Using the Deutsch-Jozsa algorithm to determine parts of an array and apply a specified function to each independent part

Samir Lipovaca
slipovaca@nyse.com

Keywords: Deutsch-Jozsa, algorithm, quantum computer, vector, partition, array, register

Abstract: Using the Deutsch-Jozsa algorithm, we will develop a method for solving a class of problems in which we need to determine parts of an array and then apply a specified function to each independent part. Since present quantum computers are not robust enough for code writing and execution, we will build a model of a vector quantum computer that implements the Deutsch-Jozsa algorithm from a machine language view using the APL2 programming language. The core of the method is an operator (DJBOX) which allows evaluation of an arbitrary function \( f \) by the Deutsch-Jozsa algorithm. Two key functions of the method are GET_PARTITION and CALC_WITH_PARTITIONS. The GET_PARTITION function determines parts of an array based on the function \( f \). The CALC_WITH_PARTITIONS function determines parts of an array based on the function \( f \) and then applies another function to each independent part. We will imagine the method is implemented on the above vector quantum computer. We will show that the method can be successfully executed.

1) Introduction

The Deutsch-Jozsa algorithm is one of the first examples of a quantum algorithm. Quantum algorithms are designed for execution on quantum computers. Quantum algorithms have the potential to be more efficient than classical algorithms by taking advantage of the quantum superposition and entanglement principles. We are given a black box which takes \( n \) qubits \( x_1, x_2, \ldots, x_n \) and computes a \( \{0,1\} \) valued function \( f(x_1, x_2, \ldots, x_n) \). We know that the function is either constant for all inputs or else \( f \) is balanced, that is, equal to 1 for exactly half of all possible inputs. The task is to determine whether \( f \) is constant or balanced. The Deutsch-Jozsa algorithm produces an answer that is always correct with just 1 evaluation of \( f \). For comparison, the best deterministic classical algorithm requires \( \frac{2^n}{2} + 1 \) queries.

A data structure is an arrangement of data in a computer’s memory. Data structures include arrays, linked lists, stacks, binary trees, and hash tables, among others. Algorithms manipulate the data in these structures in various ways, such as searching for a particular data item and sorting the data. The array is the most commonly used data storage structure. It is built into most programming languages. Consecutive memory cells are used to store an array’s data. For example, 10 consecutive memory cells store data for an array of 10 elements, one element per cell. Array elements are accessed using an index number. Usually, the first element is numbered 0, so that the indices in an array of 10 elements run from 0 to 9. For example, \( A[4] \) returns content of fifth element of the array \( A \).

Occasionally, it becomes necessary to apply a function independently to successive parts of an
array. For example, the problem may be to multiply all elements in each part of an array $A$. If the array $A$ contains integers smaller than 10 and the parts were as follows:

$$2 \ 2 \ 3 \ 1 \ 2 \ 7 \ 5 \ 2 \ 1$$

then the desired result would be $12 \ 2 \ 7 \ 10$. Essentially, in this class of problems [1] we need to determine parts (partitioning) of an array and then apply a specified function to each independent part. The function may be inherently difficult to calculate.

The following sections discuss an approach for solving this class of problems using the Deutsch-Jozsa algorithm. First, we must characterize the concept of “parts” and then combine that with the function and the array to produce the result. In the section 2.2 we describe partitioning in the APL2. The Deutsch-Jozsa algorithm is briefly outlined in the section 2.3. Then in the section 2.4 the APL2 simulation of the Deutsch-Jozsa algorithm is described. The APL2 simulation of a vector quantum computer that implements the Deutsch-Jozsa algorithm is exposed in the section 2.5. Finally, in the section 2.6, partitioning in the vector quantum computer that implements the Deutsch-Jozsa algorithm is described. We finish with Discussion and Conclusions.

(2) Methods and Results

2.1 Partition array

A simple and unique characterization [1] of any partitioning of an array $A$ is another array $B$ of the same length whose elements are 0 or 1. Let’s call the array $B$ a partition array. Each 1 in the partition array corresponds to the beginning of a part and the following 0’s correspond to the remaining elements in that part. The partition array used above would be characterized as

$$2 \ 2 \ 3 \ 1 \ 2 \ 7 \ 5 \ 2 \ 1$$
$$1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0$$

Any array of the same length as $A$ and whose elements are 0 or 1 and whose first element is 1 represents a partition array. It is easy to see that there are $2^{N-1}$ such partition arrays of length $N$. The number of parts represented by the partition array $B$ is the number of 1’s in $B$. In a given problem, $B$ is defined or determined by some function.

2.2 Partitioning in APL2

APL2 is a computer programming language that permits rapid development of applications and models of application design. APL2 deals with whole collections of data all at once. Arrays are a fundamental unit of computation. In APL2, an array is a rectangular arrangement of data called the items of the array. Each item is a number, character or another array. Every row of an array contains the same number of items and every column in an array contains the same number of items. This rectangular design is extended to collections of data organized along any number of independent directions.

The directions along which data in an array is arranged are the axes of the array. The number of directions is the rank of the array. In APL2, ranks higher than 2 are allowed up to some implementation limit (typically 64). APL2 has special names for arrays that have rank 0, 1, or 2. A
scalar is a single number (such as 5) or character that is not arranged along any axes and so has rank 0. A vector (such as 3 1 8 0) has one axis and a rank of 1. Notice in this terminology the array \( A \) in the section (1) is a vector. A matrix having rows and columns has two axes and a rank of 2.

Let’s suppose the array \( A \) has a rank of 2 and the following items

\[
\begin{align*}
A & \quad 1 0 0 \\
A & \quad 1 0 0 \\
A & \quad 2 0 \\
A & \quad 4 0 0 \\
B & \quad 3 0 \\
B & \quad 2 0 0 \\
B & \quad 3 0 0 \\
C & \quad 1 0 0 \\
C & \quad 1 0 0 \\
C & \quad 1 0 0 \\
\end{align*}
\]

The problem is to accumulate items of the second column into bins \( A, B \) and \( C \). For example, we could imagine a large array \( A \) in which the first column contains individual atoms in a protein and the second column contains their respective interaction energies with the remaining atoms. Then the problem would be to calculate for each atom a total interaction energy with the rest of the protein. In this example the array \( A \) is already ordered by the first column such that equal items are adjacent. Generally, the array \( A \) must be sorted by respective column before the array \( B \) is determined. The problem is solved in two steps. First, the array \( B \) is determined by

\[
B = A[:,1] - 1 \Phi A[:,1]
\]

where 1 refers to the first column of \( A \). The effect of \(-1 \Phi A[:,1]\) is to move the last item of the first column of \( A \) to be the first row item. \(-1 \Phi A[:,1]\) is displayed below in the second row. It is compared with \( A[:,1] \) and whenever items are different, the value of 1 is assigned to \( B \) (the third row below).

\[
\begin{array}{cccccccc}
A & A & A & A & B & B & B & C \\
C & A & A & A & B & B & B & C \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

Then we calculate totals for each partition

\[
T = (+/ + B) \circ A[:,2]
\]

where 2 refers to the second column of \( A \). \((+/ + B) \circ A[:,2]\) divides up the second column of \( A \) producing 3 parts and sum is applied to each part which is represented by \(+/\). Combining all together a final array \( R \)

\[
R = (, [:,] B / A[:,1]), T
\]

has the following elements

\[
\begin{align*}
A & \quad 6 2 0 \\
B & \quad 5 3 0 \\
C & \quad 3 0 0 \\
\end{align*}
\]
2.3 The Deutsch-Jozsa algorithm

Let us briefly outline specific steps of the algorithm [2]. The input state

\[ |\psi_0\rangle = |0\rangle^\otimes n |1\rangle \]  

(1)

consists of the query register which describes the state of \( n \) qubits all prepared in the \( |0\rangle \) state and the answer register in the \( |1\rangle \) state. After the Hadamard transform on the query register and the Hadamard gate on the answer register we obtain

\[ |\psi_1\rangle = \sum_{x \in \{0,1\}^n} \frac{|x\rangle}{\sqrt{2^n}} \left( |0\rangle - |1\rangle \right). \]  

(2)

The query register is a superposition of all values. The answer register is in an evenly weighted superposition of \( |0\rangle \) and \( |1\rangle \) states. Next, the function \( f \) is evaluated using the transformation

\[ U_f : |x, y\rangle \rightarrow |x, y \oplus f(x)\rangle, \]

(giving

\[ |\psi_2\rangle = \sum_{x} \frac{(-1)^{f(x)}|x\rangle}{\sqrt{2^n}} \left( |0\rangle - |1\rangle \right). \]  

(3)

Using a Hadamard transform on the query register in \( |\psi_2\rangle \) we have

\[ |\psi_3\rangle = \sum_{x} \sum_{z} \frac{(-1)^{x \cdot z + f(x)}|z\rangle}{\sqrt{2^n}} \left( |0\rangle - |1\rangle \right). \]  

(4)

Assuming that \( f \) is constant the amplitude for \( |0\rangle^\otimes n \) is +1 or -1, depending on the constant value \( f(x) \) takes. Because \( |\psi_3\rangle \) is of unit length it follows that all other amplitudes must be zero, and an observation will yield 0s for all qubits in the query register.

