Rainbow Assember (RASM)

DRAFT

Donal K. Fellows

October 3, 1997

1 Basic Formatting

One instruction per line. Leading and trailing spaces are chopped before parsing. Where the first character is a semicolon, `;`, the rest of the line will be taken as a comment (and inserted into the generated code). Where the first character is a colon, `:`, the rest of the line will be taken to be a label in the code. Labels should not contain a double colon, `::`, as this is reserved for inter-module references.

I will not allow lines to have a combination of things on them — it does not improve readability significantly and it makes parsing much harder.

1.1 Arguments

I use Tcl to split an instruction into opcode and operands, so you either need to use curly braces to quote spaces (and other significant characters) or backslash-sequences. Note that the special "backslash-newline-whitespace to single space" conversion is not supported, so long strings with no newline characters need to represented by long lines in the input.

If an operand starts (after the instruction line has been split up into "words") with a colon, `:`, the remainder of that operand represents the address of a label to be pushed onto the stack. Where an address reference (label name) contains a double colon, `::`, the part before the double colon refers to the name of a compilation module to look up the label in, and the part after the double colon refers to the name to look up there. If an operand starts with an apostrophe, `'`, the remainder represents a type to be pushed onto the stack. If an operand starts with a dollar, `$`, the remainder represents a string to be pushed onto the stack. All other operands are taken to be data values.

Prior to the appending of an opcode onto the current instruction stream, a push[1] is appended for each of the operands to that opcode, in a left-to-right order.

There is also a shorthand for getArg number, which is implemented using an operand starting with a commercial at symbol, `@`, with the rest of the operand being the data value to use as an index (presuably a positive integer!) Note that getArg number is itself in-turn a shorthand for a push followed by a getArg, so quite a lot of instructions can be generated by a single line of RASM code.

2 Instructions

2.1 Meta-Instructions

module moduleName
Starts a new compilation module named *moduleName*. The must be the first instruction, even before comments (since comment text is inserted into the compiled modules to aid debugging).

constant constantName constantValue
Makes the name *constantName* an alias for the data value *constantValue* in the current compilation module. Note that:

[1] The push instruction itself is handled specially by not appending an instruction after all the operands have been pushed.

1. this naming is not propagated between compilation modules. This doesn't matter for constants that aren't supposed to be exported, and ones that are can either be modelled using function calls (never a problem as those constants will never exist in a directly executed context) or passed across directly by the compiler stage (instead of the assembler/linker stages).
2. this aliasing is efficient — should an alias be created to a large data structure, use of that alias will always get a reference to a singular shared copy of the internal representation of that structure
3. this aliasing is not preserved during printout, as it is lost during the linking stage.

At some point, I might come up with a way to improve on this, though I don't currently see the necessity of doing so.

2.2 Implemented Instructions

All these instructions exist (or will do soon).

The symbol $\vdash$ denotes the base of the stack frame. Access outside the stack frame is completely prohibited (a restriction it would be nice to drop, but it makes debugging much easier).

2.2.1 Push Instruction Variants

Do not use these instructions, as they are used during the mapping of the push instruction, dependent on the type of the data being pushed. They are placed here merely for reference.

pushString string

$\boxed{- \Rightarrow string}$

Pushes a string onto the stack

pushAddress addr

$\boxed{- \Rightarrow addr}$

Pushes address $addr$ on the stack.

pushData value

$\boxed{- \Rightarrow value}$

Pushes the value $value$ onto the top of the stack.

pushType type

$\boxed{- \Rightarrow type}$

Pushes the type $type$ onto the top of the stack (as a meta-value).

