60009 Custom Computing 2020 Past Paper Solution

Note: this is not an official solution but I'm trying to make it as accurate as possible. If you find any typos or incorrect answers and explainations, please email me at xz1919@ic.ac.uk with the title <course_number>-<course_name>-<year>-past-paper-enquiries or directly message me through Whatsapp. If you need explaination or reviews on the provided answer, please contact me as well. :)

---

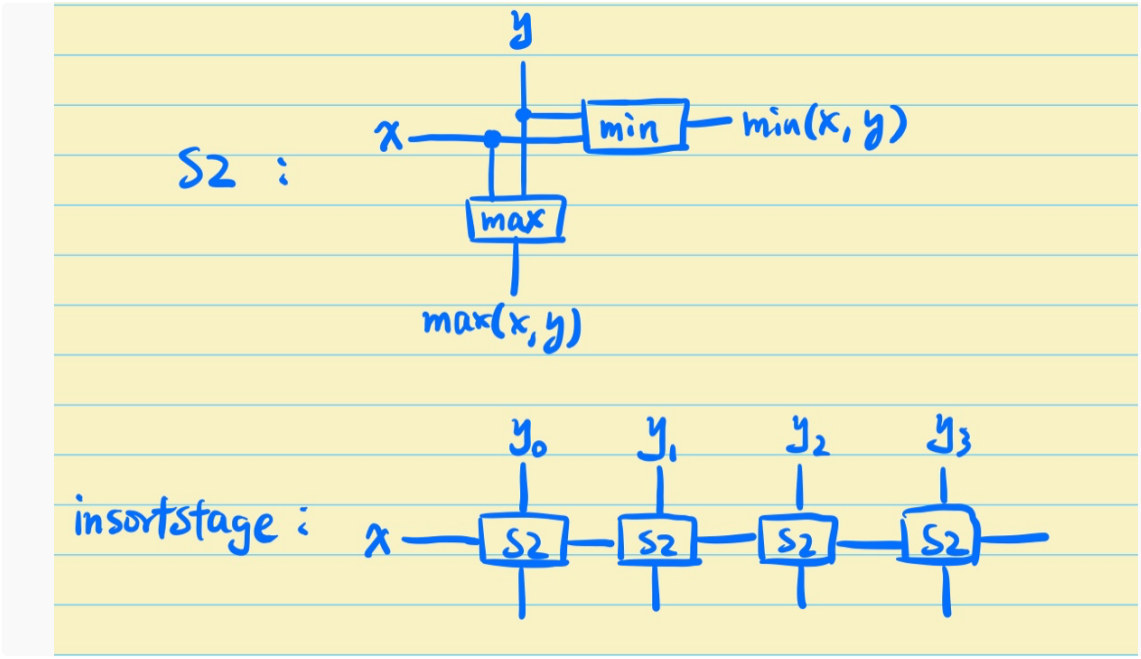
1. Below are the questions

   a. One stage of an insertion sorter is given by the following Ruby description:

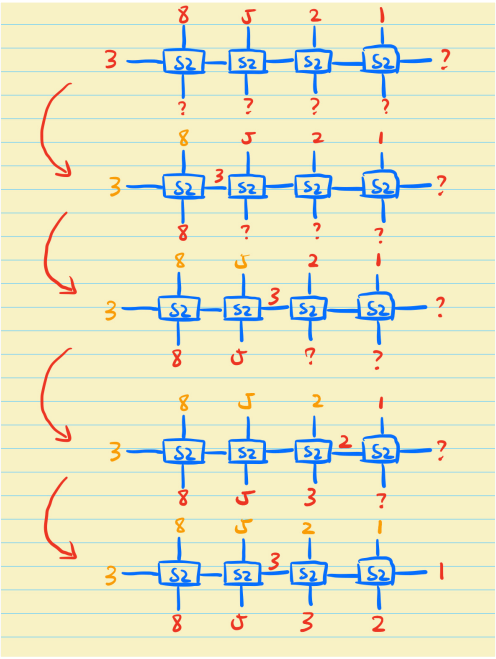
```
S2 = fork; [max, min]
insortstage = row n S2
```