2.4 APL2 simulation of the Deutsch-Jozsa algorithm

All the APL2 code is listed in the Appendix. The Hadamard gate is defined as a global array \( H \) by the DEFINEHGATE function. The array \( H \) has the following items

\[
\begin{array}{cc}
0.7071067812 & 0.7071067812 \\
0.7071067812 & -0.7071067812
\end{array}
\]

This is a computer implementation of \( \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \). Up to \( \frac{1}{\sqrt{2}} \) constant, the APPLY_H_GATE function does the Hadamard gate action on the states \( |0\rangle \) or \( |1\rangle \) resulting in
where 0 \times 1 \text{ corresponds to } \frac{1}{\sqrt{2}} (|0\text{rangle} - |1\text{rangle}) \text{ when } H \text{ acts on } |1\text{rangle}. Similarly, 0 \times 1 \text{ represents } \frac{1}{\sqrt{2}} (|0\text{rangle} + |1\text{rangle}) \text{ when } H \text{ acts on } |0\text{rangle} \text{ which is expressed by } APPLY_H\_GATE\ 0. Finally, the HNGATE function simulates, up to a normalization constant, the Hadamard transform on the query register. For example, in the $|\psi_1\rangle$ state and $n = 3$, $\sum_{x \in \{0,1\}} \frac{|x\rangle}{\sqrt{2^n}}$ is represented by

$$000 + 001 + 010 + 011 + 100 + 101 + 110 + 111.$$ 

This is the output for HNGATE 0 0 0.

The Deutsch-Jozsa black box $U_f$ which performs transformation $|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$ is simulated in the DJBOX operator. The DJBOX operator simulates essential elements of the quantum circuit for the Deutsch-Jozsa algorithm. In the APL2 sense, an operator is a function that takes another function and variables as an input and perform some calculations. The DJBOX evaluates $f \cdot y \oplus f(x)$ is simulated by $YPLUSF(F \ X[I;])$ and we assume $f(x) = 1$ for all $x$. $\sum_{x} \sum_{z} \frac{(-1)^{x \times f(x)}}{2^n} |z\rangle$ is represented by an array with $n$ rows and columns. For example, assuming $n = 3$, this array (P3CONV) is

$$\begin{bmatrix}
0 & -1 & -2 & -3 & -4 & -5 & -6 & -7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & -1 & -2 & -3 & -4 & -5 & -6 & -7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & -1 & -2 & -3 & -4 & -5 & -6 & -7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & -1 & -2 & -3 & -4 & -5 & -6 & -7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{bmatrix}$$

where the query register states are represented in the decimal representation for a shorter notation. The act of observation is simulated by $+DF3CONV$ which is summation of all items in each column of P3CONV resulting in $0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0$. This is equivalent to an observation which yields 0s for all qubits in the query register.

### 2.5 APL2 simulation of a Vector Quantum Computer that implements the Deutsch-Jozsa algorithm

This section develops a simulation of a vector quantum computer that implements the Deutsch-Jozsa algorithm from a machine language view. In this view a computer is a device that has a way of remembering numbers and combining numbers to form new numbers. We think of the computer’s memory as an ordered list of individual memory cells, each of which can hold one number. This ordered
list is called main memory. Most of the computers have some specialized high-speed storage areas called registers where most computations are performed. A vector architecture has a set of vector registers, instructions to move values from main memory into vector registers, perform computations on vector registers, and move results back into main memory. In addition, some control registers record the number of items in the vector registers and other related information.

The essentials of the model that represents a modest machine are defined in the DEFINEMACHINE function (see Appendix). A 100-item vector is used to represent main memory. The Deutsch-Jozsa query register size $QR$ is the function parameter. This parameter determines the number of the Deutsch-Jozsa query register states $QRSIZE$ by

$$QRSIZE = 2^QR.$$ 

The four control registers are needed. The first defines the length of each of 5 vector registers and is called the section size $SS$. This number is identical to $QRSIZE$. The second control register $VCT$ records the number of items actually residing in the vector registers. This number can range from zero up to the section size. The third control register $PSS$ determines the number of parts of an array to which a specified function is applied independently. This number can range from zero up to $PSS$. We set $PSS$ equal to $QR$. In addition to 5 vector registers, there are $QRSIZE$ additional vector registers denoted as the Deutsch-Jozsa vector registers. These registers store parts of an array, one part per register, to which a specified function is applied independently. The last control register is a partition vector count register $PVCT$ which records the number of items residing in the Deutsch-Jozsa vector registers.

The memory and registers are initialized to dots (.) so it is easy to distinguish items that have been used from items that have not been used. In a real machine, the initial values would be probably zero. Executing DEFINEMACHINE 3, defines the machine with 3 qubits in the Deutsch-Jozsa query register. The SHOMACHINE function (see Appendix) depicts the machine contents:

```
SHOWMACHINE

MAIN MEMORY

VCT PVCT
1 : ......................... 0 0
26 : .........................
51 : .........................
76 : .........................

VECTOR REGISTERS

V1:..............
V2:..............
V3:..............
V4:..............
V5:..............

DJ QUERY REGISTER

DJ VECTOR REGISTERS

Q1: .

V1: ..............
V2: ..............
V3: ..............
V4: ..............
V5: ..............
V6: ..............
V7: ..............
V8: ..............
```
2.6 Partitioning in the Vector Quantum Computer that implements the Deutsch-Jozsa algorithm

The GET_PARTITION function (Appendix) determines parts of an array based on the specified function F3. This function always returns 1 and is evaluated by the Deutsch-Jozsa algorithm which is simulated in the DJBOX operator. For example, for the machine with 3 qubits in the Deutsch-Jozsa query register and the array $A$ from the section 2.2, the GET_PARTITION function returns respective partition vector stored in main memory at address 76:

```
DEFFINEMACHINE 3
GET_PARTITION A[:1]
QUERY REGISTER IN 0 STATE
QUERY REGISTER IN 0 STATE
MAIN MEMORY

VCT PVCT
1 : . 0 36 61 3 86 . . . 0 3
26 : A A A A B B B C C C
51 : C A A A A B B B C C
76 : 1 0 0 0 1 0 0 1 0 0

VECTOR REGISTERS
V1: C C A A B B B C
V2: C C A A A B B B
V3: 0 0 0 0 1 0 0 1
V4: ...
V5: ...

DJ QUERY REGISTER DJ VECTOR REGISTERS
Q1: 0 0 0
V1: ...
V2: ...
V3: ...
V4: ...
V5: ...
V6: ...
V7: ...
V8: ...
```

