1. Input format. Configuration files allow this simulator to adapt itself to infinitely many possible combinations of hardware features. The purpose of the present module is to read a configuration file, check it for validity, and set up the relevant data structures.

All data in a configuration file consists simply of *tokens* separated by one or more units of white space, where a "token" is any sequence of nonspace characters that doesn't contain a percent sign. Percent signs and anything following them on a line are ignored; this convention allows a user to include comments in the file. Here's a simple (but weird) example:

```
% Silly configuration
writebuffer 200
memaddresstime 100
Dcache associativity 4 lru
Dcache blocksize 1024
unit ODD 5555555555555555555555555555555555555555555555555555555555555555
unit EVEN aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
div 40 30 20  % three-stage divide
```

It means that (1) the write buffer has capacity for 200 octabytes; (2) the memory bus takes 100 cycles to process an address; (3) there's a D-cache, in which each set has 4 blocks and the replacement policy is least-recently-used; (4) each block in the D-cache has 1024 bytes; (5) there are two functional units, one for all the odd-numbered opcodes and one for all the rest; (6) the division instructions take three pipeline stages, spending 40 cycles in the first stage, 30 in the second, and 20 in the last; (7) all other parameters have default values.

2. Four kinds of specifications can appear in a configuration file, according to the following syntax:

⟨ specification ⟩ ⟶ ⟨ PV spec ⟩ | ⟨ cache spec ⟩ | ⟨ pipe spec ⟩ | ⟨ functional spec ⟩
⟨ PV spec ⟩ ⟶ ⟨ parameter ⟩⟨ decimal value ⟩
⟨ cache spec ⟩ ⟶ ⟨ cache name ⟩⟨ cache parameter ⟩⟨ decimal value ⟩⟨ policy ⟩
⟨ pipe spec ⟩ ⟶ ⟨ operation ⟩⟨ pipeline times ⟩
⟨ functional spec ⟩ ⟶ `unit` ⟨ name ⟩⟨ 64 hexadecimal digits ⟩

3. A ⟨ PV spec ⟩ simply assigns a given value to a given parameter. The possibilities for ⟨ parameter ⟩ are as follows:

- `fetchbuffer` (default 4), maximum instructions in the fetch buffer; must be ≥ 1.
- `writebuffer` (default 2), maximum octabytes in the write buffer; must be ≥ 1.
- `reorderbuffer` (default 5), maximum instructions issued but not committed; must be ≥ 1.
- `renameregs` (default 5), maximum partial results in the reorder buffer; must be ≥ 1.
- `memslots` (default 2), maximum store instructions in the reorder buffer; must be ≥ 1.
- `localregs` (default 256), number of local registers in ring; must be 256, 512, or 1024.
- `fetchmax` (default 2), maximum instructions fetched per cycle; must be ≥ 1.
- `dispatchmax` (default 1), maximum instructions issued per cycle; must be ≥ 1.
- `peekahead` (default 1), maximum lookahead for jumps per cycle.
- `commitmax` (default 1), maximum instructions committed per cycle; must be ≥ 1.
- `fremmax` (default 1), maximum reductions in `FREM` computation per cycle; must be ≥ 1.
- `denin` (default 1), extra cycles taken if a floating point input is subnormal.
- `denout` (default 1), extra cycles taken if a floating point result is subnormal.
- `writeholdingtime` (default 0), minimum number of cycles for data to remain in the write buffer.
- `memaddresstime` (default 20), cycles to process memory address; must be ≥ 1.
- `memreadtime` (default 20), cycles to read one memory busload; must be ≥ 1.
- `memwritetime` (default 20), cycles to write one memory busload; must be ≥ 1.
- `membusbytes` (default 8), number of bytes per memory busload; must be a power of 2 that is 8 or more.
- `branchpredictbits` (default 0), number of bits in each branch prediction table entry; must be ≤ 8.
- `branchaddressbits` (default 0), number of bits in instruction address used to index the branch prediction table.
- `branchhistorybits` (default 0), number of bits in branch history used to index the branch prediction table.
- `branchdualbits` (default 0), number of bits of instruction-address-xor-branch-history used to index the branch prediction table.
- `hardwarepagetable` (default 1), is zero if page table calculations must be emulated by the operating system.
- `disablesecurity` (default 0), is 1 if the hot-seat security checks are turned off. This option is used only for testing purposes; it means that the '`s`' interrupt will not occur, and the '`p`' interrupt will be signaled only when going from a nonnegative location to a negative one.
- `memchunksmax` (default 1000), maximum number of 2^{16}-byte chunks of simulated memory; must be ≥ 1.
- `hashprime` (default 2003), prime number used to address simulated memory; must exceed `memchunksmax`, preferably by a factor of about 2.

The values of `memchunksmax` and `hashprime` affect only the speed of the simulator, not its results—unless a very huge program is being simulated. The stated defaults for `memchunksmax` and `hashprime` should be adequate for almost all applications.

4. A ⟨ cache spec ⟩ assigns a given value to a parameter affecting one of five possible caches:

⟨ cache spec ⟩ ⟶ ⟨ cache name ⟩⟨ cache parameter ⟩⟨ decimal value ⟩⟨ policy ⟩
⟨ cache name ⟩ ⟶ `ITcache` | `DTcache` | `Icache` | `Dcache` | `Scache`
⟨ policy ⟩ ⟶ ⟨ empty ⟩ | `random` | `serial` | `pseudolru` | `lru`

The possibilities for ⟨ cache parameter ⟩ are as follows:

- `associativity` (default 1), number of cache blocks per cache set; must be a power of 2. (A cache with associativity 1 is said to be "direct-mapped.")
- `blocksize` (default 8), number of bytes per cache block; must be a power of 2, at least equal to the granularity, and at most equal to 8192. The blocksize of `ITcache` and `DTcache` must be 8.
- `setsize` (default 1), number of sets of cache blocks; must be a power of 2. (A cache with set size 1 is said to be "fully associative.")
- `granularity` (default 8), number of bytes per "dirty bit," used to remember which items of data have changed since they were read from memory; must be a power of 2 and at least 8. The granularity must be 8 if `writeallocate` is 0.
- `victimsize` (default 0), number of cache blocks in the victim buffer, which holds blocks removed from the main cache sets; must be zero or a power of 2.
- `writeback` (default 0), is 1 in a "write-back" cache, which holds dirty data as long as possible; is 0 in a "write-through" cache, which cleans all data as soon as possible.
- `writeallocate` (default 0), is 1 in a "write-allocate" cache, which remembers all recently written data; is 0 in a "write-around" cache, which doesn't make space for newly written data that fails to hit an existing cache block.
- `accesstime` (default 1), number of cycles to query the cache; must be ≥ 1. (Hits in the S-cache actually require *twice* the accesstime, once to query the tag and once to transmit the data.)
- `copyintime` (default 1), number of cycles to move a cache block from its input buffer into the cache proper; must be ≥ 1.
- `copyouttime` (default 1), number of cycles to move a cache block from the cache proper to its output buffer; must be ≥ 1.
- `ports` (default 1), number of processes that can simultaneous query the cache; must be ≥ 1.

The ⟨ policy ⟩ parameter should be nonempty only on cache specifications for parameters `associativity` and `victimsize`. If no replacement policy is specified, `random` is the default. All four policies are equivalent when the `associativity` or `victimsize` is 1; `pseudolru` is equivalent to `lru` when the `associativity` or `victimsize` is 2.

The `granularity`, `writeback`, `writeallocate`, and `copyouttime` parameters affect the performance only of the D-cache and S-cache; the other three caches are read-only, so they never need to write their data.

The `ports` parameter affects the performance of the D-cache and DT-cache, and (if the `PREGO` command is used) the performance of the I-cache and IT-cache. The S-cache accommodates only one process at a time, regardless of the number of specified ports.

Only the translation caches (the IT-cache and DT-cache) are present by default. But if any specifications are given for, say, an I-cache, all of the unspecified I-cache parameters take their default values.

The existence of an S-cache (secondary cache) implies the existence of both I-cache and D-cache (primary caches for instructions and data). The block size of the secondary cache must not be less than the block size of the primary caches. The secondary cache must have the same granularity as the D-cache.

5. A ⟨ pipe spec ⟩ governs the execution time of potentially slow operations.

⟨ pipe spec ⟩ ⟶ ⟨ operation ⟩⟨ pipeline times ⟩
⟨ pipeline times ⟩ ⟶ ⟨ decimal value ⟩ | ⟨ pipeline times ⟩⟨ decimal value ⟩

Here the ⟨ operation ⟩ is one of the following:

- `mul0` through `mul8` (default 10); the values for `mul`j refer to products in which the second operand is less than 2^{8j}, where j is as small as possible. Thus, for example, `mul1` applies to nonzero one-byte multipliers.
- `div` (default 60); this applies to integer division, signed and unsigned.
- `sh` (default 1); this applies to left and right shifts, signed and unsigned.
- `mux` (default 1); the multiplex operator.
- `sadd` (default 1); the sideways addition operator.
- `mor` (default 1); the boolean matrix multiplication operators `MOR` and `MXOR`.
- `fadd` (default 4); floating point addition and subtraction.
- `fmul` (default 4); floating point multiplication.
- `fdiv` (default 40); floating point division.
- `fsqrt` (default 40); floating point square root.
- `fint` (default 4); floating point integerization.
- `fix` (default 2); conversion from floating to fixed, signed and unsigned.
- `flot` (default 2); conversion from fixed to floating, signed and unsigned.
- `feps` (default 4); floating comparison with respect to epsilon.

In each case one can specify a sequence of pipeline stages, with a positive number of cycles to be spent in each stage. For example, a specification like '`fmul 3 1`' would say that a functional unit that supports `FMUL` takes a total of four cycles to compute the floating point product in two stages; it can start working on a second product after three cycles have gone by.

If a floating point operation has a subnormal input, `denin` is added to the time for the first stage. If a floating point operation has a subnormal result, `denout` is added to the time for the last stage.

6. The fourth and final kind of specification defines a functional unit:

⟨ functional spec ⟩ ⟶ `unit` ⟨ name ⟩⟨ 64 hexadecimal digits ⟩

The symbolic name should be at most fifteen characters long. The 64 hexadecimal digits contain 256 bits, with '1' for each supported opcode; the most significant (leftmost) bit is for opcode 0 (`TRAP`), and the least significant bit is for opcode 255 (`TRIP`).

For example, we can define a load/store unit (which handles register/memory operations), a multiplication unit (which handles fixed and floating point multiplication), a boolean unit (which handles only bitwise operations), and a more general arithmetic-logical unit, as follows:

```
unit LSU 00000000000000000000000000000000ffffffcfffffffc0000000000000000
unit MUL 000080f0000000000000000000000000000000000000000000000000000000
unit BIT 00000000000000000000000000000000000000000000000ffff00ff00ff0000
unit ALU f0000000fffffffffffffffffffffff000000030000000 3fffffffffffffff
```

The order in which units are specified is important, because `MMIX`'s dispatcher will try to match each instruction with the first functional unit that supports its opcode. Therefore it is best to list more specialized units (like the `BIT` unit in this example) before more general ones; this lets the specialized units have first chance at the instructions they can handle.

There can be any number of functional units, having possibly identical specifications. One should, however, give each unit a unique name (e.g., `ALU1` and `ALU2` if there are two arithmetic-logical units), since these names are used in diagnostic messages.

Opcodes that aren't supported by any specified unit will cause an emulation trap.

7. Full details about the significance of all these parameters can be found in the `mmix-pipe` module, which defines and discusses the data structures that need to be configured and initialized.

Of course the specifications in a configuration file needn't make any sense, nor need they be practically achievable. We could, for example, specify a unit that handles only the two opcodes `NXOR` and `DIVUI`; we could specify 1-cycle division but pipelined 100-cycle shifts, or 1-cycle memory access but 100-cycle cache access. We could create a thousand rename registers and issue a hundred instructions per cycle, etc. Some combinations of parameters are clearly ridiculous.