   Provide a diagram for S2 and for insortstage when n = 4. Show on the diagram for insortstage how it can insert the number 3 into the list <8, 5, 2, 1>

   The diagram of S2 and insortstage are provided below



   The insertion process of 3 into <8, 5, 2, 1> is shown below. You can also just draw similar illustration on a single insortstage diagram. This is just to make sure the illustration is comprehensive, but it's not necessary. Notice that all of these steps happen in one cycle, so the numbers in the orange are just representing that the signal has been used.

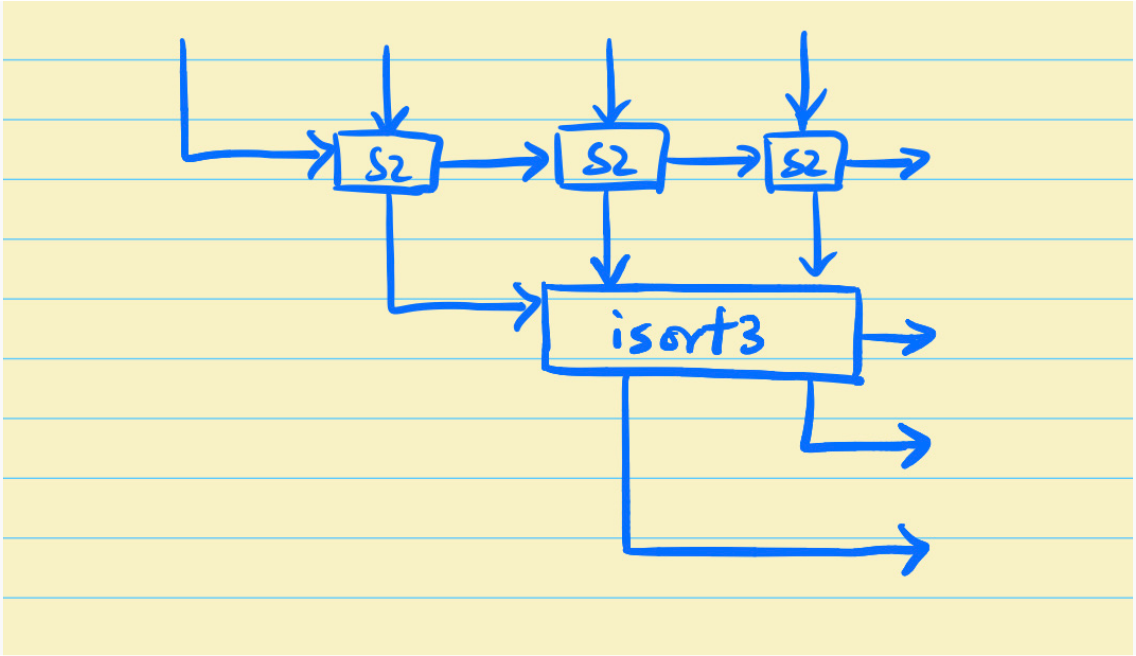


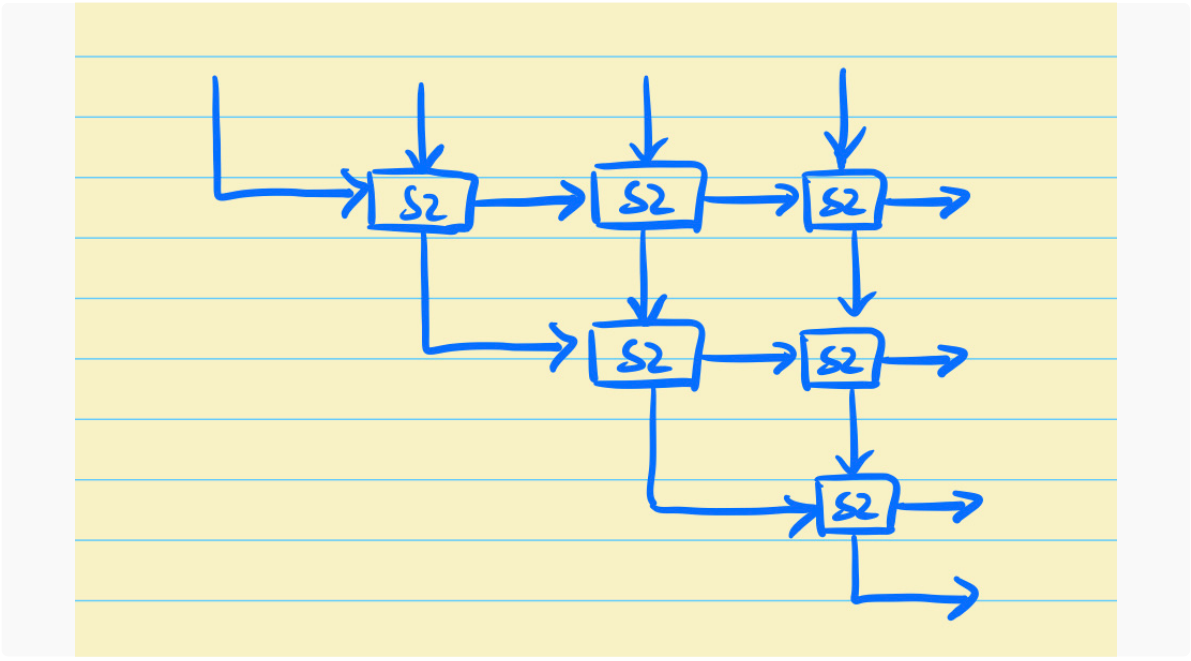
b. The inductive case in the definition of a parallel insertion sorter is given by:

```
isort (n+1) = (apl n)^~1; row n S2; fst (isort n); apr n
```

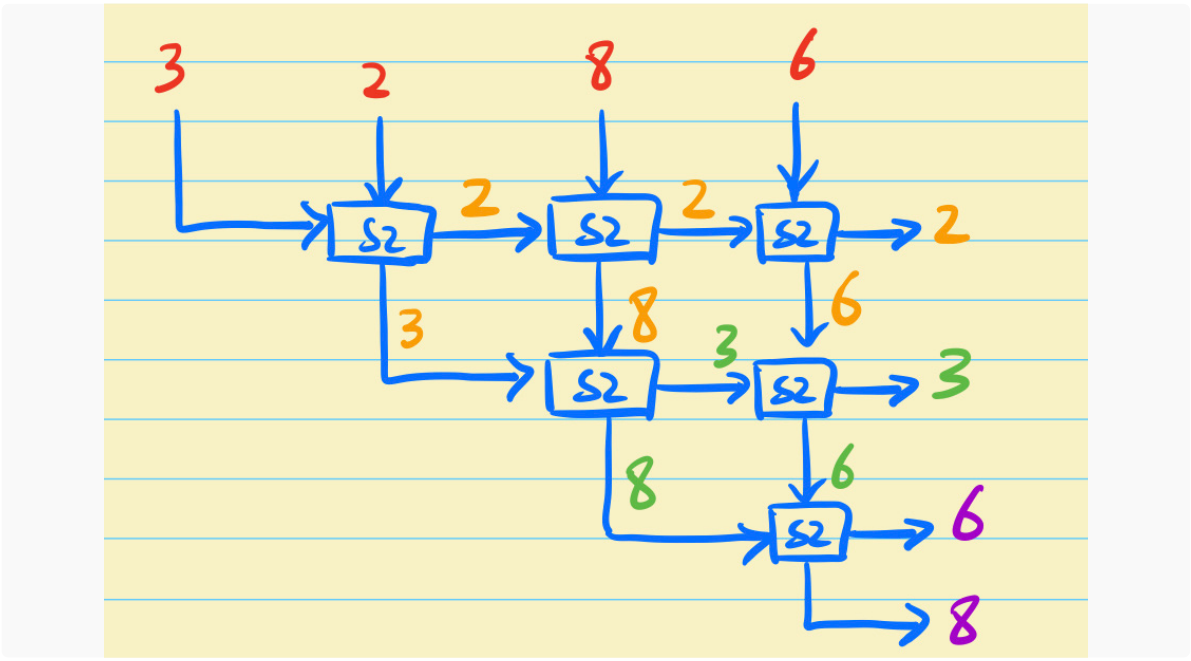
Provide a diagram for the right-hand side of the above definition when n = 3, showing the component isort_3. Provide another diagram showing how isort_4 is made up of multiple S2 components, and use this diagram to illustrate how isort_4 can be used to sort the list <3,2,8,6>. (It's not 3.2. This is a typo clearly)

