Technical Report

MOTOROLA INC.

Cambridge Research Center

MINT White Paper

Kenneth R. Traub

Motorola Technical Report MCRC-TR-2
October 31, 1989

One Kendall Square, Building 200
Cambridge MA 02139

MINT White Paper

Kenneth R. Traub

October 31, 1989

MINT stands for Monsoon INTerpreter: it is an emulator of Monsoon, a software implementation of the Monsoon instruction set architecture. MINT is designed to mimic Monsoon exactly: given an initial value for Monsoon's machine state, both MINT and the Monsoon hardware will yield the same final state after the same number of instruction cycles. Machine state includes the contents of frame store, instruction memory, statistics registers, and token queues. MINT is an emulator rather than a simulator, in the sense that it does not model internal state of the Monsoon processor that is not considered machine state at the instruction set architecture level. For example, MINT does not model the internal pipeline registers of the Monsoon processor.

While MINT can model Monsoon exactly, it also has features which allow it to serve many if not all of the roles filled by the old TTDA GITA emulator. In particular, it can gather more detailed statistics than can the hardware, as it places no limits on the number of statistics registers or on how they are updated. Through appropriate changes to MINT's token queueing system, all of the statistics collection modes available in GITA can be obtained, including infinite processor mode, finite processor mode, finite latency mode, *etc.* Moreover, it is possible to define "opcodes" for MINT that extend beyond the cabilities of the Monsoon microarchitecture. Perhaps the most useful sets of these fictitious opcodes are the TTDA-like manager operations `get-context`, `make-I-structure`, *etc.*, and split-phase memory transactions which operate in a single cycle. By using these fictitious opcodes in compiled code, the statistics gathered by MINT will, like their GITA counterparts, not be colored by any concrete implementation of resource managers on Monsoon.

The aim of this paper is to introduce the reader to the MINT program, exploring in detail how the program works. In presenting MINT, we will take liberties and gloss over many details, but try to get the essential structure and flavor of the program across. Perhaps the greatest liberty we will take is in presenting MINT in Id; MINT is actually written Common Lisp, with an implementation in C planned. As such, this paper does not document the MINT program at the level required by MINT's maintainers. Nevertheless, after reading this paper, the reader should understand exactly what MINT's capabilities are, what statistics it is capable of collecting, and also have a rough feel for the performance issues.

1 Kudos

The first version of MINT was written by Andy Shaw as his Bachelor of Science Thesis. Greg Papadopoulos, Ken Traub, and Jonathan Young have all had significant input into the design process. As will be discussed in Section 6, part of MINT can be generated by a program called MUC, which was written by Derek Chiou. MINT and MUC are currently being revised and maintained by Darin DeForest and Mike Beckerle.

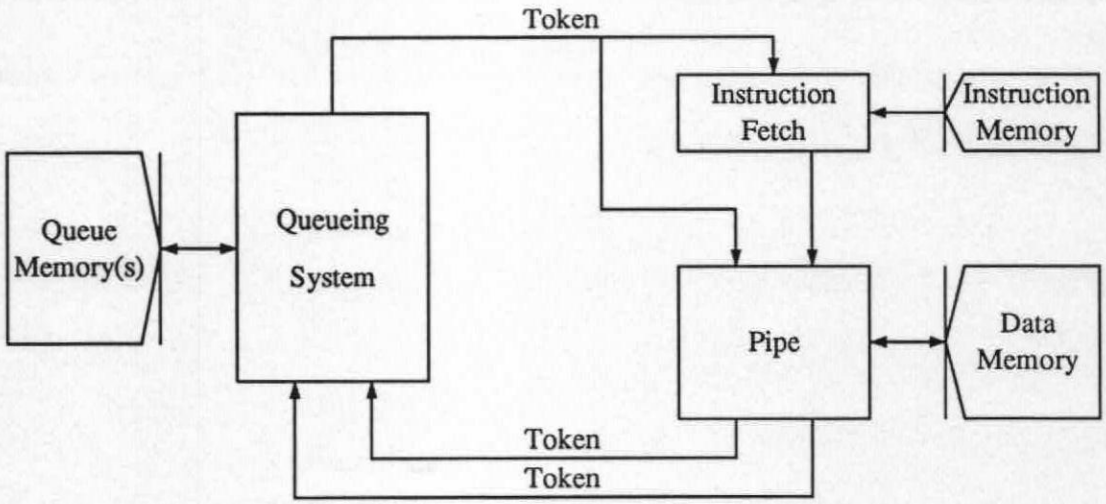


Figure 1: High-Level Flow of Data in MINT

MINT has been heavily influenced by the experience with TTDA GITA, which was primarily written by Dinarte Morais, Richard Soley, David Culler, and Ken Traub.

2 Top Level

At the highest level, MINT is a gigantic loop, where each iteration of the loop emulates the processing of one token. The flow of data in this loop is depicted in Figure 1. A token is fetched from the token queuing system, the corresponding instruction is fetched from Instruction Memory (IM), processed by the main pipeline, yielding zero, one, or two tokens which are enqueued back into the queueing system.

Figure 1 omits some details that we will pick up in later sections. First, it is assumed that we are emulating only one PE. The emulation of multiple PE's is discussed in Section 10. Second, statistics collection and the associated state is postponed until Section 9. Third, the registers feature is omitted, and will be discussed in Section 11. Fourth, Monsoon allows the pipeline to read and write instruction memory and the queues, in addition to frame memory. Those paths are present in MINT, but omitted from most of the discussion in this paper, for clarity.

There is one significant difference between Figure 1 and Monsoon that is not just an omission from this paper: the entire pipeline less the instruction fetch phase is treated as single black box. In Monsoon, the pipeline has a total of eight stages, each with well defined interfaces to the next. By not modeling the internal stages, MINT is able to operate much faster than would a true simulator of Monsoon, and is also able to accomodate fictitious opcodes which would not sensibly decompose into stages. The drawback is that MINT cannot properly emulate conflicts between tokens occupying different stages of the Monsoon pipeline at the same time. Fortunately, such conflicts can arise only in extremely rare and perverse circumstances, so their proper emulation is judged not worth the performance and flexibility penalty.

The top level of MINT is a function which takes an initial state of instruction memory, data memory, and the queueing system, and returns an updated data memory and queueing system when there are no more tokens to queue. The top level is given in Figure 2. We consider it line

```
def mint im dm qsys =
  %% We loop as long as there is something in the queueing system to do.
  {while something_to_do? qsys do

    %% First, dequeue a token and take apart its tag.
    tok, post_dq_qsys = dequeue qsys;
    Token tag_type tag_value data_type data_value = tok;
    Tag port ip map pe fp = tag_value;

    %% Fetch the instruction.
    inst = instruction_fetch pe im ip;

    %% Perform the operation, obtaining two output tokens (one or both of
    %% which may be empty), and an updated data memory.
    outtok1, outtok2, next dm = pipe inst tok dm;

    %% Enqueue the output tokens, yielding an updated queueing system.
    post_tok1_qsys = enqueue outtok1 post_dm_qsys;
    post_tok2_qsys = enqueue outtok2 post_tok1_qsys;
    next qsys = post_tok2_qsys;

  finally dm, qsys};
```

Figure 2: The top level of MINT.

by line below.

```
def mint im dm qsys =
  {while something_to_do? qsys do
     ...
   finally dm, qsys};
```

As stated earlier, MINT is just a big loop that iterates until there are no tokens to process. Hence, it terminates when the queueing system says there is nothing to do. The exact definition of the function `something_to_do?` will have a very important effect on statistics collection, as we discuss in Section 9. At the end of the loop the data memory and queueing system will have been modified, so we return the new versions to the caller.

```
tok, post_dq_qsys = dequeue qsys;
```

Dequeueing a token from the queueing system has the side effect of modifying the state of the queues. We indicate this by returning a new queueing system `post_dq_qsys` differing from the old queueing system `qsys` in that the token `tok` has been removed. (In Id, we can't just side-effect `qsys`! In realistic implementations of MINT, of course, `dequeue` returns a token and side-effects the queueing system.)

```
Token tag_type tag_value data_type data_value = tok;
Tag port ip map pe fp = tag_value;
inst = instruction_fetch im ip;
```

We take apart the token and further take apart its tag, to get at the ip value we need for the instruction fetch.

```
outtok1, outtok2, next dm = pipe inst tok dm;
```

The function pipe dispatches on the opcode contained in the instruction and executes it, side-effecting the data memory and yielding up to two tokens to queue. Again, in Id the side-effects to data memory are represented by having pipe return a new, modified, data memory. You may wonder what happens if fewer than two tokens are to be produced. The answer is that outtok1 and outtok2 may be a distinguished value NoToken instead of a token. When outtok1 or outtok2 is a token, it actually carries some additional information with it that tells the queueing system how to queue it. Exactly what this information is depends on the queueing system: if using a queueing system that emulates Monsoon, for example, this information will indicate whether the token is to be recirculated or placed in a token queue; if using a finite-latency idealized queueing system, this information will indicate how much latency to add. These issues are discussed in more detail in Section 9.

```
post_tok1_qsys = enqueue outtok1 post_dm_qsys;
post_tok2_qsys = enqueue outtok2 post_tok1_qsys;
next qsys = post_tok2_qsys;
```

The tokens emitted by pipe are enqueued. The function enqueue enqueues a token, returning the modified state of the queueing system. If the first argument to enqueue is the special value NoToken, the value returned is the same as the second argument. The queueing system returned after enqueueing the second token becomes the queueing system for the next iteration of the main loop.