2.2.2 Jumps and Call-Frames

jump

$\boxed{addr \Rightarrow -}$

The next instruction executed will be located at $addr$.

branchTrue

$\boxed{value_{boolean}\ addr \Rightarrow -}$

Jumps to the given location $addr$ if the value $value_{boolean}$ is true.

branchFalse

$\boxed{value_{boolean}\ addr \Rightarrow -}$

Jumps to the given location $addr$ if the value $value_{boolean}$ is false.

call

$\boxed{v_n \ldots v_1 \; addr \; n \Rightarrow \vdash v_n \ldots v_1}$

Stores the old program counter (replacing it with $addr$), starts a new stack frame, and initialises the frame from the top n values on the stack. The values $v_n \ldots v_1$ are moved to the new stack frame, and are not preserved in the old one.

return

$\boxed{v_n \ldots v_1 \; n \Rightarrow v_n \ldots v_1}$

Restores the previous program counter and stack frame, appending the top n values of the current stack frame to the old frame, where n is the number of values to return (which ought to be static for a given function).

getArg

$\boxed{\vdash v_0 \ldots v_{idx} \ldots v_n \; idx \Rightarrow \vdash v_0 \ldots v_{idx} \ldots v_n \; v_{idx}}$

Pushes a copy of the idx'th value (based at 0) of the stack frame onto the top of the stack. Intended for accessing arguments passed to functions and local variables.

putArg

$\boxed{\vdash v_0 \ldots v_{idx} \ldots v_n \; value \; idx \Rightarrow \vdash v_0 \ldots v_{idx-1} \; value \; v_{idx+1} \ldots v_n}$

Reverse of getArg which assigns $value$ to the idx'th position in the stack frame. Intended for setting local variables.

2.2.3 Communications and Scheduling

makeChannel

$\boxed{string \Rightarrow chanId}$

Creates a new communications channel. The channel will be deleted when the last reference to it disappears. The $string$ is for debugging purposes only.

labelChannel

$\boxed{chanId \; string \Rightarrow -}$

Identifies a channel, $chanId$, as a particular collection of wires in the editor/simulator environment identified by $string$. Only the first such identification for a collection of wires will be used, with later calls being ignored silently.

get

$\boxed{chanId \Rightarrow value}$

Performs a peek on an input channel, $chanId$, obtaining a value $value$. The input channels to a *process* are numbered sequentially from 0 up to one less than the total number of external channels that exist. This numbering is not the numbering visible to the process scheduler. The operation is a peek (get is rather similar to `chan_peek` in LARD), and it is the responsibility of the code running in the virtual machine to make sure that there is actually a value present before calling this instruction.

put

$\boxed{value \; chanId \Rightarrow -}$

Performs a poke of a value, $value$, to an output channel, $chanId$. The output channels are numbered in a manner analogous to, but separately from, the input channels. The operation is a poke, and does not ensure that the value is communicated. Furthermore, the operation should not be performed during an output action, since this could lead to the receiving end of the communication getting an inconsistent value.

signalForward

$\boxed{chanId \Rightarrow -}$

Sends a request on an output channel, $chanId$. If there are several signalForward instructions in a row (with no channel or scheduling instructions interspersed), the effects of the signalForward instructions are seen simultaneously.

waitRelease

$\boxed{chanId \Rightarrow -}$

Waits for an output channel, $chanId$, to be released. No wait occurs if the channel has already been released.

isAvailable

$\boxed{chanId \Rightarrow value_{boolean}}$

Places on the stack whether an input channel, $chanId$, has a request waiting to be serviced. This does not cause the process to become descheduled (isAvailable is rather like `chan_ready` in LARD) and can be used safely on the same channel multiple times in the same activation of a process.

release

$\boxed{chanId \Rightarrow -}$

Sends an acknowledgement on an input channel, $chanId$. Half of the request/acknowledge system used to implement rendezvous communications at the reading side.

sleep

$\boxed{- \Rightarrow -}$

Causes the process to become descheduled until something happens (i.e. the thread is woken by another thread attempting to interact with it.) Any event, even a spurious one, can cause the process to be woken, so care is required after this to see if there really is anything to do.

delay

$\boxed{value_{\Delta t} \Rightarrow -}$

Delays the executing thread for Δt time units.

