1. Introduction. This program is the heart of the meta-simulator for the ultra-configurable `MMIX` pipeline: It defines the *MMIX_run* routine, which does most of the work. Another routine, *MMIX_init*, is also defined here, and so is a header file called `mmix_pipe.h`. The header file is used by the main routine and by other routines like *MMIX_config*, which are compiled separately.

Readers of this program should be familiar with the explanation of `MMIX` architecture as presented in the main program module for MMMIX.

A lot of subtle things can happen when instructions are executed in parallel. Therefore this simulator ranks among the most interesting and instructive programs in the author's experience. The author has tried his best to make everything correct ... but the chances for error are great. Anyone who discovers a bug is therefore urged to report it as soon as possible; please see `http://mmix.cs.hm.edu/bugs/` for instructions.

It sort of boggles the mind when one realizes that the present program might someday be translated by a C compiler for `MMIX` and used to simulate *itself*.

2. This high-performance prototype of `MMIX` achieves its efficiency by means of "pipelining," a technique of overlapping that is explained for the related `DLX` computer in Chapter 3 of Hennessy & Patterson's book *Computer Architecture* (second edition). Other techniques such as "dynamic scheduling" and "multiple issue," explained in Chapter 4 of that book, are used too.

One good way to visualize the procedure is to imagine that somebody has organized a high-tech car repair shop according to similar principles. There are eight independent functional units, which we can think of as eight groups of auto mechanics, each specializing in a particular task; each group has its own workspace with room to deal with one car at a time. Group F (the "fetch" group) is in charge of rounding up customers and getting them to enter the assembly-line garage in an orderly fashion. Group D (the "decode and dispatch" group) does the initial vehicle inspection and writes up an order that explains what kind of servicing is required. The vehicles go next to one of the four "execution" groups: Group X handles routine maintenance, while groups XF, XM, and XD are specialists in more complex tasks that tend to take longer. (The XF people are good at floating the points, while the XM and XD groups are experts in multilink suspensions and differentials.) When the relevant X group has finished its work, cars drive to M station, where they send or receive messages and possibly pay money to members of the "memory" group. Finally all necessary parts are installed by members of group W, the "write" group, and the car leaves the shop. Everything is tightly organized so that in most cases the cars move in synchronized fashion from station to station, at regular 100-nanocentury intervals.

In a similar way, most `MMIX` instructions can be handled in a five-stage pipeline, F–D–X–M–W, with X replaced by XF for floating-point addition or conversion, or by XM for multiplication, or by XD for division or square root. Each stage ideally takes one clock cycle, although XF, XM, and (especially) XD are slower. If the instructions enter in a suitable pattern, we might see one instruction being fetched, another being decoded, and up to four being executed, while another is accessing memory, and yet another is finishing up by writing new information into registers; all this is going on simultaneously during one clock cycle. Pipelining with eight separate stages might therefore make the machine run up to 8 times as fast as it could if each instruction were being dealt with individually and without overlap. (Well, perfect speedup turns out to be impossible, because of the shared M and W stages; the theory of knapsack programming, to be discussed in Section 7.7 of *The Art of Computer Programming*, tells us that the maximal achievable speedup is at most $8 - 1/p - 1/q - 1/r$ when XF, XM, and XD have delays bounded by p, q, and r cycles. But we can achieve a factor of more than 7 if we are very lucky.)

Consider, for example, the `ADD` instruction. This instruction enters the computer's processing unit in F stage, taking only one clock cycle if it is in the cache of instructions recently seen. Then the D stage recognizes the command as an `ADD` and acquires the current values of \$Y and \$Z; meanwhile, of course, another instruction is being fetched by F. On the next clock cycle, the X stage adds the values together. This prepares the way for the M stage to watch for overflow and to get ready for any exceptional action that might be needed with respect to the settings of special register rA. Finally, on the fifth clock cycle, the sum is either written into \$X or the trip handler for integer overflow is invoked. Although this process has taken five clock cycles (that is, 5υ), the net increase in running time has been only 1υ.

Of course congestion can occur, inside a computer as in a repair shop. For example, auto parts might not be readily available; or a car might have to sit in D station while waiting to move to XM, thereby blocking somebody else from moving from F to D. Sometimes there won't necessarily be a steady stream of customers. In such cases the employees in some parts of the shop will occasionally be idle. But we assume that they always do their jobs as fast as possible, given the sequence of customers that they encounter. With a clever person setting up appointments—translation: with a clever programmer and/or compiler arranging `MMIX` instructions—the organization can often be expected to run at nearly peak capacity.

In fact, this program is designed for experiments with many kinds of pipelines, potentially using additional functional units (such as several independent X groups), and potentially fetching, dispatching, and executing several nonconflicting instructions simultaneously. Such complications make this program more difficult than a simple pipeline simulator would be, but they also make it a lot more instructive because we can get a better understanding of the issues involved if we are required to treat them in greater generality.

3. Here's the overall structure of the present program module.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "abstime.h"
  ⟨Preprocessor definitions⟩
  ⟨Header definitions 6⟩
  ⟨Type definitions 11⟩
  ⟨Global variables 20⟩
  ⟨External variables 4⟩
  ⟨Internal prototypes 13⟩
  ⟨External prototypes 9⟩
  ⟨Subroutines 14⟩
  ⟨External routines 10⟩
```

4. The identifier **Extern** is used in MMIX-PIPE to declare variables that are accessed in other modules. Actually all appearances of '**Extern**' are defined to be blank here, but '**Extern**' will become '**extern**' in the header file.

```
#define Extern      /* blank for us, extern for them */
  format Extern extern
```

⟨External variables 4⟩ ≡

```
  Extern int verbose;      /* controls the level of diagnostic output */
```

See also sections 29, 59, 60, 66, 69, 77, 86, 87, 98, 115, 136, 150, 168, 207, 211, 214, 238, 242, 247, 284, and 349.

This code is used in sections 3 and 5.

5. The header file repeats the basic definitions and declarations.

⟨mmix-pipe.h 5⟩ ≡

```
#define Extern extern
  ⟨Header definitions 6⟩
  ⟨Type definitions 11⟩
  ⟨External variables 4⟩
  ⟨External prototypes 9⟩
```

6. Subroutines of this program are declared first with a prototype, as in ANSI C, then with an old-style C function definition. The following preprocessor commands make this work correctly with both new-style and old-style compilers.

⟨Header definitions 6⟩ ≡

```
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
```

See also sections 7, 8, 52, 57, 129, and 166.

This code is used in sections 3 and 5.

7. Some of the names that are natural for this program are in conflict with library names on at least one of the host computers in the author's tests. So we bypass the library names here.

```
⟨ Header definitions 6 ⟩ +≡
#define random my_random
#define fsqrt my_fsqrt
#define div my_div
```

8. The amount of verbosity depends on the following bit codes.

```
⟨ Header definitions 6 ⟩ +≡
#define issue_bit (1 ≪ 0) /* show control blocks when issued, deissued, committed */
#define pipe_bit (1 ≪ 1) /* show the pipeline and locks on every cycle */
#define coroutine_bit (1 ≪ 2) /* show the coroutines when started on every cycle */
#define schedule_bit (1 ≪ 3) /* show the coroutines when scheduled */
#define uninit_mem_bit (1 ≪ 4) /* complain when reading from an uninitialized chunk of memory */
#define interactive_read_bit (1 ≪ 5) /* prompt user when reading from I/O location */
#define show_spec_bit (1 ≪ 6) /* display special read/write transactions as they happen */
#define show_pred_bit (1 ≪ 7) /* display branch prediction details */
#define show_wholecache_bit (1 ≪ 8) /* display cache blocks even when their key tag is invalid */
```

9. The *MMIX_init*( ) routine should be called exactly once, after *MMIX_config*( ) has done its work but before the simulator starts to execute any programs. Then *MMIX_run*( ) can be called as often as the user likes.

The *MMIX_silent*( ) routine is a noninteractive variant of *MMIX_run*( ): It will return the value of register $g[255].l$ when executing a `TRAP 0,Halt,0` instruction.

```
⟨ External prototypes 9 ⟩ ≡
  Extern void MMIX_init ARGS((void));
  Extern void MMIX_run ARGS((int cycs, octa breakpoint));
  Extern int MMIX_silent ARGS((void));
```

See also sections 38, 161, 175, 178, 180, 209, 212, and 252.

This code is used in sections 3 and 5.

10. ⟨ External routines 10 ⟩ ≡

```
void MMIX_init()
{
  register int i, j;
  ⟨ Initialize everything 22 ⟩;
}
int MMIX_silent()
{
  octa breakpoint;
  ⟨ Local variables 12 ⟩;
  while (true) {
    ⟨ Perform one machine cycle 64 ⟩;
    if (halted) return specval(&g[255]).o.l;
  }
}
void MMIX_run(cycs, breakpoint)
    int cycs;
    octa breakpoint;
{
  ⟨ Local variables 12 ⟩;
  while (cycs) {
    if (verbose & (issue_bit | pipe_bit | coroutine_bit | schedule_bit)) printf("***␣Cycle␣%d\n", ticks.l);
    ⟨ Perform one machine cycle 64 ⟩;
    if (verbose & pipe_bit) {
      print_pipe(); print_locks();
    }
    if (breakpoint_hit ∨ halted) {
      if (breakpoint_hit) printf("Breakpoint␣instruction␣fetched␣at␣time␣%d\n", ticks.l − 1);
      if (halted) printf("Halted␣at␣time␣%d\n", ticks.l − 1);
      break;
    }
    cycs −−;
  }
}
```

See also sections 39, 162, 176, 179, 181, 210, 213, and 253.

This code is used in section 3.

11. ⟨ Type definitions 11 ⟩ ≡

```
typedef enum {
  false, true, wow
} bool;      /* slightly extended booleans */
```

See also sections 17, 23, 37, 40, 44, 68, 76, 164, 167, 206, 246, and 371.

This code is used in sections 3 and 5.

12. ⟨ Local variables 12 ⟩ ≡

```
register int i, j, m;
bool breakpoint_hit = false;
bool halted = false;
```

See also sections 124 and 258.

This code is used in section 10.

13. Error messages that abort this program are called panic messages. The macro called *confusion* will never be needed unless this program is internally inconsistent.

```
#define errprint0(f) fprintf(stderr,f)
#define errprint1(f,a) fprintf(stderr,f,a)
#define errprint2(f,a,b) fprintf(stderr,f,a,b)
#define panic(x) { errprint0("Panic: "); x; errprint0("!\n"); expire(); }
#define confusion(m) errprint1("This can't happen: %s",m)
```

⟨ Internal prototypes 13 ⟩ ≡

```
static void expire ARGS((void));
```

See also sections 18, 24, 27, 30, 32, 34, 42, 45, 55, 62, 72, 90, 92, 94, 96, 156, 158, 169, 171, 173, 182, 184, 186, 188, 190, 192, 195, 198, 200, 202, 204, 240, 250, 254, and 377.

This code is used in section 3.

14. ⟨Subroutines 14⟩ ≡

```
static void expire()    /* the last gasp before dying */
{
  if (ticks.h) errprint2("(Clock time is %dH+%d.)\n",ticks.h,ticks.l);
  else errprint1("(Clock time is %d.)\n",ticks.l);
  exit(-2);
}
```

See also sections 19, 21, 25, 28, 31, 33, 35, 43, 46, 56, 63, 73, 91, 93, 95, 97, 157, 159, 170, 172, 174, 183, 185, 187, 189, 191, 193, 196, 199, 201, 203, 205, 208, 241, 251, 255, 378, 379, 381, 384, and 387.

This code is used in section 3.

15. The data structures of this program are not precisely equivalent to logical gates that could be implemented directly in silicon; we will use data structures and algorithms appropriate to the C programming language. For example, we'll use pointers and arrays, instead of buses and ports and latches. However, the net effect of our data structures and algorithms is intended to be equivalent to the net effect of a silicon implementation. The methods used below are essentially equivalent to those used in real machines today, except that diagnostic facilities are added so that we can readily watch what is happening.

Each functional unit in the MMIX pipeline is programmed here as a coroutine in C. At every clock cycle, we will call on each active coroutine to do one phase of its operation; in terms of the repair-station analogy described in the main program, this corresponds to getting each group of auto mechanics to do one unit of operation on a car. The coroutines are performed sequentially, although a real pipeline would have them act in parallel. We will not "cheat" by letting one coroutine access a value early in its cycle that another one computes late in its cycle, unless computer hardware could "cheat" in an equivalent way.

16. Low-level routines. Where should we begin? It is tempting to start with a global view of the simulator and then to break it down into component parts. But that task is too daunting, because there are so many unknowns about what basic ingredients ought to be combined when we construct the larger components. So let us look first at the primitive operations on which the superstructure will be built. Once we have created some infrastructure, we'll be able to proceed with confidence to the larger tasks ahead.

17. This program for the 64-bit `MMIX` architecture is based on 32-bit integer arithmetic, because nearly every computer available to the author at the time of writing (1998–1999) was limited in that way. Details of the basic arithmetic appear in a separate program module called MMIX-ARITH, because the same routines are needed also for the assembler and for the non-pipelined simulator. The definition of type **tetra** should be changed, if necessary, to conform with the definitions found there.

```
⟨ Type definitions 11 ⟩ +≡
  typedef unsigned int tetra;      /* for systems conforming to the LP-64 data model */
  typedef struct {
    tetra h, l;
  } octa;      /* two tetrabytes make one octabyte */
```

18.
```
⟨ Internal prototypes 13 ⟩ +≡
  static void print_octa ARGS((octa));
```

19.
```
⟨ Subroutines 14 ⟩ +≡
  static void print_octa(o)
      octa o;
  {
    if (o.h) printf("%x%08x", o.h, o.l); else printf("%x", o.l);
  }
```

20.
```
⟨ Global variables 20 ⟩ ≡
  extern octa zero_octa;      /* zero_octa.h = zero_octa.l = 0 */
  extern octa neg_one;      /* neg_one.h = neg_one.l = -1 */
  extern octa aux;      /* auxiliary output of a subroutine */
  extern bool overflow;      /* set by certain subroutines for signed arithmetic */
  extern int exceptions;      /* bits set by floating point operations */
  extern int cur_round;      /* the current rounding mode */
```

See also sections 36, 41, 48, 50, 51, 53, 54, 65, 70, 78, 83, 88, 99, 107, 127, 148, 154, 194, 230, 235, 248, 285, 303, 305, 315, 374, 376, and 388.

This code is used in section 3.

21. Most of the subroutines in MMIX-ARITH return an octabyte as a function of two octabytes; for example, $oplus(y,z)$ returns the sum of octabytes y and z. Multiplication returns the high half of a product in the global variable aux; division returns the remainder in aux.

```
⟨ Subroutines 14 ⟩ +≡
  extern octa oplus ARGS((octa y,octa z));      /* unsigned y + z */
  extern octa ominus ARGS((octa y,octa z));      /* unsigned y − z */
  extern octa incr ARGS((octa y,int delta));      /* unsigned y + δ (δ is signed) */
  extern octa oand ARGS((octa y,octa z));      /* y ∧ z */
  extern octa oandn ARGS((octa y,octa z));      /* y ∧ z̄ */
  extern octa shift_left ARGS((octa y,int s));      /* y ≪ s, 0 ≤ s ≤ 64 */
  extern octa shift_right ARGS((octa y,int s,int u));      /* y ≫ s, signed if ¬u */
  extern octa omult ARGS((octa y,octa z));      /* unsigned (aux, x) = y × z */
  extern octa signed_omult ARGS((octa y,octa z));      /* signed x = y × z, setting overflow */
  extern octa odiv ARGS((octa x,octa y,octa z));      /* unsigned (x, y)/z; aux = (x, y) mod z */
  extern octa signed_odiv ARGS((octa y,octa z));      /* signed y/z, when z ≠ 0; aux = y mod z */
  extern int count_bits ARGS((tetra z));      /* x = ν(z) */
  extern tetra byte_diff ARGS((tetra y,tetra z));      /* half of BDIF */
  extern tetra wyde_diff ARGS((tetra y,tetra z));      /* half of WDIF */
  extern octa bool_mult ARGS((octa y,octa z,bool xor));      /* MOR or MXOR */
  extern octa load_sf ARGS((tetra z));      /* load short float */
  extern tetra store_sf ARGS((octa x));      /* store short float */
  extern octa fplus ARGS((octa y,octa z));      /* floating point x = y ⊕ z */
  extern octa fmult ARGS((octa y,octa z));      /* floating point x = y ⊗ z */
  extern octa fdivide ARGS((octa y,octa z));      /* floating point x = y ⊘ z */
  extern octa froot ARGS((octa,int));      /* floating point x = √z */
  extern octa fremstep ARGS((octa y,octa z,int delta));      /* floating point x rem z = y rem z */
  extern octa fintegerize ARGS((octa z,int mode));      /* floating point x = round(z) */
  extern int fcomp ARGS((octa y,octa z));      /* −1, 0, 1, or 2 if y < z, y = z, y > z, y ∥ z */
  extern int fepscomp ARGS((octa y,octa z,octa eps,int sim));
    /* x = sim? [y ∼ z (ϵ)] : [y ≈ z (ϵ)] */
  extern octa floatit ARGS((octa z,int mode,int unsgnd,int shrt));      /* fix to float */
  extern octa fixit ARGS((octa z,int mode));      /* float to fix */
```

22. We had better check that our 32-bit assumption holds.

```
⟨ Initialize everything 22 ⟩ ≡
  if (shift_left(neg_one,1).h ≠ #ffffffff)
    panic(errprint0("Incorrect␣implementation␣of␣type␣tetra"));
```

See also sections 26, 61, 71, 79, 89, 116, 128, 153, 231, 236, 249, and 286.

This code is used in section 10.

23. Coroutines. As stated earlier, this program can be regarded as a system of interacting coroutines. Coroutines—sometimes called threads—are more or less independent processes that share and pass data and control back and forth. They correspond to the individual workers in an organization.

We don't need the full power of recursive coroutines, in which new threads are spawned dynamically and have independent stacks for computation; we are, after all, simulating a fixed piece of hardware. The total number of coroutines we deal with is established once and for all by the *MMIX_config* routine, and each coroutine has a fixed amount of local data.

The simulation operates one clock tick at a time, by executing all coroutines scheduled for time t before advancing to time $t+1$. The coroutines at time t may decide to become dormant or they may reschedule themselves and/or other coroutines for future times.

Each coroutine has a symbolic *name* for diagnostic purposes (e.g., `ALU1`); a nonnegative *stage* number (e.g., 2 for the second stage of a pipeline); a pointer to the next coroutine scheduled at the same time (or Λ if the coroutine is unscheduled); a pointer to a lock variable (or Λ if no lock is currently relevant); and a reference to a control block containing the data to be processed.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct coroutine_struct {
    char *name;      /* symbolic identification of a coroutine */
    int stage;      /* its rank */
    struct coroutine_struct *next;      /* its successor */
    struct coroutine_struct **lockloc;      /* what it might be locking */
    struct control_struct *ctl;      /* its data */
  } coroutine;
```

24. ⟨ Internal prototypes 13 ⟩ +≡

```
  static void print_coroutine_id ARGS((coroutine *));
  static void errprint_coroutine_id ARGS((coroutine *));
```

25. ⟨ Subroutines 14 ⟩ +≡

```
  static void print_coroutine_id(c)
      coroutine *c;
  {
    if (c) printf("%s:%d", c→name, c→stage);
    else printf("??");
  }
  static void errprint_coroutine_id(c)
      coroutine *c;
  {
    if (c) errprint2("%s:%d", c→name, c→stage);
    else errprint0("??");
  }
```

26. Coroutine control is masterminded by a ring of queues, one each for times t, $t+1$, ..., $t+ring\_size-1$, when t is the current clock time.

All scheduling is first-come-first-served, except that coroutines with higher *stage* numbers have priority. We want to process the later stages of a pipeline first, in this sequential implementation, for the same reason that a car must drive from M station into W station before another car can enter M station.

Each queue is a circular list of **coroutine** nodes, linked together by their *next* fields. A list head h with $stage = max\_stage$ comes at the end and the beginning of the queue. (All *stage* numbers of legitimate coroutines are less than max_stage.) The queued items are $h\to next$, $h\to next\to next$, etc., from back to front, and we have $c\to stage \le c\to next\to stage$ unless $c = h$.

Initially all queues are empty.

```
⟨ Initialize everything 22 ⟩ +≡
  { register coroutine *p;
    for (p = ring; p < ring + ring_size; p++) p->next = p;
  }
```

27. To schedule a coroutine c with positive delay $d < ring\_size$, we call $schedule(c, d, s)$. (The s parameter is used only if scheduling is being logged; it does not affect the computation, but we will generally set s to the state at which the scheduled coroutine will begin.)

```
⟨ Internal prototypes 13 ⟩ +≡
  static void schedule ARGS((coroutine *, int, int));
```

28.
```
⟨ Subroutines 14 ⟩ +≡
  static void schedule(c, d, s)
      coroutine *c;
      int d, s;
  {
    register int tt = (cur_time + d) % ring_size;
    register coroutine *p = &ring[tt];      /* start at the list head */
    if (d ≤ 0 ∨ d ≥ ring_size)      /* do a sanity check */
      panic(confusion("Scheduling "); errprint_coroutine_id(c); errprint1("with delay %d", d));
    while (p->next->stage < c->stage) p = p->next;
    c->next = p->next;
    p->next = c;
    if (verbose & schedule_bit) {
      printf("scheduling "); print_coroutine_id(c);
      printf("at time %d, state %d\n", ticks.l + d, s);
    }
  }
```

29.
```
⟨ External variables 4 ⟩ +≡
  Extern int ring_size;      /* set by MMIX_config, must be sufficiently large */
  Extern coroutine *ring;
  Extern int cur_time;
```

30. The all-important *ctl* field of a coroutine, which contains the data being manipulated, will be explained below. One of its key components is the *state* field, which helps to specify the next actions the coroutine will perform. When we schedule a coroutine for a new task, we often want it to begin in state 0.

```
⟨ Internal prototypes 13 ⟩ +≡
  static void startup ARGS((coroutine *, int));
```

31. ⟨ Subroutines 14 ⟩ +≡

```
static void startup(c, d)
    coroutine *c;
    int d;
{
  c->ctl->state = 0;
  schedule(c, d, 0);
}
```

32. The following routine removes a coroutine from whatever queue it's in. The case $c\rightarrow next = c$ is also permitted; such a self-loop can occur when a coroutine goes to sleep and expects to be awakened (that is, scheduled) by another coroutine. Sleeping coroutines have important data in their *ctl* field; they are therefore quite different from unscheduled or "unemployed" coroutines, which have $c\rightarrow next = \Lambda$. An unemployed coroutine is not assumed to have any valid data in its *ctl* field.

⟨ Internal prototypes 13 ⟩ +≡

```
static void unschedule ARGS((coroutine *));
```

33. ⟨ Subroutines 14 ⟩ +≡

```
static void unschedule(c)
    coroutine *c;
{ register coroutine *p;
  if (c->next) {
    for (p = c; p->next != c; p = p->next) ;
    p->next = c->next;
    c->next = Λ;
    if (verbose & schedule_bit) {
      printf("␣unscheduling␣"); print_coroutine_id(c); printf("\n");
    }
  }
}
```

34. When it is time to process all coroutines that have queued up for a particular time t, we empty the queue called $ring[t]$ and link its items in the opposite order (from front to back). The following subroutine uses the well known algorithm discussed in exercise 2.2.3–7 of *The Art of Computer Programming*.

⟨ Internal prototypes 13 ⟩ +≡

```
static coroutine *queuelist ARGS((int));
```

35. ⟨ Subroutines 14 ⟩ +≡

```
static coroutine *queuelist(t)
    int t;
{ register coroutine *p, *q = &sentinel, *r;
  for (p = ring[t].next; p != &ring[t]; p = r) {
    r = p->next;
    p->next = q;
    q = p;
  }
  ring[t].next = &ring[t];
  sentinel.next = q;
  return q;
}
```

36. ⟨ Global variables 20 ⟩ +≡
coroutine *sentinel*; /* dummy coroutine at origin of circular list */

37. Coroutines often start working on tasks that are *speculative*, in the sense that we want certain results to be ready if they prove to be useful; we understand that speculative computations might not actually be needed. Therefore a coroutine might need to be aborted before it has finished its work.

All coroutines must be written in such a way that important data structures remain intact even when the coroutine is abruptly terminated. In particular, we need to be sure that "locks" on shared resources are restored to an unlocked state when a coroutine holding the lock is aborted.

A **lockvar** variable is Λ when it is unlocked; otherwise it points to the coroutine responsible for unlocking it.

```
#define set_lock(c,l)
        { l = c; (c)→lockloc = &(l); }
#define release_lock(c,l)
        { l = Λ; (c)→lockloc = Λ; }
```

⟨ Type definitions 11 ⟩ +≡
typedef coroutine ∗**lockvar**;

38. ⟨ External prototypes 9 ⟩ +≡
Extern void *print_locks* ARGS((**void**));

39. ⟨ External routines 10 ⟩ +≡

```
void print_locks()
{
  print_cache_locks(ITcache);
  print_cache_locks(DTcache);
  print_cache_locks(Icache);
  print_cache_locks(Dcache);
  print_cache_locks(Scache);
  if (mem_lock) printf("mem␣locked␣by␣%s:%d\n", mem_lock→name, mem_lock→stage);
  if (dispatch_lock) printf("dispatch␣locked␣by␣%s:%d\n", dispatch_lock→name, dispatch_lock→stage);
  if (wbuf_lock)
    printf("head␣of␣write␣buffer␣locked␣by␣%s:%d\n", wbuf_lock→name, wbuf_lock→stage);
  if (clean_lock) printf("cleaner␣locked␣by␣%s:%d\n", clean_lock→name, clean_lock→stage);
  if (speed_lock)
    printf("write␣buffer␣flush␣locked␣by␣%s:%d\n", speed_lock→name, speed_lock→stage);
}
```

40. Many of the quantities we deal with are speculative values that might not yet have been certified as part of the "real" calculation; in fact, they might not yet have been calculated.

A **spec** consists of a 64-bit quantity o and a pointer p to a **specnode**. The value o is meaningful only if the pointer p is Λ; otherwise p points to a source of further information.

A **specnode** is a 64-bit quantity o together with links to other **specnode**s that are above it or below it in a doubly linked list. An additional *known* bit tells whether the o field has been calculated. There also is a 64-bit *addr* field, to identify the list and give further information. A **specnode** list keeps track of speculative values related to a specific register or to all of main memory; we will discuss such lists in detail later.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    octa o;
    struct specnode_struct *p;
  } spec;
  typedef struct specnode_struct {
    octa o;
    bool known;
    octa addr;
    struct specnode_struct *up, *down;
  } specnode;
```

41.
```
⟨ Global variables 20 ⟩ +≡
  spec zero_spec;      /* zero_spec.o.h = zero_spec.o.l = 0 and zero_spec.p = Λ */
```

42.
```
⟨ Internal prototypes 13 ⟩ +≡
  static void print_spec ARGS((spec));
```

43.
```
⟨ Subroutines 14 ⟩ +≡
  static void print_spec(s)
      spec s;
  {
    if (¬s.p) print_octa(s.o);
    else {
      printf(">"); print_specnode_id(s.p→addr);
    }
  }
  static void print_specnode(s)
      specnode s;
  {
    if (s.known) { print_octa(s.o); printf("!"); }
    else if (s.o.h ∨ s.o.l) { print_octa(s.o); printf("?"); }
    else printf("?");
    print_specnode_id(s.addr);
  }
```

44. The analog of an automobile in our simulator is a block of data called **control**, which represents all the relevant facts about an MMIX instruction. We can think of it as the work order attached to a car's windshield. Each group of employees updates the work order as the car moves through the shop.

A **control** record contains the original location of an instruction, and its four bytes OP X Y Z. An instruction has up to four inputs, which are **spec** records called y, z, b and ra; it also has up to three outputs, which are **specnode** records called x, a, and rl. (We usually don't mention the special input ra or the special output rl, which refer to MMIX's internal registers rA and rL.) For example, the main inputs to a DIVU command are \$Y, \$Z, and rD; the outputs are the quotient \$X and the remainder rR. The inputs to a STO command are \$Y, \$Z, and \$X; there is one "output," and the field $x.addr$ will be set to the physical address of the memory location corresponding to virtual address \$Y + \$Z.

Each **control** block also points to the coroutine that owns it, if any. And it has various other fields that contain other tidbits of information; for example, we have already mentioned the *state* field, which often governs a coroutine's actions. The i field, which contains an internal operation code number, is generally used together with *state* to switch between alternative computational steps. If, for example, the *op* field is SUB or SUBI or NEG or NEGI, the internal opcode i will be simply *sub*. We shall define all the fields of **control** records now and discuss them later.

An actual hardware implementation of MMIX wouldn't need all the information we are putting into a **control** block. Some of that information would typically be latched between stages of a pipeline; other portions would probably appear in so-called "rename registers." We simulate rename registers only indirectly, by counting how many registers of that kind would be in use if we were mimicking low-level hardware details more precisely. The *go* field is a **specnode** for convenience in programming, although we use only its *known* and o subfields. It generally contains the address of the subsequent instruction.

```
⟨ Type definitions 11 ⟩ +≡
  ⟨ Declare mmix_opcode and internal_opcode 47 ⟩
  typedef struct control_struct {
    octa loc;      /* virtual address where an instruction originated */
    mmix_opcode op; unsigned char xx, yy, zz;      /* the original instruction bytes */
    spec y, z, b, ra;      /* inputs */
    specnode x, a, go, rl;      /* outputs */
    coroutine *owner;      /* a coroutine whose ctl this is */
    internal_opcode i;      /* internal opcode */
    int state;      /* internal mindset */
    bool usage;      /* should rU be increased? */
    bool need_b;      /* should we stall until b.p ≡ Λ? */
    bool need_ra;      /* should we stall until ra.p ≡ Λ? */
    bool ren_x;      /* does x correspond to a rename register? */
    bool mem_x;      /* does x correspond to a memory write? */
    bool ren_a;      /* does a correspond to a rename register? */
    bool set_l;      /* does rl correspond to a new value of rL? */
    bool interim;      /* does this instruction need to be reissued on interrupt? */
    bool stack_alert;      /* is there potential for stack overflow? */
    unsigned int arith_exc;      /* arithmetic exceptions for event bits of rA */
    unsigned int hist;      /* history bits for use in branch prediction */
    int denin, denout;      /* execution time penalties for subnormal handling */
    octa cur_O, cur_S;      /* speculative rO and rS before this instruction */
    unsigned int interrupt;      /* does this instruction generate an interrupt? */
    void *ptr_a, *ptr_b, *ptr_c;      /* generic pointers for miscellaneous use */
  } control;
```

45.
```
⟨ Internal prototypes 13 ⟩ +≡
  static void print_control_block ARGS((control *));
```

46. ⟨Subroutines 14⟩ +≡

```
static void print_control_block(c)
    control *c;
{
  octa default_go;
  if (c→loc.h ∨ c→loc.l ∨ c→op ∨ c→xx ∨ c→yy ∨ c→zz ∨ c→owner) {
    print_octa(c→loc);
    printf(": %02x%02x%02x%02x(%s)", c→op, c→xx, c→yy, c→zz, internal_op_name[c→i]);
  }
  if (c→usage) printf("*");
  if (c→interim) printf("+");
  if (c→y.o.h ∨ c→y.o.l ∨ c→y.p) { printf(" y="); print_spec(c→y); }
  if (c→z.o.h ∨ c→z.o.l ∨ c→z.p) { printf(" z="); print_spec(c→z); }
  if (c→b.o.h ∨ c→b.o.l ∨ c→b.p ∨ c→need_b) {
    printf(" b="); print_spec(c→b);
    if (c→need_b) printf("*");
  }
  if (c→need_ra) { printf(" rA="); print_spec(c→ra); }
  if (c→ren_x ∨ c→mem_x) { printf(" x="); print_specnode(c→x); }
  else if (c→x.o.h ∨ c→x.o.l) {
    printf(" x="); print_octa(c→x.o); printf("%c", c→x.known ? '!' : '?');
  }
  if (c→ren_a) { printf(" a="); print_specnode(c→a); }
  if (c→set_l) { printf(" rL="); print_specnode(c→rl); }
  if (c→interrupt) { printf(" int="); print_bits(c→interrupt); }
  if (c→arith_exc) { printf(" exc="); print_bits(c→arith_exc ≪ 8); }
  default_go = incr(c→loc, 4);
  if (c→go.o.l ≠ default_go.l ∨ c→go.o.h ≠ default_go.h) {
    printf(" ->"); print_octa(c→go.o);
  }
  if (verbose & show_pred_bit) printf(" hist=%x", c→hist);
  if (c→i ≡ pop) {
    printf(" rS=");
    print_octa(c→cur_S);
    printf(" rO=");
    print_octa(c→cur_O);
  }
  printf(" state=%d", c→state);
}
```

47. Lists. Here is a (boring) list of all the MMIX opcodes, in order.

⟨ Declare **mmix_opcode** and **internal_opcode** 47 ⟩ ≡

```
typedef enum {
  TRAP, FCMP, FUN, FEQL, FADD, FIX, FSUB, FIXU,
  FLOT, FLOTI, FLOTU, FLOTUI, SFLOT, SFLOTI, SFLOTU, SFLOTUI,
  FMUL, FCMPE, FUNE, FEQLE, FDIV, FSQRT, FREM, FINT,
  MUL, MULI, MULU, MULUI, DIV, DIVI, DIVU, DIVUI,
  ADD, ADDI, ADDU, ADDUI, SUB, SUBI, SUBU, SUBUI,
  IIADDU, IIADDUI, IVADDU, IVADDUI, VIIIADDU, VIIIADDUI, XVIADDU, XVIADDUI,
  CMP, CMPI, CMPU, CMPUI, NEG, NEGI, NEGU, NEGUI,
  SL, SLI, SLU, SLUI, SR, SRI, SRU, SRUI,
  BN, BNB, BZ, BZB, BP, BPB, BOD, BODB,
  BNN, BNNB, BNZ, BNZB, BNP, BNPB, BEV, BEVB,
  PBN, PBNB, PBZ, PBZB, PBP, PBPB, PBOD, PBODB,
  PBNN, PBNNB, PBNZ, PBNZB, PBNP, PBNPB, PBEV, PBEVB,
  CSN, CSNI, CSZ, CSZI, CSP, CSPI, CSOD, CSODI,
  CSNN, CSNNI, CSNZ, CSNZI, CSNP, CSNPI, CSEV, CSEVI,
  ZSN, ZSNI, ZSZ, ZSZI, ZSP, ZSPI, ZSOD, ZSODI,
  ZSNN, ZSNNI, ZSNZ, ZSNZI, ZSNP, ZSNPI, ZSEV, ZSEVI,
  LDB, LDBI, LDBU, LDBUI, LDW, LDWI, LDWU, LDWUI,
  LDT, LDTI, LDTU, LDTUI, LDO, LDOI, LDOU, LDOUI,
  LDSF, LDSFI, LDHT, LDHTI, CSWAP, CSWAPI, LDUNC, LDUNCI,
  LDVTS, LDVTSI, PRELD, PRELDI, PREGO, PREGOI, GO, GOI,
  STB, STBI, STBU, STBUI, STW, STWI, STWU, STWUI,
  STT, STTI, STTU, STTUI, STO, STOI, STOU, STOUI,
  STSF, STSFI, STHT, STHTI, STCO, STCOI, STUNC, STUNCI,
  SYNCD, SYNCDI, PREST, PRESTI, SYNCID, SYNCIDI, PUSHGO, PUSHGOI,
  OR, ORI, ORN, ORNI, NOR, NORI, XOR, XORI,
  AND, ANDI, ANDN, ANDNI, NAND, NANDI, NXOR, NXORI,
  BDIF, BDIFI, WDIF, WDIFI, TDIF, TDIFI, ODIF, ODIFI,
  MUX, MUXI, SADD, SADDI, MOR, MORI, MXOR, MXORI,
  SETH, SETMH, SETML, SETL, INCH, INCMH, INCML, INCL,
  ORH, ORMH, ORML, ORL, ANDNH, ANDNMH, ANDNML, ANDNL,
  JMP, JMPB, PUSHJ, PUSHJB, GETA, GETAB, PUT, PUTI,
  POP, RESUME, SAVE, UNSAVE, SYNC, SWYM, GET, TRIP
} mmix_opcode;
```

See also section 49.

This code is used in section 44.

48. ⟨ Global variables 20 ⟩ +≡

```
char *opcode_name[] = {
  "TRAP","FCMP","FUN","FEQL","FADD","FIX","FSUB","FIXU",
  "FLOT","FLOTI","FLOTU","FLOTUI","SFLOT","SFLOTI","SFLOTU","SFLOTUI",
  "FMUL","FCMPE","FUNE","FEQLE","FDIV","FSQRT","FREM","FINT",
  "MUL","MULI","MULU","MULUI","DIV","DIVI","DIVU","DIVUI",
  "ADD","ADDI","ADDU","ADDUI","SUB","SUBI","SUBU","SUBUI",
  "2ADDU","2ADDUI","4ADDU","4ADDUI","8ADDU","8ADDUI","16ADDU","16ADDUI",
  "CMP","CMPI","CMPU","CMPUI","NEG","NEGI","NEGU","NEGUI",
  "SL","SLI","SLU","SLUI","SR","SRI","SRU","SRUI",
  "BN","BNB","BZ","BZB","BP","BPB","BOD","BODB",
  "BNN","BNNB","BNZ","BNZB","BNP","BNPB","BEV","BEVB",
  "PBN","PBNB","PBZ","PBZB","PBP","PBPB","PBOD","PBODB",
  "PBNN","PBNNB","PBNZ","PBNZB","PBNP","PBNPB","PBEV","PBEVB",
  "CSN","CSNI","CSZ","CSZI","CSP","CSPI","CSOD","CSODI",
  "CSNN","CSNNI","CSNZ","CSNZI","CSNP","CSNPI","CSEV","CSEVI",
  "ZSN","ZSNI","ZSZ","ZSZI","ZSP","ZSPI","ZSOD","ZSODI",
  "ZSNN","ZSNNI","ZSNZ","ZSNZI","ZSNP","ZSNPI","ZSEV","ZSEVI",
  "LDB","LDBI","LDBU","LDBUI","LDW","LDWI","LDWU","LDWUI",
  "LDT","LDTI","LDTU","LDTUI","LDO","LDOI","LDOU","LDOUI",
  "LDSF","LDSFI","LDHT","LDHTI","CSWAP","CSWAPI","LDUNC","LDUNCI",
  "LDVTS","LDVTSI","PRELD","PRELDI","PREGO","PREGOI","GO","GOI",
  "STB","STBI","STBU","STBUI","STW","STWI","STWU","STWUI",
  "STT","STTI","STTU","STTUI","STO","STOI","STOU","STOUI",
  "STSF","STSFI","STHT","STHTI","STCO","STCOI","STUNC","STUNCI",
  "SYNCD","SYNCDI","PREST","PRESTI","SYNCID","SYNCIDI","PUSHGO","PUSHGOI",
  "OR","ORI","ORN","ORNI","NOR","NORI","XOR","XORI",
  "AND","ANDI","ANDN","ANDNI","NAND","NANDI","NXOR","NXORI",
  "BDIF","BDIFI","WDIF","WDIFI","TDIF","TDIFI","ODIF","ODIFI",
  "MUX","MUXI","SADD","SADDI","MOR","MORI","MXOR","MXORI",
  "SETH","SETMH","SETML","SETL","INCH","INCMH","INCML","INCL",
  "ORH","ORMH","ORML","ORL","ANDNH","ANDNMH","ANDNML","ANDNL",
  "JMP","JMPB","PUSHJ","PUSHJB","GETA","GETAB","PUT","PUTI",
  "POP","RESUME","SAVE","UNSAVE","SYNC","SWYM","GET","TRIP"};
```

49. And here is a (likewise boring) list of all the internal opcodes. The smallest numbers, less than or equal to *max_pipe_op*, correspond to operations for which arbitrary pipeline delays can be configured with *MMIX_config*. The largest numbers, greater than *max_real_command*, correspond to internally generated operations that have no official OP code; for example, there are internal operations to shift the γ pointer in the register stack, and to compute page table entries.

