Week 04 Tutorial Solutions

1. The MIPS processor has 32 general purpose 32-bit registers, referenced as `$0` .. `$31`. Some of these registers are intended to be used in particular ways by programmers and by the system. For each of the registers below, give their symbolic name and describe their intended use:

    a. `$0`
    b. `$1`
    c. `$2`
    d. `$4`
    e. `$8`
    f. `$16`
    g. `$26`
    h. `$29`
    i. `$31`

Answer:

a. `$0` = `$zero`

The "zero" register. This register is read-only, and always contains the value zero. Values written to this register are discarded. It is frequently used as a source register when we want to copy constant values into another register, or as a destination register when we don't care about the result of an instruction.

b. `$1` = `$at`

The "assembler temporary" register. This register is used for various purposes by the assembler. One use is to hold the results of tests when implementing conditional branch pseudo-instructions. Programmers are encouraged to not use it directly.

c. `$2` = `$v0`

This value is used to hold the return value from a function that is returning a single 32-bit result. There is a second register (`$v1`) that can be used to return a second 32-bit result, if needed.

d. `$4` = `$a0`

This register is used to hold the first argument to a function, if that argument fits in a 32-bit register. There are three other registers, `$a1` to `$a3`, that can be used to hold arguments to functions. Arguments that don't fit into 32-bits are placed on the stack.

e. `$8` = `$t0`

Used for holding temporary values while calculating expressions. There are 10 temporary registers that can be used for this purpose: `$t0` through `$t9`. Programmers should not rely on the values of these registers persisting over a function call.

f. `$16` = `$s0`

The set of registers `$s0` through `$s7` are used to hold values that are required to persist across function calls. In other words, if a programmer stores a value in one of these registers and then calls a function, the same value should be in the register after the function returns. If necessary, the function being called needs to save and restore these values.

g. `$26` = `$k0`

This register (and register `$k1`) are reserved for use by the operating system. Programmers should not use these registers.

h. `$29` = `$sp`

The "stack pointer" register. This register holds the address of the top of the program's run-time stack. Its initial value is `0x7FFFFEFC`, and it changes down towards lower addresses. Each time a function is called, it needs to reduce `$sp` by an amount large enough to hold all of its non-register local variables, and space for saving/restoring register values.

i. `$31` = `$ra`

This register holds a return address. When a linking instruction is invoked, such as `jal`, the address of the next instruction is stored in register `$ra`. Link are often used to implement function calls: when the function wants to return, it can use `jr $ra` to return to the correct location.

2. Translate this C program to MIPS assembler

```c
// print the square of a number
#include <stdio.h>

int main(void) {
    int x, y;
    printf("Enter a number: ");
    scanf("%d", &x);
    y = x * x;
    printf("%d\n", y);
    return 0;
}
```

Store variable **x** in register $t0 and store variable **y** in register **$t1**.

Answer:

```
# print the square of a number
main:                   # x,y in $t0, $t1
    la $a0, prompt      # printf("Enter a number: ");
    li $v0, 4
    syscall

    li $v0, 5           # scanf("%d", x);
    syscall
    move $t0, $v0

    mul $t1, $t0, $t0   # y = x * x

    move $a0, $t1        # printf("%d", y);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    jr   $ra            # return

    .data
prompt:
    .asciiz "Enter a number: "
```

3. Translate this C program so it uses goto rather than if/else.
   Then translate it to MIPS assembler.

```c
#include <stdio.h>

int main(void) {
    int x, y;
    printf("Enter a number: ");
    scanf("%d", &x);

    if (x > 46340) {
        printf("square too big for 32 bits\n");
    } else {
        y = x * x;
        printf("%d\n", y);
    }

    return 0;
}
```

Answer:

Please don't write C like this, unless you are translating it to assembler:

```c
#include <stdio.h>

int main(void) {
    int x, y;
    printf("Enter a number: ");
    scanf("%d", &x);

    if (x <= 46340) goto square;
        printf("square too big for 32 bits\n");
        goto end;
    square:
        y = x * x;
        printf("%d\n", y);
    end:

    return 0;
}
```

