# m68k

**The chip itself**

---

**Motorola 68k**

- The m68k is a 16/32-bit CISC microprocessor made by Motorola in 1979.
- One of the last Big-Endian ISAs.
- It was a popular Instruction Set in the 80s, powering Apple computers, Amigas, Sinclairs, X68000, Ataris, the Sega Genesis, SUN, and NeXT etc.
- It went from 8MHz on the original 68000 to 133Mhz on the 68060.
- Most of these had 24-bit or 32-bit addresses.

---

**M68k Assembly**

- The m68k has 7 data registers, D0-D7, used to store 32-bits.
- It has 7 address registers, A0-A7 used to store addresses in memory.
- 1 register, Program Counter (PC), stores the current instruction being executed.
- 1 register, the Condition Code Register, (CCR) holds 5 bits to hold conditions about the last instruction which executed.
  - X (Extend)
  - N (Negative)
  - Z (Zero)
  - V (Overflow)
  - C (Carry)
- 3 main types of Ints:
  - (S)hort = 8-bits
  - (W)ord = 16-bits
  - (L)ong = 32-bits
- Addressing modes:
- Register: D0, A0.
- Address indirect: (A0); post-inc (A0)+; pre-dec -(A0).
- Displacement: (16,A0) adds signed 16-bit.
- Indexed: (8,A0,D1.w) or (0,A0,D1.l) (base + index*1 + disp).

- Absolute: (label).w (16-bit PC-space), (label).l (full 32-bit).
- PC-relative: (16,PC), (8,PC,D1.w) (for PIC).
- Immediate: #123, #-5, #0xDEAD.

---

**M68k Assembly Continued**

- Labels are used to refer to variables or functions:

```
        .text
        .even
        .globl  _start
_start:
        move.q #0, %d0    ; return code in D0
        rts               ; return
```

- Jumps include:
  - `jsr <subroutine>`: (J)ump (T)o (S)ubroutine - calls a function.
  - `bsr <subroutine>`: (B)ranch (T)o (S)ubroutine - pushes the next instruction to the stack and calls a function. `rts` is only after a branch.
  - `bra <subroutine>`: (BR)anch (A)lways - jump forwards or backwards.
    - `bra.s`: (BR)anch (A)lways (S)hort - jump up to 126 bytes forwards/backwards.
    - `bra.w`: (BR)anch (A)lways (W)ord - jump up to 32766 bytes forwards/backwards.

Note that branch instructions are faster and smaller in memory, but limited in how far they can jump. `jsr` can jump anywhere.

- `move.(w/l) <src>, <dst>`: (Move) - move from src to dst.
- `movea.(w/l) <src>, AR`: (Move) (A)ddress - move src to address register.
- `cmp`: (C)o(mp)arison - sets flags after a comparison.
- `bcc`: lets you branch on a compare.
- `lea <src> <dst>`: (L)oad (E)ffective (A)ddress - Load a given address and place in dst.

```
lea    (4,A0), A1 ; add 4 to A0 and store address in A1
move.w (A1), D1 ; dereference A1 and put in D1
```

- Stack manipulation: stack grows downward:
  - Pushing: `move.w #18,-(sp)`
  - Popping: `addq.l #2,sp`

---

**Running an m68k Machine**

- There are two emulators I've used.
- Hatari
- Aranym
- These can run any kind of OS. I did not go with Linux (sorry) and ran AFROS, which is a TOS/Freemint compatible OS.


Freemint is a open source version of the MiNT operating system for Atari STs.

- Freemint's Github has the kernel freemint, two versions of libc (mintlib, libcmini), and other useful libs like gemlib

As well, there's a version of gcc with support for compiling to this target. Thorsten Otto's website has gcc compilers built to target the `m68k-atari-mint-gcc`. You can download these and start compiling C code to run on your emulator.