But there remain a huge number of possibilities of interest, especially as technology continues to evolve. By experimenting with configurations that are extreme by present-day standards, we can see how much might be gained if the corresponding hardware could be built economically.

8. Basic input/output. Let's get ready to program the *MMIX_config* subroutine by building some simple infrastructure. First we need some macros to print error messages.

```
#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 errprint3(f,a,b,c) fprintf(stderr,f,a,b,c)
#define panic(x) { x; errprint0("!\n"); exit(-1); }
```

9. And we need a place to look at the input.

```
#define BUF_SIZE 100 /* we don't need long lines */
```

⟨ Global variables 9 ⟩ ≡

```
FILE *config_file; /* input comes from here */
char token[BUF_SIZE]; /* and tokens are copied to here */
bool token_prescanned; /* does token contain the next token already? */
```

See also sections 15 and 28.

This code is used in section 38.

10. The *get_token* routine copies the next token of input into the *token* buffer. After the input has ended, a final '`end`' is appended.

⟨ Subroutines 10 ⟩ ≡

```
static void get_token ARGS((void));
static void get_token() /* set token to the next token of the configuration file */
{
  static char buffer[BUF_SIZE]; /* input lines go here */
  static char *buf_pointer = buffer; /* this is our current position */
  register char *p, *q;
  if (token_prescanned) {
    token_prescanned = false; return;
  }
  while (1) { /* scan past white space */
    if (*buf_pointer ≡ '\0' ∨ *buf_pointer ≡ '\n' ∨ *buf_pointer ≡ '%') {
      if (¬fgets(buffer, BUF_SIZE, config_file)) {
        strcpy(token, "end"); return;
      }
      if (strlen(buffer) ≡ BUF_SIZE − 1 ∧ buffer[BUF_SIZE − 2] ≠ '\n')
        panic(errprint1("config␣file␣line␣too␣long:␣‘%s...’", buffer));
      buf_pointer = buffer;
    } else if (¬isspace(*buf_pointer)) break;
    else buf_pointer++;
  }
  for (p = buf_pointer, q = token; ¬isspace(*p) ∧ *p ≠ '%'; p++, q++) *q = *p;
  buf_pointer = p; *q = '\0';
  return;
}
```

See also sections 11, 16, 22, 23, 30, and 31.

This code is used in section 38.

11. The *get_int* routine is called when we wish to input a decimal value. It returns -1 if the next token isn't a string of decimal digits.

```
⟨ Subroutines 10 ⟩ +≡
  static int get_int ARGS((void));
  static int get_int( )
  { int v;
    char *p;
    get_token( );
    for (p = token, v = 0; *p ≥ '0' ∧ *p ≤ '9'; p++) v = 10 * v + *p - '0';
    if (*p) return -1;
    return v;
  }
```

12. A simple data structure makes it fairly easy to deal with parameter/value specifications.

```
⟨ Type definitions 12 ⟩ ≡
  typedef struct {
    char name[20];      /* symbolic name */
    int *v;      /* internal name */
    int defval;      /* default value */
    int minval, maxval;      /* minimum and maximum legal values */
    bool power_of_two;      /* must it be a power of two? */
  } pv_spec;
```

See also sections 13 and 14.

This code is used in section 38.

13. Cache parameters are a bit more difficult, but still not bad.

```
⟨ Type definitions 12 ⟩ +≡
  typedef enum {
    assoc, blksz, setsz, gran, vctsz, wrb, wra, acctm, citm, cotm, prts
  } c_param;
  typedef struct {
    char name[20];      /* symbolic name */
    c_param v;      /* internal code */
    int defval;      /* default value */
    int minval, maxval;      /* minimum and maximum legal values */
    bool power_of_two;      /* must it be a power of two? */
  } cpv_spec;
```

14. Operation codes are the easiest of all.

```
⟨ Type definitions 12 ⟩ +≡
  typedef struct {
    char name[8];      /* symbolic name */
    internal_opcode v;      /* internal code */
    int defval;      /* default value */
  } op_spec;
```

15. Most of the parameters are external variables declared in the header file `mmix-pipe.h`; but some are private to this module. Here we define the main tables used below.