We assumed those bins into which to accumulate items are already defined outside the F3 function. It is possible to imagine a quite general function F3 which would determine parts (bins) based on constraints of the problem we are trying to solve. The section of the code that determines 0 or 1 to be stored in a vector register cell would be a problem specific. A respective partition vector would be again stored in main memory at address 76.
The CALC_WITH_PARTITIONS function (Appendix) determines parts of an array and then applies a specified function to each independent part. For example, for the machine with 3 qubits in the Deutsch-Jozsa query register and the array $A$ from the section 2.2, F2 is the function applied to each independent part. This function is evaluated by the Deutsch-Jozsa algorithm which is simulated in the DJBOX operator. The final result is stored in main memory at address 51:

| VECTORS | DJ QUERY REGISTER | DJ VECTOR REGISTERS |
|---------|-------------------|---------------------|
| V1: C C A A B B B B | V1: 4 100 100 20 400 | V2: . . . . . . . . |
| V2: C C A A A B B B | V2: 3 30 200 300 | V3: . . . . . . . . |
| V3: 62 0 53 0 300 | V3: 3 100 100 100 | V4: . . . . . . . . |
| V4: . . . . . . . . | V4: . . . . . . . . | V5: . . . . . . . . |
| V5: . . . . . . . . | V5: . . . . . . . . | V6: . . . . . . . . |
| V6: . . . . . . . . | V6: . . . . . . . . | V7: . . . . . . . . |
| V7: . . . . . . . . | V7: . . . . . . . . | V8: . . . . . . . . |
| V8: . . . . . . . . | V8: . . . . . . . . | DJ QUERIE | R REGISTER | DJ VECTOR REGISTERS |

In this example the F2 function sums elements of each partition. Obviously the section of the code that applies a specified function to each partition is a problem specific and needs to be rewritten for every problem we are trying to solve. Note that in the first part of the CALC_WITH_PARTITIONS function we determine a respective partition vector so we could use a function call to the GET_PARTITION function to do this task instead of explicit coding.

(3) Discussion

We used the Deutsch-Jozsa algorithm to solve a class of problems in which we need to determine parts of an array and then apply a specified function to each independent part. Since present quantum computers are not robust enough for code writing and execution, we developed an APL2 model of a vector quantum computer that implements the Deutsch-Jozsa algorithm from a machine language view. The model is based on the example of the vector architecture from the reference [3]. It requires two additional vector register sets: the Deutsch-Jozsa query register and the Deutsch-Jozsa vector registers.

The Deutsch-Jozsa query register corresponds to the query register of the Deutsch-Jozsa algorithm. The Deutsch-Jozsa vector registers store parts of an array, one part per register, to which a specified function is applied independently. The array parts and respective partition vector are determined in the remaining vector registers. The totality of the vector registers forms a framework for solving the above class of problems.
Two additional control registers are needed. One control register determines the number of parts of an array to which a specified function is applied independently. The other is a partition vector count register which records the number of items residing in the Deutsch-Jozsa vector registers.

We know that the function \( f \) evaluated by the Deutsch-Jozsa algorithm is either constant for all inputs or else \( f \) is balanced, that is, equal to 1 for exactly half of all possible inputs. The Deutsch-Jozsa algorithm produces an answer that is always correct with just 1 evaluation of \( f \). In our model we assume \( f(x) = 1 \) for all \( x \). \( f(x) \), before returns 1, does one of two possible computations while it is evaluated by the Deutsch-Jozsa algorithm. \( f(x) \) determines parts of an array or it applies a specified function to each independent part. For example, F3 function when evaluated by the DJBOX operator compares two vector register cell contents and store the result in the third vector register cell. The address of all three cells is equal to the decimal representation of the query register state \( x \). For example, if \( x = 100 \), the address is equal to 4. Similarly, F2 function sums elements of the vector stored in a Deutsch Jozsa vector register and stores the sum in the corresponding vector register. The addresses of the respective cells are constrained by the decimal representation of the query register state \( x \). Therefore, both the length of the vector registers and the number of the Deutsch-Jozsa vector registers are equal to the number of the query register states \( \text{QRSIZE} \).

The DJBOX operator is the core of the method, since the Deutsch-Jozsa algorithm needs to evaluate an arbitrary function \( f \) which returns always 1. The operator is the simplest design choice to “an arbitrary function \( f \)“ condition since the operator is a function that takes another function and variables as an input and perform some calculations. The operator concept is a part of the APL2 syntax, thus the APL2 seems a natural language for the model of a vector quantum computer that implements the Deutsch-Jozsa algorithm from a machine language view. An additional bonus of implementing the APL2 on a quantum computer is that the APL2 deals with whole collections of data all at once. This is similar to quantum parallelism which enables all possible values of a function to be evaluated simultaneously, even we apparently only evaluated the function once.

It was shown [4] that computer languages can be reduced to an algebraic representation. By algebraic we mean that the computer language can be represented with operator expressions using operators that have an algebra similar to that of the raising and lowering operators in quantum mechanics. To establish the algebraic representation we associate a harmonic raising operator \( a_i^+ \) and a lowering operator \( a_i \) with each memory location \( i \). These operators satisfy the commutation relations

\[
\begin{align*}
[a_i, a_j^+] &= \delta_{ij} \\
[a_i, a_j] &= 0 \\
[a_i^+, a_j^+] &= 0
\end{align*}
\]

where \( \delta_{ij} \) is 1 if \( i = j \) and zero otherwise. We also define a pair of raising and lowering operators for the register \( r \) and \( r^+ \) with commutation relations
\[ [r_i, r_j^+] = \delta_{ij} \]
\[ [r_i, r_j] = 0 \]
\[ [r_i^+, r_j^+] = 0. \]

The ground state of the quantum computer is the state with the values at all memory locations set to zero. It is represented by the vector \(|0,0,0,\ldots\rangle\). A general state of the quantum computer will be represented by a vector of the form
\[ |n, m, p, \ldots\rangle = \mathcal{N}(r^+)^n (a_0^+)^m (a_1^+)^p \ldots |0,0,0,\ldots\rangle \]
where \( \mathcal{N} \) is a normalization constant and with the first number being the value in the register, the second number the value at memory location 0, the third number the value at memory location 1, and so on.

Let us imagine that the APL2 is, indeed, implemented on a quantum computer with the vector architecture. All the APL2 statements are represented with the above raising and lowering operators. Then the code for both functions GET_PARTITION and CALC_WITH_PARTITIONS could be executed, without any modification, on such a computer. On the other hand, the DJBOX operator just simulates the Deutsch Jozsa black box, since the code is looping by the query register states:

\begin{verbatim}
L0: - > (QRSIZE < I+1) + LX # LOOP BY THE QUERY REGISTER STATES
LOAD_QUERY_REGISTER X[I:] # LOAD THE QUERY REGISTER WITH STATE X[I:]
# NOTE: X IS AN ARRAY OF ALL SUPERPOSITION VALUES IN THE QUERY REGISTER
T=HNGATE X[I:] # APPLY HADAMARD GATE ON THE STATE X[I:]
P= ~c(T='+') * (T='-') # IDENTIFY WHERE IS NOT + OR - (FIND STATES)
TTP=P<1 # ENCLOSE STATES
CONV=2 # = 2 EXPRESS EACH STATE IN DECIMAL REPRESENTATION (STATES ARE IN BINARY REPRESENTATION)
Z=~P/ < T # COLLECT ALL +,-
K=0 # INITIALIZE INNER LOOP VARIABLE
TTT=TT # INITIALIZE VARIABLE WHICH HOLDS RESULT OF LOOP ITERATION
L1: - > ((QRSIZE-1) < K+K+1) + L2 # LOOP QRSIZE-1 TIMES, BEGIN LOOP WITH THE SECOND STATE
TTT+TTT, (1+2*K)=('+) * (~1*CONV[I+K]) # IF + SIGN ADD STATE,
# IF - SIGN, ADD STATE MULTIPLIED BY -1
- > L1 # GO TO NEXT LOOP ITERATION
L2: # EXIT LOOP
P3CONV=P3CONV, [1](Y YPLUSF(F X[I:] )) * (0, TTT) # ADD 0 STATE TO TTT,
# MULTIPLY STATES WITH THE OUTPUT OF Y ADD MODULO 2 F(X) AND
# ADD THEM TO P3CONV
- > L0 # GO TO NEXT STATE IN THE QUERY REGISTER
LX: # EXIT OUTER LOOP
\end{verbatim}

Respective quantum computer code would take advantage of the quantum superposition and entanglement principles instead of looping and we could pack this code into a new machine code statement. The statement would replace above looping by the query register states. Such a modified DJBOX operator could be executed on the quantum computer with the vector architecture.