Below is two diagram: the first showing isort_3 and the second one showing how isort_4 is made up of multiple S2



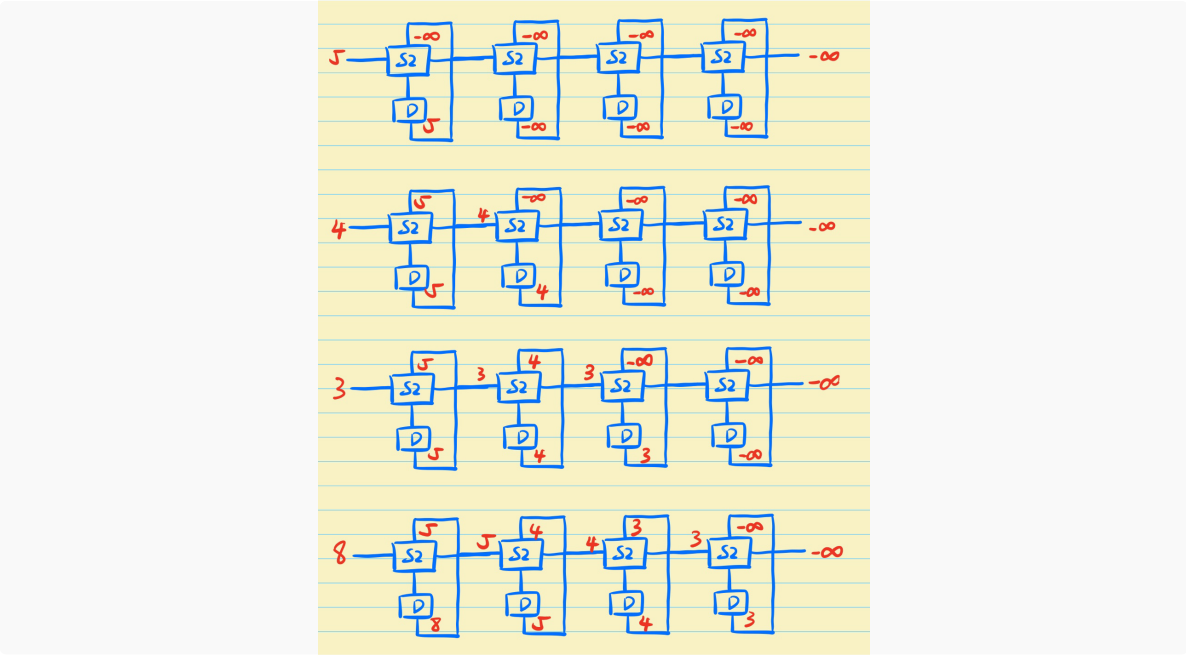


We can illustrate the list sorting using this diagram:

