RUNNING LINUX IN THE WEB BROWSER

- How hard can it be? -

A project made in Canada by Sebastian Macke

QATalk Sep. 23th 2016
THE WEBSITE: JOR1K.COM

jor1k: OpenRISC OR1K Javascript Emulator Running Linux With Network Support

www.github.com/s-macke/jor1k
INITIAL MOTIVATION

- JavaScript is the language of the Web
  - First true language which basically runs everywhere.
  - It is available immediately (one click)
- Writing an emulator is a fun way to learn a new language
JOR1K TIMELINE

2013

First line Of code

Publication on Github

2014

Run X-Window system

Network

Publication on my website and in IRC chat

Porting Wayland

2015

First talk at ORConf

New Filesystem

Network "parallel" computing

Audio

2016

Google Summer Of Code Mentor porting RISC-V architecture

More than a dozen news sites reports. (Slashdot, Reddit, Phoronix, Hacker News)

C Development Website
Let's start here
WHAT IS NECESSARY TO SIMULATE AN ARCHITECTURE TO BOOT LINUX?

Goal: take an easy architecture
WHICH ARCHITECTURE?

X86 architecture software developer manual
  • 4618 pages
  • More than 500 instructions

ARMv8 Architecture Reference Manual:
  • 5740 pages
  • contains 3125 times the word „unpredictable“
  • contains 2290 times the word „undefined“

For comparison: MS Office file format spec ~6000 pages
Lines of code for the implementation of CPU architectures in the Linux kernel

358 pages specification
SIMULATE ON THE CIRCUIT LEVEL

- ARM 1
- 25,000 transistors
- Runs at 20Hz
- 45,000 lines of code

(Recorded from visual6502.org)
SIMULATE ON THE MACHINE CODE LEVEL

International Obscured C Code contest entry of 2013 from Adrian Cable

SIMULATE ON THE MACHINE CODE LEVEL

Intel 8086/186 CPU (29000 transistors)

SIMULATE ON THE MACHINE CODE LEVEL

1MB RAM

SIMULATE ON THE MACHINE CODE LEVEL

8072A 3.5" floppy disk controller (1.44MB/720KB)

SIMULATE ON THE MACHINE CODE LEVEL

Fixed disk controller (supports a single hard drive up to 528MB)

SIMULATE ON THE MACHINE CODE LEVEL

Hercules graphics card with 720x348 2-color graphics (64KB video RAM), and CGA 80x25 16-color text mode support

SIMULATE ON THE MACHINE CODE LEVEL

8253 programmable interval timer (PIT)

SIMULATE ON THE MACHINE CODE LEVEL

8259 programmable interrupt controller (PIC)

SIMULATE ON THE MACHINE CODE LEVEL

8042 keyboard controller with 83-key XT-style keyboard

SIMULATE ON THE MACHINE CODE LEVEL

MC146818 real-time clock

SIMULATE ON THE MACHINE CODE LEVEL

PC speaker

SIMULATE ON THE MACHINE CODE LEVEL

International Obscured C Code contest entry of 2013 from Adrian Cable

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL

SIMULATE ON THE MACHINE CODE LEVEL
1. CPU (OPENRISC)

- 32-Bit address and data bus
- 32 registers
- Instruction size of 4 byte (reduced instruction set)
- arithmetic and logical operations
  - add, sub, mul, div, or, and, xor, shift left, shift right
  - three operand instructions for signed and unsigned integers
- Load-Store (word, half-word, byte, signed and unsigned load)
- Conditional and unconditional branching (absolute and relative)
- Comparison (signed and unsigned)
- Two modes of operation: supervisor and user
- Exceptions (timer interrupts, switch of modes, bus error, reset, div by 0)
- Special purpose registers (Status and Control, Timer, PIC, MMU, Debug, Power Management)
- optional floating point
1. A CPU (OPENRISC)

Example: add with immediate Half Word \( (r_D = r_A + I) \)

| 31 | . | . | 26 25 | . | . | 21 20 | . | . | 16 15 | . | . | . | . | . | . | . | . | . | . | 0 |
|----|---|---|------|---|---|------|---|---|------|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode 0x27 | D | A | I | 6 bits | 5 bits | 5 bits | 16 bits |

```
switch(ins >>> 26)
{
    ....
    case 0x27: // addi instruction
        r[(ins>>>21)&0x1F] = r[(ins>>>16)&0x1F] + (ins << 16 >> 16);
        break;
    ....
}
```

