1. main.py:

```
.data
    .align 2    # Align data to word boundaries for better memory
access performance.


    # Keywords
    qubit_str: .string "qubit "         # Keyword for qubit
declaration.
    print_str: .string "print"          # Keyword for print statement.
    gate1_str: .string "qubitop1"       # Keyword for single-qubit
gate operations.
    gate2_str: .string "qubitop2"       # Keyword for two-qubit gate
operations.
    end_str: .string "end"              # Keyword to signify program
termination.


    # Error messages
    err_syntax: .string "Syntax Error\n"        # Message for general
syntax errors.
    err_var: .string "Variable not found\n"     # Message for
undefined variable access.
    err_undeclared: .string "Variable not declared\n" # Message for
undeclared variable use.


    qubitnum: .word 1      # Tracks the number of qubits in the
system.


    # Single Qubit Gates
    qubit1: .float 0.707,0,0.707,0    # Example single qubit state
(superposition).
    res: .float 0,0,0,0               # Buffer to store the result of
a gate operation.
    row1: .word 2                     # Number of rows in the first
operand.
```

```
    row2: .word 2                       # Number of columns in the first
operand (matches rows of the second operand).

    col2: .word 1                       # Number of columns in the
second operand.

    Qgate: .word 0                      # Placeholder for chosen gate
operation.

    # Common single-qubit gates.

    Hgate: .float 0.707,0,0.707,0, 0.707,0,-0.707,0   # Hadamard gate.

    Igate: .float 1,0,0,0, 0,0,1,0                  # Identity gate.

    Xgate: .float 0,0,1,0, 1,0,0,0                  # Pauli-X gate.

    Ygate: .float 0,0,0,-1, 0,1,0,0                 # Pauli-Y gate.

    Zgate: .float 1,0,0,0, 0,0,-1,0                 # Pauli-Z gate.

    Sgate: .float 1,0,0,0, 0,0,0,1                  # S gate (phase
gate).

    Tgate: .float 1,0,0,0, 0,0,0.707,0.707          # T gate (π/8
phase gate).

    SNgate: .float 0.707,0.707,0.707,-0.707, 0.707,-0.707,0.707,0.707
# Example custom gate.


    # Two Qubit Gates

    qubit2: .float 0,0,1,0       # Second qubit in a simple state.

    qubit3: .float 1,0,0,0       # Third qubit in the ground state.

    res_2: .float 0,0,0,0,0,0,0,0    # Buffer to store results of two-
qubit operations.


    tensor: .float 0,0,0,0,0,0,0,0  # Buffer for tensor product
computation.

    space: .word 0,0,0,0            # Helper array for intermediate
operations.

    row1_2: .word 4                 # Rows in the first operand (for
two-qubit operations).

    row2_2: .word 4                 # Columns in the first operand
(matches rows of the second operand).

    col2_2: .word 1                 # Columns in the second operand.

    Qgate_2: .word 1                # Placeholder for selected two-
qubit gate.
```

```
    # Common two-qubit gates.
    CNgate1: .float 1,0,0,0,0,0,0,0     # Controlled-NOT gate
configuration.
    CNgate2: .float 0,0,1,0,0,0,0,0
    CNgate3: .float 0,0,0,0,0,0,1,0
    CNgate4: .float 0,0,0,0,1,0,0,0
    SWgate1: .float 1,0,0,0,0,0,0,0     # SWAP gate.
    SWgate2: .float 0,0,0,0,1,0,0,0
    SWgate3: .float 0,0,1,0,0,0,0,0
    SWgate4: .float 0,0,0,0,0,0,1,0
    CZgate1: .float 1,0,0,0,0,0,0,0     # Controlled-Z gate.
    CZgate2: .float 0,0,1,0,0,0,0,0
    CZgate3: .float 0,0,0,0,-1,0,0,0
    CZgate4: .float 0,0,0,0,0,0,-1,0
    CYgate1: .float 1,0,0,0,0,0,0,0     # Controlled-Y gate.
    CYgate2: .float 0,0,0,0,0,0,0,-1
    CYgate3: .float 0,0,0,0,0,0,0,0
    CYgate4: .float 0,0,0,1,0,0,0,0


    # Symbol table and storage
    symbol_table: .space 1000   # Space to store mappings of variable
names to memory locations.
    value_storage: .space 2000  # Space for actual qubit state values.
    buffer: .space 100          # Input buffer for user commands.
    .align 2
    prompt: .string ">> "       # Prompt string for user input.
    .align 2
    newline: .string "\n"


.text
.globl main
main:
```

```
    # Initialize memory pointers
    la s11, value_storage  # Points to the starting location of the
value storage region.


    input_loop:
        # Print input prompt
        la a0, prompt
        li a7, 4           # Syscall for printing a string.
        ecall


        # Read user input
        la a0, buffer      # Address of the input buffer.
        li a1, 100         # Maximum input length.
        li a7, 8           # Syscall for reading a string.
        ecall


        # Check if input corresponds to a single-qubit gate operation.
        check_1Qubitgate_op:
            la t0, buffer               # Load input into temporary
register.
            la t1, gate1_str            # Load single-qubit gate
keyword.
            jal starts_with             # Check if input starts with
this keyword.
            bnez a0, check_2Qubitgate_op   # If not, move to two-qubit
gate check.


            # Handle single-qubit gate operations.
            jal ra, handle_qubitop1_stmt  # Parse and execute single-
qubit gate operations.


            beqz a0, input_loop        # Return to input loop if
operation succeeds.


            # Handle syntax errors if operation fails.
```

```
            j syntaxError

        check_2Qubitgate_op:
            la t0, buffer
            la t1, gate2_str            # Load two-qubit gate keyword.
            jal starts_with             # Check if input matches.
            bnez a0, try_next_command   # If not, try the next command
type.

            # Handle two-qubit gate operations.
            jal ra, handle_qubitop2_stmt

            beqz a0, input_loop        # Return to input loop if
operation succeeds.

            # Handle syntax errors for failed operations.
            j syntaxError

        try_next_command:
            # Check for other command types: print, qubit declaration,
or end.
            la t0, buffer
            la t1, print_str
            jal check_print_cmd
            beqz a0, handle_print_stmt

            la t0, buffer               # Example: qubit a =
[(1,0),(0,1)]
            la t1, qubit_str            # Check for qubit
declarations.
            jal starts_with
            beqz a0, handle_qubit_decl

            la t0, buffer
```

```
            la t1, end_str              # Check for "end" keyword to
terminate.
            jal check_end_cmd
            beqz a0, handle_end_cmd


        # Fall-through case: Syntax error if no command matches.
        j syntaxError


# Data related codes
.include "store.s"         # Include file for variable storage
functions.
.include "search.s"        # Include file for symbol table search
functions.


# Miscellenious codes
.include "errors.s"        # Include file for error handling.
.include "utils.s"         # Include file for utility functions.
.include "gateFunctions.s" # Include file for gate operation
functions.


# keywords related codes
.include "print.s"         # Include file for handling print
statements.
.include "qubit.s"         # Include file for qubit-specific
operations.
.include "oneQubitGate.s"  # Include file for single-qubit gate
operations.
.include "twoQubitGate.s"  # Include file for two-qubit gate
operations.
```

