value to $a0 for printing
    syscall

    # Print a space after each number
    li   $v0, 4  # Print string syscall
    la   $a0, space
    syscall

    addi $t3, $t3, 1   # j++
    j print_loop2      # Jump back to print_loop2

newlineRow:
    # Print newline after each row
    li   $v0, 4       # Print string syscall
    la   $a0, newline   # Load newline address
    syscall

    addi $t1, $t1, 1   # i++
    j print_loop1      # Jump back to print_loop1

end4:
    lw   $ra, 0($sp)   # Restore return address
    addi $sp, $sp, 8   # Clean up the stack

    # Exit program
    li   $v0, 10
    syscall
```

Terminal:

```
(base) nyuad@ADUAED06217LPLX:~/Downloads/lab2coa$ spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
(spim) load "matrixMultiply.s"
(spim) run
5 18
22 44
(spim) 
```

Powers Of Three

C-code:

C powersOf3.c > main()

```c
#include <stdio.h>

// Recursive function to calculate 3^n
int powerOfThree(int n) {
    if (n == 0) {
        return 1; // Base case: 3^0 = 1
    }
    return 3 * powerOfThree(n - 1); // Raecursive step: 3^n = 3 * 3^(n-1)
}

int main() {
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);

    printf("3^%d = %d\n", n, powerOfThree(n));

    return 0;
}
```

Figure 4: PowerOfThree C-code

- The functin powerOfThree takes the integer n to calculate 3^n by calling itself recursively.
- *Base Case:* If n equals 0, the function returns 1 and the recursion ends.
- *Recursive Case:* The function returns 3 * powerOfThree(n−1), so for any n > 0, it calls itself with n−1 until it reaches the base case.
- The main prompts the user input, calls the function and prints the result.

MIPS Code:

```
powersOfThree.s
.data
promptInt: .asciiz "input an integer: "
newline:   .asciiz "\n"

.text

main:
    # prompt user to input integer
    la   $a0, promptInt     # Load address of prompt string into $a0
    li   $v0, 4             # System call for printing a string
    syscall                 # Print the prompt

    # Get integer input from user
    li   $v0, 5     # System call for reading an integer
    syscall
    move $a0, $v0   # Move input n to register $a0

    # Call the powerOfThree function
    jal  powerOfThree   # Jump and link to powerOfThree function

    # Print the result (storedS in $v0)
    move $a0, $v0    # Move result to $a0 for printing
    li   $v0, 1      # System call for printing an integer
    syscall          # Print the result

    # Print a newline after the result
    la   $a0, newline   # Load address of newline string
    li   $v0, 4         # System call for printing a string
    syscall             # Print the newline

    # Exit program
    li   $v0, 10   # System call for exit
```

```
powersOfThree.s
main:
# Recursive function to calculate 3^n
powerOfThree:
    # Save registers $ra and $a0 on stack
    addi $sp, $sp, -8   # Make space on stack
    sw   $ra, 4($sp)    # Save return address
    sw   $a0, 0($sp)    # Save argument $a0

    # Base case: if n == 0, return 1
    beq  $a0, $zero, baseCase

    # Recursion
    addi $a0, $a0, -1   # Decrement n by 1
    jal  powerOfThree   # Recursive call

    # Multiply result by 3
    li   $t0, 3          # Load 3 into $t0
    mul  $v0, $v0, $t0   # Multiply $v0 by 3 (v0 contains the result of powerOfThree(n-1))

    # Restore registers and return
    lw   $ra, 4($sp)    # Restore return address
    lw   $a0, 0($sp)    # Restore argument $a0
    addi $sp, $sp, 8    # Clean up stack
    jr   $ra            # Return to caller

baseCase:
    li   $v0, 1         # If n == 0, return 1
    lw   $ra, 4($sp)    # Restore return address
    lw   $a0, 0($sp)    # Restore argument $a0
    addi $sp, $sp, 8    # Clean up stack
    jr   $ra            # Return to caller
```

Terminal:

```
(base) nyuad@ADUAED06217LPLX:~/Downloads/lab2coa$ spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
(spim) load "powersOfThree.s"
(spim) run
input an integer: 11
177147
(spim)
```

Sources:

- Lab manual (lab2)
- Extras.doc (google doc)
- SPIM_instruction_set.pdf