The "working parts" of MINT are embodied in the functions instruction_fetch and pipe, and in the functions which operate on the queueing system: enqueue, dequeue, and something_to_do?. We discuss these in detail in the following sections. First, however, we examine in more detail the data structures and their associated types.

3 Data Structures and Functions

3.1 Tokens and Token Queues

A token has four components: tag type, tag value, data type, and data value. The type components are each just unsigned integers. The value components are of type data, defined below.

```
type tok = Token N data N data;
```

As mentioned in the previous section, the pipe function needs to return a value that is either a token or an indication of no token. This is called an *output token.*

```
type output_tok = NoToken | YesToken tok enqueueing_command;
```

The second component of the YesToken disjunct, as mentioned previously, is an indication of how to enqueue the token, whose meaning depends on the queueing system in use.

The functions which operate on the token queue, all discussed earlier, have the following signatures.

```
typeof enqueue          = output_tok -> queueing_system -> queueing_system;
typeof dequeue          = queueing_system -> (tok, queueing_system);
typeof something_to_do? = queueing_system -> B;
```

3.2 Data

The data on which the Monsoon hardware operates are 64 bit words. At various times, the hardware may interpret such a word as an unsigned integer, as a two's complement signed integer, as an IEEE double precision floating point number, or as a tag. While we could just always represent data as an integer within MINT, for efficiency reasons it is desirable to represent data in one of four forms, depending on the context in which it is used:

```
type data = Bits N | Integer N | Float N | Tag N N N N N;
```

The five components of the Tag disjunct are, from left to right, *port*, *ip*, *map*, *pe*, and *fp*.

It is very important not to be misled by the representation of data as a union type. In the hardware, there is no way to tell if a given word of data memory contains an integer, a floating point number, or a tag; instead, the hardware interprets the 64 bits as one of these depending on what it wants to do with the data. Therefore, in MINT, *it is forbidden to do something different depending on which disjunct of the union type a given data word is.* To do so would be to rely on information that is not actually present in Monsoon.[1] The representation of data as a union type is purely an efficiency hack; it allows MINT to avoid repeated parsing of a word if, for example, it is read and used as a tag repeatedly.

Because the internal functions of MINT are not allowed to ask what type of data a given data word is, before using a word of data it must use one of the following four functions:

```
typeof view_data_as_bits = data -> N;
def view_data_as_bits d =
  {case d of
      Bits  i                  = i
    | Int   i                  = integer_to_bits i
    | Float x                  = float_to_bits x
    | Tag   port ip map pe fp  = tag_to_bits port ip map pe fp};

typeof view_data_as_integer = data -> N;
def view_data_as_integer d =
  {case d of
      Bits  i                  = bits_to_integer i
    | Int   i                  = i
    | Float x                  = float_to_integer x
    | Tag   port ip map pe fp  = tag_to_integer port ip map pe fp};
```

[1] Of course, the software running on Monsoon may enforce a convention wherein the type bits associated with a data word indicate whether the data word is an integer, float, or tag. Such a convention could not have any impact on the internal operation of MINT, but can and would be very useful to a program like a debugger which examines the state of MINT's memory after a program has executed.

```
typeof view_data_as_float = data -> N;
def view_data_as_integer d =
  {case d of
      Bits  i                  = bits_to_float i
    | Int   i                  = integer_to_float i
    | Float x                  = x
    | Tag   port ip map pe fp = tag_to_float port ip map pe fp};

typeof view_data_as_tag = data -> (N, N, N, N, N);
def view_data_as_tag d =
  {case d of
      Bits  i                  = bits_to_tag i
    | Int   i                  = integer_to_tag i
    | Float x                  = float_to_tag x
    | Tag   port ip map pe fp = port, ip, map, pe, fp};
```

A conversion function like `float_to_tag` answers the question: if in Monsoon, I stored the result of a floating point add in a word of memory, then later fetched it out and used it as the left argument to `change-tag`, what tag would I obtain? Such a question is highly machine-dependent, as it depends on the details of the floating point representation, and on the encoding of a tag. These conversion functions are the *only* pieces of code within MINT that have knowledge of how integers, floats, and tags are encoded into bits within the Monsoon architecture. The conversion functions will never be invoked, however, if a word is always used in a consistent way. That is, if every word of data memory written as a float is always read as a float, then `view_data_as_float` will never call the conversion functions x`_to_float`. In particular, a correctly compiled Id program can never cause a call to a conversion function within MINT. The detailed machine knowledge that MINT has is thus *isolated* to these few places.

To give a feel for what a conversion function is like, here are two of them, defined for the "Rev 2" version of Monsoon:

```
%% Convert an unsigned integer to a 64-bit two's complement signed integer.
def bits_to_int i =
  if i < 2^63 then
    i
  else
    i - 2^64;

%% Convert a tag to a 64-bit unsigned integer.
def tag_to_bits port ip map pe fp =
  port * 2^63 +
  ip   * 2^32 +
  map  * 2^55 +
  pe   * 2^22 +
  fp;
```

3.3 Data Memory

Data memory is simply an array, indexed by PE and FP:

```
typesyn dm = 2d_array (N, N, data);
```

The three components of a word of data memory, from left to right, are presence bits, type bits, and data. Presence and type are simply represented as unsigned integers.

The contents of data memory may be fetched simply by array subscripting: `dm[pe, fp]`. The Dennis-style `dm_append` function "writes" data memory by returning a new data memory differing at one location:

```
typeof dm_append = dm -> N -> N -> (N, N, data) -> dm;
def dm_append dm pe fp new_contents =
  {(low_pe, hi_pe), (lo_fp, hi_fp) = 2d_bounds dm;
   in
     {2d_array ((low_pe, hi_pe), (lo_fp, hi_fp))
       [p, f] =
         if (p == pe) and (f == fp) then
           new_contents
         else
           dm[p, f]
       || p <- low_pe to hi_pe & f <- low_fp to hi_fp}};
```

Of course, in a real implementation of MINT the `dm_append` function would just side-effect the data memory array.

3.4 Instruction Memory

Instruction memory is simply an array of instructions:

```
typesyn im = 2d_array inst;
type inst = Instruction N N N;
```

The three components of an `Instruction` are, from left to right, *opcode*, *f1*, and *f2*. Depending on the instruction, the fields *f1* and *f2* are interpreted as frame offsets, tag adjustments, destinations, or register specifications. The following functions are used when a field is to be interpreted as a destination or long *r* value.

```
typeof dest_port = N -> N;
typeof dest_s    = N -> N;
typeof long_r    = N -> N -> N;
```

`dest_port` extracts the *port* value from an *f1* or *f2* field, `dest_s` extracts the IP offset, and `long_r` combines the *f1* and *f2* values into a single "long" *r* value.

4 Instruction Fetch

The function `instruction_fetch` fetches the instruction indicated by a token's tag from instruction memory. It is, not surprisingly, quite simple:

```
typeof instruction_fetch = N -> N -> im -> inst;
def instruction_fetch pe ip im =
  im[pe, ip];
```

5 The Pipeline

The function pipe is where most of the work in MINT goes on. It has the following signature:

```
typeof pipe = inst -> tok -> dm -> (output_tok, output_tok, dm);
```

As discussed earlier, pipe takes an instruction, and token, and data memory, and returns two output tokens and an updated data memory. Each output token, it should be remembered, is either an actual token (together with an enqueueing command) or the value NoToken.