2.2.4 Arithmetic Operations

add

$\boxed{x\ y \Rightarrow x + y}$

Addition. Can take any enumerable values as input, and generates integer output.

sub

$\boxed{x\ y \Rightarrow x - y}$

Subtraction. Can take any enumerable values as input, and generates integer output.

mult

$\boxed{x\ y \Rightarrow x \times y}$

Multiplication. Can take any enumerable values as input, and generates integer output.

div

$\boxed{x\ y \Rightarrow x \div y}$

Division. Can take any enumerable values as input, and generates integer output. Attempts to divide by zero generate a checked error that terminates the process.

mod

$\boxed{x\ y \Rightarrow x \bmod y}$

Remainder. Can take any enumerable values as input, and generates integer output. The same caveat that applies to division by zero applies here also.

neg

$\boxed{x \Rightarrow -x}$

Negation. Can take any enumerable value as input, and generates integer output.

abs

$\boxed{x \Rightarrow |x|}$

Absolute value (negation if less than zero). Can take any enumerable value as input, and generates integer output.

2.2.5 Bitwise/Logical Operations

and

$\boxed{x\ y \Rightarrow x \& y}$

Bitwise and. Can take any enumerable value as input, and generates integer output, unless *both* inputs were boolean values, in which case it generates boolean output.

or

$\boxed{x\ y \Rightarrow x | y}$

Bitwise (inclusive) or. Can take any enumerable value as input, and generates integer output, unless *both* inputs were boolean values, in which case it generates boolean output.

xor

$\boxed{x\ y \Rightarrow x\hat{\ }y}$

Bitwise exclusive or. Can take any enumerable value as input, and generates integer output, unless *both* inputs were boolean values, in which case it generates boolean output.

shl

$\boxed{x\ y \Rightarrow x \ll y}$

Shift left without check for overflow. Can take any enumerable values as input, and generates integer output.

shr

$\boxed{x\ y \Rightarrow x \gg y}$

Shift right without check for underflow. Can take any enumerable values as input, and generates integer output.

not

$\boxed{x \Rightarrow \bar{x}}$

Bitwise inversion. Can take any enumerable value as input and will generate integer output unless the input was a boolean value, when the generated value will also be a boolean.

2.2.6 Comparisons

eql

$\boxed{x\ y \Rightarrow x = y}$

Equality. Defined over everything, though for some opaque handles it is possible that they might be identified as being different when they are in fact the same. Guaranteed to work correctly for ordinary values, types, strings, and channel and register identifiers.

neq

$\boxed{x\ y \Rightarrow x \neq y}$

Inverted equality. See eql for notes, as these instructions are implemented using the same code.

leq

$\boxed{x\ y \Rightarrow x \leq y}$

Less than or equal to. Defined for ordinary values and strings.

lt

$\boxed{x\ y \Rightarrow x < y}$

Less than. Defined for ordinary values and strings.

geq

$\boxed{x\ y \Rightarrow x \geq y}$

Greater than or equal to. Defined for ordinary values and strings.

gt

$\boxed{x\ y \Rightarrow x > y}$

Greater than. Defined for ordinary values and strings.

2.2.7 Stack Manipulation

pop

$\boxed{any \Rightarrow -}$

Removes one item from the top of the stack.

popAll

$\boxed{\vdash \ldots \Rightarrow \vdash}$

Clears the current stack frame.[2]

exch

$\boxed{x\ y \Rightarrow y\ x}$

Exchanges the top two stack elements.

dup

$\boxed{x \Rightarrow x\ x}$

Duplicates the top stack element.

roll

$\boxed{v_{n-1} \ldots v_0\ n\ j \Rightarrow v_{(j-1) \bmod n} \ldots v_0\ v_{n-1} \ldots v_{j \bmod n}}$

[2] I think I probably ought to mark this popAll as deprecated, now that all channel identifiers have to be maintained on the stack.

