Rabbit VM

Maxwell Bernstein

May 16, 2015

Contents

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 will 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.

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

2.2.1 Flags

Bit	*Flag*	*Meaning*
`0x0`	`SF`	Sign flag. On if sign bit of result is on.
`0x1`	`ZF`	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`.

Value	*Instruction*	*Usage*	*Explanation*	*Description*
0x0	halt	halt		Stop the execution of th
0x1	move	move %r1, %r2	r[2] := r[1]	Move one space into anc
0x2	add	add %r1, %r2, %r3	r[1] := r[2] + r[3]	Add two spaces into a th
0x3	sub	sub %r1, %r2, %r3	r[1] := r[2] - r[3]	Subtract two spaces into
0x4	mul	mul %r1, %r2, %r3	r[1] := r[2] * r[3]	Multiply two spaces into
0x5	div	div %r1, %r2, %r3	r[1] := r[2] / r[3]	Divide two spaces into a
0x6	shr	shr %r1, %r2, %r3	r[1] := r[2] >> r[3]	Shift right one space a n
0x7	shl	shl %r1, %r2, %r3	r[1] := r[2] << r[3]	Shift left one space a nu
0x8	nand	nand %r1, %r2, %r3	r[1] := not(r[2] & r[3])	NAND two spaces.
0x9	xor	xor %r1, %r2, %r3	r[1] := r[2] ^ r[3]	XOR two spaces.
0xA	br	br %r1	goto r[1]	
0xB	brz	brz %r1	if (ZF set) goto r[1]	Branch if ZF is set.
0xC	brnz	brnz %r1	if (!(ZF set)) goto r[1]	Branch if ZF is not set.
0xD	in	in %r1	r[1] := getchar()	Read one character from
0xE	out	out %r1	putchar(r[1])	Print one character from

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`.

Macro	*Usage*	*Expansion*
cmp	cmp %r1, %r2	sub %tmp, %r1, %r2
not	not %r1, %r2	nand %r1, %r2, %r2
or	or %r1, %r2, %r3	(A nand A) nand (B nand B)
and	and %r1, %r2, %r3	(A nand B) nand (A nand B)
push	push %r1	move (%sp), $1, sub %sp, %sp, $1
pop	pop %r1	add %sp, %sp, $1, move %r1, (%sp)
call	call %r1	push %ip, br %r1
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 byte TODO

2.4.1 TODO FIGURE OUT IF DEREFERENCE IS BYTE OR WORD

3 How?

3.1 Instruction formats

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

```
    +--Immediate?
    | +--Addressing mode bit C
    | |+--Addressing mode bit B
    | ||+--Addressing mode bit A
    | ||| +Dead space+    regB
    v vvv vvvvvvvvvvv    vvvv
IIIIM DDD 00000000000CCCCBBBBAAAA
^^^^                 ^^^^    ^^^^
Opcode               regC    regA

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

If the immediate bit is on, then the instruction disregards `rC` and instead looks for its third argument in the 32 bits after the first instruction. For example:

```
1: 0001 1 000 000000000000 0001 0000 0000
2: 0000 0 000 000000000000 0000 0000 0111
```

represents a `move` instruction with the immediate bit set. It will therefore look for an immediate value in the following word (in this case, the value is 7), and then store it in `r1`.

Addition works in a similar fashion:

```
1: 0010 1 000 000000000000 0001 0001 0000
2: 0000 0 000 000000000000 0000 0000 0001
```

represents an `add` instruction with the immediate bit set. It looks for an immediate value in the following word (in this case, 1), adds it to the value

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 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 Peephole optimization

Ensure that, with macro expansion, we are not setting a register repeatedly. For example:

```
push %r1
```

expands to

```
move (%sp), %r1
add %sp, %sp, $1
```

. If we had multiple pushes, it's quite possible that we would have multiple unnecessary `loadi` instructions, and not touch `r1` otherwise. Peephole optimization notices this and removes it.

3.2.3 Assembling

3.3 TODO

3.3.1 Floating point

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

3.3.2 Memory layout

The memory layout is completely flat right now.