

1. Introduction. This CWEB program simulates how the MMIX computer might be implemented with a high-performance pipeline in many different configurations. All of the complexities of MMIX's architecture are treated, except for multiprocessing and low-level details of memory mapped input/output.

The present program module, which contains the main routine for the MMIX meta-simulator, is primarily devoted to administrative tasks. Other modules do the actual work after this module has told them what to do.

2. A user typically invokes the meta-simulator with a UNIX-like command line of the general form `'mmix configfile progfile'` where the `configfile` describes the characteristics of an MMIX implementation and the `progfile` contains a program to be downloaded and run. Rules for configuration files appear in the module called `mmix-config`. The program file is either an "MMIX binary file" dumped by MMIX-SIM, or an ASCII text file that describes hexadecimal data in a rudimentary format. It is assumed to be binary if its name ends with the extension `'.mmb'`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mmix-pipe.h"
char *config_file_name, *prog_file_name;
<Global variables 5>
<Subroutines 10>
int main(argc, argv)
    int argc;
    char *argv[];
{
    <Parse the command line 3>;
    MMIX_config(config_file_name);
    MMIX_init();
    mmix_io_init();
    <Input the program 4>;
    <Run the simulation interactively 13>;
    printf("Simulation ended at time %d.\n", ticks.l);
    print_stats();
    return 0;
}
```

3. The command line might also contain options, some day. For now I'm forgetting them and simplifying everything until I gain further experience.

```
<Parse the command line 3> ≡
if (argc ≠ 3) {
    fprintf(stderr, "Usage: %s configfile progfile\n", argv[0]);
    exit(-3);
}
config_file_name = argv[1];
prog_file_name = argv[2];
```

This code is used in section 2.

```
4. <Input the program 4> ≡
if (strlen(prog_file_name) > 4 ∧ strcmp(prog_file_name + strlen(prog_file_name) - 4, ".mmb") ≡ 0)
    <Input an MMIX binary file 9>
else <Input a rudimentary hexadecimal file 6>;
fclose(prog_file);
```

This code is used in section 2.

5. Hexadecimal input to memory. A rudimentary hexadecimal input format is implemented here so that the simulator can be run with essentially arbitrary data in the simulated memory. The rules of this format are extremely simple: Each line of the file either begins with (i) 12 hexadecimal digits followed by a colon; or (ii) a space followed by 16 hexadecimal digits. In case (i), the 12 hex digits specify a 48-bit physical address, called the current location. In case (ii), the 16 hex digits specify an octabyte to be stored in the current location; the current location is then increased by 8. The current location should be a multiple of 8, but its three least significant bits are actually ignored. Arbitrary comments can follow the specification of a new current location or a new octabyte, as long as each line is less than 99 characters long. For example, the file

```
0123456789ab: SILLY EXAMPLE
0123456789abcdef first octabyte
fedbca9876543210 second
```

places the octabyte `#0123456789abcdef` into memory location `#0123456789a8` and `#fedbca9876543210` into location `#0123456789b0`.

```
#define BUF_SIZE 100
⟨Global variables 5⟩ ≡
    octa cur_loc;
    octa cur_dat;
    bool new_chunk;
    char buffer[BUF_SIZE];
    FILE *prog_file;
```

See also sections 16 and 25.

This code is used in section 2.

```
6. ⟨Input a rudimentary hexadecimal file 6⟩ ≡
{
    prog_file = fopen(prog_file_name, "r");
    if (!prog_file) {
        fprintf(stderr, "Panic: Can't open MMIX hexadecimal file %s!\n", prog_file_name);
        exit(-3);
    }
    new_chunk = true;
    while (1) {
        if (!fgets(buffer, BUF_SIZE, prog_file)) break;
        if (buffer[strlen(buffer) - 1] != '\n') {
            fprintf(stderr, "Panic: Hexadecimal file line too long: '%s...'\n", buffer);
            exit(-3);
        }
        if (buffer[12] == ':') ⟨Change the current location 7⟩
        else if (buffer[0] == ' ') ⟨Read an octabyte and advance cur_loc 8⟩
        else {
            fprintf(stderr, "Panic: Improper hexadecimal file line: '%s'\n", buffer);
            exit(-3);
        }
    }
}
```

This code is used in section 4.