The function pipe as usually implemented within MINT simply uses the opcode of the instruction as an index into an opcode array, each element of which is a function that performs a specific opcode.

```
typeof pipe_function_table = vector (N -> N -> N ->
                                     N ->
                                     N -> N -> N -> N -> N ->
                                     N -> data -> dm ->
                                     (output_tok, output_tok, dm));

def pipe inst tok dm =
  {%% First, take apart the instruction, the token, and its tag.
   Instruction opcode f1 f2 = inst;
   Token tag_type tag_value data_type data_value = tok;
   Tag port ip map pe fp = tag_value;

   %% Get the appropriate function from the opcode table...
   pipe_function = pipe_function_table[opcode];
   in
     %% ... and run it.
     pipe_function opcode f1 f2
                   tag_type port ip map pe fp
                   data_type data_value
                   dm
   };
```

We've shown the destructuring of the instruction and the incoming token here in pipe, which passes the components as arguments to the functions stored in the opcode table. Of course, it is fairly arbitrary whether this destructuring takes place here, in the top level mint function, or down in each of the pipe functions themselves. We show the destructuring in pipe mainly for expository reasons: it keeps both the top level mint function and the opcode table functions free from clutter. The actual code of MINT takes a different approach, for maximum efficiency.

We illustrate some typical opcode table functions in the next few sections.

5.1 Example Pipe Function: Negate

The following code is the pipe function for a unary floating point negate instruction, with one (explicit) destination, which we interpret one line at a time.

```
def pipe_function_float_negate_1_dest
    opcode f1 f2 tag_type port ip map pe fp data_type data_value dm =
  {
```

The arguments are as described in the definition of `pipe`.

```
result_value = Float (0.0 - view_data_as_float data_value);
```

Before negating the value on the arriving token, it must be converted to a float if necessary. Note that `view_data_as_float` is simply an identity if the incoming token had a floating point number, which we expect always to be the case. If it was something else, then it gets converted according to the architecture-specific rules, and we get a strange result (though one which matches exactly what the hardware would have done). The constructor `Float` converts the result into type `data`, so that it can become part of the result token.

```
tag1 = Tag (dest_port f1) (ip + dest_s f1) map pe fp;
```

The tag for the token to be emitted is formed by taking the incoming token's *map*, *pe*, and *fp* fields, the destination's port, and by adding the destination's IP offset to the incoming token's *ip*. For a unary instruction with a single destination, the *f1* field of the instruction is used as the destination.

```
tok1 = Token tag_type tag1 data_type result_value;
```

The result token is constructed using the destination tag and result value constructed earlier. The type of the tag is that of the incoming token, as is the result type.

```
outtok1 = YesToken1 tok1 [How to queue];
```

We form an "output token" from the token above. We omit showing what value is used as the enqueueing command, deferring such issues until Section 8.

```
in outtok1, NoToken, dm};
```

One of the "tokens" returned is the `NoToken` indicator since this instruction only produces one token. The data memory is returned unmodified.

5.2 Example Pipe Function: Plus

The pipe function for `plus` is considerably more complex, because of the matching operation. Here is the code for a plus instruction with one explicit destination:

```
def pipe_function_plus_2_dest
    opcode f1 f2 tag_type port ip map pe fp data_type data_value dm =
  {ea = EA_relative f2 fp;
   old_presence, temp_type, temp_value, new_dm =
     WM_binary pe ea data_type data_value dm;
   in
     {case old_presence of
        %% Frame slot is empty.
        0 = NoToken, NoToken, new_dm

        %% Frame slot is non-empty.
      | 1 =
          {result_value =
             Float (view_data_as_float data_value +
                    view_data_as_float temp_value);
           outtok1, outtok2 =
             form_one_token_explicit
               data_type result_value tag_type f1 ip map pe fp;
           in
             outtok1, outtok2, new_dm}}};
```

Because this instruction is more complex than `negate`, we've used some subroutines. The function `EA_relative` computes the effective address relative to the current frame pointer.

```
def EA_relative r fp =
  fp + r;
```

The function `WM_binary` does all the frame operations to implement the standard binary wait/match operation. The values `temp_type` and `temp_value` it returns are the value extracted from frame memory if a match occured, otherwise they are irrelevant. It also returns the old value of the presence bits as well as an updated data memory.

```
def WM_binary pe ea data_type data_value dm =
  {presence, temp_type, temp_value = dm[pe, ea];
   new_dm =
     {case presence of
        %% Presence = 0 means empty.
        0 = dm_append dm pe ea (1, data_type, data_value)
        %% Presence = 1 means full.
      | 1 = dm_append dm pe ea (0, temp_type, temp_value)}
   in
     presence, temp_type, temp_value, new_dm};
```

For simplicity, we've not shown what happens of the presence bits are other than "empty" or "full"; these cases would all generate some sort of exception token. Note that in the case where a match occured, the presence is reset to zero but the type and data fields are left as is. This is to reflect what actually takes place in Monsoon.

The function `form_one_token_explicit` forms the output tokens. It's simply a wrapping up of what we did in the code form `negate`:

```
def form_one_token_explicit
    data_type result_value tag_type dest ip map pe fp =
  {tok1 = FT_explicit_dest data_type result_value tag_type dest ip map pe fp;
   outtok1 = YesToken1 tok1 [How to queue];
   in
     outtok1, NoToken};

def FT_explicit_dest data_type data_value tag_type dest ip map pe fp =
  {tag1 = Tag (dest_port dest) (ip + dest_s dest) map pe fp;
   tok1 = Token tag_type tag1 data_type data_value;
   in
     tok1};
```

5.3 Example Pipe Function: Minus

`Minus` is implemented exactly like `plus`, with one additional detail. Because `plus` is associative, there was no need to distinguish whether the left or right token arrived first. In `minus`, this is necessary. To obtain the code for `minus` the following line in `plus`:

```
result_value =
  Float (view_data_as_float data_value + view_data_as_float temp_value);
```

is replaced by the following two lines:

```
a_value, b_value = flip data_value temp_value port;
result_value =
  Float (view_data_as_float a_value - view_data_as_float b_value);
```

where the function `flip` is defined as:

```
typeof flip = data -> data -> N -> (data, data);
def flip data_value temp_value port =
  {case port of
     0 = data_value, temp_value
   | 1 = temp_value, data_value};
```

5.4 Example Pipe Function: Plus with two destinations

A `plus` instruction with two destinations issues a token to the explicit destination, and also to the left port of the instruction whose *ip* is one greater than the *ip* of the `plus` instruction itself.

The following line in the one-destination `plus`:

```
outtok1, outtok2 =
  form_one_token_explicit data_type result_value tag_type f1 ip map pe fp;
```

is replaced by the line

```
outtok1, outtok2 =
  form_two_tokens_one_explicit data_type result_value tag_type f1 ip map pe fp;
```

where `form_two_tokens_one_explicit` is defined as:

```
def form_two_tokens_one_explicit
    data_type result_value tag_type dest ip map pe fp =
  {tok1 = FT_explicit_dest data_type result_value tag_type dest ip map pe fp;
   tok2 = FT_implicit_dest data_type result_value tag_type ip map pe fp;
   outtok1 = YesToken tok1 [How to queue];
   outtok2 = YesToken tok2 [How to queue];
   in
     outtok1, outtok2};
```

The new function `FT_implicit_dest` is defined as:

```
def FT_implicit_dest data_type data_value tag_type ip map pe fp =
  {tag1 = Tag 0 (ip + 1) map pe fp;
   tok1 = Token tag_type tag1 data_type data_value;
   in
     tok1};
```

6 More on Pipe Functions: Order from Chaos

MINT takes a very general view of what an opcode can do: it can perform arbitrary updates to data memory, and produce up to two tokens. Obviously, if one is to write a sensible set of pipe functions there must be more of an underlying discipline. It would be both tedious and error-prone to manually write 1024 individual functions, each of which did quite similar things.

For that reason, it is expected that the pipe functions will be derived from specifications, where the specifications are given in a language that restricts the functionality of an opcode. Typically, such a language will allow opcodes to be derived through the orthogonal combination of an effective address operation, a wait/match operation, a computational operation, and a token forming operation. One such specification language is the actual microcode for the Monsoon processor. Another is the abstract ETS instruction set definition language.

There are two basic approaches to deriving actual pipe functions from such specifications, the interpreted approach and the compiled approach. In the interpreted approach, there is only one piece of code for all pipe operations, but that piece of code does different things based on the opcode. A hypothetical example of this approach might begin as follows:

```
def pipe_function_generic_interpreted
    opcode f1 f2 tag_type port ip map pe fp data_type data_value dm =
  {ea_op, wm_op, comp_op, ft_op = decode_opcode opcode;
   ea = ea_op f2 fp;
   old_presence, temp_type, temp_value, new_dm =
     wm_op pe ea data_type data_value dm;
   ...};
```

The interpreted approach is easy to implement, but will not be terribly fast.