Equivalent MIPS assembler:

```
# print the square of a number
main:                       # x,y in $t0, $t1
    la $a0, prompt          # printf("Enter a number: ");
    li $v0, 4
    syscall

    li $v0, 5               # scanf("%d", x);
    syscall
    move $t0, $v0

    ble $t0, 46340, square  # if (x <= 46340) goto square;

    la $a0, too_big         # printf("square too big for 32 bits\n");
    li $v0, 4
    syscall

    b end                   # goto end;

square:
    mul $t1, $t0, $t0       # y = x * x

    move $a0, $t1           # printf("%d", y);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall
end:
    jr   $ra                # return

    .data
prompt:
    .asciiz "Enter a number: "
too_big:
    .asciiz "square too big for 32 bits\n"
```

4. Translate this C program so it uses goto rather than if/else.
   Then translate it to MIPS assembler.

```c
#include <stdio.h>

int main(void) {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);

    if (x > 100 && x < 1000) {
        printf("medium\n");
    } else {
        printf("small/big\n");
    }
}
```

Consider this alternate version of the above program, use its approach to produce simpler MIPS assembler.

```c
#include <stdio.h>

int main(void) {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);

    char *message = "small/big\n";
    if (x > 100 && x < 1000) {
        message = "medium";
    }

    printf("%s", message);
}
```

Answer:

Please don't write C like this, unless you are translating it to assembler:

```c
#include <stdio.h>

int main(void) {
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);

    if (x <= 100) goto small_big;
    if (x >= 1000) goto small_big;
        printf("medium\n");
        goto end;
    small_big:
        printf("small/big\n");
    end:

    return 0;
}
```

Equivalent MIPS assembler:

```
main:                       # x in $t0
    la $a0, prompt          # printf("Enter a number: ");
    li $v0, 4
    syscall

    li $v0, 5               # scanf("%d", x);
    syscall
    move $t0, $v0

    ble $t0, 100, small_big  # if (x <= 100) goto small_big;
    bge $t0, 1000, small_big # if (x >= 1000) goto small_big;

    la $a0, medium_str       # printf("medium\n");

    li $v0, 4
    syscall

    b end                   # goto end;

small_big:
    la   $a0, small_big_str #printf("small/big\n");
    li   $v0, 4
    syscall
end:
    jr   $ra                # return

    .data
prompt:
    .asciiz "Enter a number: "
medium_str:
    .asciiz "medium\n"
small_big_str:
    .asciiz "small/big\n"
```

Simpler MIPS assembler:

```
main:                       # x in $t0, message in $t1
    la $a0, prompt          # printf("Enter a number: ");
    li $v0, 4
    syscall

    li $v0, 5               # scanf("%d", x);
    syscall
    move $t0, $v0

    la $t1, small_big_str   # message = "small/big\n";
    ble $t0, 100, end       # if (x <= 100) goto small_big;
    bge $t0, 1000, end      # if (x >= 1000) goto small_big;

    la $t1, medium_str      # printf("medium\n");

end:
    move $a0, $t1           # printf("%s", message);
    li   $v0, 4
    syscall

    jr   $ra                # return

    .data
prompt:
    .asciiz "Enter a number: "
medium_str:
    .asciiz "medium\n"
small_big_str:
    .asciiz "small/big\n"
```

5. Translate this C program so it uses goto rather than if/else.
   Then translate it to MIPS assembler.

```c
#include <stdio.h>

int main(void) {
    for (int x = 24; x < 42; x += 3) {
        printf("%d\n",x);
    }
}
```

Answer:

Please don't write C like this, unless you are translating it to assembler:

```c
#include <stdio.h>

int main(void) {
    int x = 24;

    loop:
    if (x >= 42) goto end;

        printf("%d", x);
        printf("%c", '\n');

        x += 3;

        goto loop;
    end:

    return 0;
}
```