```

7. <Change the current location 7> ≡
{
  if (sscanf(buffer, "%4x%8x", &cur_loc.h, &cur_loc.l) ≠ 2) {
    fprintf(stderr, "Panic: Improper hexadecimal file location: '%s'!\n", buffer);
    exit(-3);
  }
  new_chunk = true;
}

```

This code is used in section 6.

```

8. <Read an octabyte and advance cur_loc 8> ≡
{
  if (sscanf(buffer + 1, "%8x%8x", &cur_dat.h, &cur_dat.l) ≠ 2) {
    fprintf(stderr, "Panic: Improper hexadecimal file data: '%s'!\n", buffer);
    exit(-3);
  }
  if (new_chunk) mem_write(cur_loc, cur_dat);
  else mem_hash[last_h].chunk[(cur_loc.l & #ffff) >> 3] = cur_dat;
  cur_loc.l += 8;
  if ((cur_loc.l & #fff8) ≠ 0) new_chunk = false;
  else {
    new_chunk = true;
    if ((cur_loc.l & #ffff0000) ≡ 0) cur_loc.h++;
  }
}

```

This code is used in section 6.

9. Binary input to memory. When the program file was dumped by MMIX-SIM, it has the simple format discussed in exercise 1.4.3'–20 of the MMIX fascicle [*The Art of Computer Programming*, Volume 1, Fascicle 1]. We assume that such a program has text, data, pool, and stack segments, as in the conventions of that book. We load it into four 2^{32} -byte pages of physical memory, one for each segment; page zero of segment i is mapped to physical location $2^{32}i$. Page tables are kept in physical locations starting at $2^{32} \times 4$; static traps begin at $2^{32} \times 5$ and dynamic traps at $2^{32} \times 6$. (These conventions agree with the special register settings $rT = \#8000000500000000$, $rTT = \#8000000600000000$, $rV = \#369c200400000000$ assumed by the stripped-down simulator.)

```

⟨Input an MMIX binary file 9⟩ ≡
{
  prog_file = fopen(prog_file_name, "rb");
  if (¬prog_file) {
    fprintf(stderr, "Panic: Can't open MMIX binary file %s!\n", prog_file_name);
    exit(-3);
  }
  while (1) {
    if (¬undump_octa()) break;
    new_chunk = true;
    cur_loc = cur_dat;
    if (cur_loc.h & #9fffffff) bad_address = true;
    else bad_address = false, cur_loc.h >>= 29;
    /* apply trivial mapping function for each segment */
    ⟨Input consecutive octabytes beginning at cur_loc 11⟩;
  }
  ⟨Set up the canned environment 12⟩;
}

```

This code is used in section 4.

10. The *undump_octa* routine reads eight bytes from the binary file *prog_file* into the global octabyte *cur_dat*, taking care as usual to be big-endian regardless of the host computer's bias.

```

⟨Subroutines 10⟩ ≡
static bool undump_octa ARGS((void));
static bool undump_octa()
{
  register int t0, t1, t2, t3;
  t0 = fgetc(prog_file); if (t0 ≡ EOF) return false;
  t1 = fgetc(prog_file); if (t1 ≡ EOF) goto oops;
  t2 = fgetc(prog_file); if (t2 ≡ EOF) goto oops;
  t3 = fgetc(prog_file); if (t3 ≡ EOF) goto oops;
  cur_dat.h = (t0 << 24) + (t1 << 16) + (t2 << 8) + t3;
  t0 = fgetc(prog_file); if (t0 ≡ EOF) goto oops;
  t1 = fgetc(prog_file); if (t1 ≡ EOF) goto oops;
  t2 = fgetc(prog_file); if (t2 ≡ EOF) goto oops;
  t3 = fgetc(prog_file); if (t3 ≡ EOF) goto oops;
  cur_dat.l = (t0 << 24) + (t1 << 16) + (t2 << 8) + t3;
  return true;
oops: fprintf(stderr, "Premature end of file on %s!\n", prog_file_name);
  return false;
}

```

See also sections 17 and 20.

This code is used in section 2.

