Skiron

Experiments in CPU Design in D

MITHUN HUNSUR

Agenda

- Skiron
- Why D?
- Idioms
- Ecosystem wonders
- Lessons learnt
- Annoyances
- The Future

Skiron

State of Skiron

- Hobbyist architecture to learn about design compromises + D (ab)use
- Currently focusing on instruction set simulation
- 32-bit RISC-inspired pure load-store architecture
- Suite of associated software – currently primary product
- No hardware implementation *yet*
- Inspired by MIPS/ARM, while rapidly backing away from x86

- *Unlikely* to take off ☺

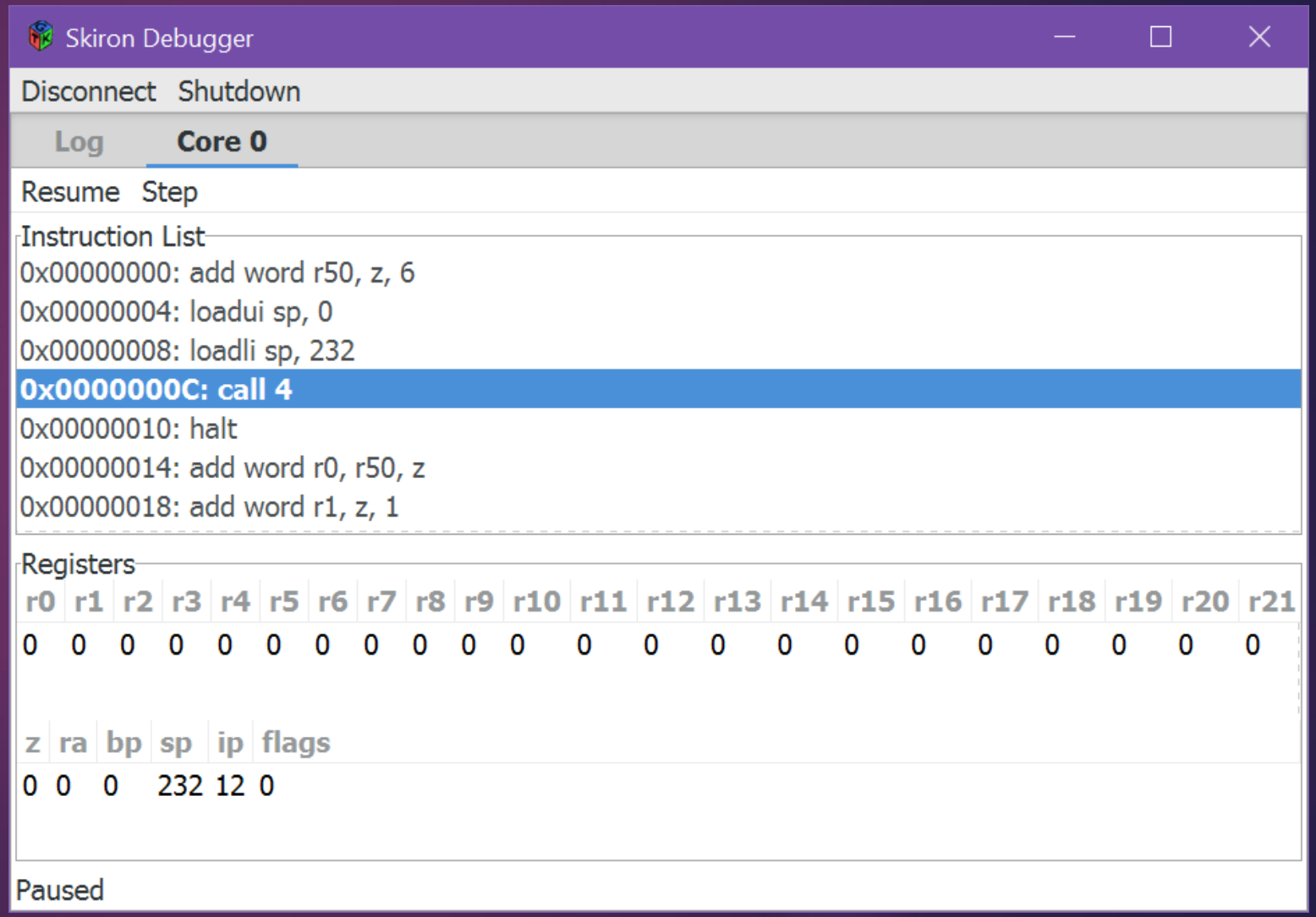
Skiron Projects

Non-GC

- *common*
- *emulator*

GC

- *assembler*
- *disassembler*
- *debugger_backend*
- *debugger_graphical*
- *test_runner*
- *docgen*



Current register layout

- 64 general-purpose integer registers
- Last 5 are reserved:
  - z
  - ra
  - bp
  - sp
  - ip

Current instruction set

- ~23 core instructions
- Load/store
- Arithmetic
- Control flow
- Not much else! Other features need to be implemented first

- Pseudoinstructions compose useful functionality from opcodes
- Example: stack manipulation

Current opcode layouts

A

6 bits	2 bits	2 bits	6 bits	6 bits	6 bits	2 bits	2 bits
opcode	encoding	variant	register1	register2	register3	padding	operandSize

B

6 bits	2 bits	2 bits	6 bits	16 bits
opcode	encoding	variant	register1	immediate

C

6 bits	2 bits	2 bits	20 bits	2 bits
opcode	encoding	variant	immediate	operandSize

D

6 bits	2 bits	2 bits	6 bits	6 bits	8 bits	2 bits
opcode	encoding	variant	register1	register2	immediate	operandSize

But let's leave it at that for now

- Plenty of work to be done on the CPU (simulation) side of things
  - Context switching
  - Multi-core systems
  - Proper IO
  - Interrupts
  - Instruction pipelining
  - Caching
  - Branch prediction
  - *And so much more*
- If you'd like to learn more about CPU design compromises, check out Andrew Waterman's Design of the RISC-V Instruction Set Architecture:

http://www.eecs.berkeley.edu/~waterman/papers/phd-thesis.pdf

Why D?

What about the other languages?

C/C++

- Slow compile times
- Obtuse syntax, especially with metaprogramming
- Painful to use
- Wrote a basic x86 ISA emulator in C++ a couple years ago
- Metaprogramming is just awful
- Classic example: enum <-> string

Go

No.

Rust

- Serious competitor
- Why not?

- Metaprogramming not as in-depth
- Slow compile times
- Poor Windows support at the time
- Slow to develop with

Back to D

- Nice, pragmatic language
- Very powerful meta-programming
- Very flexible
- Can be used for all kinds of problems
- Easy cross-platform
  - At least on x86 ☺
- For the most part, it Just Works™

D niceties that *aren't* metaprogramming

- *final switch*
- Binary constants
- *std.algorithm*, *std.range* (with caveats)
- Module system now, instead of ~~2014~~ ~~2017~~ 2020?
- Pleasant syntax
- Easy refactoring, especially with UFCS
- *static if*
- Unit tests
- Fast compile times
- Many other things!

“One source of truth”

- Design details need to be easy to change
- Not easy by default
- Handle assembler, emulator, documentation generation, all at once
- “Constrained” problem
- *Metaprogramming to the rescue*

Idioms

Constants flow down

- Changing one thing will update everything – constants directly drive the code
- Example:

```
enum RegisterBitCount = 6;
```

- Will drive
  - Number of available registers
    - Indices of the special registers
    - Documentation
    - Graphical debugger
  - Instruction encodings

Automatic serialization

- Everyone and their dog has written a serialization library in D – it's very easy
- Using mine:

```
mixin GenerateIdEnum!("DebugMessageId");

struct Initialize
{
	uint memorySize;
	uint coreCount;
	uint textBegin;
	uint textEnd;

	mixin Serializable!DebugMessageId;
}
```

```
struct CoreGetState
{
	uint core;

	mixin Serializable!DebugMessageId;
}
```