Instruction emulation implementation: ~ 600 lines of code

Sign extension from 16-Bit to 32-Bit
COMPARISON: "AND" INSTRUCTION EMULATION FOR ARM

```c
void armv5_and()
{
    uint32_t icode = ICODE;
    int m, rd;
    uint32_t cpsr=REG_CPSR;
    uint32_t Rn,op2,result;
    uint32_t S;
    if(!check_condition(icode)) {
        return;
    }
    rd=(icode>>12)&0xf;
    rn=(icode>>16)&0xf;
    Rn=ARM9_ReadReg(rn);
    cpsr&= ~(FLAG_N | FLAG_Z | FLAG_C);
    cpsr |= get_data_processing_operand(icode);
    op2 = AM_SCRATCH1;
    result=Rn&op2;
    ARM9_WriteReg(result,rd);
    S=testbit(20,icode);
    if(S) {
        if(!result) {
            cpsr|=FLAG_Z;
        }
        if(ISNEG(result)) {
            cpsr|= FLAG_N;
        }
        if(rd==15) {
            if(MODE_HAS_SPSR) {
                SET_REG_CPSR(REG_SPSR);
            } else {
                fprintf(stderr,"Mode has no spsr in line %d\n",__LINE__);
            }
        } else {
            REG_CPSR=cpsr;
        }
    }
    dbgprintf("AND result op1 %08x,op2 %08x, result %08x\n",Rn,op2,result);
}
```
2. MEMORY PROTECTION WITH AN MMU - PHENOMENOLOGY

This happens when you execute `*(int*)NULL = 0xDEADBEEF;`

**Windows 95** (bluescreen or freeze or something random)

**Windows 10**

**Linux**

**Windows NT 4.0**

```
#include<stdio.h>

int main()
{
    *(int*)NULL = 0xDEADBEEF;
    return 0;
}
```
2. THE MEMORY MANAGEMENT UNIT

Run each memory access through a table

- Page size of 8kB. 4 byte per page
  - Total page table size: 2MB to address full 4GB
- *Read, Write, Execute* flags for each page
- In Linux each process gets its own page directory and page table
3. TIMER

CPU.prototype.AdvanceTimer = function(cycles) {
    if ((TTMR>>>30) == 0) return; // check if timer is enabled
    delta = getDelta(TTCR, TTMR); // distance between timer and alarm
    TTCR += cycles; // advance timer

    if (delta < cycles) {
        if (TTMR&(1<<29)) // if interrupt enabled
            TTMR |= (1<<28); // set pending timer interrupt
    }
}

CPU.prototype.CheckTimer = function() {
    if ((SR_TEE) && (TTMR & (1<<28)))
        Exception(EXCEPT_TICK);
}
4. PROGRAMMABLE INTERRUPT CONTROLLER

32-Bit PIC Status register, each bit enables one interrupt line
32-Bit PIC Mask Register, each bit enables one interrupt line

```javascript
CPU.prototype.RaiseInterrupt = function(line) {
    PICSR |= 1 << line;
    CheckForInterrupt();
}

CPU.prototype.ClearInterrupt = function(line) {
    PICSR &= ~(1 << line);
}

CPU.prototype.CheckInterrupt = function() {
    if (!SR_IEE) return; // check CPU interrupt enable flag
    if (PICMR & PICSR) { // compare interrupt mask and interrupt set register
        Exception(EXCEPT_INT);
    }
}
```
5. TERMINAL SUPPORT AND SERIAL INTERFACE

- Terminal output: 360 lines of code
- Terminal input: 90 lines code
- Serial controller:
  - Connected to an interrupt line
  - Memory mapped I/O
  - 220 lines code
WHAT YOU NEED ON THE SOFTWARE SIDE?

