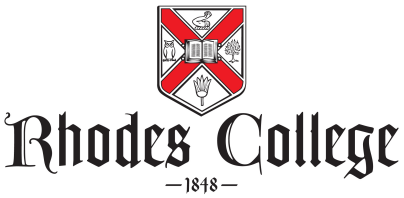
# COMP 231-01
# Introduction to Computer Organization
# Final Exam Review Solutions




For this exam, the main addition to the exam is **functions** in assembly language and the register conventions we talked about in class. When you're designing assembly language programs on the test, you're responsible for using registers correctly according to the conventions. I recommend studying and reviewing the **register conventions** section of the assembly textbook and then working on coding the following exercises. We're also covering the FDXW cycle and memory, but those questions will appear as short answer problems.

To study for this exam, I recommend reviewing your work for the course, most especially the exam reviews and tests, as well as this final exam review. This exam is cumulative and will include questions from across the semester.

- The function detailed below is a recursive C function for calculating the result of an exponentiation operation. Note that this function is designed for integers greater than or equal to 0.

```
int exponentiate(int base, int power)
{
        if(power <= 0)
        {
                return 1;
        }
        else
        {
                return base * exponentiate(base, power-1);
        }
}
```

Translate this function into assembly language, using the calling conventions.

```
#Note that this function assumes the stack pointer has already been
#initiated appropriately in the main function.
#base is passed in through r4 and power through r5, by the register conventions
exponentiate:
        bgt r5, r0, ONERETURN
        subi sp, sp, 8 #Reserve stack space for two values

        subi r5, r5, 1 #We can pass this without saving its original value since we don't use it aft

        stw r4, 0(sp)
        stw ra, 4(sp)

        call exponentiate

        ldw ra, 4(sp)
        ldw r4, 0(sp)

        mul r2, r2, r4

        addi sp, sp, 8 #Deallocating stack space.

        ret

        ONERETURN:
                movi r2, 1
                ret
```

- The code below is an assembly language program that has been improperly translated and does not follow the calling conventions. Find and correct the errors in this program. (There are intended to be 8 errors. Some will be related to the calling conventions, but others will be related to other aspects of the program.)

```
\#This function takes the values in memory location 1000 and applies the MATHFUNC to it.
\#Then, this function adds that same value (from MEM 1000) with the results of the MATHFUNC operation
\_start
ldw r9, 1000

call MATHFUNC

add r0, r9, r2

MATHFUNC: \#This function triples the argument value and adds 2 to it, then returns that value.
call FUNC
addi r9, 2
ret


FUNC:  \#This function takes in an argument, triples the given value, and returns that value.
multi r9, r9, 3
```

  Error list:

  - _start doesn't have a colon (_start:).
  - ldw takes in an offset/register value; this could be written correctly as ldw r9, 1000(r0).
  - r9 is a caller-save register, and we don't save it before making a call!
  - r0 can not be written to; it always contains 0.
  - MATHFUNC - r9 is a caller-save register, and we don't save it before making a call!
  - MATHFUNC - the argument (r9) is not passed in! r9 should be stored in r4.
  - FUNC - the argument (r9) is not passed in! r9 should be stored in r4.
  - MATHFUNC - the return value is not passed out in r2.
  - FUNC - the return value is not passed out in r2.
  - MATHFUNC does not save the ra! This causes an infinite loop.
  - addi requires three arguments; this could be correctly written addi r9, r9, 2.
  - FUNC - there is no return statement!
  - There is no branch statement after main and before MATHFUNC...and so execution will continue into MATHFUNC, which is not intended.

  This program ended up with a lot more errors than just 8 (13, in total). Give yourself a brief round of applause if you caught 10 or more of these errors!