Memory mapped devices

- Along similar lines to serialization

```
class Screen : Device {
        @MemoryMap(0, AccessMode.Read)
        uint width;

        @MemoryMap(4, AccessMode.Read)
        uint height;

        @MemoryMap(8, AccessMode.ReadWrite)
        Pixel[] pixels;

        mixin DeviceImpl;
}
```

The “descriptor enum” pattern

- So let’s look at a problem
- **How do we define instructions in such a way that we can easily utilise them?**

- What do we want to know about an instruction?
  - Its internal name (“AddA”)
  - Its user name (“add”)
  - Its index (4)
  - Its operand format (destination, source, source)
  - Its description (“Add `src1` and `src2` together, and store the result in `dst`.”)

OpcodeDescriptor enum


```
struct OpcodeDescriptor
{
    string name;
    ubyte opcode;
    OperandFormat operandFormat;
    string description;
}
```

```
enum Opcodes
{
    …,
    AddA = OpcodeDescriptor(
        "add",
        4,
        OperandFormat.DstSrcSrc,
        "Add `src1` and `src2` together,
and store the result in `dst`."),
    …
}
```

What can you do with a descriptor enum?

- Let's look at one of my favourite examples from the assembler:

```
// Construct the AA of pseudoinstructions => assemble functions
auto generatePseudoAssemble()
{
    enum opcodes = [EnumMembers!Opcodes];
    return "[%s]".format(
        opcodes.filter!(a => a.operandFormat == OperandFormat.Pseudo)
               .map!(a => `"%s": &assemble%s`.format(a.name, a.to!string()))
               .join(", "));
}
this.pseudoAssemble = mixin(generatePseudoAssemble());
```

Application in the emulator

- This is used within the emulator to a similar, even more powerful effect.
- The "opcode dispatcher" will *automatically* dispatch the given opcode to the function that simulates it (massively cut down):

```
string generateOpcodeSwitch() {
    string s = `switch (instruction.opcode) {`;
    foreach (member; EnumMembers!Opcodes) {
        s ~= format(
`case Opcodes.%1$s.opcode:
        switch (instruction.operandSize) {
            case OperandSize.Byte4:
                this.run%1$s!uint(opcode); break;
        }
        break;`, member.to!string());
        s ~= "}\n";
        return s;
}
```

```
void runAddA(Type = uint)(ref Core core, Opcode opcode)
{
    core.setDst!Type(opcode,
        core.getSrc1!Type(opcode) +
        core.getSrc2!Type(opcode));
}
```

Encoding definitions

- Next problem: defining what a particular encoding for opcodes will look like.
- Want it to be self-explanatory
- Why not use *std.bitmanip.bitfields*?

6 bits	2 bits	2 bits	6 bits	6 bits	6 bits	2 bits	2 bits
`opcode`	`encoding`	`variant`	`register1`	`register2`	`register3`	`padding`	`operandSize`

DefineEncoding


```
mixin DefineEncoding!(Encoding.A,
        "Used for three-register instructions.",
        ubyte,              "opcode",         OpcodeBitCount,
        "The opcode number.",
        Encoding,           "encoding",       EncodingBitCount,
        "The encoding in use.",
        Variant,            "variant",        VariantBitCount,
        "The variant/modifier to apply to register3.",
        Register,           "register1",      RegisterBitCount,
        "The destination register.",
        Register,           "register2",      RegisterBitCount,
        "The first source register.",
        Register,           "register3",      RegisterBitCount,
        "The second source register.",
        ubyte,              "_padding",       2,
        "",
        OperandSize,        "operandSize",    OperandSizeBitCount,
        "The sizes of the operands being used.",
);
```

```
struct EncodingDescriptor
{
        struct Field
        {
                string type;
                string name;
                int size;
                string description;
        }

        string name;
        string description;
        Field[] fields;
}
```

6 bits	2 bits	2 bits	6 bits	6 bits	6 bits	2 bits	2 bits
opcode	encoding	variant	register1	register2	register3	padding	operandSize