```

11. ⟨Input consecutive octabytes beginning at cur_loc 11⟩ ≡
while (1) {
  if ( $\neg$ undump_octa()) {
    fprintf(stderr, "Unexpected_end_of_file_on_%s!\n", prog_file_name);
    break;
  }
  if ( $\neg$ (cur_dat.h  $\vee$  cur_dat.l)) break;
  if (bad_address) {
    fprintf(stderr, "Panic: Unsupported_virtual_address_%08x%08x!\n", cur_loc.h, cur_loc.l);
    exit(-5);
  }
  if (new_chunk) mem_write(cur_loc, cur_dat);
  else mem_hash[last_h].chunk[(cur_loc.l & #ffff)  $\gg$  3] = cur_dat;
  cur_loc.l += 8;
  if ((cur_loc.l & #fff8)  $\neq$  0) new_chunk = false;
  else {
    new_chunk = true;
    if ((cur_loc.l & #fff0000)  $\equiv$  0) {
      bad_address = true;
      cur_loc.h = (cur_loc.h  $\ll$  29) + 1;
    }
  }
}

```

This code is used in section 9.

12. The primitive operating system assumed in simple programs of *The Art of Computer Programming* will set up text segment, data segment, pool segment, and stack segment as in MMIX-SIM. The runtime stack will be initialized if we UNSAVE from the last location loaded in the .mmb file.