(4) Conclusions

In this paper we developed, using the Deutsch-Jozsa algorithm, a method to solve a class of problems in which we need to determine parts of an array and then apply a specified function to each independent part. To evaluate the method, since present quantum computers are not robust enough for code
writing and execution, we built an APL2 model of a vector quantum computer that implements the Deutsch-Jozsa algorithm from a machine language view. The core of the method is an operator which allows evaluation of an arbitrary function \( f \) by the Deutsch-Jozsa algorithm. The operators’ concept is already a part of the APL2 syntax. Although the APL2 dealing with whole collections of data all at once is similar to quantum parallelism, it is founded on an implicit looping through the data. Quantum parallelism does not need any consecutive processing. Thus the APL2 implemented on a vector quantum computer would have statements represented with operator expressions using operators that have an algebra similar to that of the raising and lowering operators in quantum mechanics. Taking advantage of the quantum superposition and entanglement principles, such an APL2 (“quantum APL2”) would be significantly more powerful and efficient than its classical version. We outlined how to adapt the DJBOX operator code so two key functions for the solution of the above class of the problems (GET_PARTITION, CALC_WITH_PARTITIONS) can be executed on the vector quantum computer. Once when the vector quantum computer with “quantum APL2” is robust enough for code writing and execution, the above method could be implemented and further explored.

A novel feature of our model of the vector quantum computer is the Deutsch-Jozsa vector registers. These registers are exclusively used for the Deutsch-Jozsa algorithm execution. Thus, the Deutsch-Jozsa algorithm is an integral part of the vector quantum computer architecture. Although we assumed a constant function \( f \) evaluated by the Deutsch-Jozsa algorithm, our method can be easily generalized for a balanced function too.

In this paper we were not concerned whether we have designed the most efficient vector quantum computer, nor how we could best implement it. We wanted to show that it is possible, in principle, to use the Deutsch-Jozsa algorithm as an integral part of the vector quantum architecture to solve the class of problems in which we need to determine parts of an array and then apply a specified function to each independent part. Since the Deutsch-Jozsa algorithm takes advantage of the quantum superposition and entanglement principles, we believe our method would be more powerful and efficient than its classical version.

Acknowledgments
The results presented on these pages are the outcome of independent research not supported by any institution or government grant.

References

[1] “Boolean Functions and Techniques”, Second Edition, p.4, 1982, STSC, Inc.
Bob Smith’s paper that was referenced here, was published in the APL79 Conference Proceedings, Part 1.

[2] M. A. Nielsen and I. L. Chuang, p.34, “Quantum Computation and Quantum Information”,
(Cambridge University Press, Cambridge, 2000).

[3] James A. Brown, Sandra Pakin, Raymond P. Polivka, “APL2 at Glance”, p.342, 1988, Prentice Hall,
Englewood Cliffs, N.J.

[4] Stephen Blaha, “Cosmos and Consciousness”, 2000, 1stBooks Library, Bloomington, IN.
Appendix

DEFINEMACHINE function defines global variables of the vector machine.

```
DEFINEMACHINE A:IO
  a DESCRIPTION: DEFINE VECTOR MACHINE GLOBALS
  `IO=1 a ORIGIN
  a DEFINE A DEFINES Deutsch-Jossa QUERY REGISTER SIZE
  a DEFINE GLOBAL MEMORY AREAS
  MM=100p,' a MAIN MEMORY
  QR=A a SET Deutsch-Jossa QUERY REGISTER SIZE
  QRSIZE=2*QR a NUMBER OF Deutsch-Jossa QUERY REGISTER STATES
  SS=QRSIZE a SECTION SIZE
  PSS=QR a PARTITION SECTION SIZE
  VR=(5 SS)p,' a 5 VECTOR REGISTERS
  VCT=0 a VECTOR COUNT REGISTER
  PVC+0 a PARTITION VECTOR COUNT REGISTER
  DJ+(1 QR)p,' a Deutsch-Jossa QUERY REGISTER
  DJVR=(QRSIZE(2*QRSIZE))p,' a Deutsch-Jossa VECTOR REGISTERS
```

SHOMACHINE function displays contents of the vector machine

```
SHOMACHINE:IO:R:N
  a DESCRIPTION: DEPICTS THE MACHINE'S CONTENTS
  `IO=1 a ORIGIN
  a INPUT: NONE
  a DEPICT MAIN MEMORY, VECTOR
  a AND PARTITION VECTOR COUNT REGISTERS
  N='MAIN MEMORY' 'VCT' 'PVCT'
  R=(MM)+25 a DEFINE 4 ROWS FOR MAIN MEMORY
  N,[0.5]((1 p1+25~1+R),':',R 25pMM)VCT PVCT
  a DEPICT VECTOR REGISTERS
  N='VECTOR REGISTERS'
  N,[0.5](('V',®5 1p15),':',VR)(' ')
  a DEPICT Deutsch-Jossa QUERY AND VECTOR
  a REGISTERS
  N='DJ QUERY REGISTER' 'DJ VECTOR REGISTERS'
  N,[0.5](('Q',®1 1p11),':',DJ)((('V',®QRSIZE 1p1QRSIZE),':',DJVR)
```

SETMEMORY function loads data V into the main memory at address A.

```
SETMEMORY V:IO
  a DESCRIPTION: PUT V INTO MAIN MEMORY AT ADDRESS A
  `IO=1 a ORIGIN
  a INPUT: A MAIN MEMORY ADDRESS
  a V DATA TO PUT INTO MAIN MEMORY
  V*,V a IGNORE SHAPE OF DATA
  ((pV)+(A-1)+MM)+V a MOVE DATA TO MEMORY
```
LOADV function loads vector register V from address in A with stride S.

\[
\text{LOADV} \ R; \ IO; \ V; \ A; \ S
\]

- **DESCRIPTION:** LOAD VECTOR REGISTER V FROM ADDRESS IN A WITH STRIDE S
- **IO=1** ≡ ORIGIN
- **INPUT:** \( R(3 \text{ ITEMS VECTOR}(V, A, S)) \)
- \( V \) IS VECTOR REGISTER
- \( A \) IS MEMORY ADDRESS
- \( S \) IS STRIDE

\((V \ A S) \rightarrow R \) \( \rightarrow \) DECOMPOSE \( R \) INTO \( V, A, \) AND \( S \)

\[
\text{LOAD VECTOR REGISTER V FROM ADDRESS IN A WITH STRIDE S}
\]

\[
\text{VR}(V; VCT) = \text{MM}[\text{MM}[A] + S \times 1 + VCT]
\]

- **UPDATE MAIN MEMORY WITH THE LENGTH OF LOADED DATA**
- \( \text{MM}[A] = \text{MM}[A] + VCT \)

STOREV function stores vector register V in address in A with stride S into main memory.

\[
\text{STOREV} \ R; \ IO; \ V; \ A; \ S; \ P
\]

- **DESCRIPTION:** STORE VECTOR REGISTER IN ADDRESS IN A WITH STRIDE S INTO MAIN MEMORY
- **IO=1** ≡ ORIGIN
- **INPUT:** \( R(3 \text{ ITEMS VECTOR}(V, A, S)) \)
- \( V \) IS VECTOR REGISTER
- \( A \) IS MEMORY ADDRESS
- \( S \) IS STRIDE

\((V \ A S) \rightarrow R \) \( \rightarrow \) STORE VECTOR REGISTER V IN ADDRESS IN A

- **WITH STRIDE S INTO MAIN MEMORY**
- \( \text{MM}[[\text{MM}[A] + S \times 1 + VCT)] = \text{VR}(V; VCT) \)
- **UPDATE MAIN MEMORY WITH THE LENGTH OF STORED DATA**
- \( \text{MM}[A] = \text{MM}[A] + VCT \)

LOADVCT function loads vector count from memory address A.

\[
\text{LOADVCT} \ A; \ IO
\]

- **DESCRIPTION:** LOAD VECTOR COUNT FROM MEMORY ADDRESS A
- **IO=1** ≡ ORIGIN
- **INPUT:** MEMORY ADDRESS A

\( \text{VCT} = \text{SS}[\text{MM}[A]] \) \( \equiv \) VECTOR COUNT IS MINIMUM

- BETWEEN SECTION SIZE (SS) AND VALUE IN MAIN MEMORY AT ADDRESS A
- \( \text{MM}[A] = \text{MM}[A] - \text{VCT} \) \( \equiv \) REDUCE STORAGE COUNT