EnumDocumented

- Documentation's important
- How do we keep enums documented?
- Presenting EnumDocumented

```
mixin EnumDocumentedDefault!("Variant",
        "Identity",
                "Pass the operand through unchanged.",
        "ShiftLeft1",
                "Shift the operand 1 bit to the left.",
        "ShiftLeft2",
                "Shift the operand 2 bits to the left.",
);
```

EnumDocumented output


```
enum Variant { Identity, ShiftLeft1, ShiftLeft2, }
enum VariantDocs = [
        tuple(Variant.Identity, "Pass the operand through unchanged."),
        tuple(Variant.ShiftLeft1, "Shift the operand 1 bit to the left."),
        tuple(Variant.ShiftLeft2, "Shift the operand 2 bits to the left."),
];
```

```
foreach (pair; VariantDocs)
{
        file.writefln("* **%s**", pair[0].to!string());
        file.writefln("    * *Index*: %s", cast(uint)pair[0]);
        file.writefln("    * *Description*: %s", pair[1]);
}
```

Ecosystem wonders

Hidden gems: multiSort

- Was using the old Stack Overflow answer – a predicate that's unwieldy to work with:

```
descriptors.sort!((a,b) {
        if (a.operandFormat != OperandFormat.Pseudo && b.operandFormat != OperandFormat.Pseudo) {
                if (a.opcode < b.opcode) return true;
                if (b.opcode < a.opcode) return false;
        }

        if (a.operandFormat == OperandFormat.Pseudo && b.operandFormat != OperandFormat.Pseudo)
                return false;

        if (b.operandFormat == OperandFormat.Pseudo && a.operandFormat != OperandFormat.Pseudo)
                return true;

        return a.name < b.name;
});
```

Hidden gems: multiSort

- And then I discovered multiSort:

```
auto descriptors = [EnumMembers!Opcodes];
descriptors.multiSort!(
    (a, b) => a.operandFormat != OperandFormat.Pseudo &&
              b.operandFormat == OperandFormat.Pseudo,
    (a, b) => a.opcode < b.opcode,
    (a, b) => a.name < b.name);
```

GtkD

- GtkD is very full-featured
- Bindings feel practically native

SimpleWindow is pretty simple

- https://github.com/adamdruppe/arsd

```
void handleWindow(ref State state, ScreenDevice screen, Keyboard keyboard, Thread processThread)
{
	auto window = new SimpleWindow(screen.width, screen.height, "Skiron Emulator");
	auto displayImage = new Image(window.width, window.height);

	window.eventLoop(16, () {
		if (!processThread.isRunning) {
			window.close();
			return;
		}

		copyImage(displayImage, screen);

		auto screenPainter = window.draw();
		screenPainter.drawImage(Point(0, 0), displayImage);
	},
	(KeyEvent ke) {
		// Temporary
		keyboard.key = ke.key;
	});
}
```

Visual D's unexpected advantage

- I use premake to generate my build files (thanks Manu!)
- On Windows, I use Visual D
- In recent versions, Visual Studio includes a profiler
- So I thought… could it work?

A horrifying realisation

- It did work.

```
8.2 %   while (this.cores.any!(a => a.running) || this.client.isValid)
        {
11.7 %      foreach (ref core; this.cores.filter!(a => a.running))
            {
7.1 %           core.step();
```


WATICAN

Lessons learnt

Be careful with your delegates!

- Taking a closer look at that code:

```
while (this.cores.any!(a => a.running) || this.client.isValid)
{
        foreach (ref core; this.cores.filter!(a => a.running))
        {
```

- Look carefully at the arguments to the delegates, while pondering the fact that Core is a struct

- **These delegates were copying the cores, getting the value, then throwing them away**

Oops.

