1_Structure_Types_QASM

December 23, 2025

1 Task 8.1

1.1 Objective 1 : QASM Types and Casting

1.1.1 Identifiers

- Start with [A-Za-z], an underscore or an element from the Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl
- set of permissible continuation characters consists of all members of the aforementioned character sets with the addition of decimal numerals [0-9]
- may not override a reserved identifier

1.1.2 Variables

- named according to the rules for identifiers
- may be assigned values within a program
- can be initialized on declaration if they represent a classical type
- Declaration and initialization must be done one variable at a time

```
qubit q0;
qubit q1
qubit q2;
```

```
int[32] a;
float[32] b = 5.5;
bit[3] c;
bool my_bool = false;
```

1.1.3 Quantum Types

Qubits

- qubit
- qubit[size]
- qreg will be removed in teh future
- Global Variables

```
include "stdgates.inc";

qubit[5] q1;
const uint SIZE = 4;
uint runtime_u = 2;
```

```
qubit[SIZE] q2;
x q1[0];
z q2[SIZE - 2];
x q1[runtime_u];
```

Physical Qubits

- can be referenced by syntax $0,$1, ...
- can not be declared
- global

```
qubit gamma;
qubit  ;
qubit[20] qubit_array;
CX $0, $1;
```

1.1.4 Classical scalar types

Classical bits and registers

- bit
- bit[size]
- creg

```
bit[20] bit_array;
bit[8] name = "00001111";
```

Integers

- int[size]
- int
- unit[size]
- uint
- bit level operations can not be done on types without specified width

```
uint[32] my_uint = 10;
int[16] my_int;
my_int = int[16](my_uint);
```

Floating point numbers

- float[size]
- float

```
float[32] my_float =  ;
float my_machine_float = 2.3;
```

Void Type

- void do not return value

Angles

- angle[size] same as unint, 1 –> 2 * / 2^size
- arithmetic operations are defined by unsigned-integer arithmetic

```
angle[20] my_angle =  / 2;
angle my_machine_angle;


angle[4] my_pi =  ;  // "1000"
angle[6] my_pi_over_two =  /2;  // "010000"
angle[8] my_angle = 7 * (  / 8);  // "01110000"
```

Complex Numbers

- complex[float[size]]
- imaginary is written same as integer or floating point followed by `im`
- real() and imag() can be used to extract real and imaginary components
- may not be supported by real hardware

```
complex[float[64]] c;
c = 2.5 + 3.5im;
complex[float] d = 2.0+sin( /2) + (3.1 * 5.5 im);
float d_real = real(d);
```

Boolean Types

- bool name –> ture , false
- can be converted from `bit` to Boolean using `bool(c)`

```
bit my_bit = 0;
bool my_bool;
// Assign a cast bit to a boolean
my_bool = bool(my_bit);
```

1.1.5 Compile Time Constants

- const
- const types are required when specifying widths
- all scalar literals const

```
// Valid
const uint SIZE = 32;
qubit[SIZE] q1;  // Declares a 32-qubit register called `q1`.
int[SIZE] i1;

//Invalid
uint runtime_size = 32;
qubit[runtime_size] q2;
int[runtime_size] i2;
```

“‘ //Valid const float[64] f1 = 2.5; uint[8] runtime_u = 7;

const int[8] i1 = int8; // `i1` has compile-time value 2 const uint u1 = 2 * uint(f1); // `u1` has compile-time value 4

// Invalid const bit[2] b1 = bit2; // `float[64]` cannot be cast to `bit[2]` const int[16] i2 = int16; // Casting runtime values is not `const`

Built-in Constants

- , , e

Built-in constant expression functions

- arccos , arcsin , arctan
- ceiling , floor
- cos, sin , tan
- exp, log
- mod, pow , sqrt
- popcount
- rotl , rotr

1.1.6 Literals

- int –> decimal , 0x hex, 0x with _ , 0X uppercase , 0o ocatal , 0b binary , 0B with _ , large values with _
- float –> (d)+.(d)* , .(d)+ , scientific notation e10 , 2e+1 , 2.0E-1
- boolean –> true or false
- bit string –> ” zeros or ones with or without _ for readability”
- timing –> duration with ns, s, us, ms, s or dt (backend dependant)

1.1.7 Arrays

- Static size
- first argument is base type of the array

```
array[int[32], 5] myArray = {0, 1, 2, 3, 4};
array[float[32], 3, 2] multiDim = {{1.1, 1.2}, {2.1, 2.2}, {3.1, 3.2}};

int[32] firstElem = myArray[0]; // 0
int[32] lastElem = myArray[4]; // 4
int[32] alsoLastElem = myArray[-1];
```

