

# Rabbit VM RFC

Maxwell Bernstein

May 16, 2015

## Contents

<b>1</b>	<b>Why?</b>	<b>1</b>
<b>2</b>	<b>What?</b>	<b>2</b>
2.1	Definitions . . . . .	2
2.2	Registers . . . . .	2
2.2.1	Flags . . . . .	2
2.3	Instruction set . . . . .	2
2.3.1	Real instructions . . . . .	2
2.3.2	Assembler macros . . . . .	3
2.4	Addressing modes . . . . .	3
<b>3</b>	<b>How?</b>	<b>4</b>
3.1	Instruction formats . . . . .	4
3.2	Stages of compilation . . . . .	5
3.2.1	<b>TODO</b> Preprocessing . . . . .	5
3.2.2	<b>TODO</b> Peephole optimization . . . . .	5
3.2.3	<b>TODO</b> Assembling . . . . .	5
3.3	C interface . . . . .	5
3.4	<b>TODO</b> . . . . .	7
3.4.1	Floating point . . . . .	7
3.4.2	Memory layout . . . . .	7

## 1 Why?

I would like to make a RISC architecture that is capable of comfortably sitting on top of nearly any other architecture. If it's possible to compile to

Rabbit, then a program can run on more or less any hardware or virtualized architecture.

Rabbit can be optimized per-architecture, while maintaining the same interface. It may, for example, take advantage of Intel's SIMD behind the scenes.

## 2 What?

### 2.1 Definitions

**space:** A register or memory location.

### 2.2 Registers

Registers are 32 bits wide.

<i>Value</i>	<i>Register</i>	<i>Use</i>
0x0	<b>zero</b>	Contains 0. MIPS style.
0x1 .. 0x9	<b>r1 .. r9</b>	General purpose.
0xA	<b>ip</b>	Instruction pointer.
0xB	<b>sp</b>	Stack pointer.
0xC	<b>ret</b>	Returned value.
0xD	<b>tmp</b>	Temporary register.
0xE	<b>flags</b>	Flags used for comparison.

#### 2.2.1 Flags

<i>Bit</i>	<i>Flag</i>	<i>Meaning</i>
0x0	<b>SF</b>	Sign flag. On if sign bit of result is on.
0x1	<b>ZF</b>	Zero flag. On if result is zero or numbers were the same.
0x2 .. 0x20		Reserved.

### 2.3 Instruction set

#### 2.3.1 Real instructions

When it makes sense, the destination register is the first argument to an instruction. The last argument to the following instructions may also be an immediate value, denoted with a prefix of **\$**: **move**, **add**, **sub**, **mul**, **div**, **shr**, **shl**, **nand**, **xor**, **br**, **brz**, **brnz**.

<i>Value</i>	<i>Instruction</i>	<i>Usage</i>	<i>Explanation</i>	<i>Description</i>
0x0	halt	halt		Stop the execution of the program.
0x1	move	move %rB, %rC	r[B] := r[C]	Move one space into another.
0x2	add	add %rA, %rB, %rC	r[A] := r[B] + r[C]	Add two spaces into a third.
0x3	sub	sub %rA, %rB, %rC	r[A] := r[B] - r[C]	Subtract two spaces into a third.
0x4	mul	mul %rA, %rB, %rC	r[A] := r[B] * r[C]	Multiply two spaces into a third.
0x5	div	div %rA, %rB, %rC	r[A] := r[B] / r[C]	Divide two spaces into a third.
0x6	shr	shr %rA, %rB, %rC	r[A] := r[B] >> r[C]	Shift right one space a number of times.
0x7	shl	shl %rA, %rB, %rC	r[A] := r[B] << r[C]	Shift left one space a number of times.
0x8	nand	nand %rA, %rB, %rC	r[A] := not(r[B] & r[C])	NAND two spaces.
0x9	xor	xor %rA, %rB, %rC	r[A] := r[B] ^ r[C]	XOR two spaces.
0xA	br	br %rC	goto r[C]	Branch.
0xB	brz	brz %rC	if (ZF set) goto r[C]	Branch if ZF is set.
0xC	brnz	brnz %rC	if (!ZF set) goto r[C]	Branch if ZF is not set.
0xD	in	in %rC	r[C] := getchar()	Read one character from the keyboard.
0xE	out	out %rC	putchar(r[C])	Print one character from the register.

### 2.3.2 Assembler macros

The last argument to the following macros may also be an immediate value, denoted with a prefix of \$: cmp, not, push, call.

<i>Macro</i>	<i>Usage</i>	<i>Expansion</i>
cmp	cmp A, B	sub %tmp, A, B
not	not A, B	nand A, B, B
or	or A, B, C	(A nand A) nand (B nand B)
and	and A, B, C	nand A, B, C // not A, A
push	push A	move (%sp), A // sub %sp, %sp, \$1
pop	pop A	add %sp, %sp, \$1 // move A, (%sp)
call	call A	push %ip // br A
ret	ret	pop %ip

### 2.4 Addressing modes

There are two addressing modes: %reg and (%reg). The former uses the value in the register, and the latter uses the word at the address in the register.

### 3.1 Instruction formats

instr %rA

in **r1**, then stores the result in **r1**. So this instruction would be an increment instruction.

The addressing mode bits are simple; if a register's addressing mode bit is on, then the address in the register is dereferenced when the instruction is being executed, and that data is used instead. For example:

```
1: 0010 0 100 000000000000 0111 0001 0000
```

Performs an addition operation that adds the contents of **zero** with **r1** and stores the result in memory at the address in **r7**.

## 3.2 Stages of compilation

### 3.2.1 TODO Preprocessing

The preprocessor will be responsible for macro expansion and label to address translation. Macros exist in the form of instruction expansions, done behind the scenes.

### 3.2.2 TODO Peephole optimization

### 3.2.3 TODO Assembling

## 3.3 C interface

```
#ifndef RABBIT_H
#define RABBIT_H

struct rabbit_s;
typedef struct rabbit_s *rabbit_t;
typedef uint32_t rabbit_word;

typedef enum {
    RB_SUCCESS,
    RB_FAIL,
    RB_OVERFLOW,
    RB_ILLEGAL,
} rabbit_status;

enum {
    RB_ZERO,
    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,
} rabbit_reg;

enum {
    RB_HALT,
    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,
} rabbit_instr;

const unsigned char RB_ADDRA = 1U << 0,
    RB_ADDRB = 1U << 1,
    RB_ADDRD = 1U << 2,
    RB_IMMED = 1U << 3;

rabbit_t rabbit_new (rabbit_word mem_size);
void rabbit_free (rabbit_t *r);

```

```

rabbit_word rabbit_pack_reg (unsigned char opcode,
    unsigned char modes,
    unsigned char regc,
    unsigned char regb,
    unsigned char rega);

rabbit_word rabbit_pack_imm (unsigned char opcode,
    unsigned char modes,
    unsigned char regc,
    unsigned char regb);

void rabbit_load_code (rabbit_t r, rabbit_word *code, rabbit_word len);
rabbit_status rabbit_run (rabbit_t r);

#endif

```

## 3.4 TODO

### 3.4.1 Floating point

Floating point computation is left to the client (an exercise for the reader, if you will).

### 3.4.2 Memory layout

The memory layout is completely flat right now.