LOADPVCT function loads partition vector count from memory address A.

\[
\text{LOADPVCT} \ A; \ IO
\]

- **DESCRIPTION:** LOAD PARTITION VECTOR COUNT FROM MEMORY ADDRESS A
- **IO=1** ≡ ORIGIN
- **INPUT:** MEMORY ADDRESS A

\( \text{PVCT} = \text{MM}[A] \)

- \( \text{MM}[A] = 0 \) \( \equiv \) REDUCE STORAGE COUNT TO 0
PSTOREV function stores vector register V that contains results of applying a function to partitions into main memory at address A with stride S.

\[ \text{PSTOREV } R; \text{IO}; V; A; S \]

**DESCRIPTION:** STORE VECTOR REGISTER THAT CONTAINS RESULTS OF APPLYING A FUNCTION TO PARTITIONS INTO MAIN MEMORY AT ADDRESS A WITH STRIDE S

**INPUT:**
- \( R \) (3 ITEMS VECTOR(V, A, S))
- \( V \) is vector register
- \( A \) is main memory address
- \( S \) is stride

\((V \ A \ S) -> R\)
- STORE VECTOR REGISTERS INTO MAIN MEMORY
- AT ADDRESS A WITH STRIDE S
- \( \text{MM}[\text{MM}[A] + S + \text{PVCT}] = \text{VR} + \text{PVCT} \)
- UPDATE MAIN MEMORY WITH THE LENGTH OF STORED DATA
- \( \text{MM}[A] = \text{MM}[A] + \text{PVCT} \)

DEFINETHGATE function defines the Haddamard gate as global variable H.

\[ \text{DEFINEHGATE } \]

**DESCRIPTION:** DEFINE HADAMARD GATE (H)

**INPUT:**
- ORIGIN

\[ R = \text{DEFINE 2 BY 2 ARRAY WITH ELEMENTS} \]
- \( 1 \ 1 \)
- \( 1 -1 \)
- \( H = \text{MULTIPLY WITH } 1 / \text{SQRT}(2) \)
- \( H = (1 + 2 \times 0.5) \times H \)

APPLY_H_GATE function simulates, up to a constant, the Haddamard gate action on states \(|0> \) or \(|1> \).

\[ \text{APPLY_H_GATE } A; \text{IO}; Q1; Q2; S; O1; O2; T01; T02 \]

**DESCRIPTION:** SIMULATES ACTION OF THE HADAMARD GATE (H) ON STATES \(|0> \) OR \(|1> \)

**NEGLECTING SQRT(1/2) NORMALIZATION CONSTANT**

**WHEN H ACTS ON \(|0> \), THE FUNCTION RETURNS \(|0)+|1> \) STATE**

**LALELD AS \('0 '+'+1' \)**

**WHEN H ACTS ON \(|1> \), THE FUNCTION RETURNS \(|0>-|1> \) STATE**

**LALELD AS \('0 '-1' \)**

\[ R = \text{APPLY_H_GATE } A; \text{IO}; Q1; Q2; S; O1; O2; T01; T02 \]

**INPUT:**
- ORIGIN

\[ Q1 = 2 \times 1 \]
\[ Q2 = 2 \times 0 \]
\[ S = (A = 1) \times Q1 \times Q2 \]

**IF A = 1 ASSIGN \(|1> \) STATE TO S**

**OTHERWISE IF A = 0 ASSIGN \(|0> \) STATE TO S**

**H ACTS ON S**

\( Q1 = (1 + 2 \times 0.5) \times (Q1 \times Q2) \)

\( T01 = \text{IF } '0' \ '1' \text{ THIS WILL BE RETURNED} \)

\( Q2 = (1 + 2 \times 0.5) \times (Q1 - Q2) \)

\( T02 = \text{IF } '0' \ '1' \text{ THIS WILL BE RETURNED} \)

\( R = Q1 \times Q2 \times (T02)(T01) \)

\( \text{IF } 0 = O1 \text{ RETURN } T01 \text{ OTHERWISE RETURN } T02 \)
HNGATE function simulates, up to a normalization constant, the Hadamard transform on the query register.

```
R = HNGATE V: |0|0: |0: |C: |B: |S: |F: |I
# DESCRIPTION: SIMULATES UP TO NORMALIZATION CONSTANT
# HADAMARD TRANSFORM ON THE QUERY REGISTER
# |0|0: ORIGIN
# |I: INPUT: QUERY REGISTER STATE(V)
# FOR EXAMPLE IF THE QUERY REGISTER DESCRIBES
# THE STATE OF 3 QUBITS ALL PREPARED IN THE |0> STATE,
# V IS 0 0 0

DEF HNGATE = CREATE HADAMARD ARRAY H
T = APPLY_H_GATE^V & APPLY H ON EACH COMPONENT OF V AND
STORE RESULT IN T
->(0=pt)+L0 & CHECK IF T HAS ONLY ONE COMPONENT WHICH
INDICATES THAT V HAS ONLY ONE STATE(0 OR 1); IF NOT
JUMP TO LABEL L0
R=T & STORE T IN FINAL RESULT
->LX & RETURN R
L0: & V HAS MORE THAN 1 STATE WHICH MEANS T HAS MORE THAN
ONE COMPONENT
F9=T[1]CALC_SIGN T[2] & CALCULATE SIGNS FOR THE TERMS IN TENSOR
# PRODUCT OF STATES FOR THE FIRST 2 COMPONENTS OF T
# + SIGN IS REPRESENTED BY 1
# - SIGN IS REPRESENTED BY -1
R=T[1]COMPPOSE_STATE T[2] & NEGLCETING NORMALIZATION CONSTANT
# DO TENSOR PRODUCT OF STATES FOR THE FIRST 2 COMPONENTS OF T
# NOTE: COMPPOSE_STATE USES FS VARIABLE
->(2=pt)+LX & CHECK IF V HAS ONLY 2 STATES; IF SO
RETURN R
I=2 & INITIALIZE LOOP VARIABLE
# V HAS MORE THAN 2 STATES
# LOOP THROUGH THE REST OF T COMPONENTS
L1:->((pt)+I+1)+LX & START WITH I=3; RETURN FINAL R WHEN
# THE REST OF T COMPONENTS ARE PROCESSED
F9=R CALC_SIGN T[1] & CALCULATE SIGNS FOR THE TERMS IN TENSOR
# PRODUCT BETWEEN R AND I-TH COMPONENT OF T
R=R COMPPOSE_STATE T[1] & NEGLCETING NORMALIZATION CONSTANT
# DO TENSOR PRODUCT BETWEEN R AND I-TH COMPONENT OF T
->L1 & GO BACK TO THE START OF THE LOOP
LX: & EXIT
```

For example, below is displayed the output of the HNGATE function when the query register describes the state of 3 qubits all prepared in the |0> state.

```
HNGATE 0 0 0
000 +001 +010 +011 +100 +101 +110 +111
```
CALC_SIGN function calculates +,- signs for terms in the tensor product of 2 quantum states. 
\[ R \leftarrow A \text{ CALC}_\text{SIGN} B; \{ 0 \}; C; BP; BM; S; L \]
- DESCRIPTION: CALCULATE SIGNS FOR THE TERMS IN TENSOR PRODUCT OF STATES A AND B
- THIS FUNCTION IS CALLED FROM HNGATE FUNCTION
- INPUT: STATES A AND B

\[ L \leftarrow (1+0=p \rightarrow A) \circ (\rightarrow A)(A) \times IF \ A \]
- IF ONE COMPONENT OF THE APPLY_H_GATE FUNCTION OUTPUT,
- ASSIGN +A TO L SINCE DEPTH OF A IS 4, OTHERWISE IF A IS ONE COMPONENT OF R
- WHICH IS BUILT IN HNGATE FUNCTION, ASSIGN A TO L SINCE DEPTH OF A IS 2
- DEPTH OF L IN BOTH CASES IS 2

\[ C \leftarrow L; \ldots; (\rightarrow B) \times \text{CATENATE EVERY ELEMENT OF L TO EVERY ELEMENT OF} \]
- \( \rightarrow B \), SINCE DEPTH OF B IS 4; DEPTH OF \( \rightarrow B \) IS 2;
- L \leftarrow L; (\rightarrow B) IS AN ARRAY SO WE ROLL IT INTO A VECTOR C
- BP \leftarrow \text{C} \times +', \times IF C HAS + SIGN (VALUE OF BP IS 1);
- WHERE C DOES NOT HAVE + SIGN, VALUE OF BP IS 0