The compiled approach takes the specifications and actually generates code for the pipe functions, in whatever language MINT is written. That is, the MINT program will consist largely of code generated by another program. There is already a program called MUC which does just this, taking Monsoon microcode as the input specifications. It would be even easier to create a similar program for a more abstract specification langauge, such as ETS.

7 More on Pipe Functions: Fictitious Opcodes

The general view that MINT takes of what an opcode can do means that it is possible to write opcodes that could not actually be implemented in hardware. This is not important if you are using MINT to help diagnose a hardware problem, but can be very useful if you are using MINT to study the parallelism in an Id program. There are certain operations which may take many Monsoon instructions to perform, but would have been performed and accounted as a single instruction in TTDA GITA. These fall into two categories: split-phase memory transactions, and manager operations. Split-phase memory operations always take two instructions on the Monsoon hardware. Manager operations such as `get-context` and `make-i-structure` can take hundreds, and may be serialized due to locks on system data structures. Because they take more instructions, the paralellism profiles gathered will be skewed relative to those that would have been obtained from TTDA GITA, which accounts for these operations as a single instruction.

But by suitable hand-crafting of pipe instructions, it is easy to create opcodes for MINT which do a complete split-phase transaction or manager operation in a single instruction cycle. The resulting system will no longer model the behavior of the hardware, but will generate statistics that are not biased by particular manager implementations and other details. Furthermore, MINT will run somewhat faster since it will be interpreting fewer dataflow instructions.

As an example, we consider the split-phase `i-fetch` instruction. The instructions as they would be executed on Monsoon have the following definition in MINT. For simplicity, we have assumed that only one deferred read is possible.

```
def pipe_i_fetch_1st_phase
    opcode f1 f2 tag_type port ip map pe fp data_type data_value dm =
  {ea = EA_relative f2 fp;
   old_presence, temp_type, temp_value, new_dm =
     WM_binary pe ea data_type data_value dm;
   in
     {case old_presence of
        0 = NoToken, NoToken, new_dm
      | 1 =
          {%% First, add the offset to the I-structure pointer to get
           %% a pointer to the location to be fetched.
           a_value, b_value = flip data_value temp_value port;
           isd_port, isd_ip, isd_map, isd_pe, isd_fp = view_data_as_tag a_value;
           offset = view_data_as_integer b_value;
           isa_pe, isa_fp = pointer_increment isd_map isd_pe isd_fp offset;

           %% Create the tag for the fetch request token.
           tag1 = Tag isd_port I_Fetch_IP isd_map isa_pe isa_fp;

           %% Create the return address, to be sent in the data part.
           dest = f1;
           return_tag = Tag (dest_port dest) (ip + dest_s dest) map pe fp;

           %% Send the fetch request token.
           tok1 = Token I_Req_Tag_Type tag1 tag_type return_tag;
```

```
            outtok1 = YesToken tok1 [How to queue];
            in
              outtok1, NoToken, new_dm}}};

def pipe_i_fetch_2nd_phase
    opcode f1 f2 tag_type port ip map pe fp data_type data_value dm =
  {element_presence, element_type, element_value = dm[pe, fp];
   in
     {case element_presence of
        %% Empty, no deferred read:  store the return address and enter deferred state.
        0 =
          {new_dm = dm_append dm pe fp (2, data_type, data_value);
           in
             NoToken, NoToken, new_dm}

        %% Full:  send back the data to the return address.
      | 1 =
          {tok1 = Token data_type data_value element_type element_value;
           outtok1 = YesToken tok1 1;
           in
             outtok1, NoToken, dm}

        %% Already has a deferred:  error.
      | 2 =
          {...}}};
```

It is possible to implement i-fetch in a single opcode, omitting the fetch request token.

```
def pipe_fictitious_i_fetch
    opcode f1 f2 tag_type port ip map pe fp data_type data_value dm =
  {ea = EA_relative_short_r f2 fp;
   old_presence, temp_type, temp_value, new_dm =
     WM_binary ea data_type data_value dm;
   in
     {case old_presence of
        0 = NoToken, NoToken, new_dm
      | 1 =
        {a_value, b_value = flip data_value temp_value port;
         _, _, isd_map, isd_pe, isd_fp = view_data_as_tag a_value;
         offset = view_data_as_integer b_value;
         element_pe, element_fp =
           pointer_increment isd_map isd_pe isd_fp offset;
         element_presence, element_type, element_value = new_dm[pe, element_fp];
         {case element_presence of
            %% Empty, no deferred read:  store the return address and enter deferred state.
            0 =
              {return_tag = Tag (dest_port f1) (dest_s f1) map pe fp;
               newer_dm =
```

```
        dm_append new_dm element_pe element_fp (2, tag_type, return_tag);
      in
        NoToken, NoToken, newer_dm}

    %% Full: send the data.
  | 1 =
      {outtok1, outtok2 =
          form_one_token_explicit
            element_type element_value tag_type f1 ip map pe fp;
       in
        outtok1, outtok2, new_dm};

    %% Already has a deferred: error.
  | 2 =
      {...}}};
```

This function could not be accomplished in the Monsoon hardware, since it can access two different locations in data memory.

8 Queueing Systems

The queueing system records tokens emitted by the pipe, for later processing. The definition of the queueing system controls the relationship between when a token is emitted and when it later processed relative to other tokens that were emitted. It also controls the termination of MINT's top level loop.

The definition of the queueing system can have a profound impact on the order in which instructions are executed. If no statistics are being collected, and if the register feature is not being used, and if the program is determinstic (*i.e.*, compiled from the deterministic subset of Id), then instruction ordering will have no impact on the final answer computed by MINT. If statistics are collected, then the queueing system directly affects what the statistical data is actually obtained, for the queueing system will control what operations are executed as part of what timestep. These issues are explored in the section on statistics, Section 9. If the register feature is being used, then the queueing system must take care in dispatching tokens which are part of a sequential thread, so that thread's register state is preserved. These issues are tackled in Section 11.

In this section, we just present some simple queueing systems to give the reader a feel for the various possibilities. More queueing systems will be presented when we turn to statistics and registers. A queueing system is defined by saying exactly what the `queueing_system` type is, and by giving definitions for the functions `enqueue`, `dequeue`, and `something_to_do?`.

One thing all queueing systems will have in common is the behavior of `enqueue` when given the `NoToken` value. All definitions of `enqueue` will have the following form:

```
def enqueue outtok qsys =
  {case outtok of
     NoToken = qsys
   | (YesToken tok how_to_queue) = [Do something else.]};
```

8.1 Simple LIFO Queueing System

The simplest possible queueing system is a single LIFO queue. The queueing system is just an array of tokens, together with an integer giving the index of the next available location in the array. As the queue fills up the index grows more positive, and the queue is empty when the index returns to zero. The "how to enqueue" information is ignored.

```
typesyn queueing_system = (vector tok), N;

def enqueue outtok qsys =
  {case outtok of
     NoToken =
       qsys};
   | (YesToken tok how_to_queue) =
       {qarray, next_empty = qsys;
        new_qsys = (append qarray next_empty tok), next_empty + 1;
        in
          new_qsys};

def dequeue qsys =
  {(qarray, next_empty) = qsys;
   tok = qarray[next_empty - 1];
   new_qsys = qarray, next_empty - 1;
   in
     tok, new_qsys};

def something_to_do? qsys =
  {qarray, next_empty = qsys;
   in
     next_empty > 0};
```

This could easily be changed to support a FIFO queue.

8.2 Monsoon Queueing System

(This section is rather long, and may be omitted if the reader does not care to see the details of how the Monsoon queueing system is emulated.)

The Monsoon queueing system precisely emulates the scheduling of instructions on the Monsoon hardware, and as such is useful in debugging of both hardware and system software. There are two queues, called the system queue and the user queue. Each token can be enqueued onto either the front or back of either the system or user queue. In addition, a token can be "recirculated," so that it takes priority over the tokens at the front of both queues. To accurately model the token processing order in Monsoon, the queueing system must account for the latency of the Monsoon pipeline. That is, a token that is enqueued should not be dequeued for at least eight cycles. This compensates for the fact that the entire pipeline is simulated by MINT in one cycle.

The enqueueing command contained in an output token passed to enqueue can be one of five values:

`Recirculate` The token should be recirculated; that is, always processed exactly eight cycles later.

`Push_System` The token should be pushed onto the front of the system queue; that is, enqueued on the system queue in LIFO mode.

`Enqueue_System` The token should be enqueued onto the back of the system queue; that is, enqueued on the system queue in FIFO mode.

`Push_User` Analogous to `Push_System`, but for the user queue.