```
⟨ Declare mmix_opcode and internal_opcode 47 ⟩ +≡
#define max_pipe_op feps
#define max_real_command trip
  typedef enum {
    mul0,     /* multiplication by zero */
    mul1,     /* multiplication by 1–8 bits */
    mul2,     /* multiplication by 9–16 bits */
    mul3,     /* multiplication by 17–24 bits */
    mul4,     /* multiplication by 25–32 bits */
    mul5,     /* multiplication by 33–40 bits */
    mul6,     /* multiplication by 41–48 bits */
    mul7,     /* multiplication by 49–56 bits */
    mul8,     /* multiplication by 57–64 bits */
    div,    /* DIV[U][I] */
    sh,    /* S[L,R][U][I] */
    mux,    /* MUX[I] */
    sadd,    /* SADD[I] */
    mor,    /* M[X]OR[I] */
    fadd,    /* FADD, FSUB */
    fmul,    /* FMUL */
    fdiv,    /* FDIV */
    fsqrt,    /* FSQRT */
    fint,    /* FINT */
    fix,    /* FIX[U] */
    flot,    /* [S]FLOT[U][I] */
    feps,    /* FCMPE, FUNE, FEQLE */
    fcmp,    /* FCMP */
    funeq,    /* FUN, FEQL */
    fsub,    /* FSUB */
    frem,    /* FREM */
    mul,    /* MUL[I] */
    mulu,    /* MULU[I] */
    divu,    /* DIVU[I] */
    add,    /* ADD[I] */
    addu,    /* [2,4,8,16,]ADDU[I], INC[M][H,L] */
    sub,    /* SUB[I], NEG[I] */
    subu,    /* SUBU[I], NEGU[I] */
    set,    /* SET[M][H,L], GETA[B] */
    or,    /* OR[I], OR[M][H,L] */
    orn,    /* ORN[I] */
    nor,    /* NOR[I] */
    and,    /* AND[I] */
    andn,    /* ANDN[I], ANDN[M][H,L] */
    nand,    /* NAND[I] */
    xor,    /* XOR[I] */
    nxor,    /* NXOR[I] */
    shlu,    /* SLU[I] */
```

```
  shru,     /* SRU[I] */
  shl,     /* SL[I] */
  shr,     /* SR[I] */
  cmp,     /* CMP[I] */
  cmpu,     /* CMPU[I] */
  bdif,     /* BDIF[I] */
  wdif,     /* WDIF[I] */
  tdif,     /* TDIF[I] */
  odif,     /* ODIF[I] */
  zset,     /* ZS[N][N,Z,P][I], ZSEV[I], ZSOD[I] */
  cset,     /* CS[N][N,Z,P][I], CSEV[I], CSOD[I] */
  get,     /* GET */
  put,     /* PUT[I] */
  ld,     /* LD[B,W,T,O][U][I], LDHT[I], LDSF[I] */
  ldptp,     /* load page table pointer */
  ldpte,     /* load page table entry */
  ldunc,     /* LDUNC[I] */
  ldvts,     /* LDVTS[I] */
  preld,     /* PRELD[I] */
  prest,     /* PREST[I] */
  st,     /* STO[U][I], STCO[I], STUNC[I] */
  syncd,     /* SYNCD[I] */
  syncid,     /* SYNCID[I] */
  pst,     /* ST[B,W,T][U][I], STHT[I] */
  stunc,     /* STUNC[I], in write buffer */
  cswap,     /* CSWAP[I] */
  br,     /* B[N][N,Z,P][B] */
  pbr,     /* PB[N][N,Z,P][B] */
  pushj,     /* PUSHJ[B] */
  go,     /* GO[I] */
  prego,     /* PREGO[I] */
  pushgo,     /* PUSHGO[I] */
  pop,     /* POP */
  resume,     /* RESUME */
  save,     /* SAVE */
  unsave,     /* UNSAVE */
  sync,     /* SYNC */
  jmp,     /* JMP[B] */
  noop,     /* SWYM */
  trap,     /* TRAP */
  trip,     /* TRIP */
  incgamma,     /* increase γ pointer */
  decgamma,     /* decrease γ pointer */
  incrl,     /* increase rL and β */
  sav,     /* intermediate stage of SAVE */
  unsav,     /* intermediate stage of UNSAVE */
  resum     /* intermediate stage of RESUME */
} internal_opcode;
```

50. ⟨Global variables 20⟩ +≡

```
char *internal_op_name[] = {"mul0","mul1","mul2","mul3","mul4","mul5","mul6","mul7","mul8",
    "div","sh","mux","sadd","mor","fadd","fmul","fdiv","fsqrt","fint","fix","flot",
    "feps","fcmp","funeq","fsub","frem","mul","mulu","divu","add","addu","sub","subu",
    "set","or","orn","nor","and","andn","nand","xor","nxor","shlu","shru","shl","shr",
    "cmp","cmpu","bdif","wdif","tdif","odif","zset","cset","get","put","ld","ldptp",
    "ldpte","ldunc","ldvts","preld","prest","st","syncd","syncid","pst","stunc","cswap",
    "br","pbr","pushj","go","prego","pushgo","pop","resume","save","unsave","sync","jmp",
    "noop","trap","trip","incgamma","decgamma","incrl","sav","unsav","resum"};
```

51. We need a table to convert the external opcodes to internal ones.

⟨Global variables 20⟩ +≡

```
internal_opcode internal_op[256] = {
    trap,fcmp,funeq,funeq,fadd,fix,fsub,fix,
    flot,flot,flot,flot,flot,flot,flot,flot,
    fmul,feps,feps,feps,fdiv,fsqrt,frem,fint,
    mul,mul,mulu,mulu,div,div,divu,divu,
    add,add,addu,addu,sub,sub,subu,subu,
    addu,addu,addu,addu,addu,addu,addu,addu,
    cmp,cmp,cmpu,cmpu,sub,sub,subu,subu,
    shl,shl,shlu,shlu,shr,shr,shru,shru,
    br,br,br,br,br,br,br,br,
    br,br,br,br,br,br,br,br,
    pbr,pbr,pbr,pbr,pbr,pbr,pbr,pbr,
    pbr,pbr,pbr,pbr,pbr,pbr,pbr,pbr,
    cset,cset,cset,cset,cset,cset,cset,cset,
    cset,cset,cset,cset,cset,cset,cset,cset,
    zset,zset,zset,zset,zset,zset,zset,zset,
    zset,zset,zset,zset,zset,zset,zset,zset,
    ld,ld,ld,ld,ld,ld,ld,ld,
    ld,ld,ld,ld,ld,ld,ld,ld,
    ld,ld,ld,ld,cswap,cswap,ldunc,ldunc,
    ldvts,ldvts,preld,preld,prego,prego,go,go,
    pst,pst,pst,pst,pst,pst,pst,pst,
    pst,pst,pst,pst,st,st,st,st,
    pst,pst,pst,pst,st,st,st,st,
    syncd,syncd,prest,prest,syncid,syncid,pushgo,pushgo,
    or,or,orn,orn,nor,nor,xor,xor,
    and,and,andn,andn,nand,nand,nxor,nxor,
    bdif,bdif,wdif,wdif,tdif,tdif,odif,odif,
    mux,mux,sadd,sadd,mor,mor,mor,mor,
    set,set,set,set,addu,addu,addu,addu,
    or,or,or,or,andn,andn,andn,andn,
    jmp,jmp,pushj,pushj,set,set,put,put,
    pop,resume,save,unsave,sync,noop,get,trip};
```

52. While we're into boring lists, we might as well define all the special register numbers, together with an inverse table for use in diagnostic outputs. These codes have been designed so that special registers 0–7 are unencumbered, 9–11 can't be `PUT` by anybody, 8 and 12–18 can't be `PUT` by the user. Pipeline delays might occur when `GET` is applied to special registers 21–31 or when `PUT` is applied to special registers 8 or 15–20. The `SAVE` and `UNSAVE` commands store and restore special registers 0–6 and 23–27 followed by 19 and 21.

```
⟨ Header definitions 6 ⟩ +≡
#define rA 21 /* arithmetic status register */
#define rB 0 /* bootstrap register (trip) */
#define rC 8 /* continuation register */
#define rD 1 /* dividend register */
#define rE 2 /* epsilon register */
#define rF 22 /* failure location register */
#define rG 19 /* global threshold register */
#define rH 3 /* himult register */
#define rI 12 /* interval counter */
#define rJ 4 /* return-jump register */
#define rK 15 /* interrupt mask register */
#define rL 20 /* local threshold register */
#define rM 5 /* multiplex mask register */
#define rN 9 /* serial number */
#define rO 10 /* register stack offset */
#define rP 23 /* prediction register */
#define rQ 16 /* interrupt request register */
#define rR 6 /* remainder register */
#define rS 11 /* register stack pointer */
#define rT 13 /* trap address register */
#define rU 17 /* usage counter */
#define rV 18 /* virtual translation register */
#define rW 24 /* where-interrupted register (trip) */
#define rX 25 /* execution register (trip) */
#define rY 26 /* Y operand (trip) */
#define rZ 27 /* Z operand (trip) */
#define rBB 7 /* bootstrap register (trap) */
#define rTT 14 /* dynamic trap address register */
#define rWW 28 /* where-interrupted register (trap) */
#define rXX 29 /* execution register (trap) */
#define rYY 30 /* Y operand (trap) */
#define rZZ 31 /* Z operand (trap) */
```

53.
```
⟨ Global variables 20 ⟩ +≡
  char *special_name[32] = {"rB","rD","rE","rH","rJ","rM","rR","rBB","rC","rN","rO","rS",
      "rI","rT","rTT","rK","rQ","rU","rV","rG","rL","rA","rF","rP","rW","rX","rY","rZ",
      "rWW","rXX","rYY","rZZ"};
```

54. Here are the bit codes that affect trips and traps. The first eight cases also apply to the upper half of rQ; the next eight apply to rA.

```
#define P_BIT (1 ≪ 0)      /* instruction in privileged location */
#define S_BIT (1 ≪ 1)      /* security violation */
#define B_BIT (1 ≪ 2)      /* instruction breaks the rules */
#define K_BIT (1 ≪ 3)      /* instruction for kernel only */
#define N_BIT (1 ≪ 4)      /* virtual translation bypassed */
#define PX_BIT (1 ≪ 5)     /* permission lacking to execute from page */
#define PW_BIT (1 ≪ 6)     /* permission lacking to write on page */
#define PR_BIT (1 ≪ 7)     /* permission lacking to read from page */
#define PROT_OFFSET 5      /* distance from PR_BIT to protection code position */
#define X_BIT (1 ≪ 8)      /* floating inexact */
#define Z_BIT (1 ≪ 9)      /* floating division by zero */
#define U_BIT (1 ≪ 10)     /* floating underflow */
#define O_BIT (1 ≪ 11)     /* floating overflow */
#define I_BIT (1 ≪ 12)     /* floating invalid operation */
#define W_BIT (1 ≪ 13)     /* float-to-fix overflow */
#define V_BIT (1 ≪ 14)     /* integer overflow */
#define D_BIT (1 ≪ 15)     /* integer divide check */
#define H_BIT (1 ≪ 16)     /* trip handler bit */
#define F_BIT (1 ≪ 17)     /* forced trap bit */
#define E_BIT (1 ≪ 18)     /* external (dynamic) trap bit */
```

⟨ Global variables 20 ⟩ +≡

```
char bit_code_map[] = "EFHDVWIOUZXrwxnkbsp";
```

55. ⟨ Internal prototypes 13 ⟩ +≡

```
static void print_bits ARGS((int));
```

56. ⟨ Subroutines 14 ⟩ +≡

```
static void print_bits(x)
    int x;
{
  register int b, j;
  for (j = 0, b = E_BIT; (x & (b + b − 1)) ∧ b; j++, b ≫= 1)
    if (x & b) printf("%c", bit_code_map[j]);
}
```

57. The lower half of rQ holds external interrupts of highest priority. Most of them are implementation-dependent, but a few are defined in general.

⟨ Header definitions 6 ⟩ +≡

```
#define POWER_FAILURE (1 ≪ 0)      /* try to shut down calmly and quickly */
#define PARITY_ERROR (1 ≪ 1)       /* try to save the file systems */
#define NONEXISTENT_MEMORY (1 ≪ 2) /* a memory address can't be used */
#define REBOOT_SIGNAL (1 ≪ 4)      /* it's time to start over */
#define INTERVAL_TIMEOUT (1 ≪ 6)   /* the timer register, rI, has reached zero */
#define STACK_OVERFLOW (1 ≪ 7)     /* data has been stored on the rC page */
```

58. Dynamic speculation. Now that we understand some basic low-level structures, we're ready to look at the larger picture.

This simulator is based on the idea of "dynamic scheduling with register renaming," as introduced in the 1960s by R. M. Tomasulo [*IBM Journal of Research and Development* **11** (1967), 25–33]. Moreover, the dynamic scheduling method is extended here to "speculative execution," as implemented in several processors of the 1990s and described in section 4.6 of Hennessy and Patterson's *Computer Architecture*, second edition (1995). The essential idea is to keep track of the pipeline contents by recording all dependencies between unfinished computations in a queue called the *reorder buffer*. An entry in the reorder buffer might, for example, correspond to an instruction that adds together two numbers whose values are still being computed; those numbers have been allocated space in earlier positions of the reorder buffer. The addition will take place as soon as both of its operands are known, but the sum won't be written immediately into the destination register. It will stay in the reorder buffer until reaching the *hot seat* at the front of the queue. Finally, the addition leaves the hot seat and is said to be *committed*.

Some instructions in the reorder buffer may in fact be executed only on speculation, meaning that they won't really be called for unless a prior branch instruction has the predicted outcome. Indeed, we can say that all instructions not yet in the hot seat are being executed speculatively, because an external interrupt might occur at any time and change the entire course of computation. Organizing the pipeline as a reorder buffer allows us to look ahead and keep busy computing values that have a good chance of being needed later, instead of waiting for slow instructions or slow memory references to be completed.

The reorder buffer is in fact a queue of **control** records, conceptually forming part of a circle of such records inside the simulator, corresponding to all instructions that have been dispatched or *issued* but not yet committed, in strict program order.

The best way to get an understanding of speculative execution is perhaps to imagine that the reorder buffer is large enough to hold hundreds of instructions in various stages of execution, and to think of an implementation of `MMIX` that has dozens of functional units—more than would ever actually be built into a chip. Then one can readily visualize the kinds of control structures and checks that must be made to ensure correct execution. Without such a broad viewpoint, a programmer or hardware designer will be inclined to think only of the simple cases and to devise algorithms that lack the proper generality. Thus we have a somewhat paradoxical situation in which a difficult general problem turns out to be easier to solve than its simpler special cases, because it enforces clarity of thinking.

Instructions that have completed execution and have not yet been committed are analogous to cars that have gone through our hypothetical repair shop and are waiting for their owners to pick them up. However, all analogies break down, and the world of automobiles does not have a natural counterpart for the notion of speculative execution. That notion corresponds roughly to situations in which people are led to believe that their cars need a new piece of equipment, but they suddenly change their mind once they see the price tag, and they insist on having the equipment removed even after it has been partially or completely installed.

Speculatively executed instructions might make no sense: They might divide by zero or refer to protected memory areas, etc. Such anomalies are not considered catastrophic or even exceptional until the instruction reaches the hot seat.

The person who designs a computer with speculative execution is an optimist, who has faith that the vast majority of the machine's predictions will come true. The person who designs a reliable implementation of such a computer is a pessimist, who understands that all predictions might come to naught. The pessimist does, however, take pains to optimize the cases that do turn out well.

59. Let's consider what happens to a single instruction, say `ADD $1,$2,$3`, as it travels through the pipeline in a normal situation. The first time this instruction is encountered, it is placed into the I-cache (that is, the instruction cache), so that we won't have to access memory when we need to perform it again. We will assume for simplicity in this discussion that each I-cache access takes one clock cycle, although other possibilities are allowed by *MMIX_config*.

Suppose the simulated machine fetches the example `ADD` instruction at time 1000. Fetching is done by a coroutine whose *stage* number is 0. A cache block typically contains 8 or 16 instructions. The fetch unit of our machine is able to fetch up to *fetch_max* instructions on each clock cycle and place them in the fetch buffer, provided that there is room in the buffer and that all the instructions belong to the same cache block.

The dispatch unit of our simulator is able to issue up to *dispatch_max* instructions on each clock cycle and move them from the fetch buffer to the reorder buffer, provided that functional units are available for those instructions and there is room in the reorder buffer. A functional unit that handles `ADD` is usually called an ALU (arithmetic logic unit), and our simulated machine might have several of them. If they aren't all stalled in stage 1 of their pipelines, and if the reorder buffer isn't full, and if the machine isn't in the process of deissuing instructions that were mispredicted, and if fewer than *dispatch_max* instructions are ahead of the `ADD` in the fetch buffer, and if all such prior instructions can be issued without using up all the free ALUs, our `ADD` instruction will be issued at time 1001. (In fact, all of these conditions are usually true.)

We assume that $L > 3$, so that \$1, \$2, and \$3 are local registers. For simplicity we'll assume in fact that the register stack is empty, so that the `ADD` instruction is supposed to set $l[1] \leftarrow l[2] + l[3]$. The operands l[2] and l[3] might not be known at time 1001; they are **spec** values, which might point to **specnode** entries in the reorder buffer for previous instructions whose destinations are l[2] and l[3]. The dispatcher fills the next available control block of the reorder buffer with information for the `ADD`, containing appropriate **spec** values corresponding to l[2] and l[3] in its y and z fields. The x field of this control block will be inserted into a doubly linked list of **specnode** records, corresponding to l[1] and to all instructions in the reorder buffer that have l[1] as a destination. The boolean value $x.known$ will be set to *false*, meaning that this speculative value still needs to be computed. Subsequent instructions that need l[1] as a source will point to x, if they are issued before the sum $x.o$ has been computed. Double linking is used in the **specnode** list because the `ADD` instruction might be cancelled before it is finally committed; thus deletions might occur at either end of the list for l[1].

At time 1002, the ALU handling the `ADD` will stall if its inputs y and z are not both known (namely if $y.p \neq \Lambda$ or $z.p \neq \Lambda$). In fact, it will also stall if its third input rA is not known; the current speculative value of rA, except for its event bits, is represented in the *ra* field of the control block, and we must have $ra.p \equiv \Lambda$. In such a case the ALU will look to see if the **spec** values pointed to by $y.p$ and/or $z.p$ and/or $ra.p$ become defined on this clock cycle, and it will update its own input values accordingly.

But let's assume that y, z, and *ra* are already known at time 1002. Then $x.o$ will be set to $y.o + z.o$ and $x.known$ will become *true*. This will make the result destined for l[1] available to be used in other commands at time 1003.

If no overflow occurs when adding $y.o$ to $z.o$, the *interrupt* and *arith_exc* fields of the control block for `ADD` are set to zero. But when overflow does occur (shudder), there are two cases, based on the V-enable bit of rA, which is found in field $b.o$ of the control block. If this bit is 0, the V-bit of the *arith_exc* field in the control block is set to 1; the *arith_exc* field will be ored into rA when the `ADD` instruction is eventually committed. But if the V-enable bit is 1, the trip handler should be called, interrupting the normal sequence. In such a case, the *interrupt* field of the control block is set to specify a trip, and the fetcher and dispatcher are told to forget what they have been doing; all instructions following the `ADD` in the reorder buffer must now be deissued. The virtual starting address of the overflow trip handler, namely location 32, is hastily passed to the fetch routine, and instructions will be fetched from that location as soon as possible. (Of course the overflow and the trip handler are still speculative until the `ADD` instruction is committed. Other exceptional conditions might cause the `ADD` itself to be terminated before it gets to the hot seat. But the pipeline keeps charging ahead, always trying to guess the most probable outcome.)

The commission unit of this simulator is able to commit and/or deissue up to *commit_max* instructions on each clock cycle. With luck, fewer than *commit_max* instructions will be ahead of our `ADD` instruction at time 1003, and they will all be completed normally. Then l[1] can be set to $x.o$, and the event bits of rA

can be updated from *arith_exc*, and the `ADD` command can pass through the hot seat and out of the reorder buffer.

⟨ External variables 4 ⟩ +≡
```
Extern int fetch_max, dispatch_max, peekahead, commit_max;
  /* limits on instructions that can be handled per clock cycle */
```

60. The instruction currently occupying the hot seat is the only issued-but-not-yet-committed instruction that is guaranteed to be truly essential to the machine's computation. All other instructions in the reorder buffer are being executed on speculation; if they prove to be needed, well and good, but we might want to jettison them all if, say, an external interrupt occurs.

Thus all instructions that change the global state in complicated ways—like `LDVTS`, which changes the virtual address translation caches—are performed only when they reach the hot seat. Fortunately the vast majority of instructions are sufficiently simple that we can deal with them more efficiently while other computations are taking place.

In this implementation the reorder buffer is simply housed in an array of control records. The first array element is *reorder_bot*, and the last is *reorder_top*. Variable *hot* points to the control block in the hot seat, and $hot - 1$ to its predecessor, etc. Variable *cool* points to the next control block that will be filled in the reorder buffer. If $hot \equiv cool$ the reorder buffer is empty; otherwise it contains the control records hot, $hot - 1, \ldots, cool + 1$, except of course that we wrap around from *reorder_bot* to *reorder_top* when moving down in the buffer.

⟨ External variables 4 ⟩ +≡
```
Extern control *reorder_bot, *reorder_top;
  /* least and greatest entries in the ring containing the reorder buffer */
Extern control *hot, *cool;      /* front and rear of the reorder buffer */
Extern control *old_hot;      /* value of hot at beginning of cycle */
Extern int deissues;      /* the number of instructions that need to be deissued */
```

61. ⟨ Initialize everything 22 ⟩ +≡
```
hot = cool = reorder_top;
deissues = 0;
```

62. ⟨ Internal prototypes 13 ⟩ +≡
```
static void print_reorder_buffer ARGS((void));
```

63. ⟨Subroutines 14⟩ +≡

```
static void print_reorder_buffer()
{
  printf("Reorder␣buffer");
  if (hot ≡ cool) printf("␣(empty)\n");
  else { register control *p;
    if (deissues) printf("␣(%d␣to␣be␣deissued)", deissues);
    if (doing_interrupt) printf("␣(interrupt␣state␣%d)", doing_interrupt);
    printf(":\n");
    for (p = hot; p ≠ cool; p = (p ≡ reorder_bot ? reorder_top : p − 1)) {
      print_control_block(p);
      if (p→owner) {
        printf("␣"); print_coroutine_id(p→owner);
      }
      printf("\n");
    }
  }
  printf("␣%d␣available␣rename␣register%s,␣%d␣memory␣slot%s\n", rename_regs,
      rename_regs ≠ 1 ? "s" : "", mem_slots, mem_slots ≠ 1 ? "s" : "");
}
```

64. Here is an overview of what happens on each clock cycle.

```
⟨Perform one machine cycle 64⟩ ≡
  {
    ⟨Check for external interrupt 314⟩;
    dispatch_count = 0;
    old_hot = hot;      /* remember the hot seat position at beginning of cycle */
    old_tail = tail;    /* remember the fetch buffer contents at beginning of cycle */
    suppress_dispatch = (deissues ∨ dispatch_lock);
    if (doing_interrupt) ⟨Perform one cycle of the interrupt preparations 318⟩
    else ⟨Commit and/or deissue up to commit_max instructions 67⟩;
    ⟨Execute all coroutines scheduled for the current time 125⟩;
    if (¬suppress_dispatch) ⟨Dispatch one cycle's worth of instructions 74⟩;
    ticks = incr(ticks, 1);     /* and the beat moves on */
    dispatch_stat[dispatch_count]++;
  }
```

This code is used in section 10.

65. ⟨Global variables 20⟩ +≡

```
int dispatch_count;     /* how many dispatched on this cycle */
bool suppress_dispatch;     /* should dispatching be bypassed? */
int doing_interrupt;     /* how many cycles of interrupt preparations remain */
lockvar dispatch_lock;     /* lock to prevent instruction issues */
```

66. ⟨External variables 4⟩ +≡

```
Extern int *dispatch_stat;     /* how often did we dispatch 0, 1, ... instructions? */
Extern bool security_disabled;     /* omit security checks for testing purposes? */
```

67. ⟨ Commit and/or deissue up to *commit_max* instructions 67 ⟩ ≡

```
{
  for (m = commit_max; m > 0 ∧ deissues > 0; m−−) ⟨ Deissue the coolest instruction 145 ⟩;
  for ( ; m > 0; m−−) {
    if (hot ≡ cool) break;      /* reorder buffer is empty */
    if (¬security_disabled) ⟨ Check for security violation, break if so 149 ⟩;
    if (hot→owner) break;      /* hot seat instruction isn't finished */
    ⟨ Commit the hottest instruction, or break if it's not ready 146 ⟩;
    i = hot→i;
    if (hot ≡ reorder_bot) hot = reorder_top;
    else hot−−;
    if (i ≡ resum) break;      /* allow the resumed instruction to see the new rK */
  }
}
```

This code is used in section 64.

68. The dispatch stage. It would be nice to present the parts of this simulator by dealing with the fetching, dispatching, executing, and committing stages in that order. After all, instructions are first fetched, then dispatched, then executed, and finally committed. However, the fetch stage depends heavily on difficult questions of memory management that are best deferred until we have looked at the simpler parts of simulation. Therefore we will take our initial plunge into the details of this program by looking first at the dispatch phase, assuming that instructions have somehow appeared magically in the fetch buffer.

The fetch buffer, like the circular priority queue of all coroutines and the circular queue used for the reorder buffer, lives in an array that is best regarded as a ring of elements. The elements are structures of type **fetch**, which have five fields: A 32-bit *inst*, which is an `MMIX` instruction; a 64-bit *loc*, which is the virtual address of that instruction; an *interrupt* field, which is nonzero if, for example, the protection bits in the relevant page table entry for this address do not permit execution access; a boolean *noted* field, which becomes *true* after the dispatch unit has peeked at the instruction to see whether it is a jump or probable branch; and a *hist* field, which records the recent branch history. (The least significant bits of *hist* correspond to the most recent branches.)

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    octa loc;      /* virtual address of instruction */
    tetra inst;      /* the instruction itself */
    unsigned int interrupt;      /* bit codes that might cause interruption */
    bool noted;      /* have we peeked at this instruction? */
    unsigned int hist;      /* if we peeked, this was the peek_hist */
  } fetch;
```

69. The oldest and youngest entries in the fetch buffer are pointed to by *head* and *tail*, just as the oldest and youngest entries in the reorder buffer are called *hot* and *cool*. The fetch coroutine will be adding entries at the *tail* position, which starts at *old_tail* when a cycle begins, in parallel with the actions simulated by the dispatcher. Therefore the dispatcher is allowed to look only at instructions in *head*, *head* − 1, ..., *old_tail* + 1, although a few more recently fetched instructions will usually be present in the fetch buffer by the time this part of the program is executed.

```
⟨ External variables 4 ⟩ +≡
  Extern fetch *fetch_bot, *fetch_top;
    /* least and greatest entries in the ring containing the fetch buffer */
  Extern fetch *head, *tail;      /* front and rear of the fetch buffer */
```

70.
```
⟨ Global variables 20 ⟩ +≡
  fetch *old_tail;      /* rear of the fetch buffer available on the current cycle */
```

71.
```
#define UNKNOWN_SPEC ((specnode *) 1)
⟨ Initialize everything 22 ⟩ +≡
  head = tail = fetch_top;
  inst_ptr.p = UNKNOWN_SPEC;
```

72.
```
⟨ Internal prototypes 13 ⟩ +≡
  static void print_fetch_buffer ARGS((void));
```

73. ⟨Subroutines 14⟩ +≡

```
static void print_fetch_buffer()
{
  printf("Fetch␣buffer");
  if (head ≡ tail) printf("␣(empty)\n");
  else { register fetch *p;
    if (resuming) printf("␣(resumption␣state␣%d)", resuming);
    printf(":\n");
    for (p = head; p ≠ tail; p = (p ≡ fetch_bot ? fetch_top : p − 1)) {
      print_octa(p→loc);
      printf(":␣%08x(%s)", p→inst, opcode_name[p→inst ≫ 24]);
      if (p→interrupt) print_bits(p→interrupt);
      if (p→noted) printf("*");
      printf("\n");
    }
  }
  printf("Instruction␣pointer␣is␣");
  if (inst_ptr.p ≡ Λ) print_octa(inst_ptr.o);
  else {
    printf("waiting␣for␣");
    if (inst_ptr.p ≡ UNKNOWN_SPEC) printf("dispatch");
    else if (inst_ptr.p→addr.h ≡ (tetra) −1) print_coroutine_id(((control *) inst_ptr.p→up)→owner);
    else print_specnode_id(inst_ptr.p→addr);
  }
  printf("\n");
}
```

74. The best way to understand the dispatching process is once again to "think big," by imagining a huge fetch buffer and the potential ability to issue dozens of instructions per cycle, although the actual numbers are typically quite small.

If the fetch buffer is not empty after *dispatch_max* instructions have been dispatched, the dispatcher also looks at up to *peekahead* further instructions to see if they are jumps or other commands that change the flow of control. Much of this action would happen in parallel on a real machine, but our simulator works sequentially.

In the following program, *true_head* records the head of the fetch buffer as instructions are actually dispatched, while *head* refers to the position currently being examined (possibly peeking into the future).

If the fetch buffer is empty at the beginning of the current clock cycle, a "dispatch bypass" allows the dispatcher to issue the first instruction that enters the fetch buffer on this cycle. Otherwise the dispatcher is restricted to previously fetched instructions.

⟨Dispatch one cycle's worth of instructions 74⟩ ≡

```
{ register fetch *true_head, *new_head;
  true_head = head;
  if (head ≡ old_tail ∧ head ≠ tail) old_tail = (head ≡ fetch_bot ? fetch_top : head − 1);
  peek_hist = cool_hist;
  for (j = 0; j < dispatch_max + peekahead; j++)
    ⟨Look at the head instruction, and try to dispatch it if j < dispatch_max 75⟩;
  head = true_head;
}
```

This code is used in section 64.

75. ⟨ Look at the *head* instruction, and try to dispatch it if $j < dispatch\_max$ 75 ⟩ ≡

```
{
  register mmix_opcode op;
  register int yz, f;
  register bool freeze_dispatch = false;
  register func *u = Λ;
  if (head ≡ old_tail) break;      /* fetch buffer empty */
  if (head ≡ fetch_bot) new_head = fetch_top; else new_head = head − 1;
  op = head→inst ≫ 24; yz = head→inst & #ffff;
  ⟨ Determine the flags, f, and the internal opcode, i 80 ⟩;
  ⟨ Install default fields in the cool block 100 ⟩;
  if (f & rel_addr_bit) ⟨ Convert relative address to absolute address 84 ⟩;
  if (head→noted) peek_hist = head→hist;
  else ⟨ Redirect the fetch if control changes at this inst 85 ⟩;
  if (j ≥ dispatch_max ∨ dispatch_lock ∨ nullifying) {
    head = new_head; continue;      /* can't dispatch, but can peek ahead */
  }
  if (cool ≡ reorder_bot) new_cool = reorder_top; else new_cool = cool − 1;
  ⟨ Dispatch an instruction to the cool block if possible, otherwise goto stall 101 ⟩;
  ⟨ Assign a functional unit if available, otherwise goto stall 82 ⟩;
  ⟨ Check for sufficient rename registers and memory slots, or goto stall 111 ⟩;
  if ((op & #e0) ≡ #40) ⟨ Record the result of branch prediction 152 ⟩;
  ⟨ Issue the cool instruction 81 ⟩;
  cool = new_cool; cool_O = new_O; cool_S = new_S;
  cool_hist = peek_hist; continue;
stall: ⟨ Undo data structures set prematurely in the cool block and break 123 ⟩;
}
```

This code is used in section 74.

76. An instruction can be dispatched only if a functional unit is available to handle it. A functional unit consists of a 256-bit vector that specifies a subset of MMIX's opcodes, and an array of coroutines for the pipeline stages. There are k coroutines in the array, where k is the maximum number of stages needed by any of the opcodes supported.

⟨ Type definitions 11 ⟩ +≡

```
typedef struct func_struct {
  char name[16];      /* symbolic designation */
  tetra ops[8];       /* big-endian bitmap for the opcodes supported */
  int k;      /* number of pipeline stages */
  coroutine *co;      /* pointer to the first of k consecutive coroutines */
} func;
```

77. ⟨ External variables 4 ⟩ +≡

```
Extern func *funit;      /* pointer to array of functional units */
Extern int funit_count;      /* the number of functional units */
```

78. It is convenient to have a 256-bit vector of all the supported opcodes, because we need to shut off a lot of special actions when an opcode is not supported.

⟨ Global variables 20 ⟩ +≡

```
control *new_cool;      /* the reorder position following cool */
int resuming;      /* set nonzero if resuming an interrupted instruction */
tetra support[8];      /* big-endian bitmap for all opcodes supported */
```

79. ⟨ Initialize everything 22 ⟩ +≡

```
{ register func *u;
  for (u = funit; u ≤ funit + funit_count; u++)
    for (i = 0; i < 8; i++) support[i] |= u→ops[i];
}
```

80. `#define sign_bit ((unsigned) #80000000)`

⟨ Determine the flags, f, and the internal opcode, i 80 ⟩ ≡

```
if (¬(support[op ≫ 5] & (sign_bit ≫ (op & 31)))) {
    /* oops, this opcode isn't supported by any functional unit */
  f = flags[TRAP], i = trap;
} else f = flags[op], i = internal_op[op];
if (i ≡ trip ∧ (head→loc.h & sign_bit)) f = 0, i = noop;
```

This code is used in section 75.

81. ⟨ Issue the *cool* instruction 81 ⟩ ≡

```
if (cool→interim) {
  cool→usage = false;
  if (cool→op ≡ SAVE) ⟨ Get ready for the next step of SAVE 341 ⟩
  else if (cool→op ≡ UNSAVE) ⟨ Get ready for the next step of UNSAVE 335 ⟩
  else if (cool→i ≡ preld ∨ cool→i ≡ prest) ⟨ Get ready for the next step of PRELD or PREST 228 ⟩
  else if (cool→i ≡ prego) ⟨ Get ready for the next step of PREGO 229 ⟩
}
else if (cool→i ≤ max_real_command) {
  if ((flags[cool→op] & ctl_change_bit) ∨ cool→i ≡ pbr)
    if (inst_ptr.p ≡ Λ ∧ (inst_ptr.o.h & sign_bit) ∧ ¬(cool→loc.h & sign_bit) ∧ cool→i ≠ trap)
      cool→interrupt |= P_BIT;      /* jumping from nonnegative to negative */
  true_head = head = new_head;      /* delete instruction from fetch buffer */
  resuming = 0;
}
if (freeze_dispatch) set_lock(u→co, dispatch_lock);
cool→owner = u→co; u→co→ctl = cool;
startup(u→co, 1);      /* schedule execution of the new inst */
if (verbose & issue_bit) {
  printf("Issuing␣"); print_control_block(cool);
  printf("␣"); print_coroutine_id(u→co); printf("\n");
}
dispatch_count++;
```

This code is used in section 75.

82. We assign the first functional unit that supports *op* and is totally unoccupied, if possible; otherwise we assign the first functional unit that supports *op* and has stage 1 unoccupied.

```
⟨ Assign a functional unit if available, otherwise goto stall 82 ⟩ ≡
  { register int t = op ≫ 5, b = sign_bit ≫ (op & 31);
    if (cool→i ≡ trap ∧ op ≠ TRAP) {      /* opcode needs to be emulated */
      u = funit + funit_count;      /* this unit supports just TRIP and TRAP */
      goto unit_found;
    }
    for (u = funit; u ≤ funit + funit_count; u++)
      if (u→ops[t] & b) {
        for (i = 0; i < u→k; i++)
          if (u→co[i].next) goto unit_busy;
        goto unit_found;
      unit_busy: ;
      }
    for (u = funit; u < funit + funit_count; u++)
      if ((u→ops[t] & b) ∧ (u→co→next ≡ Λ)) goto unit_found;
    goto stall;      /* all units for this op are busy */
  }
  unit_found:
```

This code is used in section 75.

83. The *flags* table records special properties of each operation code in binary notation: #1 means Z is an immediate value, #2 means rZ is a source operand, #4 means Y is an immediate value, #8 means rY is a source operand, #10 means rX is a source operand, #20 means rX is a destination, #40 means YZ is part of a relative address, #80 means the control changes at this point.

```
#define X_is_dest_bit #20
#define rel_addr_bit #40
#define ctl_change_bit #80
```

⟨ Global variables 20 ⟩ +≡

```
unsigned char flags[256] = {
    #8a,#2a,#2a,#2a,#2a,#26,#2a,#26,    /* TRAP, ... */
    #26,#25,#26,#25,#26,#25,#26,#25,    /* FLOT, ... */
    #2a,#2a,#2a,#2a,#2a,#26,#2a,#26,    /* FMUL, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* MUL, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* ADD, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* 2ADDU, ... */
    #2a,#29,#2a,#29,#26,#25,#26,#25,    /* CMP, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* SL, ... */
    #50,#50,#50,#50,#50,#50,#50,#50,    /* BN, ... */
    #50,#50,#50,#50,#50,#50,#50,#50,    /* BNN, ... */
    #50,#50,#50,#50,#50,#50,#50,#50,    /* PBN, ... */
    #50,#50,#50,#50,#50,#50,#50,#50,    /* PBNN, ... */
    #3a,#39,#3a,#39,#3a,#39,#3a,#39,    /* CSN, ... */
    #3a,#39,#3a,#39,#3a,#39,#3a,#39,    /* CSNN, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* ZSN, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* ZSNN, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* LDB, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* LDT, ... */
    #2a,#29,#2a,#29,#3a,#39,#2a,#29,    /* LDSF, ... */
    #2a,#29,#0a,#09,#0a,#09,#aa,#a9,    /* LDVTS, ... */
    #1a,#19,#1a,#19,#1a,#19,#1a,#19,    /* STB, ... */
    #1a,#19,#1a,#19,#1a,#19,#1a,#19,    /* STT, ... */
    #1a,#19,#1a,#19,#0a,#09,#1a,#19,    /* STSF, ... */
    #0a,#09,#0a,#09,#0a,#09,#aa,#a9,    /* SYNCD, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* OR, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* AND, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* BDIF, ... */
    #2a,#29,#2a,#29,#2a,#29,#2a,#29,    /* MUX, ... */
    #20,#20,#20,#20,#30,#30,#30,#30,    /* SETH, ... */
    #30,#30,#30,#30,#30,#30,#30,#30,    /* ORH, ... */
    #c0,#c0,#e0,#e0,#60,#60,#02,#01,    /* JMP, ... */
    #80,#80,#00,#02,#01,#00,#20,#8a};    /* POP, ... */
```

84. ⟨ Convert relative address to absolute address 84 ⟩ ≡

```
{
  if (i ≡ jmp) yz = head→inst & #ffffff;
  if (op & 1) yz −= (i ≡ jmp ? #1000000 : #10000);
  cool→y.o = incr(head→loc, 4), cool→y.p = Λ;
  cool→z.o = incr(head→loc, yz ≪ 2), cool→z.p = Λ;
}
```

This code is used in section 75.