- Two ways to fix:
  - Use ref
  - Rewrite so that filter/any aren't used

Fixed version


```
< 0.1 %     while (running || this.client.isValid)
            {
  0.1 %         running = false;

  0.7 %         foreach (ref core; this.cores)
                {
  0.3 %             if (!core.running)
                        continue;

 30.0 %         core.step();
```

Mixins aren't *always* necessary

- Classical case: converting enum-with-extensions to string
- Could have used *std.conv*, but wanted @nogc + strings for non-enum values

```
char[] registerName(Register index, char[] buffer) @nogc nothrow {
        string generateRegisterIf() {
                string ret = "";
                foreach (member; EnumMembers!Register) {
                        string name = member.to!string();
                        ret ~= "if (index == Register.%s) return \"%s\".sformat(buffer);\n".format(
                                name, name.toLower());
                }

                ret ~= `else return "r%s".sformat(buffer, cast(ubyte)index);`;
                return ret;
        }
        mixin(generateRegisterIf());
}
```

Mixins aren't *always* necessary

- Wait a second.

```
char[] registerName(Register index, char[] buffer) @nogc nothrow {
        foreach (member; EnumMembers!Register) {
                enum loweredName = member.to!string.toLower.idup;

                if (index == member)
                        return "%s".sformat(buffer, loweredName);
        }

        return "r%s".sformat(buffer, cast(ubyte)index);
}
```

Top-level GC

- So there's lots of code using the GC
- But our emulator is @nogc!

Top-level GC

- So there's lots of code using the GC
- But our emulator is @nogc!

```
void main(string[] args)
{
        // Read config
        // Validate user path
        // Assemble if required
        // Read and parse program
        // Create IO devices
        // Create state
        // Create threads to drive state and debugger
        // Spawn a window
        // Wait for threads
        // Print performance stats
}
```

Potential improvements

BECAUSE D ISN'T PERFECT

Metaprogramming improvements

- Allowing symbols within declarations
- Would really love to be able to do something like this:

```
enum $(name) {
        foreach (member; __traits(allMembers, Module))
                $(member),
}
```

Metaprogramming improvements

- For comparison, what the actual version looks like:

```
string generateIdEnum(alias Module)(string name)
{
    string s = "enum " ~ name ~ " : ubyte {";
    foreach (member; __traits(allMembers, Module)) {
        // WORKAROUND: Issue 15907
        static if (member != "object" && member != "common") {
            alias memberField =
                Identity!(__traits(getMember, Module, member));

            s ~= member ~ ",\n";
        }
    }
    s ~= "}";
    return s;
}
```

Metaprogramming improvements

- CTFE state!

```
ubyte opcodeIndex = 0;
enum Opcodes
{
    …,
    AddA = OpcodeDescriptor(
        "add",
        opcodeIndex++,
        OperandFormat.DstSrcSrc,
        "Add `src1` and `src2` together, and store the result in `dst`."),
    …
}
```

Annoyances

BECAUSE D *REALLY* ISN'T PERFECT

GC

- Classes are tied to the GC
- Exceptions are tied to the GC
- Delegates are tied to the GC

- In *theory* can be used without the GC
  - Considerably less powerful
  - Or just not doable easily

GC in the standard library

- There's lots of GC in the standard library!
- So much reinvention to avoid landmines; examples relevant to Skiron:
  - NonBlockingSocket
  - sformat
  - Containers
  - StopWatch (https://issues.dlang.org/show_bug.cgi?id=15991)

Spot the bug


```
mixin EnumDocumentedDefault!("Variant",
        "Identity",
                "Pass the operand through unchanged."
        "ShiftLeft1",
                "Shift the operand 1 bit to the left.",
        "ShiftLeft2",
                "Shift the operand 2 bits to the left.",
);
```

The Future

Real hardware

- Port to a FPGA so that it's running on actual hardware
- FPGAs are usually "programmed" in a hardware description language like Verilog or VHDL
- Obviously, not D ☺

- Would like to investigate generating HDL code from D

Self-hosting

- The Holy Grail
- Be able to run the Skiron software suite on Skiron hardware

Shoutouts

- The GtkD team for making fantastic bindings
- Brian Schott for creating std.experimental.lexer
- Rainer Schuetze and team for Visual D
- Manu Evans for premake, and for ardently pushing for a GC-reduced D

Thanks for listening!

Questions?

Email: me@philpax.me

Website: http://philpax.me

Twitter: @Philpax_