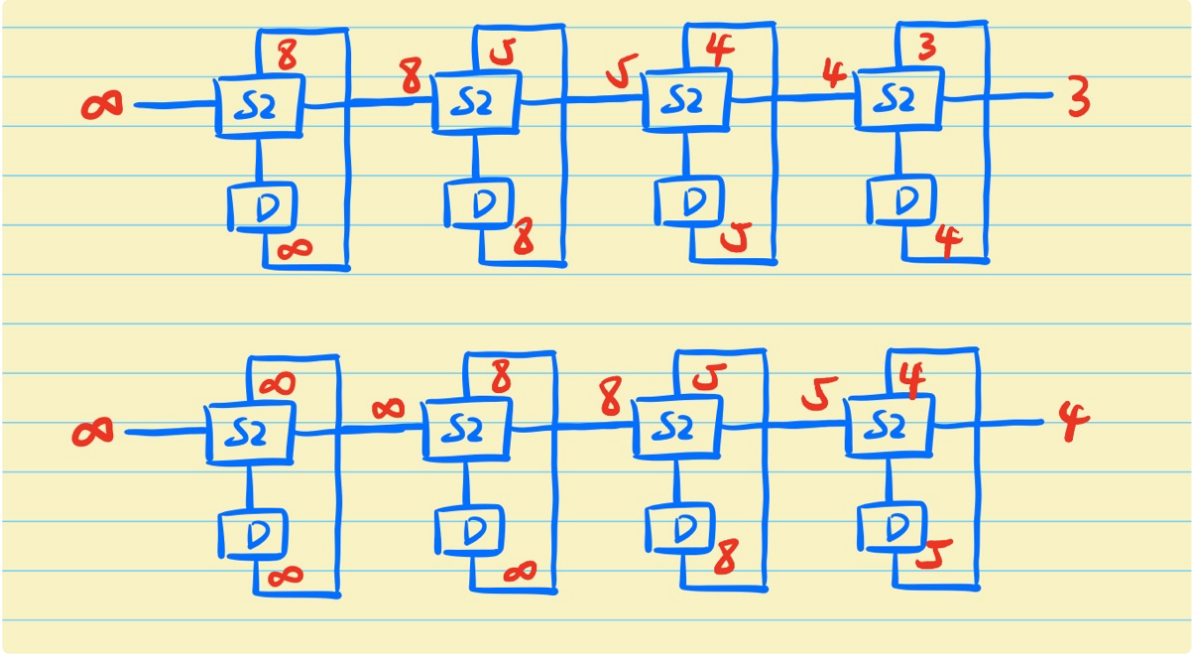


c. Describe how `insortstage` in Part a, together with registers initialised to minus infinity, can be used in building a state machine for implementing insertion sort when n = 4. (No Ruby description is needed.) Provide diagrams to show how this state machine can be used in sorting the time sequence 5, 4, 3. 8. What values should be streamed in, so that the sorted values can be streamed out?

We can use a loop and place a register within each loop as shown below



Positive infinity should be streamed in so that values can be pushed out like this:



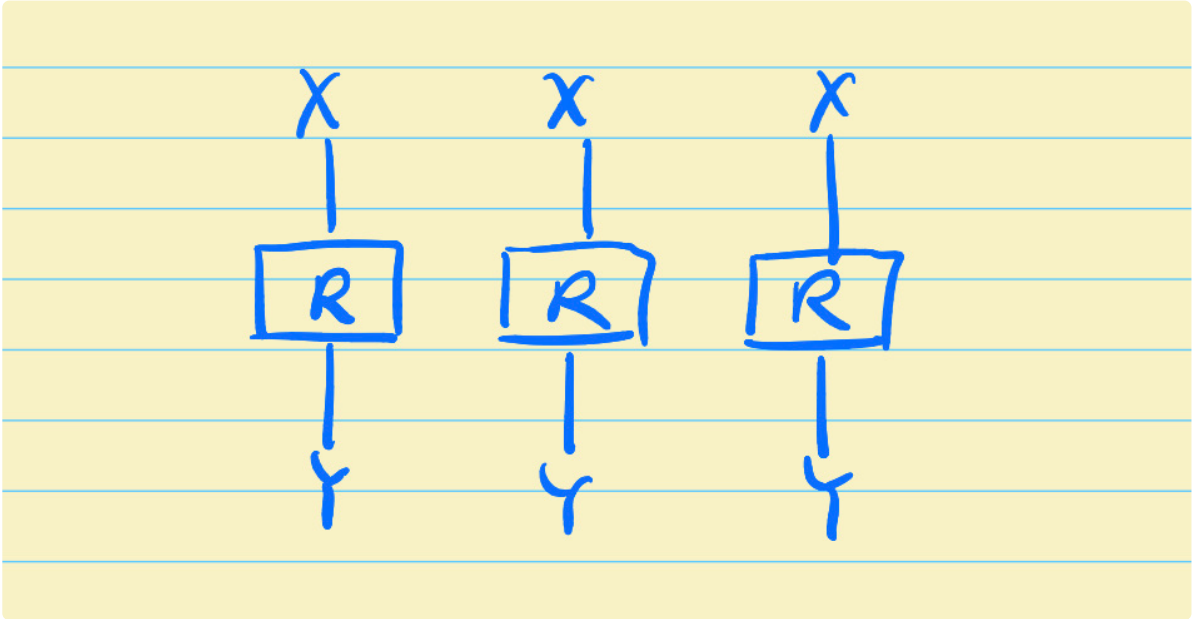
2. Below are the questions

   a. Given R: X ~ Y, what are the type and the recursive definition of the map combinator, map_n R? Provide a diagram of mapn R when n=3.

   The type of map is $\langle X \rangle_n \sim \langle Y \rangle_n$ and the recursive definition of it is

```
map 1 = R
map n R = [R, map (n-1 R)] \ apl (n-1)
```

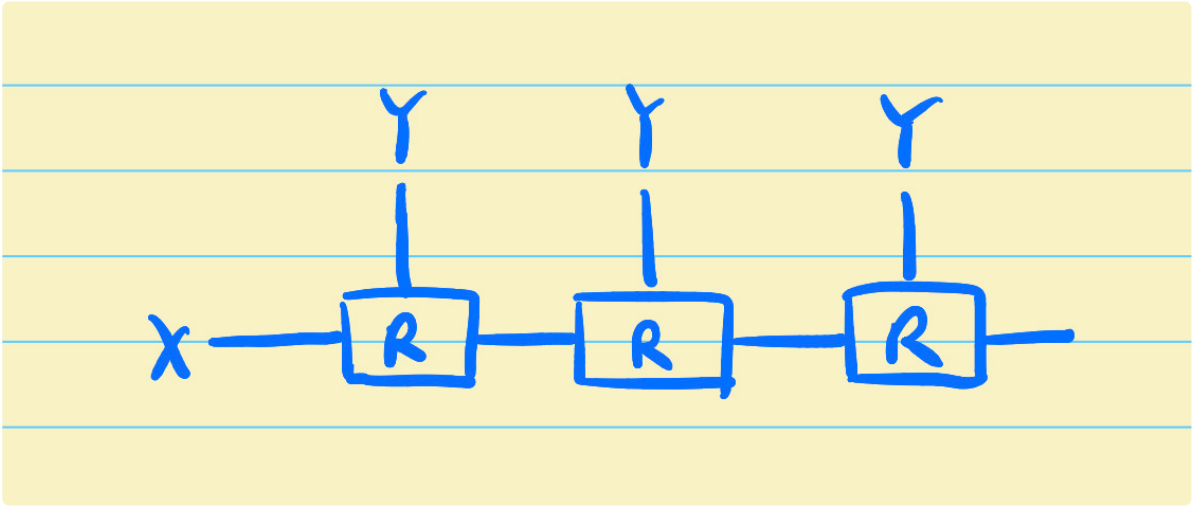
   A diagram of it would be



b. Given R: <X, Y> ~ X, what are the type and the recursive definition of the left reduction combinator, rdl_n R? Provide a diagram of rdl_n R when n=3.