2. utils.s:

```
.globl starts_with           # Make the `starts_with` routine
globally accessible.
.globl prefix_match          # Global label for successful prefix
match.
.globl prefix_diff           # Global label for prefix mismatch.
```

```
.globl starts_with_char1      # Special case to compare a single
character.


# Main routine to check if a string (pointed to by t0) starts with a
given prefix (pointed to by t1).

starts_with:

    lb t2, 0(t1)             # Load the first byte of the prefix
string into t2.

    beqz t2, prefix_match    # If the prefix is empty (null terminator
reached), it's a match.

    lb t3, 0(t0)             # Load the first byte of the input string
into t3.

    beqz t3, prefix_diff     # If the input string ends before the
prefix, it's a mismatch.

    bne t2, t3, prefix_diff  # If the characters do not match, it's a
mismatch.

    addi t0, t0, 1           # Move to the next character in the input
string.

    addi t1, t1, 1           # Move to the next character in the
prefix string.

    j starts_with            # Repeat for the next character.


# Special case: Check if a single character (pointed to by t1) matches
the start of a string (pointed to by t0).

starts_with_char1:

    lb t2, 0(t1)             # Load the single character into t2.

    lb t3, 0(t0)             # Load the first character of the input
string into t3.

    beq t2, t3, prefix_match # If they match, jump to `prefix_match`.

    j prefix_diff            # Otherwise, it's a mismatch.


# Label for a successful match.

prefix_match:

    li a0, 0                 # Set return value to 0 (indicates a
match).

    ret                      # Return to the caller.
```

```
# Label for a mismatch.

prefix_diff:

    li a0, 1                 # Set return value to 1 (indicates a
mismatch).

    ret                      # Return to the caller.
```

3. errors.s

```
.globl not_found_var       # Declare `not_found_var` as a globally
accessible label.

.globl syntaxError         # Declare `syntaxError` as a globally
accessible label.

.globl VariableNotDeclared # Declare `VariableNotDeclared` as a
globally accessible label.


# Routine to handle syntax errors.

syntaxError:

    la a0, err_syntax       # Load the address of the "Syntax Error"
message into a0.

    li a7, 4                # Syscall number for printing a string.

    ecall                   # Make the syscall to output the error
message.

    j input_loop            # Jump back to the input loop to wait for
the next user input.


# Routine to handle "Variable not found" errors.

not_found_var:

    la a0, err_var          # Load the address of the "Variable not
found" message into a0.

    li a7, 4                # Syscall number for printing a string.

    ecall                   # Make the syscall to output the error
message.

    j input_loop            # Jump back to the input loop to wait for
the next user input.


# Routine to handle "Variable not declared" errors.
```

```
VariableNotDeclared:

    la a0, err_undeclared   # Load the address of the "Variable not
declared" message into a0.

    li a7, 4                # Syscall number for printing a string.

    ecall                   # Make the syscall to output the error
message.

    j input_loop            # Jump back to the input loop to wait for
the next user input.
```