85. The location of the next instruction to be fetched is in a **spec** variable called *inst_ptr*. A slightly tricky optimization of the POP instruction is made in the common case that the speculative value of rJ is known.

```
⟨ Redirect the fetch if control changes at this inst 85 ⟩ ≡
  { register int predicted = 0;
    if ((op & #e0) ≡ #40) ⟨ Predict a branch outcome 151 ⟩;
    head→noted = true;
    head→hist = peek_hist;
    if (predicted ∨ (f & ctl_change_bit) ∨ (i ≡ syncid ∧ ¬(cool→loc.h & sign_bit))) {
      old_tail = tail = new_head;      /* discard all remaining fetches */
      ⟨ Restart the fetch coroutine 287 ⟩;
      switch (i) {
      case jmp: case br: case pbr: case pushj: inst_ptr = cool→z; break;
      case pop: if (g[rJ].up→known ∧ j < dispatch_max ∧ ¬dispatch_lock ∧ ¬nullifying) {
          inst_ptr.o = incr(g[rJ].up→o, yz ≪ 2), inst_ptr.p = Λ; break;
        }      /* otherwise fall through, will wait on cool→go */
      case go: case pushgo: case trap: case resume: case syncid: inst_ptr.p = UNKNOWN_SPEC; break;
      case trip: inst_ptr = zero_spec; break;
      }
    }
  }
```

This code is used in section 75.

86. At any given time the simulated machine is in two main states, the "hot state" corresponding to instructions that have been committed and the "cool state" corresponding to all the speculative changes currently being considered. The dispatcher works with cool instructions and puts them into the reorder buffer, where they gradually get warmer and warmer. Intermediate instructions, between *hot* and *cool*, have intermediate temperatures.

A machine register like l[101] or g[250] is represented by a specnode whose *o* field is the current hot value of the register. If the *up* and *down* fields of this specnode point to the node itself, the hot and cool values of the register are identical. Otherwise *up* and *down* are pointers to the coolest and hottest ends of a doubly linked list of specnodes, representing intermediate speculative values (sometimes called "rename registers"). The rename registers are implemented as the *x* or *a* specnodes inside control blocks, for speculative instructions that use this register as a destination. Speculative instructions that use the register as a source operand point to the next-hottest specnode on the list, until the value becomes known. The doubly linked list of specnodes is an input-restricted deque: A node is inserted at the cool end when the dispatcher issues an instruction with this register as destination; a node is removed from the cool end if an instruction needs to be deissued; a node is removed from the hot end when an instruction is committed.

The special registers rA, rB, ... occupy the same array as the global registers g[32], g[33], ... For example, rB is internally the same as g[0], because $rB = 0$.

```
⟨ External variables 4 ⟩ +≡
  Extern specnode g[256];      /* global registers and special registers */
  Extern specnode *l;      /* the ring of local registers */
  Extern int lring_size;      /* the number of on-chip local registers (must be a power of 2) */
  Extern int max_rename_regs, max_mem_slots;      /* capacity of reorder buffer */
  Extern int rename_regs, mem_slots;      /* currently unused capacity */
```

87. Special register rC was the clock in the original definition of MMIX. But now the clock is just an external variable, called *ticks*.

```
⟨ External variables 4 ⟩ +≡
  Extern octa ticks;      /* the internal clock */
```

88. ⟨ Global variables 20 ⟩ +≡
```
int lring_mask;      /* for calculations modulo lring_size */
```

89. The *addr* fields in the specnode lists for registers are used to identify that register in diagnostic messages. Such addresses are negative; memory addresses are positive.

All registers are initially zero except rG, which is initially 255, and rN, which has a constant value identifying the time of compilation. (The macro `ABSTIME` is defined externally in the file `abstime.h`, which should have just been created by ABSTIME; ABSTIME is a trivial program that computes the value of the standard library function *time*(Λ). We assume that this number, which is the number of seconds in the "UNIX epoch," is less than 2^{32}. Beware: Our assumption will fail in February of 2106.)

```
#define VERSION 1      /* version of the MMIX architecture that we support */
#define SUBVERSION 0      /* secondary byte of version number */
#define SUBSUBVERSION 2      /* further qualification to version number */
```

⟨ Initialize everything 22 ⟩ +≡
```
rename_regs = max_rename_regs;
mem_slots = max_mem_slots;
lring_mask = lring_size - 1;
for (j = 0; j < 256; j++) {
  g[j].addr.h = sign_bit, g[j].addr.l = j, g[j].known = true;
  g[j].up = g[j].down = &g[j];
}
g[rG].o.l = 255;
g[rN].o.h = (VERSION ≪ 24) + (SUBVERSION ≪ 16) + (SUBSUBVERSION ≪ 8);
g[rN].o.l = ABSTIME;      /* see comment and warning above */
for (j = 0; j < lring_size; j++) {
  l[j].addr.h = sign_bit, l[j].addr.l = 256 + j, l[j].known = true;
  l[j].up = l[j].down = &l[j];
}
```

90. ⟨ Internal prototypes 13 ⟩ +≡
```
static void print_specnode_id ARGS((octa));
```

91. ⟨ Subroutines 14 ⟩ +≡
```
static void print_specnode_id(a)
    octa a;
{
  if (a.h ≡ sign_bit) {
    if (a.l < 32) printf("%s", special_name[a.l]);
    else if (a.l < 256) printf("g[%d]", a.l);
    else printf("l[%d]", a.l - 256);
  } else if (a.h ≠ (tetra) -1) {
    printf("m["); print_octa(a); printf("]");
  }
}
```

92. The *specval* subroutine produces a **spec** corresponding to the currently coolest value of a given local or global register.

⟨ Internal prototypes 13 ⟩ +≡
```
static spec specval ARGS((specnode *));
```

93. ⟨ Subroutines 14 ⟩ +≡
```
static spec specval(r)
    specnode *r;
{ spec res;
  if (r→up→known) res.o = r→up→o, res.p = Λ;
  else res.p = r→up;
  return res;
}
```

94. The *spec_install* subroutine introduces a new speculative value at the cool end of a given doubly linked list.

⟨ Internal prototypes 13 ⟩ +≡
```
static void spec_install ARGS((specnode *, specnode *));
```

95. ⟨ Subroutines 14 ⟩ +≡
```
static void spec_install(r, t)      /* insert t into list r */
    specnode *r, *t;
{
  t→up = r→up;
  t→up→down = t;
  r→up = t;
  t→down = r;
  t→addr = r→addr;
}
```

96. Conversely, *spec_rem* takes such a value out.

⟨ Internal prototypes 13 ⟩ +≡
```
static void spec_rem ARGS((specnode *));
```

97. ⟨ Subroutines 14 ⟩ +≡
```
static void spec_rem(t)      /* remove t from its list */
    specnode *t;
{ register specnode *u = t→up, *d = t→down;
  u→down = d; d→up = u;
}
```

98. Some special registers are so central to MMIX's operation, they are carried along with each control block in the reorder buffer instead of being treated as source and destination registers of each instruction. For example, the register stack pointers rO and rS are treated in this way. The normal specnodes for rO and rS, namely $g[rO]$ and $g[rS]$, are not actually used; the cool values are called *cool_O* and *cool_S*. (Actually *cool_O* and *cool_S* correspond to the register values divided by 8, since rO and rS are always multiples of 8.)

The arithmetic status register, rA, is also treated specially. Its event bits are kept up to date only at the "hot" end, by accumulating values of *arith_exc*; an instruction to GET the value of rA will be executed only in the hot seat. The other bits of rA, which are needed to control trip handlers and floating point rounding, are treated in the normal way.

⟨ External variables 4 ⟩ +≡
```
Extern octa cool_O, cool_S;      /* values of rO, rS before the cool instruction */
```

99. ⟨ Global variables 20 ⟩ +≡
```
unsigned int cool_L, cool_G;      /* values of rL and rG before the cool instruction */
unsigned int cool_hist, peek_hist;      /* history bits for branch prediction */
octa new_O, new_S;      /* values of rO, rS after cool */
```

100. ⟨ Install default fields in the *cool* block 100 ⟩ ≡
```
cool→op = op; cool→i = i;
cool→xx = (head→inst ≫ 16) & #ff; cool→yy = (head→inst ≫ 8) & #ff; cool→zz = (head→inst) & #ff;
cool→loc = head→loc;
cool→y = cool→z = cool→b = cool→ra = zero_spec;
cool→x.o = cool→a.o = cool→rl.o = zero_octa;
cool→x.known = false;
cool→x.up = Λ;
cool→a.known = false;
cool→a.up = Λ;
cool→rl.known = true;
cool→rl.up = Λ;
cool→need_b = cool→need_ra = cool→ren_x = cool→mem_x = cool→ren_a = cool→set_l = false;
cool→arith_exc = cool→denin = cool→denout = 0;
if ((head→loc.h & sign_bit) ∧ ¬(g[rU].o.h & #8000)) cool→usage = false;
else cool→usage = ((op & (g[rU].o.h ≫ 16)) ≡ g[rU].o.h ≫ 24 ? true : false);
new_O = cool→cur_O = cool_O; new_S = cool→cur_S = cool_S;
cool→interrupt = head→interrupt;
cool→hist = peek_hist;
cool→go.o = incr(cool→loc, 4);
cool→go.known = false, cool→go.addr.h = −1, cool→go.up = (specnode *) cool;
cool→interim = cool→stack_alert = false;
```
This code is used in section 75.

101. ⟨ Dispatch an instruction to the *cool* block if possible, otherwise **goto** *stall* 101 ⟩ ≡
```
if (new_cool ≡ hot) goto stall;      /* reorder buffer is full */
⟨ Make sure cool_L and cool_G are up to date 102 ⟩;
⟨ Install the operand fields of the cool block 103 ⟩;
if (f & X_is_dest_bit) ⟨ Install register X as the destination, or insert an internal command and goto
      dispatch_done if X is marginal 110 ⟩;
switch (i) {
  ⟨ Special cases of instruction dispatch 117 ⟩
default: break;
}
dispatch_done:
```
This code is used in section 75.

102. The `UNSAVE` operation begins by loading register rG from memory. We don't really need to know the value of rG until twelve other registers have been unsaved, so we aren't fussy about it here.

⟨ Make sure *cool_L* and *cool_G* are up to date 102 ⟩ ≡
```
if (¬g[rL].up→known) goto stall;
cool_L = g[rL].up→o.l;
if (¬g[rG].up→known ∧ ¬(op ≡ UNSAVE ∧ cool→xx ≡ 1)) goto stall;
cool_G = g[rG].up→o.l;
```
This code is used in section 101.

103. ⟨ Install the operand fields of the *cool* block 103 ⟩ ≡

```
if (resuming) ⟨ Insert special operands when resuming an interrupted operation 324 ⟩
else {
  if (f & #10) ⟨ Set cool→b from register X 106 ⟩
  if (third_operand[op] ∧ (cool→i ≠ trap)) ⟨ Set cool→b and/or cool→ra from special register 108 ⟩;
  if (f & #1) cool→z.o.l = cool→zz;
  else if (f & #2) ⟨ Set cool→z from register Z 104 ⟩
  else if ((op & #f0) ≡ #e0) ⟨ Set cool→z as an immediate wyde 109 ⟩;
  if (f & #4) cool→y.o.l = cool→yy;
  else if (f & #8) ⟨ Set cool→y from register Y 105 ⟩
}
```

This code is used in section 101.

104. ⟨ Set *cool→z* from register Z 104 ⟩ ≡

```
{
  if (cool→zz ≥ cool_G) cool→z = specval(&g[cool→zz]);
  else if (cool→zz < cool_L) cool→z = specval(&l[(cool_O.l + cool→zz) & lring_mask]);
}
```

This code is used in section 103.

105. ⟨ Set *cool→y* from register Y 105 ⟩ ≡

```
{
  if (cool→yy ≥ cool_G) cool→y = specval(&g[cool→yy]);
  else if (cool→yy < cool_L) cool→y = specval(&l[(cool_O.l + cool→yy) & lring_mask]);
}
```

This code is used in section 103.

106. ⟨ Set *cool→b* from register X 106 ⟩ ≡

```
{
  if (cool→xx ≥ cool_G) cool→b = specval(&g[cool→xx]);
  else if (cool→xx < cool_L) cool→b = specval(&l[(cool_O.l + cool→xx) & lring_mask]);
  if (f & rel_addr_bit) cool→need_b = true;      /* br, pbr */
}
```

This code is used in section 103.

107. If an operation requires a special register as third operand, that register is listed in the *third_operand* table.

```
⟨ Global variables 20 ⟩ +≡
  unsigned char third_operand[256] = {
      0, rA, 0, 0, rA, rA, rA, rA,      /* TRAP, ... */
      rA, rA, rA, rA, rA, rA, rA, rA,      /* FLOT, ... */
      rA, rE, rE, rE, rA, rA, rA, rA,      /* FMUL, ... */
      rA, rA, 0, 0, rA, rA, rD, rD,      /* MUL, ... */
      rA, rA, 0, 0, rA, rA, 0, 0,      /* ADD, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* 2ADDU, ... */
      0, 0, 0, 0, rA, rA, 0, 0,      /* CMP, ... */
      rA, rA, 0, 0, 0, 0, 0, 0,      /* SL, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* BN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* BNN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* PBN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* PBNN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* CSN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* CSNN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* ZSN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* ZSNN, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* LDB, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* LDT, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* LDSF, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* LDVTS, ... */
      rA, rA, 0, 0, rA, rA, 0, 0,      /* STB, ... */
      rA, rA, 0, 0, 0, 0, 0, 0,      /* STT, ... */
      rA, rA, 0, 0, 0, 0, 0, 0,      /* STSF, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* SYNCD, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* OR, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* AND, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* BDIF, ... */
      rM, rM, 0, 0, 0, 0, 0, 0,      /* MUX, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* SETH, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* ORH, ... */
      0, 0, 0, 0, 0, 0, 0, 0,      /* JMP, ... */
      rJ, 0, 0, 0, 0, 0, 0, 255};      /* POP, ... */
```

108. The *cool→b* field is busy in operations like `STB` or `STSF`, which need rA. So we use *cool→ra* instead, when rA is needed.

```
⟨ Set cool→b and/or cool→ra from special register 108 ⟩ ≡
  {
    if (third_operand[op] ≡ rA ∨ third_operand[op] ≡ rE)  cool→need_ra = true, cool→ra = specval(&g[rA]);
    if (third_operand[op] ≠ rA)  cool→need_b = true, cool→b = specval(&g[third_operand[op]]);
  }
```

This code is used in section 103.

109. ⟨ Set *cool→z* as an immediate wyde 109 ⟩ ≡

```
{
  switch (op & 3) {
  case 0: cool→z.o.h = yz ≪ 16; break;
  case 1: cool→z.o.h = yz; break;
  case 2: cool→z.o.l = yz ≪ 16; break;
  case 3: cool→z.o.l = yz; break;
  }
  if (i ≠ set) {      /* register X should also be the Y operand */
    cool→y = cool→b;
    cool→b = zero_spec;
  }
}
```

This code is used in section 103.

110. ⟨ Install register X as the destination, or insert an internal command and **goto** *dispatch_done* if X is marginal 110 ⟩ ≡

```
{
  if (cool→xx ≥ cool_G) {
    if (i ≠ pushgo ∧ i ≠ pushj ∧ i ≠ cswap) cool→ren_x = true, spec_install(&g[cool→xx], &cool→x);
  } else if (cool→xx < cool_L) {
    if (i ≠ cswap) cool→ren_x = true, spec_install(&l[(cool_O.l + cool→xx) & lring_mask], &cool→x);
  } else {      /* we need to increase L before issuing head→inst */
  increase_L: if (((cool_S.l − cool_O.l − cool_L − 1) & lring_mask) ≡ 0)
      ⟨ Insert an instruction to advance gamma 113 ⟩
    else ⟨ Insert an instruction to advance beta and L 112 ⟩;
  }
}
```

This code is used in section 101.

111. ⟨ Check for sufficient rename registers and memory slots, or **goto** *stall* 111 ⟩ ≡

```
if (rename_regs < (cool→ren_x ? 1 : 0) + (cool→ren_a ? 1 : 0)) goto stall;
if (cool→mem_x) {
  if (mem_slots) mem_slots−−; else goto stall;
}
rename_regs −= (cool→ren_x ? 1 : 0) + (cool→ren_a ? 1 : 0);
```

This code is used in section 75.

112. The *incrl* instruction advances β and rL by 1 at a time when we know that $\beta \neq \gamma$, in the ring of local registers.

⟨ Insert an instruction to advance beta and L 112 ⟩ ≡

```
{
  cool→i = incrl;
  spec_install(&l[(cool_O.l + cool_L) & lring_mask], &cool→x);
  cool→need_b = cool→need_ra = false;
  cool→y = cool→z = zero_spec;
  cool→x.known = true;      /* cool→x.o = zero_octa */
  spec_install(&g[rL], &cool→rl);
  cool→rl.o.l = cool_L + 1;
  cool→ren_x = cool→set_l = true;
  op = SETH;      /* this instruction to be handled by the simplest units */
  cool→interim = true;
  goto dispatch_done;
}
```

This code is used in section 110.

113. The *incgamma* instruction advances γ and rS by storing an octabyte from the local register ring to virtual memory location $cool\_S \ll 3$.

⟨ Insert an instruction to advance gamma 113 ⟩ ≡

```
{
  cool→need_b = cool→need_ra = false;
  cool→i = incgamma;
  new_S = incr(cool_S, 1);
  cool→b = specval(&l[cool_S.l & lring_mask]);
  cool→y.p = Λ, cool→y.o = shift_left(cool_S, 3);
  cool→z = zero_spec;
  cool→mem_x = true, spec_install(&mem, &cool→x);
  op = STOU;      /* this instruction needs to be handled by load/store unit */
  cool→interim = true;
  cool→stack_alert = ¬(cool→y.o.h & sign_bit);
  goto dispatch_done;
}
```

This code is used in sections 110, 119, and 337.

114. The *decgamma* instruction decreases γ and rS by loading an octabyte from virtual memory location $(cool\_S - 1) \ll 3$ into the local register ring. The value of β may need to be decreased too (by decreasing rL).

```
⟨ Insert an instruction to decrease gamma 114 ⟩ ≡
  {
    if (cool_O.l + cool_L ≡ cool_S.l + lring_size) {      /* don't let γ pass β */
      if (cool→i ≡ pop ∧ cool→xx ≡ cool_L ∧ cool_L > 1) {
        cool→i = or;      /* we'll preserve the main result by moving it down */
        head→inst -= #10000;      /* decrease X field of POP in fetch buffer */
        op = OR;
        cool→y = specval(&l[(cool_O.l + cool→xx - 1) & lring_mask]);
        spec_install(&l[(cool_O.l + cool→xx - 2) & lring_mask], &cool→x);
      } else {      /* decrease rL by 1 */
        spec_install(&g[rL], &cool→rl);
        cool→rl.o.l = cool_L - 1;
        cool→set_l = true;
      }
    }
    if (cool→i ≠ or) {
      cool→i = decgamma;
      new_S = incr(cool_S, -1);
      cool→y.p = Λ, cool→y.o = shift_left(new_S, 3);
      spec_install(&l[new_S.l & lring_mask], &cool→x);
      op = LDOU;      /* this instruction needs to be handled by load/store unit */
      cool→ptr_a = (void *) mem.up;
    }
    cool→z = cool→b = zero_spec;
    cool→need_b = false;
    cool→ren_x = cool→interim = true;
    goto dispatch_done;
  }
```

This code is used in section 120.

115. Storing into memory requires a doubly linked data list of specnodes like the lists we use for local and global registers. In this case the head of the list is called *mem*, and the *addr* fields are physical addresses in memory.

```
⟨ External variables 4 ⟩ +≡
  Extern specnode mem;
```

116. The *addr* field of a memory specnode is all 1s until the physical address has been computed.

```
⟨ Initialize everything 22 ⟩ +≡
  mem.addr.h = mem.addr.l = -1;
  mem.up = mem.down = &mem;
```

117. The CSWAP operation is treated as a partial store, with $X as a secondary output. Partial store (*pst*) commands read an octabyte from memory before they write it.

```
⟨ Special cases of instruction dispatch 117 ⟩ ≡
case cswap: cool→ren_a = true;
  spec_install(cool→xx ≥ cool_G ? &g[cool→xx] : &l[(cool_O.l + cool→xx) & lring_mask], &cool→a);
  cool→i = pst;
case st: if ((op & #fe) ≡ STCO) cool→b.o.l = cool→xx;
case pst: cool→mem_x = true, spec_install(&mem, &cool→x); break;
case ld: case ldunc: cool→ptr_a = (void *) mem.up; break;
```

See also sections 118, 119, 120, 121, 122, 227, 312, 322, 332, 337, 347, and 355.

This code is used in section 101.

118. When new data is PUT into special registers 8 or 15–20 (namely rC, rK, rQ, rU, rV, rG, or rL) it can affect many things. Therefore we stop issuing further instructions until such PUTs are committed. Moreover, we will see later that such drastic PUTs defer execution until they reach the hot seat.

```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case put: if (cool→yy ≠ 0 ∨ cool→xx ≥ 32) goto illegal_inst;
  if (cool→xx ≥ 8) {
    if (cool→xx ≤ 11 ∧ cool→xx ≠ 8) goto illegal_inst;
    if (cool→xx ≤ 18 ∧ ¬(cool→loc.h & sign_bit)) goto privileged_inst;
  }
  if (cool→xx ≡ 8 ∨ (cool→xx ≥ 15 ∧ cool→xx ≤ 20)) freeze_dispatch = true;
  cool→ren_x = true, spec_install(&g[cool→xx], &cool→x); break;
case get: if (cool→yy ∨ cool→zz ≥ 32) goto illegal_inst;
  if (cool→zz ≡ rO) cool→z.o = shift_left(cool_O, 3);
  else if (cool→zz ≡ rS) cool→z.o = shift_left(cool_S, 3);
  else cool→z = specval(&g[cool→zz]); break;
illegal_inst: cool→interrupt |= B_BIT; goto noop_inst;
case ldvts: if (cool→loc.h & sign_bit) break;
privileged_inst: cool→interrupt |= K_BIT;
noop_inst: cool→i = noop; break;
```

119. A `PUSHGO` instruction with X ≥ G causes L to increase momentarily by 1, even if L = G. But the value of L will be decreased before the `PUSHGO` is complete, so it will never actually exceed G. Moreover, we needn't insert an *incrl* command.

```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case pushgo: inst_ptr.p = &cool→go;
case pushj:
  { register unsigned int x = cool→xx;
    if (x ≥ cool_G) {
      if (((cool_S.l − cool_O.l − cool_L − 1) & lring_mask) ≡ 0)
        ⟨ Insert an instruction to advance gamma 113 ⟩
      x = cool_L; cool_L++;
      cool→ren_x = true, spec_install(&l[(cool_O.l + x) & lring_mask], &cool→x);
    }
    cool→x.known = true, cool→x.o.h = 0, cool→x.o.l = x;
    cool→ren_a = true, spec_install(&g[rJ], &cool→a);
    cool→a.known = true, cool→a.o = incr(cool→loc, 4);
    cool→set_l = true, spec_install(&g[rL], &cool→rl);
    cool→rl.o.l = cool_L − x − 1;
    new_O = incr(cool_O, x + 1);
  } break;
case syncid: if (cool→loc.h & sign_bit) break;
case go: inst_ptr.p = &cool→go; break;
```

120. We need to know the topmost "hidden" element of the register stack when a `POP` instruction is dispatched. This element is usually present in the local register ring, unless $\gamma = \alpha$.

Once it is known, let x be its least significant byte. We will be decreasing rO by $x + 1$, so we may have to decrease γ repeatedly in order to maintain the condition rS ≤ rO.

```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case pop: if (cool→xx ∧ cool_L ≥ cool→xx) cool→y = specval(&l[(cool_O.l + cool→xx − 1) & lring_mask]);
pop_unsave: if (cool_S.l ≡ cool_O.l) ⟨ Insert an instruction to decrease gamma 114 ⟩;
  { register tetra x;
    register unsigned int new_L;
    register specnode *p = l[(cool_O.l − 1) & lring_mask].up;
    if (p→known) x = (p→o.l) & #ff; else goto stall;
    if ((tetra)(cool_O.l − cool_S.l) ≤ x) ⟨ Insert an instruction to decrease gamma 114 ⟩;
    new_O = incr(cool_O, −x − 1);
    if (cool→i ≡ pop) new_L = x + (cool→xx ≤ cool_L ? cool→xx : cool_L + 1);
    else new_L = x;
    if (new_L > cool_G) new_L = cool_G;
    if (x < new_L) cool→ren_x = true, spec_install(&l[(cool_O.l − 1) & lring_mask], &cool→x);
    cool→set_l = true, spec_install(&g[rL], &cool→rl);
    cool→rl.o.l = new_L;
    if (cool→i ≡ pop) {
      cool→z.o.l = yz ≪ 2;
      if (inst_ptr.p ≡ UNKNOWN_SPEC ∧ new_head ≡ tail) inst_ptr.p = &cool→go;
    }
    break;
  }
```

121. ⟨ Special cases of instruction dispatch 117 ⟩ +≡
case *mulu*: *cool→ren_a* = *true*, *spec_install*(&*g*[*rH*], &*cool→a*); **break**;
case *div*: **case** *divu*: *cool→ren_a* = *true*, *spec_install*(&*g*[*rR*], &*cool→a*); **break**;

122. It's tempting to say that we could avoid taking up space in the reorder buffer when no operation needs to be done. A `JMP` instruction qualifies as a no-op in this sense, because the change of control occurs before the execution stage. However, even a no-op might have to be counted in the usage register rU, so it might get into the execution stage for that reason. A no-op can also cause a protection interrupt, if it appears in a negative location. Even more importantly, a program might get into a loop that consists entirely of jumps and no-ops; then we wouldn't be able to interrupt it, because the interruption mechanism needs to find the current location in the reorder buffer! At least one functional unit therefore needs to provide explicit support for `JMP`, `JMPB`, and `SWYM`.

The `SWYM` instruction with `F_BIT` set is a special case: This is a request from the fetch coroutine for an update to the IT-cache, when the page table method isn't implemented in hardware.

⟨ Special cases of instruction dispatch 117 ⟩ +≡
case *noop*: **if** (*cool→interrupt* & `F_BIT`) {
  cool→go.o = *cool→y.o* = *cool→loc*;
  inst_ptr = *specval*(&*g*[*rT*]);
}
break;

123. ⟨ Undo data structures set prematurely in the *cool* block and **break** 123 ⟩ ≡
if (*cool→ren_x* ∨ *cool→mem_x*) *spec_rem*(&*cool→x*);
if (*cool→ren_a*) *spec_rem*(&*cool→a*);
if (*cool→set_l*) *spec_rem*(&*cool→rl*);
if (*inst_ptr.p* ≡ &*cool→go*) *inst_ptr.p* = `UNKNOWN_SPEC`;
break;
This code is used in section 75.

124. The execution stages. MMIX's *raison d'être* is its ability to execute instructions. So now we want to simulate the behavior of its functional units.

Each coroutine scheduled for action at the current tick of the clock has a *stage* number corresponding to a particular subset of the MMIX hardware. For example, the coroutines with $stage = 2$ are the second stages in the pipelines of the functional units. A coroutine with $stage = 0$ works in the fetch unit. Several artificially large stage numbers are used to control special coroutines that do things like write data from buffers into memory.

In this program the current coroutine of interest is called *self*; hence *self*→*stage* is the current stage number of interest. Another key variable, *self*→*ctl*, is called *data*; this is the control block being operated on by the current coroutine. We typically are simulating an operation in which *data*→*x* is being computed as a function of *data*→*y* and *data*→*z*. The *data* record has many fields, as described earlier when we defined **control** structures; for example, *data*→*owner* is the same as *self*, during the execution stage, if it is nonnull.

This part of the simulator is written as if each functional unit is able to handle all 256 operations. In practice, of course, a functional unit tends to be much more specialized; the actual specialization is governed by the dispatcher, which issues an instruction only to a functional unit that supports it. Once an instruction has been dispatched, however, we can simulate it most easily if we imagine that its functional unit is universal.

Coroutines with higher *stage* numbers are processed first. The three most important variables that govern a coroutine's behavior, once *self*→*stage* is given, are the external operation code *data*→*op*, the internal operation code *data*→*i*, and the value of *data*→*state*. We typically have *data*→*state* = 0 when a coroutine is first fired up.

```
⟨ Local variables 12 ⟩ +≡
  register coroutine *self;      /* the current coroutine being executed */
  register control *data;      /* the control block of the current coroutine */
```

125. When a coroutine has done all it wants to on a single cycle, it says **goto** *done*. It will not be scheduled to do any further work unless the *schedule* routine has been called since it began execution. The *wait* macro is a convenient way to say "Please schedule me to resume again at the current *data*→*state*" after a specified time; for example, *wait*(1) will restart a coroutine on the next clock tick.

```
#define wait(t) { schedule(self, t, data→state); goto done; }
#define pass_after(t) schedule(self + 1, t, data→state)
#define sleep { self→next = self; goto done; }      /* wait forever */
#define awaken(c, t) schedule(c, t, c→ctl→state)
```

```
⟨ Execute all coroutines scheduled for the current time 125 ⟩ ≡
  cur_time++; if (cur_time ≡ ring_size) cur_time = 0;
  for (self = queuelist(cur_time); self ≠ &sentinel; self = sentinel.next) {
    sentinel.next = self→next; self→next = Λ;      /* unschedule this coroutine */
    data = self→ctl;
    if (verbose & coroutine_bit) {
      printf("␣running␣"); print_coroutine_id(self); printf("␣");
      print_control_block(data); printf("\n");
    }
    switch (self→stage) {
    case 0: ⟨ Simulate an action of the fetch coroutine 288 ⟩;
    case 1: ⟨ Simulate the first stage of an execution pipeline 130 ⟩;
    default: ⟨ Simulate later stages of an execution pipeline 135 ⟩;
    ⟨ Cases for control of special coroutines 126 ⟩;
    }
  terminate: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  done: ;
  }
```

This code is used in section 64.

126. A special coroutine whose *stage* number is *vanish* simply goes away at its scheduled time.

```
⟨ Cases for control of special coroutines 126 ⟩ ≡
case vanish: goto terminate;
```

See also sections 215, 217, 222, 224, 232, 237, and 257.

This code is used in section 125.

127.
```
⟨ Global variables 20 ⟩ +≡
  coroutine mem_locker;      /* trivial coroutine that vanishes */
  coroutine Dlocker;      /* another */
  control vanish_ctl;      /* such coroutines share a common control block */
```

128.
```
⟨ Initialize everything 22 ⟩ +≡
  mem_locker.name = "Locker";
  mem_locker.ctl = &vanish_ctl;
  mem_locker.stage = vanish;
  Dlocker.name = "Dlocker";
  Dlocker.ctl = &vanish_ctl;
  Dlocker.stage = vanish;
  vanish_ctl.go.o.l = 4;
  for (j = 0; j < DTcache→ports; j++) DTcache→reader[j].ctl = &vanish_ctl;
  if (Dcache)
    for (j = 0; j < Dcache→ports; j++) Dcache→reader[j].ctl = &vanish_ctl;
  for (j = 0; j < ITcache→ports; j++) ITcache→reader[j].ctl = &vanish_ctl;
  if (Icache)
    for (j = 0; j < Icache→ports; j++) Icache→reader[j].ctl = &vanish_ctl;
```

129. Here is a list of the *stage* numbers for special coroutines to be defined below.

```
⟨ Header definitions 6 ⟩ +≡
#define max_stage 99      /* exceeds all stage numbers */
#define vanish 98      /* special coroutine that just goes away */
#define flush_to_mem 97      /* coroutine for flushing from a cache to memory */
#define flush_to_S 96      /* coroutine for flushing from a cache to the S-cache */
#define fill_from_mem 95      /* coroutine for filling a cache from memory */
#define fill_from_S 94      /* coroutine for filling a cache from the S-cache */
#define fill_from_virt 93      /* coroutine for filling a translation cache */
#define write_from_wbuf 92      /* coroutine for emptying the write buffer */
#define cleanup 91      /* coroutine for cleaning the caches */
```

130. At the very beginning of stage 1, a functional unit will stall if necessary until its operands are available. As soon as the operands are all present, the *state* is set nonzero and execution proper begins.

```
⟨ Simulate the first stage of an execution pipeline 130 ⟩ ≡
switch1: switch (data→state) {
  case 0: ⟨ Wait for input data if necessary; set state = 1 if it's there 131 ⟩;
  case 1: ⟨ Begin execution of an operation 132 ⟩;
  case 2: ⟨ Pass data to the next stage of the pipeline 134 ⟩;
  case 3: ⟨ Finish execution of an operation 144 ⟩;
    ⟨ Special cases for states in the first stage 266 ⟩;
  }
```

This code is used in section 125.

131. If some of our input data has been computed by another coroutine on the current cycle, we grab it now but wait for the next cycle. (An actual machine wouldn't have latched the data until then.)

```
⟨ Wait for input data if necessary; set state = 1 if it's there 131 ⟩ ≡
  j = 0;
  if (data→y.p) {
    j++;
    if (data→y.p→known) data→y.o = data→y.p→o, data→y.p = Λ;
    else j += 10;
  }
  if (data→z.p) {
    j++;
    if (data→z.p→known) data→z.o = data→z.p→o, data→z.p = Λ;
    else j += 10;
  }
  if (data→b.p) {
    if (data→need_b) j++;
    if (data→b.p→known) data→b.o = data→b.p→o, data→b.p = Λ;
    else if (data→need_b) j += 10;
  }
  if (data→ra.p) {
    if (data→need_ra) j++;
    if (data→ra.p→known) data→ra.o = data→ra.p→o, data→ra.p = Λ;
    else if (data→need_ra) j += 10;
  }
  if (j < 10) data→state = 1;
  if (j) wait(1);      /* otherwise we fall through to case 1 */
```

This code is used in section 130.

132. Simple register-to-register instructions like `ADD` are assumed to take just one cycle, but others like `FADD` almost certainly require more time. This simulator can be configured so that `FADD` might take, say, four pipeline stages of one cycle each $(1 + 1 + 1 + 1)$, or two pipeline stages of two cycles each $(2 + 2)$, or a single unpipelined stage lasting four cycles (4), etc. In any case the simulator computes the results now, for simplicity, placing them in *data→x* and possibly also in *data→a* and/or *data→interrupt*. The results will not be officially made *known* until the proper time.

```
⟨ Begin execution of an operation 132 ⟩ ≡
  switch (data→i) {
    ⟨ Cases to compute the results of register-to-register operation 137 ⟩;
    ⟨ Cases to compute the virtual address of a memory operation 265 ⟩;
    ⟨ Cases for stage 1 execution 155 ⟩;
  default: ;
  }
  ⟨ Set things up so that the results become known when they should 133 ⟩;
```

This code is used in section 130.

133. If the internal opcode *data*→*i* is *max_pipe_op* or less, a special pipeline sequence like $1+1+1+1$ or $2+2$ or $15+10$, etc., has been configured. Otherwise we assume that the pipeline sequence is simply 1.

Suppose the pipeline sequence is $t_1+t_2+\cdots+t_k$. Each t_j is positive and less than 256, so we represent the sequence as a string *pipe_seq*[*data*→*i*] of unsigned "characters," terminated by 0. Given such a string, we want to do the following: Wait (t_1-1) cycles and pass *data* to stage 2; wait t_2 cycles and pass *data* to stage 3; ...; wait t_{k-1} cycles and pass *data* to stage k; wait t_k cycles and make the results *known*.

The value of *denin* is added to t_1; the value of *denout* is added to t_k.

```
⟨ Set things up so that the results become known when they should 133 ⟩ ≡
  data→state = 3;
  if (data→i ≤ max_pipe_op) { register unsigned char *s = pipe_seq[data→i];
    j = s[0] + data→denin;
    if (s[1]) data→state = 2;      /* more than one stage */
    else j += data→denout;
    if (j > 1) wait(j − 1);
  }
  goto switch1;
```

This code is used in section 132.

134. When we're in stage j, the coroutine for stage $j+1$ of the same functional unit is *self* + 1.

```
⟨ Pass data to the next stage of the pipeline 134 ⟩ ≡
pass_data: if ((self + 1)→next) wait(1);      /* stall if the next stage is occupied */
  { register unsigned char *s = pipe_seq[data→i];
    j = s[self→stage];
    if (s[self→stage + 1] ≡ 0) j += data→denout, data→state = 3;      /* the next stage is the last */
    pass_after(j);
  }
passit: (self + 1)→ctl = data;
  data→owner = self + 1;
  goto done;
```

This code is used in section 130.

135. ⟨ Simulate later stages of an execution pipeline 135 ⟩ ≡

```
switch2: if (data→b.p ∧ data→b.p→known) data→b.o = data→b.p→o, data→b.p = Λ;
  switch (data→state) {
  case 0: panic(confusion("switch2"));
  case 1: ⟨ Begin execution of a stage-two operation 351 ⟩;
  case 2: goto pass_data;
  case 3: goto fin_ex;
    ⟨ Special cases for states in later stages 272 ⟩;
  }
```

This code is used in section 125.

136. The default pipeline times use only one stage; they can be overridden by *MMIX_config*. The total number of stages supported by this simulator is limited to 90, since it must never interfere with the *stage* numbers for special coroutines defined below. (The author doesn't feel guilty about making this restriction.)

```
⟨ External variables 4 ⟩ +≡
#define pipe_limit 90
  Extern unsigned char pipe_seq[max_pipe_op + 1][pipe_limit + 1];
```

137. The simplest of all register-to-register operations is *set*, which occurs for commands like `SETH` as well as for commands like `GETA`. (We might as well start with the easy cases and work our way up.)

⟨ Cases to compute the results of register-to-register operation 137 ⟩ ≡
case *set*: *data→x.o* = *data→z.o*; **break**;

See also sections 138, 139, 140, 141, 142, 143, 343, 344, 345, 346, 348, and 350.

This code is used in section 132.

138. Here are the basic boolean operations, which account for 24 of `MMIX`'s 256 opcodes.

⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case *or*: *data→x.o.h* = *data→y.o.h* | *data→z.o.h*;
data→x.o.l = *data→y.o.l* | *data→z.o.l*;
break;
case *orn*: *data→x.o.h* = *data→y.o.h* | ∼*data→z.o.h*;
data→x.o.l = *data→y.o.l* | ∼*data→z.o.l*;
break;
case *nor*: *data→x.o.h* = ∼(*data→y.o.h* | *data→z.o.h*);
data→x.o.l = ∼(*data→y.o.l* | *data→z.o.l*);
break;
case *and*: *data→x.o.h* = *data→y.o.h* & *data→z.o.h*;
data→x.o.l = *data→y.o.l* & *data→z.o.l*;
break;
case *andn*: *data→x.o.h* = *data→y.o.h* & ∼*data→z.o.h*;
data→x.o.l = *data→y.o.l* & ∼*data→z.o.l*;
break;
case *nand*: *data→x.o.h* = ∼(*data→y.o.h* & *data→z.o.h*);
data→x.o.l = ∼(*data→y.o.l* & *data→z.o.l*);
break;
case *xor*: *data→x.o.h* = *data→y.o.h* ⊕ *data→z.o.h*;
data→x.o.l = *data→y.o.l* ⊕ *data→z.o.l*;
break;
case *nxor*: *data→x.o.h* = *data→y.o.h* ⊕ ∼*data→z.o.h*;
data→x.o.l = *data→y.o.l* ⊕ ∼*data→z.o.l*;
break;

139. The implementation of `ADDU` is only slightly more difficult. It would be trivial except for the fact that internal opcode *addu* is used not only for the `ADDU[I]` and `INC[M][H,L]` operations, in which we simply want to add *data→y.o* to *data→z.o*, but also for operations like `4ADDU`.

⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case *addu*: *data→x.o* = *oplus*((*data→op* & #f8) ≡ #28 ?
shift_left(*data→y.o*, 1 + ((*data→op* ≫ 1) & #3)) : *data→y.o*, *data→z.o*);
break;
case *subu*: *data→x.o* = *ominus*(*data→y.o*, *data→z.o*); **break**;

140. Signed addition and subtraction produce the same results as their unsigned counterparts, but overflow must also be detected. Overflow occurs when adding y to z if and only if y and z have the same sign but their sum has a different sign. Overflow occurs in the calculation $x = y - z$ if and only if it occurs in the calculation $y = x + z$.

```
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case add: data→x.o = oplus(data→y.o, data→z.o);
  if (((data→y.o.h ⊕ data→z.o.h) & sign_bit) ≡ 0 ∧ ((data→y.o.h ⊕ data→x.o.h) & sign_bit) ≠ 0)
    data→interrupt |= V_BIT;
  break;
case sub: data→x.o = ominus(data→y.o, data→z.o);
  if (((data→x.o.h ⊕ data→z.o.h) & sign_bit) ≡ 0 ∧ ((data→y.o.h ⊕ data→x.o.h) & sign_bit) ≠ 0)
    data→interrupt |= V_BIT;
  break;
```

141. The shift commands might take more than one cycle, or they might even be pipelined, if the default value of *pipe_seq*[*sh*] is changed. But we compute shifts all at once here, because other parts of the simulator will take care of the pipeline timing. (Notice that *shlu* is changed to *sh*, for this reason. Similar changes to the internal op codes are made for other operators below.)

```
#define shift_amt (data→z.o.h ∨ data→z.o.l ≥ 64 ? 64 : data→z.o.l)
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case shlu: data→x.o = shift_left(data→y.o, shift_amt); data→i = sh; break;
case shl: data→x.o = shift_left(data→y.o, shift_amt); data→i = sh;
  { octa tmpo;
    tmpo = shift_right(data→x.o, shift_amt, 0);
    if (tmpo.h ≠ data→y.o.h ∨ tmpo.l ≠ data→y.o.l) data→interrupt |= V_BIT;
  } break;
case shru: data→x.o = shift_right(data→y.o, shift_amt, 1); data→i = sh; break;
case shr: data→x.o = shift_right(data→y.o, shift_amt, 0); data→i = sh; break;
```

142. The `MUX` operation has three operands, namely *data*→*y*, *data*→*z*, and *data*→*b*; the third operand is the current (speculative) value of rM, the special mask register. Otherwise `MUX` is unexceptional.

```
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case mux: data→x.o.h = (data→y.o.h & data→b.o.h) + (data→z.o.h & ∼data→b.o.h);
  data→x.o.l = (data→y.o.l & data→b.o.l) + (data→z.o.l & ∼data→b.o.l);
  break;
```

143. Comparisons are a breeze.

```
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case cmp: if ((data→y.o.h & sign_bit) > (data→z.o.h & sign_bit)) goto cmp_neg;
  if ((data→y.o.h & sign_bit) < (data→z.o.h & sign_bit)) goto cmp_pos;
case cmpu: if (data→y.o.h < data→z.o.h) goto cmp_neg;
  if (data→y.o.h > data→z.o.h) goto cmp_pos;
  if (data→y.o.l < data→z.o.l) goto cmp_neg;
  if (data→y.o.l > data→z.o.l) goto cmp_pos;
cmp_zero: break;      /* data→x is zero */
cmp_pos: data→x.o.l = 1; break;      /* data→x.o.h is zero */
cmp_neg: data→x.o = neg_one; break;
```

144. The other operations will be deferred until later, now that we understand the basic ideas. But one more piece of code ought to be written before we move on, because it completes the execution stage for the simple cases already considered.

The *ren_x* and *ren_a* fields tell us whether the x and/or a fields contain valid information that should become officially known.

```
⟨ Finish execution of an operation 144 ⟩ ≡
fin_ex: if (data→ren_x) data→x.known = true;
  else if (data→mem_x) {
    data→x.known = true;
    if (¬(data→x.addr.h & #ffff0000)) data→x.addr.l &= -8;
  }
  if (data→ren_a) data→a.known = true;
  if (data→loc.h & sign_bit) data→ra.o.l = 0;      /* no trips enabled for the operating system */
  if (data→interrupt & #ffff) ⟨ Handle interrupt at end of execution stage 307 ⟩;
die: data→owner = Λ; goto terminate;      /* this coroutine now fades away */
```

This code is used in section 130.

145. The commission/deissue stage. Control blocks leave the reorder buffer either at the hot end (when they're committed) or at the cool end (when they're deissued). We hope most of them are committed, but from time to time our speculation is incorrect and we must deissue a sequence of instructions that prove to be unwanted. Deissuing must take priority over committing, because the dispatcher cannot do anything until the machine's cool state has stabilized.

Deissuing changes the cool state by undoing the most recently issued instructions, in reverse order. Committing changes the hot state by doing the least recently issued instructions, in their original order. Both operations are similar, so we assume that they take the same time; at most *commit_max* instructions are deissued and/or committed on each clock cycle.

```
⟨ Deissue the coolest instruction 145 ⟩ ≡
  {
    cool = (cool ≡ reorder_top ? reorder_bot : cool + 1);
    if (verbose & issue_bit) {
      printf("Deissuing␣"); print_control_block(cool);
      if (cool→owner) { printf("␣"); print_coroutine_id(cool→owner); }
      printf("\n");
    }
    if (cool→ren_x) rename_regs++, spec_rem(&cool→x);
    if (cool→ren_a) rename_regs++, spec_rem(&cool→a);
    if (cool→mem_x) mem_slots++, spec_rem(&cool→x);
    if (cool→set_l) spec_rem(&cool→rl);
    if (cool→owner) {
      if (cool→owner→lockloc) *(cool→owner→lockloc) = Λ, cool→owner→lockloc = Λ;
      if (cool→owner→next) unschedule(cool→owner);
    }
    cool_O = cool→cur_O; cool_S = cool→cur_S;
    deissues--;
  }
```

This code is used in section 67.

146. ⟨ Commit the hottest instruction, or **break** if it's not ready 146 ⟩ ≡

```
{
  if (nullifying) ⟨ Nullify the hottest instruction 147 ⟩
  else {
    if (hot→i ≡ get ∧ hot→zz ≡ rQ) new_Q = oandn(g[rQ].o, hot→x.o);
    else if (hot→i ≡ put ∧ hot→xx ≡ rQ) hot→x.o.h |= new_Q.h, hot→x.o.l |= new_Q.l;
    if (hot→mem_x) ⟨ Commit to memory if possible, otherwise break 256 ⟩;
    if (hot→stack_alert) stack_overflow = true;
    else if (stack_overflow ∧ ¬hot→interim) {
      g[rQ].o.l |= STACK_OVERFLOW, new_Q.l |= STACK_OVERFLOW, stack_overflow = false;
      if (verbose & issue_bit) {
        printf("␣setting␣rQ="); print_octa(g[rQ].o); printf("\n");
      }
    }
    if (verbose & issue_bit) {
      printf("Committing␣"); print_control_block(hot); printf("\n");
    }
    if (hot→ren_x) rename_regs++, hot→x.up→o = hot→x.o, spec_rem(&(hot→x));
    if (hot→ren_a) rename_regs++, hot→a.up→o = hot→a.o, spec_rem(&(hot→a));
    if (hot→set_l) hot→rl.up→o = hot→rl.o, spec_rem(&(hot→rl));
    if (hot→arith_exc) g[rA].o.l |= hot→arith_exc;
    if (hot→usage) {
      g[rU].o.l++; if (g[rU].o.l ≡ 0) {
        g[rU].o.h++; if ((g[rU].o.h & #7fff) ≡ 0) g[rU].o.h -= #8000;
      }
    }
  }
  if (hot→interrupt ≥ H_BIT) ⟨ Begin an interruption and break 317 ⟩;
}
```

This code is used in section 67.

147. A load or store instruction is "nullified" if it is about to be captured by a trap interrupt. In such cases it will be the only item in the reorder buffer; thus nullifying is sort of a cross between deissuing and committing. (It is important to have stopped dispatching when nullification is necessary, because instructions such as *incgamma* and *decgamma* change rS, and we need to change it back when an unexpected interruption occurs.)

⟨ Nullify the hottest instruction 147 ⟩ ≡

```
{
  if (verbose & issue_bit) {
    printf("Nullifying␣"); print_control_block(hot); printf("\n");
  }
  if (hot→ren_x) rename_regs++, spec_rem(&hot→x);
  if (hot→ren_a) rename_regs++, spec_rem(&hot→a);
  if (hot→mem_x) mem_slots++, spec_rem(&hot→x);
  if (hot→set_l) spec_rem(&hot→rl);
  cool_O = hot→cur_O, cool_S = hot→cur_S;
  nullifying = false;
}
```

This code is used in section 146.

148. Interrupt bits in rQ might be lost if they are set between a `GET` and a `PUT`. Therefore we don't allow `PUT` to zero out bits that have become 1 since the most recently committed `GET`.

```
⟨ Global variables 20 ⟩ +≡
  octa new_Q;      /* when rQ increases in any bit position, so should this */
  bool stack_overflow;      /* stack overflow not yet reported */
```

149. An instruction will not be committed immediately if it violates the basic security rule of `MMIX`: An instruction in a nonnegative location should not be performed unless all eight of the internal interrupts have been enabled in the interrupt mask register rK. Conversely, an instruction in a negative location should not be performed if the `P_BIT` is enabled in rK.

Such instructions take one extra cycle before they are committed. The nonnegative-location case turns on the `S_BIT` of both rK and rQ, leading to an immediate interrupt (unless the current instruction is *trap*, *put*, or *resume*).

```
⟨ Check for security violation, break if so 149 ⟩ ≡
  {
    if (hot→loc.h & sign_bit) {
      if ((g[rK].o.h & P_BIT) ∧ ¬(hot→interrupt & P_BIT)) {
        hot→interrupt |= P_BIT;
        g[rQ].o.h |= P_BIT;
        new_Q.h |= P_BIT;
        if (verbose & issue_bit) {
          printf("␣setting␣rQ="); print_octa(g[rQ].o); printf("\n");
        }
        break;
      }
    } else if ((g[rK].o.h & #ff) ≠ #ff ∧ ¬(hot→interrupt & S_BIT)) {
      hot→interrupt |= S_BIT;
      g[rQ].o.h |= S_BIT;
      new_Q.h |= S_BIT;
      g[rK].o.h |= S_BIT;
      if (verbose & issue_bit) {
        printf("␣setting␣rQ="); print_octa(g[rQ].o);
        printf(",␣rK="); print_octa(g[rK].o); printf("\n");
      }
      break;
    }
  }
```

This code is used in section 67.

150. Branch prediction. An `MMIX` programmer distinguishes statically between "branches" and "probable branches," but many modern computers attempt to do better by implementing dynamic branch prediction. (See, for example, section 4.3 of Hennessy and Patterson's *Computer Architecture*, second edition.) Experience has shown that dynamic branch prediction can significantly improve the performance of speculative execution, by reducing the number of instructions that need to be deissued.

This simulator has an optional *bp_table* containing 2^{a+b+c} entries of n bits each, where n is between 1 and 8. Usually n is 1 or 2 in practice, but 8 bits are allocated per entry for convenience in this program. The *bp_table* is consulted and updated on every branch instruction (every `B` or `PB` instruction, but not `JMP`), for advice on past history of similar situations. It is indexed by the a least significant bits of the address of the instruction, the b most recent bits of global branch history, and the next c bits of both address and history (exclusive-ored).

A *bp_table* entry begins at zero and is regarded as a signed n-bit number. If it is nonnegative, we will follow the prediction in the instruction, namely to predict a branch taken only in the `PB` case. If it is negative, we will predict the opposite of the instruction's recommendation. The n-bit number is increased (if possible) if the instruction's prediction was correct, decreased (if possible) if the instruction's prediction was incorrect.

(Incidentally, a large value of n is not necessarily a good idea. For example, if $n = 8$ the machine might need 128 steps to recognize that a branch taken the first 150 times is not taken the next 150 times. And if we modify the update criteria to avoid this problem, we obtain a scheme that is rarely better than a simple scheme with smaller n.)

The values a, b, c, and n in this discussion are called *bp_a*, *bp_b*, *bp_c*, and *bp_n* in the program.

```
⟨ External variables 4 ⟩ +≡
  Extern int bp_a, bp_b, bp_c, bp_n;      /* parameters for branch prediction */
  Extern char *bp_table;      /* either Λ or an array of 2^{a+b+c} items */
```

151. Branch prediction is made when we are either about to issue an instruction or peeking ahead. We look at the *bp_table*, but we don't want to update it yet.

```
⟨ Predict a branch outcome 151 ⟩ ≡
  {
    predicted = op & #10;      /* start with the instruction's recommendation */
    if (bp_table) { register int h;
      m = ((head→loc.l & bp_cmask) ≪ bp_b) + (head→loc.l & bp_amask);
      m = ((cool_hist & bp_bcmask) ≪ bp_a) ⊕ (m ≫ 2);
      h = bp_table[m];
      if (h & bp_npower) predicted ⊕= #10;
    }
    if (predicted) peek_hist = (peek_hist ≪ 1) + 1;
    else peek_hist ≪= 1;
  }
```

This code is used in section 85.

152. We update the *bp_table* when an instruction is issued. And we store the opposite table value in *cool→x.o.l*, just in case our prediction turns out to be wrong.

```
⟨ Record the result of branch prediction 152 ⟩ ≡
  if (bp_table) { register int reversed, h, h_up, h_down;
    reversed = op & #10;
    if (peek_hist & 1) reversed ⊕= #10;
    m = ((head→loc.l & bp_cmask) ≪ bp_b) + (head→loc.l & bp_amask);
    m = ((cool_hist & bp_bcmask) ≪ bp_a) ⊕ (m ≫ 2);
    h = bp_table[m];
    h_up = (h + 1) & bp_nmask; if (h_up ≡ bp_npower) h_up = h;
    if (h ≡ bp_npower) h_down = h; else h_down = (h − 1) & bp_nmask;
    if (reversed) {
      bp_table[m] = h_down, cool→x.o.l = h_up;
      cool→i = pbr + br − cool→i;      /* reverse the sense */
      bp_rev_stat++;
    } else {
      bp_table[m] = h_up, cool→x.o.l = h_down;      /* go with the flow */
      bp_ok_stat++;
    }
    if (verbose & show_pred_bit) {
      printf("␣predicting␣"); print_octa(cool→loc);
      printf("␣%s;␣bp[%x]=%d\n", reversed ? "NG" : "OK", m,
          bp_table[m] − ((bp_table[m] & bp_npower) ≪ 1));
    }
    cool→x.o.h = m;
  }
```

This code is used in section 75.

153. The calculations in the previous sections need several precomputed constants, depending on the parameters a, b, c, and n.

```
⟨ Initialize everything 22 ⟩ +≡
  bp_amask = ((1 ≪ bp_a) − 1) ≪ 2;      /* least a bits of instruction address */
  bp_cmask = ((1 ≪ bp_c) − 1) ≪ (bp_a + 2);      /* the next c address bits */
  bp_bcmask = (1 ≪ (bp_b + bp_c)) − 1;      /* least b + c bits of history info */
  bp_nmask = (1 ≪ bp_n) − 1;      /* least significant n bits */
  bp_npower = 1 ≪ (bp_n − 1);      /* 2^(n−1), the sign bit of an n-bit number */
```

154.
```
⟨ Global variables 20 ⟩ +≡
  int bp_amask, bp_cmask, bp_bcmask, bp_nmask, bp_npower;
  int bp_rev_stat, bp_ok_stat;      /* how often we overrode and agreed */
  int bp_bad_stat, bp_good_stat;      /* how often we failed and succeeded */
```

155. After a branch or probable branch instruction has been issued and the value of the relevant register has been computed in the reorder buffer as *data→b.o*, we're ready to determine if the prediction was correct or not.

```
⟨ Cases for stage 1 execution 155 ⟩ ≡
case br: case pbr: j = register_truth(data→b.o, data→op);
  if (j) data→go.o = data→z.o; else data→go.o = data→y.o;
  if (j ≡ (data→i ≡ pbr)) bp_good_stat++;
  else { /* oops, misprediction */
    bp_bad_stat++;
    ⟨ Recover from incorrect branch prediction 160 ⟩;
  }
  goto fin_ex;
```

See also sections 313, 325, 327, 328, 329, 331, and 356.

This code is used in section 132.

156. The *register_truth* subroutine is used by `B`, `PB`, `CS`, and `ZS` commands to decide whether an octabyte satisfies the conditions of the opcode, *data→op*.

```
⟨ Internal prototypes 13 ⟩ +≡
  static int register_truth ARGS((octa, mmix_opcode));
```

157.

```
⟨ Subroutines 14 ⟩ +≡
  static int register_truth(o, op)
      octa o;
      mmix_opcode op;
  { register int b;
    switch ((op ≫ 1) & #3) {
    case 0: b = o.h ≫ 31; break;      /* negative? */
    case 1: b = (o.h ≡ 0 ∧ o.l ≡ 0); break;      /* zero? */
    case 2: b = (o.h < sign_bit ∧ (o.h ∨ o.l)); break;      /* positive? */
    case 3: b = o.l & #1; break;      /* odd? */
    }
    if (op & #8) return b ⊕ 1;
    else return b;
  }
```

158. The *issued_between* subroutine determines how many speculative instructions were issued between a given control block in the reorder buffer and the current *cool* pointer, when *cc* = *cool*.

```
⟨ Internal prototypes 13 ⟩ +≡
  static int issued_between ARGS((control *, control *));
```

159.

```
⟨ Subroutines 14 ⟩ +≡
  static int issued_between(c, cc)
      control *c, *cc;
  {
    if (c > cc) return c − 1 − cc;
    return (c − reorder_bot) + (reorder_top − cc);
  }
```

160. If more than one functional unit is able to process branch instructions and if two of them simultaneously discover misprediction, or if misprediction is detected by one unit just as another unit is generating an interrupt, we assume that an arbitration takes place so that only the hottest one actually deissues the cooler instructions.

Changes to the *bp_table* aren't undone when they were made on speculation in an instruction being deissued; nor do we worry about cases where the same *bp_table* entry is being updated by two or more active coroutines. After all, the *bp_table* is just a heuristic, not part of the real computation. We correct the *bp_table* only if we discover that a prediction was wrong, so that we will be less likely to make the same mistake later.

```
⟨ Recover from incorrect branch prediction 160 ⟩ ≡
  i = issued_between(data, cool);
  if (i < deissues) goto die;
  deissues = i;
  old_tail = tail = head; resuming = 0;      /* clear the fetch buffer */
  ⟨ Restart the fetch coroutine 287 ⟩;
  inst_ptr.o = data→go.o, inst_ptr.p = Λ;
  if (¬(data→loc.h & sign_bit)) {
    if (inst_ptr.o.h & sign_bit) data→interrupt |= P_BIT;
    else data→interrupt &= ∼P_BIT;
  }
  if (bp_table) {
    bp_table[data→x.o.h] = data→x.o.l;      /* this is what we should have stored */
    if (verbose & show_pred_bit) {
      printf("␣mispredicted␣"); print_octa(data→loc);
      printf(";␣bp[%x]=%d\n", data→x.o.h, data→x.o.l − ((data→x.o.l & bp_npower) ≪ 1));
    }
  }
  cool_hist = (j ? (data→hist ≪ 1) + 1 : data→hist ≪ 1);
```

This code is used in section 155.

161. ⟨ External prototypes 9 ⟩ +≡

```
Extern void print_stats ARGS((void));
```

162. ⟨ External routines 10 ⟩ +≡

```
void print_stats()
{
  register int j;
  if (bp_table) printf("Predictions:␣%d␣in␣agreement,␣%d␣in␣opposition;␣%d␣good,␣%d␣bad\n",
      bp_ok_stat, bp_rev_stat, bp_good_stat, bp_bad_stat);
  else printf("Predictions:␣%d␣good,␣%d␣bad\n", bp_good_stat, bp_bad_stat);
  printf("Instructions␣issued␣per␣cycle:\n");
  for (j = 0; j ≤ dispatch_max; j++) printf("␣␣%d␣␣␣%d\n", j, dispatch_stat[j]);
}
```

163. Cache memory. It's time now to consider `MMIX`'s MMU, the memory management unit. This part of the machine deals with the critical problem of getting data to and from the computational units. In a RISC architecture all interaction between main memory and the computer registers is specified by load and store instructions; thus memory accesses are much easier to deal with than they would be on a machine with more complex kinds of interaction. But memory management is still difficult, if we want to do it well, because main memory typically operates at a much slower speed than the registers do. High-speed implementations of `MMIX` introduce intermediate "caches" of storage in order to keep the most important data accessible, and cache maintenance can be complicated when all the details are taken into account. (See, for example, Chapter 5 of Hennessy and Patterson's *Computer Architecture*, second edition.)

This simulator can be configured to have up to three auxiliary caches between registers and memory: An I-cache for instructions, a D-cache for data, and an S-cache for both instructions and data. The S-cache, also called a *secondary cache*, is supported only if both I-cache and D-cache are present. Arbitrary access times for each cache can be specified independently; we might assume, for example, that data items in the I-cache or D-cache can be sent to a register in one or two clock cycles, but the access time for the S-cache might be say 5 cycles, and main memory might require 20 cycles or more. Our speculative pipeline can have many functional units handling load and store instructions, but only one load or store instruction can be updating the D-cache or S-cache or main memory at a time. (However, the D-cache can have several read ports; furthermore, data might be passing between the S-cache and memory while other data is passing between the reorder buffer and the D-cache.)

Besides the optional I-cache, D-cache, and S-cache, there are required caches called the IT-cache and DT-cache, for translation of virtual addresses to physical addresses. A translation cache is often called a "translation lookaside buffer" or TLB; but we call it a cache since it is implemented in nearly the same way as an I-cache.

164. Consider a cache that has blocks of 2^b bytes each and associativity 2^a; here $b \geq 3$ and $a \geq 0$. The I-cache, D-cache, and S-cache are addressed by 48-bit physical addresses, as if they were part of main memory; but the IT and DT caches are addressed by 64-bit keys, obtained from a virtual address by blanking out the lower s bits and inserting the value of n, where the page size s and the process number n are found in rV. We will consider all caches to be addressed by 64-bit keys, so that both cases are handled with the same basic methods.

Given a 64-bit key, we ignore the low-order b bits and use the next c bits to address the *cache set*; then the remaining $64 - b - c$ bits should match one of 2^a *tags* in that set. The case $a = 0$ corresponds to a so-called *direct-mapped* cache; the case $c = 0$ corresponds to a so-called *fully associative* cache. With 2^c sets of 2^a blocks each, and 2^b bytes per block, the cache contains 2^{a+b+c} bytes of data, in addition to the space needed for tags. Translation caches have $b = 3$ and they also usually have $c = 0$.

If a tag matches the specified bits, we "hit" in the cache and can use and/or update the data found there. Otherwise we "miss," and we probably want to replace one of the cache blocks by the block containing the item sought. The item chosen for replacement is called a *victim*. The choice of victim is forced when the cache is direct-mapped, but four strategies for victim selection are available when we must choose from among 2^a entries for $a > 0$:

- "Random" selection chooses the victim by extracting the least significant a bits of the clock.
- "Serial" selection chooses $0, 1, \ldots, 2^a - 1, 0, 1, \ldots, 2^a - 1, 0, \ldots$ on successive trials.
- "LRU (Least Recently Used)" selection chooses the victim that ranks last if items are ranked inversely to the time that has elapsed since their previous use.
- "Pseudo-LRU" selection chooses the victim by a rough approximation to LRU that is simpler to implement in hardware. It requires a bit table $r_1 \ldots r_{2^a-1}$. Whenever we use an item with binary address $(i_1 \ldots i_a)_2$ in the set, we adjust the bit table as follows:

$$r_1 \leftarrow 1 - i_1, \quad r_{1i_1} \leftarrow 1 - i_2, \quad \ldots, \quad r_{1i_1 \ldots i_{a-1}} \leftarrow 1 - i_a;$$

here the subscripts on r are binary numbers. (For example, when $a = 3$, the use of element $(010)_2$ sets $r_1 \leftarrow 1$, $r_{10} \leftarrow 0$, $r_{101} \leftarrow 1$, where r_{101} means the same as r_5.) To select a victim, we start with $l \leftarrow 1$ and then repeatedly set $l \leftarrow 2l + r_l$, a times; then we choose element $l - 2^a$. When $a = 1$, this scheme is equivalent to LRU. When $a = 2$, this scheme was implemented in the Intel 80486 chip.

⟨ Type definitions 11 ⟩ +≡
```
typedef enum {
  random, serial, pseudo_lru, lru
} replace_policy;
```

165. A cache might also include a "victim" area, which contains the last 2^v victim blocks removed from the main cache area. The victim area can be searched in parallel with the specified cache set, thereby increasing the chance of a hit without making the search go slower. Each of the three replacement policies can be used also in the victim cache.

166. A cache also has a *granularity* 2^g, where $b \geq g \geq 3$. This means that we maintain, for each cache block, a set of 2^{b-g} "dirty bits," which identify the 2^g-byte groups that have possibly changed since they were last read from memory. Thus if $g = b$, an entire cache block is either dirty or clean; if $g = 3$, the dirtiness of each octabyte is maintained separately.

Two policies are available when new data is written into all or part of a cache block. We can *write-through*, meaning that we send all new data to memory immediately and never mark anything dirty; or we can *write-back*, meaning that we update the memory from the cache only when absolutely necessary. Furthermore we can *write-allocate*, meaning that we keep the new data in the cache, even if the cache block being written has to be fetched first because of a miss; or we can *write-around*, meaning that we keep the new data only if it was part of an existing cache block.

(In this discussion, "memory" is shorthand for "the next level of the memory hierarchy"; if there is an S-cache, the I-cache and D-cache write new data to the S-cache, not directly to memory. The I-cache, IT-cache, and DT-cache are read-only, so they do not need the facilities discussed in this section. Moreover, the D-cache and S-cache can be assumed to have the same granularity.)

```
⟨ Header definitions 6 ⟩ +≡
#define WRITE_BACK 1    /* use this if not write-through */
#define WRITE_ALLOC 2    /* use this if not write-around */
```

167. We have seen that many flavors of cache can be simulated. They are represented by **cache** structures, containing arrays of **cacheset** structures that contain arrays of **cacheblock** structures for the individual blocks. We use a full byte to store each *dirty* bit, and we use full integer words to store *rank* fields for LRU processing, etc.; memory economy is less important than simplicity in this simulator.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    octa tag;      /* bits of key not included in the cache block address */
    char *dirty;      /* array of 2^(g−b) dirty bits, one per granule */
    octa *data;      /* array of 2^(b−3) octabytes, the data in a cache block */
    int rank;      /* auxiliary information for non-random policies */
  } cacheblock;
  typedef cacheblock *cacheset;      /* array of 2^a or 2^v blocks */
  typedef struct {
    int a, b, c, g, v;      /* lg of associativity, blocksize, setsize, granularity, and victimsize */
    int aa, bb, cc, gg, vv;
      /* associativity, blocksize, setsize, granularity, and victimsize (all powers of 2) */
    int tagmask;      /* −2^(b+c) */
    replace_policy repl, vrepl;      /* how to choose victims and victim-victims */
    int mode;      /* optional WRITE_BACK and/or WRITE_ALLOC */
    int access_time;      /* cycles to know if there's a hit */
    int copy_in_time;      /* cycles to copy a new block into the cache */
    int copy_out_time;      /* cycles to copy an old block from the cache */
    cacheset *set;      /* array of 2^c sets of arrays of cache blocks */
    cacheset victim;      /* the victim cache, if present */
    coroutine filler;      /* a coroutine for copying new blocks into the cache */
    control filler_ctl;      /* its control block */
    coroutine flusher;      /* a coroutine for writing dirty old data from the cache */
    control flusher_ctl;      /* its control block */
    cacheblock inbuf;      /* filling comes from here */
    cacheblock outbuf;      /* flushing goes to here */
    lockvar lock;      /* nonzero when the cache is being changed significantly */
    lockvar fill_lock;      /* nonzero when filler should pass data back */
    int ports;      /* how many coroutines can be reading the cache? */
    coroutine *reader;      /* array of coroutines that might be reading simultaneously */
    char *name;      /* "Icache", for example */
  } cache;
```

168.
```
⟨ External variables 4 ⟩ +≡
  Extern cache *Icache, *Dcache, *Scache, *ITcache, *DTcache;
```

169. Now we are ready to define some basic subroutines for cache maintenance. Let's begin with a trivial routine that tests if a given cache block is dirty.

```
⟨ Internal prototypes 13 ⟩ +≡
  static bool is_dirty ARGS((cache *, cacheblock *));
```

170. ⟨ Subroutines 14 ⟩ +≡

```
static bool is_dirty(c, p)
    cache *c;      /* the cache containing it */
    cacheblock *p;      /* a cache block */
{
  register int j;
  register char *d = p→dirty;
  for (j = 0; j < c→bb; d++, j += c→gg)
    if (*d) return true;
  return false;
}
```

171. For diagnostic purposes we might want to display an entire cache block.

⟨ Internal prototypes 13 ⟩ +≡

```
static void print_cache_block ARGS((cacheblock, cache *));
```

172. ⟨ Subroutines 14 ⟩ +≡

```
static void print_cache_block(p, c)
    cacheblock p;
    cache *c;
{ register int i, j, b = c→bb ≫ 3, g = c→gg ≫ 3;
  printf("%08x%08x:␣", p.tag.h, p.tag.l);
  for (i = j = 0; j < b; j++, i += ((j & (g − 1)) ? 0 : 1))
    printf("%08x%08x%c", p.data[j].h, p.data[j].l, p.dirty[i] ? '*' : '␣');
  printf("␣(%d)\n", p.rank);
}
```

173. ⟨ Internal prototypes 13 ⟩ +≡

```
static void print_cache_locks ARGS((cache *));
```

174. ⟨ Subroutines 14 ⟩ +≡

```
static void print_cache_locks(c)
    cache *c;
{
  if (c) {
    if (c→lock) printf("%s␣locked␣by␣%s:%d\n", c→name, c→lock→name, c→lock→stage);
    if (c→fill_lock) printf("%sfill␣locked␣by␣%s:%d\n", c→name, c→fill_lock→name, c→fill_lock→stage);
  }
}
```

175. The *print_cache* routine prints the entire contents of a cache. This can be a huge amount of data, but it can be very useful when debugging. Fortunately, the task of debugging favors the use of small caches, since interesting cases arise more often when a cache is fairly small.

⟨ External prototypes 9 ⟩ +≡

```
Extern void print_cache ARGS((cache *, bool));
```

176. ⟨ External routines 10 ⟩ +≡

```
void print_cache(c, dirty_only)
    cache *c;
    bool dirty_only;
{
  if (c) { register int i, j;
    printf("%s of %s:", dirty_only ? "Dirty blocks" : "Contents", c→name);
    if (c→filler.next) {
      printf(" (filling ");
      print_octa(c→name[1] ≡ 'T' ? c→filler_ctl.y.o : c→filler_ctl.z.o);
      printf(")");
    }
    if (c→flusher.next) {
      printf(" (flushing ");
      print_octa(c→outbuf.tag);
      printf(")");
    }
    printf("\n");
    ⟨ Print all of c's cache blocks 177 ⟩;
  }
}
```

177. We don't print the cache blocks that have an invalid tag, unless requested to be verbose.

⟨ Print all of c's cache blocks 177 ⟩ ≡

```
for (i = 0; i < c→cc; i++)
  for (j = 0; j < c→aa; j++)
    if ((¬(c→set[i][j].tag.h & sign_bit) ∨ (verbose & show_wholecache_bit)) ∧
          (¬dirty_only ∨ is_dirty(c, &c→set[i][j]))) {
      printf("[%d][%d] ", i, j);
      print_cache_block(c→set[i][j], c);
    }
for (j = 0; j < c→vv; j++)
  if ((¬(c→victim[j].tag.h & sign_bit) ∨ (verbose & show_wholecache_bit)) ∧
        (¬dirty_only ∨ is_dirty(c, &c→victim[j]))) {
    printf("V[%d] ", j);
    print_cache_block(c→victim[j], c);
  }
```

This code is used in section 176.

178. The *clean_block* routine simply initializes a given cache block.

⟨ External prototypes 9 ⟩ +≡

```
Extern void clean_block ARGS((cache *, cacheblock *));
```

179. ⟨ External routines 10 ⟩ +≡

```
void clean_block(c, p)
    cache *c;
    cacheblock *p;
{
  register int j;
  p→tag.h = sign_bit, p→tag.l = 0;
  for (j = 0; j < c→bb ≫ 3; j++) p→data[j] = zero_octa;
  for (j = 0; j < c→bb ≫ c→g; j++) p→dirty[j] = false;
}
```

180. The *zap_cache* routine invalidates all tags of a given cache, effectively restoring it to its initial condition.

⟨ External prototypes 9 ⟩ +≡

```
Extern void zap_cache ARGS((cache *));
```

181. We clear the *dirty* entries here, just to be tidy, although they could actually be left in arbitrary condition when the tags are invalid.

⟨ External routines 10 ⟩ +≡

```
void zap_cache(c)
    cache *c;
{
  register int i, j;
  for (i = 0; i < c→cc; i++)
    for (j = 0; j < c→aa; j++) {
      clean_block(c, &(c→set[i][j]));
    }
  for (j = 0; j < c→vv; j++) {
    clean_block(c, &(c→victim[j]));
  }
}
```

182. The *get_reader* subroutine finds the index of an available reader coroutine for a given cache, or returns a negative value if no readers are available.

⟨ Internal prototypes 13 ⟩ +≡

```
static int get_reader ARGS((cache *));
```

183. ⟨ Subroutines 14 ⟩ +≡

```
static int get_reader(c)
    cache *c;
{ register int j;
  for (j = 0; j < c→ports; j++)
    if (c→reader[j].next ≡ Λ) return j;
  return -1;
}
```

184. The subroutine *copy_block*(c, p, cc, pp) copies the dirty items from block p of cache c into block pp of cache cc, assuming that the destination cache has a sufficiently large block size. (In other words, we assume that $cc\rightarrow b \geq c\rightarrow b$.) We also assume that both blocks have compatible tags, and that both caches have the same granularity.

⟨ Internal prototypes 13 ⟩ +≡
```
static void copy_block ARGS((cache *, cacheblock *, cache *, cacheblock *));
```

185. ⟨ Subroutines 14 ⟩ +≡
```
static void copy_block(c, p, cc, pp)
    cache *c, *cc;
    cacheblock *p, *pp;
{
  register int j, jj, i, ii, lim;
  register int off = p→tag.l & (cc→bb − 1);
  if (c→g ≠ cc→g ∨ p→tag.h ≠ pp→tag.h ∨ p→tag.l − off ≠ pp→tag.l) panic(confusion("copy block"));
  for (j = 0, jj = off ≫ c→g; j < c→bb ≫ c→g; j++, jj++)
    if (p→dirty[j]) {
      pp→dirty[jj] = true;
      for (i = j ≪ (c→g − 3), ii = jj ≪ (c→g − 3), lim = (j + 1) ≪ (c→g − 3); i < lim; i++, ii++)
        pp→data[ii] = p→data[i];
    }
}
```

186. The *choose_victim* subroutine selects the victim to be replaced when we need to change a cache set. We need only one bit of the *rank* fields to implement the r table when $policy = pseudo\_lru$, and we don't need *rank* at all when $policy = random$. Of course we use an a-bit counter to implement $policy = serial$. In the other case, $policy = lru$, we need an a-bit *rank* field; the least recently used entry has rank 0, and the most recently used entry has rank $2^a - 1 = aa - 1$.

⟨ Internal prototypes 13 ⟩ +≡
```
static cacheblock *choose_victim ARGS((cacheset, int, replace_policy));
```

187. ⟨ Subroutines 14 ⟩ +≡
```
static cacheblock *choose_victim(s, aa, policy)
    cacheset s;
    int aa;     /* setsize */
    replace_policy policy;
{
  register cacheblock *p;
  register int l, m;
  switch (policy) {
  case random: return &s[ticks.l & (aa − 1)];
  case serial: l = s[0].rank; s[0].rank = (l + 1) & (aa − 1); return &s[l];
  case lru:
    for (p = s; p < s + aa; p++)
      if (p→rank ≡ 0) return p;
  default: panic(confusion("lru victim"));     /* what happened? nobody has rank zero */
  case pseudo_lru:
    for (l = 1, m = aa ≫ 1; m; m ≫= 1) l = l + l + s[l].rank;
    return &s[l − aa];
  }
}
```

188. The *note_usage* subroutine updates the *rank* entries to record the fact that a particular block in a cache set is now being used.

```
⟨ Internal prototypes 13 ⟩ +≡
  static void note_usage ARGS((cacheblock *, cacheset, int, replace_policy));
```

189.
```
⟨ Subroutines 14 ⟩ +≡
  static void note_usage(l, s, aa, policy)
      cacheblock *l;      /* a cache block that's probably worth preserving */
      cacheset s;      /* the set that contains l */
      int aa;      /* setsize */
      replace_policy policy;
  {
    register cacheblock *p;
    register int j, m, r;
    if (aa ≡ 1 ∨ policy ≤ serial) return;
    if (policy ≡ lru) {
      r = l→rank;
      for (p = s; p < s + aa; p++)
        if (p→rank > r) p→rank--;
      l→rank = aa − 1;
    }
    else {      /* policy ≡ pseudo_lru */
      r = l − s;
      for (j = 1, m = aa ≫ 1; m; m ≫= 1)
        if (r & m) s[j].rank = 0, j = j + j + 1;
        else s[j].rank = 1, j = j + j;
    }
    return;
  }
```

190. The *demote_usage* subroutine is sort of the opposite of *note_usage*; it changes the rank of a given block to *least* recently used.

```
⟨ Internal prototypes 13 ⟩ +≡
  static void demote_usage ARGS((cacheblock *, cacheset, int, replace_policy));
```

191. ⟨ Subroutines 14 ⟩ +≡

```
static void demote_usage(l, s, aa, policy)
    cacheblock *l;    /* a cache block we probably don't need */
    cacheset s;    /* the set that contains l */
    int aa;    /* setsize */
    replace_policy policy;
{
  register cacheblock *p;
  register int j, m, r;
  if (aa ≡ 1 ∨ policy ≤ serial) return;
  if (policy ≡ lru) {
    r = l→rank;
    for (p = s; p < s + aa; p++)
      if (p→rank < r) p→rank++;
    l→rank = 0;
  }
  else {    /* policy ≡ pseudo_lru */
    r = l − s;
    for (j = 1, m = aa ≫ 1; m; m ≫= 1)
      if (r & m) s[j].rank = 1, j = j + j + 1;
      else s[j].rank = 0, j = j + j;
  }
  return;
}
```

192. The *cache_search* routine looks for a given key α in a given cache, and returns a cache block if there's a hit; otherwise it returns Λ. If the search hits, the set in which the block was found is stored in global variable *hit_set*. Notice that we need to check more bits of the tag when we search in the victim area.