`Enqueue_User` Analogous to `Enqueue_System`, but for the user queue.

As in the hardware, if an instruction produces two tokens both specifiying `Recirculate`, one is lost.

The data structure for this queueing system is an 11-tuple, whose components are as follows:

`sys_array` System queue token array.

`sys_head` Index into `sys_array` of next empty slot for FIFO enqueueing. After enqueueing, `sys_head` is incremented.

`sys_tail` Index into `sys_array` of next full slot for dequeueing; also one greater than next empty slot for LIFO enqueueing. After dequeueing, `sys_tail` is incremented; before LIFO enqueueing, `sys_tail` is decremented.

`sys_count` Number of tokens in system queue.

`user_array` Like `sys_array`, but for the user queue.

`user_head` Like `sys_head`, but for the user queue.

`user_tail` Like `sys_tail`, but for the user queue.

`user_count` Like `sys_count`, but for the user queue.

`pipe_array` Eight-element token array for simulating the pipeline latency.

`pipe_index` Index into `pipe_array` of next token to process.

`recirc_buffer` Holds a token enqueued in `Recirculate` mode after the call to `enqueue` but before the next call to `dequeue`; or is the value `NoToken` if no token was enqueued in `recirculate` mode.

In Id:

```
typesyn queueing_system = vector tok, N, N, N,
                          vector tok, N, N, N,
                          vector tok, N, outtok
```

You might expect that the tokens emitted by `pipe` would be entered into the eight-stage delay line, and that tokens emerging from the delay line would be subject to enqueueing or recirculation. As a practical matter, the program is simpler if the tokens enter the queue first, and go through the delay line after being dequeued. The reader should convince himself that the two schemes are entirely equivalent.

When a token is enqueued, it is either entered into the user or system queue, or placed into a one-token recirculate buffer, according to the enqueueing command. When a token is dequeued, it is taken from the head of the delay line, the delay line is shifted, and a new token entered into the delay line. The token entered into the delay line is, in order of decending priority, from the recirulate buffer, the system queue, or the user queue. If no token is available from any of these sources, a "bubble" token is entered into the delay line. Rather than actually shift the contents of pipe_array, a pointer into it is advanced modulo eight.

For simplicity, the code given below does not attempt to deal with overflow of the token queue (which should cause the processor to halt).

Enqueueing:

```
def enqueue outtok qsys =
  {case outtok of
     NoToken = qsys
   | (YesToken tok how2q) =
       {case how2q of
          Recirculate    = store_into_recirc_buffer outtok qsys
        | Push_System    = push_system_queue tok qsys
        | Enqueue_System = enqueue_system_queue tok qsys
        | Push_User      = push_user_queue tok qsys
        | Enqueue_User   = enqueue_user_queue tok qsys}};

def store_into_recirc_buffer
    outtok
    (sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, user_tail, user_count,
     pipe_array, pipe_index, recirc_buffer) =
  sys_array, sys_head, sys_tail, sys_count,
  user_array, user_head, user_tail, user_count,
  pipe_array, pipe_index, outtok;

def push_system_queue
    tok
    (sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, user_tail, user_count,
     pipe_array, pipe_index, recirc_buffer) =
  {next_sys_tail = mod (sys_tail - 1) Token_Queue_Size;
   in
     (append sys_array next_sys_tail tok),
     sys_head,
     next_sys_tail,
     sys_count + 1,
     user_array, user_head, user_tail, user_count,
     pipe_array, pipe_index, recirc_buffer};

def enqueue_system
    tok
    (sys_array, sys_head, sys_tail, sys_count,
```

```
    user_array, user_head, user_tail, user_count,
    pipe_array, pipe_index, recirc_buffer) =
  (append sys_array sys_head tok),
  mod (sys_head + 1) Token_Queue_Size,
  sys_tail,
  sys_count + 1,
  user_array, user_head, user_tail, user_count,
  pipe_array, pipe_index, recirc_buffer;

def push_user
    tok
    (sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, user_tail, user_count,
     pipe_array, pipe_index, recirc_buffer) =
  {next_user_tail = mod (user_tail - 1) Token_Queue_Size;
   in
     sys_array, sys_head, sys_tail, sys_count,
     (append user_array next_user_tail tok),
     user_head,
     next_user_tail,
     user_count + 1,
     pipe_array, pipe_index, recirc_buffer};

def enqueue_user
    tok
    (sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, user_tail, user_count,
     pipe_array, pipe_index, recirc_buffer) =
  sys_array, sys_head, sys_tail, sys_count,
  (append user_array user_head tok),
  mod (user_head + 1) Token_Queue_Size,
  user_tail,
  user_count + 1,
  pipe_array, pipe_index, recirc_buffer;
```

Dequeueing:

```
def dequeue qsys =
  {tok = current_pipe_token qsys;
   in
     if (get_recirc_buffer qsys)~=NoToken then
       tok, pop_recirc_buffer_to_pipe qsys
     else if (sys_queue_has_token qsys) then
       tok, pop_sys_queue_to_pipe qsys
     else if (user_queue_has_token qsys) then
       tok, pop_user_queue_to_pipe qsys
     else
       tok, pop_nothing_to_pipe qsys};
```

```
def get_recirc_buffer (sys_array, sys_head, sys_tail, sys_count,
                       user_array, user_head, user_tail, user_count,
                       pipe_array, pipe_index, recirc_buffer) =
  recirc_buffer;

def sys_queue_has_token (sys_array, sys_head, sys_tail, sys_count,
                         user_array, user_head, user_tail, user_count,
                         pipe_array, pipe_index, recirc_buffer) =
  sys_count > 0;

def user_queue_has_token (sys_array, sys_head, sys_tail, sys_count,
                          user_array, user_head, user_tail, user_count,
                          pipe_array, pipe_index, recirc_buffer) =
  user_count > 0;

def pop_recirc_buffer_to_pipe (sys_array, sys_head, sys_tail, sys_count,
                               user_array, user_head, user_tail, user_count,
                               pipe_array, pipe_index, recirc_buffer) =
  {YesToken tok _ = recirc_buffer;
   in
     sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, user_tail, user_count,
     append pipe_array pipe_index tok,
     mod (pipe_index + 1) 8,
     NoToken};

def pop_sys_queue_to_pipe (sys_array, sys_head, sys_tail, sys_count,
                           user_array, user_head, user_tail, user_count,
                           pipe_array, pipe_index, recirc_buffer) =
  {tok = sys_array[sys_tail];
   in
     sys_array, sys_head, mod (sys_tail+1) Token_Queue_Size, sys_count-1,
     user_array, user_head, user_tail, user_count,
     append pipe_array pipe_index tok,
     mod (pipe_index + 1) 8,
     NoToken};

def pop_user_queue_to_pipe (sys_array, sys_head, sys_tail, sys_count,
                            user_array, user_head, user_tail, user_count,
                            pipe_array, pipe_index, recirc_buffer) =
  {tok = user_array[user_tail];
   in
     sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, mod (user_tail+1) Token_Queue_Size, user_count-1,
     append pipe_array pipe_index tok,
     mod (pipe_index + 1) 8,
```

```
    NoToken};

def pop_nothing_to_pipe (sys_array, sys_head, sys_tail, sys_count,
                         user_array, user_head, user_tail, user_count,
                         pipe_array, pipe_index, recirc_buffer) =
  {tok = The_Bubble_Token;
   in
     sys_array, sys_head, sys_tail, sys_count,
     user_array, user_head, user_tail, user_count,
     append pipe_array pipe_index tok,
     mod (pipe_index + 1) 8,
     NoToken};
```

To verify that there is nothing to do, we must make sure that both queues are empty, that there are only bubbles in the delay line, and there is no token in the recirculate buffer.

```
def something_to_do? qsys =
  {sys_array, sys_head, sys_tail, sys_count,
   user_array, user_head, user_tail, user_count,
   pipe_array, pipe_index, recirc_buffer =
     qsys;
   in
     (sys_count > 0) or
     (user_count > 0) or
     (pipe_array_not_empty? qsys) or
     (recirc_buffer ~= NoToken)};

def pipe_array_not_empty? qsys =
  {sys_array, sys_head, sys_tail, sys_count,
   user_array, user_head, user_tail, user_count,
   pipe_array, pipe_index, recirc_buffer =
     qsys;
   not_empty? = false;
   in
     {for i <- 0 to 7 do
        next not_empty? = not_empty? or (pipe_array[i] ~= The_Bubble_Token)
      finally empty?}};
```

9 Statistics

Statistics are collected in MINT in much the same way that they are in Monsoon. MINT has a bank of *statistics registers*, an array of numbers. Each opcode may modify these statistics registers as it pleases, typically by incrementing one or more of them. For example, all floating point arithmetic opcodes may increment register 7 in the event that they actually perform arithmetic, thus collecting statistics on the number of floating point operations.