```
#define rQ 16
⟨Set up the canned environment 12⟩ ≡
  if (cur_loc.h ≠ 3) {
    fprintf(stderr, "Panic: MMIX binary file didn't set up the stack!\n");
    exit(-6);
  }
  inst_ptr.o = mem_read(incr(cur_loc, -8 * 14)); /* Main */
  inst_ptr.p = Λ;
  cur_loc.h = #60000000;
  g[255].o = incr(cur_loc, -8); /* place to UNSAVE */
  cur_dat.l = #90;
  if (mem_read(cur_dat).h) inst_ptr.o = cur_dat; /* start at #90 if nonzero */
  head-inst = (UNSAVE ≪ 24) + 255, tail--; /* prefetch a fabricated command */
  head-loc = incr(inst_ptr.o, -4); /* in case the UNSAVE is interrupted */
  g[rT].o.h = #80000005, g[rTT].o.h = #80000006;
  cur_dat.h = (RESUME ≪ 24) + 1, cur_dat.l = 0, cur_loc.h = 5, cur_loc.l = 0;
  mem_write(cur_loc, cur_dat); /* the primitive trap handler */
  cur_dat.l = cur_dat.h, cur_dat.h = (NEGI ≪ 24) + (255 ≪ 16) + 1;
  cur_loc.h = 6, cur_loc.l = 8;
  mem_write(cur_loc, cur_dat); /* the primitive dynamic trap handler */
  cur_dat.h = (GET ≪ 24) + rQ, cur_dat.l = (PUTI ≪ 24) + (rQ ≪ 16), cur_loc.l = 0;
  mem_write(cur_loc, cur_dat); /* more of the primitive dynamic trap handler */
  cur_dat.h = 0, cur_dat.l = 7; /* generate a PTE with rwx permission */
  cur_loc.h = 4; /* beginning of skeleton page table */
  mem_write(cur_loc, cur_dat); /* PTE for the text segment */
  ITcache-set[0][0].tag = zero_octa;
  ITcache-set[0][0].data[0] = cur_dat; /* prime the IT cache */
  cur_dat.l = 6; /* PTE with read and write permission only */
  cur_dat.h = 1, cur_loc.l = 3 ≪ 13;
  mem_write(cur_loc, cur_dat); /* PTE for the data segment */
  cur_dat.h = 2, cur_loc.l = 6 ≪ 13;
  mem_write(cur_loc, cur_dat); /* PTE for the pool segment */
  cur_dat.h = 3, cur_loc.l = 9 ≪ 13;
  mem_write(cur_loc, cur_dat); /* PTE for the stack segment */
  g[rK].o = neg_one; /* enable all interrupts */
  g[rV].o.h = #369c2004;
  page_bad = false, page_r = 4 ≪ (32 - 13), page_s = 32, page_mask.l = #ffffffff;
  page_b[1] = 3, page_b[2] = 6, page_b[3] = 9, page_b[4] = 12;
```

This code is used in section 9.

13. Interaction. When prompted for instructions, this simulator understands the following terse commands:

- \langle positive integer \rangle : Run for this many clock cycles.
- \textcircled{h} (hexadecimal integer): Set the instruction pointer to this virtual address; successive instructions will be fetched from here.
- **k**: Toggle the sign bit of the instruction pointer.
- **b** (hexadecimal integer): Set the breakpoint to this virtual address; simulation will pause when an instruction from the breakpoint address enters the fetch buffer.
- **v** (hexadecimal integer): Set the desired level of diagnostic output; each bit in the hexadecimal integer enables certain printouts when the simulator is running. Bit #1 shows instructions when issued, deissued, or committed; #2 shows the pipeline and locks after each cycle; #4 shows each coroutine activation; #8 each coroutine scheduling; #10 reports when reading from an uninitialized chunk of memory; #20 asks for online input when reading from addresses $\geq 2^{48}$; #40 reports all I/O to memory address $\geq 2^{48}$; #80 shows details of branch prediction; #100 displays full cache contents including blocks with invalid tags.
- $-$ (integer): Deissue this many instructions.
- **l** (integer) or **g** (integer): Show current “hot” contents of a local or global register.
- **m** (hexadecimal integer): Show current contents of a physical memory address. (This value may not be up to date; newer values might appear in the write buffer and/or in the caches.)
- **f** (hexadecimal integer): Insert a tetrabyte into the fetch buffer. (Use with care!)
- **i** (integer): Set the interval counter **rI** to the given value; this will trigger an interrupt after the specified number of cycles.
- **IT**, **DT**, **I**, **D**, or **S**: Show current contents of a cache.
- **D*** or **S***: Show dirty blocks of a cache.
- **p**: Show current contents of the pipeline.
- **s**: Show current statistics on branch prediction and speed of instruction issue.
- **h**: Help (show the possibilities for interaction).
- **q**: Quit.

\langle Run the simulation interactively 13 $\rangle \equiv$

```

while (1) {
    printf("mmmix>"); fflush(stdout);
    fgets(buffer, BUF_SIZE, stdin);
    switch (buffer[0]) {
        default: what_say: printf("Eh? Sorry, I don't understand. (Type_h_for_help)\n");
            continue;
        case 'q': case 'x': goto done;
    }
    }
done:

```

This code is used in section 2.

14. \langle Cases for interaction 14 $\rangle \equiv$