```
#define cache_addr(c, alf)  c→set[(alf.l & ~(c→tagmask)) ≫ c→b]
```

⟨ Internal prototypes 13 ⟩ +≡

```
static cacheblock *cache_search ARGS((cache *, octa));
```

193. ⟨ Subroutines 14 ⟩ +≡

```
static cacheblock *cache_search(c, alf)
    cache *c;    /* the cache to be searched */
    octa alf;    /* the key */
{
  register cacheset s;
  register cacheblock *p;
  s = cache_addr(c, alf);    /* the set corresponding to alf */
  for (p = s; p < s + c→aa; p++)
    if (((p→tag.l ⊕ alf.l) & c→tagmask) ≡ 0 ∧ p→tag.h ≡ alf.h) goto hit;
  s = c→victim;
  if (¬s) return Λ;    /* cache miss, and no victim area */
  for (p = s; p < s + c→vv; p++)
    if (((p→tag.l ⊕ alf.l) & (−c→bb)) ≡ 0 ∧ p→tag.h ≡ alf.h) goto hit;
  return Λ;    /* double miss */
hit: hit_set = s; return p;
}
```

194. ⟨ Global variables 20 ⟩ +≡
```
cacheset hit_set;
```

195. If $p = cache\_search(c, alf)$ hits and if we call $use\_and\_fix(c, p)$ immediately afterwards, cache c is updated to record the usage of key alf. A hit in the victim area moves the cache block to the main area, unless the *filler* routine of cache c is active. A pointer to the (possibly moved) cache block is returned.

⟨ Internal prototypes 13 ⟩ +≡
```
static cacheblock *use_and_fix ARGS((cache *, cacheblock *));
```

196. ⟨ Subroutines 14 ⟩ +≡
```
static cacheblock *use_and_fix(c, p)
    cache *c;
    cacheblock *p;
{
  if (hit_set ≠ c→victim) note_usage(p, hit_set, c→aa, c→repl);
  else {
    note_usage(p, hit_set, c→vv, c→vrepl);      /* found in victim cache */
    if (¬c→filler.next) {
      register cacheset s = cache_addr(c, p→tag);
      register cacheblock *q = choose_victim(s, c→aa, c→repl);
      note_usage(q, s, c→aa, c→repl);
      ⟨ Swap cache blocks p and q 197 ⟩;
      return q;
    }
  }
  return p;
}
```

197. We can simply permute the pointers inside the cacheblock structures of a cache, instead of copying the data, if we are careful not to let any of those pointers escape into other data structures.

⟨ Swap cache blocks p and q 197 ⟩ ≡
```
{
  octa t;
  register char *d = p→dirty;
  register octa *dd = p→data;
  t = p→tag; p→tag = q→tag; q→tag = t;
  p→dirty = q→dirty; q→dirty = d;
  p→data = q→data; q→data = dd;
}
```
This code is used in sections 196 and 205.

198. The *demote_and_fix* routine is analogous to *use_and_fix*, except that we don't want to promote the data we found.

⟨ Internal prototypes 13 ⟩ +≡
```
static cacheblock *demote_and_fix ARGS((cache *, cacheblock *));
```

199. ⟨ Subroutines 14 ⟩ +≡

```
static cacheblock *demote_and_fix(c,p)
    cache *c;
    cacheblock *p;
{
  if (hit_set ≠ c→victim) demote_usage(p, hit_set, c→aa, c→repl);
  else demote_usage(p, hit_set, c→vv, c→vrepl);
  return p;
}
```

200. The subroutine *load_cache*(c,p) is called at a moment when c→*lock* has been set and c→*inbuf* has been filled with clean data to be placed in the cache block p.

⟨ Internal prototypes 13 ⟩ +≡

```
static void load_cache ARGS((cache *, cacheblock *));
```

201. ⟨ Subroutines 14 ⟩ +≡

```
static void load_cache(c,p)
    cache *c;
    cacheblock *p;
{
  register int i;
  register octa *d;
  for (i = 0; i < c→bb ≫ c→g; i++) p→dirty[i] = false;
  d = p→data; p→data = c→inbuf.data; c→inbuf.data = d;
  p→tag = c→inbuf.tag;
  hit_set = cache_addr(c, p→tag); use_and_fix(c,p);      /* p not moved */
}
```

202. The subroutine *flush_cache*$(c,p,keep)$ is called at a "quiet" moment when c→*flusher.next* $= \Lambda$. It puts cache block p into c→*outbuf* and fires up the c→*flusher* coroutine, which will take care of sending the data to lower levels of the memory hierarchy. Cache block p is also marked clean.

⟨ Internal prototypes 13 ⟩ +≡

```
static void flush_cache ARGS((cache *, cacheblock *, bool));
```

203. ⟨ Subroutines 14 ⟩ +≡

```
static void flush_cache(c,p,keep)
    cache *c;
    cacheblock *p;      /* a block inside cache c */
    bool keep;      /* should we preserve the data in p? */
{
  register octa *d;
  register char *dd;
  register int j;
  c→outbuf.tag = p→tag;
  if (keep) for (j = 0; j < c→bb ≫ 3; j++) c→outbuf.data[j] = p→data[j];
  else d = c→outbuf.data, c→outbuf.data = p→data, p→data = d;
  dd = c→outbuf.dirty, c→outbuf.dirty = p→dirty, p→dirty = dd;
  for (j = 0; j < c→bb ≫ c→g; j++) p→dirty[j] = false;
  c→outbuf.rank = c→bb;      /* this many valid bytes */
  startup(&c→flusher, c→copy_out_time);      /* will not be aborted */
}
```

204. The *alloc_slot* routine is called when we wish to put new information into a cache after a cache miss. It returns a pointer to a cache block in the main area where the new information should be put. The tag of that cache block is invalidated; the calling routine should take care of filling it and giving it a valid tag in due time. The cache's *filler* routine should not be active when *alloc_slot* is called.

Inserting new information might also require writing old information into the next level of the memory hierarchy, if the block being replaced is dirty. This routine returns Λ in such cases if the cache is flushing a previously discarded block. Otherwise it schedules the *flusher* coroutine.

This routine returns Λ also if the given key happens to be in the cache. Such cases are rare, but the following scenario shows that they aren't impossible: Suppose the DT-cache access time is 5, the D-cache access time is 1, and two processes simultaneously look for the same physical address. One process hits in DT-cache but misses in D-cache, waiting 5 cycles before trying *alloc_slot* in the D-cache; meanwhile the other process missed in D-cache but didn't need to use the DT-cache, so it might have updated the D-cache.

A key value is never negative. Therefore we can invalidate the tag in the chosen slot by forcing it to be negative.

```
⟨ Internal prototypes 13 ⟩ +≡
  static cacheblock *alloc_slot ARGS((cache *, octa));
```

205.

```
⟨ Subroutines 14 ⟩ +≡
  static cacheblock *alloc_slot(c, alf)
      cache *c;
      octa alf;      /* key that probably isn't in the cache */
  {
    register cacheset s;
    register cacheblock *p, *q;
    if (cache_search(c, alf)) return Λ;
    if (c→flusher.next ∧ c→outbuf.tag.h ≡ alf.h ∧ ¬((c→outbuf.tag.l ⊕ alf.l) & −c→bb)) return Λ;
    s = cache_addr(c, alf);      /* the set corresponding to alf */
    if (c→victim) p = choose_victim(c→victim, c→vv, c→vrepl);
    else p = choose_victim(s, c→aa, c→repl);
    if (is_dirty(c, p)) {
      if (c→flusher.next) return Λ;
      flush_cache(c, p, false);
    }
    if (c→victim) {
      q = choose_victim(s, c→aa, c→repl);
      ⟨ Swap cache blocks p and q 197 ⟩;
      q→tag.h |= sign_bit;      /* invalidate the tag */
      return q;
    }
    p→tag.h |= sign_bit; return p;
  }
```

206. Simulated memory. How should we deal with the potentially gigantic memory of `MMIX`? We can't simply declare an array m that has 2^{48} bytes. (Indeed, up to 2^{63} bytes are needed, if we consider also the physical addresses $\geq 2^{48}$ that are reserved for memory-mapped input/output.)

We could regard memory as a special kind of cache, in which every access is required to hit. For example, such an "M-cache" could be fully associative, with 2^a blocks each having a different tag; simulation could proceed until more than $2^a - 1$ tags are required. But then the predefined value of a might well be so large that the sequential search of our *cache_search* routine would be too slow.

Instead, we will allocate memory in chunks of 2^{16} bytes at a time, as needed, and we will use hashing to search for the relevant chunk whenever a physical address is given. If the address is 2^{48} or greater, special routines called *spec_read* and *spec_write*, supplied by the user, will be called upon to do the reading or writing. Otherwise the 48-bit address consists of a 32-bit *chunk address* and a 16-bit *chunk offset*.

Chunk addresses that are not used take no space in this simulator. But if, say, 1000 such patterns occur, the simulator will dynamically allocate approximately 65MB for the portions of main memory that are used. Parameter *mem_chunks_max* specifies the largest number of different chunk addresses that are supported. This parameter does not constrain the range of simulated physical addresses, which cover the entire 256 large-terabyte range permitted by `MMIX`.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    tetra tag;      /* 32-bit chunk address */
    octa *chunk;    /* either Λ or an array of 2^13 octabytes */
  } chunknode;
```

207. The parameter *hash_prime* should be a prime number larger than the parameter *mem_chunks_max*, preferably more than twice as large but not much bigger than that. The default values *mem_chunks_max* = 1000 and *hash_prime* = 2003 are set by *MMIX_config* unless the user specifies otherwise.

```
⟨ External variables 4 ⟩ +≡
  Extern int mem_chunks;      /* this many chunks are allocated so far */
  Extern int mem_chunks_max;  /* up to this many different chunks per run */
  Extern int hash_prime;      /* larger than mem_chunks_max, but not enormous */
  Extern chunknode *mem_hash; /* the simulated main memory */
```

208. The separately compiled procedures *spec_read*( ) and *spec_write*( ) have the same calling conventions as the general procedures *mem_read*( ) and *mem_write*( ), but with an additional *size* parameter, which specifies that $1 \ll size$ bytes should be read or written.

```
⟨ Subroutines 14 ⟩ +≡
  extern octa spec_read ARGS((octa addr, int size));   /* for memory mapped I/O */
  extern void spec_write ARGS((octa addr, octa val, int size));   /* likewise */
```

209. If the program tries to read from a chunk that hasn't been allocated, the value zero is returned, optionally with a comment to the user.

Chunk address 0 is always allocated first. Then we can assume that a matching chunk tag implies a nonnull *chunk* pointer.

This routine sets *last_h* to the chunk found, so that we can rapidly read other words that we know must belong to the same chunk. For this purpose it is convenient to let *mem_hash*[*hash_prime*] be a chunk full of zeros, representing uninitialized memory.

```
⟨ External prototypes 9 ⟩ +≡
  Extern octa mem_read ARGS((octa addr));
```

210. ⟨ External routines 10 ⟩ +≡

```
octa mem_read(addr)
    octa addr;
{
  register tetra off, key;
  register int h;
  off = (addr.l & #ffff) ≫ 3;
  key = (addr.l & #ffff0000) + addr.h;
  for (h = key % hash_prime; mem_hash[h].tag ≠ key; h--) {
    if (mem_hash[h].chunk ≡ Λ) {
      if (verbose & uninit_mem_bit)
        errprint2("uninitialized␣memory␣read␣at␣%08x%08x", addr.h, addr.l);
      h = hash_prime; break;      /* zero will be returned */
    }
    if (h ≡ 0) h = hash_prime;
  }
  last_h = h;
  return mem_hash[h].chunk[off];
}
```

211. ⟨ External variables 4 ⟩ +≡

```
Extern int last_h;      /* the hash index that was most recently correct */
```

212. ⟨ External prototypes 9 ⟩ +≡

```
Extern void mem_write ARGS((octa addr, octa val));
```

213. ⟨ External routines 10 ⟩ +≡

```
void mem_write(addr, val)
    octa addr, val;
{
  register tetra off, key;
  register int h;
  off = (addr.l & #ffff) ≫ 3;
  key = (addr.l & #ffff0000) + addr.h;
  for (h = key % hash_prime; mem_hash[h].tag ≠ key; h--) {
    if (mem_hash[h].chunk ≡ Λ) {
      if (++mem_chunks > mem_chunks_max)
        panic(errprint1("More␣than␣%d␣memory␣chunks␣are␣needed", mem_chunks_max));
      mem_hash[h].chunk = (octa *) calloc(1 ≪ 13, sizeof(octa));
      if (mem_hash[h].chunk ≡ Λ)
        panic(errprint1("I␣can't␣allocate␣memory␣chunk␣number␣%d", mem_chunks));
      mem_hash[h].tag = key;
      break;
    }
    if (h ≡ 0) h = hash_prime;
  }
  last_h = h;
  mem_hash[h].chunk[off] = val;
}
```

214. The memory is characterized by several parameters, depending on the characteristics of the memory bus being simulated. Let *bus_words* be the number of octabytes read or written simultaneously (usually *bus_words* is 1 or 2; it must be a power of 2). The number of clock cycles needed to read or write $c*bus\_words$ octabytes that all belong to the same cache block is assumed to be $mem\_addr\_time + c * mem\_read\_time$ or $mem\_addr\_time + c * mem\_write\_time$, respectively.

```
⟨ External variables 4 ⟩ +≡
  Extern int mem_addr_time;      /* cycles to transmit an address on memory bus */
  Extern int bus_words;      /* width of memory bus, in octabytes */
  Extern int mem_read_time;      /* cycles to read from main memory */
  Extern int mem_write_time;      /* cycles to write to main memory */
  Extern lockvar mem_lock;      /* is nonnull when the bus is busy */
```

215. One of the principal ways to write memory is to invoke a *flush_to_mem* coroutine, which is the *Scache→flusher* if there is an S-cache, or the *Dcache→flusher* if there is a D-cache but no S-cache.

When such a coroutine is started, its *data→ptr_a* will be *Scache* or *Dcache*. The data to be written will just have been copied to the cache's *outbuf*.

```
⟨ Cases for control of special coroutines 126 ⟩ +≡
case flush_to_mem:
  { register cache *c = (cache *) data→ptr_a;
    switch (data→state) {
    case 0: if (mem_lock) wait(1);
      data→state = 1;
    case 1: set_lock(self, mem_lock);
      data→state = 2;
      ⟨ Write the dirty data of c→outbuf and wait for the bus 216 ⟩;
    case 2: goto terminate;      /* this frees mem_lock and c→outbuf */
    }
  }
```

216. ⟨ Write the dirty data of *c→outbuf* and wait for the bus 216 ⟩ ≡

```
{
  register int off, last_off, count, first, ii;
  register int del = c→gg ≫ 3;      /* octabytes per granule */
  octa addr;
  addr = c→outbuf.tag; off = (addr.l & #ffff) ≫ 3;
  for (i = j = 0, first = 1, count = 0; j < c→bb ≫ c→g; j++) {
    ii = i + del;
    if (¬c→outbuf.dirty[j]) i = ii, off += del, addr.l += del ≪ 3;
    else while (i < ii) {
        if (first) {
          count++; last_off = off; first = 0;
          mem_write(addr, c→outbuf.data[i]);
        } else {
          if ((off ⊕ last_off) & (−bus_words)) count++;
          last_off = off;
          mem_hash[last_h].chunk[off] = c→outbuf.data[i];
        }
        i++; off++; addr.l += 8;
      }
  }
  wait(mem_addr_time + count * mem_write_time);
}
```

This code is used in section 215.

217. Cache transfers. We have seen that the *Dcache→flusher* sends data directly to the main memory if there is no S-cache. But if both D-cache and S-cache exist, the *Dcache→flusher* is a more complicated coroutine of type *flush_to_S*. In this case we need to deal with the fact that the S-cache blocks might be larger than the D-cache blocks; furthermore, the S-cache might have a write-around and/or write-through policy, etc. But one simplifying fact does help us: We know that the flusher coroutine will not be aborted until it has run to completion.

Some machines, such as the Alpha 21164, have an additional cache between the S-cache and memory, called the B-cache (the "backup cache"). A B-cache could be simulated by extending the logic used here; but such extensions of the present program are left to the interested reader.

```
⟨ Cases for control of special coroutines 126 ⟩ +≡
case flush_to_S:
  { register cache *c = (cache *) data→ptr_a;
    register int block_diff = Scache→bb − c→outbuf.rank;
    p = (cacheblock *) data→ptr_b;
    switch (data→state) {
    case 0: if (Scache→lock) wait(1);
      data→state = 1;
    case 1: set_lock(self, Scache→lock);
      data→ptr_b = (void *) cache_search(Scache, c→outbuf.tag);
      if (data→ptr_b) data→state = 4;
      else if (Scache→mode & WRITE_ALLOC) data→state = (block_diff ? 2 : 3);
      else data→state = 6;
      wait(Scache→access_time);
    case 2: ⟨ Fill Scache→inbuf with clean memory data 219 ⟩;
    case 3: ⟨ Allocate a slot p in the S-cache 218 ⟩;
      if (block_diff) ⟨ Copy Scache→inbuf to slot p 220 ⟩
      else for (j = 0; j < Scache→bb ≫ 3; j++) p→data[j] = c→outbuf.data[j];
      for (j = 0; j < Scache→bb ≫ Scache→g; j++) p→dirty[j] = false;
    case 4: copy_block(c, &(c→outbuf), Scache, p);
      hit_set = cache_addr(Scache, c→outbuf.tag); use_and_fix(Scache, p);      /* p not moved */
      data→state = 5; wait(Scache→copy_in_time);
    case 5: if ((Scache→mode & WRITE_BACK) ≡ 0) {      /* write-through */
        if (Scache→flusher.next) wait(1);
        flush_cache(Scache, p, true);
      }
      goto terminate;
    case 6: ⟨ Handle write-around when flushing to the S-cache 221 ⟩;
    }
  }
```

218. ⟨ Allocate a slot p in the S-cache 218 ⟩ ≡

```
if (Scache→filler.next) wait(1);      /* perhaps an unnecessary precaution? */
p = alloc_slot(Scache, c→outbuf.tag);
if (¬p) wait(1);
data→ptr_b = (void *) p;
p→tag = c→outbuf.tag; p→tag.l = c→outbuf.tag.l & (−Scache→bb);
```

This code is used in section 217.

219. We only need to read *block_diff* bytes, but it's easier to read them all and to charge only for reading the ones we needed.

```
⟨ Fill Scache→inbuf with clean memory data 219 ⟩ ≡
  { register int count = block_diff ≫ 3;
    register int off, delay;
    octa addr;
    if (mem_lock) wait(1);
    addr.h = c→outbuf.tag.h; addr.l = c→outbuf.tag.l & -Scache→bb;
    off = (addr.l & #ffff) ≫ 3;
    for (j = 0; j < Scache→bb ≫ 3; j++)
      if (j ≡ 0) Scache→inbuf.data[j] = mem_read(addr);
      else Scache→inbuf.data[j] = mem_hash[last_h].chunk[j + off];
    set_lock(&mem_locker, mem_lock);
    delay = mem_addr_time + (int)((count + bus_words - 1)/(bus_words)) * mem_read_time;
    startup(&mem_locker, delay);
    data→state = 3; wait(delay);
  }
```

This code is used in section 217.

220.

```
⟨ Copy Scache→inbuf to slot p 220 ⟩ ≡
  {
    register octa *d = p→data;
    p→data = Scache→inbuf.data; Scache→inbuf.data = d;
  }
```

This code is used in section 217.

221. Here we assume that the granularity is 8.

```
⟨ Handle write-around when flushing to the S-cache 221 ⟩ ≡
  if (Scache→flusher.next) wait(1);
  Scache→outbuf.tag.h = c→outbuf.tag.h;
  Scache→outbuf.tag.l = c→outbuf.tag.l & (-Scache→bb);
  for (j = 0; j < Scache→bb ≫ Scache→g; j++) Scache→outbuf.dirty[j] = false;
  copy_block(c, &(c→outbuf), Scache, &(Scache→outbuf));
  startup(&Scache→flusher, Scache→copy_out_time);
  goto terminate;
```

This code is used in section 217.

222. The S-cache gets new data from memory by invoking a *fill_from_mem* coroutine; the I-cache or D-cache may also invoke a *fill_from_mem* coroutine, if there is no S-cache. When such a coroutine is invoked, it holds *mem_lock*, and its caller has gone to sleep. A physical memory address is given in *data→z.o*, and *data→ptr_a* specifies either *Icache*, *Dcache*, or *Scache*. Furthermore, *data→ptr_b* specifies a block within that cache, determined by the *alloc_slot* routine. The coroutine simulates reading the contents of the specified memory location, places the result in the *x.o* field of its caller's control block, and wakes up the caller. It proceeds to fill the cache's *inbuf* and, ultimately, the specified cache block, before waking the caller again.

Let $c = data→ptr\_a$. The caller is then *c→fill_lock*, if this variable is nonnull. However, the caller might not wish to be awoken or to receive the data (for example, if it has been aborted). In such cases *c→fill_lock* will be Λ; the filling action continues without the wakeup calls. If $c = Scache$, the S-cache will be locked and the caller will not have been aborted.

```
⟨ Cases for control of special coroutines 126 ⟩ +≡
case fill_from_mem:
  { register cache *c = (cache *) data→ptr_a;
    register coroutine *cc = c→fill_lock;
    switch (data→state) {
    case 0: data→x.o = mem_read(data→z.o);
      if (cc) {
        cc→ctl→x.o = data→x.o;
        awaken(cc, mem_read_time);
      }
      data→state = 1;
      ⟨ Read data into c→inbuf and wait for the bus 223 ⟩;
    case 1: release_lock(self, mem_lock);
      data→state = 2;
    case 2: if (c ≠ Scache) {
        if (c→lock) wait(1);
        set_lock(self, c→lock);
      }
      if (cc) awaken(cc, c→copy_in_time);      /* the second wakeup call */
      load_cache(c, (cacheblock *) data→ptr_b);
      data→state = 3; wait(c→copy_in_time);
    case 3: goto terminate;
    }
  }
```

223. If c's cache size is no larger than the memory bus, we wait an extra cycle, so that there will be two wakeup calls.

```
⟨ Read data into c→inbuf and wait for the bus 223 ⟩ ≡
  {
    register int count, off;
    c→inbuf.tag = data→z.o; c→inbuf.tag.l &= -c→bb;
    count = c→bb ≫ 3, off = (c→inbuf.tag.l & #ffff) ≫ 3;
    for (i = 0; i < count; i++, off++) c→inbuf.data[i] = mem_hash[last_h].chunk[off];
    if (count ≤ bus_words) wait(1 + mem_read_time)
    else wait((int)(count/bus_words) * mem_read_time);
  }
```

This code is used in section 222.

224. The *fill_from_S* coroutine has the same conventions as *fill_from_mem*, except that the data comes directly from the S-cache if it is present there. This is the *filler* coroutine for the I-cache and D-cache if an S-cache is present.

```
⟨ Cases for control of special coroutines 126 ⟩ +≡
case fill_from_S:
  { register cache *c = (cache *) data→ptr_a;
    register coroutine *cc = c→fill_lock;
    p = (cacheblock *) data→ptr_c;
    switch (data→state) {
    case 0: p = cache_search(Scache, data→z.o);
      if (p) goto S_non_miss;
      data→state = 1;
    case 1: ⟨ Start the S-cache filler 225 ⟩;
      data→state = 2; sleep;
    case 2: if (cc) {
        cc→ctl→x.o = data→x.o;      /* this data has been supplied by Scache→filler */
        awaken(cc, Scache→access_time);      /* we propagate it back */
      }
      data→state = 3; sleep;      /* when we awake, the S-cache will have our data */
    S_non_miss: if (cc) {
        cc→ctl→x.o = p→data[(data→z.o.l & (Scache→bb − 1)) ≫ 3];
        awaken(cc, Scache→access_time);
      }
    case 3: ⟨ Copy data from p into c→inbuf 226 ⟩;
      data→state = 4; wait(Scache→access_time);
    case 4: Scache→lock = Λ;      /* we had been holding that lock */
      data→state = 5;
    case 5: if (c→lock) wait(1);
      set_lock(self, c→lock);
      load_cache(c, (cacheblock *) data→ptr_b);
      data→state = 6; wait(c→copy_in_time);
    case 6: if (cc) awaken(cc, 1);      /* second wakeup call */
      goto terminate;
    }
  }
```

225. We are already holding the *Scache→lock*, but we're about to take on the *Scache→fill_lock* too (with the understanding that one is "stronger" than the other). For a short time the *Scache→lock* will point to us but we will point to *Scache→fill_lock*; this will not cause difficulty, because the present coroutine is not abortable.

```
⟨ Start the S-cache filler 225 ⟩ ≡
  if (Scache→filler.next ∨ mem_lock) wait(1);
  p = alloc_slot(Scache, data→z.o);
  if (¬p) wait(1);
  set_lock(&Scache→filler, mem_lock);
  set_lock(self, Scache→fill_lock);
  data→ptr_c = Scache→filler_ctl.ptr_b = (void *) p;
  Scache→filler_ctl.z.o = data→z.o;
  startup(&Scache→filler, mem_addr_time);
```

This code is used in section 224.

226. The S-cache blocks might be wider than the blocks of the I-cache or D-cache, so the copying in this step isn't quite trivial.

```
⟨ Copy data from p into c→inbuf 226 ⟩ ≡
  { register int off;
    c→inbuf.tag = data→z.o; c→inbuf.tag.l &= -c→bb;
    for (j = 0, off = (c→inbuf.tag.l & (Scache→bb - 1)) >> 3; j < c→bb >> 3; j++, off++)
      c→inbuf.data[j] = p→data[off];
    release_lock(self, Scache→fill_lock);
    set_lock(self, Scache→lock);
  }
```

This code is used in section 224.

227. The instruction `PRELD X,$Y,$Z` generates $\lfloor X/2^b \rfloor$ commands if there are 2^b bytes per block in the D-cache. These commands will try to preload blocks \$Y + \$Z, \$Y + \$Z + 2^b, ..., into the cache if it is not too busy.

Similar considerations apply to the instructions `PREGO X,$Y,$Z` and `PREST X,$Y,$Z`.

```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case preld: case prest: if (¬Dcache) goto noop_inst;
  if (cool→xx ≥ Dcache→bb) cool→interim = true;
  cool→ptr_a = (void *) mem.up; break;
case prego: if (¬Icache) goto noop_inst;
  if (cool→xx ≥ Icache→bb) cool→interim = true;
  cool→ptr_a = (void *) mem.up; break;
```

228. If the block size is 64, a command like `PREST 200,$Y,$Z` is actually issued as four commands `PREST 200,$Y,$Z; PREST 191,$Y,$Z; PREST 127,$Y,$Z; PREST 63,$Y,$Z`. An interruption will then be able to resume properly. In the pipeline, the instruction `PREST 200,$Y,$Z` is considered to affect bytes \$Y + \$Z + 192 through \$Y + \$Z + 200, or fewer bytes if \$Y + \$Z is not a multiple of 64. (Remember that these instructions are only hints; we act on them only if it is reasonably convenient to do so.)

```
⟨ Get ready for the next step of PRELD or PREST 228 ⟩ ≡
  head→inst = (head→inst & ~((Dcache→bb - 1) << 16)) - #10000;
```

This code is used in section 81.

229. ⟨ Get ready for the next step of `PREGO` 229 ⟩ ≡

```
  head→inst = (head→inst & ~((Icache→bb - 1) << 16)) - #10000;
```

This code is used in section 81.

230. Another coroutine, called *cleanup*, is occasionally called into action to remove dirty data from the D-cache and S-cache. If it is invoked by starting in state 0, with its *i* field set to *sync*, it will clean everything. It can also be invoked in state 4, with its *i* field set to *syncd* and with a physical address in its *z.o* field; then it simply makes sure that no D-cache or S-cache blocks associated with that address are dirty.

Field *x.o.h* should be set to zero if items are expected to remain in the cache after being cleaned; otherwise field *x.o.h* should be set to *sign_bit*.

The coroutine that invokes *cleanup* should hold *clean_lock*. If that coroutine dies, because of an interruption, the *cleanup* coroutine will terminate prematurely.

We assume that the D-cache and S-cache have some sort of way to identify their first dirty block, if any, in *access_time* cycles.

```
⟨ Global variables 20 ⟩ +≡
  coroutine clean_co;
  control clean_ctl;
  lockvar clean_lock;
```

231. ⟨ Initialize everything 22 ⟩ +≡
```
clean_co.ctl = &clean_ctl;
clean_co.name = "Clean";
clean_co.stage = cleanup;
clean_ctl.go.o.l = 4;
```

232. ⟨ Cases for control of special coroutines 126 ⟩ +≡
```
case cleanup: p = (cacheblock *) data→ptr_b;
  switch (data→state) {
    ⟨ Cases 0 through 4, for the D-cache 233 ⟩;
    ⟨ Cases 5 through 9, for the S-cache 234 ⟩;
  case 10: goto terminate;
  }
```

233. ⟨ Cases 0 through 4, for the D-cache 233 ⟩ ≡

```
case 0: if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);
  startup(&Dcache→reader[j], Dcache→access_time);
  set_lock(self, Dcache→lock);
  i = j = 0;
Dclean_loop: p = (i < Dcache→cc ? &(Dcache→set[i][j]) : &(Dcache→victim[j]));
  if (p→tag.h & sign_bit) goto Dclean_inc;
  if (¬is_dirty(Dcache, p)) {
    p→tag.h |= data→x.o.h; goto Dclean_inc;
  }
  data→y.o.h = i, data→y.o.l = j;
Dclean: data→state = 1; data→ptr_b = (void *) p; wait(Dcache→access_time);
case 1: if (Dcache→flusher.next) wait(1);
  flush_cache(Dcache, p, data→x.o.h ≡ 0);
  p→tag.h |= data→x.o.h;
  release_lock(self, Dcache→lock);
  data→state = 2; wait(Dcache→copy_out_time);
case 2: if (¬clean_lock) goto done;      /* premature termination */
  if (Dcache→flusher.next) wait(1);
  if (data→i ≠ sync) goto Sprep;
  data→state = 3;
case 3: if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);
  startup(&Dcache→reader[j], Dcache→access_time);
  set_lock(self, Dcache→lock);
  i = data→y.o.h, j = data→y.o.l;
Dclean_inc: j++;
  if (i < Dcache→cc ∧ j ≡ Dcache→aa) j = 0, i++;
  if (i ≡ Dcache→cc ∧ j ≡ Dcache→vv) {
    data→state = 5; wait(Dcache→access_time);
  }
  goto Dclean_loop;
case 4: if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);
  startup(&Dcache→reader[j], Dcache→access_time);
  set_lock(self, Dcache→lock);
  p = cache_search(Dcache, data→z.o);
  if (p) {
    demote_and_fix(Dcache, p);
    if (is_dirty(Dcache, p)) goto Dclean;
  }
  data→state = 9; wait(Dcache→access_time);
```

This code is used in section 232.

234. ⟨ Cases 5 through 9, for the S-cache 234 ⟩ ≡

```
case 5: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (¬Scache) goto done;
  if (Scache→lock) wait(1);
  set_lock(self, Scache→lock);
  i = j = 0;
Sclean_loop: p = (i < Scache→cc ? &(Scache→set[i][j]) : &(Scache→victim[j]));
  if (p→tag.h & sign_bit) goto Sclean_inc;
  if (¬is_dirty(Scache, p)) {
    p→tag.h |= data→x.o.h; goto Sclean_inc;
  }
  data→y.o.h = i, data→y.o.l = j;
Sclean: data→state = 6; data→ptr_b = (void *) p; wait(Scache→access_time);
case 6: if (Scache→flusher.next) wait(1);
  flush_cache(Scache, p, data→x.o.h ≡ 0);
  p→tag.h |= data→x.o.h;
  release_lock(self, Scache→lock);
  data→state = 7; wait(Scache→copy_out_time);
case 7: if (¬clean_lock) goto done;      /* premature termination */
  if (Scache→flusher.next) wait(1);
  if (data→i ≠ sync) goto done;
  data→state = 8;
case 8: if (Scache→lock) wait(1);
  set_lock(self, Scache→lock);
  i = data→y.o.h, j = data→y.o.l;
Sclean_inc: j++;
  if (i < Scache→cc ∧ j ≡ Scache→aa) j = 0, i++;
  if (i ≡ Scache→cc ∧ j ≡ Scache→vv) {
    data→state = 10; wait(Scache→access_time);
  }
  goto Sclean_loop;
Sprep: data→state = 9;
case 9: if (self→lockloc) release_lock(self, Dcache→lock);
  if (¬Scache) goto done;
  if (Scache→lock) wait(1);
  set_lock(self, Scache→lock);
  p = cache_search(Scache, data→z.o);
  if (p) {
    demote_and_fix(Scache, p);
    if (is_dirty(Scache, p)) goto Sclean;
  }
  data→state = 10; wait(Scache→access_time);
```

This code is used in section 232.

235. Virtual address translation. Special arrays of coroutines and control blocks come into play when we need to implement `MMIX`'s rather complicated page table mechanism for virtual address translation. In effect, we have up to ten control blocks *outside* of the reorder buffer that are capable of executing instructions just as if they were part of that buffer. The "opcodes" of these non-abortable instructions are special internal operations called *ldptp* and *ldpte*, for loading page table pointers and page table entries.

Suppose, for example, that we need to translate a virtual address for the DT-cache in which the virtual page address $(a_4a_3a_2a_1a_0)_{1024}$ of segment i has $a_4 = a_3 = 0$ and $a_2 \neq 0$. Then the rules say that we should first find a page table pointer p_2 in physical location $2^{13}(r + b_i + 2) + 8a_2$, then another page table pointer p_1 in location $p_2 + 8a_1$, and finally the page table entry p_0 in location $p_1 + 8a_0$. The simulator achieves this by setting up three coroutines c_0, c_1, c_2 whose control blocks correspond to the pseudo-instructions

$$\begin{aligned}&\texttt{LDPTP}\ \ x,[2^{63}+2^{13}(r+b_i+2)],8a_2\\&\texttt{LDPTP}\ \ x,x,8a_1\\&\texttt{LDPTE}\ \ x,x,8a_0\end{aligned}$$

where x is a hidden internal register and the other quantities are immediate values. Slight changes to the normal functionality of `LDO` give us the actions needed to implement `LDPTP` and `LDPTE`. Coroutine c_j corresponds to the instruction that involves a_j and computes p_j; when c_0 has computed its value p_0, we know how to translate the original virtual address.

The `LDPTP` and `LDPTE` commands return zero if their y operand is zero or if the page table does not properly match rV.

```
#define LDPTP PREGO    /* internally this won't cause confusion */
#define LDPTE GO
```

⟨ Global variables 20 ⟩ +≡

```
control IPTctl[5], DPTctl[5];    /* control blocks for I and D page translation */
coroutine IPTco[10], DPTco[10];    /* each coroutine is a two-stage pipeline */
char *IPTname[5] = {"IPT0","IPT1","IPT2","IPT3","IPT4"};
char *DPTname[5] = {"DPT0","DPT1","DPT2","DPT3","DPT4"};
```

236. ⟨ Initialize everything 22 ⟩ +≡

```
for (j = 0; j < 5; j++) {
  DPTco[2*j].ctl = &DPTctl[j]; IPTco[2*j].ctl = &IPTctl[j];
  if (j > 0) DPTctl[j].op = IPTctl[j].op = LDPTP, DPTctl[j].i = IPTctl[j].i = ldptp;
  else DPTctl[0].op = IPTctl[0].op = LDPTE, DPTctl[0].i = IPTctl[0].i = ldpte;
  IPTctl[j].loc = DPTctl[j].loc = neg_one;
  IPTctl[j].go.o = DPTctl[j].go.o = incr(neg_one, 4);
  IPTctl[j].ptr_a = DPTctl[j].ptr_a = (void *) &mem;
  IPTctl[j].ren_x = DPTctl[j].ren_x = true;
  IPTctl[j].x.addr.h = DPTctl[j].x.addr.h = -1;
  IPTco[2*j].stage = DPTco[2*j].stage = 1;
  IPTco[2*j+1].stage = DPTco[2*j+1].stage = 2;
  IPTco[2*j].name = IPTco[2*j+1].name = IPTname[j];
  DPTco[2*j].name = DPTco[2*j+1].name = DPTname[j];
}
ITcache→filler_ctl.ptr_c = (void *) &IPTco[0]; DTcache→filler_ctl.ptr_c = (void *) &DPTco[0];
page_bad = true;
```

237. Page table calculations are invoked by a coroutine of type *fill_from_virt*, which is used to fill the IT-cache or DT-cache. The calling conventions of *fill_from_virt* are analogous to those of *fill_from_mem* or *fill_from_S*: A virtual address is supplied in *data→y.o*, and *data→ptr_a* points to a cache (*ITcache* or *DTcache*), while *data→ptr_b* is a block in that cache. We wake up the caller, who holds the cache's *fill_lock*, as soon as the translation of the given address has been calculated, unless the caller has been aborted. (No second wakeup call is necessary.)

```
⟨ Cases for control of special coroutines 126 ⟩ +≡
case fill_from_virt:
  { register cache *c = (cache *) data→ptr_a;
    register coroutine *cc = c→fill_lock;
    register coroutine *co = (coroutine *) data→ptr_c;      /* &IPTco[0] or &DPTco[0] */
    octa aaaaa;
    switch (data→state) {
    case 0: ⟨ Start up auxiliary coroutines to compute the page table entry 243 ⟩;
      data→state = 1;
    case 1: if (data→b.p) {
        if (data→b.p→known) data→b.o = data→b.p→o, data→b.p = Λ;
        else wait(1);
      }
      ⟨ Compute the new entry for c→inbuf and give the caller a sneak preview 245 ⟩;
      data→state = 2;
    case 2: if (c→lock) wait(1);
      set_lock(self, c→lock);
      load_cache(c, (cacheblock *) data→ptr_b);
      data→state = 3; wait(c→copy_in_time);
    case 3: data→b.o = zero_octa; goto terminate;
    }
  }
```

238. The current contents of rV, the special virtual translation register, are kept unpacked in several global variables *page_r*, *page_s*, etc., for convenience. Whenever rV changes, we recompute all these variables.

```
⟨ External variables 4 ⟩ +≡
  Extern unsigned int page_n;      /* the 10-bit n field of rV, times 8 */
  Extern int page_r;      /* the 27-bit r field of rV */
  Extern int page_s;      /* the 8-bit s field of rV */
  Extern int page_f;      /* the 3-bit f field of rV */
  Extern int page_b[5];      /* the 4-bit b fields of rV; page_b[0] = 0 */
  Extern octa page_mask;      /* the least significant s bits */
  Extern bool page_bad;      /* does rV violate the rules? */
```

239. ⟨ Update the *page* variables 239 ⟩ ≡

```
{ octa rv;
  rv = data→z.o;
  page_f = rv.l & 7, page_bad = (page_f > 1);
  page_n = rv.l & #1ff8;
  rv = shift_right(rv, 13, 1);
  page_r = rv.l & #7ffffff;
  rv = shift_right(rv, 27, 1);
  page_s = rv.l & #ff;
  if (page_s < 13 ∨ page_s > 48) page_bad = true;
  else if (page_s < 32) page_mask.h = 0, page_mask.l = (1 ≪ page_s) − 1;
  else page_mask.h = (1 ≪ (page_s − 32)) − 1, page_mask.l = #ffffffff;
  page_b[4] = (rv.l ≫ 8) & #f;
  page_b[3] = (rv.l ≫ 12) & #f;
  page_b[2] = (rv.l ≫ 16) & #f;
  page_b[1] = (rv.l ≫ 20) & #f;
}
```

This code is used in section 329.

240. Here's how we compute a tag of the IT-cache or DT-cache from a virtual address, and how we compute a physical address from a translation found in the cache.

```
#define trans_key(addr) incr(oandn(addr, page_mask), page_n)
```

⟨ Internal prototypes 13 ⟩ +≡

```
static octa phys_addr ARGS((octa, octa));
```

241. ⟨ Subroutines 14 ⟩ +≡

```
static octa phys_addr(virt, trans)
    octa virt, trans;
{ octa t;
  t = oandn(trans, page_mask);      /* zero out the ynp fields of a PTE */
  return oplus(t, oand(virt, page_mask));
}
```

242. Cheap (and slow) versions of `MMIX` leave the page table calculations to software. If the global variable *no_hardware_PT* is set true, *fill_from_virt* begins its actions in state 1, not state 0. (See the `RESUME_TRANS` operation.)