4. store.s

```
.globl store_var # Make the `store_var` label globally accessible.

.globl store_loop # Make the `store_loop` label globally accessible.

.globl store_here # Make the `store_here` label globally accessible.

# Routine to store a variable in the symbol table.

store_var:

la t0, symbol_table # Load the address of the symbol table into t0.

store_loop:

lw t1, 0(t0) # Load the current slot's value (first word in the slot).

beqz t1, store_here # If the slot is empty (contains 0), jump to
`store_here` to store the variable.

addi t0, t0, 8 # Move to the next slot (assuming each slot is 8 bytes:
4 for the name, 4 for the address).

j store_loop # Repeat the loop to check the next slot.

store_here:

lb t1, 0(a0) # Load the first character of the variable name from the
input in `a0`.

sw t1, 0(t0) # Store the variable name into the current slot's first
word.

sw a1, 4(t0) # Store the address (passed in `a1`) into the second word
of the current slot.

ret # Return to the caller.
```

5. search.s

```
.globl next_var       # Make `next_var` label globally accessible.
```

```
.globl not_found      # Make `not_found` label globally accessible.


# Routine to find a variable in the symbol table.

find_var:

    la t0, symbol_table   # Load the starting address of the symbol
table into t0.

    lb t1, 0(a0)          # Load the first character of the variable
name we're searching for into t1.


find_loop:

    lw t2, 0(t0)          # Load the name stored in the current slot
of the symbol table into t2.

    beqz t2, not_found    # If the slot is empty (contains 0), the
variable was not found.


    bne t1, t2, next_var  # If the first character does not match,
check the next slot.


    # Found the variable:

    lw a1, 4(t0)          # Load the address associated with the
variable into a1.

    li a0, 1              # Indicate success by setting a0 to 1.

    ret                   # Return to the caller.


next_var:

    addi t0, t0, 8        # Move to the next slot (advance by 8 bytes:
4 for name and 4 for address).

    j find_loop           # Repeat the loop to check the next slot.


not_found:

    li a0, 0              # Indicate failure by setting a0 to 0.

    ret                   # Return to the caller.
```

6. print.s

```
.globl check_print_cmd
.globl check_print_paren
.globl handle_print_stmt
.globl skip_spaces_print
.globl got_print_var


check_print_cmd:
    lb t2, 0(t1)           # Load current char from `print_str`.
    beqz t2, check_print_paren  # If end of `print_str`, check for
'('.
    lb t3, 0(t0)           # Load current char from input buffer.
    beqz t3, prefix_diff   # If end of input buffer, no match.
    bne t2, t3, prefix_diff # If mismatch, no match.
    addi t0, t0, 1         # Advance input pointer.
    addi t1, t1, 1         # Advance `print_str` pointer.
    j check_print_cmd      # Continue comparison.


check_print_paren:
    lb t3, 0(t0)           # Load next char from input.
    li t2, '('             # Check for '('.
    bne t3, t2, prefix_diff # If not '(', no match.
    addi t0, t0, 1         # Skip '(' in input.
    li a0, 0               # Match found.
    ret


handle_print_stmt:
    # Skip "print(" in the input buffer.
    la t0, buffer
    addi t0, t0, 6         # Move pointer past "print(".


skip_spaces_print:
```

```
    lb t1, 0(t0)           # Load current char from input.
    beqz t1, not_found     # If end of input, error.
    li t2, ' '             # Check for spaces.
    bne t1, t2, got_print_var # If not a space, stop skipping.
    addi t0, t0, 1         # Skip space.
    j skip_spaces_print    # Repeat.

got_print_var:
    # Locate the variable in the symbol table.
    mv a0, t0              # Move variable name pointer to a0.
    jal find_var           # Call `find_var` to locate variable.
    beqz a0, not_found_var # If variable not found, show error.

    # Start printing the variable's matrix.
    mv s0, a1              # Load variable's matrix pointer.

    # Print the matrix format "[("
    li a0, '['
    li a7, 11              # Syscall for character output.
    ecall
    li a0, '('
    li a7, 11
    ecall

    # Print first complex number (real and imaginary parts).
    flw fa0, 0(s0)         # Load real part of the first number.
    li a7, 2               # Syscall for float output.
    ecall
    li a0, ','             # Print comma separator.
    li a7, 11
    ecall
    flw fa0, 4(s0)         # Load imaginary part of the first number.
```

```
li a7, 2
ecall

li a0, ')'            # Close the first complex number.
li a7, 11
ecall
li a0, ','            # Print comma separator.
li a7, 11
ecall
li a0, '('            # Open second complex number.
li a7, 11
ecall

# Print second complex number (real and imaginary parts).
flw fa0, 8(s0)        # Load real part of the second number.
li a7, 2
ecall
li a0, ','            # Print comma separator.
li a7, 11
ecall
flw fa0, 12(s0)       # Load imaginary part of the second number.
li a7, 2
ecall
li a0, ')'            # Close the second complex number.
li a7, 11
ecall
li a0, ']'            # Close the matrix.
li a7, 11
ecall

# Print newline character.
la a0, newline
```

```
    li a7, 4                # Syscall for string output.
    ecall

    j input_loop            # Return to input loop.
```