```

case 'h': case '?'': printf("The interactive commands are as follows:\n");
    printf("  <n> to run for n cycles\n");
    printf("  0<x> to take next instruction from location x\n");
    printf("  k to change the sign bit of the instruction location\n");
    printf("  b<x> to pause when location x is fetched\n");
    printf("  v<x> to print specified diagnostics when running;\n");
    printf("  x=1[insts enter/leave pipe]+2[whole pipeline each cycle]+\n");
    printf("  x=4[coroutine activations]+8[coroutine scheduling]+\n");
    printf("  x=10[uninitialized read]+20[online I/O read]+\n");
    printf("  x=40[I/O read/write]+80[branch prediction details]+\n");
    printf("  x=100[invalid cache blocks displayed too]\n");
    printf("  -<n> to deissue n instructions\n");
    printf("  l<n> to print current value of local register n\n");
    printf("  g<n> to print current value of global register n\n");
    printf("  m<x> to print current value of memory address x\n");
    printf("  f<x> to insert instruction x into the fetch buffer\n");
    printf("  i<n> to initiate a timer interrupt after n cycles\n");
    printf("  IT, DT, I, D, or S to print current cache contents\n");
    printf("  D* or S* to print dirty blocks of a cache\n");
    printf("  p to print current pipeline contents\n");
    printf("  s to print current stats\n");
    printf("  h to print this message\n");
    printf("  q to exit\n");
    printf(" (Here <n> is a decimal integer, <x> is hexadecimal.)\n");
continue;

```

See also sections 15, 18, 19, 21, 22, 23, and 24.

This code is used in section 13.

15. \langle Cases for interaction 14 $\rangle + \equiv$

```

case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8':
case '9':
    if (sscanf(buffer, "%d", &n)  $\neq$  1) goto what_say;
    printf("Running %d at time %d", n, ticks.l);
    if (bp.h  $\equiv$  (tetra) - 1  $\wedge$  bp.l  $\equiv$  (tetra) - 1) printf("\n");
    else printf("  with breakpoint %08x%08x\n", bp.h, bp.l);
    MMIX_run(n, bp); continue;
case '0': inst_ptr.o = read_hex(buffer + 1); goto new_inst_ptr;
case 'k': inst_ptr.o.h  $\oplus$  = #80000000; /* shortcut to kernel mode */
    if ( $\neg$ ticks.l  $\wedge$  head) head-loc.h  $\oplus$  = #80000000; /* fix the UNSAVE loc */
new_inst_ptr: if (inst_ptr.o.h & #80000000) g[rK].o.h &= -2; /* disable interrupts on P_BIT */
    inst_ptr.p =  $\Lambda$ ; continue;
case 'b': bp = read_hex(buffer + 1); continue;
case 'v': verbose = read_hex(buffer + 1).l; continue;

```

16. \langle Global variables 5 $\rangle + \equiv$

```

int n, m; /* temporary integer */
octa bp = {-1, -1}; /* breakpoint */
octa tmp; /* an octabyte of temporary interest */
static unsigned char d[BUF_SIZE];

```


17. Here's a simple program to read an octabyte in hexadecimal notation from a buffer. It changes the buffer by storing a null character after the input.

```

⟨Subroutines 10⟩ +=
  octa read_hex ARGS((char *));
  octa read_hex(p)
    char *p;
  {
    register int j, k;
    octa val;
    val.h = val.l = 0;
    for (j = 0; ; j++) {
      if (p[j] ≥ '0' ∧ p[j] ≤ '9') d[j] = p[j] - '0';
      else if (p[j] ≥ 'a' ∧ p[j] ≤ 'f') d[j] = p[j] - 'a' + 10;
      else if (p[j] ≥ 'A' ∧ p[j] ≤ 'F') d[j] = p[j] - 'A' + 10;
      else break;
    }
    p[j] = '\0';
    for (j--, k = 0; k ≤ j; k++) {
      if (k ≥ 8) val.h += d[j - k] << (4 * k - 32);
      else val.l += d[j - k] << (4 * k);
    }
    return val;
  }

```

18. ⟨Cases for interaction 14⟩ +=

```

case '-': if (sscanf(buffer + 1, "%d", &n) ≠ 1 ∨ n < 0) goto what_say;
  if (cool ≤ hot) m = hot - cool; else m = (hot - reorder_bot) + 1 + (reorder_top - cool);
  if (n > m) deissues = m; else deissues = n;
  continue;
case '1': if (sscanf(buffer + 1, "%d", &n) ≠ 1 ∨ n < 0) goto what_say;
  if (n ≥ bring_size) goto what_say;
  printf("␣␣1 [%d]=%08x%08x\n", n, l[n].o.h, l[n].o.l); continue;
case 'm': tmp = mem_read(read_hex(buffer + 1));
  printf("␣␣m [%s]=%08x%08x\n", buffer + 1, tmp.h, tmp.l); continue;

```

19. The register stack pointers, rO and rS, are not kept up to date in the *g* array. Therefore we have to deduce their values by examining the pipeline.

⟨Cases for interaction 14⟩ +=