⟨ External variables 4 ⟩ +≡

```
Extern bool no_hardware_PT;
```

243. Note: The operating system is supposed to ensure that changes to the page table entries do not appear in the pipeline when a translation cache is being updated. The internal `LDPTP` and `LDPTE` instructions use only the "hot state" of the memory system.

```
⟨ Start up auxiliary coroutines to compute the page table entry 243 ⟩ ≡
  aaaaa = data→y.o;
  i = aaaaa.h ≫ 29;      /* the segment number */
  aaaaa.h &= #1fffffff;      /* the address within segment i */
  aaaaa = shift_right(aaaaa, page_s, 1);      /* the page address */
  for (j = 0; aaaaa.l ≠ 0 ∨ aaaaa.h ≠ 0; j++) {
    co[2 * j].ctl→z.o.h = 0, co[2 * j].ctl→z.o.l = (aaaaa.l & #3ff) ≪ 3;
    aaaaa = shift_right(aaaaa, 10, 1);
  }
  if (page_b[i + 1] < page_b[i] + j)      /* address too large */
    ;      /* nothing needs to be done, since data→b.o is zero */
  else {
    if (j ≡ 0) j = 1, co[0].ctl→z.o = zero_octa;
    ⟨ Issue j pseudo-instructions to compute a page table entry 244 ⟩;
  }
```

This code is used in section 237.

244. The first stage of coroutine c_j is $co[2 * j]$. It will pass the jth control block to the second stage, $co[2 * j + 1]$, which will load page table information from memory (or hopefully from the D-cache).

```
⟨ Issue j pseudo-instructions to compute a page table entry 244 ⟩ ≡
  j--;
  aaaaa.l = page_r + page_b[i] + j;
  co[2 * j].ctl→y.p = Λ;
  co[2 * j].ctl→y.o = shift_left(aaaaa, 13);
  co[2 * j].ctl→y.o.h += sign_bit;
  for (; ; j--) {
    co[2 * j].ctl→x.o = zero_octa; co[2 * j].ctl→x.known = false;
    co[2 * j].ctl→owner = &co[2 * j];
    startup(&co[2 * j], 1);
    if (j ≡ 0) break;
    co[2 * (j - 1)].ctl→y.p = &co[2 * j].ctl→x;
  }
  data→b.p = &co[0].ctl→x;
```

This code is used in section 243.

245. At this point the translation of the given virtual address $data→y.o$ is the octabyte $data→b.o$. Its least significant three bits are the protection code $p = p_r p_w p_x$; its page address field is scaled by 2^s. It is entirely zero, including the protection bits, if there was a page table failure.

The z field of the caller receives this translation.

```
⟨ Compute the new entry for c→inbuf and give the caller a sneak preview 245 ⟩ ≡
  c→inbuf.tag = trans_key(data→y.o);
  c→inbuf.data[0] = data→b.o;
  if (cc) {
    cc→ctl→z.o = data→b.o;
    awaken(cc, 1);
  }
```

This code is used in section 237.

246. The write buffer. The dispatcher has arranged things so that speculative stores into memory are recorded in a doubly linked list leading upward from *mem*. When such instructions finally are committed, they enter the "write buffer," which holds octabytes that are ready to be written into designated physical memory addresses (or into the D-cache and/or S-cache). The "hot state" of the computation is reflected not only by the registers and caches but also by the instructions that are pending in the write buffer.

```
⟨ Type definitions 11 ⟩ +≡
  typedef struct {
    octa o;      /* data to be stored */
    octa addr;      /* its physical address */
    tetra stamp;      /* when last committed (mod 2^32) */
    internal_opcode i;      /* is this write special? */
    int size;      /* parameter for spec_write */
  } write_node;
```

247. We represent the buffer in the usual way as a circular list, with elements *write_tail* + 1, *write_tail* + 2, ..., *write_head*.

The data will sit at least *holding_time* cycles before it leaves the write buffer. This speeds things up when different fields of the same octabyte are being stored by different instructions.

```
⟨ External variables 4 ⟩ +≡
  Extern write_node *wbuf_bot, *wbuf_top;      /* least and greatest write buffer nodes */
  Extern write_node *write_head, *write_tail;      /* front and rear of the write buffer */
  Extern lockvar wbuf_lock;      /* is the data in write_head being written? */
  Extern int holding_time;      /* minimum holding time */
  Extern lockvar speed_lock;      /* should we ignore holding_time? */
```

248.
```
⟨ Global variables 20 ⟩ +≡
  coroutine write_co;      /* coroutine that empties the write buffer */
  control write_ctl;      /* its control block */
```

249.
```
⟨ Initialize everything 22 ⟩ +≡
  write_co.ctl = &write_ctl;
  write_co.name = "Write";
  write_co.stage = write_from_wbuf;
  write_ctl.ptr_a = (void *) &mem;
  write_ctl.go.o.l = 4;
  startup(&write_co, 1);
  write_head = write_tail = wbuf_top;
```

250.
```
⟨ Internal prototypes 13 ⟩ +≡
  static void print_write_buffer ARGS((void));
```

251. ⟨ Subroutines 14 ⟩ +≡

```
static void print_write_buffer()
{
  printf("Write␣buffer");
  if (write_head ≡ write_tail) printf("␣(empty)\n");
  else { register write_node *p;
    printf(":\n");
    for (p = write_head; p ≠ write_tail; p = (p ≡ wbuf_bot ? wbuf_top : p − 1)) {
      printf("m["); print_octa(p→addr); printf("]="); print_octa(p→o);
      if (p→i ≡ stunc) printf("␣unc");
      else if (p→i ≡ sync) printf("␣sync");
      printf("␣(age␣%d)\n", ticks.l − p→stamp);
    }
  }
}
```

252. The entire present state of the pipeline computation can be visualized by printing first the write buffer, then the reorder buffer, then the fetch buffer. This shows the progression of results from oldest to youngest, from sizzling hot to ice cold.

⟨ External prototypes 9 ⟩ +≡

```
Extern void print_pipe ARGS((void));
```

253. ⟨ External routines 10 ⟩ +≡

```
void print_pipe()
{
  print_write_buffer();
  print_reorder_buffer();
  print_fetch_buffer();
}
```

254. The *write_search* routine looks to see if any instructions ahead of a given place in the *mem* list of the reorder buffer are storing into a given physical address, or if there's a pending instruction in the write buffer for that address. If so, it returns a pointer to the value to be written. If not, it returns Λ. If the answer is currently unknown, because at least one possibly relevant physical address has not yet been computed, the subroutine returns the special code value DUNNO.

The search starts at the *x.up* field of a control block for a store instruction, otherwise at the *ptr_a* field of the control block, unless *ptr_a* points to a committed instruction.

The *i* field in the write buffer is usually *st* or *pst*, inherited from a store or partial store command. It may also be *sync* (from SYNC 1 or SYNC 3) or *stunc* (from STUNC).

```
#define DUNNO ((octa *) 1)    /* an impossible non-Λ pointer */
```

⟨ Internal prototypes 13 ⟩ +≡

```
static octa *write_search ARGS((control *, octa));
```

255. ⟨ Subroutines 14 ⟩ +≡

```
static octa *write_search(ctl, addr)
    control *ctl;
    octa addr;
{ register specnode *p = (ctl→mem_x ? ctl→x.up : (specnode *) ctl→ptr_a);
  register write_node *q = write_tail;
  addr.l &= -8;
  if (p ≡ &mem) goto qloop;
  if (p > &hot→x ∧ ctl ≤ hot) goto qloop;      /* already committed */
  if (p < &ctl→x ∧ (ctl ≤ hot ∨ p > &hot→x)) goto qloop;
  for ( ; p ≠ &mem; p = p→up) {
    if (p→addr.h ≡ (tetra) -1) return DUNNO;
    if ((p→addr.l & -8) ≡ addr.l ∧ p→addr.h ≡ addr.h) return (p→known ? &(p→o) : DUNNO);
  }
qloop: for ( ; ; ) {
    if (q ≡ write_head) return Λ;
    if (q ≡ wbuf_top) q = wbuf_bot; else q++;
    if (q→addr.l ≡ addr.l ∧ q→addr.h ≡ addr.h) return &(q→o);
  }
}
```

256. When we're committing new data to memory, we can update an existing item in the write buffer if it has the same physical address, unless that item is already in the process of being written out. Increasing the value of *holding_time* will increase the chance that this economy is possible, but it will also increase the number of buffered items when writes are to different locations.

A store instruction that sets any of the eight interrupt bits `rwxnkbsp` will not affect memory, even if it doesn't cause an interrupt.

When "store" is followed by "store uncached" at the same address, or vice versa, we believe the most recent hint.

```
⟨ Commit to memory if possible, otherwise break 256 ⟩ ≡
  { register write_node *q = write_tail;
    if (hot→interrupt & (F_BIT + #ff)) goto done_with_write;
    if (hot→x.addr.h & #ffff0000) {
      if (hot→op ≥ STB ∧ hot→op < STSF) q→size = (hot→op & #f) ≫ 2;
      else if (hot→op ≥ STSF ∧ hot→op < STCO) q→size = 2;
      else q→size = 3;
    }
    if (hot→i ≠ sync)
      for ( ; ; ) {
        if (q ≡ write_head) break;
        if (q ≡ wbuf_top) q = wbuf_bot; else q++;
        if (q→i ≡ sync) break;
        if (q→addr.l ≡ hot→x.addr.l ∧ q→addr.h ≡ hot→x.addr.h ∧ (q ≠ write_head ∨ ¬wbuf_lock))
          goto addr_found;
      }
    { register write_node *p = (write_tail ≡ wbuf_bot ? wbuf_top : write_tail − 1);
      if (p ≡ write_head) break;      /* the write buffer is full */
      q = write_tail; write_tail = p;
      q→addr = hot→x.addr;
    }
  addr_found: q→o = hot→x.o;
    q→stamp = ticks.l;
    q→i = hot→i;
  done_with_write: spec_rem(&(hot→x));
    mem_slots++;
  }
```

This code is used in section 146.

257. A special coroutine whose duty is to empty the write buffer is always active. It holds the *wbuf_lock* while it is writing the contents of *write_head*. It holds *Dcache→fill_lock* while waiting for the D-cache to fill a block.

```
⟨ Cases for control of special coroutines 126 ⟩ +≡
case write_from_wbuf: p = (cacheblock *) data→ptr_b;
  switch (data→state) {
  case 4: ⟨ Forward the new data past the D-cache if it is write-through 263 ⟩;
    data→state = 5;
  case 5: if (write_head ≡ wbuf_bot) write_head = wbuf_top; else write_head−−;
  write_restart: data→state = 0;
  case 0: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
    if (write_head ≡ write_tail) wait(1);      /* write buffer is empty */
    if (write_head→i ≡ sync) ⟨ Ignore the item in write_head 264 ⟩;
    if (write_head→addr.h & #ffff0000) goto mem_direct;
    if ((int)(ticks.l − write_head→stamp) < holding_time ∧ ¬speed_lock) wait(1);      /* data too raw */
    if (¬Dcache) goto mem_direct;      /* not cached */
    if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);      /* D-cache busy */
    startup(&Dcache→reader[j], Dcache→access_time);
    ⟨ Write the data into the D-cache and set state = 4, if there's a cache hit 262 ⟩;
    data→state = ((Dcache→mode & WRITE_ALLOC) ∧ write_head→i ≠ stunc ? 1 : 3);
    wait(Dcache→access_time);
  case 1: ⟨ Try to put the contents of location write_head→addr into the D-cache 261 ⟩;
    data→state = 2; sleep;
  case 2: data→state = 0; sleep;      /* wake up when the D-cache has the block */
  case 3: ⟨ Handle write-around when writing to the D-cache 259 ⟩;
  mem_direct: ⟨ Write directly from write_head to memory 260 ⟩;
  }
```

258.
```
⟨ Local variables 12 ⟩ +≡
  register cacheblock *p, *q;
```

259. The granularity is guaranteed to be 8 in write-around mode (see *MMIX_config*). Although an uncached store will not be stored in the D-cache (unless it hits in the D-cache), it will go into a secondary cache.

```
⟨ Handle write-around when writing to the D-cache 259 ⟩ ≡
  if (Dcache→filler.next) goto write_restart;
  if ((Scache ∧ Scache→lock) ∨ (¬Scache ∧ mem_lock)) goto write_restart;
  if (Dcache→flusher.next) wait(1);
  Dcache→outbuf.tag.h = write_head→addr.h;
  Dcache→outbuf.tag.l = write_head→addr.l & (−Dcache→bb);
  for (j = 0; j < Dcache→bb ≫ Dcache→g; j++) Dcache→outbuf.dirty[j] = false;
  Dcache→outbuf.data[(write_head→addr.l & (Dcache→bb − 1)) ≫ 3] = write_head→o;
  Dcache→outbuf.dirty[(write_head→addr.l & (Dcache→bb − 1)) ≫ Dcache→g] = true;
  Dcache→outbuf.rank = Dcache→gg;      /* this many valid bytes */
  set_lock(self, wbuf_lock);
  startup(&Dcache→flusher, Dcache→copy_out_time);
  data→state = 5; wait(Dcache→copy_out_time);
```

This code is used in section 257.

260. ⟨ Write directly from *write_head* to memory 260 ⟩ ≡
```
if (mem_lock) wait(1);
set_lock(self, wbuf_lock);
set_lock(&mem_locker, mem_lock);      /* a coroutine of type vanish */
startup(&mem_locker, mem_addr_time + mem_write_time);
if (write_head→addr.h & #ffff0000) spec_write(write_head→addr, write_head→o, write_head→size);
else mem_write(write_head→addr, write_head→o);
data→state = 5; wait(mem_addr_time + mem_write_time);
```
This code is used in section 257.

261. A subtlety needs to be mentioned here: While we're trying to update the D-cache, another instruction might be filling the same cache block (although not because of the same physical address). Therefore we **goto** *write_restart* here instead of saying *wait*(1).

⟨ Try to put the contents of location *write_head→addr* into the D-cache 261 ⟩ ≡
```
if (Dcache→filler.next) goto write_restart;
if ((Scache ∧ Scache→lock) ∨ (¬Scache ∧ mem_lock)) goto write_restart;
p = alloc_slot(Dcache, write_head→addr);
if (¬p) goto write_restart;
if (Scache) set_lock(&Dcache→filler, Scache→lock)
else set_lock(&Dcache→filler, mem_lock);
set_lock(self, Dcache→fill_lock);
data→ptr_b = Dcache→filler_ctl.ptr_b = (void *) p;
Dcache→filler_ctl.z.o = write_head→addr;
startup(&Dcache→filler, Scache ? Scache→access_time : mem_addr_time);
```
This code is used in section 257.

262. Here it is assumed that *Dcache→access_time* is enough to search the D-cache and update one octabyte in case of a hit. The D-cache is not locked, since other coroutines that might be simultaneously reading the D-cache are not going to use the octabyte that changes. Perhaps the simulator is being too lenient here.

⟨ Write the data into the D-cache and set *state* = 4, if there's a cache hit 262 ⟩ ≡
```
p = cache_search(Dcache, write_head→addr);
if (p) {
  p = use_and_fix(Dcache, p);
  set_lock(self, wbuf_lock);
  data→ptr_b = (void *) p;
  p→data[(write_head→addr.l & (Dcache→bb − 1)) ≫ 3] = write_head→o;
  p→dirty[(write_head→addr.l & (Dcache→bb − 1)) ≫ Dcache→g] = true;
  data→state = 4; wait(Dcache→access_time);
}
```
This code is used in section 257.

263. ⟨ Forward the new data past the D-cache if it is write-through 263 ⟩ ≡
```
if ((Dcache→mode & WRITE_BACK) ≡ 0) {      /* write-through */
  if (Dcache→flusher.next) wait(1);
  flush_cache(Dcache, p, true);
}
```
This code is used in section 257.

264. ⟨ Ignore the item in *write_head* 264 ⟩ ≡

```
{
  set_lock(self, wbuf_lock);
  data→state = 5;
  wait(1);
}
```

This code is used in section 257.

265. Loading and storing. A RISC machine is often said to have a "load/store architecture," perhaps because loading and storing are among the most difficult things a RISC machine is called upon to do.

We want memory accesses to be efficient, so we try to access the D-cache at the same time as we are translating a virtual address via the DT-cache. Usually we hit in both caches, but numerous cases must be dealt with when we miss. Is there an elegant way to handle all the contingencies? Alas, the author of this program was unable to think of anything better than to throw lots of code at the problem — knowing full well that such a spaghetti-like approach is fraught with possibilities for error.

Instructions like `LDO` x, y, z operate in two pipeline stages. The first stage computes the virtual address $y + z$, waiting if necessary until y and z are both known; then it starts to access the necessary caches. In the second stage we ascertain the corresponding physical address and hopefully find the data in the cache (or in the speculative *mem* list or the write buffer).

An instruction like `STB` x, y, z shares some of the computation of `LDO` x, y, z, because only one byte is being stored but the other seven bytes must be found in the cache. In this case, however, x is treated as an input, and *mem* is the output. The second stage of a store command can begin even though x is not known during the first stage.

Here's what we do at the beginning of stage 1.

```
#define ld_st_launch 7  /* state when load/store command has its memory address */
⟨ Cases to compute the virtual address of a memory operation 265 ⟩ ≡
case preld: case prest: case prego:
  data→z.o = incr(data→z.o, data→xx & -(data→i ≡ prego ? Icache : Dcache)→bb);
    /* (I hope the adder is fast enough) */
case ld: case ldunc: case ldvts: case st: case pst: case syncd: case syncid: start_ld_st:
  data→y.o = oplus(data→y.o, data→z.o);
  data→state = ld_st_launch; goto switch1;
case ldptp: case ldpte: if (data→y.o.h) goto start_ld_st;
  data→x.o = zero_octa; data→x.known = true; goto die;  /* page table fault */
```

This code is used in section 132.

266.

```
#define PRW_BITS (data→i < st ? PR_BIT : data→i ≡ pst ? PR_BIT + PW_BIT : (data→i ≡
        syncid ∧ (data→loc.h & sign_bit)) ? 0 : PW_BIT)
⟨ Special cases for states in the first stage 266 ⟩ ≡
case ld_st_launch: if ((self + 1)→next) wait(1);  /* second stage must be clear */
  ⟨ Handle special cases for operations like prego and ldvts 289 ⟩;
  if (data→y.o.h & sign_bit) ⟨ Do load/store stage 1 with known physical address 271 ⟩;
  if (page_bad) {
    if (data→i < preld ∨ data→i ≡ st ∨ data→i ≡ pst) data→interrupt |= PRW_BITS;
    goto fin_ex;
  }
  if (DTcache→lock ∨ (j = get_reader(DTcache)) < 0) wait(1);
  startup(&DTcache→reader[j], DTcache→access_time);
  ⟨ Look up the address in the DT-cache, and also in the D-cache if possible 267 ⟩;
  pass_after(DTcache→access_time); goto passit;
```

See also sections 310, 326, 360, and 363.

This code is used in section 130.

267. When stage 2 of a load/store command begins, the state will depend on what transpired in stage 1. For example, *data→state* will be *DT_miss* if the virtual address key can't be found in the DT-cache; then stage 2 will have to compute the physical address the hard way.

The *data→state* will be *DT_hit* if the physical address is known via the DT-cache, but the data may or may not be in the D-cache. The *data→state* will be *hit_and_miss* if the DT-cache hits and the D-cache doesn't. And *data→state* will be *ld_ready* if *data→x.o* is the desired octabyte (for example, if both caches hit).

```
#define DT_miss 10 /* second stage state when DT-cache doesn't hold the key */
#define DT_hit 11 /* second stage state when physical address is known */
#define hit_and_miss 12 /* second stage state when D-cache misses */
#define ld_ready 13 /* second stage state when data has been read */
#define st_ready 14 /* second stage state when data needn't be read */
#define prest_win 15 /* second stage state when we can fill a block with zeroes */
```

⟨ Look up the address in the DT-cache, and also in the D-cache if possible 267 ⟩ ≡

```
p = cache_search(DTcache, trans_key(data→y.o));
if (¬Dcache ∨ Dcache→lock ∨ (j = get_reader(Dcache)) < 0 ∨ (data→i ≥ st ∧ data→i ≤ syncid))
  ⟨ Do load/store stage 1 without D-cache lookup 270 ⟩;
startup(&Dcache→reader[j], Dcache→access_time);
if (p) ⟨ Do a simultaneous lookup in the D-cache 268 ⟩
else data→state = DT_miss;
```

This code is used in section 266.

268. We assume that it is possible to look up a virtual address in the DT-cache at the same time as we look for a corresponding physical address in the D-cache, provided that the lower $b+c$ bits of the two addresses are the same. (They will always be the same if $b+c \le page\_s$; otherwise the operating system can try to make them the same by "page coloring" whenever possible.) If both caches hit, the physical address is known in $\max(DTcache{\rightarrow}access\_time, Dcache{\rightarrow}access\_time)$ cycles.

If the lower $b+c$ bits of the virtual and physical addresses differ, the machine will not know this until the DT-cache has hit. Therefore we simulate the operation of accessing the D-cache, but we go to *DT_hit* instead of to *hit_and_miss* because the D-cache will experience a spurious miss.

```
#define max(x,y) ((x) < (y) ? (y) : (x))
⟨ Do a simultaneous lookup in the D-cache 268 ⟩ ≡
  { octa *m;
    p = use_and_fix(DTcache,p), data→z.o = p→data[0];
    ⟨ Check the protection bits and get the physical address 269 ⟩;
    m = write_search(data, data→z.o);
    if (m ≡ DUNNO) data→state = DT_hit;
    else if (m) data→x.o = *m, data→state = ld_ready;
    else if (Dcache→b + Dcache→c > page_s ∧
          ((data→y.o.l ⊕ data→z.o.l) & ((Dcache→bb ≪ Dcache→c) − (1 ≪ page_s)))) data→state = DT_hit;
        /* spurious D-cache lookup */
    else {
      q = cache_search(Dcache, data→z.o);
      if (q) {
        if (data→i ≡ ldunc) q = demote_and_fix(Dcache, q);
        else q = use_and_fix(Dcache, q);
        data→x.o = q→data[(data→z.o.l & (Dcache→bb − 1)) ≫ 3];
        data→state = ld_ready;
      } else data→state = hit_and_miss;
    }
    pass_after(max(DTcache→access_time, Dcache→access_time));
    goto passit;
  }
```

This code is used in section 267.

269. The protection bits $p_r p_w p_x$ in a translation cache are shifted five positions right from the interrupt codes `PR_BIT`, `PW_BIT`, `PX_BIT`. If the data is protected, we abort the load/store operation immediately; this protects the privacy of other users.

```
⟨ Check the protection bits and get the physical address 269 ⟩ ≡
  if (data→stack_alert) {
    if (data→z.o.l & (PW_BIT ≫ PROT_OFFSET)) data→stack_alert = false;
    else data→z.o = g[rC].o;      /* use the continuation page for stack overflow */
  }
  j = PRW_BITS;
  if (((data→z.o.l ≪ PROT_OFFSET) & j) ≠ (unsigned int) j) {
    if (data→i ≡ syncd ∨ data→i ≡ syncid) goto sync_check;
    if (data→i ≠ preld ∧ data→i ≠ prest) data→interrupt |= j & ∼(data→z.o.l ≪ PROT_OFFSET);
    data→stack_alert = false;
    goto fin_ex;
  }
  data→z.o = phys_addr(data→y.o, data→z.o);
```

This code is used in sections 268, 270, and 272.

270. ⟨ Do load/store stage 1 without D-cache lookup 270 ⟩ ≡

```
{ octa *m;
  if (p) {
    p = use_and_fix(DTcache, p), data→z.o = p→data[0];
    ⟨ Check the protection bits and get the physical address 269 ⟩;
    if (data→i ≥ st ∧ data→i ≤ syncid) data→state = st_ready;
    else {
      m = write_search(data, data→z.o);
      if (m ∧ m ≠ DUNNO) data→x.o = *m, data→state = ld_ready;
      else data→state = DT_hit;
    }
  } else data→state = DT_miss;
  pass_after(DTcache→access_time); goto passit;
}
```

This code is used in section 267.

271. ⟨ Do load/store stage 1 with known physical address 271 ⟩ ≡

```
{ octa *m;
  if (¬(data→loc.h & sign_bit)) {
    if (data→i ≡ syncd ∨ data→i ≡ syncid) goto sync_check;
    if (data→i ≠ preld ∧ data→i ≠ prest) data→interrupt |= N_BIT;
    goto fin_ex;
  }
  data→z.o = data→y.o; data→z.o.h −= sign_bit;
  if (data→z.o.h & #ffff0000) {
    switch (data→i) {
    case ldvts: case preld: case prest: case prego: case syncd: case syncid: goto fin_ex;
    case ld: case ldunc: if (mem_lock) wait(1);
      if (data→op < LDSF) i = (data→op & #f) ≫ 2;
      else if (data→op < CSWAP) i = 2;
      else i = 3;
      data→x.o = spec_read(data→z.o, i);
      goto make_ld_ready;
    case pst:
      if ((data→op ⊕ CSWAP) ≤ 1) {
        data→x.o = spec_read(data→z.o, 3); goto make_ld_ready;
      }
      data→x.o = zero_octa;
    case st: data→state = st_ready; pass_after(1); goto passit;
    default: ;
    }
  } else if (data→i ≥ st ∧ data→i ≤ syncid) {
    data→state = st_ready; pass_after(1); goto passit;
  }
  m = write_search(data, data→z.o);
  if (m) {
    if (m ≡ DUNNO) data→state = DT_hit;
    else data→x.o = *m, data→state = ld_ready;
    pass_after(1); goto passit;
  } else if (¬Dcache) {
    if (mem_lock) wait(1);
    data→x.o = mem_read(data→z.o);
  make_ld_ready: set_lock(&mem_locker, mem_lock);
    data→state = ld_ready;
    startup(&mem_locker, mem_addr_time + mem_read_time);
    pass_after(mem_addr_time + mem_read_time); goto passit;
  }
  if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) {
    data→state = DT_hit; pass_after(1); goto passit;
  }
  startup(&Dcache→reader[j], Dcache→access_time);
  q = cache_search(Dcache, data→z.o);
  if (q) {
    if (data→i ≡ ldunc) q = demote_and_fix(Dcache, q);
    else q = use_and_fix(Dcache, q);
    data→x.o = q→data[(data→z.o.l & (Dcache→bb − 1)) ≫ 3];
    data→state = ld_ready;
  } else data→state = hit_and_miss;
```

```
    pass_after(Dcache→access_time); goto passit;
  }
```

This code is used in section 266.

272. The program for the second stage is, likewise, rather long-winded, yet quite similar to the cache manipulations we have already seen several times.

Several instructions might be trying to fill the DT-cache for the same page. (A similar situation faced us in the *write_from_wbuf* coroutine.) The second stage therefore needs to do some translation cache searching just as the first stage did. In this stage, however, we don't go all out for speed, because DT-cache misses are rare.

```
#define DT_retry 8 /* second stage state when DT-cache should be searched again */
#define got_DT 9 /* second stage state when DT-cache entry has been computed */
⟨ Special cases for states in later stages 272 ⟩ ≡
square_one: data→state = DT_retry;
case DT_retry: if (DTcache→lock ∨ (j = get_reader(DTcache)) < 0) wait(1);
  startup(&DTcache→reader[j], DTcache→access_time);
  p = cache_search(DTcache, trans_key(data→y.o));
  if (p) {
    p = use_and_fix(DTcache, p), data→z.o = p→data[0];
    ⟨ Check the protection bits and get the physical address 269 ⟩;
    if (data→i ≥ st ∧ data→i ≤ syncid) data→state = st_ready;
    else data→state = DT_hit;
  } else data→state = DT_miss;
  wait(DTcache→access_time);
case DT_miss: if (DTcache→filler.next) {
    if (data→i ≡ preld ∨ data→i ≡ prest) goto fin_ex; else goto square_one; }
  if (no_hardware_PT ∨ page_f) {
    if (data→i ≡ preld ∨ data→i ≡ prest) goto fin_ex; else goto emulate_virt; }
  p = alloc_slot(DTcache, trans_key(data→y.o));
  if (¬p) goto square_one;
  data→ptr_b = DTcache→filler_ctl.ptr_b = (void *) p;
  DTcache→filler_ctl.y.o = data→y.o;
  set_lock(self, DTcache→fill_lock);
  startup(&DTcache→filler, 1);
  data→state = got_DT;
  if (data→i ≡ preld ∨ data→i ≡ prest) goto fin_ex; else sleep;
case got_DT: release_lock(self, DTcache→fill_lock);
  ⟨ Check the protection bits and get the physical address 269 ⟩;
  if (data→i ≥ st ∧ data→i ≤ syncid) goto finish_store;
      /* otherwise we fall through to ld_retry below */
```

See also sections 273, 276, 279, 280, 299, 311, 354, 364, and 370.

This code is used in section 135.

273. The second stage might also want to fill the D-cache (and perhaps the S-cache) as we get the data.

Several load instructions might be trying to fill the same cache block. So we should go back and look in the D-cache again if we miss and cannot allocate a slot immediately.

A `PRELD` or `PREST` instruction, which is just a "hint," doesn't do anything more if the caches are already busy.

```
⟨ Special cases for states in later stages 272 ⟩ +≡
ld_retry: data→state = DT_hit;
case DT_hit: if (data→i ≡ preld ∨ data→i ≡ prest) goto fin_ex;
  ⟨ Check for a hit in pending writes 278 ⟩;
  if ((data→z.o.h & #ffff0000) ∨ ¬Dcache) ⟨ Do load/store stage 2 without D-cache lookup 277 ⟩;
  if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);
  startup(&Dcache→reader[j], Dcache→access_time);
  q = cache_search(Dcache, data→z.o);
  if (q) {
    if (data→i ≡ ldunc) q = demote_and_fix(Dcache, q);
    else q = use_and_fix(Dcache, q);
    data→x.o = q→data[(data→z.o.l & (Dcache→bb − 1)) ≫ 3];
    data→state = ld_ready;
  } else data→state = hit_and_miss;
  wait(Dcache→access_time);
case hit_and_miss: if (data→i ≡ ldunc) goto avoid_D;
  ⟨ Try to get the contents of location data→z.o in the D-cache 274 ⟩;
```

274.

```
⟨ Try to get the contents of location data→z.o in the D-cache 274 ⟩ ≡
  ⟨ Check for prest with a fully spanned cache block 275 ⟩;
  if (Dcache→filler.next) goto ld_retry;
  if ((Scache ∧ Scache→lock) ∨ (¬Scache ∧ mem_lock)) goto ld_retry;
  q = alloc_slot(Dcache, data→z.o);
  if (¬q) goto ld_retry;
  if (Scache) set_lock(&Dcache→filler, Scache→lock)
  else set_lock(&Dcache→filler, mem_lock);
  set_lock(self, Dcache→fill_lock);
  data→ptr_b = Dcache→filler_ctl.ptr_b = (void *) q;
  Dcache→filler_ctl.z.o = data→z.o;
  startup(&Dcache→filler, Scache ? Scache→access_time : mem_addr_time);
  data→state = ld_ready;
  if (data→i ≡ preld ∨ data→i ≡ prest) goto fin_ex; else sleep;
```

This code is used in section 273.

275. If a *prest* instruction makes it to the hot seat, we have been assured by the user of `PREST` that the current values of bytes in virtual addresses $data{\rightarrow}y.o-(data{\rightarrow}xx\,\&\,-Dcache{\rightarrow}bb)$ through $data{\rightarrow}y.o+(data{\rightarrow}xx\,\&\,(Dcache{\rightarrow}bb-1))$ are irrelevant. Hence we can pretend that we know they are zero. This is advantageous if it saves us from filling a cache block from the S-cache or from memory.

```
⟨ Check for prest with a fully spanned cache block 275 ⟩ ≡
  if (data→i ≡ prest ∧
        (data→xx ≥ Dcache→bb ∨ ((data→y.o.l & (Dcache→bb − 1)) ≡ 0)) ∧
        ((data→y.o.l + (data→xx & (Dcache→bb − 1)) + 1) ⊕ data→y.o.l) ≥ (unsigned int) Dcache→bb)
    goto prest_span;
```

This code is used in section 274.

276. ⟨ Special cases for states in later stages 272 ⟩ +≡

```
prest_span: data→state = prest_win;
case prest_win: if (data ≠ old_hot ∨ Dlocker.next) wait(1);
  if (Dcache→lock) goto fin_ex;
  q = alloc_slot(Dcache, data→z.o);      /* OK if Dcache→filler is busy */
  if (q) {
    clean_block(Dcache, q);
    q→tag = data→z.o; q→tag.l &= -Dcache→bb;
    set_lock(&Dlocker, Dcache→lock);
    startup(&Dlocker, Dcache→copy_in_time);
  }
  goto fin_ex;
```

277. ⟨ Do load/store stage 2 without D-cache lookup 277 ⟩ ≡

```
{
avoid_D: if (mem_lock) wait(1);
  set_lock(&mem_locker, mem_lock);
  startup(&mem_locker, mem_addr_time + mem_read_time);
  data→x.o = mem_read(data→z.o);
  data→state = ld_ready; wait(mem_addr_time + mem_read_time);
}
```

This code is used in section 273.

278. ⟨ Check for a hit in pending writes 278 ⟩ ≡

```
{
  octa *m = write_search(data, data→z.o);
  if (m ≡ DUNNO) wait(1);
  if (m) {
    data→x.o = *m;
    data→state = ld_ready;
    wait(1);
  }
}
```

This code is used in section 273.

279. The requested octabyte will arrive sooner or later in *data→x.o*. Then a load instruction is almost done, except that we might need to massage the input a little bit.

```
⟨ Special cases for states in later stages 272 ⟩ +≡
case ld_ready: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (data→i ≥ st) goto finish_store;
  switch (data→op ≫ 1) {
  case LDB ≫ 1: case LDBU ≫ 1: j = (data→z.o.l & #7) ≪ 3; i = 56; goto fin_ld;
  case LDW ≫ 1: case LDWU ≫ 1: j = (data→z.o.l & #6) ≪ 3; i = 48; goto fin_ld;
  case LDT ≫ 1: case LDTU ≫ 1: j = (data→z.o.l & #4) ≪ 3; i = 32;
  fin_ld: data→x.o = shift_right(shift_left(data→x.o, j), i, data→op & #2);
  default: goto fin_ex;
  case LDHT ≫ 1: if (data→z.o.l & 4) data→x.o.h = data→x.o.l;
    data→x.o.l = 0; goto fin_ex;
  case LDSF ≫ 1: if (data→z.o.l & 4) data→x.o.h = data→x.o.l;
    if ((data→x.o.h & #7f800000) ≡ 0 ∧ (data→x.o.h & #7fffff)) {
      data→x.o = load_sf(data→x.o.h);
      data→state = 3; wait(denin_penalty);
    }
    else data→x.o = load_sf(data→x.o.h); goto fin_ex;
  case LDPTP ≫ 1: if ((data→x.o.h & sign_bit) ≡ 0 ∨ (data→x.o.l & #1ff8) ≠ page_n) data→x.o = zero_octa;
    else data→x.o.l &= −(1 ≪ 13);
    goto fin_ex;
  case LDPTE ≫ 1: if ((data→x.o.l & #1ff8) ≠ page_n) data→x.o = zero_octa;
    else data→x.o = incr(oandn(data→x.o, page_mask), data→x.o.l & #7);
    data→x.o.h &= #ffff; goto fin_ex;
  case UNSAVE ≫ 1: ⟨ Handle an internal UNSAVE when it's time to load 336 ⟩;
  }
```

280.

```
⟨ Special cases for states in later stages 272 ⟩ +≡
finish_store: data→state = st_ready;
case st_ready: switch (data→i) {
  case st: case pst: ⟨ Finish a store command 281 ⟩;
  case syncd: data→b.o.l = (Dcache ? Dcache→bb : 8192); goto do_syncd;
  case syncid: data→b.o.l = (Icache ? Icache→bb : 8192);
    if (Dcache ∧ (unsigned int) Dcache→bb < data→b.o.l) data→b.o.l = Dcache→bb;
    goto do_syncid;
  default: ;
  }
```

281. Store instructions have an extra complication, because some of them need to check for overflow.

```
⟨ Finish a store command 281 ⟩ ≡
  data→x.addr = data→z.o;
  if (data→b.p) wait(1);
  switch (data→op ≫ 1) {
  case STUNC ≫ 1: data→i = stunc;
  default: data→x.o = data→b.o; goto fin_ex;
  case STSF ≫ 1: set_round; data→b.o.h = store_sf(data→b.o);
    data→interrupt |= exceptions;
    if ((data→b.o.h & #7f800000) ≡ 0 ∧ (data→b.o.h & #7fffff)) {
      if (data→z.o.l & 4) data→x.o.l = data→b.o.h;
      else data→x.o.h = data→b.o.h;
      data→state = 3; wait(denout_penalty);
    }
  case STHT ≫ 1: if (data→z.o.l & 4) data→x.o.l = data→b.o.h;
    else data→x.o.h = data→b.o.h;
    goto fin_ex;
  case STB ≫ 1: case STBU ≫ 1: j = (data→z.o.l & #7) ≪ 3; i = 56; goto fin_st;
  case STW ≫ 1: case STWU ≫ 1: j = (data→z.o.l & #6) ≪ 3; i = 48; goto fin_st;
  case STT ≫ 1: case STTU ≫ 1: j = (data→z.o.l & #4) ≪ 3; i = 32;
  fin_st: ⟨ Insert data→b.o into the proper field of data→x.o, checking for arithmetic exceptions if signed 282 ⟩;
    goto fin_ex;
  case CSWAP ≫ 1: ⟨ Finish a CSWAP 283 ⟩;
  case SAVE ≫ 1: ⟨ Handle an internal SAVE when it's time to store 342 ⟩;
  }
```

This code is used in section 280.

282.

```
⟨ Insert data→b.o into the proper field of data→x.o, checking for arithmetic exceptions if signed 282 ⟩ ≡
  {
    octa mask;
    if (¬(data→op & 2)) { octa before, after;
      before = data→b.o; after = shift_right(shift_left(data→b.o, i), i, 0);
      if (before.l ≠ after.l ∨ before.h ≠ after.h) data→interrupt |= V_BIT;
    }
    mask = shift_right(shift_left(neg_one, i), j, 1);
    data→b.o = shift_right(shift_left(data→b.o, i), j, 1);
    data→x.o.h ⊕= mask.h & (data→x.o.h ⊕ data→b.o.h);
    data→x.o.l ⊕= mask.l & (data→x.o.l ⊕ data→b.o.l);
  }
```

This code is used in section 281.

283. The `CSWAP` operation has four inputs ($X, $Y, $Z, rP) as well as three outputs ($X, $M_8[A]$, rP). To keep from exceeding the capacity of the control blocks in our pipeline, we wait until this instruction reaches the hot seat, thereby allowing us non-speculative access to rP.

```
⟨ Finish a CSWAP 283 ⟩ ≡
  if (data ≠ old_hot) wait(1);
  if (data→x.o.h ≡ g[rP].o.h ∧ data→x.o.l ≡ g[rP].o.l) {
    data→a.o.l = 1;      /* data→a.o.h is zero */
    data→x.o = data→b.o;
  } else {
    g[rP].o = data→x.o;      /* data→a.o is zero */
    if (verbose & issue_bit) {
      printf("␣setting␣rP=");  print_octa(g[rP].o);  printf("\n");
    }
  }
  data→i = cswap;      /* cosmetic change, affects the trace output only */
  goto fin_ex;
```

This code is used in section 281.

284. The fetch stage. Now that we've mastered the most difficult memory operations, we can relax and apply our knowledge to the slightly simpler task of filling the fetch buffer. Fetching is like loading/storing, except that we use the I-cache instead of the D-cache. It's slightly simpler because the I-cache is read-only. Further simplifications would be possible if there were no `PREGO` instruction, because there is only one fetch unit. However, we want to implement `PREGO` with reasonable efficiency, in order to see if that instruction is worthwhile; so we include the complications of simultaneous I-cache and IT-cache readers, which we have already implemented for the D-cache and DT-cache.

The fetch coroutine is always present, as the one and only coroutine with *stage* number zero.