- **binutils**
  - assembler, disassembler, linker, elf-file format, application binary interface

- **GNU C Compiler**
  - 6411

- **Linux kernel**
  - 6183

- **C library (e.g. musl)**
  - Interface between user space program and kernel and basic functionality for C programs
  - 2108

- **Busybox**
  - stripped down unix tools in a single executable including shell
  - 0

The first version of jor1k had around 2000 lines of code!
@juliusb poke53281: that is one of the coolest things I've ever seen :)

olo fk

juliusb: Hey, I thought I was the coolest thing you've ever seen

olo fk

ok, you said one of the coolest. Fair enough

Dec. 04th 2012: OpenRISC IRC Chat
SPEEDY JAVASCRIPT
• is very creative with type coercion
  • 0 == "" => true
  • "" - "" => 0
  • if (new Array() == false) => true
  • {} + {} => NaN
  • {} + [] => 0
  • {} + {} + [] => "NaN"
  • ("NaN") => "NaN"
  • ({}) + ({}) + [] => ("[object Object][object Object]"
  • 0 > null => false
  • 0 >= null => true
  • 0 == null => false
  • [1,2,3]+[4,5,6] => “1,2,34,5,6“

• is considered “slow”
  • At least four companies are writing optimized compilers to squeeze out the maximum performance.
ALL NUMBERS ARE DOUBLE

JavaScript doesn’t know about integers, only doubles

- \( y = 9999999999999999 \Rightarrow y = 10000000000000000 \) (double)

But there are logical operations which act on 32-Bit

- \( y = 1.1234 | 2 \Rightarrow y = 3 \)
- \( y = 0xFFFFFFFF | 0 \Rightarrow y = -1 \)
- \( y = -1 >>> 0 \Rightarrow y = 0xFFFFFFFF \)

The JavaScript-engines optimize for accuracy

- \( y = 0x7FFFFFFF \) (treated internal as integer)

There are also typed arrays

- \( x = \text{new} \ \text{Uint32Array}(\text{length}) \)
What's wrong with the following code concerning the compiled code?

```c
for(;;) {
    Advance_Timer();
    Check_Interrupt();
    ppc = Translate_Virtual_To_Physical(vpc);
    ins = ram.Read32(ppc);
    vpc += 4; // advance program counter
    // decode instruction
    switch (ins&0x7F) {
        ....
    }
}
```

**What happens internal?**

1. Add 4 to vpc (integer)
2. Check for overflow
3. Deoptimize into double if overflow
4. Cascade of deoptimizations where pc is used.

---

**HOW TO PREVENT DEOPTIMIZATIONS?**

What happens internal?
HOW TO PREVENT DEOPTIMIZATIONS?

Prevent overflow by adding a “|0”

- for(;;) {
  Advance_Timer();
  Check_Interrupt();
  ppc = Translate_Virtual(vpc);
  ins = ram.Read32(ppc);
  vpc = (vpc + 4)|0;
  // decode instruction
  switch (ins&0x7F) {
    ....
  }
}

What happens internal?
1. Add 4 to vpc (integer)
2. Ignore noop „|0“
HOW TO PREVENT DEOPTIMIZATIONS?

Add more typing helpers

```c
• for(;;) {
    Advance_Timer();
    Check_Interrupt();
    ppc = Translate_Virtual_To_Physical(vpc|0)|0;
    ins = ram.Read32(ppc|0)|0;
    vpc = (vpc + 4)|0;
    // decode instruction
    switch (ins&0x7F) {
        ....
    }
}
```
WHAT IS ASM.JS?

The mode “use strict”; adds additional error messages for accessing undefined variables.

The mode “use asm”; adds additional error messages to give you a guarantee for fixed type variables that must be compiled only once.

- Only a subset of Javascript is allowed
- Fully compatible

- Implemented in Firefox in 2013
- Implemented in Edge in 2015
WHAT IS ASM.JS?

But the syntax is nasty

- \( \text{group0[SPR_IMMUCFGR]} = 0x18; \)
- \( \text{h[group0p + (SPR_IMMUCFGR \ll 2) \gg 2]} = 0x18; \)

\( h \) is the heap and \( \text{group0p} \) is the pointer to the table.

In this case the “view” of the heap is 32 Bit. Hence the last operation for the index must be “\( \gg 2 \)”.

