

## 1 Handwritten (35%)

### 2.8 (10%)

Translate the following RISC-V code to C. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

```
addi x30, x10, 8
addi x31, x10, 0
sd x31, 0(x30)
ld x30, 0(x30)
add x5, x30, x31
```

### 2.9 (10%)

For each RISC-V instruction in Exercise 2.8, show the value of the opcode (op), source register (rs1), and destination register (rd) fields. For the I-type instructions, show the value of the immediate field, and for the R-type instructions, show the value of the second source register (rs2). For non U- and UJ-type instructions, show the funct3 field, and for R-type and S-type instructions, also show the funct7 field.

### 2.16 (15%)

Assume that we would like to expand the RISC-V register file to 128 registers and expand the instruction set to contain four times as many instructions.

#### 2.16.1 (5%)

How would this affect the size of each of the bit fields in the R-type instructions?

#### 2.16.2 (5%)

How would this affect the size of each of the bit fields in the I-type instructions?

#### 2.16.3 (5%)

How could each of the two proposed changes decrease the size of a RISC-V assembly program? On the other hand, how could the proposed change increase the size of an RISC-V assembly program?

## 2 Programming (70%)

We will test the following problems on [RISC-V software stack](#). The packages we use are [spike](#), [proxy kernel](#) with newlib and [gcc](#). And the riscv-isa we will use to test the program is [RV64IMAFDC](#).

	Applications			
Distributions	OpenEmbedded	Gentoo	busybox	
Compilers	clang/LLVM		GCC	
System Libraries	newlib		glibc	
OS Kernels	Proxy Kernel		Linux	
Implementations	Rocket	Spike	ANGEL	QEMU

Before we start programming, we will use **docker** to set up our environment (Refer to the supplementary.pdf to see how to install docker and use it).

```
docker pull ntuca2020/hw2 // (4G)
docker run --name=test -it ntuca2020/hw2
cd /root
ls
```

The folder structure in the docker image looks like the following:

```
/root
|-- Examples
|   |-- Example1          // inline assembly test
|   |-- Example2          // link with .s file test
|   '-- Example3          // setup debug environment
'-- Problems
    |-- fibonacci          // fibonacci number
    |   |-- Makefile
    |   |-- fibonacci.c
    |   '-- fibonacci.s
    |-- convert            // string to int
    |   |-- Makefile
    |   |-- convert.c
    |   '-- convert.s
    '-- matrix             // matrix multiplication
        |-- Makefile
        |-- matrix.c
        '-- matrix.s
```

make and make test to try it out.

You only need to submit \*.s file of each problem.

## Fibonacci (20%)

Implement Fibonacci number in assembly. ( $F_0 = 0, F_1 = 1$ , output  $F_n$ , no overflow)

```
unsigned long long int fibonacci(int);
```

Input:  
70

Output:  
190392490709135

## Convert (20%)

- Convert an ASCII string containing a positive or negative integer decimal string to an integer. Input length is at most 15 bytes.
- '+' and '-' will appear optionally. And once they appear, they will only appear once in the first byte.
- If a non-digit character appears anywhere in the string, your program should stop and return -1.
- The return value will be printed out in 32bit-int format.

```
int convert(char *);
```

Input:

```
+123
+0000000123
-123
-0000000321
2147483647
2147483648
-2147483648
-123123AAA
+123123AAA
123123AAA
```

Output:

```
123
123
-123
-321
2147483647
-2147483648
-2147483648
-1
-1
-1
```

## Matrix multiplication (15%)

Do matrix multiplication of size 128x128 with some additional operations.

```
for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
        for (int k = 0; k < SIZE; k++)
            C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % 1024;
Elements in A and B are unsigned 16 bits numbers of range [0,1023]
```

We will score based on the cycle counts. You can use C as an initial attempt.

```
asm volatile ("rdcycle %0" : "=r" (start));
// matrix multiplication
asm volatile ("rdcycle %0" : "=r" (end));
```

Grading:

- Below 20,000,000 cycles (2%)
- Below 18,000,000 cycles (2%)
- Below 16,000,000 cycles (2%)
- Below 14,000,000 cycles (2%)
- Below 12,000,000 cycles (2%)
- Below 10,000,000 cycles (1%)
- Below 9,000,000 cycles (1%)
- Below 8,000,000 cycles (1%)
- Below 7,000,000 cycles (1%)
- Below 6,000,000 cycles (1%)

## Report on matrix multiplication (15%)

- Briefly explain how you get below 6,000,000 cycles.
- Or you can answer the following questions:
  - How many cycles does it take by just doing the naive matrix multiplication?
  - How many load and store does it need (roughly) during the whole computation? (Considering the registers it use)
  - Is there any way to keep registers being used as much as possible before they're replaced? (Hint: blocking)
  - How many loop controls does it need (roughly) during the whole computation?

## Submission

- All \*.s file should be written assembly.
- Zip and upload your file to NTU cool in the following format, the file name should be <Student ID>.zip:

```
b08922000          <-- zip this folder
|-- fibonacci.s
|-- convert.s
|-- matrix.s
'-- report.pdf     // including handwritten part and report on matrix multiplication part
```

- If there's any question, please send email to [110fall.ca@gmail.com](mailto:110fall.ca@gmail.com).