In normal circumstances, the fetch coroutine accesses a cache block containing the instruction whose virtual address is given by *inst_ptr* (the instruction pointer), and transfers up to *fetch_max* instructions from that block to the fetch buffer. Complications arise if the instruction isn't in the cache, or if we can't translate the virtual address because of a miss in the IT-cache. Moreover, *inst_ptr* is a **spec** variable whose value might not even be known; if *inst_ptr*.*p* is nonnull, we don't know what to fetch.

```
⟨ External variables 4 ⟩ +≡
  Extern spec inst_ptr;      /* the instruction pointer (aka program counter) */
  Extern octa *fetched;      /* buffer for incoming instructions */
```

285. The fetch coroutine usually begins a cycle in state *fetch_ready*, with the most recently fetched octabytes in positions *fetch_lo*, *fetch_lo* + 1, ..., *fetch_hi* − 1 of a buffer called *fetched*. Once that buffer has been exhausted, the coroutine reverts to state 0; with luck, the buffer might have more data by the time the next cycle rolls around.

```
⟨ Global variables 20 ⟩ +≡
  int fetch_lo, fetch_hi;      /* the active region of that buffer */
  coroutine fetch_co;
  control fetch_ctl;
```

286.
```
⟨ Initialize everything 22 ⟩ +≡
  fetch_co.ctl = &fetch_ctl;
  fetch_co.name = "Fetch";
  fetch_ctl.go.o.l = 4;
  startup(&fetch_co, 1);
```

287.
```
⟨ Restart the fetch coroutine 287 ⟩ ≡
  if (fetch_co.lockloc) *(fetch_co.lockloc) = Λ, fetch_co.lockloc = Λ;
  unschedule(&fetch_co);
  startup(&fetch_co, 1);
```

This code is used in sections 85, 160, 308, 309, and 316.

288. Some of the actions here are done not only by the fetcher but also by the first and second stages of a *prego* operation.

```
#define wait_or_pass(t)
        if (data→i ≡ prego) { pass_after(t); goto passit; }
        else wait(t)
⟨Simulate an action of the fetch coroutine 288⟩ ≡
switch0: switch (data→state) {
  new_fetch: data→state = 0;
  case 0: ⟨Wait, if necessary, until the instruction pointer is known 290⟩;
    data→y.o = inst_ptr.o;
    data→state = 1; data→interrupt = 0; data→x.o = data→z.o = zero_octa;
  case 1: start_fetch: if (data→y.o.h & sign_bit) ⟨Begin fetch with known physical address 296⟩;
    if (page_bad) goto bad_fetch;
    if (ITcache→lock ∨ (j = get_reader(ITcache)) < 0) wait(1);
    startup(&ITcache→reader[j], ITcache→access_time);
    ⟨Look up the address in the IT-cache, and also in the I-cache if possible 291⟩;
    wait_or_pass(ITcache→access_time);
    ⟨Other cases for the fetch coroutine 298⟩
  }
```

This code is used in section 125.

289. ⟨Handle special cases for operations like *prego* and *ldvts* 289⟩ ≡

```
if (data→i ≡ prego) goto start_fetch;
```

See also section 352.

This code is used in section 266.

290. ⟨Wait, if necessary, until the instruction pointer is known 290⟩ ≡

```
if (inst_ptr.p) {
  if (inst_ptr.p ≠ UNKNOWN_SPEC ∧ inst_ptr.p→known) inst_ptr.o = inst_ptr.p→o, inst_ptr.p = Λ;
  wait(1);
}
```

This code is used in section 288.

291.

```
#define got_IT 19 /* state when IT-cache entry has been computed */
#define IT_miss 20 /* state when IT-cache doesn't hold the key */
#define IT_hit 21 /* state when physical instruction address is known */
#define Ihit_and_miss 22 /* state when I-cache misses */
#define fetch_ready 23 /* state when instructions have been read */
#define got_one 24 /* state when a "preview" octabyte is ready */
⟨Look up the address in the IT-cache, and also in the I-cache if possible 291⟩ ≡
  p = cache_search(ITcache, trans_key(data→y.o));
  if (¬Icache ∨ Icache→lock ∨ (j = get_reader(Icache)) < 0) ⟨Begin fetch without I-cache lookup 295⟩;
  startup(&Icache→reader[j], Icache→access_time);
  if (p) ⟨Do a simultaneous lookup in the I-cache 292⟩
  else data→state = IT_miss;
```

This code is used in section 288.

292. We assume that it is possible to look up a virtual address in the IT-cache at the same time as we look for a corresponding physical address in the I-cache, provided that the lower $b+c$ bits of the two addresses are the same. (See the remarks about "page coloring," when we made similar assumptions about the DT-cache and D-cache.)

```
⟨ Do a simultaneous lookup in the I-cache 292 ⟩ ≡
  {
    ⟨ Update IT-cache usage and check the protection bits 293 ⟩;
    data→z.o = phys_addr(data→y.o, p→data[0]);
    if (Icache→b + Icache→c > page_s ∧
          ((data→y.o.l ⊕ data→z.o.l) & ((Icache→bb ≪ Icache→c) − (1 ≪ page_s)))) data→state = IT_hit;
        /* spurious I-cache lookup */
    else {
      q = cache_search(Icache, data→z.o);
      if (q) {
        q = use_and_fix(Icache, q);
        ⟨ Copy the data from block q to fetched 294 ⟩;
        data→state = fetch_ready;
      } else data→state = Ihit_and_miss;
    }
    wait_or_pass(max(ITcache→access_time, Icache→access_time));
  }
```

This code is used in section 291.

293.

```
⟨ Update IT-cache usage and check the protection bits 293 ⟩ ≡
  p = use_and_fix(ITcache, p);
  if (¬(p→data[0].l & (PX_BIT ≫ PROT_OFFSET))) goto bad_fetch;
```

This code is used in sections 292 and 295.

294. At this point *inst_ptr.o* equals *data→y.o*.

```
⟨ Copy the data from block q to fetched 294 ⟩ ≡
  if (data→i ≠ prego) {
    for (j = 0; j < Icache→bb ≫ 3; j++) fetched[j] = q→data[j];
    fetch_lo = (inst_ptr.o.l & (Icache→bb − 1)) ≫ 3;
    fetch_hi = Icache→bb ≫ 3;
  }
```

This code is used in sections 292 and 296.

295.

```
⟨ Begin fetch without I-cache lookup 295 ⟩ ≡
  {
    if (p) {
      ⟨ Update IT-cache usage and check the protection bits 293 ⟩;
      data→z.o = phys_addr(data→y.o, p→data[0]);
      data→state = IT_hit;
    } else data→state = IT_miss;
    wait_or_pass(ITcache→access_time);
  }
```

This code is used in section 291.

296. ⟨ Begin fetch with known physical address 296 ⟩ ≡

```
{
  if (data→i ≡ prego ∧ ¬(data→loc.h & sign_bit)) goto fin_ex;
  data→z.o = data→y.o; data→z.o.h −= sign_bit;
known_phys: if (data→z.o.h & #ffff0000) goto bad_fetch;
  if (¬Icache) ⟨ Read from memory into fetched 297 ⟩;
  if (Icache→lock ∨ (j = get_reader(Icache)) < 0) {
    data→state = IT_hit; wait_or_pass(1);
  }
  startup(&Icache→reader[j], Icache→access_time);
  q = cache_search(Icache, data→z.o);
  if (q) {
    q = use_and_fix(Icache, q);
    ⟨ Copy the data from block q to fetched 294 ⟩;
    data→state = fetch_ready;
  } else data→state = Ihit_and_miss;
  wait_or_pass(Icache→access_time);
}
```

This code is used in section 288.

297. ⟨ Read from memory into fetched 297 ⟩ ≡

```
{ octa addr;
  addr = data→z.o;
  if (mem_lock) wait(1);
  set_lock(&mem_locker, mem_lock);
  startup(&mem_locker, mem_addr_time + mem_read_time);
  addr.l &= −(bus_words ≪ 3);
  fetched[0] = mem_read(addr);
  for (j = 1; j < bus_words; j++) fetched[j] = mem_hash[last_h].chunk[((addr.l & #ffff) ≫ 3) + j];
  fetch_lo = (data→z.o.l ≫ 3) & (bus_words − 1); fetch_hi = bus_words;
  data→state = fetch_ready;
  wait(mem_addr_time + mem_read_time);
}
```

This code is used in section 296.

298. ⟨ Other cases for the fetch coroutine 298 ⟩ ≡

```
case IT_miss: if (ITcache→filler.next) {
    if (data→i ≡ prego) goto fin_ex; else wait(1); }
  if (no_hardware_PT ∨ page_f) ⟨ Insert dummy instruction for page table emulation 302 ⟩;
  p = alloc_slot(ITcache, trans_key(data→y.o));
  if (¬p)      /* hey, it was present after all */
  {
    if (data→i ≡ prego) goto fin_ex; else goto new_fetch; }
  data→ptr_b = ITcache→filler_ctl.ptr_b = (void *) p;
  ITcache→filler_ctl.y.o = data→y.o;
  set_lock(self, ITcache→fill_lock);
  startup(&ITcache→filler, 1);
  data→state = got_IT;
  if (data→i ≡ prego) goto fin_ex; else sleep;
case got_IT: release_lock(self, ITcache→fill_lock);
  if (¬(data→z.o.l & (PX_BIT ≫ PROT_OFFSET))) goto bad_fetch;
  data→z.o = phys_addr(data→y.o, data→z.o);
fetch_retry: data→state = IT_hit;
case IT_hit: if (data→i ≡ prego) goto fin_ex; else goto known_phys;
case Ihit_and_miss: ⟨ Try to get the contents of location data→z.o in the I-cache 300 ⟩;
```

See also section 301.

This code is used in section 288.

299. ⟨ Special cases for states in later stages 272 ⟩ +≡

```
case IT_miss: case Ihit_and_miss: case IT_hit: case fetch_ready: goto switch0;
```

300. ⟨ Try to get the contents of location data→z.o in the I-cache 300 ⟩ ≡

```
  if (Icache→filler.next) goto fetch_retry;
  if ((Scache ∧ Scache→lock) ∨ (¬Scache ∧ mem_lock)) goto fetch_retry;
  q = alloc_slot(Icache, data→z.o);
  if (¬q) goto fetch_retry;
  if (Scache) set_lock(&Icache→filler, Scache→lock)
  else set_lock(&Icache→filler, mem_lock);
  set_lock(self, Icache→fill_lock);
  data→ptr_b = Icache→filler_ctl.ptr_b = (void *) q;
  Icache→filler_ctl.z.o = data→z.o;
  startup(&Icache→filler, Scache ? Scache→access_time : mem_addr_time);
  data→state = got_one;
  if (data→i ≡ prego) goto fin_ex; else sleep;
```

This code is used in section 298.

301. The I-cache filler will wake us up with the octabyte we want, before it has filled the entire cache block. In that case we can fetch one or two instructions before the rest of the block has been loaded.

```
⟨ Other cases for the fetch coroutine 298 ⟩ +≡
bad_fetch: if (data→i ≡ prego) goto fin_ex;
  data→interrupt |= PX_BIT;
swym_one: fetched[0].h = fetched[0].l = SWYM ≪ 24;
  goto fetch_one;
case got_one: fetched[0] = data→x.o;      /* a "preview" of the new cache data */
fetch_one: fetch_lo = 0; fetch_hi = 1;
  data→state = fetch_ready;
case fetch_ready: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (data→i ≡ prego) goto fin_ex;
  for (j = 0; j < fetch_max; j++) {
    register fetch *new_tail;
    if (tail ≡ fetch_bot) new_tail = fetch_top;
    else new_tail = tail − 1;
    if (new_tail ≡ head) break;      /* fetch buffer is full */
    ⟨ Install a new instruction into the tail position 304 ⟩;
    tail = new_tail;
    if (sleepy) {
      sleepy = false; sleep;
    }
    inst_ptr.o = incr(inst_ptr.o, 4);
    if (fetch_lo ≡ fetch_hi) goto new_fetch;
  }
  wait(1);
```

302. ⟨ Insert dummy instruction for page table emulation 302 ⟩ ≡

```
{
  if (cache_search(ITcache, trans_key(inst_ptr.o))) goto new_fetch;
  data→interrupt |= F_BIT;
  sleepy = true;
  goto swym_one;
}
```

This code is used in section 298.

303. ⟨ Global variables 20 ⟩ +≡

```
bool sleepy;      /* have we just emitted the page table emulation call? */
```

304. At this point we check for egregiously invalid instructions. (Sometimes the dispatcher will actually allow such instructions to occupy the fetch buffer, for internally generated commands.)

```
⟨ Install a new instruction into the tail position 304 ⟩ ≡
  tail→loc = inst_ptr.o;
  if (inst_ptr.o.l & 4) tail→inst = fetched[fetch_lo++].l;
  else tail→inst = fetched[fetch_lo].h;
  tail→interrupt = data→interrupt;
  i = tail→inst ≫ 24;
  if (i ≥ RESUME ∧ i ≤ SYNC ∧ (tail→inst & bad_inst_mask[i − RESUME])) tail→interrupt |= B_BIT;
  tail→noted = false;
  if (inst_ptr.o.l ≡ breakpoint.l ∧ inst_ptr.o.h ≡ breakpoint.h) breakpoint_hit = true;
```

This code is used in section 301.

305. The commands `RESUME`, `SAVE`, `UNSAVE`, and `SYNC` should not have nonzero bits in the positions defined here.

⟨ Global variables 20 ⟩ +≡
```
int bad_inst_mask[4] = {#fffffe, #ffff, #ffff00, #fffff8};
```

306. Interrupts. The scariest thing about the design of a pipelined machine is the existence of interrupts, which disrupt the smooth flow of a computation in ways that are difficult to anticipate. Fortunately, however, the discipline of a reorder buffer, which forces instructions to be committed in order, allows us to deal with interrupts in a fairly natural way. Our solution to the problems of dynamic scheduling and speculative execution therefore solves the interrupt problem as well.

`MMIX` has three kinds of interrupts, which show up as bit codes in the *interrupt* field when an instruction is ready to be committed: `H_BIT` invokes a trip handler, for `TRIP` instructions and arithmetic exceptions; `F_BIT` invokes a forced-trap handler, for `TRAP` instructions and unimplemented instructions that need to be emulated in software; `E_BIT` invokes a dynamic-trap handler, for external interrupts like I/O signals or for internal interrupts caused by improper instructions. In all three cases, the pipeline control has already been redirected to fetch new instructions starting at the correct handler address by the time an interrupted instruction is ready to be committed.

307. Most instructions come to the following part of the program, if they have finished execution with any 1s among the eight trip bits or the eight trap bits.

If the trip bits aren't all zero, we want to update the event bits of rA, or perform an enabled trip handler, or both. If the trap bits are nonzero, we need to hold onto them until we get to the hot seat, when they will be joined with the bits of rQ and probably cause an interrupt. A load or store instruction with nonzero trap bits will be nullified, not committed.

Underflow that is exact and not enabled is ignored, in accordance with the IEEE standard conventions. (This applies also to underflow triggered by `RESUME_SET`.)

```
#define is_load_store(i)  (i ≥ ld ∧ i ≤ cswap)
⟨ Handle interrupt at end of execution stage 307 ⟩ ≡
  {
    if ((data→interrupt & #ff) ∧ is_load_store(data→i)) goto state_5;
    j = data→interrupt & #ff00;
    data→interrupt −= j;
    if ((j & (U_BIT + X_BIT)) ≡ U_BIT ∧ ¬(data→ra.o.l & U_BIT)) j &= ∼U_BIT;
    data→arith_exc = (j & ∼data→ra.o.l) ≫ 8;
    if (j & data→ra.o.l) ⟨ Prepare for exceptional trip handler 308 ⟩;
    if (data→interrupt & #ff) goto state_5;
  }
```

This code is used in section 144.

308. Since execution is speculative, an exceptional condition might not be part of the "real" computation. Indeed, the present coroutine might have already been deissued.

```
⟨ Prepare for exceptional trip handler 308 ⟩ ≡
  {
    i = issued_between(data, cool);
    if (i < deissues) goto die;
    deissues = i;
    old_tail = tail = head; resuming = 0;      /* clear the fetch buffer */
    ⟨ Restart the fetch coroutine 287 ⟩;
    cool_hist = data→hist;
    for (i = j & data→ra.o.l, m = 16; ¬(i & D_BIT); i ≪= 1, m += 16) ;
    data→arith_exc |= (j & ∼(#10000 ≫ (m ≫ 4))) ≫ 8;      /* trips taken are not logged as events */
    data→go.o.h = 0, data→go.o.l = m;
    inst_ptr.o = data→go.o, inst_ptr.p = Λ;
    data→interrupt |= H_BIT;
    goto state_4;
  }
```

This code is used in section 307.

309. ⟨ Prepare to emulate the page translation 309 ⟩ ≡

```
  i = issued_between(data, cool);
  if (i < deissues) goto die;
  deissues = i;
  old_tail = tail = head; resuming = 0;      /* clear the fetch buffer */
  ⟨ Restart the fetch coroutine 287 ⟩;
  cool_hist = data→hist;
  inst_ptr.p = UNKNOWN_SPEC;
  data→interrupt |= F_BIT;
```

This code is used in section 310.

310. We need to stop dispatching when calling a trip handler from within the reorder buffer, lest we issue an instruction that uses $g[255]$ or rB as an operand.

```
⟨ Special cases for states in the first stage 266 ⟩ +≡
emulate_virt: ⟨ Prepare to emulate the page translation 309 ⟩;
state_4: data→state = 4;
case 4: if (dispatch_lock) wait(1);
  set_lock(self, dispatch_lock);
state_5: data→state = 5;
case 5: if (data ≠ old_hot) wait(1);
  if ((data→interrupt & F_BIT) ∧ data→i ≠ trap) {
    inst_ptr.o = g[rT].o, inst_ptr.p = Λ;
    if (is_load_store(data→i)) nullifying = true;
  }
  if (data→interrupt & #ff) {
    g[rQ].o.h |= data→interrupt & #ff;
    new_Q.h |= data→interrupt & #ff;
    if (verbose & issue_bit) {
      printf("␣setting␣rQ="); print_octa(g[rQ].o); printf("\n");
    }
  }
  goto die;
```

311. The instructions of the previous section appear in the switch for coroutine stage 1 only. We need to use them also in later stages.

```
⟨ Special cases for states in later stages 272 ⟩ +≡
case 4: goto state_4;
case 5: goto state_5;
```

312.
```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case trap: if ((flags[op] & X_is_dest_bit) ∧ cool→xx < cool_G ∧ cool→xx ≥ cool_L) goto increase_L;
  if (¬g[rT].up→known ∨ ¬g[rJ].up→known) goto stall;
  inst_ptr = specval(&g[rT]);      /* traps and emulated ops */
  cool→need_b = true, cool→b = specval(&g[255]);
case trip:
  if (¬g[rJ].up→known) goto stall;
  cool→ren_x = true, spec_install(&g[255], &cool→x);
  cool→x.known = true, cool→x.o = g[rJ].up→o;
  if (i ≡ trip) cool→go.o = zero_octa;
  cool→ren_a = true, spec_install(&g[i ≡ trap ? rBB : rB], &cool→a); break;
```

313.
```
⟨ Cases for stage 1 execution 155 ⟩ +≡
case trap: data→interrupt |= F_BIT; data→a.o = data→b.o; goto fin_ex;
case trip: data→interrupt |= H_BIT; data→a.o = data→b.o; goto fin_ex;
```

314. The following check is performed at the beginning of every cycle. An instruction in the hot seat can be externally interrupted only if it is ready to be committed and not already marked for tripping or trapping.

```
⟨ Check for external interrupt 314 ⟩ ≡
  g[rI].o = incr(g[rI].o, −1);
  if (g[rI].o.l ≡ 0 ∧ g[rI].o.h ≡ 0) {
    g[rQ].o.l |= INTERVAL_TIMEOUT, new_Q.l |= INTERVAL_TIMEOUT;
    if (verbose & issue_bit) {
      printf("␣setting␣rQ="); print_octa(g[rQ].o); printf("\n");
    }
  }
  trying_to_interrupt = false;
  if (((g[rQ].o.h & g[rK].o.h) ∨ (g[rQ].o.l & g[rK].o.l)) ∧ cool ≠ hot ∧
      ¬(hot→interrupt & (E_BIT + F_BIT + H_BIT)) ∧ ¬doing_interrupt ∧
      ¬(hot→i ≡ resum)) {
    if (hot→owner) trying_to_interrupt = true;
    else {
      hot→interrupt |= E_BIT;
      ⟨ Deissue all but the hottest command 316 ⟩;
      inst_ptr.o = g[rTT].o; inst_ptr.p = Λ;
    }
  }
```

This code is used in section 64.

315.
```
⟨ Global variables 20 ⟩ +≡
  bool trying_to_interrupt;      /* encouraging interruptible operations to pause */
  bool nullifying;      /* stopping dispatch to nullify a load/store command */
```

316. It's possible that the command in the hot seat has been deissued, but only if the simulator has done so at the user's request. Otherwise the test '$i \ge deissues$' here will always succeed.

The value of *cool_hist* becomes flaky here. We could try to keep it strictly up to date, but the unpredictable nature of external interrupts suggests that we are better off leaving it alone. (It's only a heuristic for branch prediction, and a sufficiently strong prediction will survive one-time glitches due to interrupts.)

```
⟨ Deissue all but the hottest command 316 ⟩ ≡
  i = issued_between(hot, cool);
  if (i ≥ deissues) {
    deissues = i;
    tail = head; resuming = 0;      /* clear the fetch buffer */
    ⟨ Restart the fetch coroutine 287 ⟩;
    if (is_load_store(hot→i)) nullifying = true;
  }
```

This code is used in section 314.

317. Even though an interrupted instruction has officially been either "committed" or "nullified," it stays in the hot seat for two or three extra cycles, while we save enough of the machine state to resume the computation later.

```
⟨ Begin an interruption and break 317 ⟩ ≡
  {
    if (¬(hot→interrupt & H_BIT)) g[rK].o = zero_octa;      /* trap */
    if (((hot→interrupt & H_BIT) ∧ hot→i ≠ trip) ∨
          ((hot→interrupt & F_BIT) ∧ hot→i ≠ trap) ∨
          (hot→interrupt & E_BIT)) doing_interrupt = 3, suppress_dispatch = true;
    else doing_interrupt = 2;      /* trip or trap started by dispatcher */
    break;
  }
```

This code is used in section 146.

318. If a memory failure occurs, we should set rF here, either in case 2 or case 1. The simulator doesn't do anything with rF at present.

```
⟨ Perform one cycle of the interrupt preparations 318 ⟩ ≡
  switch (doing_interrupt--) {
  case 3: ⟨ Set resumption registers (rB, $255) or (rBB, $255) 319 ⟩; break;
  case 2: ⟨ Set resumption registers (rW, rX) or (rWW, rXX) 320 ⟩; break;
  case 1: ⟨ Set resumption registers (rY, rZ) or (rYY, rZZ) 321 ⟩;
    if (hot ≡ reorder_bot) hot = reorder_top; else hot--;
    break;
  }
```

This code is used in section 64.

319. ⟨ Set resumption registers (rB, $255) or (rBB, $255) 319 ⟩ ≡

```
j = hot→interrupt & H_BIT;
g[j ? rB : rBB].o = g[255].o;
g[255].o = g[rJ].o;
if (verbose & issue_bit) {
  if (j) {
    printf("␣setting␣rB=");  print_octa(g[rB].o);
  } else {
    printf("␣setting␣rBB=");  print_octa(g[rBB].o);
  }
  printf(",␣$255=");  print_octa(g[255].o);  printf("\n");
}
```

This code is used in section 318.

320. Here's where we manufacture the "ropcodes" for resumption.

```
#define RESUME_AGAIN 0    /* repeat the command in rX as if in location rW − 4 */
#define RESUME_CONT 1    /* same, but substitute rY and rZ for operands */
#define RESUME_SET 2    /* set register $X to rZ */
#define RESUME_TRANS 3    /* install (rY, rZ) into IT-cache or DT-cache, then RESUME_AGAIN */
#define pack_bytes(a, b, c, d)  ((((((unsigned)(a) ≪ 8) + (b)) ≪ 8) + (c)) ≪ 8) + (d)
```

⟨ Set resumption registers (rW, rX) or (rWW, rXX) 320 ⟩ ≡

```
j = pack_bytes(hot→op, hot→xx, hot→yy, hot→zz);
if (hot→interrupt & H_BIT) {    /* trip */
  g[rW].o = incr(hot→loc, 4);
  g[rX].o.h = sign_bit, g[rX].o.l = j;
  if (verbose & issue_bit) {
    printf("␣setting␣rW=");  print_octa(g[rW].o);
    printf(",␣rX=");  print_octa(g[rX].o);  printf("\n");
  }
} else {    /* trap */
  g[rWW].o = hot→go.o;
  g[rXX].o.l = j;
  if (hot→interrupt & F_BIT) {    /* forced */
    if (hot→i ≠ trap) j = RESUME_TRANS;    /* emulate page translation */
    else if (hot→op ≡ TRAP) j = #80;    /* TRAP */
    else if (flags[hot→op] & X_is_dest_bit) j = RESUME_SET;    /* emulation */
    else j = #80;    /* emulation when r[X] is not a destination */
  } else {    /* dynamic */
    if (hot→interim)
      j = (hot→i ≡ frem ∨ hot→i ≡ syncd ∨ hot→i ≡ syncid ? RESUME_CONT : RESUME_AGAIN);
    else if (is_load_store(hot→i)) j = RESUME_AGAIN;
    else j = #80;    /* normal external interruption */
  }
  g[rXX].o.h = (j ≪ 24) + (hot→interrupt & #ff);
  if (verbose & issue_bit) {
    printf("␣setting␣rWW=");  print_octa(g[rWW].o);
    printf(",␣rXX=");  print_octa(g[rXX].o);  printf("\n");
  }
}
```

This code is used in section 318.

321. ⟨ Set resumption registers (rY, rZ) or (rYY, rZZ) 321 ⟩ ≡

```
j = hot→interrupt & H_BIT;
if ((hot→interrupt & F_BIT) ∧ hot→op ≡ SWYM) g[rYY].o = hot→go.o;
else g[j ? rY : rYY].o = hot→y.o;
if (hot→i ≡ st ∨ hot→i ≡ pst) g[j ? rZ : rZZ].o = hot→x.o;
else g[j ? rZ : rZZ].o = hot→z.o;
if (verbose & issue_bit) {
  if (j) {
    printf("␣setting␣rY=");  print_octa(g[rY].o);
    printf(",␣rZ=");  print_octa(g[rZ].o);  printf("\n");
  } else {
    printf("␣setting␣rYY=");  print_octa(g[rYY].o);
    printf(",␣rZZ=");  print_octa(g[rZZ].o);  printf("\n");
  }
}
```

This code is used in section 318.

322. Whew; we've successfully interrupted the computation. The remaining task is to restart it again, as transparently as possible.

The RESUME instruction waits for the pipeline to drain, because it has to do such drastic things. For example, an interrupt may be occurring at this very moment, changing the registers needed for resumption.

⟨ Special cases of instruction dispatch 117 ⟩ +≡

```
case resume: if (cool ≠ old_hot) goto stall;
  inst_ptr = specval(&g[cool→zz ? rWW : rW]);
  if (¬(cool→loc.h & sign_bit)) {
    if (cool→zz) cool→interrupt |= K_BIT;
    else if (inst_ptr.o.h & sign_bit) cool→interrupt |= P_BIT;
  }
  if (cool→interrupt) {
    inst_ptr.o = incr(cool→loc, 4);  cool→i = noop;
  } else {
    cool→go.o = inst_ptr.o;
    if (cool→zz) {
      ⟨ Magically do an I/O operation, if cool→loc is rT 372 ⟩;
      cool→ren_a = true, spec_install(&g[rK], &cool→a);
      cool→a.known = true, cool→a.o = g[255].o;
      cool→ren_x = true, spec_install(&g[255], &cool→x);
      cool→x.known = true, cool→x.o = g[rBB].o;
    }
    cool→b = specval(&g[cool→zz ? rXX : rX]);
    if (¬(cool→b.o.h & sign_bit)) ⟨ Resume an interrupted operation 323 ⟩;
  } break;
```

323. Here we set *cool→i* = *resum*, since we want to issue another instruction after the RESUME itself.

The restrictions on inserted instructions are designed to ensure that those instructions will be the very next ones issued. (If, for example, an *incgamma* instruction were necessary, it might cause a page fault and we'd lose the operand values for RESUME_SET or RESUME_CONT.)

A subtle point arises here: If RESUME_TRANS is being used to compute the page translation of virtual address zero, we don't want to execute the dummy SWYM instruction from virtual address −4! So we avoid the SWYM altogether.

```
⟨ Resume an interrupted operation 323 ⟩ ≡
  {
    cool→xx = cool→b.o.h ≫ 24, cool→i = resum;
    head→loc = incr(inst_ptr.o, −4);
    switch (cool→xx) {
    case RESUME_SET: cool→b.o.l = (SETH ≪ 24) + (cool→b.o.l & #ff0000);
      head→interrupt |= cool→b.o.h & #ff00;
      resuming = 2;
    case RESUME_CONT: resuming += 1 + cool→zz;
      if (((cool→b.o.l ≫ 24) & #fa) ≠ #b8) {      /* not syncd or syncid */
        m = cool→b.o.l ≫ 28;
        if ((1 ≪ m) & #8f30) goto bad_resume;
        m = (cool→b.o.l ≫ 16) & #ff;
        if (m ≥ cool_L ∧ m < cool_G) goto bad_resume;
      }
    case RESUME_AGAIN: resume_again: head→inst = cool→b.o.l;
      m = head→inst ≫ 24;
      if (m ≡ RESUME) goto bad_resume;      /* avoid uninterruptible loop */
      if (¬cool→zz ∧ m > RESUME ∧ m ≤ SYNC ∧ (head→inst & bad_inst_mask[m − RESUME]))
        head→interrupt |= B_BIT;
      head→noted = false; break;
    case RESUME_TRANS: if (cool→zz) {
        cool→y = specval(&g[rYY]), cool→z = specval(&g[rZZ]);
        if ((cool→b.o.l ≫ 24) ≠ SWYM) goto resume_again;
        cool→i = resume; break;      /* see "subtle point" above */
      }
    default: bad_resume: cool→interrupt |= B_BIT, cool→i = noop;
      resuming = 0; break;
    }
  }
```

This code is used in section 322.

324. ⟨ Insert special operands when resuming an interrupted operation 324 ⟩ ≡

```
{
  if (resuming & 1) {
    cool→y = specval(&g[rY]);
    cool→z = specval(&g[rZ]);
  } else {
    cool→y = specval(&g[rYY]);
    cool→z = specval(&g[rZZ]);
  }
  if (resuming ≥ 3) {      /* RESUME_SET */
    cool→need_ra = true, cool→ra = specval(&g[rA]);
  }
  cool→usage = false;
}
```

This code is used in section 103.

325. **#define** *do_resume_trans* 17 /* *state* for performing RESUME_TRANS actions */

⟨ Cases for stage 1 execution 155 ⟩ +≡

```
case resume: case resum: if (data→xx ≠ RESUME_TRANS) goto fin_ex;
  data→ptr_a = (void *)((data→b.o.l ≫ 24) ≡ SWYM ? ITcache : DTcache);
  data→state = do_resume_trans;
  data→z.o = incr(oandn(data→z.o, page_mask), data→z.o.l & 7);
  data→z.o.h &= #ffff;
  goto resume_trans;
```

326. ⟨ Special cases for states in the first stage 266 ⟩ +≡

```
case do_resume_trans: resume_trans:
  { register cache *c = (cache *) data→ptr_a;
    if (c→lock) wait(1);
    if (c→filler.next) wait(1);
    p = alloc_slot(c, trans_key(data→y.o));
    if (p) {
      c→filler_ctl.ptr_b = (void *) p;
      c→filler_ctl.y.o = data→y.o;
      c→filler_ctl.b.o = data→z.o;
      c→filler_ctl.state = 1;
      schedule(&c→filler, c→access_time, 1);
    }
    goto fin_ex;
  }
```

327. Administrative operations. The internal instructions that handle the register stack simply reduce to things we already know how to do. (Well, the internal instructions for saving and unsaving do sometimes lead to special cases, based on *data→op*; for the most part, though, the necessary mechanisms are already present.)

```
⟨ Cases for stage 1 execution 155 ⟩ +≡
case noop: if (data→interrupt & F_BIT) goto emulate_virt;
case incrl: case unsave: goto fin_ex;
case jmp: case pushj: data→go.o = data→z.o;
  goto fin_ex;
case sav: if (¬(data→mem_x)) goto fin_ex;
case incgamma: case save: data→i = st;
  goto switch1;
case decgamma: case unsav: data→i = ld;
  goto switch1;
```

328. We can GET special registers ≥ 21 (that is, rA, rF, rP, rW–rZ, or rWW–rZZ) only in the hot seat, because those registers are implicit outputs of many instructions.

The same applies to rK, since it is changed by TRAP and by emulated instructions.

Likewise, rQ must not be prematurely gotten.

```
⟨ Cases for stage 1 execution 155 ⟩ +≡
case get: if (data→zz ≥ 21 ∨ data→zz ≡ rK ∨ data→zz ≡ rQ) {
    if (data ≠ old_hot) wait(1);
    data→z.o = g[data→zz].o;
  }
  data→x.o = data→z.o; goto fin_ex;
```

329. A PUT is, similarly, delayed in the cases that hold *dispatch_lock*. This program does not restrict the 1 bits that might be PUT into rQ, although the contents of that register can have drastic implications.

```
⟨ Cases for stage 1 execution 155 ⟩ +≡
case put: if (data→xx ≡ 8 ∨ (data→xx ≥ 15 ∧ data→xx ≤ 20)) {
    if (data ≠ old_hot) wait(1);
    switch (data→xx) {
    case rV: ⟨ Update the page variables 239 ⟩; break;
    case rQ: new_Q.h |= data→z.o.h & ∼g[rQ].o.h; new_Q.l |= data→z.o.l & ∼g[rQ].o.l;
      data→z.o.l |= new_Q.l; data→z.o.h |= new_Q.h; break;
    case rL: if (data→z.o.h ≠ 0) data→z.o.h = 0, data→z.o.l = g[rL].o.l;
      else if (data→z.o.l > g[rL].o.l) data→z.o.l = g[rL].o.l;
    default: break;
    case rG: ⟨ Update rG 330 ⟩; break;
    }
  } else if (data→xx ≡ rA ∧ (data→z.o.h ≠ 0 ∨ data→z.o.l ≥ #40000))
    data→interrupt |= B_BIT, data→z.o.h = 0, data→z.o.l &= #3ffff;
  data→x.o = data→z.o; goto fin_ex;
```

330. When rG decreases, we assume that up to *commit_max* marginal registers can be zeroed during each clock cycle. (Remember that we're currently in the hot seat, and holding *dispatch_lock*.)

```
⟨ Update rG 330 ⟩ ≡
  if (data→z.o.h ≠ 0 ∨ data→z.o.l ≥ 256 ∨ data→z.o.l < g[rL].o.l ∨ data→z.o.l < 32)
    data→interrupt |= B_BIT, data→z.o = g[rG].o;
  else if (data→z.o.l < g[rG].o.l) {
    data→interim = true;      /* potentially interruptible */
    for (j = 0; j < commit_max; j++) {
      g[rG].o.l--;
      g[g[rG].o.l].o = zero_octa;
      if (data→z.o.l ≡ g[rG].o.l) break;
    }
    if (j ≡ commit_max) {
      if (¬trying_to_interrupt) wait(1);
    } else data→interim = false;
  }
```

This code is used in section 329.

331. Computed jumps put the desired destination address into the *go* field.

```
⟨ Cases for stage 1 execution 155 ⟩ +≡
case go: data→x.o = data→go.o; goto add_go;
case pop: data→x.o = data→y.o;
  data→y.o = data→b.o;      /* move rJ to y field */
case pushgo: add_go: data→go.o = oplus(data→y.o, data→z.o);
  if ((data→go.o.h & sign_bit) ∧ ¬(data→loc.h & sign_bit)) data→interrupt |= P_BIT;
  data→go.known = true; goto fin_ex;
```

332. The instruction `UNSAVE` z generates a sequence of internal instructions that accomplish the actual unsaving. This sequence is controlled by the instruction currently in the fetch buffer, which changes its X and Y fields until all global registers have been loaded. The first instructions of the sequence are `UNSAVE` $0,0,z$; `UNSAVE` $1,rZ,z-8$; `UNSAVE` $1,rY,z-16$; ...; `UNSAVE` $1,rB,z-96$; `UNSAVE` $2,255,z-104$; `UNSAVE` $2,254,z-112$; etc. If an interrupt occurs before these instructions have all been committed, the execution register will contain enough information to restart the process.

After the global registers have all been loaded, `UNSAVE` continues by acting rather like `POP`. An interrupt occurring during this last stage will find rS < rO; a context switch might then take us back to restoring the local registers again. But no information will be lost, even though the register from which we began unsaving has long since been replaced.

```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case unsave: if (cool→interrupt & B_BIT) cool→i = noop;
  else {
    cool→interim = true;
    op = LDOU;      /* this instruction needs to be handled by load/store unit */
    cool→i = unsav;
    switch (cool→xx) {
    case 0: if (cool→z.p) goto stall;
      ⟨ Set up the first phase of unsaving 334 ⟩; break;
    case 1: case 2: ⟨ Generate an instruction to unsave g[yy] 333 ⟩; break;
    case 3: cool→i = unsave, cool→interim = false, op = UNSAVE;
      goto pop_unsave;
    default: cool→interim = false, cool→i = noop, cool→interrupt |= B_BIT; break;
    }
  }
  break;      /* this takes us to dispatch_done */
```

333. ⟨ Generate an instruction to unsave $g[yy]$ 333 ⟩ ≡

```
cool→ren_x = true, spec_install(&g[cool→yy], &cool→x);
new_O = new_S = incr(cool_O, -1);
cool→z.o = shift_left(new_O, 3);
cool→ptr_a = (void *) mem.up;
```

This code is used in section 332.

334. ⟨ Set up the first phase of unsaving 334 ⟩ ≡

```
cool→ren_x = true, spec_install(&g[rG], &cool→x);
cool→ren_a = true, spec_install(&g[rA], &cool→a);
new_O = new_S = shift_right(cool→z.o, 3, 1);
cool→set_l = true, spec_install(&g[rL], &cool→rl);
cool→ptr_a = (void *) mem.up;
```

This code is used in section 332.

335. ⟨ Get ready for the next step of `UNSAVE` 335 ⟩ ≡

```
switch (cool→xx) {
case 0: head→inst = pack_bytes(UNSAVE, 1, rZ, 0); break;
case 1: if (cool→yy ≡ rP) head→inst = pack_bytes(UNSAVE, 1, rR, 0);
  else if (cool→yy ≡ 0) head→inst = pack_bytes(UNSAVE, 2, 255, 0);
  else head→inst = pack_bytes(UNSAVE, 1, cool→yy - 1, 0); break;
case 2: if (cool→yy ≡ cool_G) head→inst = pack_bytes(UNSAVE, 3, 0, 0);
  else head→inst = pack_bytes(UNSAVE, 2, cool→yy - 1, 0); break;
}
```

This code is used in section 81.

336. ⟨ Handle an internal `UNSAVE` when it's time to load 336 ⟩ ≡

```
if (data→xx ≡ 0) {
  data→a.o = data→x.o; data→a.o.h &= #ffffff; /* unsaved rA */
  data→x.o.l = data→x.o.h ≫ 24; data→x.o.h = 0; /* unsaved rG */
  if (data→a.o.h ∨ (data→a.o.l & #fffc0000)) {
    data→a.o.h = 0, data→a.o.l &= #3ffff; data→interrupt |= B_BIT;
  }
  if (data→x.o.l < 32) {
    data→x.o.l = 32; data→interrupt |= B_BIT;
  }
}
goto fin_ex;
```

This code is used in section 279.