Project Emscripten allows to translate C++ to asm.js JavaScript.
"asm.js requires a style of coding only compilers can output. A person writing actual asm.js code by hand would need to be insane as asm.js code required style of coding is horribly disorganized"[1]

"Unfortunately asm.js requires one giant array to put things on. No one in their right mind codes like that by hand."[2]

From Grant Galitz (JavaScript GameBoy Advance Emulator)

---

binarymax 1093 days ago [-]

Very impressive work. I saw Brendan Eich speak At jquery uk in April and he said asm.js was meant for compilers and people should not be programming it directly...I couldn't help but smile and think 'oh yeah!'

---

On Hacker News Sep 17th 2013 regarding jor1k

Sephr 493 days ago [-]

That's quite an impressive implementation! I didn't realize hand-written asm.js could be that readable.

---

On Hacker News May 10th 2015 regarding jor1k
FURTHER OPTIMIZATIONS

Reducing the code per EMULATED instruction which has to be executed
THE MMU: SOFTWARE TLB LOCKUP

Implemented in two stages

1. Full translation table in memory

2. TLB variables (tlb buffer with one entry)

Translation fastpath of virtual to physical addresses:

```c
if ((tlb_check ^ virtual_addr) >> 12)
{
    ...  
    tlb_check = ...  
    tlb_trans = ...
}
physical_addr = tlb_trans ^ virtual_addr;
```
FENCE TECHNIQUE

For every instruction the program counter (pc) must be translated to the physical pc. However, the pc advances usually by 4 and is not leaving the current page. This property can be used by the fence technique.

The fastpath for one instruction looks like this:

```c
for(;;) {
    if ((physical_pc|0) != (fence|0)) {
        ins = int32ram[physical_pc >> 2]|0;
        physical_pc = physical_pc + 4|0;

        switch (ins&0x7F) {
            ....
        }
    } else {
        Advance_Timer();
        Check_Interrupt();
        ....
    }
}
```

The idea here is that the virtual pc is computed only when needed by translating ppc (physical pc) back to the virtual pc address. The variable fence is used to break out of the fast path when ppc reaches a jump or the end of the current page.
<table>
<thead>
<tr>
<th>Javascript code</th>
<th>Block description</th>
<th>Generated x86 asm code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(;;) {</td>
<td>.set .Llabel132981, .</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if ((fence</td>
<td>0) != (ppc</td>
<td>0)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Movl (nil), %eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cmpl %eax, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>je .Lfrom133000</td>
</tr>
<tr>
<td>ins = ram[ppc &gt;&gt; 2]</td>
<td>MoveGroup</td>
<td>movl %eax, %ecx</td>
</tr>
<tr>
<td>if</td>
<td>BitOpI:bitand</td>
<td>andl $0xfffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td>AsmJSLoadHeap</td>
<td>cmpl $0xfffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ja .Lfrom133022</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl 0x0000(%ecx), %ecx</td>
</tr>
<tr>
<td>ppc = ppc + 4</td>
<td>MoveGroup</td>
<td>movl %ecx, 0x2c(%esp)</td>
</tr>
<tr>
<td>if</td>
<td>instruction</td>
<td>addl $4, %eax</td>
</tr>
<tr>
<td></td>
<td>AddI</td>
<td>movl %eax, (nil)</td>
</tr>
<tr>
<td></td>
<td>AsmJSStoreGlobalVar</td>
<td></td>
</tr>
<tr>
<td>switch(ins&amp;0x7F) {</td>
<td>MoveGroup</td>
<td>movl 0x2c(%esp), %edx</td>
</tr>
<tr>
<td></td>
<td>instruction</td>
<td>andl $0x7f, %edx</td>
</tr>
<tr>
<td></td>
<td>BitOpI:bitand</td>
<td>TableSwitch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subl $3, %edx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmpl $0x71, %edx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jae .Lfrom133081</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl $0xfffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jmp *0x0(%ecx,%eax,4)</td>
</tr>
<tr>
<td>Javascript code</td>
<td>Block description</td>
<td>Generated x86 asm code</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>for(;;) {</td>
<td>.set .Llabel132981, .</td>
<td></td>
</tr>
<tr>
<td>if ((fence</td>
<td>0) != (ppc</td>
<td>0)) {</td>
</tr>
<tr>
<td>ins = ram[ppc &gt;&gt; 2]</td>
<td>MoveGroup BitOpI:bitand</td>
<td>movl %eax, %ecx andl $0xffffffff, %ecx</td>
</tr>
<tr>
<td>ins = ram[ppc &gt;&gt; 2]</td>
<td>AsmJSLoadHeap</td>
<td>cmpl $0xffffffff, %ecx ja .Lfrom133022 movl 0x0000(%ecx), %ecx</td>
</tr>
<tr>
<td>ppc = ppc + 4</td>
<td>MoveGroup instruction</td>
<td>movl %ecx, 0x2c(%esp)</td>
</tr>
<tr>
<td>switch(ins&amp;0x7F) {</td>
<td>BitOpI:bitand</td>
<td>movl 0x2c(%esp), %edx andl $0x7f, %edx</td>
</tr>
<tr>
<td></td>
<td>TableSwitch</td>
<td>subl $3, %edx cmpl $0x71, %edx jae .Lfrom133081 movl $0xffffffff, %ecx jmp *0x0(%ecx,%eax,4)</td>
</tr>
</tbody>
</table>