⟨ Global variables 9 ⟩ +≡

```
int fetch_buf_size, write_buf_size, reorder_buf_size, mem_bus_bytes, hardware_PT;
int max_cycs = 60;
pv_spec PV[] = {
    {"fetchbuffer", &fetch_buf_size, 4, 1, INT_MAX, false},
    {"writebuffer", &write_buf_size, 2, 1, INT_MAX, false},
    {"reorderbuffer", &reorder_buf_size, 5, 1, INT_MAX, false},
    {"renameregs", &max_rename_regs, 5, 1, INT_MAX, false},
    {"memslots", &max_mem_slots, 2, 1, INT_MAX, false},
    {"localregs", &lring_size, 256, 256, 1024, true},
    {"fetchmax", &fetch_max, 2, 1, INT_MAX, false},
    {"dispatchmax", &dispatch_max, 1, 1, INT_MAX, false},
    {"peekahead", &peekahead, 1, 0, INT_MAX, false},
    {"commitmax", &commit_max, 1, 1, INT_MAX, false},
    {"fremmax", &frem_max, 1, 1, INT_MAX, false},
    {"denin", &denin_penalty, 1, 0, INT_MAX, false},
    {"denout", &denout_penalty, 1, 0, INT_MAX, false},
    {"writeholdingtime", &holding_time, 0, 0, INT_MAX, false},
    {"memaddresstime", &mem_addr_time, 20, 1, INT_MAX, false},
    {"memreadtime", &mem_read_time, 20, 1, INT_MAX, false},
    {"memwritetime", &mem_write_time, 20, 1, INT_MAX, false},
    {"membusbytes", &mem_bus_bytes, 8, 8, INT_MAX, true},
    {"branchpredictbits", &bp_n, 0, 0, 8, false},
    {"branchaddressbits", &bp_a, 0, 0, 32, false},
    {"branchhistorybits", &bp_b, 0, 0, 32, false},
    {"branchdualbits", &bp_c, 0, 0, 32, false},
    {"hardwarepagetable", &hardware_PT, 1, 0, 1, false},
    {"disablesecurity", (int *) &security_disabled, 0, 0, 1, false},
    {"memchunksmax", &mem_chunks_max, 1000, 1, INT_MAX, false},
    {"hashprime", &hash_prime, 2003, 2, INT_MAX, false}};
cpv_spec CPV[] = {
    {"associativity", assoc, 1, 1, INT_MAX, true},
    {"blocksize", blksz, 8, 8, 8192, true},
    {"setsize", setsz, 1, 1, INT_MAX, true},
    {"granularity", gran, 8, 8, 8192, true},
    {"victimsize", vctsz, 0, 0, INT_MAX, true},
    {"writeback", wrb, 0, 0, 1, false},
    {"writeallocate", wra, 0, 0, 1, false},
    {"accesstime", acctm, 1, 1, INT_MAX, false},
    {"copyintime", citm, 1, 1, INT_MAX, false},
    {"copyouttime", cotm, 1, 1, INT_MAX, false},
    {"ports", prts, 1, 1, INT_MAX, false}};
op_spec OP[] = {
    {"mul0", mul0, 10}, {"mul1", mul1, 10}, {"mul2", mul2, 10}, {"mul3", mul3, 10}, {"mul4", mul4, 10},
    {"mul5", mul5, 10}, {"mul6", mul6, 10}, {"mul7", mul7, 10}, {"mul8", mul8, 10},
    {"div", div, 60}, {"sh", sh, 1}, {"mux", mux, 1}, {"sadd", sadd, 1}, {"mor", mor, 1},
    {"fadd", fadd, 4}, {"fmul", fmul, 4}, {"fdiv", fdiv, 40}, {"fsqrt", fsqrt, 40}, {"fint", fint, 4},
    {"fix", fix, 2}, {"flot", flot, 2}, {"feps", feps, 4}};
int PV_size, CPV_size, OP_size;      /* the number of entries in PV, CPV, OP */
```

16. The *new_cache* routine creates a **cache** structure with default values. (These default values are "hard-wired" into the program, not actually read from the *CPV* table.)

```
⟨ Subroutines 10 ⟩ +≡
  static cache *new_cache ARGS((char *));
  static cache *new_cache(name)
      char *name;
  { register cache *c = (cache *) calloc(1, sizeof(cache));
    if (¬c) panic(errprint1("Can't␣allocate␣%s", name));
    c→aa = 1;      /* default associativity, should equal CPV[0].defval */
    c→bb = 8;      /* default blocksize */
    c→cc = 1;      /* default setsize */
    c→gg = 8;      /* default granularity */
    c→vv = 0;      /* default victimsize */
    c→repl = random;      /* default replacement policy */
    c→vrepl = random;      /* default victim replacement policy */
    c→mode = 0;      /* default mode is write-through and write-around */
    c→access_time = c→copy_in_time = c→copy_out_time = 1;
    c→filler.ctl = &(c→filler_ctl);
    c→filler_ctl.ptr_a = (void *) c;
    c→filler_ctl.go.o.l = 4;
    c→flusher.ctl = &(c→flusher_ctl);
    c→flusher_ctl.ptr_a = (void *) c;
    c→flusher_ctl.go.o.l = 4;
    c→ports = 1;
    c→name = name;
    return c;
  }
```

17. ⟨ Initialize to defaults 17 ⟩ ≡

```
  PV_size = (sizeof PV)/sizeof(pv_spec);
  CPV_size = (sizeof CPV)/sizeof(cpv_spec);
  OP_size = (sizeof OP)/sizeof(op_spec);
  ITcache = new_cache("ITcache");
  DTcache = new_cache("DTcache");
  Icache = Dcache = Scache = Λ;
  for (j = 0; j < PV_size; j++) *(PV[j].v) = PV[j].defval;
  for (j = 0; j < OP_size; j++) {
    pipe_seq[OP[j].v][0] = OP[j].defval;
    pipe_seq[OP[j].v][1] = 0;      /* one stage */
  }
```

This code is used in section 38.

18. Reading the specs. Before we're ready to process the configuration file, we need to count the number of functional units, so that we know how much space to allocate for them.

A special background unit is always provided, just to make sure that `TRAP` and `TRIP` instructions are handled by somebody.

```
⟨ Count and allocate the functional units 18 ⟩ ≡
  funit_count = 0;
  while (strcmp(token, "end") ≠ 0) {
    get_token( );
    if (strcmp(token, "unit") ≡ 0) {
      funit_count++;
      get_token( ); get_token( );      /* a unit might be named unit or end */
    }
  }
  funit = (func *) calloc(funit_count + 1, sizeof(func));
  if (¬funit) panic(errprint0("Can't allocate the functional units"));
  strcpy(funit[funit_count].name, "%%");
  funit[funit_count].ops[0] = #80000000;      /* TRAP */
  funit[funit_count].ops[7] = #1;      /* TRIP */
```

This code is used in section 38.

19. Now we can read the specifications and obey them. This program doesn't bother to be very tolerant of errors, nor does it try to be very efficient.

Incidentally, the specifications don't have to be broken into individual lines in any meaningful way. We simply read them token by token.

```
⟨ Record all the specs 19 ⟩ ≡
  rewind(config_file);
  funit_count = 0;
  token[0] = '\0';
  while (strcmp(token, "end") ≠ 0) {
    get_token( );
    if (strcmp(token, "end") ≡ 0) break;
    ⟨ If token is a parameter name, process a PV spec 20 ⟩;
    ⟨ If token is a cache name, process a cache spec 21 ⟩;
    ⟨ If token is an operation name, process a pipe spec 24 ⟩;
    if (strcmp(token, "unit") ≡ 0) ⟨ Process a functional spec 25 ⟩;
    panic(errprint1("Configuration syntax error: Specification can't start with `%s'", token));
  }
```

This code is used in section 38.