Rolls the stack contents like the POSTSCRIPT `roll` operator. The n argument says how many values are to be rotated, and the j argument says by how much and in what direction the rotation is to take place, with the way it is written above being the way it works with positive numbers, and the reverse operation is performed by using the opposite sign for j.

index

$\boxed{v_n \ldots v_1\ j \Rightarrow v_n \ldots v_1\ a_j}$

Duplicates the j'th element on the stack at the top of the stack. *Or is it counting from zero? Need to check...*

killStackRange

$\boxed{\vdash v_0 \ldots v_{m-1} \ldots v_{k-n} \ldots v_{k-1}\ m\ n \Rightarrow \vdash v_0 \ldots v_{m-1}\ v_{k-n} \ldots v_{k-1}}$

Deletes all the values from the current stack frame apart from the m bottom-most items and the n top-most items.

popAll is the same as killStackRange 0 0, and you can build a simulated stack frame of n items (minus the calling information) using killStackRange 0 n

2.2.8 Type Manipulation

getType

$\boxed{value_{type} \Rightarrow type}$

Gets the type of $value$. A caught error will occur if applied to an untyped value (e.g. a string or type).

setType

$\boxed{value\ type \Rightarrow value_{type}}$

Produces a new value using $value$ as a basis for the value and $type$ as the type of the new value (the value will ordinarily be the same, but setting the type of a value to boolean causes a change in the value itself).

2.2.9 Thread Manipulation

parallel

$\boxed{\vdash v_0 \ldots v_m\ addr_0 \ldots addr_{n-1}\ n \Rightarrow \vdash v_0 \ldots v_m}$

Executes the code starting at $addr_0$ in parallel with the code starting at $addr_1$ in parallel with ... in parallel with the code starting at $addr_{n-1}$. Execution resumes at the instruction following this one once both of those parallel executions have terminated.

Each of the executing sub-threads has a copy of the (top-most) stack frame (after all the addresses have been popped) as its *sole* stack frame, and after the termination of all of them, the stack frame of the parent thread will have the same stack frame at the top as all of the sub-threads received, though it will also have all the stack frames below the current one as well, unlike the sub-threads. Thus, it is a checked error for a sub-thread to attempt to return from its initial stack frame, even if the parent of that thread was not in its topmost frame.[3]

exit

$\boxed{- \Rightarrow \mathrm{N/A}}$

Terminates the executing thread. If this thread is the only non-terminated child of another thread (the parent), then the parent will become runnable. It is an error for a top-level thread to terminate.

[3] In other words, the number of return instructions performed by a thread at any point must not exceed the number of call instructions performed by that thread — a sane restriction, in my opinion.

2.2.10 Compound Value Manipulation

makeList

$\boxed{v_0 \ldots v_{n-1}\ n \Rightarrow list}$

Makes a list[4] of length n out of elements v_0 to v_{n-1}.

split

$\boxed{list \Rightarrow v_0 \ldots v_{n-1}\ n}$

Splits a list $list$ up into its elements and length. The exact reverse of makeList.

length

$\boxed{list \Rightarrow length}$

Gets the length of the list $list$.

project

$\boxed{list\ index \Rightarrow value}$

Gets the $index$'th item of list $list$. Indexing is based at 0.

insert

$\boxed{list\ index\ value \Rightarrow newList}$

Creates a new list that is a copy of $list$ except that the $index$'th item has been updated to $value$. Intended for use during assignment to array variables.[5]

makeUnion

$\boxed{value\ unionType\ index \Rightarrow value_{union}}$

For making unions. The index may either be an enumerable value which says which tag to use (numbering the tags from zero in the order they are defined in in the $unionType$ description,) or a string containing the name of the tag, which will be looked up in the given type. *I think I've implemented this OK, but I would appreciate other people's opinions.*

openUnion

$\boxed{value_{union} \Rightarrow value\ index}$

For unmaking unions (the reverse of makeUnion). The $index$ value will be of enumeration type, where the enumeration is built from the list of tag names. The type of the union is not put on the stack by this instruction, as it can always be obtained using the getType instruction. *I think I've implemented this OK, but I would appreciate other people's opinions.*

2.2.11 Register Manipulation

makeRegister

$\boxed{string \Rightarrow regId}$

Creates a new register and returns a handle to that register for use with the load and store instructions. The register will be garbage-collected when the last reference to it disappears. The $string$ is for debugging purposes.

