This exam has 12 questions, for a total of 100 points.

1. **4 points**  What is the output of the following Python program?

   ```python
   a = [[0], 1]
b = a[0]
c = a
c[0] = [1]
print(b[0])
   ```

   **Solution:**
   
   0

2. **4 points**  What is the output of the following Python program?

   ```python
   a = [[0], 1]
b = a[0]
c = a
c[0][0] = 1
print(b[0])
   ```

   **Solution:**
   
   1

3. **4 points**  What is the output of the following Python program?

   ```python
   def f(x : int) -> None:
       x = 0
       
y = 1
f(y)
print(y)
   ```

   **Solution:**
   
   1
4. **4 points** Why does our compiler spill variables of `tuple` type to the root stack instead of the regular procedure call stack?

**Solution:** We spill them to the root stack so that the garbage collector has easy access to all the live tuples. It separates them from the other non-vector variables that the garbage collector must ignore.

5. **4 points** Why must the prelude of a function push the contents of the `rbp` register to the procedure call stack?

**Solution:** The `rbp` register is a callee-saved register, so when we return from this function, its contents must be the same as they were upon entry to this function. But we change `rbp` in this function, so we have to restore its original value in the conclusion. Thus, we push it on the stack in the prelude and pop it back off in the conclusion.
6. **10 points** Given the following program, what would be the output of the Expose Allocation pass? Recall that you may use the new AST nodes `GlobalValue`, `Allocate`, and `Collect`.

```
print( (42,) [0] )
```

**Solution:** (2 points each)
The tuple creation should be translated into code that

- checks for space,
- calls collect,
- allocate the tuple,
- initializes the tuple, and
- returns the address of the tuple.

```
print({
    init.321 = 42
    if free_ptr + 16 < fromspace_end:
        else:
            collect(16)
    alloc.320 = allocate(1, tuple[int])
    alloc.320[0] = init.321
    alloc.320[0])
```
7. **12 points** Given the input program on the left, fill in the blanks in the output of Select Instructions on the right.

```plaintext
_start:
init.321 = 42
tmp.322 = free_ptr
tmp.323 = tmp.322 + 16
tmp.324 = fromspace_end
if tmp.323 < tmp.324:
goto _block.328
else:
goto _block.329

_block.328:
goto _block.327

_block.329:
collect(16)
goto _block.327

_block.327:
alloc.320 = allocate(1,tuple[int])
alloc.320[0] = init.321
tmp.325 = alloc.320
tmp.326 = tmp.325[0]
print(tmp.326)
return 0
```

**Solution:** (2 points each)

(a) `free_ptr(%rip)`
(b) `fromspace_end(%rip)`
(c) `callq collect`
(d) `addq $16, free_ptr(%rip)`
(e) `movq init.321, 8(%r11)`
(f) `movq 8(%r11), %r11`
8. [12 points] Draw the interference graph for the following program fragment by adding edges between the nodes below. You do not need to include edges between two registers. The live-after set for each instruction is given to the right of each instruction and the types of each variable is listed below.

Recall that the caller-saved registers are

rax rcx rdx rsi rdi r8 r9 r10 r11

and the callee-saved registers are

rsp rbp rbx r12 r13 r14 r15

a : NoneType, b : tuple[int], c : tuple[int], d : tuple[int]

```
block1:  
    movq %r15, %rdi  { rdi d }
    movq $16, %rsi  { rdi d rsi }
    callq collect  { d }
    jmp block2  { d }

block2:  { d }
    movq free_ptr(%rip), %r11  { d }
    addq $16, free_ptr(%rip)  { d }
    movq %3, 0(%r11)  { r11 d }
    movq %r11, b  { b d }
    movq b, %r11  { b d }
    movq $0, 8(%r11)  { b d }
    movq $0, a  { b d }
    movq b, c  { c d }
    cmpq c, d  { }
    je block7  { }
    jmp block8  { }
```

Solution:
- Edges between d and the caller-saved registers. (2 points)
- Edges between d and the callee-saved registers. (2 points)
- The edges between variables a, b, c, and d. (2 points each)
9. [12 points] Given the following output of Remove Complex Operands, apply the Explain Control pass to translate the program to $C_{Fun}$. You may use concrete or abstract syntax for your answer. Make sure to distinguish regular calls (concrete syntax $fun(arg_1, \ldots, arg_n)$) from tail calls (concrete syntax $tail\ fun(arg_1, \ldots, arg_n)$). A variable inside braces such as $\{dub\}$ represents a FunRef AST node.

```python
def dub(f:Callable[[int], int], x:int) -> int:
    tmp.0 = f(x)
    return f(tmp.0)

def inc(x:int) -> int:
    return x + 1

def main() -> int:
    fun.1 = {dub}
    fun.2 = {inc}
    tmp.3 = input_int()
    tmp.4 = fun.1(fun.2, tmp.3)
    print(tmp.4)
    return 0
```