---

**What about Rust?**

- I tried to compile Rust to this target too: My custom target
- LLVM doesn't compile to this target well. There's some support for bare-metal (m68k-unknown-elf) but this doesn't give you any user space. It also outputs ELF files which don't run on our emulators out of the box anyway (they use a.out).
- Stack trace that shows the issue in LLVM's Instruction Selection:

```
/lib/librustc_driver-67dd58abd922ea00.so(+0x3c22f0f) [0x7faf1a822f0f]
/lib64/libc.so.6(+0x1a070) [0x7faf16a28070]
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(+0x763347d) [0x7faf1443347d]
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(+0x762d5fc) [0x7faf1442d5fc]
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(+0x7629bd2) [0x7faf14429bd2]
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(+0x7e9d192) [0x7faf14c9d192]
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(_ZN4llvm16SelectionDAGISel17CodeGenAndEmitDAGEv+0x8
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(_ZN4llvm16SelectionDAGISel20SelectAllBasicBlocksERK
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(_ZN4llvm16SelectionDAGISel20runOnMachineFunctionERN
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(+0x8b7a6c9) [0x7faf1597a6c9]
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(_ZN4llvm13FPPassManager13runOnFunctionERNS_8Functio
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(_ZN4llvm13FPPassManager11runOnModuleERNS_6ModuleE+0
/lib/libLLVM.so.21.1-rust-1.93.0-nightly(_ZN4llvm6legacy15PassManagerImpl3runERNS_6ModuleE+0
```

```
/lib/librustc_driver-67dd58abd922ea00.so(+0x64b31c5) [0x7faf1d0b31c5]
/lib/librustc_driver-67dd58abd922ea00.so(+0x64b2dfb) [0x7faf1d0b2dfb]
/lib/librustc_driver-67dd58abd922ea00.so(+0x64b4ff5) [0x7faf1d0b4ff5]
/lib/librustc_driver-67dd58abd922ea00.so(_RINvNtNtCsaEKiIOnD17M_3std3sys9backtrace28___rust_
/lib/librustc_driver-67dd58abd922ea00.so(+0x6253539) [0x7faf1ce53539]
/lib/librustc_driver-67dd58abd922ea00.so(+0x6258def) [0x7faf1ce58def]
/lib64/libc.so.6(+0x71f54) [0x7faf16a7ff54]
/lib64/libc.so.6(+0xf532c) [0x7faf16b0332c]
```

---

### Time to write C

- If you don't want to write any libc or anything, you can stop right here. Just use the C toolchain to compile to your emulator and you're good to go.
- I wanted to go further – I wanted to write my own version of `libc` to really understand the bits and bytes before main.

```c
// How does our binary get argc and argv?
int main(int argc, char* argv[]) {
    printf("hello world\n");
    return 0; // what happens here?
}
```

Does anyone know?

---

### start.ld

- First, we have to layout our binary.
- `.text` is where the instructions are for your program
- `.data` is where initialized global/static variables go. Don't edit them.
- `.bss` is where uninitialized global/static variables go.

```
OUTPUT_FORMAT(a.out-mintprg)
OUTPUT_ARCH(m68k)
ENTRY(_start)

SECTIONS {
  /* Start at 0 for PRG (BFD will add the PRG header). */
  . = 0;

  /* --- TEXT + RO --- */
  .text : ALIGN(2) {
    KEEP(*(.text._start))          /* ensure _start first */
    *(.text .text.*)
    *(.rodata .rodata.*)
```

```
    KEEP(*(.init))
    KEEP(*(.fini))
  }

  /* --- DATA --- */
  .data : ALIGN(2) {
    *(.data .data.*)
    *(.sdata .sdata.*)
    *(.init_array .init_array.*)
    *(.fini_array .fini_array.*)
  }

  /* --- BSS --- */
  .bss (NOLOAD) : ALIGN(2) {
    *(.bss .bss.*)
    *(COMMON)
  }

  /* Strip sections not used on TOS */
  /DISCARD/ : {
    *(.comment) *(.note*) *(.eh_frame*) *(.stab*) *(.debug*) *(.gnu.*)
  }
}
```