All registers contain $value_{unknown}$ initially.

[4] Lists are the fundamental system type used to implement tuples and vectors.

[5] The current implementation is inefficient, copying the whole list each time. Future implementations will probably only copy shared lists, otherwise updating in-place (i.e. when they have the sole copy.)

load

$\boxed{regId \Rightarrow value}$

Gets the current value of the register whose handle is $regId$.

store

$\boxed{value\ regId \Rightarrow -}$

Assigns the value $value$ to the register whose handle is $regId$.

2.2.12 Miscellaneous

noop

$\boxed{- \Rightarrow -}$

No operation.

error

$\boxed{string \Rightarrow \mathrm{N/A}}$

Causes an entire simulation (not just a thread) to terminate with the message in $string$.

tclExec

$\boxed{string \Rightarrow data\ boolean_{true} \text{ or } boolean_{false}}$

Executes $string$ as a Tcl command in the host system, producing either a $true$ value and a data value (if a data value is the result of the script) or a $false$ value *with no data value* if the result of the script is an empty string. All other script results cause exceptions out to the hosting simulation environment.

This instruction is intended for use in the implementation of source nodes, and shouldn't be generated from any ordinary *Yellow* or *Green* source code.

NDChoice

$\boxed{addr_0 \ldots addr_{n-1}\ n \Rightarrow -}$

Jumps to an address chosen randomly from the supplied collection. If $n = 0$, no jump is performed and the next instruction to be executed will be the one following this instruction. Note that this instruction might trigger rescheduling of the process.

labelDevice

$\boxed{regId\ string \Rightarrow -}$

Identifies the current thread of execution as implementing the device named $string$ by the editor/simulator environment. Furthermore, the register, $regId$, will be taken as containing a value that represents the (major) state of the device.

2.3 Projected Instructions

All of these are not currently implemented through into the execution model, though I intend to get around to them. You might be able to assemble code that uses them though.

An empty list, I have nothing projected for the near future. . .

Maybe also strings (more completely than at present), files and some other fancy features.

2.4 Derived Instructions

These are not currently implemented directly, but are instead built using calls to a standard procedure, which are transparently inserted in place of the instructions. The locations (all of which are in module `__basic__`) of the implementations of these instructions are mentioned solely for reference.

waitFor

$\boxed{chanId \Rightarrow -}$

Waits until a value is available on input channel $chanId$. This instruction is simulated by a call to `waitFor`

waitGet

$\boxed{chanId \Rightarrow value}$

Waits until a value is available on input channel $chanId$ and then reads that value from the channel (i.e. a waitFor followed by a get.) This instruction is simulated by a call to `waitGet`

read

$\boxed{chanId \Rightarrow value}$

Like waitGet except in that it also releases input channel $chanId$. The simplest way to implement the reading part of the `chanId?variable` command in *Yellow*. This instruction is simulated by a call to `read`

write

$\boxed{value\ chanId \Rightarrow -}$

Writes value $value$ on output channel $chanId$, sending a request on that channel and receiving an acknowledgement of that request. The simplest way to implement the writing part of the `chanId!expression` command in *Yellow*. This instruction is simulated by a call to `write`

3 Standard Scheduling Model

Each separate thread of execution (termed a thread) runs non-preemptively, but certain instructions can cause rescheduling. It is not defined when these reschedulings will actually occur (with the exception of the sleep instruction, which always reschedules the thread, and the delay instruction, which always reschedules when a non-zero argument is passed to it), but where multiple instructions of the same kind are placed in a row (such as several signalForward instructions,) these instructions will get executed at once. A full request/acknowledge cycle on a channel is guaranteed to cause rescheduling of both the reader and writer sides of the communication at least once. The majority of instructions have no effect outside the thread, and those don't cause any rescheduling.

In theory, I could use a preemptive multi-tasking implementation, and you wouldn't be able to tell the difference with normal code. It's just that I don't implement it like that at the moment.