7. qubit.s

```
.globl handle_qubit_decl
.globl got_var_start
.globl new_variable
.globl found_equals
.globl parse_numbers
.globl update_storage
.globl parse_float
.globl done_parse

handle_qubit_decl:
    # Skip "qubit" and spaces
    la t0, buffer
    addi t0, t0, 5  # Skip "qubit"

    # Skip spaces to variable name
    skip_spaces_1:
        lb t1, 0(t0)
        li t2, ' '
        bne t1, t2, got_var_start
        addi t0, t0, 1
        j skip_spaces_1
got_var_start:
    # Save variable name pointer and current position
    mv s0, t0       # Save name pointer

    # Check if variable already exists
```

```
    mv a0, s0
    jal find_var
    beqz a0, new_variable    # If not found (a0 = 0), create new
variable

    # Variable exists, use existing location in a1
    mv t3, a1
    j find_equals    # Skip storage allocation, use existing location

new_variable:
    # Store new variable in symbol table
    mv a0, s0          # Variable name
    mv a1, s11         # Current storage location
    jal store_var
    mv t3, s11         # Save storage location for later

find_equals:
    lb t1, 0(t0)
    li t2, '='
    beq t1, t2, found_equals
    addi t0, t0, 1
    j find_equals

found_equals:
    # Skip equals and spaces
    addi t0, t0, 1
skip_spaces_2:
    lb t1, 0(t0)
    li t2, ' '
    bne t1, t2, find_bracket
    addi t0, t0, 1
    j skip_spaces_2
```

```
find_bracket:
    # Skip to opening '['
    lb t1, 0(t0)
    li t2, '['
    bne t1, t2, find_bracket_next
    j parse_numbers
find_bracket_next:
    addi t0, t0, 1
    j find_bracket

parse_numbers:
    # Skip '[' and '('
    addi t0, t0, 2

    # Parse first number (1)
    mv a0, t0
    jal parse_float
    mv t0, a1

    # Store first number
    fsw f0, 0(t3)

    # Skip comma
    addi t0, t0, 1

    # Parse second number (0)
    mv a0, t0
    jal parse_float
    mv t0, a1
    fsw f0, 4(t3)
```

```
    # Skip ),( - three characters
    addi t0, t0, 3

    # Parse third number (0)
    mv a0, t0
    jal parse_float
    mv t0, a1
    fsw f0, 8(t3)

    # Skip comma
    addi t0, t0, 1

    # Parse fourth number (1)
    mv a0, t0
    jal parse_float
    mv t0, a1
    fsw f0, 12(t3)

    # Only update storage pointer for new variables
    beq t3, s11, update_storage
    j input_loop

update_storage:
    addi s11, s11, 16
    j input_loop

parse_float:
    # Save return address
    addi sp, sp, -4
    sw ra, 0(sp)

    mv t4, a0          # Save string pointer
```

```
li t5, 0          # Integer part
li t6, 0          # Decimal part
li s2, 0          # Decimal position
li s3, 0          # Is negative

# Check for negative sign
lb t1, 0(t4)
li t2, '-'
bne t1, t2, parse_int_part
li s3, 1
addi t4, t4, 1

  parse_int_part:
      lb t1, 0(t4)
      li t2, '.'
      beq t1, t2, parse_decimal
      li t2, ','
      beq t1, t2, finish_parse
      li t2, ')'
      beq t1, t2, finish_parse

      # Convert char to int and add to total
      addi t1, t1, -48   # ASCII to int
      li t2, 10
      mul t5, t5, t2
      add t5, t5, t1

      addi t4, t4, 1
      j parse_int_part

  parse_decimal:
      addi t4, t4, 1     # Skip decimal point
```

```
parse_decimal_part:
    lb t1, 0(t4)
    li t2, ','
    beq t1, t2, finish_parse
    li t2, ')'
    beq t1, t2, finish_parse

    # Convert char to int and add to decimal
    addi t1, t1, -48   # ASCII to int
    li t2, 10
    mul t6, t6, t2
    add t6, t6, t1
    addi s2, s2, 1     # Increment decimal position

    addi t4, t4, 1
    j parse_decimal_part

finish_parse:
    # Convert to float
    fcvt.s.w f0, t5    # Convert integer part

    # If we have decimal part
    beqz t6, check_negative

    # Convert decimal part
    fcvt.s.w f1, t6
    li t1, 1
    li t2, 10
decimal_divide_loop:
    beqz s2, combine_parts
    mul t1, t1, t2
```

```
            addi s2, s2, -1
            j decimal_divide_loop

        combine_parts:
            fcvt.s.w f2, t1
            fdiv.s f1, f1, f2
            fadd.s f0, f0, f1

        check_negative:
            beqz s3, done_parse
            fneg.s f0, f0

done_parse:
    # Return new position in a1
    mv a1, t4

    # Restore return address
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

8. gateFunctions.s