---

**crt0.S**

- C's runtime, the stuff that runs before main.

```
    .title "crt0.S for m68k-coff"
    #define STACKSIZE   0x4000
/*
 * Define an empty environment.
 */
    .data
    .align 2
SYM (environ):
    .long 0
    .align  2
    .text
/*
 * These symbols are defined in C code, so they need to always be
 * named with SYM because of the difference between object file formats.
 */
    .extern SYM (main)
    .extern SYM (exit)
```

```
    .extern SYM (hardware_init_hook)
    .extern SYM (software_init_hook)
    .extern SYM (atexit)
    .extern SYM(__do_global_dtors)
/*
* These values are set in the linker script, so they must be
* explicitly named here without SYM.
*/
    .extern __stack
    .extern __bss_start
    .extern _end
/*
* Set things up so the application will run. For historical reasons
* this is called 'start'.  We set things up to provide '_start'
* as with other systems, but also provide a weak alias called
* 'start' for compatibility with existing linker scripts.
*/
    .global SYM (start)
    .weak SYM (start)
    .set SYM (start),SYM(_start)
    .global SYM (_start)
```

---

**crt0.S continued**

```
SYM (_start):
    /* See if user supplied their own stack (__stack != 0).  If not, then
     * default to using the value of %sp as set by the ROM monitor.
     */
    movel   IMM(__stack), a0
    cmpl    IMM(0), a0
    jbeq    1f
    movel   a0, sp
1:
    /* set up initial stack frame */
    link    a6, IMM(-8)
/*
 * zero out the bss section.
 */
    movel   IMM(__bss_start), d1
    movel   IMM(_end), d0
    cmpl    d0, d1
    jbeq    3f
    movl    d1, a0
    subl    d1, d0
```

```
    subql   IMM(1), d0
2:
    clrb    (a0)+
    subql   IMM(1), d0
    jbpl    2b

 /*
 * call the main routine from the application to get it going.
 * main (argc, argv, environ)
 * we pass argv as a pointer to NULL.
 */
    movel   IMM (__FINI_SECTION__),(sp)
    PICCALL SYM (atexit)
    PICCALL __INIT_SECTION__
    pea     0
    PICPEA  SYM (environ),a0
    pea     sp@(4)
    pea     0
    PICCALL SYM (main)
    movel   d0, sp@-
```

---

### Now we're in main

That leaves one question: How does writing to the console work?

```
int main(int argc, char* argv[]) {
    if (argc > 1) {
        puts(argv[1]); // but how does this work?
    }
}
```

- We need something called a System Call.

---

### System Calls

- System calls ask the kernel to do something on the user's behalf.
- Operating systems provide 3 things: Virtualization, Concurrency, Persistence.
- We cannot have any of these if users can do what they want to.
- Thus there's a need for a contract between the user and the OS.

Freemint's System Call for Fwrite

We can call this special `trap #1` after loading our arguments onto the stack and the OS will write for us.

```
pea       buf           ; Offset 8
move.l    count,-(sp)   ; Offset 4
move.w    handle,-(sp)  ; Offset 2
move.w    #64,-(sp)     ; Offset 0
trap      #1            ; GEMDOS
lea       $C(sp),sp     ; Correct stack
```

Translated to C:

```c
/* Fwrite - Write bytes to a file handle. */
static inline int32_t Fwrite(int16_t handle, int32_t count, const void *buf) {
  return (int32_t)trap_1_wwll(0x40, handle, (long)count, (long)buf);
}
```

