Rabbit VM Implementation

Maxwell Bernstein

March 29, 2016

Contents

1 VM structure **1**

2 Registers **3**

3 Instructions **3**
3.1 Decoding 3
3.2 Modes 4
3.3 Following mode rules 4

4 VM setup **4**
4.1 Check CLI arguments 4
4.2 Open code file 4
4.3 Get size of code file 5
4.4 Allocate memory 5
4.5 Read code into memory 5
4.6 Fetch-decode-execute loop 5

```
/* Exit/error codes and statuses. */
typedef enum rabbits {
    RB_SUCCESS = 0, RB_FAIL, RB_OVERFLOW, RB_ILLEGAL,
} rabbits;
```

1 VM structure

```
/* Most instructions have a destination address and two operands. This struct
 * holds those. */
struct abc_s {
    rabbitw *dst, b, c;
```

```
};

/* Using the options/modes given by the user, fetch the destination address and
 * operands. */
static struct abc_s getabc(rabbitw *regs, rabbitw *mem, struct unpacked_s i) {
    rabbitw *dst = i.modes.rega_deref ? &mem[regs[i.rega]] : &regs[i.rega];
    rabbitw bval = i.modes.regb_deref ? mem[regs[i.regb]] : regs[i.regb];
    rabbitw cval = i.modes.immediate ? fetch_immediate() : regs[i.regc];
    cval = i.modes.regc_deref ? mem[cval] : cval;
    return (struct abc_s) { .dst = dst, .b = bval, .c = cval };
}

#define fetch_immediate() mem[regs[RB_IP]++]

#include <stdio.h>
#include <inttypes.h>
#include <stdlib.h>
#include <sys/stat.h>

typedef uint32_t rabbitw;

<<declarations>>

int main(int argc, char **argv) {
    <<check_cli_arguments>>

    <<open_code_file>>

    <<stat_code_file>>

    <<allocate_memory>>

    <<read_code_into_memory>>

    <<fetch_decode_execute_loop>>

    return 0;
}

#undef fetch_immediate
```

2 Registers

```
/* Registers available for use. */
typedef enum rabbitr {
    RB_ZERO = 0, RB_R1, RB_R2, RB_R3, RB_R4, RB_R5, RB_R6, RB_R7, RB_R8, RB_R9,
    RB_IP, RB_SP, RB_RET, RB_TMP, RB_FLAGS, RB_NUMREGS,
} rabbitr;
```

3 Instructions

```
/* Options for operand use. Is space C an immediate value or a register? Should
 * any of spaces A, B, and C be dereferenced? */
struct modes_s {
    uint8_t immediate : 1;
    uint8_t regc_deref : 1;
    uint8_t regb_deref : 1;
    uint8_t rega_deref : 1;
};

/* Holds an unpacked representation of an instruction (sans immediate value, if
 * it has one). */
struct unpacked_s {
    struct modes_s modes;
    uint8_t opcode : 4;
    uint8_t regc : 4;
    uint8_t regb : 4;
    uint8_t rega : 4;
};
```

3.1 Decoding

```
/* Transform the packed representation of an instruction into the unpacked
 * representation. */
struct unpacked_s decode(rabbitw instr) {
    /* Fetch the space modes from the instruction. */
    uint8_t modes = (instr >> 24) & 0xF;

    /* Offsets of the space modes in the mode nibble. */
    static const unsigned char RB_ADDRA = 1U << 0,
RB_ADDRB = 1U << 1,
```

```
RB_ADDRC = 1U << 2,
RB_IMMED = 1U << 3;

    return (struct unpacked_s) {
.modes = {
    .immediate = modes & RB_IMMED,
    .regc_deref = modes & RB_ADDRC,
    .regb_deref = modes & RB_ADDRB,
    .rega_deref = modes & RB_ADDRA,
},
.opcode = instr >> 28,
.regc = (instr >> 8) & 0xF,
.regb = (instr >> 4) & 0xF,
.rega = instr & 0xF,
    };
}
```

3.2 Modes

3.3 Following mode rules

4 VM setup

4.1 Check CLI arguments

```
if (argc != 2) {
    fprintf(stderr, "Need to pass file to execute.\n");
    return RB_FAIL;
}
```

4.2 Open code file

```
char *fn = argv[1];
FILE *fp = fopen(fn, "rb");
if (fp == NULL) {
    fprintf(stderr, "Can't open `%s'.\n", fn);
    return RB_FAIL;
}
```

4.3 Get size of code file

```
/* Get the size of the file so that we can allocate memory for it. */
struct stat st;
if (fstat(fileno(fp), &st) != 0) {
    fprintf(stderr, "Can't stat `%s'.\n", fn);
    fclose(fp);
    return RB_FAIL;
}
```