4 Data Representation

Data is represented using references to shared values. When the number of references to a particular data value drops to zero, that value is deleted.[6] From the point-of-view of the simulation system, all data values are first-class, though you are probably better off only passing strings, enumerable and compound data values (and in the future possibly types) through channels, as other types do not (necessarily) have useful printable representations and the ability to be recreated should the internal representation be lost. Each node may have a type associated with it, though it may not necessarily make any sense to do so (since associating a type with a type, a string or a thread is a little loopy, and there is currently no defined use for a type attached to a channel, though an argument could well be made for implementing this in the future.)

The most important kinds of data values are enumerable and compound data values, which are used together with the former forming leaves on the tree-structure generated by the latter. Whether an enumerable data value is

[6]This scheme makes reloading a state during deadlock/model checking much easier and faster, as I can store the data value directly for reloading as opposed to having to convert it into some persistent form.

a boolean, integer or enumeration is determined solely by the type associated with the node, and the same holds true for the difference between records and vectors (both based on compound data value nodes which implement lists). I think I'll implement unions with something like a pair, with a string in the first member determining which member of the union is present and, thus, what the type of the value in the second member.

5 Standard Device Invocation Stack Frame

The following is the initial stack frame layout that all standard devices (those described below) have. I suggest that something similar be used for devices generated through the compilation of *Rainbow* text, such as user-defined function nodes, tables and chunks of *Yellow*, but that is not technically necessary. Note that none of the standard devices register themselves with the calling system, as this is something that is better done by the code that calls them (after each thread has been started with the parallel instruction, but before the jump into the code that implements the behaviour of the device) since it promotes code reuse.

$$\vdash \mathit{regId}\ k\ m\ n\ \mathit{parm}_0 \ldots \mathit{parm}_{k-1}\ \mathit{input}_0 \ldots \mathit{input}_{m-1}\ \mathit{output}_0 \ldots \mathit{output}_{n-1}$$

Register regId is used to represent the major state of the device to the simulation system (for model and deadlock checking). The meaning of parameters $\mathit{parm}_0 \ldots \mathit{parm}_{k-1}$ is determined by the device itself. Note that counts of parameters, input channels and output channels are provided so as to permit the setting of defaults to parameters and variable numbers of inputs and outputs, though I suppose it would be possible to make the number of parameters at least fixed for a particular device.

5.1 Standard Devices

The following devices have been implemented using this technique in module `____stddevs____` in the file .../lib/stddevs.rasm.

Arbitrate Arbitrates take two parameters (the delay going forward across the node, and the delay going backwards across the node,) two input channels and one output channel. They are implemented by the code located at `arbitrate`

Buffer (both empty and initialised). Empty buffers take one parameter (the time between completing a receive action and starting a transmit action). Initialised buffers take two parameters; the initial contents of the buffer, and the delay time as with the empty buffer. Both kinds of buffer take one input channel and one output channel. They are implemented by the code located at `emptyBuffer` and `initialisedBuffer`

Duplicate Duplicates take two parameters (the delay going forward across the node, and the delay going backwards across the node,) one input channel, and at least two output channels. They adapt to cope with the number of output channels that they are supplied with. They are implemented by the code located at `duplicate`

Function (the standard function nodess from the editor[7]) Functions take two parameters (the delay due to the calculation of the function, and the delay when the function is resetting itself upon receipt of an acknowledgement) and a number of input and output channels that is determined by the particular function. In practise, this means either one or two inputs, and one output. They are implemented by the following code, where what the code does should be pretty much obvious from its name:

- `arithmetic_plus`
- `arithmetic_minus`
- `arithmetic_multiply`
- `arithmetic_divide`
- `compare_equal`
- `compare_less`
- `compare_more`

[7] I could probably do these using the technique of generating ordinary source. I didn't.