- The function detailed below is a recursive C function for calculating the result of a factorial operation. Note that this function is designed for integers greater than or equal to 0.

```
int factorial(int input)
{
        if(input <= 1)
        {
                return 1;
        }
        else
        {
                return input * factorial(input - 1);
        }
}
```

Write an assembly program to calculate the factorial value of 10. To do this, write the main function and also translate this into assembly language, using the calling conventions.

```
#Note that this function assumes the stack pointer has already been
#initiated appropriately in the main function.
#base is passed in through r4 and power through r5, by the register conventions

_start:
        movia sp, stack #Initialize the stack.  Remember to set up the corresponding part at the end

        movi r4, 10
        call factorial
        #We now have the result of 10 factorial stored in r2!
br END #Prevent accidental execution of factorial!
factorial:
        blt r5, r0, ONERETURN
        subi sp, sp, 4 #Reserve stack space for one value

        subi r4, r4, 1 #We can pass this without saving its original value since we don't use it aft

        stw ra, 0(sp)

        call factorial

        ldw ra, 0(sp)

        mul r2, r2, r4

        addi sp, sp, 4 #Deallocating stack space.

        ret

        ONERETURN:
                movi r2, 1
                ret
```

```
END:
.skip 500 #These two lines set up a 500-byte storage space for the stack.
stack:
.end
```

- What is meant by the term spatial locality?

  Variables located near each other in memory (ex., values stored in the same row of an array)

- What is meant by the term temporal locality?

  A variable that has been used recently is likely to be used again.

- What is meant by the term pipelining?

  Pipelining refers to the process by which commands are staged; instead of completing a whole command before moving to the next one, the command is split into individual parts so that resources can be maximized. In this approach, the fetch component of one command might be in progress while the ALU addition component of another command is executing.

- Why is out of order processing used in assembly language?

  OOP (out-of-order processing) is used to speed up execution of code; if the CPU can be working on more than one instruction at once, the average time for an instruction to be completed can be reduced.

- What is the inherent conflict between branches and pipelining?

  Pipelining seeks to improve run time by working on several instructions at once. This process is hampered if the CPU doesn't know which branch will be executed, though there are several strategies to counter this.

# NIOS II Instruction Reference

This is a list of the syntax for the instructions that you may use from the NIOS instruction set:

| instruction | usage |
|---|---|
| ldw | ldw rB, off(rA) |
| ldb | ldb rB, off(rA) |
| ldbu | ldbu rB, off(rA) |
| ldbu | ldbu rB, off(rA) |
| ldh | ldh rB, off(rA) |
| ldhu | ldhu rB, off(rA) |
| stw | stw rB, off(rA) |
| stb | stw rB, off(rA) |
| sth | stw rB, off(rA) |
| add | add rC, rA, rB |
| sub | sub rC, rA, rB |
| mul | mul rC, rA, rB |
| div | div rC, rA, rB |
| addi | addi rC, rA, IMMED16 |
| subi | subi rC, rA, IMMED16 |
| muli | muli rC, rA, IMMED16 |
| divu | divu rC, rA, rB |
| and | and rC, rA, rB |
| or | or rC, rA, rB |
| xor | xor rC, rA, rB |
| andi | andi rC, rA, IMMED16 |
| ori | ori rC, rA, IMMED16 |
| xori | xori rC, rA, IMMED16 |
| andhi | andhi rC, rA, IMMED16 |
| orhi | orhi rC, rA, IMMED16 |
| xorhi | xorhi rC, rA, IMMED16 |
| mov | mov rC, rA |
| movi | movi rB, IMMED16 |
| movui | movui rB, IMMED16 |
| srl | srl rC, rA, rB |
| srli | srli rC, rA, IMMED5 |
| sra | sra rC, rA, rB |
| srai | srai rC, rA, IMMED5 |
| sll | sll rC, rA, rB |
| slli | slli rC, rA, IMMED5 |
| jmp | jmp rA |
| br | br LABEL |
| beq | beq rA, rB, LABEL |
| bne | bne rA, rB, LABEL |
| blt | blt rA, rB, LABEL |
| ble | ble rA, rB, LABEL |
| bgt | bgt rA, rB, LABEL |
| bge | bge rA, rB, LABEL |
| bltu | bltu rA, rB, LABEL |
| bleu | bleu rA, rB, LABEL |
| bgtu | bgtu rA, rB, LABEL |
| bgeu | bgeu rA, rB, LABEL |
| call | call LABEL |
| ret | ret |