4.4 Allocate memory

```
/* Allocate memory with program first, stack second. */
size_t stacksize = 1000;
off_t size = st.st_size;
rabbitw *mem = malloc(stacksize + size * sizeof *mem);
if (mem == NULL) {
    fprintf(stderr, "Not enough memory. Could not allocate stack of size"
    "%zu + program of size %lld.\n", stacksize, size);
    return RB_FAIL;
}
```

4.5 Read code into memory

```
/* Read the file into memory. */
size_t i = 0;
rabbitw word;
while (fread(&word, sizeof word, 1, fp) != 0) {
    mem[i++] = word;
}
fclose(fp);

/* Instructions available for use. */
typedef enum rabbiti {
    RB_HALT = 0, RB_MOVE, RB_ADD, RB_SUB, RB_MUL, RB_DIV, RB_SHR, RB_SHL,
    RB_NAND, RB_XOR, RB_BR, RB_BRZ, RB_BRNZ, RB_IN, RB_OUT, RB_NUMINSTRS,
} rabbiti;
```

4.6 Fetch-decode-execute loop

```
struct abc_s abc;
```

```
rabbitw regs[RB_NUMINSTRS] = { 0 };
regs[RB_SP] = i;

/* Main fetch-decode-execute loop. */
while (1) {
    /* Fetch the current instruction word. */
    rabbitw word = mem[regs[RB_IP]++];

    /* Decode it. */
    struct unpacked_s i = decode(word);

    /* Execute it. */
    switch (i.opcode) {
    case RB_HALT:
free(mem);
return RB_SUCCESS;
break;
    case RB_MOVE: {
/* Move is special because it has one source instead of two
 * operands. */
rabbitw src = i.modes.immediate ? fetch_immediate() : regs[i.regc];
rabbitw *dst = i.modes.regb_deref ? &mem[regs[i.regb]] : &regs[i.regb];
*dst = i.modes.regc_deref ? mem[src] : src;
break;
    }
    case RB_ADD:
abc = getabc(regs, mem, i);
*abc.dst = abc.b + abc.c;
break;
    case RB_SUB:
abc = getabc(regs, mem, i);
rabbitw res = abc.b - abc.c;
*abc.dst = res;
if (res == 0) {
    /* Set zero flag. */
    regs[RB_FLAGS] |= 0x2U;
}
break;
    case RB_MUL:
abc = getabc(regs, mem, i);
```

```
*abc.dst = abc.b * abc.c;
break;
    case RB_DIV:
abc = getabc(regs, mem, i);
if (abc.c == 0) {
    free(mem);
    return RB_ILLEGAL;
}

*abc.dst = abc.b / abc.c;
break;
    case RB_SHR:
abc = getabc(regs, mem, i);
*abc.dst = abc.b >> abc.c;
break;
    case RB_SHL:
abc = getabc(regs, mem, i);
*abc.dst = abc.b << abc.c;
break;
    case RB_NAND:
abc = getabc(regs, mem, i);
*abc.dst = ~(abc.b & abc.c);
break;
    case RB_XOR:
abc = getabc(regs, mem, i);
*abc.dst = abc.b ^ abc.c;
break;
    case RB_BR: {
/* Branch is special because it only has one argument. */
rabbitw src = i.modes.immediate ? fetch_immediate() : regs[i.regc];
regs[RB_IP] = i.modes.regc_deref ? mem[src] : src;
break;
    }
    case RB_BRZ:
/* Branch if zero is special because it only has one argument. */
if ((regs[RB_FLAGS] & 0x2U) == 0) {
    rabbitw src = i.modes.immediate ? fetch_immediate() : regs[i.regc];
    regs[RB_IP] = i.modes.regc_deref ? mem[src] : src;
}
break;
```

```
    case RB_BRNZ:
/* Branch not zero is special because it only has one argument. */
if ((regs[RB_FLAGS] & 0x2U) != 0) {
    rabbitw src = i.modes.immediate ? fetch_immediate() : regs[i.regc];
    regs[RB_IP] = i.modes.regc_deref ? mem[src] : src;
}
break;
    case RB_IN: {
/* Input is special because it does not have an argument. */
rabbitw dst = i.modes.immediate ? fetch_immediate() : regs[i.regc];
rabbitw *dstp = i.modes.regc_deref ? &mem[regs[i.regc]] : &regs[i.regc];
*dstp = getchar();
break;
    }
    case RB_OUT: {
/* Output is special because it has one argument and no
 * destination. */
rabbitw src = i.modes.immediate ? fetch_immediate() : regs[i.regc];
src = i.modes.regc_deref ? mem[src] : src;
putchar(src);
break;
    }
    default:
free(mem);
return RB_ILLEGAL;
break;
    }
}
```