BM \leftarrow \text{C} \times -', \times IF C HAS - SIGN (VALUE OF BM IS 1);
- WHERE C DOES NOT HAVE - SIGN, VALUE OF BM IS 0

S \leftarrow BP \times +1 \circ BM \times MULTIPLY BM WITH -1 SINCE BM SHOWS WHERE C HAS - SIGN AND
- ADD TO BP: S STILL HAS ZEROS WHERE C IS NOT + OR -
- S \leftarrow S \times 0 \times IDENTIFY WHERE S HAS ZEROS (VALUE OF S \times 0 IS 1 IN THAT CASE;
- OTHERWISE IS 0) AND ADD S \times 0 TO S;
- THE FINAL S HAS ONLY 1 OR -1
- DEPTH OF S IS 2

R \leftarrow S \times \times MULIPLY ELEMENTS IN EACH PART OF S AND ASSIGN
- PRODUCT FOR EACH PART TO R: WHERE THE FINAL SIGN IS +, R HAS VALUE OF 1
- WHERE THE FINAL SIGN IS -, R HAS VALUE OF -1

COMPOSE_STATE function calculates the tensor product of 2 quantum states. 
\[ R \leftarrow A \text{ COMPOSE_STATE} B; \{ 0 \}; C; PM; V; M; PV; FV; I; L \]
- DESCRIPTION:NEGLECTING NORMALIZATION CONSTANT
- CALCULATE TENSOR PRODUCT OF 2 STATES
- THIS FUNCTION IS CALLED FROM THE HNGATE FUNCTION
- INPUT: STATES A AND B

\[ L \leftarrow (1+0=p \rightarrow A) \circ (\rightarrow A)(A) \times IF \ A \]
- IF ONE COMPONENT OF THE APPLY_H_GATE FUNCTION OUTPUT,
- ASSIGN +A TO L SINCE DEPTH OF A IS 4, OTHERWISE IF A IS ONE COMPONENT OF R
- WHICH IS BUILT IN HNGATE FUNCTION, ASSIGN A TO L SINCE DEPTH OF A IS 2
- DEPTH OF L IN BOTH CASES IS 2

\[ C \leftarrow L; \ldots; (\rightarrow B) \times \text{CATENATE EVERY ELEMENT OF L TO EVERY ELEMENT OF} \]
- \( \rightarrow B \), SINCE DEPTH OF B IS 4; DEPTH OF \( \rightarrow B \) IS 2;
- L \leftarrow L; (\rightarrow B) IS AN ARRAY SO WE ROLL IT INTO A VECTOR C
- PM \leftarrow (C \times +') \circ (C \times -') \times IDENTIFY WHERE VALUE OF C IS +
- OR - (VALUE OF PM IS 1); OTHERWISE VALUE OF PM IS 0

V \leftarrow (-PM) \times (C \times -) \times SELECT ALL OTHER VALUES OF C (NOT EQUAL TO + OR -)

M \leftarrow (pcC[1]) \times 0 \times CREATE VECTOR M THAT HAS ALL COMPONENTS EQUAL TO 0
- M AND C[1] HAVE THE SAME NUMBER OF ELEMENTS

M[1] \leftarrow 1 \times SET THE FIRST ELEMENT EQUAL TO 1

PV \leftarrow ((C \times -) \times pcC[1]) \times PM \times CREATE A PARTITION VECTOR;
- PV IS EQUAL TO 1 AT THE FIRST ELEMENT OF EVERY COMPONENT OF C
- OTHERWISE IS 0

PV \leftarrow (+PV) \times V \times PARTITION V USING PV

R \leftarrow PV[1] \times ASSIGN THE FIRST COMPONENT OF PV TO R

I \leftarrow 1 \times INITIALIZE LOOP COUNTER

L0: \rightarrow (PV[FV] \times C \times I = 0 \times L0) \times START WITH I = 2 AND LOOP THROUGH PV COMPONENTS

R \leftarrow R; C \leftarrow (((C \times FS[I] = 0) \times (+) \times (-')) \times PV[I] \times USING GLOBAL FS DETERMINE
- WHICH SIGN TO USE (IF FS[I] = -1 TAKE -, OTHERWISE TAKE +) AND
- CATEenate THE SIGN AND PV[I] TO R

\rightarrow L0 \times GO BACK TO THE START OF THE LOOP

LX: \times EXIT
LOAD_QUERY_REGISTER function loads the query register with a vector.

LOAD_QUERY_REGISTER A;IO

DESCRIPTION: LOADS QUERY REGISTER WITH VECTOR A

IO=1 ORIGIN

INPUT: VECTOR A

LOAD_QUERY_REGISTER WITH VECTOR A

DJ[:(pA)]=A

LOAD_PARTITIONS_INTO_DJVR function loads the Deutsch-Jozsa vector registers with partitions.

LOAD_PARTITIONS_INTO_DJVR L;IO:I

DESCRIPTION: LOADS Deutsch-Jozsa VECTOR REGISTERS WITH PARTITIONS

IO=1 ORIGIN

INPUT: NESTED VECTOR P OF PARTITIONS

VECTOR L THAT HAS LENGTH OF EACH PARTITION

I=0 SET LOOP COUNTER TO 0

L0:->((P[I]*I+1)+L1 LOOP BY PARTITIONS; WHEN ALL PARTITIONS ARE

PROCESSED EXIT AT L1

DJVR[I;1+L[I]]=P[I] STORE I-TH PARTITION TO I-TH DJ VECTOR REGISTER

STARTING AT POSITION 2

DJVR[I;1]=L[I] STORE LENGTH OF I-TH PARTITION TO I-TH DJ VECTOR REGISTER

AT POSITION 1

L0: GO TO L0 TO LOAD NEXT PARTITION

L1: EXIT

CALC_PARTITIONS function partitions a data vector using given partition vector.

CALC_PARTITIONS V;IO:L;A:B:S;P:LN

DESCRIPTION: PARTITION DATA VECTOR USING GIVEN PARTITION VECTOR

IO=1 ORIGIN

INPUT: NESTED VECTOR V WHICH HAS 4 COMPONENTS:

1. DATA VECTOR LENGTH
2. DATA VECTOR
3. PARTITION VECTOR
4. STRIDE

(L A B S)V PARSE VECTOR V COMPONENTS INTO 4 VARIABLES:

L IS DATA VECTOR LENGTH MAIN MEMORY ADDRESS
A IS DATA VECTOR MAIN MEMORY ADDRESS
B IS PARTITION VECTOR MAIN MEMORY ADDRESS
S IS STRIDE

P=(+\MM[MM[B]+S\times-1+\MM[L]]+\MM[MM[A]+S\times-1+\MM[L]]) PARTITION DATA VECTOR

AT A WITH LENGTH AT L AND STRIDE S USING PARTITION VECTOR AT S WITH

LENGTH AT L AND STRIDE S

LN=P LN STORE LENGTH OF EACH PARTITION IN LN

N=P LN RETURN PARTITIONS AND RESPECTIVE LENGTHS
COUNT_PARTITIONS function calculates how many partitions there are in a partition vector.

```
COUNT_PARTITIONS V:IO:A:S:T
 definitions: calculates how many partitions there are in a partition vector
IO=1 origin
input: nested vector V with 2 components
  1. main memory address which stores partition vector main memory address
  2. stride
global: PVCT (number of partitions)
this function is called from calc_with_partitions function
(A S)=V parse vector V into 2 variables A and S:
  A is main memory address which stores partition vector main memory address
  S is stride
PVCT=\(\frac{\left(\text{MM}[A]-PVCT\right)}{S}+1+PVCT\) count
partitions in the partition vector at main memory
address (MM[A]-PVCT) with stride S
note: we have to subtract PVCT from MM[A] since
the partition vector was calculated in a previous
step(function calc_with_partitions) so the content of the main memory at A, MM[A], was
changed to MM[A]+PVCT
```