---

### System Calls Continued

```c
static inline long trap_1_wwll(short n, short a, long b, long c) {
  register long retval __asm__("d0");
  short _a = (short)(a);
  long _b = (long)(b);
  long _c = (long)(c);

  __asm__ volatile("movl    %4,%%sp@-\n\t"
                   "movl    %3,%%sp@-\n\t"
                   "movw    %2,%%sp@-\n\t"
                   "movw    %1,%%sp@-\n\t"
                   "trap    #1\n\t"
                   "lea %%sp@(12),%%sp"
                   : "=r"(retval)                        /* outputs */
                   : "g"(n), "r"(_a), "r"(_b), "r"(_c)  /* inputs  */
                   : "d1", "d2", "a0", "a1", "a2", "cc", "memory" /* clobbered */
                  );
  return retval;
}
```

### Implementing `fwrite`

This code:

```c
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) {
  if (size == 0 || nmemb == 0)
    return 0;

  size_t total_bytes = size * nmemb;
  long ret = Fwrite(stream->handle, total_bytes, ptr);
  if (ret <= 0)
```

```
    return 0;

  return ret / size;
}
```

Turns into this:

```
$ m68k-atari-mint-objdump -dr build/objs/stdio/fwrite.o

build/objs/stdio/fwrite.o:     file format a.out-zero-big

Disassembly of section .text:

00000000 <_fwrite>:
   0:   48e7 3c20       moveml %d2-%d5/%a2,%sp@-
   4:   262f 001c       movel %sp@(28),%d3
   8:   202f 0020       movel %sp@(32),%d0
   c:   4a83            tstl %d3
   e:   6608            bnes 18 <.L2>

00000018 <.L2>:
  18:   4a80            tstl %d0
  1a:   67f4            beqs 10 <.L4>
  1c:   206f 0024       moveal %sp@(36),%a0
  20:   2810            movel %a0@,%d4
  22:   2f00            movel %d0,%sp@-
  24:   2f03            movel %d3,%sp@-
  26:   4eb9 0000 0000  jsr 0 <_fwrite>
  28:   32              ___mulsi3          # what the
  2c:   508f            addql #8,%sp
  2e:   2a2f 0018       movel %sp@(24),%d5 # Here's the syscall
  32:   2f05            movel %d5,%sp@-    # We load the buffer
  34:   2f00            movel %d0,%sp@-    # Then the length
  36:   3f04            movew %d4,%sp@-    # then the handle
  38:   3f3c 0040       movew #64,%sp@-    # Then #64, syscall for fwrite
  3c:   4e41            trap #1            # trap
  3e:   4fef 000c       lea %sp@(12),%sp   # correct the stack
  ...
```

---

**What is `__mulsi3`?**

```
$ m68k-atari-mint-objdump -D /usr/lib64/gcc/m68k-atari-mint/12/libgcc.a | rg __mulsi3 -A 20
00000000 <___mulsi3>:
   0:   302f 0004       movew %sp@(4),%d0
   4:   c0ef 000a       muluw %sp@(10),%d0
```

```
 8:   322f 0006       movew %sp@(6),%d1
 c:   c2ef 0008       muluw %sp@(8),%d1
10:   d041            addw %d1,%d0
12:   4840            swap %d0
14:   4240            clrw %d0
16:   322f 0006       movew %sp@(6),%d1
1a:   c2ef 000a       muluw %sp@(10),%d1
1e:   d081            addl %d1,%d0
20:   4e75            rts
```

- The m68k doesn't have hardware instructions for multiplying two 32-bit numbers (`mul.l` doesn't exist) on the original m68k – it was added to the 68020.
- Since I'm targeting the 68k, it's added in for me.