**Solution:**

- Regular call inside `dub`. (2 points)
- Tail call inside `dub`. (2 points)
- Return statement inside `inc`. (2 points)
- Assignment statements do not change. (2 points)
- Regular call inside `main`. (2 points)
- Start labels. (2 points)

```python
def dub(f:Callable[[int], int], x:int) -> int:
    with dubstart:
        tmp.0 = f(x)
        tail f(tmp.0)

def inc(x:int) -> int:
    with incstart:
        return x + 1

def main() -> int:
    with mainstart:
        fun.1 = {dub}
        fun.2 = {inc}
        tmp.3 = input_int()
        tmp.4 = fun.1(fun.2, tmp.3)
        print(tmp.4)
        return 0
```
10. **12 points** Given the following CFun program, apply the Select Instructions pass. A variable inside braces such as \{id\} represents a FunRef AST node.

```python
def id(x:int) -> int:
    idstart:
    return x

def main() -> int:
    mainstart:
    fun.0 = \{id\}
    tmp.1 = fun.0(42)
    print(tmp.1)
    return 0
```

Recall that the following six registers are used for passing arguments to functions.

```
rdi rsi rdx rcx r8 r9
```

**Solution:**

```python
def id() -> int:
    # no parameters (1 point)
    idstart:
    movq %rdi, x
    # parameter passing (2 points)
    movq x, %rax
    # return x (1 point)
    jmp idconclusion

def main() -> int:
    mainstart:
    leaq id(%rip), fun.0
    # FunRef (2 points)
    movq $42, %rdi
    # parameter passing (1 point)
    callq *fun.0
    # indirect call (2 points)
    movq %rax, tmp.1
    # call result (1 point)
    movq tmp.1, %rdi
    # parameter passing (1 point)
    callq print_int
    movq $0, %rax
    # return 0 (1 point)
    jmp mainconclusion
```
11. **[10 points]** Recall that the Limit Functions pass changes all the functions in the program so that they have at most 6 parameters (the number of argument-passing registers), making it easier to implement efficient tail calls. The `limit_type` auxiliary function changes each type annotation in the program as part of the Limit Functions pass. Fill in the blanks in `limit_type`.

```python
def limit_type(t):
    match t:
    case TupleType(ts):
        new_ts = [___(a)___ for t in ts]

        return ___(b)___

    case FunctionType(ps, rt):
        new_ps = [limit_type(t) for t in ps]
        new_rt = limit_type(rt)
        n = len(arg_registers)
        if len(new_ps) > n:
            front = new_ps[0 : n-1]
            back = new_ps[n-1 :]
            return ___(c)___
        else:
            return ___(d)___

    case _:
        return ___(e)___
```

**Solution:** (2 points each)

(a) `limit_type(t)`
(b) `TupleType(new_ts)`
(c) `FunctionType(front + [TupleType(back)], new_rt)`
(d) `FunctionType(new_ps, new_rt)`
(e) `t`
12. **12 points** Given the following x86 code for a function named `map_vec`, write down the code for its prelude and conclusion.

```
map_vecstart:
    movq  %rdi, -16(%rbp)
    movq  %rsi, -8(%r15)
    movq  8(%r15), %r11
    movq  %r11, %rsi
    callq  -16(%rbp)
    movq  %rax, %rdi
    callq  -16(%rbp)
    movq  %r15, %rdi
    movq  -8(%r15), %r11
    movq  16(%r11), %rsi
    movq  %rsi, %r11
    callq  -16(%rbp)
    movq  16(%r11), %rsi
    movq  %r11, %rsi
    movq  %rax, 8(%r11)
    movq  $0, %rdi
    movq  %rsi, %rax
    jmp map_vecconclusion

block6:
    movq  free_ptr(%rip), %r11
    addq  $24, free_ptr(%rip)
    movq  5, 0(%r11)
    movq  %r11, %rsi
    movq  %r11, %rsi
    movq  %rbx, 8(%r11)
    movq  %r15, %rdi
    addq  $24, free_ptr(%rip)
    movq  fromspace_end(%rip), %rsi
    addq  $24, %rdi
    jmp map_vecconclusion

block7:
    movq  $0, %rsi
    jmp block6
```

**Solution:** The prelude should:

- Save `rbp` (1 point)
- Set `rbp` to the `rsp` (1 point)
- Save `rbx` (1 point)
- Subtract 8 from the `rsp` (`align(8 + 8) − 8 = 8`) (1 point)
- Initialize 1 slot of the rootstack and add 8 to `r15`. (2 points)
- Jump to `map_vecstart` (1 point)

The conclusion should:

- Subtract 8 from `r15` (1 point)
- Add 8 to `rsp` (1 points)
- Restore `rbx` (1 points)
- Restore `rbp` (1 points)
• Return (1 points)

```assembly
.align 16
map_vec:
pushq %rbp
movq %rsp, %rbp
pushq %rbx
subq $8, %rsp
movq $0, 0(%r15)
addq $8, %r15
jmp map_vecstart

map_vecconclusion:
subq $8, %r15
addq $8, %rsp
popq %rbx
popq %rbp
retq
```