Unnecessary load
<table>
<thead>
<tr>
<th>Javascript code</th>
<th>Block description</th>
<th>Generated x86 asm code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(;;) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.set .Llabel132981, .</td>
</tr>
<tr>
<td>if ((fence</td>
<td>0) != (ppc</td>
<td>0)) {</td>
</tr>
<tr>
<td></td>
<td>BitOpI:bitand</td>
<td>Movl (nil), %eax</td>
</tr>
<tr>
<td></td>
<td>AsmJSLoadHeap</td>
<td>Cmpl %eax, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>je .Lfrom133000</td>
</tr>
<tr>
<td>ins = ram[ppc &gt;&gt; 2]</td>
<td>MoveGroup</td>
<td>movl %eax, %ecx</td>
</tr>
<tr>
<td></td>
<td>BitOpI:bitand</td>
<td>andl $0xfffffffffc, %ecx</td>
</tr>
<tr>
<td></td>
<td>AsmJSLoadHeap</td>
<td>cmpl $0xfffffffffc, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ja .Lfrom133022</td>
</tr>
<tr>
<td></td>
<td>MoveGroup</td>
<td>movl 0x0000(%ecx), %ecx</td>
</tr>
<tr>
<td>ppc = ppc + 4</td>
<td>instruction Addl</td>
<td>addl $4, %eax</td>
</tr>
<tr>
<td></td>
<td>AsmJSStoreGlobalVar</td>
<td>mov %eax, (nil)</td>
</tr>
<tr>
<td>switch(ins&amp;0x7F) {</td>
<td>MoveGroup</td>
<td>movl 0x2c(%esp), %edx</td>
</tr>
<tr>
<td></td>
<td>instruction BitOpI:bitand</td>
<td>movl $0x7f, %edx</td>
</tr>
<tr>
<td></td>
<td>TableSwitch</td>
<td>subl $3, %edx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmpl $0x71, %edx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jae .Lfrom133081</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl $0xffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jmp *0x0(%ecx,%eax,4)</td>
</tr>
</tbody>
</table>

1 sub to save 12 bytes in a table
Add dummy case 0:
<table>
<thead>
<tr>
<th>Javascript code</th>
<th>Block description</th>
<th>Generated x86 asm code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(;;) {</td>
<td>.set .Llabel132981, .</td>
<td></td>
</tr>
<tr>
<td>if ((fence</td>
<td>0) != (ppc</td>
<td>0)) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Movl (nil), %eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cmpl %eax, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>je .Lfrom133000</td>
</tr>
<tr>
<td>ins = ram[ppc &gt;&gt; 2]</td>
<td>MoveGroup</td>
<td>movl %eax, %ecx</td>
</tr>
<tr>
<td></td>
<td>BitOpI:bitand</td>
<td>andl $0xffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td>AsmJSLoadHeap</td>
<td>cmp $0xffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ja .Lfrom133022</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Movl 0x0000(%ecx), %ecx</td>
</tr>
<tr>
<td>ppc = ppc + 4</td>
<td>MoveGroup</td>
<td>movl %ecx, 0x2c(%esp)</td>
</tr>
<tr>
<td></td>
<td>AsmJSStoreGlobalVar</td>
<td>addl $4, %eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl %eax, (nil)</td>
</tr>
<tr>
<td>switch(ins&amp;0x7F)</td>
<td>MoveGroup</td>
<td>movl 0x2c(%esp), %edx</td>
</tr>
<tr>
<td></td>
<td>BitOpI:bitand</td>
<td>andl $0x7f, %edx</td>
</tr>
<tr>
<td></td>
<td>TableSwitch</td>
<td>subl $3, %edx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmp $0x71, %edx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ja .Lfrom133081</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movl $0xffffffff, %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jmp *0x0(%ecx,%eax,4)</td>
</tr>
</tbody>
</table>