LOADLPVCT function loads a partition vector count from an address in the main memory while looping through partitions and loading them in the Deutsch-Jozsa vector registers.

```
LOADLPVCT A:IO
 description: load partition vector count from memory address A
while looping through partitions and loading them in deutsch-jozsa vector
registers
IO=1 origin
input: memory address A
load partition vector count from memory address A
PVCT=\(\text{MM}[A]\) count is min between
partition section size (PSS) and value in main memory at address A
MM[A]-MM[A]-PVCT reduce storage count
```

PREPARE_T_PARTITIONS function returns a specified number of partitions and respective partition lengths.

```
PREPARE_T_PARTITIONS V:IO:LP:P:LN:T:B
 description: returns the most PSS partitions
with respective partition lengths
this function is called from calc_with_partitions function
global: PSS (partition section size)
IO=1 origin
input: nested vector V with 3 components:
  1. partitions loop counter (we are looping through partitions in
     calc_with_partitions function)
  2. partitions
  3. respective partition lengths
(LP P LN)=V parse out components of V into 3 variables:
  LP is partitions loop counter
  P contains partitions
  LN contains respective partition lengths
T=(LP*PSS)+PSS define PSS indices
B=T\leq P only select indices less then or equal to the
total number of partitions
R=LP[B:T]LN[B:T] return partitions and respective partition lengths for defined
indices
```
GET_PARTITION function calculates a partition vector.

```
GET_PARTITION V;IO;L
\* DESCRIPTION: CALCULATES PARTITION VECTOR
\* IO=1 \* ORIGIN
\* INPUT: GROUP BY (BINS) VECTOR V

\* LOAD MAIN MEMORY
26 SETMEMORY V \* LOAD V IN MAIN MEMORY AT ADDRESS 26
51 SETMEMORY \(-1\) V \* LOAD ROTATED V IN MAIN MEMORY AT ADDRESS 51
\* ROTATED V HAS THE LAST ELEMENT OF V AT THE FIRST POSITION
L+\(p\) V \* FIND LENGTH OF V
2 SETMEMORY L \* LOAD L IN MAIN MEMORY AT ADDRESS 2
3 SETMEMORY 26 L \* MAIN MEMORY ADDRESS OF THE VECTOR V
4 SETMEMORY 51 \(-1\) V \* MAIN MEMORY ADDRESS
5 SETMEMORY L \* PARTITION COUNT ADDRESS IN MAIN MEMORY
6 SETMEMORY 76 \* PARTITION VECTOR ADDRESS IN MAIN MEMORY
LOADPVC T 5 \* LOAD PARTITION VECTOR COUNT
LOOP:LOADVCT 2 \* LOAD VECTOR COUNT
 \(-0=VCT\)+LX \* EXIT IF VECTOR COUNT IS ZERO
LOADV 1 3 1 \* LOAD V1 VECTOR REGISTER WITH V
LOADV 2 4 1 \* LOAD V2 VECTOR REGISTER WITH \(-1\) V
1(F3 DJBOX)0 \* USE DJBOX OPERATOR TO CALCULATE
\* PARTITION VECTOR FOR V
STOREV 3 6 1 \* STORE PARTITION VECTOR IN MAIN MEMORY
\(-0=\) LOOP \* LOOP BY VECTOR COUNT
LX \* EXIT THE LOOP
\* UPDATE PARTITION VECTOR COUNT
COUNT_PARTITIONS 6 1
\* LOAD PARTITION VECTOR IN MAIN MEMORY
5 SETMEMORY PVC T
SHOWMACHINE \* DISPLAY MACHINE'S CONTENTS
```
CALC_WITH_PARTITIONS function applies a specified function to each partition.

CALC_WITH_PARTITIONS AR: 10; S: P; LN; LP; TP; TLN; L; V; D
a DESCRIPTION: APPLIES A SPECIFIED FUNCTION TO EACH PARTITION
  10 = ORIGIN
a INPUT: 2 COLUMNS ARRAY AR(COLUMN1=GROUP BY DATA (BINS); COLUMN2= DATA TO PARTITION)
a
a PARSE OUT AR INTO 2 VARIABLES:
  V = AR[;1] = GROUP BY (BINS) VECTOR
  D = AR[;2] = DATA TO PARTITION
  26 SET MEMORY V = LOAD V IN MAIN MEMORY AT ADDRESS 26
  51 SET MEMORY "14V" = LOAD ROTATED V IN MAIN MEMORY AT ADDRESS 51
a
a Rotated V has the last element of V at the first position
L = RV V = FIND THE LENGTH OF V
  2 SET MEMORY L = LOAD L IN MAIN MEMORY AT ADDRESS 2
  3 SET MEMORY 26 = V IN MAIN MEMORY ADDRESS
  4 SET MEMORY 51 = "14V" IN MAIN MEMORY ADDRESS
  5 SET MEMORY 76 = PARTITION COUNT ADDRESS IN MAIN MEMORY
  6 SET MEMORY 76 = PARTITION VECTOR ADDRESS IN MAIN MEMORY
LOADPVCT 5 = LOAD PARTITION VECTOR COUNT
LOOP: LOADPVCT 2 = LOAD VECTOR COUNT
  - (0=PVC) = EXIT IF VECTOR COUNT IS ZERO
  LOADV 1 3 1 = LOAD V1 VECTOR REGISTER WITH V
  LOADV 2 4 1 = LOAD V2 VECTOR REGISTER WITH "14V
  1 (F3 DJBOX) = USE DJBOX OPERATOR TO CALCULATE PARTITION VECTOR FOR V
  STOREV 3 6 1 = STORE PARTITION VECTOR IN MAIN MEMORY
  -> LOOP = LOOP BY VECTOR COUNT
LX: = EXIT THE LOOP