20. ⟨ If *token* is a parameter name, process a PV spec 20 ⟩ ≡

```
for (j = 0; j < PV_size; j++)
  if (strcmp(token, PV[j].name) ≡ 0) {
    n = get_int( );
    if (n < PV[j].minval)
      panic(errprint2("Configuration␣error:␣%s␣must␣be␣>=␣%d", PV[j].name, PV[j].minval));
    if (n > PV[j].maxval)
      panic(errprint2("Configuration␣error:␣%s␣must␣be␣<=␣%d", PV[j].name, PV[j].maxval));
    if (PV[j].power_of_two ∧ (n & (n − 1)))
      panic(errprint1("Configuration␣error:␣%s␣must␣be␣a␣power␣of␣2", PV[j].name));
    *(PV[j].v) = n;
    break;
  }
if (j < PV_size) continue;
```

This code is used in section 19.

21. ⟨ If *token* is a cache name, process a cache spec 21 ⟩ ≡

```
if (strcmp(token, "ITcache") ≡ 0) {
  pcs(ITcache); continue;
} else if (strcmp(token, "DTcache") ≡ 0) {
  pcs(DTcache); continue;
} else if (strcmp(token, "Icache") ≡ 0) {
  if (¬Icache) Icache = new_cache("Icache");
  pcs(Icache); continue;
} else if (strcmp(token, "Dcache") ≡ 0) {
  if (¬Dcache) Dcache = new_cache("Dcache");
  pcs(Dcache); continue;
} else if (strcmp(token, "Scache") ≡ 0) {
  if (¬Icache) Icache = new_cache("Icache");
  if (¬Dcache) Dcache = new_cache("Dcache");
  if (¬Scache) Scache = new_cache("Scache");
  pcs(Scache); continue;
}
```

This code is used in section 19.

22. ⟨ Subroutines 10 ⟩ +≡

```
static void ppol ARGS((replace_policy *));
static void ppol(rr)      /* subroutine to scan for a replacement policy */
    replace_policy *rr;
{
  get_token( );
  if (strcmp(token, "random") ≡ 0) *rr = random;
  else if (strcmp(token, "serial") ≡ 0) *rr = serial;
  else if (strcmp(token, "pseudolru") ≡ 0) *rr = pseudo_lru;
  else if (strcmp(token, "lru") ≡ 0) *rr = lru;
  else token_prescanned = true;      /* oops, we should rescan that token */
}
```

23. ⟨ Subroutines 10 ⟩ +≡

```
static void pcs ARGS((cache *));
static void pcs(c)      /* subroutine to process a cache spec */
    cache *c;
{
  register int j, n;
  get_token( );
  for (j = 0; j < CPV_size; j++)
    if (strcmp(token, CPV[j].name) ≡ 0) break;
  if (j ≡ CPV_size)
    panic(errprint1("Configuration␣syntax␣error:␣‘%s’␣isn’t␣a␣cache␣parameter␣name", token));
  n = get_int( );
  if (n < CPV[j].minval)
    panic(errprint2("Configuration␣error:␣%s␣must␣be␣>=␣%d", CPV[j].name, CPV[j].minval));
  if (n > CPV[j].maxval)
    panic(errprint2("Configuration␣error:␣%s␣must␣be␣<=␣%d", CPV[j].name, CPV[j].maxval));
  if (CPV[j].power_of_two ∧ (n & (n − 1)))
    panic(errprint1("Configuration␣error:␣%s␣must␣be␣power␣of␣2", CPV[j].name));
  switch (CPV[j].v) {
  case assoc: c↣aa = n; ppol(&(c↣repl)); break;
  case blksz: c↣bb = n; break;
  case setsz: c↣cc = n; break;
  case gran: c↣gg = n; break;
  case vctsz: c↣vv = n; ppol(&(c↣vrepl)); break;
  case wrb: c↣mode = (c↣mode & ∼WRITE_BACK) + n * WRITE_BACK; break;
  case wra: c↣mode = (c↣mode & ∼WRITE_ALLOC) + n * WRITE_ALLOC; break;
  case acctm: if (n > max_cycs) max_cycs = n;
    c↣access_time = n; break;
  case citm: if (n > max_cycs) max_cycs = n;
    c↣copy_in_time = n; break;
  case cotm: if (n > max_cycs) max_cycs = n;
    c↣copy_out_time = n; break;
  case prts: c↣ports = n; break;
  }
}
```

24. ⟨ If *token* is an operation name, process a pipe spec 24 ⟩ ≡

```
for (j = 0; j < OP_size; j++)
  if (strcmp(token, OP[j].name) ≡ 0) {
    for (i = 0; ; i++) {
      n = get_int( );
      if (n < 0) break;
      if (n ≡ 0) panic(errprint0("Configuration␣error:␣Pipeline␣cycles␣must␣be␣positive"));
      if (n > 255) panic(errprint0("Configuration␣error:␣Pipeline␣cycles␣must␣be␣<=␣255"));
      if (n > max_cycs) max_cycs = n;
      if (i ≥ pipe_limit)
        panic(errprint1("Configuration␣error:␣More␣than␣%d␣pipeline␣stages", pipe_limit));
      pipe_seq[OP[j].v][i] = n;
    }
    token_prescanned = true;
    break;
  }
if (j < OP_size) continue;
```

This code is used in section 19.

25. ⟨ Process a functional spec 25 ⟩ ≡

```
{
  get_token( );
  if (strlen(token) > 15)
    panic(errprint1("Configuration␣error:␣‘%s’␣is␣more␣than␣15␣characters␣long", token));
  strcpy(funit[funit_count].name, token);
  get_token( );
  if (strlen(token) ≠ 64)
    panic(errprint1("Configuration␣error:␣unit␣%s␣doesn't␣have␣64␣hex␣digit␣specs",
        funit[funit_count].name));
  for (i = j = n = 0; j < 64; j++) {
    if (token[j] ≥ '0' ∧ token[j] ≤ '9') n = (n ≪ 4) + (token[j] − '0');
    else if (token[j] ≥ 'a' ∧ token[j] ≤ 'f') n = (n ≪ 4) + (token[j] − 'a' + 10);
    else if (token[j] ≥ 'A' ∧ token[j] ≤ 'F') n = (n ≪ 4) + (token[j] − 'A' + 10);
    else panic(errprint1("Configuration␣error:␣‘%c’␣is␣not␣a␣hex␣digit", token[j]));
    if ((j & #7) ≡ #7) funit[funit_count].ops[i++] = n, n = 0;
  }
  funit_count++;
  continue;
}
```

This code is used in section 19.