MINT can gather more detailed statistics than the Monsoon hardware for two reasons: MINT has no limit as to the number of registers, and an opcode can make arbitrary modifications to them (in Monsoon, an opcode is limited to an increment of a single register).

The modification to MINT to support statistics is a simple one: the statistics registers are simply passed around the top-level loop as another kind of machine state.

```
%% stats argument was added.
def mint im dm stats qsys =
  {while not (something_to_do? qsys) do

     tok, post_dq_qsys = dequeue qsys;
     Token tag_type tag_value data_type data_value = tok;
     Tag port ip map pe fp = tag_value;

     inst = instruction_fetch im ip;

     %% stats argument and result to pipe is new.
     outtok1, outtok2, next dm, next stats = pipe inst tok dm stats;

     post_tok1_qsys = enqueue outtok1 post_dm_qsys;
     post_tok2_qsys = enqueue outtok2 post_tok1_qsys;
     next qsys = post_tok2_qsys;

  finally dm, qsys, stats};
```

In the program above, only `pipe` can modify the statistics registers, but it would be a simple matter to allow `instruction_fetch` and the token queue functions to do so as well. The actual MINT program has this freedom.

The sort of statistics collected by the above program are fairly uninteresting: when the program finishes you have the final values of the statistics registers, which are grand totals of various events. To obtain things like parallelism profiles, it is necessary to partition the instructions executed into *timesteps*. The values of the statistics registers at the end of each timestep, when plotted against the timestep, yield profiles. The top level of MINT, then, becomes a doubly-nested loop. The inner loop executes tokens until the end of the timestep is reached, while the outer loop transfers statistics registers into profiles, and continues until there is nothing left to do.

The partitioning of machine cycles into timesteps is controlled by the queueing system. In addition to `enqueue` and `dequeue`, there are now three more functions provided by the queueing system:

```
typeof something_to_do? = queueing_system -> bool;
typeof timestep_finished? = queueing_system -> bool;
typeof advance_timestep = queueing_system -> queueing_system;
```

As before, `something_to_do?` returns true if there are any tokens in the queueing system at all. `Timestep_finished?` returns true if there are no more tokens left for the current timestep. `Advance_timestep` prepares a queueing system for the next timestep.

The inner loop of MINT is the same as above, except that the line

```
{while not (something_to_do? qsys) do
```

is replaced by the line

```
{while not (timestep_finished? qsys) do
```

The outer loop, which gathers the statistics at the end of each timestep and accumulates profiles, is a separate program called the *execution manager*. It looks like this:

```
def execution_manager im dm qsys =
  {profiles = 2d_I_array ((0,N_stats-1), (0,Max_timestep-1));
   timestep = 0;
   in

     %% Main loop:
     {while something_to_do? qsys do

       %% Start the next timestep with all zeros in the stats.
       in_stats = make_zeroed_1d_array (0,N_stats-1);

       %% Execute tokens until the timestep ends.
       next dm, post_timestep_qsys, out_stats = mint im dm in_stats qsys;

       %% Transfer the statistics to the profiles.
       {for i <- 0 to N_stats-1 do
          profiles[timestep, i] = out_stats[i]};

       next timestep = timestep + 1;

       %% Prepare the queueing system for the next timestep.
       next qsys = advance_timestep post_timestep_qsys;
      in
        dm, profiles, qsys}};
```

The profiles generated from a given Monsoon program will obviously depend heavily on the queueing system used. The choice of queueing system controls what in TTDA GITA was called the "statistics mode." In the next few sections we describe some queueing systems for gathering particular kinds of statistics.

9.1 Grand Total Mode

The simplest statistics mode is one where the entire program is executed in one timestep. The only statistics you get out of this mode are grand totals.

The code for the Grand Total Mode queueing system is the simple LIFO queueing system described in Section 8.1, with the following additional definitions:

```
def timestep_finished? qsys =
  something_to_do? qsys;

def advance_timestep qsys =
  qsys;
```

Note that `advance_timestep` is a no-op, since there will only be one timestep!

9.2 Idealized Mode

In idealized mode, there is a single FIFO queue, and a mark which separates the tokens to be consumed during the current timestep, and the tokens enqueued for execution in the next timestep. Within a given timestep, only tokens before the mark are consumed, and all tokens produced are enqueued after the mark.

The queueing system data structure is a four tuple, whose components are as follows:

`tok_array` An array holding the tokens.

`head` The index into `tok_array` of the next empty slot in which to place a token.

`mark` The index into `tok_array` of the first slot into which new tokens were put during the current timestep; the index which is one greater than the index of the last token to process in this timestep.

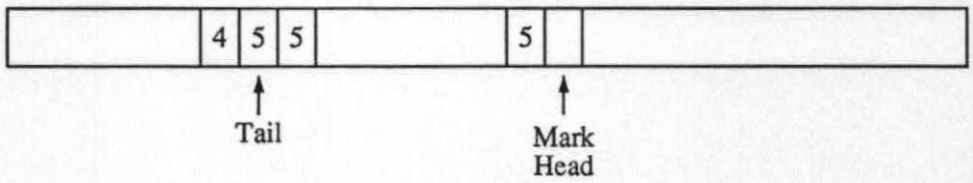
`tail` The index into `tok_array` of the next token to process.

The various configurations of `head`, `mark`, and `tail` are illustrated in Figure 3.

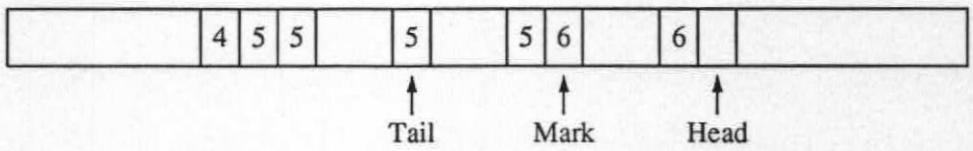
The code for this queueing system is as follows. For simplicity, the issues of wrapping around the queue indices are ignored.

```
typesyn queueing_system = (vector tok), N, N, N;

def enqueue outtok qsys =
  {case outtok of
      NoToken = qsys
    | (YesToken tok how_to_queue) =
       {tok_array, head, mark, tail = qsys
        in
          (append tok_array head tok), head + 1, mark, tail}};

def dequeue qsys =
  {tok_array, head, mark, tail = qsys;
   tok = tok_array[tail];
   new_qsys = tok_array, head, mark, tail + 1;
   in
     tok, new_qsys};

def timestep_finished? qsys =
  {tok_array, head, mark, tail = qsys;
   in
     tail == mark};

def advance_timestep qsys =
  {tok_array, head, mark, tail = qsys;
   in
     tok_array, head, head, tail};
```

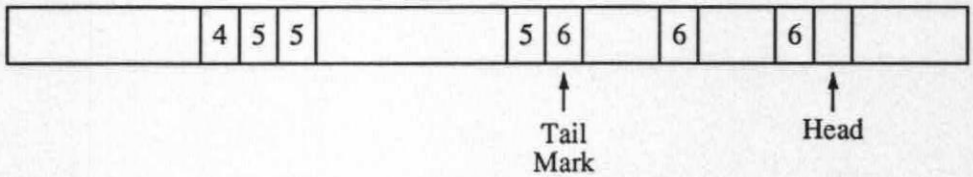
Before dequeueing first token of timestep 5



During timestep 5



After processing last token of timestep 5



After advancing timestep; before dequeueing first token of timestep 6

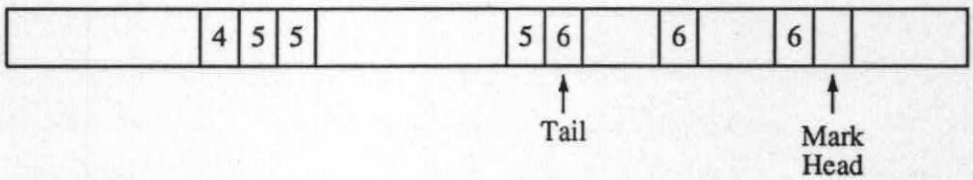


Figure 3: Queue Pointers for the Idealized Mode Queueing System

```
def something_to_do? qsys =
  {tok_array, head, mark, tail = qsys;
   in
     tail ~= head};
```