Unnecessary check
Interesting, but most surprising for me is the asm.js part, I heard it can be faster, but by this much... holy Emacs, with standard core it runs at 6.5 MIPS, but with asm.js it's running at a 100 MIPS. Even in Chromium it runs at 50MIPS with asm.js and at 9MIPS with the standard core.
Hi Sebastian,

I spent some time converting jslinux to asm.js. It is still slower than your excellent jor1k but your benchmark at https://github.com/s-macke/jor1k/wiki/Benchmark-with-other-emulators needs to be revised 😊

Best regards,

Fabrice.
BENCHMARK FROM SEP 2015

The chart shows benchmark results for different programs and architectures:

- **JSLinux** and **v86** are evaluated for x86 architecture.
- **jor1k** and **jor1k-riscv** are evaluated for OpenRISC and RISC-V architectures.

The y-axis represents seconds, with lower values indicating better performance. The chart highlights the performance difference across different benchmarks:

- **dd**: jor1k-riscv is significantly faster than the other options.
- **gzip**: jor1k-riscv again shows better performance.
- **bzip2**: jor1k-riscv performs better than x86 and OpenRISC.

The conclusion is that **Lower is better**, indicating that lower times are preferable across all benchmarks considered.
BENCHMARK FROM SEP 2015

- Firefox on Core2Duo: 120 MIPS
- Firefox on Core i7 2600: 75 MIPS
- Firefox on Celeron G1820: 180 MIPS
- Firefox on Core i7 4770: 246 MIPS
- Chrome: on Core i7 2600: 60 MIPS
- IE 11 on Core i7 2600: 68 MIPS
- Safari on Apple A7: 81 MIPS

Approx. speed of the CPUs of the year 1997

OpenRISC CPU implementation: 1609 lines of code
RISC-V CPU implementation : 2181 lines of code
MODULARIZATION AND PARALLELIZATION
SEPARATE INTO TWO THREADS

Master
- Parameters from html file
  - Terminal screen
  - Keyboard events
  - Ethernet data transfer
  - Framebuffer screen
  - Touchscreen events
  - Sound buffer
  - File upload and download

Worker
- OpenRISC-CPU
  - Memory
  - RAM
  - Terminal
  - Keyboard
  - Ethernet
  - ATA hard drive
  - Framebuffer
  - Touchscreen
  - Sound
  - Real time clock
  - Virtio
  - 9p
  - Filesystem

Messages

interrupts
MULTIPLE CORES

You need additionally:

• Core ID (consecutive number)
• Software interrupt: one core can send interrupt to another core
• One global timer, but separate alarms for each core.
• New synchronization instructions
  • Load-Link/Store Conditional (read-modify write operation, atomic)

Currently implemented (buggy) in one thread with time shaping

Mozilla announced SharedArrayBuffer and atomics for Javascript to allow real multithreading
Java?

This is JavaScript

HOW TO NOT IDLE IN JAVASCRIPT?
HOW TO STAY 100% BUSY IN JAVASCRIPT BUT REMAIN RESPONSIVE?

Javascript finishes current job before continuing.

Solution: `setTimeout(function(){…}, 0);

Run 20 ms then idle for 0 ms
Run 20 ms then idle for 0 ms
Run 20 ms then idle for 0 ms

Warning: Unresponsive script

A script on this page may be causing problems. You can stop the script now, open the script in the debugger, or click the message to view details.

Script: http://

Don't ask me again
WHAT ABOUT THE WORKER THREAD?