```

case 'g': if (sscanf(buffer + 1, "%d", &n) ≠ 1 ∨ n < 0) goto what_say;
  if (n ≥ 256) goto what_say;
  if (n ≡ rO ∨ n ≡ rS) {
    if (hot ≡ cool) /* pipeline empty */
      g[rO].o = sl3(cool_O), g[rS].o = sl3(cool_S);
    else g[rO].o = sl3(hot-cur_O), g[rS].o = sl3(hot-cur_S);
  }
  printf("␣␣g [%d]=%08x%08x\n", n, g[n].o.h, g[n].o.l);
  continue;

```

20. \langle Subroutines 10 $\rangle + \equiv$

```
static octa sl3 ARGS((octa));
static octa sl3(y) /* shift left by 3 bits */
    octa y;
{
    register tetra yhl = y.h << 3, ylh = y.l >> 29;
    y.h = yhl + ylh; y.l <<= 3;
    return y;
}
```

21. \langle Cases for interaction 14 $\rangle + \equiv$

```
case 'I': print_cache(buffer[1] == 'T' ? ITcache : Icache, false); continue;
case 'D': print_cache(buffer[1] == 'T' ? DTcache : Dcache,
    buffer[1] == '*'); continue;
case 'S': print_cache(Scache, buffer[1] == '*'); continue;
case 'p': print_pipe(); print_locks(); continue;
case 's': print_stats(); continue;
case 'i': if (sscanf(buffer + 1, "%d", &n) == 1) g[rI].o = incr(zero_octa, n);
    continue;
```

22. \langle Cases for interaction 14 $\rangle + \equiv$

```
case 'f': tmp = read_hex(buffer + 1);
{
    register fetch *new_tail;
    if (tail == fetch_bot) new_tail = fetch_top;
    else new_tail = tail - 1;
    if (new_tail == head) printf("Sorry, the fetch buffer is full!\n");
    else {
        tail-loc = inst_ptr.o;
        tail-inst = tmp.l;
        tail-interrupt = 0;
        tail-noted = false;
        tail = new_tail;
    }
    continue;
}
```

23. A hidden case here, for me when debugging. It essentially disables the translation caches, by mapping everything to zero.

\langle Cases for interaction 14 $\rangle + \equiv$

```
case 'd': if (ticks.l) printf("Sorry: I disable ITcache and DTcache only at the beginning!\n");
    else {
        ITcache->set[0][0].tag = zero_octa;
        ITcache->set[0][0].data[0] = seven_octa;
        DTcache->set[0][0].tag = zero_octa;
        DTcache->set[0][0].data[0] = seven_octa;
        g[rK].o = neg_one;
        page_bad = false;
        page_mask = neg_one;
        inst_ptr.p = (specnode *) 1;
    } continue;
```

24. And another case, for me when kludging. At the moment, it simply lists the functional unit names. But I might decide to put other stuff here when giving a demo.

⟨Cases for interaction 14⟩ +≡

```

case '!' : {
    register int j;
    for (j = 0; j < funit_count; j++) printf("unit_□%s_□%d\n", funit[j].name, funit[j].k);
}
continue;

```

25. ⟨Global variables 5⟩ +≡