---

### Let's implement some calls

How would you implement `memcpy`, which copies `n` bytes from src to dest?

```
void *memcpy(void *restrict dest, const void *restrict src, size_t n);
```

---

### A simple implementation

```
void *memcpy(void *restrict dest, const void *restrict src, size_t n) {
    for (; n; n--) *d++ = *s++;
    return dest;
}
```

---

### This is Musl's version

```
void *memcpy(void *restrict dest, const void *restrict src, size_t n)
{
    unsigned char *d = dest;
    const unsigned char *s = src;

#ifdef __GNUC__

#if __BYTE_ORDER == __LITTLE_ENDIAN
#define LS >>
#define RS <<
#else
#define LS <<
#define RS >>
```

```
#endif

    typedef uint32_t __attribute__((__may_alias__)) u32;
    uint32_t w, x;

    for (; (uintptr_t)s % 4 && n; n--) *d++ = *s++;

    if ((uintptr_t)d % 4 == 0) {
        for (; n>=16; s+=16, d+=16, n-=16) {
            *(u32 *)(d+0) = *(u32 *)(s+0);
            *(u32 *)(d+4) = *(u32 *)(s+4);
            *(u32 *)(d+8) = *(u32 *)(s+8);
            *(u32 *)(d+12) = *(u32 *)(s+12);
        }
        if (n&8) {
            *(u32 *)(d+0) = *(u32 *)(s+0);
            *(u32 *)(d+4) = *(u32 *)(s+4);
            d += 8; s += 8;
        }
        if (n&4) {
            *(u32 *)(d+0) = *(u32 *)(s+0);
            d += 4; s += 4;
        }
        if (n&2) {
            *d++ = *s++; *d++ = *s++;
        }
        if (n&1) {
            *d = *s;
        }
        return dest;
    }
```

---

**Continued**

```
    if (n >= 32) switch ((uintptr_t)d % 4) {
    case 1:
        w = *(u32 *)s;
        *d++ = *s++;
        *d++ = *s++;
        *d++ = *s++;
        n -= 3;
        for (; n>=17; s+=16, d+=16, n-=16) {
            x = *(u32 *)(s+1);
            *(u32 *)(d+0) = (w LS 24) | (x RS 8);
```

```
        w = *(u32 *)(s+5);
        *(u32 *)(d+4) = (x LS 24) | (w RS 8);
        x = *(u32 *)(s+9);
        *(u32 *)(d+8) = (w LS 24) | (x RS 8);
        w = *(u32 *)(s+13);
        *(u32 *)(d+12) = (x LS 24) | (w RS 8);
    }
    break;
case 2:
    w = *(u32 *)s;
    *d++ = *s++;
    *d++ = *s++;
    n -= 2;
    for (; n>=18; s+=16, d+=16, n-=16) {
        x = *(u32 *)(s+2);
        *(u32 *)(d+0) = (w LS 16) | (x RS 16);
        w = *(u32 *)(s+6);
        *(u32 *)(d+4) = (x LS 16) | (w RS 16);
        x = *(u32 *)(s+10);
        *(u32 *)(d+8) = (w LS 16) | (x RS 16);
        w = *(u32 *)(s+14);
        *(u32 *)(d+12) = (x LS 16) | (w RS 16);
    }
    break;
case 3:
    w = *(u32 *)s;
    *d++ = *s++;
    n -= 1;
    for (; n>=19; s+=16, d+=16, n-=16) {
        x = *(u32 *)(s+3);
        *(u32 *)(d+0) = (w LS 8) | (x RS 24);
        w = *(u32 *)(s+7);
        *(u32 *)(d+4) = (x LS 8) | (w RS 24);
        x = *(u32 *)(s+11);
        *(u32 *)(d+8) = (w LS 8) | (x RS 24);
        w = *(u32 *)(s+15);
        *(u32 *)(d+12) = (x LS 8) | (w RS 24);
    }
    break;
}
if (n&16) {
    *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
}
```

```
    if (n&8) {
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    }
    if (n&4) {
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    }
    if (n&2) {
        *d++ = *s++; *d++ = *s++;
    }
    if (n&1) {
        *d = *s;
    }
    return dest;
#endif

    for (; n; n--) *d++ = *s++;
    return dest;
}
```

---

**What did we learn?**

- Maybe Intel shouldn't have won
- Retrocomputing can be fun
- Don't write your own libc, it'll take me months if I continue on