- `setTimeout(function(){...}, 0);` doesn’t work in worker thread. Message queue is never processed.
- But `setTimeout(function(){...}, 4);` works (4ms waiting time)
WHAT ABOUT THE WORKER THREAD?

Can we get the number of messages in the queue or have non-blocking access?

NOPE!
SOLUTION

Play message ping pong, so that at least one message is always in the queue.
“Pong“ message executes the CPU for 20ms.
This was harmless!

Next time:
The horror of exact timing in JavaScript

How to implement a (streaming) audio device into JavaScript?

- Unreliable speed of Javascript
- Only millisecond time resolution
- Events trigger only, when you are idle
- Message queue between worker and master
• Yo dawg, I heard you like browsing the web, so I put a browser in your browser so you can browse while you browse! (Twitter user Scott Elcomb)
NETWORK

Server in the USA

• connected via websockets
• Sending and receiving ethernet frames connected to a Linux TAP device

Full working intranet

• Start jor1k in two windows and open a ssh session between them.

Major network applications available

• wget, curl, nc, ping, traceroute, telnet, ssh, nmap
• Openssl with certificates
• Web browsers: lynx, links, dillo
The Filesystem

How to implement an efficient filesystem with a size of 200MB and 5000 files that runs on the website?
THE FILESYSTEM

How long does it take to get the size of a directory structure with 10000 files over the internet?

• NFS
• Samba
• Sshfs
• On demand block device

Problem is mainly latency, not throughput

Advantages of our filesystem:
- Read only filesystem on server
THE FILESYSTEM

Implement filesystem outside of the emulator to have full control

- tmpfs like. Use virtio/9p as “file system as hard ware device” to exchange commands with Linux

Load the filesystem layout and metadata during the Linux boot process.

```json
{
  "name": "mtd_probe", "mode": "100775", "size": 3996, "c": 1,
  {
    "name": "v4l_id", "mode": "100775", "size": 4300, "c": 1,
    {
      "name": "collect", "mode": "100775", "size": 10444, "c": 1,
      { "name": "ata_id", "mode": "100775", "size": 10352, "c": 1,
      { "name": "accelerometer", "mode": "100775", "size": 14812, "c": 1
    },
  },
  { "name": "libdev.so.1.3.0", "mode": "100775", "size": 142420, "c": 1,
  { "name": "libdev.so.1", "mode": "120777", "path": "libdev.so.1.3.0"
  }
},
```

Load compressed files on demand.

- OpenRISC binaries compress really well
- .bz2 currently, in future .xz
- Ordinary web server needed

Future: dependencies between files. packages

- http 2.0 will help heres
BOOT PROCESS TIMELINE

- vmlinux.bz2
- basefs.json
- etc/*
- busybox.bz2
- extendedfs.json
- libc.so.bz2
- libz.so.bz2
- licurses.so.bz2

Load of filesystem from server
Parallel load of files from filesystem
Non-parallelized decompression of file

Kernel boot
Login screen
ADDITIONAL FEATURES OF THE FILESYSTEM

Atomic file operations
Full control from the outside
Watching Files
Upload files into home folder
Download home folder (as .tar)
My own cloud: Sync with server
  • Unique user id (http://s-macke.github.io/jorlk/?user=cdqKKPjfa)
  • Currently 1MB quota
  • server only needs upload.php
IS THE EMULATOR USEFUL?

Well, sort of ...

- Technology demonstration, Advertisement
- Interactive online tutorials
- Easy way to port and present terminal software
- Teaching programming languages
- Rogue like Network access
- Fast testing environment for binaries
- JavaScript benchmark
- You can play games like Doom, Monkey Island, Elite II and Toppler
FUTURE

• Full virtio driver support (done)
• Virtio-GPU and full-screen X-Window system
• More terminals, better user interface
• Download already booted Linux (state file)
• Status, statistics and debug screen
• Run Java
• Run Firefox
THANKS

• Stefan Kristiansson for the toolchain and infinite help in the chat.
• Ben Burns for implementing the network and providing the relay server
• Prannoy Pilligundla for implementing RISC-V
• Lawrence Angrave and Neelabh Gupta for the C-development website
• Jonas Bonn for the Linux kernel support
• Christian Svensson for the OpenRISC Debian distribution