- `binary_and`
- `binary_or`
- `binary_xor`
- `binary_not`
- `logical_and`
- `logical_or`
- `logical_xor`
- `logical_not`

Merge Merges take two parameters (the delay going forward across the node, and the delay going backwards across the node,) at least two input channels and one output channel. The first input channel is always the chooser channel. They are implemented by the code located at `merge`

Sink Sinks take one input parameter (the time to accept a value,) one input channel, and no output channels. They are implemented by the code located at `sink`

Split Splits take two parameters (the delay going forward across the node, and the delay going backwards across the node,) two input channels, and at least one output channel. The first input channel is the chooser channel. They are implemented by the code located at `split`

Source (constant and non-constant) Constant sources take two parameters (the constant to produce and the delay between one interaction and the next) and non-constant sources also take two parameters (the first parameter is a string containing a Tcl script which produces the value stream, and the second is the delay between one interaction and the next). Both kinds of sources take no input channels and one output channel. Constant sources are implmented by the code located at `constant` and non-constant sources are implemented by the code located at `source`

Synchronise Synchronise nodes take two parameters (the delay going forward across the node, and the delay going backwards across the node,) and the same number of input channels as output channels. The number of input channels must be at least two. They are implemented by the code located at `synchronise`

Tuple Tuple nodes take two parameters (the delay going forward across the node, and the delay going backwards across the node,) at least two input channels, and one output channel. They are implemented by the code located at `tuple`

Untuple Untuple nodes take two parameters (the delay going forward across the node, and the delay going backwards across the node,) one input channel, and at least two output channels. The number of output channels is constrained to be the same as the number of elements in the tuples being received on the input channel. They are implemented by the code located at `untuple`

A Examples

Note that these examples assume that the derived instructions are available for brevity, as well as using a slightly modified version of the device stack frame. I will update these to be correct when the details of the exact supported instruction set and standard stack frame have been finalised.

A.1 Function Node Implementations

```
; This is a quick example of something to add 1 to a
; stream of values (i.e. an increment function)

;;;;;; First, we wait for a value to become available on
;;;;;; the channel at argument 0
:WaitForSignal
    isAvailable   @0
    branchTrue    :ICanReadNow
    sleep
    jump          :WaitForSignal

;;;;;; Now that we have it, we read that value and call a
;;;;;; function to perform the addition before passing on
;;;;;; the incremented value on the channel at argument 1
:ICanReadNow
    get           @0
    push          1
    call          :Addition 2
    put           @1
    signalForward @1
    waitRelease   @1

;;;;;; We've done with the input. Release it and back to the
;;;;;; beginning to handle the next value
    release       @0
    jump          :WaitForSignal

;;;;;; Addition function
:Addition
    add           @0  @1
    return        1
```

This would be semantically the same as:

```
:beginning
    waitGet @0
    add     1
    write   @1
    release @0
    jump    :beginning
```

A.2 Buffer Node Implementations

An uninitialised buffer process (not a device) could be implemented as the following (though this example probably doesn't have the channels in the correct argument positions):

```
:start at the beginning
    push {:start at the beginning} @1 @0
    read
    exch
```

```
    write
    jump
```

though the following is more idiomatic code (and much easier for me to write as well):

```
:beginning
    read  @0
    write @1
    jump  :beginning
```

or even (relying on the fact that the entry point to a block of code is address 0, which is not something that I would expect to be true after the introduction of modules):

```
    read  @0
    write @1
    jump  :0
```

A.3 If Block Behaviour and Implementation

Here I am using code that has the following behaviour (described using a pseudo-*Yellow* fragment):

loop *true* →
    sync *ctl*?*c* during {
        sync *splitIn*?*x* during
            if *c* then *trueSrc*!*x* else *falseSrc*!*x* end_if
        end_sync
    } ∥ {
        if *c* then
            sync *trueSink*?*x* during *mergeOut*!*x* end_sync
        else
            sync *falseSink*?*x* during *mergeOut*!*x* end_sync
        end_if
    } end_sync
end_loop