9.3 Idealized Finite Processor Mode

Idealized Finite Processor mode is a variation on idealized mode which limits the number of tokens processed in a timestep to a maximum, called `N_proc`. This is a simple modification to the idealized queueing system above: we simply add a count which says how many tokens have been processed in the current timestep, and test this to see if the timestep has finished.

```
typesyn queueing_system = (vector tok), N, N, N, N;
```

```
def enqueue outtok qsys =
  {case outtok of
      NoToken = qsys
    | (YesToken tok how_to_queue) =
        {tok_array, head, mark, tail, count = qsys
         in
           (append tok_array head tok), head + 1, mark, tail, count}};

def dequeue qsys =
  {tok_array, head, mark, tail, count = qsys;
   tok = tok_array[tail];
   new_qsys = tok_array, head, mark, tail + 1, count + 1;
   in
     tok, new_qsys};

def timestep_finished? qsys =
  {tok_array, head, mark, tail, count = qsys;
   in
     (tail == mark) or (count == N_proc)};

def advance_timestep qsys =
  {tok_array, head, mark, tail, count = qsys;
   in
     tok_array, head, head, tail, 0};

def something_to_do? qsys =
  {tok_array, head, mark, tail, count = qsys;
   in
     tail ~= head};
```

9.4 Idealized Finite Latency Mode

Idealized Finite Latency mode is like the idealized modes described above, except that each token gets to specify a "latency" value l, $l > 0$, as part of its enqueueing command. A token with latency l will not be a candidate for execution until at least l timesteps after it is enqueued.

We will only sketch the implementation of finite latency mode. Assume that the maximum possible l is L. Then the queueing system consists of a main FIFO queue Q_0 plus L additional queues Q_1 through Q_L. All tokens are dequeued from Q_0. Each token with latency l that is produced is enqueued into queue Q_l. When the timestep is complete, Q_1 is emptied into Q_0, Q_2 is emptied into Q_1, and so forth.

This combines perfectly well with Idealized Finite Processor mode: a limit is placed on the number of tokens that may be dequeued from Q_0 in one timestep. If there are tokens remaining there at the end of the timestep, the tokens transferred from Q_1 enter behind them.

10 Emulating Multiple Processors

The "finite processor" modes discussed in the previous section only emulate a system with a fixed number of processors in a very crude sense. It is possible, however, to give MINT a queueing system which makes it emulate very precisely the behavior of a Monsoon with a given number of processors. Essentially, we must provide a set of queues for each processor, along with buffers which simulate the latency through the token network.

The code for MINT that we've already presented makes use of the pe field of tokens, but only as part of the indexing into the data memory and instruction memory arrays. Because all tokens went into the same token queue regardless of pe field, pe was nothing more than an extension of the fp and ip addresses. To emulate multiple processors, the pe field will also serve as an index into the queueing system, directing the token to the emulated queues of a particular processor.

The dequeue operation will now dequeue from some processor, chosen by the queueing system. Typically, the queueing system will round-robin among the processors, so that successive calls to dequeue will get tokens from consecutive processors. If statistics are being collected, the notion of when a timestamp ends will have something to do with when all processors have been cycled through by dequeue.

The enqueue operation will need an additional piece of data besides the token to be enqueued: it needs the PE number of the processor producing that token, or equivalently the pe field from the token that was passed to pipe to produce the token being enqueued. This is because a different amount of latency may be called for depending on whether the token stays on the same pe or goes across the network, or indeed, depending on the exact path the token is to follow.

The modification to the top level of MINT is simple: a pe argument is added to enqueue:

```
def mint im dm stats qsys =
  {while not (timestep_finished?) qsys do

    %% These lines are the same as before.
    tok, post_dq_qsys = dequeue qsys;
    Token tag_type tag_value data_type data_value = tok;
    Tag port ip map pe fp = tag_value;

    ...

    post_tok1_qsys = enqueue outtok1 pe post_dm_qsys;
    post_tok2_qsys = enqueue outtok2 pe post_tok1_qsys;
    next qsys = post_tok2_qsys;

  finally dm, qsys, stats};
```

The queueing systems for a multiple processor simulation can be quite complex. We will illustrate a rather simple one in which each processor has a single FIFO queue, and a timestep is defined such that within one timestep each PE may process up to one token. Any tokens produced during the execution of one timestep are available in the next timestep; no network latency is simulated. The results will be similar to the idealized finite-processor simulation with no latency, but with an important difference. In the idealized finite-processor case, if there were

five tokens ready and the number of processors was at least five, then all five tokens would be processed in one timestep. In the multiple processor simulation, the five tokens would all have to carry *different* PE numbers in order for them all to be executed in one timestep.

The basic idea is to have an array of queues, one per PE. Successive calls to dequeue must retrieve a token from successive PE's queues, with a timestep ending when all PE's have been polled. In addition to the array of queues, therefore, we need an index that says what PE's queue to examine in the next call to dequeue. The type of the queueing system is as follows:

```
typesyn queueing_system = (vector (vector tok, N, N, N)), N;
```

The first component is the array of queueing systems, indexed 1 through n, and the second is the index of the next queue to dequeue from. Each queue is like the idealized queue from Section 9.2: it has a token array, a head pointer, a mark, and a tail pointer, with meanings as illustrated in Figure 3.

The reason we need marks in each of the queues is fairly subtle. Suppose that we just dequeued a token for PE 4, and it caused tokens to be enqueued for PE 2 and PE 6. Now the token for PE 2 is no problem, since we'll not try to dequeue from PE 2 until the next timestep. But, as we continue this timestep we'll dequeue from PE 5, then from PE 6. The marks insure that we don't inadvertantly process the token in PE 6's queue that was produced in the current timestep.

With that in mind, here is the code for the queueing system. First, the code to enqueue. This is straightforward: we just enqueue into the appropriate PE's queue, as indicated by the PE field on the token. The enqueue_on_pe routine is just like the enqueue routine from Section 9.2.

```
def enqueue outtok, qsys =
  {case outtok of
      NoToken = qsys
    | (YesToken tok how_to_queue) =
        really_enqueue tok qsys};

def really_enqueue tok qsys =
  {Token tag_type tag_value data_type data_value = tok;
   Tag port ip map pe fp = tag_value;
   qbank, dequeue_ptr = qsys;
   new_qbank = enqueue_on_pe tok pe qbank;
   new_qsys = new_qbank, dequeue_ptr;
   in
     new_qsys};

def enqueue_on_pe tok pe bank =
  {tok_array, head, mark, tail = qbank[pe];
   new_pe_queue = (append tok_array head tok), head + 1, mark, tail;
   in
     append qbank pe new_pe_queue};
```

Dequeueing is slightly more complicated. The second component of the queueing system says which PE we are to dequeue from next. But it could be that that PE does not have a token ready for the current timestep. What we really want, therefore, is for dequeue to dequeue a

token from the queue of the next highest numbered PE that has a token ready. The function `find_next_active_pe` returns the index of the desired PE, or zero if there are no more PE's left with tokens.

```
def find_next_active_pe qbank from =
  if from > N_pes then
    0
  else
    {tok_array, head, mark, tail = qbank[from];
     in
       if tail == mark then
         find_next_active_pe qbank (from + 1);
       else
         from};
```

Given that, `dequeue` is straightforward. It is similar to the idealized mode `dequeue`, except that it also advances the index of the next PE to poll. (The definition of `timestep_finished?` will insure that the call to `find_next_active_pe` within `dequeue` will never return zero.)

```
def dequeue qsys =
  {qbank, dequeue_ptr = qsys;
   pe = find_next_active_pe qbank dequeue_ptr;
   tok, new_qbank = dequeue_from_pe pe qbank;
   new_qsys = new_qbank, (pe + 1);
   in
     tok, new_qsys};

def dequeue_from_pe pe qbank =
  {tok_array, head, mark, tail = qbank[dequeue_ptr];
   tok = tok_array[tail];
   new_pe_queue = tok_array, head, mark, tail + 1;
   in
     tok, (append qbank pe new_pe_queue)};
```

A timestep is finished if there are no more higher numbered PE's with tokens ready:

```
def timestep_finished? qsys =
  {qbank, dequeue_ptr = qsys;
   in
     find_next_active_pe qbank dequeue_ptr == 0};
```

To prepare for the next timestep, we advance the marks in all queues, and reset the PE index to one.

```
def advance_timestep qsys =
  {qbank, dequeue_ptr = qsys;
   new_qbank =
     {vector (1,N_pes)
       | [i] = {tok_array, head, mark, tail = qbank[i];
                in
                  tok_array, head, head, tail};
         || i <- 1 to N_pes};
   in
     new_qbank, 1};
```

Finally, the program is finished if none of the queues has a token ready.

```
def something_to_do? qsys =
  {qbank, dequeue_ptr = qsys;
   in
     find_next_active_pe qbank 1 ~= 0};
```

The use of `find_next_active_pe` could be eliminated if we simply dequeued a "bubble" token, as in Section 8.2. This would result in more calls to `pipe`, though it would have the advantage of allowing statistics to be collected for idle cycles.