The type of rdl_n R is <X, <Y>$_n$> ~ X and the recursive definition of it would be

```
rdl 1 R = R
rdl n R = snd (apl (n-1))^~1; rsh; fst R; rdl (n-1) R
```



c. Provide the type and the definition of a parallel adder maddn which takes a list of n numbers, each of type Num, and produces their sum. (again, another typo. It should be sum, not num) Your definition should make use of an 2-input adder, the append left wiring block, and the left reduction combinator rdl_n in Part b. Provide the type and the definition of a component inprodn which takes a list of two vectors, each represented by a list containing n numbers, and produces their inner product.

The type of the parallel adder is <Num>$_n$ ~ Num and the definition is

```
madd n = (apl (n-1))^~1; rdl n add
```

The type of the inprod_n is <<Num>$_n$, <Num>$_n$> ~ Num and its definition is

```
inprod n = zip n; map n mult; madd n
```

d. Given the component group$_{k\ n}$ which relates a list of kn elements to a list of k elements each of which is a list of n elements, provide an equation that expresses a left reduction as a left reduction of left reduction. Introduce another equation which can be used to parametrically pipeline madd$_n$.

The new definition of left reduction is

```
rdl k n R = snd (group k n); rdl k (rdl n R)
```

The equation that can be used to parametrically pipeline madd$_n$ is

```
snd (group k n); rdl k (rdl n R; D)
```

e. The first stage of a fully-connected element of a convolutional neural network, FC1m n, takes a list containing a vector V represented by a list of n numbers, and an nxm matrix M represented by a list of n elements, each containing a list of m numbers, and produces a list of two mxn matrices. The first mxn matrix contains m copies of the vector with n numbers, while the second mxn matrix contains a transpose version of the nxm matrix. Using the wiring blocks mfork and tran with suitable parameters, provide the type and the definition of FC1m n.

The second stage of this fully-connected element, FC2m n, takes the output of FC1m n and produces the matrix-vector product of M and V. Using the wiring block tran and the component inprod in Part c with suitable parameters, provide the type and the definition of FC2m n.

The type of the block FC1$_{mn}$ is <<Num>$_n$, <<Num>$_m$>$_n$> ~ <<<Num>$_n$>$_m$, <<Num>$_n$>$_m$> and its definition would be

```
FC1 m n = [mfork m, tran n m]
```

The type of the block F2$_{mn}$ is <<<Num>$_n$>$_m$, <<Num>$_n$>$_m$> ~ <Num>$_m$

```
FC2 m n = zip m; map m (inprod n); tran 1 n
```

3. Section B

a. A Newton-Raphson iteration to compute the cube root of a number a can be implemented using the following C code, where a is the input, and the output is the final value of variable x:

```
float a;

float x = 1.0;
float h = (1.0 - a) / (3.0);
while (fabs(h) >= EPS) {
  x = x - h;
  h = (x * x * x - a) / (3 * x * x);
}
```

Given that: the type of kernel input and output elements is available to you as floatType, the fabs function is defined for you already, and that EPS is defined for you as a compile-time constant, write a MaxJ kernel which, given a single stream input a of floating-point type (floatType) computes the cube root of each stream element as a single stream output y. Your design should convert the while-loop in the C code to a for-loop in the MaxJ code which runs for M iterations, where M is available as a compile-time parameter. Remember to use a flag variable to ensure code execution in the for-loop matches that in the while-loop. In your solution, omit the kernel constructor, class declaration and import declarations.