```
.globl quantum_gate

.text
quantum_gate:
        # Prologue
        addi sp, sp, -32
        sw ra, 28(sp)
        sw s0, 24(sp)
        sw s1, 20(sp)
        sw s2, 16(sp)
```

```
sw s3, 12(sp)
sw s4, 8(sp)
sw s5, 4(sp)
sw s6, 0(sp)

# Load input parameters
lw t4, 0(a0)            # Load qubitnum
li t6, 1
bne t4, t6, two_qubit   # Branch if not single qubit

# Single qubit operations
lw t5, 0(a1)            # Load Qgate

# Gate selection logic
H: li t1, 0
   bne t5, t1, I
   la t0, Hgate
   j next
I: li t1, 1
   bne t5, t1, X
   la t0, Igate
   j next
X: li t1, 2
   bne t5, t1, Y
   la t0, Xgate
   j next
Y: li t1, 3
   bne t5, t1, Z
   la t0, Ygate
   j next
Z: li t1, 4
   bne t5, t1, S
```

```
            la t0, Zgate
            j next
         S: li t1, 5
            bne t5, t1, T
            la t0, Sgate
            j next
         T: li t1, 6
            bne t5, t1, SN
            la t0, Tgate
            j next
         SN: li t1, 7
            bne t5, t1, quantum_exit
            la t0, SNgate
            j next

two_qubit:
           la s0, qubit2
           la s1, qubit3
           la s3, tensor
           li t1, 2
           mv t2, t1

tensorprod:
          flw ft0, 0(s0)     # Load values
          flw ft1, 4(s0)
          flw ft2, 0(s1)
          flw ft3, 4(s1)

          # Compute products
          fmul.s ft4, ft0, ft2
          fmul.s ft5, ft1, ft3
          fmul.s ft6, ft0, ft3
```

```
fmul.s ft7, ft1, ft2
fsub.s ft8, ft4, ft5
fadd.s ft9, ft6, ft7

# Store results
fsw ft8, 0(s3)
fsw ft9, 4(s3)
addi s3, s3, 8
addi s1, s1, 8
addi t1, t1, -1
bgt t1, zero, tensorprod
addi t2, t2, -1
addi s0, s0, 8
addi s1, s1, -16
li t1, 2
bgt t2, zero, tensorprod

# Two-qubit gate selection
lw t5, 0(a2)
CN1: li t1, 0
   bne t5, t1, CN2
   la t0, CNgate1
   j next
CN2: li t1, 1
   bne t5, t1, CN3
   la t0, CNgate2
   j next
CN3: li t1, 2
   bne t5, t1, CN4
   la t0, CNgate3
   j next
CN4: li t1, 3
```

```
   bne t5, t1, SW1
   la t0, CNgate4
   j next

SW1: li t1, 4
   bne t5, t1, SW2
   la t0, SWgate1
   j next
SW2: li t1, 5
   bne t5, t1, SW3
   la t0, SWgate2
   j next
SW3: li t1, 6
   bne t5, t1, SW4
   la t0, SWgate3
   j next
SW4: li t1, 7
   bne t5, t1, CY1
   la t0, SWgate4
   j next

CY1: li t1, 8
   bne t5, t1, CY2
   la t0, CYgate1
   j next
CY2: li t1, 9
   bne t5, t1, CY3
   la t0, CYgate2
   j next
CY3: li t1, 10
   bne t5, t1, CY4
   la t0, CYgate3
```

```
            j next
         CY4: li t1, 11
            bne t5, t1, CZ1
            la t0, CYgate4
            j next

         CZ1: li t1, 12
            bne t5, t1, CZ2
            la t0, CZgate1
            j next
         CZ2: li t1, 13
            bne t5, t1, CZ3
            la t0, CZgate2
            j next
         CZ3: li t1, 14
            bne t5, t1, CZ4
            la t0, CZgate1
            j next
         CZ4: li t1, 15
            bne t5, t1, quantum_exit
            la t0, CZgate1
            j next


next:   la t2, res
        la a6, row1
        beq t4, t6, next1
        la t2, res_2
        la a6, row1_2
next1:  lw t3, 0(a6)
nextrow: fmv.s.x ft10, zero
         fmv.s.x ft11, zero
```