Notice that within the multiple processor framework, a variety of queueing systems may be emulated within each PE. Furthermore, there could be a global set of queues to simulate network latency. We could enhance the system given above, for example, by having all inter-PE tokens enter a bucket brigade of queues to simulate network latency. Or we could provide both user and system queues for each PE, along with a delay line to simulate pipeline latency. Both of these enhancements could be combined, too.

Perhaps the most sophisticated multiple-processor queueing system is one that accurately models the behavior of a real multiple-processor Monsoon configuration, including proper modeling of network latency and blocking effects for Monsoon's shuffle-exchange network. The queueing system would have a number queues to simulate the latency through the network, as well as a system queue, user queue, and pipeline delay line for each PE. The `advance_timestep` procedure would be quite complex, as it would advance the network queues, entering any tokens emerging from the network into the PE queues, and then enter network-bound tokens produced in the previous timestep into the network. Proper modeling of the various priorities of the system queue, user queue, and incoming network buffer introduces even more complexity, as does proper emulation of blocking due to network buffer overflow. The construction of a fully accurate queueing system emulating a multiple-processor Monsoon is a large research project in itself.

The complexity of totally accurate queueing systems is the prime reason why the idealized finite-processor, finite-latency systems are so attractive. A good research topic is to thoroughly evaluate the situations in which the idealized systems provide useful approximations, and where they break down.

11 Emulating Registers

Monsoon has a eight sets of registers connected to the processing pipeline; each token entering the pipeline can potentially read or write registers in one of these sets, the sets being rotated

with every token. If token A produces another token B with a queueing mode of "recirculation," then token B will access the same set of registers as did token A, because the pipeline depth is also eight.

To execute code properly when the register feature is used, MINT must:

1. Include data structures to simulate the registers.
2. Insure that a token enqueued in "recirculate" mode, when subsequently dequeued, sees the same registers as did the token that produced it.

These two requirements interact: the longer the queueing system waits to process a recirculating token, the more storage will be needed for registers. For example, if a queueing system is such that when a recirculating token is enqueued it pops out from the very next call to `dequeue`, then only one set of registers is needed. If, on the other hand, a recirculating token is dequeued on the eighth call after it was enqueued, eight sets of registers are required, because each of those eight tokens might represent a sequential thread.

The top-level function `mint` needs to pass around a data structure representing the register sets. Furthermore, if a queueing system requires multiple sets of registers, the function `dequeue` must return an additional value indicating which set is to be used for that token. The modified version of `mint` is as follows:

```
%% registers argument was added.
def mint im dm stats registers qsys =
  {while not (something_to_do? qsys) do

    %% register_index result was added.
    tok, register_index, post_dq_qsys = dequeue qsys;
    Token tag_type tag_value data_type data_value = tok;
    Tag port ip map pe fp = tag_value;

    inst = instruction_fetch im ip;

    %% registers argument and result were added,
    %% as was register_index argument.
    outtok1, outtok2, next dm, next stats, next registers =
      pipe inst tok dm stats registers register_index;

    post_tok1_qsys = enqueue outtok1 post_dm_qsys;
    post_tok2_qsys = enqueue outtok2 post_tok1_qsys;
    next qsys = post_tok2_qsys;

  finally dm, qsys, stats};
```

Modifying the Monsoon queueing system from Section 8.2 to handle registers is quite simple. The `registers` argument to `pipe` is an array of eight register sets, and the `register_index` result returned from `dequeue` is simply the `pipe_index` value already maintained by the queueing system.

Modifying the idealized mode queueing system to handle registers is quite a bit harder. The most appropriate interpretation of sequential thread code in the idealized simulation would be to treat recirculating tokens no differently from ordinary tokens: a recirculating token generated

in timestep i would be processed in timestep $i+1$. The difficulty, of course, is that an arbitrary number of recirculating tokens might be produced in any given timestep, requiring an arbitrarily large amount of storage for their registers. Assuming the storage management issues could be worked out, however, the remaining details are not particulary difficult, and are left as an exercise for the reader (hint: every recirculating token generated in a given timestep would carry a unique thread number, used as the register index when subsequently dequeued).

Another possible treatment of register code in the idealized setting is to process all tokens belonging to a sequential thread in the *same* timestep. That is, if a recirculating token is produced then it will be immediately dequeued and processed, without advancing the timestep. This does not assign terribly meaningful parallelism statistics to thread code; in fact, it does exactly the wrong thing since it says that all instructions belonging to a sequential thread can go on in parallel! Nevertheless, it is a way to get an approximation of idealized statistics in the case where thread code is very infrequent, without encountering the difficulties described in the previous paragraph. This treatment of threads, incidentally, is precisely how the Monsoon hardware will treat them if the "alternating queues" method of obtaining idealized statistics is used.

The code for this simplified treatment of threads in the idealized setting is given below. The queueing system is just like that of Section 9.2, but with an extra component that holds the current recirculating token, or `NoToken` if none was generated. As in the hardware, if an instruction tries to produce two recirculating tokens at once, one is lost.

```
typesyn qsys = vector tok, N, N, N, outtok;

def enqueue outtok qsys =
  {case outtok of
      NoToken = qsys
    | (YesToken tok how_to_queue) =
       {tok_array, head, mark, tail, recirc_buf = qsys
        in
          {case how_to_queue of
             Recirc =
               tok_array, head, mark, tail, outtok
           | Normal =
               (append tok_array head tok), head + 1, mark, tail, recirc_buf}}};

def dequeue qsys =
  {tok_array, head, mark, tail, recirc_buf = qsys;
   in
     {case recirc_buf of
        YesToken tok _ =
          tok, (tok_array, head, mark, tail, NoToken)
      | NoToken =
          {tok = tok_array[tail];
           new_qsys = tok_array, head, mark, tail + 1, recirc_buf;
           in
             tok, new_qsys}}};

def timestep_finished? qsys =
```

```
  {tok_array, head, mark, tail, recirc_buf = qsys;
   in
     (recirc_buf == NoToken) and (tail == mark)};

def something_to_do? qsys =
  {tok_array, head, mark, tail, recirc_buf = qsys;
   in
     %% Recirculation buffer is always empty when something_to_do? is called.
     tail ~= head};

def advance_timestep qsys =
  {tok_array, head, mark, tail, recirc_buf = qsys;
   in
     tok_array, mark, head, head, recirc_buf};
```

12 General Observations

There are several aspects of MINT which make it an attractive research tool.

- It cleanly separates the computational aspects of the emulation (`pipe`) from the aspects connected with scheduling (the queueing system). The computational side is only concerned with producing output tokens from input tokens, while the scheduling side is only concerned with the order in which tokens are processed, and the meaning of a timestep for the purposes of statistics gathering. Each side is understandable in isolation.

- It provides for both precise emulation of the Monsoon architecture, while at the same time allowing for more abstract emulation through the use of fictitious opcodes and idealized queueing systems. While MINT has knowledge of bit-level representations of data types within Monsoon, these are concentrated into a small number of conversion functions. It is possible to detect whether a program makes use of these conversion functions, and therefore whether it is depending on the bit-level representations.

- It provides an interface that is compatible with the interface to the Monsoon hardware itself, so that programs which manipulate machine state (loader, execution manager, debugger) may be shared between the two.

Of course, MINT has its limitations, too. Among them are the following:

- If statistics are being gathered, all tokens belonging to one timestep must be processed contiguously and the timesteps must be consecutive. That is, MINT must first process all tokens belonging to timestep 1, then all tokens for timestep 2, *etc.* This precludes the kind of time-warping that was done in TTDA GITA, to allow tokens to be scheduled in an order that resulted in better paging performance of GITA itself.

- Gathering of statistics must be expressed in terms of manipulating statistics registers, at least if the interface to the execution manager is to be preserved.

- Pipeline conflicts are not properly emulated. Fortunately, these are rare, if they occur at all. The interested reader may consult the MINT documentation for a catalog of such conflicts.

The last two items are not terribly serious. Experience will tell if the first item is serious or not. It seems that, in principle, any notion of timestep and statistics can be molded into the framework provided by MINT. There is a great advantage in not associating timesteps with each token, as it frees the `pipe` section of MINT from having to compute appropriate timesteps for the tokens it produces. Whether there is a severe performance penalty because of paging behavior, though, remains to be seen.