Equivalent MIPS assembler:

```
main:                # int main(void) {
                     # int i;  // in register $t0

    li    $t0, 24    # i = 24;

loop:                # loop:
    bge  $t0, 42 end # if (i > 10) goto end;

    move $a0, $t0    #   printf("%d" i);
    li   $v0, 1
    syscall

    li   $a0, '\n'     # printf("%c", '\n');
    li   $v0, 11
    syscall

    add  $t0, $t0 3  #   i += 3;

    b    loop        # goto loop;

end:
    jr   $ra         # return
```

6. Translate this C program so it uses goto rather than if/else.
   Then translate it to MIPS assembler.

```c
// print a triangle
#include <stdio.h>

int main (void) {
    for (int i = 1; i <= 10; i++) {
        for (int j = 0; j < i; j++) {
            printf("*");
        }
        printf("\n");
    };
    return 0;
}
```

Answer:

Please don't write C like this, unless you are translating it to assembler:

```
// print a triangle
#include <stdio.h>

int main (void) {
    for (int i = 1; i <= 10; i++) {
        for (int j = 0; j < i; j++) {
            printf("*");
        }
        printf("\n");
    };
    return 0;
}
```

Equivalent MIPS assembler:

```
# i in register $t1
# j in register $t2
main:
    li $t1, 1           # i = 1
loop0:
    bgt $t1, 10, end0   # if (i > 10) goto end0;

    li $t2, 0           # j = 0
loop1:
    bge $t2, $t1, end1  # if (j >= i) goto end1;

    li   $a0, '*'       # printf("%c", '*');
    li   $v0, 11
    syscall

    add $t2, $t2, 1     # j++

    b loop1
end1:

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    add $t1, $t1, 1     # i++

    b loop0

end0:
    jr $31
```

7. Translate this C program so it uses goto rather than if/else.
   Then translate it to MIPS assembler.

```
// Simple factorial calculator  - without error checking

#include <stdio.h>

int main (void) {
    int n;
    printf("n  = ");
    scanf("%d", &n);

    int fac = 1;
    for (int i = 1; i <= n; i++) {
        fac *= i;
    }

    printf ("n! = %d\n", fac);
    return 0;
}
```

Answer:

Please don't write C like this, unless you are translating it to assembler:

```c
// Simple factorial calculator  - without error checking

#include <stdio.h>

int main (void) {
    int n;
    printf("n  = ");
    scanf("%d", &n);

    int fac = 1;
    int i = 1;

    loop:
    if (i > n) goto end;
        fac *= i;
        i++;
    goto loop;
    end:

    printf ("n! = %d\n", fac);
    return 0;
}
```

Equivalent MIPS assembler:

```
### COMP1521 19t2 ... week 04 lab
### Compute factorials -- no functions (except main)


main:

    # code for main()
    li  $s0, 0      # n = 0

    la  $a0, msg1
    li  $v0, 4
    syscall         # printf("n  = ")

    li  $v0, 5
    syscall         # scanf("%d", into $v0)

    ### >>>>>
    move    $s0, $v0

    li  $s2, 1          # fac = 1

main_fac_init:
    li  $s1, 1          # i = 1
main_fac_cond:
    bgt $s1, $s0, main_fac_f    # i <= n  -->  if (i > n) break
    mul $s2, $s2, $s1       # fac = fac * i
main_fac_step:
    addi    $s1, $s1, 1     # i++
    j   main_fac_cond
main_fac_f:

    ### <<<<<

    la  $a0, msg2
    li  $v0, 4
    syscall         # printf("n! = ")

    move    $a0, $s2    # assume $s2 holds n!
    li  $v0, 1
    syscall         # printf("%d", fac)

    la  $a0, eol
    li  $v0, 4
    syscall         # printf("\n")

    jr  $ra     # return 0

    .data
msg1:   .asciiz "n  = "
msg2:   .asciiz "n! = "
eol:    .asciiz "\n"
```