337. Of course `SAVE` is handled essentially like `UNSAVE`, but backwards.

⟨ Special cases of instruction dispatch 117 ⟩ +≡

```
case save: if (cool→xx < cool_G) cool→interrupt |= B_BIT;
  if (cool→interrupt & B_BIT) cool→i = noop;
  else if (((cool_S.l − cool_O.l − cool_L − 1) & lring_mask) ≡ 0)
    ⟨ Insert an instruction to advance gamma 113 ⟩
  else {
    cool→interim = true;
    cool→i = sav;
    switch (cool→zz) {
    case 0: ⟨ Set up the first phase of saving 338 ⟩; break;
    case 1: if (cool_O.l ≠ cool_S.l) ⟨ Insert an instruction to advance gamma 113 ⟩
      cool→zz = 2; cool→yy = cool_G;
    case 2: case 3: ⟨ Generate an instruction to save g[yy] 339 ⟩; break;
    default: cool→interim = false, cool→i = noop, cool→interrupt |= B_BIT; break;
    }
  }
  break;
```

338. If an interrupt occurs during the first phase, say between two *incgamma* instructions, the value *cool→zz* = 1 will get things restarted properly. (Indeed, if context is saved and unsaved during the interrupt, many *incgamma* instructions may no longer be necessary.)

⟨ Set up the first phase of saving 338 ⟩ ≡

```
cool→zz = 1;
cool→ren_x = true, spec_install(&l[(cool_O.l + cool_L) & lring_mask], &cool→x);
cool→x.known = true, cool→x.o.h = 0, cool→x.o.l = cool_L;
cool→set_l = true, spec_install(&g[rL], &cool→rl);
new_O = incr(cool_O, cool_L + 1);
```

This code is used in section 337.

339. ⟨ Generate an instruction to save g[yy] 339 ⟩ ≡

```
op = STOU; /* this instruction needs to be handled by load/store unit */
cool→mem_x = true, spec_install(&mem, &cool→x);
cool→z.o = shift_left(cool_O, 3);
new_O = new_S = incr(cool_O, 1);
if (cool→zz ≡ 3 ∧ cool→yy > rZ) ⟨ Do the final SAVE 340 ⟩
else cool→b = specval(&g[cool→yy]);
```

This code is used in section 337.

340. The final `SAVE` instruction not only stores rG and rA, it also places the final address in global register X.

```
⟨ Do the final SAVE 340 ⟩ ≡
  {
    cool→i = save;
    cool→interim = false;
    cool→ren_a = true, spec_install(&g[cool→xx], &cool→a);
  }
```

This code is used in section 339.

341.
```
⟨ Get ready for the next step of SAVE 341 ⟩ ≡
  switch (cool→zz) {
  case 1: head→inst = pack_bytes(SAVE, cool→xx, 0, 1); break;
  case 2: if (cool→yy ≡ 255) head→inst = pack_bytes(SAVE, cool→xx, 0, 3);
    else head→inst = pack_bytes(SAVE, cool→xx, cool→yy + 1, 2); break;
  case 3: if (cool→yy ≡ rR) head→inst = pack_bytes(SAVE, cool→xx, rP, 3);
    else head→inst = pack_bytes(SAVE, cool→xx, cool→yy + 1, 3); break;
  }
```

This code is used in section 81.

342.
```
⟨ Handle an internal SAVE when it's time to store 342 ⟩ ≡
  {
    if (data→interim) data→x.o = data→b.o;
    else {
      if (data ≠ old_hot) wait(1);      /* we need the hottest value of rA */
      data→x.o.h = g[rG].o.l ≪ 24;
      data→x.o.l = g[rA].o.l;
      data→a.o = data→y.o;
    }
    goto fin_ex;
  }
```

This code is used in section 281.

343. More register-to-register ops. Now that we've finished most of the hard stuff, we can relax and fill in the holes that we left in the all-register parts of the execution stages.

First let's complete the fixed point arithmetic operations, by dispensing with multiplication and division.

```
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case mulu: data→x.o = omult(data→y.o, data→z.o);
  data→a.o = aux;
  goto quantify_mul;
case mul: data→x.o = signed_omult(data→y.o, data→z.o);
  if (overflow) data→interrupt |= V_BIT;
quantify_mul: aux = data→z.o;
  for (j = mul0; aux.l ∨ aux.h; j++) aux = shift_right(aux, 8, 1);
  data→i = j; break;      /* j is mul0 or mul1 or ... or mul8 */
case divu: data→x.o = odiv(data→b.o, data→y.o, data→z.o);
  data→a.o = aux; data→i = div; break;
case div: if (data→z.o.l ≡ 0 ∧ data→z.o.h ≡ 0) {
    data→interrupt |= D_BIT; data→a.o = data→y.o;
    data→i = set;      /* divide by zero needn't wait in the pipeline */
  } else {
    data→x.o = signed_odiv(data→y.o, data→z.o);
    if (overflow) data→interrupt |= V_BIT;
    data→a.o = aux;
  } break;
```

344. Next let's polish off the bitwise and bytewise operations.

```
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case sadd: data→x.o.l = count_bits(data→y.o.h & ∼data→z.o.h) + count_bits(data→y.o.l & ∼data→z.o.l);
  break;
case mor: data→x.o = bool_mult(data→y.o, data→z.o, data→op & #2); break;
case bdif: data→x.o.h = byte_diff(data→y.o.h, data→z.o.h);
  data→x.o.l = byte_diff(data→y.o.l, data→z.o.l); break;
case wdif: data→x.o.h = wyde_diff(data→y.o.h, data→z.o.h);
  data→x.o.l = wyde_diff(data→y.o.l, data→z.o.l); break;
case tdif: if (data→y.o.h > data→z.o.h) data→x.o.h = data→y.o.h − data→z.o.h;
tdif_l: if (data→y.o.l > data→z.o.l) data→x.o.l = data→y.o.l − data→z.o.l; break;
case odif: if (data→y.o.h > data→z.o.h) data→x.o = ominus(data→y.o, data→z.o);
  else if (data→y.o.h ≡ data→z.o.h) goto tdif_l;
  break;
```

345. The conditional set (`CS`) instructions are, rather surprisingly, more difficult to implement than the zero set (`ZS`) instructions, although the `ZS` instructions do more. The reason is that dynamic instruction dependencies are more complicated with `CS`. Consider, for example, the instructions

```
LDO x,a,b;  FDIV y,c,d;  CSZ y,x,0;  INCL y,1.
```

If the value of `x` is zero, the `INCL` instruction need not wait for the division to be completed. (We do not, however, abort the division in such a case; it might invoke a trip handler, or change the inexact bit, etc. Our policy is to treat common cases efficiently and to treat all cases correctly, but not to treat all cases with maximum efficiency.)

⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡

```
case zset: if (register_truth(data→y.o, data→op)) data→x.o = data→z.o;
      /* otherwise data→x.o is already zero */
  goto fin_ex;
case cset: if (register_truth(data→y.o, data→op)) data→x.o = data→z.o, data→b.p = Λ;
  else if (data→b.p ≡ Λ) data→x.o = data→b.o;
  else {
    data→state = 0; data→need_b = true; goto switch1;
  } break;
```

346. Floating point computations are mostly handled by the routines in MMIX-ARITH, which record anomalous events in the global variable *exceptions*. But we consider the operation trivial if an input is infinite or NaN; and we may need to increase the execution time when subnormals are present.

```
#define ROUND_OFF 1
#define ROUND_UP 2
#define ROUND_DOWN 3
#define ROUND_NEAR 4
#define is_subnormal(x) ((x.h & #7ff00000) ≡ 0 ∧ ((x.h & #fffff) ∨ x.l))
#define is_trivial(x) ((x.h & #7ff00000) ≡ #7ff00000)
#define set_round cur_round = (data→ra.o.l < #10000 ? ROUND_NEAR : data→ra.o.l ≫ 16)
⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡
case fadd: set_round; data→x.o = fplus(data→y.o, data→z.o);
fin_bflot: if (is_subnormal(data→y.o)) data→denin = denin_penalty;
fin_uflot: if (is_subnormal(data→x.o)) data→denout = denout_penalty;
fin_flot: if (is_subnormal(data→z.o)) data→denin = denin_penalty;
  data→interrupt |= exceptions;
  if (is_trivial(data→y.o) ∨ is_trivial(data→z.o)) goto fin_ex;
  if (data→i ≡ fsqrt ∧ (data→z.o.h & sign_bit)) goto fin_ex;
  break;
case fsub: data→a.o = data→z.o;
  if (fcomp(data→z.o, zero_octa) ≠ 2) data→a.o.h ⊕= sign_bit;
  set_round; data→x.o = fplus(data→y.o, data→a.o);
  data→i = fadd;  /* use pipeline times for addition */
  goto fin_bflot;
case fmul: set_round; data→x.o = fmult(data→y.o, data→z.o); goto fin_bflot;
case fdiv: set_round; data→x.o = fdivide(data→y.o, data→z.o); goto fin_bflot;
case fsqrt: set_round; data→x.o = froot(data→z.o, data→y.o.l); goto fin_uflot;
case fint: set_round; data→x.o = fintegerize(data→z.o, data→y.o.l); goto fin_uflot;
case fix: set_round; data→x.o = fixit(data→z.o, data→y.o.l);
  if (data→op & #2) exceptions &= ∼W_BIT;  /* unsigned case doesn't overflow */
  goto fin_flot;
case flot: set_round; data→x.o = floatit(data→z.o, data→y.o.l, data→op & #2, data→op & #4);
  data→interrupt |= exceptions; break;
```

347. ⟨ Special cases of instruction dispatch 117 ⟩ +≡

```
case fsqrt: case fint: case fix: case flot: if (cool→y.o.l > 4) goto illegal_inst;
  break;
```

348. ⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡

```
case feps: j = fepscomp(data→y.o, data→z.o, data→b.o, data→op ≠ FEQLE);
  if (j ≡ 2) data→i = fcmp;
  else if (is_subnormal(data→y.o) ∨ is_subnormal(data→z.o)) data→denin = denin_penalty;
  switch (data→op) {
  case FUNE: if (j ≡ 2) goto cmp_pos; else goto cmp_zero;
  case FEQLE: goto cmp_fin;
  case FCMPE: if (j) goto cmp_zero_or_invalid;
  default: ;
  }
case fcmp: j = fcomp(data→y.o, data→z.o);
  if (j < 0) goto cmp_neg;
cmp_fin: if (j ≡ 1) goto cmp_pos;
cmp_zero_or_invalid: if (j ≡ 2) data→interrupt |= I_BIT;
  goto cmp_zero;
case funeq: if (fcomp(data→y.o, data→z.o) ≡ (data→op ≡ FUN ? 2 : 0)) goto cmp_pos;
  else goto cmp_zero;
```

349. ⟨ External variables 4 ⟩ +≡

```
  Extern int frem_max;
  Extern int denin_penalty, denout_penalty;
```

350. The floating point remainder operation is especially interesting because it can be interrupted when it's in the hot seat.

⟨ Cases to compute the results of register-to-register operation 137 ⟩ +≡

```
case frem: if (is_trivial(data→y.o) ∨ is_trivial(data→z.o)) {
    data→x.o = fremstep(data→y.o, data→z.o, 2500);
    data→interrupt |= exceptions; goto fin_ex;
  }
  if ((self + 1)→next) wait(1);
  data→interim = true;
  j = 1;
  if (is_subnormal(data→y.o) ∨ is_subnormal(data→z.o)) j += denin_penalty;
  pass_after(j);
  goto passit;
```

351. ⟨ Begin execution of a stage-two operation 351 ⟩ ≡

```
  j = 1;
  if (data→i ≡ frem) {
    data→x.o = fremstep(data→y.o, data→z.o, frem_max);
    if (exceptions & E_BIT) {
      data→y.o = data→x.o;
      if (trying_to_interrupt ∧ data ≡ old_hot) goto fin_ex;
    } else {
      data→state = 3;
      data→interim = false;
      data→interrupt |= exceptions;
      if (is_subnormal(data→x.o)) j += denout_penalty;
    }
    wait(j);
  }
```

This code is used in section 135.

352. System operations. Finally we need to implement some operations for the operating system; then the hardware simulation will be done!

A `LDVTS` instruction is delayed until it reaches the hot seat, because it changes the IT and DT caches. The operating system should use `SYNC` after `LDVTS` if the effects are needed immediately; the system is also responsible for ensuring that the page table permission bits agree with the `LDVTS` permission bits when the latter are nonzero. (Also, if write permission is taken away from a page, the operating system must have previously used `SYNCD` to write out any dirty bytes that might have been cached from that page; `SYNCD` will be inoperative after write permission goes away.)

```
⟨ Handle special cases for operations like prego and ldvts 289 ⟩ +≡
  if (data→i ≡ ldvts) ⟨ Do stage 1 of LDVTS 353 ⟩;
```

353.
```
⟨ Do stage 1 of LDVTS 353 ⟩ ≡
  {
    if (data ≠ old_hot) wait(1);
    if (DTcache→lock ∨ (j = get_reader(DTcache)) < 0) wait(1);
    startup(&DTcache→reader[j], DTcache→access_time);
    data→z.o.h = 0, data→z.o.l = data→y.o.l & #7;
    p = cache_search(DTcache, data→y.o);      /* N.B.: Not trans_key(data→y.o) */
    if (p) {
      data→x.o.l = 2;
      if (data→z.o.l) {
        p = use_and_fix(DTcache, p);
        p→data[0].l = (p→data[0].l & −8) + data→z.o.l;
      } else {
        p = demote_and_fix(DTcache, p);
        p→tag.h |= sign_bit;      /* invalidate the tag */
      }
    }
    pass_after(DTcache→access_time); goto passit;
  }
```
This code is used in section 352.

354.
```
⟨ Special cases for states in later stages 272 ⟩ +≡
case ld_st_launch: if (ITcache→lock ∨ (j = get_reader(ITcache)) < 0) wait(1);
  startup(&ITcache→reader[j], ITcache→access_time);
  p = cache_search(ITcache, data→y.o);      /* N.B.: Not trans_key(data→y.o) */
  if (p) {
    data→x.o.l |= 1;
    if (data→z.o.l) {
      p = use_and_fix(ITcache, p);
      p→data[0].l = (p→data[0].l & −8) + data→z.o.l;
    } else {
      p = demote_and_fix(ITcache, p);
      p→tag.h |= sign_bit;      /* invalidate the tag */
    }
  }
  data→state = 3; wait(ITcache→access_time);
```

355. The `SYNC` operation interacts with the pipeline in interesting ways. `SYNC 0` and `SYNC 4` are the simplest; they just lock the dispatch and wait until they get to the hot seat, after which the pipeline has drained. `SYNC 1` and `SYNC 3` put a "barrier" into the write buffer so that subsequent store instructions will not merge with previous stores. `SYNC 2` and `SYNC 3` lock the dispatch until all previous load instructions have left the pipeline. `SYNC 5`, `SYNC 6`, and `SYNC 7` remove things from caches once they get to the hot seat.

```
⟨ Special cases of instruction dispatch 117 ⟩ +≡
case sync: if (cool→zz > 3) {
    if (¬(cool→loc.h & sign_bit)) goto privileged_inst;
    if (cool→zz ≡ 4) freeze_dispatch = true;
  } else {
    if (cool→zz ≠ 1) freeze_dispatch = true;
    if (cool→zz & 1) cool→mem_x = true, spec_install(&mem, &cool→x);
  } break;
```

356.

```
⟨ Cases for stage 1 execution 155 ⟩ +≡
case sync: switch (data→zz) {
  case 0: case 4: if (data ≠ old_hot) wait(1);
    halted = (data→zz ≠ 0); goto fin_ex;
  case 2: case 3: ⟨ Wait if there's an unfinished load ahead of us 357 ⟩;
    release_lock(self, dispatch_lock);
  case 1: data→x.addr = zero_octa; goto fin_ex;
  case 5: if (data ≠ old_hot) wait(1);
    ⟨ Clean the data caches 361 ⟩;
  case 6: if (data ≠ old_hot) wait(1);
    ⟨ Zap the translation caches 358 ⟩;
  case 7: if (data ≠ old_hot) wait(1);
    ⟨ Zap the instruction and data caches 359 ⟩;
  }
```

357.

```
⟨ Wait if there's an unfinished load ahead of us 357 ⟩ ≡
  {
    register control *cc;
    for (cc = data; cc ≠ hot; ) {
      cc = (cc ≡ reorder_top ? reorder_bot : cc + 1);
      if (cc→owner ∧ (cc→i ≡ ld ∨ cc→i ≡ ldunc ∨ cc→i ≡ pst)) wait(1);
    }
  }
```

This code is used in section 356.

358. Perhaps the delay should be longer here.

```
⟨ Zap the translation caches 358 ⟩ ≡
  if (DTcache→lock ∨ (j = get_reader(DTcache)) < 0) wait(1);
  startup(&DTcache→reader[j], DTcache→access_time);
  set_lock(self, DTcache→lock);
  zap_cache(DTcache);
  data→state = 10; wait(DTcache→access_time);
```

This code is used in section 356.

359. ⟨ Zap the instruction and data caches 359 ⟩ ≡

```
if (¬Icache) {
  data→state = 11; goto switch1;
}
if (Icache→lock ∨ (j = get_reader(Icache)) < 0) wait(1);
startup(&Icache→reader[j], Icache→access_time);
set_lock(self, Icache→lock);
zap_cache(Icache);
data→state = 11; wait(Icache→access_time);
```

This code is used in section 356.

360. ⟨ Special cases for states in the first stage 266 ⟩ +≡

```
case 10: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (ITcache→lock ∨ (j = get_reader(ITcache)) < 0) wait(1);
  startup(&ITcache→reader[j], ITcache→access_time);
  set_lock(self, ITcache→lock);
  zap_cache(ITcache);
  data→state = 3; wait(ITcache→access_time);
case 11: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (wbuf_lock) wait(1);
  write_head = write_tail, write_ctl.state = 0;      /* zap the write buffer */
  if (¬Dcache) {
    data→state = 12; goto switch1;
  }
  if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);
  startup(&Dcache→reader[j], Dcache→access_time);
  set_lock(self, Dcache→lock);
  zap_cache(Dcache);
  data→state = 12; wait(Dcache→access_time);
case 12: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (¬Scache) goto fin_ex;
  if (Scache→lock) wait(1);
  set_lock(self, Scache→lock);
  zap_cache(Scache);
  data→state = 3; wait(Scache→access_time);
```

361. ⟨ Clean the data caches 361 ⟩ ≡

```
if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
⟨ Wait till write buffer is empty 362 ⟩;
if (clean_co.next ∨ clean_lock) wait(1);
set_lock(self, clean_lock);
clean_ctl.i = sync; clean_ctl.state = 0; clean_ctl.x.o.h = 0;
startup(&clean_co, 1);
data→state = 13;
data→interim = true;
wait(1);
```

This code is used in section 356.

362. ⟨ Wait till write buffer is empty 362 ⟩ ≡

```
if (write_head ≠ write_tail) {
  if (¬speed_lock) set_lock(self, speed_lock);
  wait(1);
}
```

This code is used in sections 361 and 364.

363. The cleanup process might take a huge amount of time, so we must allow it to be interrupted. (Servicing the interruption might, of course, put more stuff into the cache.)

⟨ Special cases for states in the first stage 266 ⟩ +≡

```
case 13: if (¬clean_co.next) {
    data→interim = false; goto fin_ex;      /* it's done! */
  }
  if (trying_to_interrupt) goto fin_ex;      /* accept an interruption */
  wait(1);
```

364. Now we consider `SYNCD` and `SYNCID`. When control comes to this part of the program, *data→y.o* is a virtual address and *data→z.o* is the corresponding physical address; *data→xx* + 1 is the number of bytes we are supposed to be syncing; *data→b.o.l* is the number of bytes we can handle at once (either *Icache→bb* or *Dcache→bb* or 8192).

We need a more elaborate scheme to implement `SYNCD` and `SYNCID` than we have used for the "hint" instructions `PRELD`, `PREGO`, and `PREST`, because `SYNCD` and `SYNCID` are not merely hints. They cannot be converted into a sequence of cache-block-size commands at dispatch time, because we cannot be sure that the starting virtual address will be aligned with the beginning of a cache block. We need to realize that the bytes specified by `SYNCD` or `SYNCID` might cross a virtual page boundary—possibly with different protection bits on each page. We need to allow for interrupts. And we also need to keep the fetch buffer empty until a user's `SYNCID` has completely brought the memory up to date.

```
⟨ Special cases for states in later stages 272 ⟩ +≡
do_syncid: data→state = 30;
case 30: if (data ≠ old_hot) wait(1);
  if (¬Icache) {
    data→state = (data→loc.h & sign_bit ? 31 : 33); goto switch2;
  }
  ⟨ Clean the I-cache block for data→z.o, if any 365 ⟩;
  data→state = (data→loc.h & sign_bit ? 31 : 33); wait(Icache→access_time);
case 31: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  ⟨ Wait till write buffer is empty 362 ⟩;
  if (((data→b.o.l − 1) & ∼data→y.o.l) < data→xx) data→interim = true;
  if (¬Dcache) goto next_sync;
  ⟨ Clean the D-cache block for data→z.o, if any 366 ⟩;
  data→state = 32; wait(Dcache→access_time);
case 32: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (¬Scache) goto next_sync;
  ⟨ Clean the S-cache block for data→z.o, if any 367 ⟩;
  data→state = 35; wait(Scache→access_time);
do_syncd: data→state = 33;
case 33: if (data ≠ old_hot) wait(1);
  if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  ⟨ Wait till write buffer is empty 362 ⟩;
  if (((data→b.o.l − 1) & ∼data→y.o.l) < data→xx) data→interim = true;
  if (¬Dcache) {
    if (data→i ≡ syncd) goto fin_ex; else goto next_sync; }
  ⟨ Use cleanup on the cache blocks for data→z.o, if any 368 ⟩;
  data→state = 34;
case 34: if (¬clean_co.next) goto next_sync;
  if (trying_to_interrupt ∧ data→interim ∧ data ≡ old_hot) {
    data→z.o = zero_octa;      /* anticipate RESUME_CONT */
    goto fin_ex;      /* accept an interruption */
  }
  wait(1);
next_sync: data→state = 35;
case 35: if (self→lockloc) *(self→lockloc) = Λ, self→lockloc = Λ;
  if (data→interim) ⟨ Continue this command on the next cache block 369 ⟩;
  data→go.known = true;
  goto fin_ex;
```

365. ⟨ Clean the I-cache block for *data→z.o*, if any 365 ⟩ ≡

```
if (Icache→lock ∨ (j = get_reader(Icache)) < 0) wait(1);
startup(&Icache→reader[j], Icache→access_time);
set_lock(self, Icache→lock);
p = cache_search(Icache, data→z.o);
if (p) {
  demote_and_fix(Icache, p);
  clean_block(Icache, p);
}
```

This code is used in section 364.

366. ⟨ Clean the D-cache block for *data→z.o*, if any 366 ⟩ ≡

```
if (Dcache→lock ∨ (j = get_reader(Dcache)) < 0) wait(1);
startup(&Dcache→reader[j], Dcache→access_time);
set_lock(self, Dcache→lock);
p = cache_search(Dcache, data→z.o);
if (p) {
  demote_and_fix(Dcache, p);
  clean_block(Dcache, p);
}
```

This code is used in section 364.

367. ⟨ Clean the S-cache block for *data→z.o*, if any 367 ⟩ ≡

```
if (Scache→lock) wait(1);
set_lock(self, Scache→lock);
p = cache_search(Scache, data→z.o);
if (p) {
  demote_and_fix(Scache, p);
  clean_block(Scache, p);
}
```

This code is used in section 364.

368. ⟨ Use *cleanup* on the cache blocks for *data→z.o*, if any 368 ⟩ ≡

```
if (clean_co.next ∨ clean_lock) wait(1);
set_lock(self, clean_lock);
clean_ctl.i = syncd;
clean_ctl.state = 4;
clean_ctl.x.o.h = data→loc.h & sign_bit;
clean_ctl.z.o = data→z.o;
schedule(&clean_co, 1, 4);
```

This code is used in section 364.

369. We use the fact that cache block sizes are divisors of 8192.

```
⟨ Continue this command on the next cache block 369 ⟩ ≡
  {
    data→interim = false;
    data→xx -= ((data→b.o.l - 1) & ~data→y.o.l) + 1;
    data→y.o = incr(data→y.o, data→b.o.l);
    data→y.o.l &= -data→b.o.l;
    data→z.o.l = (data→z.o.l & -8192) + (data→y.o.l & 8191);
    if ((data→y.o.l & 8191) ≡ 0) goto square_one;      /* maybe crossed a page boundary */
    if (data→i ≡ syncd) goto do_syncd; else goto do_syncid;
  }
```

This code is used in section 364.

370. If the first page lacks proper protection, we still must try the second, in the rare case that a page boundary is spanned.

```
⟨ Special cases for states in later stages 272 ⟩ +≡
sync_check: if ((data→y.o.l ⊕ (data→y.o.l + data→xx)) ≥ 8192) {
    data→xx -= (8191 & ~data→y.o.l) + 1;
    data→y.o = incr(data→y.o, 8192);
    data→y.o.l &= -8192;
    goto square_one;
  }
  goto fin_ex;
```

371. Input and output. We're done implementing the hardware, but there's still a small matter of software remaining, because we sometimes want to pretend that a real operating system is present without actually having one loaded. This simulator therefore implements a special feature: If `RESUME 1` is issued in location rT, the ten special I/O traps of MMIX-SIM are performed instantaneously behind the scenes.

Of course all claims of accurate simulation go out the door when this feature is used.

```
#define max_sys_call Ftell
⟨ Type definitions 11 ⟩ +≡
  typedef enum {
    Halt, Fopen, Fclose, Fread, Fgets, Fgetws, Fwrite, Fputs, Fputws, Fseek, Ftell
  } sys_call;
```

372. ⟨ Magically do an I/O operation, if *cool→loc* is rT 372 ⟩ ≡

```
if (cool→loc.l ≡ g[rT].o.l ∧ cool→loc.h ≡ g[rT].o.h) {
  register unsigned char yy, zz;
  octa ma, mb;
  if (g[rXX].o.l & #ffff0000) goto magic_done;
  yy = g[rXX].o.l ≫ 8, zz = g[rXX].o.l & #ff;
  if (yy > max_sys_call) goto magic_done;
  ⟨ Prepare memory arguments ma = M[a] and mb = M[b] if needed 380 ⟩;
  switch (yy) {
  case Halt: ⟨ Either halt or print warning 373 ⟩; break;
  case Fopen: g[rBB].o = mmix_fopen(zz, mb, ma); break;
  case Fclose: g[rBB].o = mmix_fclose(zz); break;
  case Fread: g[rBB].o = mmix_fread(zz, mb, ma); break;
  case Fgets: g[rBB].o = mmix_fgets(zz, mb, ma); break;
  case Fgetws: g[rBB].o = mmix_fgetws(zz, mb, ma); break;
  case Fwrite: g[rBB].o = mmix_fwrite(zz, mb, ma); break;
  case Fputs: g[rBB].o = mmix_fputs(zz, g[rBB].o); break;
  case Fputws: g[rBB].o = mmix_fputws(zz, g[rBB].o); break;
  case Fseek: g[rBB].o = mmix_fseek(zz, g[rBB].o); break;
  case Ftell: g[rBB].o = mmix_ftell(zz); break;
  }
magic_done: g[255].o = neg_one; /* this will enable interrupts */
}
```

This code is used in section 322.

373. ⟨ Either halt or print warning 373 ⟩ ≡

```
if (¬zz) halted = true;
else if (zz ≡ 1) {
  octa trap_loc;
  trap_loc = incr(g[rWW].o, −4);
  if (¬(trap_loc.h ∨ trap_loc.l ≥ #f0)) print_trip_warning(trap_loc.l ≫ 4, incr(g[rW].o, −4));
}
```

This code is used in section 372.

374. ⟨ Global variables 20 ⟩ +≡

```
char arg_count[] = {1, 3, 1, 3, 3, 3, 3, 2, 2, 2, 1};
```

375. The input/output operations invoked by `TRAP`s are done by subroutines in an auxiliary program module called MMIX-IO. Here we need only declare those subroutines, and write three primitive interfaces on which they depend.

376. ⟨ Global variables 20 ⟩ +≡

```
extern octa mmix_fopen ARGS((unsigned char, octa, octa));
extern octa mmix_fclose ARGS((unsigned char));
extern octa mmix_fread ARGS((unsigned char, octa, octa));
extern octa mmix_fgets ARGS((unsigned char, octa, octa));
extern octa mmix_fgetws ARGS((unsigned char, octa, octa));
extern octa mmix_fwrite ARGS((unsigned char, octa, octa));
extern octa mmix_fputs ARGS((unsigned char, octa));
extern octa mmix_fputws ARGS((unsigned char, octa));
extern octa mmix_fseek ARGS((unsigned char, octa));
extern octa mmix_ftell ARGS((unsigned char));
extern void print_trip_warning ARGS((int, octa));
```

377. ⟨ Internal prototypes 13 ⟩ +≡

```
int mmgetchars ARGS((char *, int, octa, int));
void mmputchars ARGS((unsigned char *, int, octa));
char stdin_chr ARGS((void));
octa magic_read ARGS((octa));
void magic_write ARGS((octa, octa));
```

378. We need to cut through all the complications of buffers and caches in order to do magical I/O. The *magic_read* routine finds the current octabyte in a given physical address by looking at the write buffer, D-cache, S-cache, and memory until finding it.

⟨ Subroutines 14 ⟩ +≡

```
octa magic_read(addr)
    octa addr;
{
  register write_node *q;
  register cacheblock *p;
  for (q = write_tail; ; ) {
    if (q ≡ write_head) break;
    if (q ≡ wbuf_top) q = wbuf_bot; else q++;
    if ((q→addr.l & -8) ≡ (addr.l & -8) ∧ q→addr.h ≡ addr.h) return q→o;
  }
  if (Dcache) {
    p = cache_search(Dcache, addr);
    if (p) return p→data[(addr.l & (Dcache→bb - 1)) ≫ 3];
    if (((Dcache→outbuf.tag.l ⊕ addr.l) & -Dcache→bb) ≡ 0 ∧ Dcache→outbuf.tag.h ≡ addr.h)
      return Dcache→outbuf.data[(addr.l & (Dcache→bb - 1)) ≫ 3];
    if (Scache) {
      p = cache_search(Scache, addr);
      if (p) return p→data[(addr.l & (Scache→bb - 1)) ≫ 3];
      if (((Scache→outbuf.tag.l ⊕ addr.l) & -Scache→bb) ≡ 0 ∧ Scache→outbuf.tag.h ≡ addr.h)
        return Scache→outbuf.data[(addr.l & (Scache→bb - 1)) ≫ 3];
    }
  }
  return mem_read(addr);
}
```

379. The *magic_write* routine changes the octabyte in a given physical address by changing it wherever it appears in a buffer or cache. Any "dirty" or "least recently used" status remains unchanged. (Yes, this *is* magic.)

```
⟨ Subroutines 14 ⟩ +≡
  void magic_write(addr, val)
      octa addr, val;
  {
    register write_node *q;
    register cacheblock *p;
    for (q = write_tail; ; ) {
      if (q ≡ write_head) break;
      if (q ≡ wbuf_top) q = wbuf_bot; else q++;
      if ((q→addr.l & −8) ≡ (addr.l & −8) ∧ q→addr.h ≡ addr.h) q→o = val;
    }
    if (Dcache) {
      p = cache_search(Dcache, addr);
      if (p) p→data[(addr.l & (Dcache→bb − 1)) ≫ 3] = val;
      if (((Dcache→inbuf.tag.l ⊕ addr.l) & −Dcache→bb) ≡ 0 ∧ Dcache→inbuf.tag.h ≡ addr.h)
        Dcache→inbuf.data[(addr.l & (Dcache→bb − 1)) ≫ 3] = val;
      if (((Dcache→outbuf.tag.l ⊕ addr.l) & −Dcache→bb) ≡ 0 ∧ Dcache→outbuf.tag.h ≡ addr.h)
        Dcache→outbuf.data[(addr.l & (Dcache→bb − 1)) ≫ 3] = val;
      if (Scache) {
        p = cache_search(Scache, addr);
        if (p) p→data[(addr.l & (Scache→bb − 1)) ≫ 3] = val;
        if (((Scache→inbuf.tag.l ⊕ addr.l) & −Scache→bb) ≡ 0 ∧ Scache→inbuf.tag.h ≡ addr.h)
          Scache→inbuf.data[(addr.l & (Scache→bb − 1)) ≫ 3] = val;
        if (((Scache→outbuf.tag.l ⊕ addr.l) & −Scache→bb) ≡ 0 ∧ Scache→outbuf.tag.h ≡ addr.h)
          Scache→outbuf.data[(addr.l & (Scache→bb − 1)) ≫ 3] = val;
      }
    }
    mem_write(addr, val);
  }
```

380. The conventions of our imaginary operating system require us to apply the trivial memory mapping in which segment i appears in a 2^{32}-byte page of physical addresses starting at $2^{32}i$.

```
⟨ Prepare memory arguments ma = M[a] and mb = M[b] if needed 380 ⟩ ≡
  if (arg_count[yy] ≡ 3) {
    octa arg_loc;
    arg_loc = g[rBB].o;
    if (arg_loc.h & #9fffffff) mb = zero_octa;
    else arg_loc.h ≫= 29, mb = magic_read(arg_loc);
    arg_loc = incr(g[rBB].o, 8);
    if (arg_loc.h & #9fffffff) ma = zero_octa;
    else arg_loc.h ≫= 29, ma = magic_read(arg_loc);
  }
```

This code is used in section 372.

381. The subroutine *mmgetchars*(*buf*, *size*, *addr*, *stop*) reads characters starting at address *addr* in the simulated memory and stores them in *buf*, continuing until *size* characters have been read or some other stopping criterion has been met. If *stop* < 0 there is no other criterion; if *stop* $= 0$ a null character will also terminate the process; otherwise *addr* is even, and two consecutive null bytes starting at an even address will terminate the process. The number of bytes read and stored, exclusive of terminating nulls, is returned.

```
⟨ Subroutines 14 ⟩ +≡
  int mmgetchars(buf, size, addr, stop)
      char *buf;
      int size;
      octa addr;
      int stop;
  {
    register char *p;
    register int m;
    octa a, x;
    if (((addr.h & #9fffffff) ∨ (incr(addr, size − 1).h & #9fffffff)) ∧ size) {
      fprintf(stderr, "Attempt␣to␣get␣characters␣from␣off␣the␣page!\n");
      return 0;
    }
    for (p = buf, m = 0, a = addr, a.h >>= 29; m < size; ) {
      x = magic_read(a);
      if ((a.l & #7) ∨ m > size − 8) ⟨ Read and store one byte; return if done 382 ⟩
      else ⟨ Read and store up to eight bytes; return if done 383 ⟩
    }
    return size;
  }
```

382.

```
⟨ Read and store one byte; return if done 382 ⟩ ≡
  {
    if (a.l & #4) *p = (x.l >> (8 * ((~a.l) & #3))) & #ff;
    else *p = (x.h >> (8 * ((~a.l) & #3))) & #ff;
    if (¬*p ∧ stop ≥ 0) {
      if (stop ≡ 0) return m;
      if ((a.l & #1) ∧ *(p − 1) ≡ '\0') return m − 1;
    }
    p++, m++, a = incr(a, 1);
  }
```

This code is used in section 381.

383. ⟨ Read and store up to eight bytes; **return** if done 383 ⟩ ≡

```
{
  *p = x.h >> 24;
  if (¬*p ∧ (stop ≡ 0 ∨ (stop > 0 ∧ x.h < #10000))) return m;
  *(p + 1) = (x.h >> 16) & #ff;
  if (¬*(p + 1) ∧ stop ≡ 0) return m + 1;
  *(p + 2) = (x.h >> 8) & #ff;
  if (¬*(p + 2) ∧ (stop ≡ 0 ∨ (stop > 0 ∧ (x.h & #ffff) ≡ 0))) return m + 2;
  *(p + 3) = x.h & #ff;
  if (¬*(p + 3) ∧ stop ≡ 0) return m + 3;
  *(p + 4) = x.l >> 24;
  if (¬*(p + 4) ∧ (stop ≡ 0 ∨ (stop > 0 ∧ x.l < #10000))) return m + 4;
  *(p + 5) = (x.l >> 16) & #ff;
  if (¬*(p + 5) ∧ stop ≡ 0) return m + 5;
  *(p + 6) = (x.l >> 8) & #ff;
  if (¬*(p + 6) ∧ (stop ≡ 0 ∨ (stop > 0 ∧ (x.l & #ffff) ≡ 0))) return m + 6;
  *(p + 7) = x.l & #ff;
  if (¬*(p + 7) ∧ stop ≡ 0) return m + 7;
  p += 8, m += 8, a = incr(a, 8);
}
```

This code is used in section 381.

384. The subroutine *mmputchars*(*buf*, *size*, *addr*) puts *size* characters into the simulated memory starting at address *addr*.

```
⟨ Subroutines 14 ⟩ +≡
  void mmputchars(buf, size, addr)
      unsigned char *buf;
      int size;
      octa addr;
  {
    register unsigned char *p;
    register int m;
    octa a, x;
    if (((addr.h & #9fffffff) ∨ (incr(addr, size − 1).h & #9fffffff)) ∧ size) {
      fprintf(stderr, "Attempt␣to␣put␣characters␣off␣the␣page!\n");
      return;
    }
    for (p = buf, m = 0, a = addr, a.h >>= 29; m < size; ) {
      if ((a.l & #7) ∨ m > size − 8) ⟨ Load and write one byte 385 ⟩
      else ⟨ Load and write eight bytes 386 ⟩;
    }
  }
```

385. ⟨ Load and write one byte 385 ⟩ ≡

```
{
  register int s = 8 * ((~a.l) & #3);
  x = magic_read(a);
  if (a.l & #4) x.l ⊕= (((x.l ≫ s) ⊕ *p) & #ff) ≪ s;
  else x.h ⊕= (((x.h ≫ s) ⊕ *p) & #ff) ≪ s;
  magic_write(a, x);
  p++, m++, a = incr(a, 1);
}
```

This code is used in section 384.

386. ⟨ Load and write eight bytes 386 ⟩ ≡

```
{
  x.h = (*p ≪ 24) + (*(p + 1) ≪ 16) + (*(p + 2) ≪ 8) + *(p + 3);
  x.l = (*(p + 4) ≪ 24) + (*(p + 5) ≪ 16) + (*(p + 6) ≪ 8) + *(p + 7);
  magic_write(a, x);
  p += 8, m += 8, a = incr(a, 8);
}
```

This code is used in section 384.

387. When standard input is being read by the simulated program at the same time as it is being used for interaction, we try to keep the two uses separate by maintaining a private buffer for the simulated program's `StdIn`. Online input is usually transmitted from the keyboard to a C program a line at a time; therefore an *fgets* operation works much better than *fread* when we prompt for new input. But there is a slight complication, because *fgets* might read a null character before coming to a newline character. We cannot deduce the number of characters read by *fgets* simply by looking at *strlen*(*stdin_buf*).

⟨ Subroutines 14 ⟩ +≡

```
char stdin_chr()
{
  register char *p;
  while (stdin_buf_start ≡ stdin_buf_end) {
    printf("StdIn>␣"); fflush(stdout);
    fgets(stdin_buf, 256, stdin);
    stdin_buf_start = stdin_buf;
    for (p = stdin_buf; p < stdin_buf + 254; p++)
      if (*p ≡ '\n') break;
    stdin_buf_end = p + 1;
  }
  return *stdin_buf_start++;
}
```

388. ⟨ Global variables 20 ⟩ +≡

```
char stdin_buf[256];      /* standard input to the simulated program */
char *stdin_buf_start;    /* current position in that buffer */
char *stdin_buf_end;      /* current end of that buffer */
```

389. Index.

⟨ Zap the instruction and data caches 359 ⟩ Used in section 356.
⟨ Zap the translation caches 358 ⟩ Used in section 356.
⟨ mmix-pipe.h 5 ⟩

May 1, 2025 at 02:27

MMIX-PIPE