```
          la s2, col2
          la t1, qubit1
          beq t4, t6, next2
          la s2, col2_2
          la t1, tensor
next2:  lw t5, 0(s2)
nextcol: la s3, row2
          beq t4, t6, next3
          la s3, row2_2
next3:  lw s5, 0(s3)
         mv s4, zero
         mv s7, zero

dotprod: flw ft0, 0(t0)
          flw ft1, 4(t0)
          flw ft2, 0(t1)
          flw ft3, 4(t1)
          fmul.s ft4, ft0, ft2
          fmul.s ft5, ft1, ft3
          fmul.s ft6, ft0, ft3
          fmul.s ft7, ft1, ft2
          fsub.s ft8, ft4, ft5
          fadd.s ft9, ft6, ft7
          fadd.s ft10, ft10, ft8
          fadd.s ft11, ft11, ft9
          addi t0, t0, 8
          slli s8, t5, 3
          add t1, t1, s8
          addi s5, s5, -1
          bne s5, zero, dotprod
          fsw ft10, 0(t2)
          fsw ft11, 4(t2)
```

```
        addi t2, t2, 8

        addi t5, t5, -1
        beq t5, zero, skip
        slli s9, s5, 3
        sub t0, t0, s9

        mul s10, s9, t5
        li s11, 8
        sub s5, s11, s10

        add t1, t1, s5
        j nextcol

skip:   addi t3, t3, -1
        bne t3, zero, nextrow

copy_results:
        mv t2, a3
        li s7, 4
        beq t4, t6, copy_loop
        li s7, 8

copy_loop:
        flw ft0, 0(t2)
        fsw ft0, 0(a3)
        addi t2, t2, 4
        addi a3, a3, 4
        addi s7, s7, -1
        bne s7, zero, copy_loop

quantum_exit:
```

```
    # Epilogue
    lw s6, 0(sp)
    lw s5, 4(sp)
    lw s4, 8(sp)
    lw s3, 12(sp)
    lw s2, 16(sp)
    lw s1, 20(sp)
    lw s0, 24(sp)
    lw ra, 28(sp)
    addi sp, sp, 32
    ret
```

9. onequbitGate.s

```
.globl handle_qubitop1_stmt

.data
    Hgate_str: .string "H"
    Igate_str: .string "I"
    Xgate_str: .string "X"
    Ygate_str: .string "Y"
    Zgate_str: .string "Z"
    Sgate_str: .string "S"
    Tgate_str: .string "T"
    SNgate_str: .string "SN"
    arrow_str: .string "->"
    space_str: .string " "
    equals_str: .string "="

.text
handle_qubitop1_stmt:
    # Prologue
    addi sp, sp, -32
    sw ra, 28(sp)
```

```
    sw s0, 24(sp)
    sw s1, 20(sp)
    sw s2, 16(sp)

    # Load buffer address directly into s0
    la s0, buffer

    # Skip "qubitop1" by searching for space
    skip_command_1:
        lb t1, 0(s0)
        li t0, ' '
        beq t1, t0, find_var_start_1
        addi s0, s0, 1
        j skip_command_1

    find_var_start_1:
        # Skip any additional spaces
        addi s0, s0, 1
        lb t1, 0(s0)
        li t0, ' '
        beq t1, t0, find_var_start_1

    # Look up variable in symbol table
    mv a0, s0      # Move current buffer position to a0
    jal ra, find_var  # Call with proper link register
    beqz a0, parse_error  # Changed to parse_error instead of
VariableNotDeclared
    mv s1, a1      # Save variable address

    # Skip variable name
    addi s0, s0, 1
```

```
find_equals_1:
    lb t1, 0(s0)
    li t0, '='
    bne t1, t0, skip_ws1_1
    addi s0, s0, 1
    j find_gate_1

skip_ws1_1:
    addi s0, s0, 1
    j find_equals_1

find_gate_1:
    # Skip whitespace
    lb t1, 0(s0)
    li t0, ' '
    bne t1, t0, check_gate_1
    addi s0, s0, 1
    j find_gate_1

check_gate_1:
    # Store gate pointer
    mv s2, s0

    # Check against each gate string
    la t0, Hgate_str
    mv t1, s2
    jal ra, starts_with_char1
    beqz a0, set_h_gate

    la t0, Igate_str
    mv t1, s2
    jal ra, starts_with_char1
```

```
beqz a0, set_i_gate

la t0, Xgate_str
mv t1, s2
jal ra, starts_with_char1
beqz a0, set_x_gate

la t0, Ygate_str
mv t1, s2
jal ra, starts_with_char1
beqz a0, set_y_gate

la t0, Zgate_str
mv t1, s2
jal ra, starts_with_char1
beqz a0, set_z_gate

la t0, Sgate_str
mv t1, s2
jal ra, starts_with_char1
beqz a0, set_s_gate

la t0, Tgate_str
mv t1, s2
jal ra, starts_with_char1
beqz a0, set_t_gate

la t0, SNgate_str
mv t1, s2
jal ra, starts_with
beqz a0, set_sn_gate
```

```
        # Invalid gate
        j parse_error

    set_h_gate:
        li s2, 0
        j find_arrow
    set_i_gate:
        li s2, 1
        j find_arrow
    set_x_gate:
        li s2, 2
        j find_arrow
    set_y_gate:
        li s2, 3
        j find_arrow
    set_z_gate:
        li s2, 4
        j find_arrow
    set_s_gate:
        li s2, 5
        j find_arrow
    set_t_gate:
        li s2, 6
        j find_arrow
    set_sn_gate:
        li s2, 7
        j find_arrow