26. Checking and allocating. The battle is only half over when we've absorbed all the data of the configuration file. We still must check for interactions between different quantities, and we must allocate space for cache blocks, coroutines, etc.

One of the most difficult tasks facing us is to determine the maximum number of pipeline stages needed by each functional unit. Let's tackle that first.

```
⟨ Allocate coroutines in each functional unit 26 ⟩ ≡
  ⟨ Build table of pipeline stages needed for each opcode 27 ⟩;
  for (j = 0; j ≤ funit_count; j++) {
    ⟨ Determine the number of stages, n, needed by funit[j] 29 ⟩;
    funit[j].k = n;
    funit[j].co = (coroutine *) calloc(n, sizeof(coroutine));
    for (i = 0; i < n; i++) {
      funit[j].co[i].name = funit[j].name;
      funit[j].co[i].stage = i + 1;
    }
  }
```

This code is used in section 38.

27. ⟨ Build table of pipeline stages needed for each opcode 27 ⟩ ≡

```
  for (j = div; j ≤ max_pipe_op; j++) int_stages[j] = (int) strlen((char *) pipe_seq[j]);
  for ( ; j ≤ max_real_command; j++) int_stages[j] = 1;
  for (j = mul0, n = 0; j ≤ mul8; j++)
    if (strlen((char *) pipe_seq[j]) > (unsigned int) n) n = (int) strlen((char *) pipe_seq[j]);
  int_stages[mul] = n;
  int_stages[ld] = int_stages[st] = int_stages[frem] = 2;
  for (j = 0; j < 256; j++) stages[j] = int_stages[int_op[j]];
```

This code is used in section 26.

28. The *int_op* conversion table is similar to the *internal_op* array of the *MMIX_run* routine, but it replaces *divu* by *div*, *fsub* by *fadd*, etc.

⟨ Global variables 9 ⟩ +≡

```
internal_opcode int_op[256] = {
    trap, fcmp, funeq, funeq, fadd, fix, fadd, fix,
    flot, flot, flot, flot, flot, flot, flot, flot,
    fmul, feps, feps, feps, fdiv, fsqrt, frem, fint,
    mul, mul, mul, mul, div, div, div, div,
    add, add, addu, addu, sub, sub, subu, subu,
    addu, addu, addu, addu, addu, addu, addu, addu,
    cmp, cmp, cmpu, cmpu, sub, sub, subu, subu,
    sh, sh, sh, sh, sh, sh, sh, sh,
    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, ld, ld, ld, ld,
    ld, ld, ld, ld, prego, prego, go, go,
    st, st, st, st, st, st, st, st,
    st, st, st, st, st, st, st, st,
    st, st, st, st, st, st, st, st,
    st, st, st, st, st, st, 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,
    noop, noop, pushj, pushj, set, set, put, put,
    pop, resume, save, unsave, sync, noop, get, trip};
int int_stages[max_real_command + 1];    /* stages as function of internal_opcode */
int stages[256];    /* stages as function of mmix_opcode */
```

29. ⟨ Determine the number of stages, n, needed by *funit*[j] 29 ⟩ ≡

```
for (i = n = 0; i < 256; i++)
  if (((funit[j].ops[i ≫ 5] ≪ (i & #1f)) & #80000000) ∧ stages[i] > n) n = stages[i];
if (n ≡ 0) panic(errprint1("Configuration␣error:␣unit␣%s␣doesn't␣do␣anything", funit[j].name));
```

This code is used in section 26.

30. The next hardest thing on our agenda is to set up the cache structure fields that depend on the parameters. For example, although we have defined the parameter in the *bb* field (the block size), we also need to compute the *b* field (log of the block size), and we must create the cache blocks themselves.

```
⟨ Subroutines 10 ⟩ +≡
  static int lg ARGS((int));
  static int lg(n)      /* compute binary logarithm */
      int n;
  { register int j, l;
    for (j = n, l = 0; j; j >>= 1) l++;
    return l − 1;
  }
```

31.
```
⟨ Subroutines 10 ⟩ +≡
  static void alloc_cache ARGS((cache *, char *));
  static void alloc_cache(c, name)
      cache *c;
      char *name;
  { register int j, k;
    if (c→bb < c→gg)
      panic(errprint1("Configuration␣error:␣blocksize␣of␣%s␣is␣less␣than␣granularity", name));
    if (name[1] ≡ 'T' ∧ c→bb ≠ 8)
      panic(errprint1("Configuration␣error:␣blocksize␣of␣%s␣must␣be␣8", name));
    c→a = lg(c→aa);
    c→b = lg(c→bb);
    c→c = lg(c→cc);
    c→g = lg(c→gg);
    c→v = lg(c→vv);
    c→tagmask = −(1 ≪ (c→b + c→c));
    if (c→a + c→b + c→c ≥ 32)
      panic(errprint1("Configuration␣error:␣%s␣has␣>=␣4␣gigabytes␣of␣data", name));
    if (c→gg ≠ 8 ∧ ¬(c→mode & WRITE_ALLOC))
      panic(errprint2("Configuration␣error:␣%s␣does␣write-around␣with␣granularity␣%d", name,
          c→gg));
    ⟨ Allocate the cache sets for cache c 32 ⟩;
    if (c→vv) ⟨ Allocate the victim cache for cache c 33 ⟩;
    c→inbuf.dirty = (char *) calloc(c→bb ≫ c→g, sizeof(char));
    if (¬c→inbuf.dirty) panic(errprint1("Can't␣allocate␣dirty␣bits␣for␣inbuffer␣of␣%s", name));
    c→inbuf.data = (octa *) calloc(c→bb ≫ 3, sizeof(octa));
    if (¬c→inbuf.data) panic(errprint1("Can't␣allocate␣data␣for␣inbuffer␣of␣%s", name));
    c→outbuf.dirty = (char *) calloc(c→bb ≫ c→g, sizeof(char));
    if (¬c→outbuf.dirty)
      panic(errprint1("Can't␣allocate␣dirty␣bits␣for␣outbuffer␣of␣%s", name));
    c→outbuf.data = (octa *) calloc(c→bb ≫ 3, sizeof(octa));
    if (¬c→outbuf.data) panic(errprint1("Can't␣allocate␣data␣for␣outbuffer␣of␣%s", name));
    if (name[0] ≠ 'S') ⟨ Allocate reader coroutines for cache c 34 ⟩;
  }
```