To convert the while-loop to a for-loop, we need to first rewrite the C code:

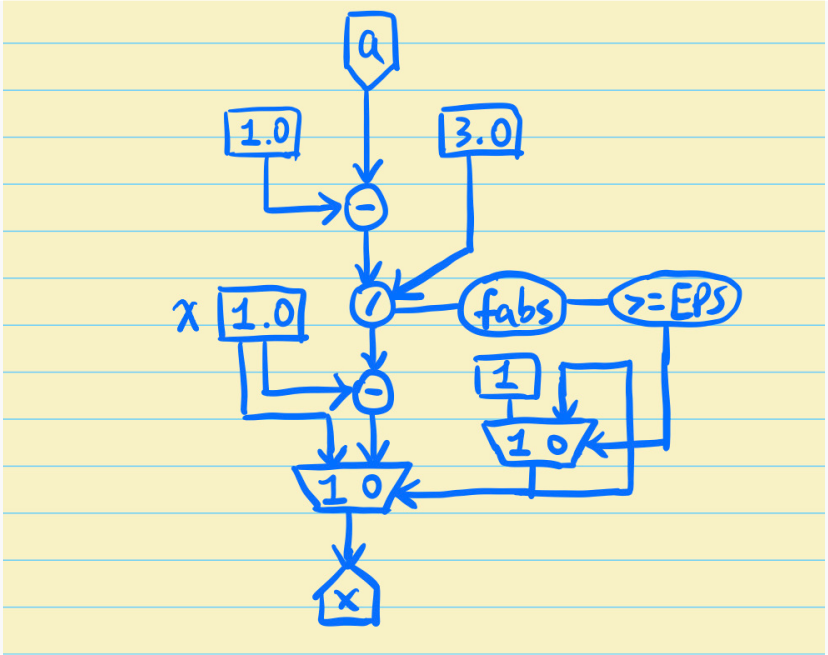
```
float a;

float x = 1.0;
float h = (1.0 - a) / 3.0;

bool finished = false;

for (int i = 0; i < M; i++) {
  bool condition = fabs(h) >= EPS;
  finished = condition ? true : finished;
  x = finished ? x : (x - h);
  h = (x * x * x - a) / (3 * x * x);
}
```

We then write a MaxJ kernel based on this code

```
DFEVar a = io.input("a", floatType);

DFEVar x = constant.var(1.0, floatType);
DFEVar h = (1.0 - a) / 3.0;
DFEVar finished = constant.var(dfeBool(), 0); //false

for (int i = 0; i < M; i++) {
  DFEVar condition = fabs(h) >= EPS;
  finished = condition ? constant.var(1) : finished;
  x = finished ? x : (x - h);
  h = (x * x * x - a) / (3 * x * x);
}

io.output("output", x, floatType);
```

b. Draw a diagram of the dataflow graph for the kernel from Part a for M = 1, before the graph has been scheduled.

The diagram is shown below. Notice that the part calculating `h = (x * x * x - a) / (3 * x * x)` can be ignored because M = 1 and thus internally in the MaxJ kernel h is not even wired up with x



c. For N stream inputs, give an expression for the total runtime in seconds. Assume that a is read over a bus with bandwidth B bytes/second, the stream element size is 4 bytes, and that the cycle time is Tc seconds.

From the given information in the question, the total time would just be $max(T_{compute}, T_{bus})$, which is $max(\frac{4N}{B}, C_{average} \cdot T_c)$, where C_{average} is the average cycle count for running the entire kernel.

We can, theoretically, do it in this way:

Assume that subtraction takes k cycles, multiplication takes m cycles and division takes n cycles to compute. We can approximate the proportion of the time spent at each location to calculate the average number of minus, add, multiply, division, etc.

d. Briefly explain why implementing a C while loop as a guarded for-loop as in Part a in hardware can lead to an inefficient design

Two reasons: 1️⃣ the maximum number of iterations might be significantly higher than the average number of iterations, which means that most of the chip area designed to run this kernel will do nothing. 2️⃣ the maximum number of iteration will take a lot of chip area, or even not fit within the chip.

Question 4 is non-examinable for the academic year 2021 - 2022.