- can not be declared inside function or gate
- total maximum dimensions is 7

1.1.8 Timing

Duration

```
duration one_second = 1000ms;
duration thousand_cycles = 1000dt;
```

```
duration two_seconds = one_second + 1s;
duration c = durationof({x $3;});
```

Strech

- subtype of duration
- resolved at compile time

1.1.9 Aliasing

- use `let` to allow qubits and registers to be refered to by other names

```
qubit[5] q;
// myreg[0] refers to the qubit q[1]
let myreg = q[1:4];
```

1.1.10 Index Set and Slicing

Register concatenation and Slicing

- use `++` to concatenate registers of the same type
- registers can be indexed and used as a subset register of the same type
- index can be sepcified as signed or unsigned integer , comman separated list of integers or a range a:b / a:c:b

```
qubit[2] one;
qubit[10] two;
// Aliased register of twelve qubits
let concatenated = one ++ two;
// First qubit in aliased qubit array
let first = concatenated[0];
// Last qubit in aliased qubit array
let last = concatenated[-1];
// Qubits zero, three and five
let qubit_selection = two[{0, 3, 5}];
// First seven qubits in aliased qubit array
let sliced = concatenated[0:6];
// Every second qubit
let every_second = concatenated[0:2:12];
// Using negative ranges to take the last 3 elements
let last_three = two[-4:-1];
// Concatenate two alias in another one
let both = sliced ++ last_three;
```

Classical bit value Slicing

- access bits from (int,uint,angle) using index similar to register slicing

```
int[32] myInt = 15; // 0xF or 0b1111
bit[1] lastBit = myInt[0]; // 1
bit[1] signBit = myInt[31]; // 0
```

```
bit[1] alsoSignBit = myInt[-1]; // 0

bit[16] evenBits = myInt[0:2:31]; // 3
bit[16] upperBits = myInt[-16:-1];
bit[16] upperReversed = myInt[-1:-16];

myInt[4:7] = "1010"; // myInt == 0xAF
```

- access array elements bits

```
array[int[32], 5] intArr = {0, 1, 2, 3, 4};
// Access bit 0 of element 0 of intArr and set it to 1
intArr[0][0] = 1;
// lowest 5 bits of intArr[4] copied to b
bit[5] b = intArr[4][0:4];
```

Array concatenation and slicing

- arrays of the same types can be concatenated
- array contents are copied

```
array[int[8], 2] first = {0, 1};
array[int[8], 3] second = {2, 3, 4};

array[int[8], 5] concat = first ++ second;
array[int[8], 4] selfConcat = first ++ first;

array[int[8], 2] secondSlice = second[1:2]; // {3, 4}

// slicing with assignment
second[1:2] = first[0:1]; // second == {2, 0, 1}

array[int[8], 4] third = {5, 6, 7, 8};
// combined slicing and concatenation
selfConcat[0:3] = first[0:1] ++ third[1:2];
// selfConcat == {0, 1, 6, 7}

subroutine_call(first ++ third) // forbidden
subroutine_call(selfConcat) // allowed
```

- can be sliced using range and can be indexed using an integer
- cannot be indexed using comma sperated list of integers

```
int[8] scalar;
array[int[8], 2] oneD;
array[int[8], 3, 2] twoD; // 3x2
array[int[8], 3, 2] anotherTwoD; // 3x2
array[int[8], 4, 3, 2] threeD; // 4x3x2
array[int[8], 2, 3, 4] anotherThreeD; // 2x3x4
```

```
threeD[0, 0, 0] = scalar; // allowed
threeD[0, 0] = oneD; // allowed
threeD[0] = twoD; // allowed

threeD[0] = oneD; // error - shape mismatch
threeD[0, 0] = scalar // error - shape mismatch
threeD = anotherThreeD // error - shape mismatch

twoD[1:2] = anotherTwoD[0:1]; // allowed
twoD[1:2, 0] = anotherTwoD[0:1, 1]; // allowed
```

1.1.11 Casting Specifics

- classical –> bool, int, uint, float, complex
- special –> bit, angle, duration , stretch
- the lesser of the two types are cast in to the greater type incase of different types on a binary operation

Allowed casts

- qubit and duration can not be casted to or from any other types
- bit can be casted into all types other than float, qubit and duration
- angle can be casted to bool and bit
- float can be casted to bool , int, uint and angle
- uint can be casted to bool, int, float and bit
- int can be casted to bool, uint, float and bit
- bool can be casted to int,uint,float and bit