a
a UPDATE PARTITION VECTOR COUNT
COUNT_PARTITIONS 6 1
a
a LOAD PARTITION VECTOR IN MAIN MEMORY
  5 SET MEMORY PVC
  LOAD D(DATA TO PARTITION) IN MAIN MEMORY
  26 SET MEMORY D
  2 SET MEMORY L = LOAD L IN MAIN MEMORY
  3 SET MEMORY 26 = D IN MAIN MEMORY ADDRESS
  4 SET MEMORY 76 = PARTITION VECTOR ADDRESS IN MAIN MEMORY
  7 SET MEMORY 51 = RESULT ADDRESS IN MAIN MEMORY OF APPLYING A SPECIFIED FUNCTION TO
  EACH PARTITION
a
a PARTITION VECTOR DATA USING GIVEN PARTITION VECTOR
(P LN) = CALC_PARTITIONS 2 3 4 1
LP=0 = LOOP COUNTER
LOOP2: LOADLPVCT 5 = LOAD PARTITION VECTOR COUNT
  - (0=PVC) = EXIT IF PARTITION VECTOR COUNT IS ZERO
a
a RETURN THE MOST POSS (PARTITION SECTION SIZE) PARTITIONS IN TP
a
a WITH RESPECTIVE PARTITION LENGTHS IN TLN
  (TP TLN) = PREPARE_T_PARTITIONS(LP P LN)
  LOAD DeutsCh-JOSSA VECTOR REGISTERS
  TP LOAD_PARTITIONS INTO DJVOR TLN
a
a APPLY FUNCTION F2 TO EACH PARTITION USING DJBOX OPERATOR
3 (F2 DJBOX) =
  PSTOREV 3 7 1 = STORE RESULT OF APPLYING F2 TO PARTITIONS IN MAIN MEMORY
LP=LP+1 = INCREMENT LOOP COUNTER
  LOOP = LOOP BY PARTITION VECTOR COUNT
LX2: = EXIT THE LOOP
SHOWMACHINE = DISPLAY MACHINE'S CONTENTS
DJBOX operator simulates essential elements of the quantum circuit for the Deutsch-Jozsa algorithm.

```
A(F DJBO X)B; [IO: STATES=P; XY; TT; CONV; Z; TT; K; I; P; CONV

DESCRIPTION: SIMULATE ESSENTIAL ELEMENTS OF THE QUANTUM CIRCUIT FOR
THE Deutsch-Jozsa ALGORITHM

I=0; ORIGIN

INPUT: A IS VECTOR REGISTER
B IS VECTOR REGISTER OR Deutsch-Jozsa VECTOR REGISTER
F IS FUNCTION USED TO CALCULATE PARTITION VECTOR

AND APPLY TO EACH PARTITION
STATES TO HNGATE QR=0; PREPARE ALL THE QUERY REGISTER QUITS IN THE |0> STATE
AND APPLY HADAMARD TRANSFORM ON THE QUERY REGISTER

THE QUERY REGISTER WILL BE A SUPERPOSITION OF ALL VALUES
P=(STATES=’+’); (STATES=’-’) IDENTIFY WHERE IS + OR -
X=(QRSIZE QR)P(-P)/STATES CREATE AN ARRAY OF ALL SUPERPOSITION VALUES IN

THE QUERY REGISTER

Y=HNGATE 1; APPLY HADAMARD GATE ON THE ANSWER REGISTER
P3CONV=(0 QRSIZE)0; INITIALIZE CONTROL VARIABLE THAT WILL BE USED TO
Determine IF FUNCTION F IS BALANCED OR CONSTANT
R1=A; SET GLOBAL VARIABLE FOR THE VECTOR REGISTER A
R0=B; SET GLOBAL VARIABLE FOR THE REGISTER B
I=0; INITIALIZE OUTER LOOP VARIABLE
L0:->(QRSIZE<1+1)+LX; LOOP BY THE QUERY REGISTER STATES
LOAD_QUERY_REGISTER X[I;] LOAD THE QUERY REGISTER WITH STATE X[I;]

NOTE: X IS AN ARRAY OF ALL SUPERPOSITION VALUES IN THE QUERY REGISTER
T=HNGATE X[I;] APPLY HADAMARD GATE ON THE STATE X[I;]
P=(-P)(T+’+’)(T-’-’) IDENTIFY WHERE IS NOT + OR -( FIND STATES)
TT=+T; ENCLOSE STATES
CONV-2; EXPRESS EACH STATE IN DECIMAL REPRESENTATION (STATES ARE IN BINARY
REPRESENTATION)
Z=(~P)/T; COLLECT ALL +,-
K=0; INITIALIZE INNER LOOP VARIABLE
TTT=’’; INITIALIZE VARIABLE WHICH HOLDS RESULT OF LOOP ITERATION
L1:->(QRSIZE-1<1+1)+L2; LOOP QRSIZE-1 TIMES, BEGIN LOOP WITH THE SECOND STATE
TT=TT; T TTT; (1+Z[K]=’+’); (~1+CONV(1+K))(CONV(1+K)); IF + SIGN ADD STATE,
-IF - SIGN, ADD STATE MULTIPLIED BY -1
-L1; GO TO NEXT LOOP ITERATION
L2; EXIT LOOP
P3CONV=P3CONV,[1](Y YPLUS(F X[I;]))(0,TTT) ADD 0 STATE TO TTT,
MULTIPLY STATES WITH THE OUTPUT OF Y ADD MODULO 2 F(X) AND
ADD THEM TO P3CONV
-L0; GO TO NEXT STATE IN THE QUERY REGISTER
LX; EXIT OUTER LOOP
T=+/P3CONV; SUM VALUES IN EACH COLUMN OF P3CONV
L3; CHECK IF T HAS ALL ZERO’S
IF NOT JUMP TO LABEL L3
‘QUERY REGISTER IN 0 STATE’
LOAD_QUERY_REGISTER 0 0 0 LOAD QUERY REGISTER WITH ZERO STATES
-L3; EXIT
L3; EXIT OUTER LOOP
T=+/HNGATE 0 0 0; IF T HAS ALL ZERO’S
IDENTIFY WHICH STATES ARE IN SUPERPOSITION IN THE QUERY REGISTER
AND EXPRESS EACH OF THESE STATES IN BINARY REPRESENTATION
’QUERY REGISTER EVALUATED IN SUPERPOSITION OF THESE STATES’,<CONVERT_TO_BINARY TT
LOAD_QUERY_REGISTER QR0; DISPLAY SUPERPOSITION WITH

YPLUS function adds modulo 2 a state and function f(x).

R=Y YPLUS F; [IO; P; V
DESCRIPTION:ADD MODULO 2 STATE Y AND FUNCTION F
I=0; ORIGIN

INPUT: STATE Y AND FUNCTION F
P=(-P)(Y=’+’)(Y=’-‘) IDENTIFY WHERE IS NOT + OR -( FIND STATES)
V=2; (F/P)/Y ADD F TO EVERY STATE OF Y, DIVIDE BY 2
AND DETERMINE REMINDER FOR EVERY STATE (ADD MODULO 2)
R=(V[1]=1)+1; RETURN 1 IF THE FIRST COMPONENT OF V IS 0,
RETURN -1 IF THE FIRST COMPONENT OF V IS 1

YPLUS function adds modulo 2 a state and function f(x).
```
F3 function compares two vector register cell contents and store the result in the third vector register cell.

```plaintext
R+F3 B:IO:A
DESCRIPTION: COMARE CONTENTS OF TWO VECTOR REGISTER CELLS
AND STORE COMPARISON RESULT (0 OR 1) IN THE THIRD VECTOR
REGISTER CELL
IO=0 ORIGIN

APPLICATION: B - QUERY REGISTER STATE IN BINARY REPRESENTATION
CONVERT BINARY INPUT B INTO ADDRESS A
A=2*B
->VR[R0;A]=.'.'+L0 IF CELL AT ADDRESS A FOR VECTOR REGISTER R0 IS EMPTY
JUMP TO L0, NOTHING TO COMPUTE
COMPARE R0 VECTOR REGISTER CELL AT THE ADDRESS A
WITH R1 VECTOR REGISTER CELL AT THE ADDRESS A
IF CONTENTS ARE THE SAME PICK 0
AND STORE 0 IN THE THIRD VECTOR REGISTER CELL
AT THE ADDRESS A
OTHERWISE STORE 1
VR[2;A]=(VR[R0;A]*VR[R1;A])>0 1
L0:
RETURN ALWAYS 1
R=1
```

F2 function sums elements of a vector stored in a Deutsch-Jozsa vector register and stores the sum in a vector register.

```plaintext
R+F2 B:IO:A:P:T
DESCRIPTION: SUM VECTOR ELEMENTS STORED IN
Deutsch-Jozsa VECTOR REGISTER 1+A AND STORE SUM
IN VECTOR REGISTER R1 AT ADDRESS 1+A
IO=1 ORIGIN

APPLICATION: B - QUERY REGISTER STATE IN BINARY REPRESENTATION
CONVERT BINARY INPUT B INTO ADDRESS A
A=2*B
->DJVR[1+A;R0]=.'.'+L0 IF CELL AT ADDRESS R0 FOR
Deutsch-Jozsa VECTOR REGISTER 1+A IS EMPTY JUMP TO L0, NOTHING TO COMPUTE
P=DJVR[1+A;R0] READ OFF HOW MANY ELEMENTS ARE IN VECTOR (CELL R0)
VR[R1;1+A]+/DJVR[1+A;1+P] SUM THOSE ELEMENTS AND STORE RESULT INTO
VECTOR REGISTER R1 AT ADDRESS 1+A
L0:
RETURN ALWAYS 1
R=1
```

CLEARMEMORY function clears main memory between two addresses.

```plaintext
R+S CLEARMEMORY E:IO
DESCRIPTION: CLEAR MAIN MEMORY BETWEEN ADDRESSES S AND E, INCLUDING S AND E
IO=1 ORIGIN

APPLICATION: MAIN MEMORY ADDRESSES S AND E
((1+E-S)+(S-1)+MM)=(1+E-S)P.'.' CLEAR MAIN MEMORY
```

CONVERT_TO_BINARY function creates the binary representation of a number in the decimal representation. It is basically “Convert integer to binary” APL2 idiom from the APL2 Idiom Library imbedded into a function.

```plaintext
R=CONVERT_TO_BINARY N:IO
DESCRIPTION: CREATE BINARY REPRESENTATION OF THE NUMBER
IN DECIMAL REPRESENTATION
IO=1 ORIGIN
APPLICATION: NUMBER N IN DECIMAL REPRESENTATION
R=((1+/[2 * 1[10]/N])p2)煊 CREATE BINARY REPRESENTATION FOR N
```