```

bool bad_address;
extern bool page_bad;
extern octa page_mask;
extern int page_r, page_s, page_b[5];
extern octa zero_octa;
extern octa neg_one;
octa seven_octa = {0, 7};
extern octa incr ARGS((octa y, int delta));    /* unsigned y +  $\delta$  ( $\delta$  is signed) */
extern void mmix_io_init ARGS((void));
extern void MMIX_config ARGS((char *));

```

26. Index.

- argc*: [2](#), [3](#).
- ARGS**: [10](#), [17](#), [20](#), [25](#).
- argv*: [2](#), [3](#).
- bad_address*: [9](#), [11](#), [25](#).
- big-endian versus little-endian: [10](#).
- binary files: [9](#).
- bp*: [15](#), [16](#).
- BUF_SIZE**: [5](#), [6](#), [13](#), [16](#).
- buffer*: [5](#), [6](#), [7](#), [8](#), [13](#), [15](#), [18](#), [19](#), [21](#), [22](#).
- Can't open...**: [6](#), [9](#).
- chunk*: [8](#), [11](#).
- config_file_name*: [2](#), [3](#).
- cool*: [18](#), [19](#).
- cool_O*: [19](#).
- cool_S*: [19](#).
- cur_dat*: [5](#), [8](#), [9](#), [10](#), [11](#), [12](#).
- cur_loc*: [5](#), [7](#), [8](#), [9](#), [11](#), [12](#).
- cur_O*: [19](#).
- cur_S*: [19](#).
- d*: [16](#).
- data*: [12](#), [23](#).
- Dcache*: [21](#).
- deissues*: [18](#).
- delta*: [25](#).
- done*: [13](#).
- DTcache*: [21](#), [23](#).
- EOF**: [10](#).
- exit*: [3](#), [6](#), [7](#), [8](#), [9](#), [11](#), [12](#).
- false*: [8](#), [9](#), [10](#), [11](#), [12](#), [21](#), [22](#), [23](#).
- Fascicle 1: [9](#).
- fclose*: [4](#).
- fetch**: [22](#).
- fetch_bot*: [22](#).
- fetch_top*: [22](#).
- fflush*: [13](#).
- fgetc*: [10](#).
- fgets*: [6](#), [13](#).
- fopen*: [6](#), [9](#).
- fprintf*: [3](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#).
- funit*: [24](#).
- funit_count*: [24](#).
- GET**: [12](#).
- head*: [12](#), [15](#), [22](#).
- Hexadecimal file line...: [6](#).
- hexadecimal files: [5](#).
- hot*: [18](#), [19](#).
- Icache*: [21](#).
- Improper hexadecimal...: [6](#), [7](#), [8](#).
- incr*: [12](#), [21](#), [25](#).
- inst*: [12](#), [22](#).
- inst_ptr*: [12](#), [15](#), [22](#), [23](#).
- interrupt*: [22](#).
- ITcache*: [12](#), [21](#), [23](#).
- j*: [17](#), [24](#).
- k*: [17](#).
- last_h*: [8](#), [11](#).
- little-endian versus big-endian: [10](#).
- loc*: [12](#), [15](#), [22](#).
- bring_size*: [18](#).
- m*: [16](#).
- main*: [2](#).
- mem_hash*: [8](#), [11](#).
- mem_read*: [12](#), [18](#).
- mem_write*: [8](#), [11](#), [12](#).
- MMIX binary file...**: [12](#).
- MMIX_config*: [2](#), [25](#).
- MMIX_init*: [2](#).
- mmix_io_init*: [2](#), [25](#).
- MMIX_run*: [15](#).
- mmmix>**: [13](#).
- n*: [16](#).
- name*: [24](#).
- neg_one*: [12](#), [23](#), [25](#).
- NEGI**: [12](#).
- new_chunk*: [5](#), [6](#), [7](#), [8](#), [9](#), [11](#).
- new_inst_ptr*: [15](#).
- new_tail*: [22](#).
- noted*: [22](#).
- octa**: [5](#), [16](#), [17](#), [20](#), [25](#).
- oops*: [10](#).
- p*: [17](#).
- P_BIT**: [15](#).
- page_b*: [12](#), [25](#).
- page_bad*: [12](#), [23](#), [25](#).
- page_mask*: [12](#), [23](#), [25](#).
- page_r*: [12](#), [25](#).
- page_s*: [12](#), [25](#).
- Premature end of file...: [10](#).