32. **#define** *sign_bit* #80000000

⟨ Allocate the cache sets for cache *c* 32 ⟩ ≡

```
c→set = (cacheset *) calloc(c→cc, sizeof(cacheset));
if (¬c→set) panic(errprint1("Can't allocate cache sets for %s", name));
for (j = 0; j < c→cc; j++) {
  c→set[j] = (cacheblock *) calloc(c→aa, sizeof(cacheblock));
  if (¬c→set[j]) panic(errprint2("Can't allocate cache blocks for set %d of %s", j, name));
  for (k = 0; k < c→aa; k++) {
    c→set[j][k].tag.h = sign_bit;      /* invalid tag */
    c→set[j][k].dirty = (char *) calloc(c→bb ≫ c→g, sizeof(char));
    if (¬c→set[j][k].dirty)
      panic(errprint3("Can't allocate dirty bits for block %d of set %d of %s", k, j, name));
    c→set[j][k].data = (octa *) calloc(c→bb ≫ 3, sizeof(octa));
    if (¬c→set[j][k].data)
      panic(errprint3("Can't allocate data for block %d of set %d of %s", k, j, name));
  }
}
```

This code is used in section 31.

33. ⟨ Allocate the victim cache for cache *c* 33 ⟩ ≡

```
{
  c→victim = (cacheblock *) calloc(c→vv, sizeof(cacheblock));
  if (¬c→victim) panic(errprint1("Can't allocate blocks for victim cache of %s", name));
  for (k = 0; k < c→vv; k++) {
    c→victim[k].tag.h = sign_bit;      /* invalid tag */
    c→victim[k].dirty = (char *) calloc(c→bb ≫ c→g, sizeof(char));
    if (¬c→victim[k].dirty)
      panic(errprint2("Can't allocate dirty bits for block %d of victim cache of %s", k,
          name));
    c→victim[k].data = (octa *) calloc(c→bb ≫ 3, sizeof(octa));
    if (¬c→victim[k].data)
      panic(errprint2("Can't allocate data for block %d of victim cache of %s", k, name));
  }
}
```

This code is used in section 31.

34. ⟨ Allocate reader coroutines for cache *c* 34 ⟩ ≡

```
{
  c→reader = (coroutine *) calloc(c→ports, sizeof(coroutine));
  if (¬c→reader) panic(errprint1("Can't allocate readers for %s", name));
  for (j = 0; j < c→ports; j++) {
    c→reader[j].stage = vanish;
    c→reader[j].name = (name[0] ≡ 'D' ? (name[1] ≡ 'T' ? "DTreader" : "Dreader") : (name[1] ≡
        'T' ? "ITreader" : "Ireader"));
  }
}
```

This code is used in section 31.

35. ⟨ Allocate the caches 35 ⟩ ≡

```
alloc_cache(ITcache, "ITcache");
ITcache→filler.name = "ITfiller"; ITcache→filler.stage = fill_from_virt;
alloc_cache(DTcache, "DTcache");
DTcache→filler.name = "DTfiller"; DTcache→filler.stage = fill_from_virt;
if (Icache) {
  alloc_cache(Icache, "Icache");
  Icache→filler.name = "Ifiller"; Icache→filler.stage = fill_from_mem;
}
if (Dcache) {
  alloc_cache(Dcache, "Dcache");
  Dcache→filler.name = "Dfiller"; Dcache→filler.stage = fill_from_mem;
  Dcache→flusher.name = "Dflusher"; Dcache→flusher.stage = flush_to_mem;
}
if (Scache) {
  alloc_cache(Scache, "Scache");
  if (Scache→bb < Icache→bb)
    panic(errprint0("Configuration␣error:␣Scache␣blocks␣smaller␣than␣Icache␣blocks"));
  if (Scache→bb < Dcache→bb)
    panic(errprint0("Configuration␣error:␣Scache␣blocks␣smaller␣than␣Dcache␣blocks"));
  if (Scache→gg ≠ Dcache→gg)
    panic(errprint0("Configuration␣error:␣Scache␣granularity␣differs␣from␣the␣Dcache"));
  Icache→filler.stage = fill_from_S;
  Dcache→filler.stage = fill_from_S; Dcache→flusher.stage = flush_to_S;
  Scache→filler.name = "Sfiller"; Scache→filler.stage = fill_from_mem;
  Scache→flusher.name = "Sflusher"; Scache→flusher.stage = flush_to_mem;
}
```

This code is used in section 38.

36. Now we are nearly done. The only nontrivial task remaining is to allocate the ring of queues for coroutine scheduling; for this we need to determine the maximum waiting time that will occur between scheduler and schedulee.

⟨ Allocate the scheduling queue 36 ⟩ ≡

```
bus_words = mem_bus_bytes ≫ 3;
j = (mem_read_time < mem_write_time ? mem_write_time : mem_read_time);
n = 1;
if (Scache ∧ Scache→bb > n) n = Scache→bb;
if (Icache ∧ Icache→bb > n) n = Icache→bb;
if (Dcache ∧ Dcache→bb > n) n = Dcache→bb;
n = mem_addr_time + ((int)(n + mem_bus_bytes − 1)/mem_bus_bytes) ∗ j;
if (n > max_cycs) max_cycs = n;      /∗ now max_cycs bounds the waiting time ∗/
ring_size = max_cycs + 1;
ring = (coroutine ∗) calloc(ring_size, sizeof(coroutine));
if (¬ring) panic(errprint0("Can't␣allocate␣the␣scheduling␣ring"));
{ register coroutine ∗p;
  for (p = ring; p < ring + ring_size; p++) {
    p→name = "";      /∗ header nodes are nameless ∗/
    p→stage = max_stage;
  }
}
```

This code is used in section 38.