find_arrow:
    lb t1, 0(s0)
    li t0, '-'
    beq t1, t0, check_arrow
```

```
    addi s0, s0, 1
    j find_arrow

    check_arrow:
        lb t1, 1(s0)
        li t0, '>'
        bne t1, t0, parse_error

        addi s0, s0, 2
    skip_ws2:
        lb t1, 0(s0)
        li t0, ' '
        bne t1, t0, find_target_qubit
        addi s0, s0, 1
        j skip_ws2

find_target_qubit:
    mv a0, s0
    jal ra, find_var  # Added ra to jal
    beqz a0, parse_error

    # Set up quantum_gate call
    la a0, qubitnum
    la a1, Qgate
    sw s2, 0(a1)
    la a2, Qgate_2
    mv a3, s1          # Use saved variable address

    jal ra, quantum_gate

# After quantum_gate call, store result in the variable
    la t0, res         # Load address of result
```

```
    lw t1, 0(t0)      # Load first word of result
    sw t1, 0(s1)      # Store first word to variable
    lw t1, 4(t0)      # Load second word
    sw t1, 4(s1)      # Store second word
    lw t1, 8(t0)      # Load third word
    sw t1, 8(s1)      # Store third word
    lw t1, 12(t0)     # Load fourth word
    sw t1, 12(s1)     # Store fourth word

    li a0, 0           # Success return value
    j parse_exit

parse_error:
    li a0, -1

parse_exit:
    # Epilogue
    lw ra, 28(sp)
    lw s0, 24(sp)
    lw s1, 20(sp)
    lw s2, 16(sp)
    addi sp, sp, 32
    ret
```

10. twoQubitGate.s

```
.globl handle_qubitop2_stmt
.data
    CNgate1_str: .string "CN1"
    CNgate2_str: .string "CN2"
    CNgate3_str: .string "CN3"
    CNgate4_str: .string "CN4"
```

```
    SWgate1_str: .string "SW1"
    SWgate2_str: .string "SW2"
    SWgate3_str: .string "SW3"
    SWgate4_str: .string "SW4"

    CYgate1_str: .string "CY1"
    CYgate2_str: .string "CY2"
    CYgate3_str: .string "CY3"
    CYgate4_str: .string "CY4"

    CZgate1_str: .string "CZ1"
    CZgate2_str: .string "CZ2"
    CZgate3_str: .string "CZ3"
    CZgate4_str: .string "CZ4"

.text
handle_qubitop2_stmt:
    # Prologue
    addi sp, sp, -32
    sw ra, 28(sp)
    sw s0, 24(sp)
    sw s1, 20(sp)
    sw s2, 16(sp)
    sw s3, 12(sp)
    sw s4, 8(sp)

    # Load buffer address
    la s0, buffer
    addi s0,s0, 8    # Skip "qubitop2" by searching for space

    skip_command_2:
    lb t1, 0(s0)
```

```
li t0, ' '
skip_leading_spaces:
    beq t1, t0, skip_leading_spaces_continue
    j syntaxError
skip_leading_spaces_continue:
    addi s0, s0, 1
    lb t1, 0(s0)
    bne t1, t0, find_var_start_2
    j skip_leading_spaces

find_var_start_2:
    # Skip any additional spaces
    addi s0, s0, 1
    lb t1, 0(s0)
    li t0, ' '
    beq t1, t0, find_var_start_2

    mv a0, s0
    jal ra, find_var
    beqz a0, parse_error
    mv s1, a1

    addi s0, s0, 1

    find_equals_2:
        lb t1, 0(s0)
        li t0, '='
        skip_spaces_before_equals:
            beq t1, t0, found_equalss
            li t2, ' '
            bne t1, t2, syntaxError
            addi s0, s0, 1
```

```
            lb t1, 0(s0)
            j skip_spaces_before_equals
        found_equalss:
            addi s0, s0, 1
            j find_gate_2


    # Parse the gate type
    find_gate_2:
        # Skip whitespace
        lb t1, 0(s0)
        li t0, ' '
        bne t1,t0, check_gate_2
        addi s0, s0, 1
        j find_gate_2
check_gate_2:
    # Save gate pointer
    mv s2, s0