- print_cache*: [21](#).
- print_locks*: [21](#).
- print_pipe*: [21](#).
- print_stats*: [2](#), [21](#).
- printf*: [2](#), [13](#), [14](#), [15](#), [18](#), [19](#), [22](#), [23](#), [24](#).
- prog_file*: [4](#), [5](#), [6](#), [9](#), [10](#).
- prog_file_name*: [2](#), [3](#), [4](#), [6](#), [9](#), [10](#), [11](#).
- PUTI**: [12](#).
- radix conversion: [17](#).
- read_hex*: [15](#), [17](#), [18](#), [22](#).
- reorder_bot*: [18](#).
- reorder_top*: [18](#).
- RESUME**: [12](#).
- rI*: [21](#).

rK: 12, 15, 23.
rO: 19.
rQ: [12](#).
rS: 19.
rT: 12.
rTT: 12.
rV: 12.
Scache: 21.
segments: 9.
set: 12, 23.
seven_octa: 23, [25](#).
sl3: 19, [20](#).
specnode: 23.
sscanf: 7, 8, 15, 18, 19, 21.
stderr: 3, 6, 7, 8, 9, 10, 11, 12.
stdin: 13.
stdout: 13.
strcmp: 4.
strlen: 4, 6.
tag: 12, 23.
tail: 12, 22.
tetra: 15, 20.
ticks: 2, 15, 23.
tmp: [16](#), 18, [22](#).
true: 6, 7, 8, 9, 10, 11.
t0: [10](#).
t1: [10](#).
t2: [10](#).
t3: [10](#).
undump_octa: 9, [10](#), 11.
Unexpected end of file...: 11.
UNSAVE: 12.
Unsupported virtual address: 11.
Usage: ...: 3.
val: [17](#).
verbose: 15.
what_say: [13](#), 15, 18, 19.
y: [20](#), [25](#).
yhl: [20](#).
ylh: [20](#).
zero_octa: 12, 21, 23, [25](#).

- ⟨ Cases for interaction [14](#), [15](#), [18](#), [19](#), [21](#), [22](#), [23](#), [24](#) ⟩ Used in section [13](#).
- ⟨ Change the current location [7](#) ⟩ Used in section [6](#).
- ⟨ Global variables [5](#), [16](#), [25](#) ⟩ Used in section [2](#).
- ⟨ Input a rudimentary hexadecimal file [6](#) ⟩ Used in section [4](#).
- ⟨ Input an MMIX binary file [9](#) ⟩ Used in section [4](#).
- ⟨ Input consecutive octabytes beginning at *cur_loc* [11](#) ⟩ Used in section [9](#).
- ⟨ Input the program [4](#) ⟩ Used in section [2](#).
- ⟨ Parse the command line [3](#) ⟩ Used in section [2](#).
- ⟨ Read an octabyte and advance *cur_loc* [8](#) ⟩ Used in section [6](#).
- ⟨ Run the simulation interactively [13](#) ⟩ Used in section [2](#).
- ⟨ Set up the canned environment [12](#) ⟩ Used in section [9](#).
- ⟨ Subroutines [10](#), [17](#), [20](#) ⟩ Used in section [2](#).

MMIX

	Section	Page
Introduction	1	1
Hexadecimal input to memory	5	2
Binary input to memory	9	4
Interaction	13	7
Index	26	12

© 1999 Donald E. Knuth

This file may be freely copied and distributed, provided that no changes whatsoever are made. All users are asked to help keep the MMIXware files consistent and “uncorrupted,” identical everywhere in the world. Changes are permissible only if the modified file is given a new name, different from the names of existing files in the MMIXware package, and only if the modified file is clearly identified as not being part of that package. (The CWEB system has a “change file” facility by which users can easily make minor alterations without modifying the master source files in any way. Everybody is supposed to use change files instead of changing the files.) The author has tried his best to produce correct and useful programs, in order to help promote computer science research, but no warranty of any kind should be assumed.