37. ⟨ Touch up last-minute trivia 37 ⟩ ≡

```
if (hash_prime ≤ mem_chunks_max)
  panic(errprint0("Configuration␣error:␣hashprime␣must␣exceed␣memchunksmax"));
mem_hash = (chunknode *) calloc(hash_prime + 1, sizeof(chunknode));
if (¬mem_hash) panic(errprint0("Can't␣allocate␣the␣hash␣table"));
mem_hash[0].chunk = (octa *) calloc(1 ≪ 13, sizeof(octa));
if (¬mem_hash[0].chunk) panic(errprint0("Can't␣allocate␣chunk␣0"));
mem_hash[hash_prime].chunk = (octa *) calloc(1 ≪ 13, sizeof(octa));
if (¬mem_hash[hash_prime].chunk) panic(errprint0("Can't␣allocate␣0␣chunk"));
mem_chunks = 1;
fetch_bot = (fetch *) calloc(fetch_buf_size + 1, sizeof(fetch));
if (¬fetch_bot) panic(errprint0("Can't␣allocate␣the␣fetch␣buffer"));
fetch_top = fetch_bot + fetch_buf_size;
reorder_bot = (control *) calloc(reorder_buf_size + 1, sizeof(control));
if (¬reorder_bot) panic(errprint0("Can't␣allocate␣the␣reorder␣buffer"));
reorder_top = reorder_bot + reorder_buf_size;
wbuf_bot = (write_node *) calloc(write_buf_size + 1, sizeof(write_node));
if (¬wbuf_bot) panic(errprint0("Can't␣allocate␣the␣write␣buffer"));
wbuf_top = wbuf_bot + write_buf_size;
if (bp_n ≡ 0) bp_table = Λ;
else {      /* a branch prediction table is desired */
  if (bp_a + bp_b + bp_c ≥ 31)
    panic(errprint0("Configuration␣error:␣Branch␣table␣has␣>=␣2␣gigabytes␣of␣data"));
  bp_table = (char *) calloc(1 ≪ (bp_a + bp_b + bp_c), sizeof(char));
  if (¬bp_table) panic(errprint0("Can't␣allocate␣the␣branch␣table"));
}
l = (specnode *) calloc(lring_size, sizeof(specnode));
if (¬l) panic(errprint0("Can't␣allocate␣local␣registers"));
j = bus_words;
if (Icache ∧ (Icache¬bb ≫ 3) > j) j = Icache¬bb ≫ 3;
fetched = (octa *) calloc(j, sizeof(octa));
if (¬fetched) panic(errprint0("Can't␣allocate␣prefetch␣buffer"));
dispatch_stat = (int *) calloc(dispatch_max + 1, sizeof(int));
if (¬dispatch_stat) panic(errprint0("Can't␣allocate␣dispatch␣counts"));
no_hardware_PT = 1 − hardware_PT;
```

This code is used in section 38.

38. Putting it all together. Here then is the desired configuration subroutine.

```
#include <stdio.h>     /* fopen, fgets, sscanf, rewind */
#include <stdlib.h>    /* calloc, exit */
#include <ctype.h>     /* isspace */
#include <string.h>    /* strcpy, strlen, strcmp */
#include <limits.h>    /* INT_MAX */
#include "mmix-pipe.h"
  ⟨ Type definitions 12 ⟩
  ⟨ Global variables 9 ⟩
  ⟨ Subroutines 10 ⟩
  void MMIX_config(filename)
      char *filename;
  { register int i, j, n;
    config_file = fopen(filename, "r");
    if (¬config_file) panic(errprint1("Can't␣open␣configuration␣file␣%s", filename));
    ⟨ Initialize to defaults 17 ⟩;
    ⟨ Count and allocate the functional units 18 ⟩;
    ⟨ Record all the specs 19 ⟩;
    ⟨ Allocate coroutines in each functional unit 26 ⟩;
    ⟨ Allocate the caches 35 ⟩;
    ⟨ Allocate the scheduling queue 36 ⟩;
    ⟨ Touch up last-minute trivia 37 ⟩;
  }
```

39. Index.

⟨ Allocate coroutines in each functional unit 26 ⟩ Used in section 38.
⟨ Allocate reader coroutines for cache c 34 ⟩ Used in section 31.
⟨ Allocate the cache sets for cache c 32 ⟩ Used in section 31.
⟨ Allocate the caches 35 ⟩ Used in section 38.
⟨ Allocate the scheduling queue 36 ⟩ Used in section 38.
⟨ Allocate the victim cache for cache c 33 ⟩ Used in section 31.
⟨ Build table of pipeline stages needed for each opcode 27 ⟩ Used in section 26.
⟨ Count and allocate the functional units 18 ⟩ Used in section 38.
⟨ Determine the number of stages, n, needed by $funit[j]$ 29 ⟩ Used in section 26.
⟨ Global variables 9, 15, 28 ⟩ Used in section 38.
⟨ If *token* is a cache name, process a cache spec 21 ⟩ Used in section 19.
⟨ If *token* is a parameter name, process a PV spec 20 ⟩ Used in section 19.
⟨ If *token* is an operation name, process a pipe spec 24 ⟩ Used in section 19.
⟨ Initialize to defaults 17 ⟩ Used in section 38.
⟨ Process a functional spec 25 ⟩ Used in section 19.
⟨ Record all the specs 19 ⟩ Used in section 38.
⟨ Subroutines 10, 11, 16, 22, 23, 30, 31 ⟩ Used in section 38.
⟨ Touch up last-minute trivia 37 ⟩ Used in section 38.
⟨ Type definitions 12, 13, 14 ⟩ Used in section 38.

November 7, 2022 at 05:29

MMIX-CONFIG