    # Match gate strings
    la t0, CNgate1_str
    mv t1, s2
    jal ra, starts_with
    beqz a0, set_CN1_gate

    la t0, CNgate2_str
    mv t1, s2
    jal ra, starts_with
    beqz a0, set_CN2_gate

    la t0, CNgate3_str
    mv t1, s2
```

```
jal ra, starts_with
beqz a0, set_CN3_gate

la t0, CNgate4_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CN4_gate

la t0, SWgate1_str
mv t1, s2
jal ra, starts_with
beqz a0, set_SW1_gate

la t0, SWgate2_str
mv t1, s2
jal ra, starts_with
beqz a0, set_SW2_gate

la t0, SWgate3_str
mv t1, s2
jal ra, starts_with
beqz a0, set_SW3_gate

la t0, SWgate4_str
mv t1, s2
jal ra, starts_with
beqz a0, set_SW4_gate

la t0, CZgate1_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CZ1_gate
```

```
la t0, CZgate2_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CZ2_gate

la t0, CZgate3_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CZ3_gate

la t0, CZgate4_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CZ4_gate

la t0, CYgate1_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CY1_gate

la t0, CYgate2_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CY2_gate

la t0, CYgate3_str
mv t1, s2
jal ra, starts_with
beqz a0, set_CY3_gate

la t0, CYgate4_str
```

```
    mv t1, s2
    jal ra, starts_with
    beqz a0, set_CY4_gate

    j parse_error

set_CN1_gate:
    li s2, 0
    j find_first_var
set_CN2_gate:
    li s2, 1
    j find_first_var
set_CN3_gate:
    li s2, 2
    j find_first_var
set_CN4_gate:
    li s2, 3
    j find_first_var

set_SW1_gate:
    li s2, 4
    j find_first_var
set_SW2_gate:
    li s2, 5
    j find_first_var
set_SW3_gate:
    li s2, 6
    j find_first_var
set_SW4_gate:
    li s2, 7
    j find_first_var
```

```
    set_CY1_gate:
        li s2, 8
        j find_first_var
    set_CY2_gate:
        li s2, 9
        j find_first_var
    set_CY3_gate:
        li s2, 10
        j find_first_var
    set_CY4_gate:
        li s2, 11
        j find_first_var

    set_CZ1_gate:
        li s2, 12
        j find_first_var
    set_CZ2_gate:
        li s2, 13
        j find_first_var
    set_CZ3_gate:
        li s2, 14
        j find_first_var
    set_CZ4_gate:
        li s2, 15
        j find_first_var

    # Parse input variables (similar to earlier)
find_first_var:
    # Skip whitespace after gate name to find first variable
    skip_ws2_2:
        lb t1, 0(s0)
        li t0, ' '
```

```
        bne t1, t0, parse_first_var
        addi s0, s0, 1
        j skip_ws2_2

    parse_first_var:
        mv a0, s0      # Save the address of the first input variable
        jal ra, find_var
        beqz a0, parse_error_2
        mv s3, a1      # Save the address of the first input
variable's value

    # Find comma between variables
    find_comma:
        lb t1, 0(s0)
        li t0, ','
        skip_spaces_before_comma:
            li t2, ' '
            beq t1, t0, found_comma
            bne t1, t2, syntaxError
            addi s0, s0, 1
            lb t1, 0(s0)
            j skip_spaces_before_comma
        found_comma:
            addi s0, s0, 1
        skip_spaces_after_comma:
            lb t1, 0(s0)
            li t2, ' '
            bne t1, t2, parse_second_var
            addi s0, s0, 1
            j skip_spaces_after_comma

    parse_second_var:
```

```
        mv a0, s0      # Save the address of the second input variable
        jal ra, find_var
        beqz a0, parse_error
        mv s4, a1      # Save the address of the second input
variable's value
        j apply_two_qubit_gate

    # Apply two-qubit gate and store the result
apply_two_qubit_gate:
    # Prepare to call the quantum gate function
    la a0, qubitnum
    li t0, 2

    sw t0, 0(a0)
    la a1, Qgate

    sw s2, 0(a1)
    mv a2, s2

    la t0, qubit1
    mv a3, t0

    # Store input qubits' values before calling
    la t0, qubit2
    lw t1, 0(s3)
    sw t1, 0(t0)
    lw t1, 4(s3)
    sw t1, 4(t0)

    la t0, qubit3
    lw t1, 0(s4)
    sw t1, 0(t0)
```

```
    lw t1, 4(s4)
    sw t1, 4(t0)

    jal ra, quantum_gate
    beqz a0, parse_error
    j store_result

    # Store result and exit
store_result:
    # Load the result from the `res` array
    la t0, res_2         # Load the address of the result
    lw t1, 0(t0)         # First word of the result
    sw t1, 0(s1)         # Store the first word in the output variable
    lw t1, 4(t0)         # Second word of the result
    sw t1, 4(s1)         # Store the second word
    lw t1, 8(t0)         # Third word
    sw t1, 8(s1)         # Store the third word
    lw t1, 12(t0)        # Fourth word
    sw t1, 12(s1)        # Store the fourth word

    li a0, 0             # Success return value
    j parse_exit

    parse_error_2:
        li a0, -1           # Error return value

    parse_exit_2:
        # Epilogue
        lw ra, 28(sp)
        lw s0, 24(sp)
        lw s1, 20(sp)
        lw s2, 16(sp)
```

```
lw s3, 12(sp)
lw s4, 8(sp)
addi sp, sp, 32
ret
```