Rabbit VM Implementation

Maxwell Bernstein

May 25, 2016

Our emulator can exit or abort with a couple of status codes. They are as follows:

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

`RB_SUCCESS` indicates no error. `RB_FAIL` indicates a generic failure. `RB_OVERFLOW` indicates a stack overflow. `RB_ILLEGAL` indicates an illegal instruction.

Emulator structure

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

#include "rabbit_types.h"
#include "rabbit_io.h"
#include "rabbit_codewords.h"

{{{declarations}}}

{{{builtin_functions}}}

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;
}

{{{macro_cleanup}}}
```

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;
```

Instructions

There are four types of instructions:

```
instr %rA, %rB, %rC
instr %rA, %rB
instr %rA
instr
```

There is only one instruction in the last category: `halt`.

All instructions are represented as 32-bit words:

```
#ifndef RABBIT_TYPES_H
#define RABBIT_TYPES_H
```

```
#include <inttypes.h>

typedef uint32_t rabbitw;

#endif
```

These 32 bits are laid out as follows:

```
     +-----Immediate bit
     |+----Addressing mode bit C
     ||+---Addressing mode bit B
     |||+--Addressing mode bit A
     |||| +Dead space+     regB
     vvvv vvvvvvvvvvv     vvvv
IIII MDDD 000000000000 CCCCBBBBAAAA
^^^^                   ^^^^    ^^^^
Opcode                 regC    regA

VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Immediate value
```

Note that every bit in "Dead space" must be turned off. If one is turned on, the result of executing that instruction is undefined.

Internally, the emulator uses structs with bit fields to represent instructions. It uses bit fields only to save space; they can be removed with no consequences.

```
/* 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;
};
```

The full unpacked representation of the instruction includes this unpacked modes struct along with the opcode and three operands.

```
/* 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;
};
```

The immediate value follows the instruction in memory; the emulator fetches it upon execution.

Decoding

With the knowledge of the instruction format and its in-emulator representation, it is not difficult to write a `decode` function that takes an instruction and returns the unpacked representation.

First, `decode` fetches the mode from the instruction. Its least significant bit is 24 and it contains at most four bits.

```
/* Fetch the space modes from the instruction. */
uint8_t modes = (instr >> 24) & 0xF;
```

As the modes are stored in a bit vector, it's useful to have masks for each of the addressing modes and the immediate mode.

```
/* Offsets of the space modes in the mode nibble. */
static const uint8_t
    RB_ADDRA_LSB = 0,
    RB_ADDRB_LSB = 1,
    RB_ADDRC_LSB = 2,
    RB_IMMED_LSB = 3;
```

From here it's reasonably trivial to fill the `unpacked_s` struct. `instr` needs to be shifted and masked a couple of times to fetch the opcode and registers, but nothing major.

```
#include "rabbit_codewords.h"

/* Transform the packed representation of an instruction into the unpacked
```

```
 * representation. */
struct unpacked_s decode(rabbitw instr) {
    {{{fetch_modes}}}

    {{{mode_offsets}}}

    return (struct unpacked_s) {
.modes = {
    .immediate  = (modes >> RB_IMMED_LSB) & 0x1,
    .regc_deref = (modes >> RB_ADDRC_LSB) & 0x1,
    .regb_deref = (modes >> RB_ADDRB_LSB) & 0x1,
    .rega_deref = (modes >> RB_ADDRA_LSB) & 0x1,
},
.opcode = instr >> 28,
.regc =  (instr >> 8) & 0xF,
.regb =  (instr >> 4) & 0xF,
.rega = instr & 0xF,
    };
}
```

Modes

Following mode rules

VM setup

Check CLI arguments

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

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;
}
```

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;
}
```

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;
}
```

Read code into memory

```
/* Read the file into memory. */
size_t i = 0;
rabbitw word = 0;
/* We cannot use fread because of endian-ness issues. */
while (read_word(fp, &word) != 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_BIF,
    RB_NUMINSTRS,
} rabbiti;
```

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;
    }
    case RB_BIF: {
rabbitw src = i.modes.immediate ? fetch_immediate() : regs[i.regc];
src = i.modes.regc_deref ? mem[src] : src;
if (src > NUM_BIFS) {
    fprintf(stderr, "Invalid bif: ‘%u’.\n", src);
    return RB_FAIL;
}
bif f = biftable[src].f;
f(regs, mem);
break;
    }
/*    case RB_CFF:
break;
*/
    default:
```

```
free(mem);
return RB_ILLEGAL;
break;
    }
}
```

Building the VM

```
CC=gcc
CFLAGS=-Wall -Wextra -Wpedantic
LFLAGS=
RABBIT_OBJS=rabbit.o rabbit_io.o rabbit_codewords.o
ASSEMBLER_OBJS=assembler.o rabbit_io.o
DISASSEMBLER_OBJS=disassembler.o rabbit_io.o rabbit_codewords.o

all: rabbit assembler disassembler

clean:
rm -f rabbit rabbit-asm rabbit-dis
rm -f $(RABBIT_OBJS) $(ASSEMBLER_OBJS) $(DISASSEMBLER_OBJS)

rabbit: $(RABBIT_OBJS)
$(CC) $(CFLAGS) $(LFLAGS) $(RABBIT_OBJS) -o rabbit

assembler: $(ASSEMBLER_OBJS)
$(CC) $(CFLAGS) $(LFLAGS) $(ASSEMBLER_OBJS) -o rabbit-asm

disassembler: $(DISASSEMBLER_OBJS)
$(CC) $(CFLAGS) $(LFLAGS) $(DISASSEMBLER_OBJS) -o rabbit-dis

%.o: %.c
$(CC) $(CFLAGS) -o $@ -c $<
```