Note that the following code uses the proposed standard device stack frame (I think), and also performs checks on the numbers of channels for sanity, failing with an error if something is up.

```
:ifBlock
; This device has no internal state.  Set a dummy value
    store         unknown @0

; Sanity-check the number of input and output channels
    eql           @1  4
    eql           @2  3
    and
    jumpTrue      :mainLoop
    error {$if block without 4 input and 3 output channels}

;;; The loop - also does the reading of the control channel
:mainLoop
    isAvailable   @3
    jumpTrue      :readCtl
    sleep
    jump          :mainLoop
:readCtl
    get           @3
    parallel      :split1 :merge1  2
    pop
```

```
        release       @3
        jump          :mainLoop

    ;;; The split part of the data handler
    :split1
        isAvailable   @4
        jumpTrue      :readSplitIn
        sleep
        jump          :split1
    :readSplitIn
        get           @4
        jumpFalse     @10 :splitFalse
    :splitTrue
        put           @7
        signalForward @7
        waitRelease   @7
        release       @4
        exit
    :splitFalse
        put           @7
        signalForward @7
        waitRelease   @7
        release       @4
        exit

    ;;; The merge part of the data handler
    :merge1
        jumpFalse     :mergeFalse
    :mergeTrue
        isAvailable   @5
        jumpTrue      :readTrueSink
        sleep
        jump          :mergeTrue
    :readTrueSink
        get           @5
        put           @9
        signalForward @9
        waitRelease   @9
        release       @5
        exit
    :mergeFalse
        isAvailable   @6
        jumpTrue      :readFalseSink
        sleep
        jump          :mergeFalse
    :readFalseSink
        get           @6
        put           @9
        signalForward @9
        waitRelease   @9
        release       @6
        exit
```

I’m sure the above code can be much more efficient.

B BNF for data and types

This is derived directly from the source file that performs the implementation of the parsing of data and types. Note that mostly `TOK_*` indicates the (lower-case version of) the word following as a keyword, but `TOK_DOTS` is "`..`", `TOK_IDENT` is all other unrecognised identifiers, and `TOK_NUMBER` is string of digits. I know that I need to update this to take into account the changes that have been subsequently introduced to the definitions of datatypes (and it would help tremendously if someone could collect all those changes together in one place for me...)

```
data : dataval
     | dataval ':' datatype

type : datatype
     | TOK_FUNCTION funcsigbody
     | TOK_PROCEDURE procsigbody

enumtype : TOK_INTEGER
         | TOK_BOOLEAN
         | TOK_ENUM '(' namelist ')'
         | '[' enumtype ':' subrangelist ']'
         | '[' subrangelist ']'

datatype : enumtype
         | '(' bundlefieldlist ')'
         | datatype '[' TOK_NUMBER ']'
         | TOK_UNION '(' namedfieldlist ')'

namelist : TOK_IDENT
         | namelist ',' TOK_IDENT

subrange : enumval TOK_DOTS enumval

subrangelist : subrange
             | subrangelist ',' subrange

namedfield : TOK_IDENT ':' datatype

namedfieldlist : namedfield
               | namedfieldlist ',' namedfield

bundlefield : TOK_IDENT ':' datatype
            | datatype

bundlefieldlist : bundlefield
                | bundlefieldlist ',' bundlefield

procsigbody : '(' sigfieldlist ')'

funcsigbody : '(' sigfieldlist ')' ':' datatype

sigfield : TOK_VALUE namelist ':' datatype
         | TOK_INPUT namelist ':' datatype
         | TOK_OUTPUT namelist ':' datatype
         | TOK_FUNCTION TOK_IDENT funcsigbody
         | TOK_PROCEDURE TOK_IDENT procsigbody

sigfieldlist : sigfield
```

```
              | sigfieldlist ';' sigfield

enumval : TOK_TRUE
        | TOK_FALSE
        | intval
        | TOK_IDENT

datavallist : dataval
            | datavallist ',' dataval

listval : '[' datavallist ']'
        | '(' datavallist ')'

dataval : enumval
        | listval
        | TOK_UNKNOWN
        | TOK_IDENT listval

intval : TOK_NUMBER
       | '-' TOK_NUMBER
```