Scribe notes on WebAssembly and RISC

Jade Sheffey and Simon Andrews

February 22, 2022

1 Pseudo-ISAs

A "pseudo-ISA" is an ISA (instruction set architecture) that was never designed to run on real hardware, only in emulators. Some popular ISAs include

- WebAssembly (refer to Section 3),
- .NET/CLR (Common Language Runtime),
- the JVM (Java Virtual Machine), amd
  - Hardware - rockwell chip
- the UCSD P-code.

Despite it not being their intended purpose, there is no reason that pseudo-ISAs can't be run on real hardware. For example, Rockwell made a chip for running JVM instructions. However, Emery thinks that this is generally a bad idea. This is because pseudo-ISAs are often designed for stack machines, which are often slower than register-based machines in the real world. Stack machines are popular for virtual machines though because they are more abstract (so you don't need to deal with limitations like having a fixed number of registers to work with).

```
push        r1
push        r2
add     r3 ← r1, r2
```

Figure 1: An example of what a stack machine program for addition might look like.

2 Abstractions

> "Hardware is an abstraction over transistors, and transistors are an abstraction over semiconductors."

There are many levels of abstractions below an ISA. First is the ABI (application binary interface). An ABI is a mapping that says which arguments for an instruction correspond to which registers. Below the ISA and ABI, there is the "microarchitecture," which encompasses things like branch prediction systems, pipelines, and caches. This layer is not visible semantically, but only at execution time.

One particularly important part of some microarchitectures is Intel's micro-ops, which it uses to implement its regular instructions (`mov`, etc.). This allows them to ship updates to their instructions if a bug is discovered. One real example is a bug with the FDIV instruction, which led to substantial rounding errors that were visible to users in programs like Excel. This was embarrasing for Intel and led to a large recall of processors. (AMD made the first provably-correct implementation of FDIV after this.)

A "leaky" abstraction is an abstraction that doesn't hide the underlying *thing* well. For example, the modulus operator in C is slow on some computer architectures and is fast on others. Compiler writers need

```
// A, B, and C are 2D arrays of size N by N.
// C is all zeros at the start.
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < N; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Figure 2: C code for multiplying matrices

C/C++
assembly (with ISA), object code (with ABI)
Microarchitecture
Transistors
Semiconductors

Figure 3: Hierarchy of abstractions

to keep track of this if they want to generate efficient code for all their targets. Another example is cache hits and misses: while it seems like it shouldn't matter what order `i`, `j`, and `k` are iterated over in the for-loops in Figure 2, it actually matters a great deal since accessing `A` and `B` in certain ways improves spatial locality and allows for better caching.

3 WebAssembly

3.1 Why not JavaScript?

JavaScript compilation is hard and slow. The main reasons for this are that JavaScript

- *is a dynamic language.* Any part of any program can be redefined at any time.
- *has a syntax that is complicated and hard to parse.*
- *is polymorphic.* The result of `x + 3` depends on whether `x` is a number or a string.
- *has weird scoping rules.* `var` in particular makes it hard to know what code will do without a lot of context.
- *has prototype-based inheritance.* Originally creating a new object was done by taking an old object and modifying it. Newer versions of JavaScript have class-based inheritance, but this is just syntactic sugar over the old system.

3.2 Emscripten

Emscripted started as an experiment by Alon Zakai. It's original goal was to create a C/C++ to JavaScript transpiler, that targeted only a small subset of JavaScript called asm.js. This subset

- *could simulate C's abstract machine.* It used a JavaScript array to simulate having a big block of memory.
- *was restricted to only features that were easy to compile and execute.*

Special comments inserted into the generated JavaScript indicated to browser engines that this restricted subset was being used, which allowed them to make some assumptions about the code and execute it faster.

Apple and Google thought this was a gross hack, and spent a lot of time trying to write better JavaScript compilers, but they weren't able to.

3.3 WASM today

Vendors eventually got together and decided that instead of continuing with crazy JavaScript hacks they should just make something better. The result is WebAssembly. It uses basically the same abstraction as what Alon came up with (big block of memory), but with files that are much easier to parse than regular JavaScript code. Another major benefit is that you can more easily have an efficient WASM JIT than an efficient JavaScript JIT,

Despite some claims to the contrary, WASM is not as fast as native code [1]. It is estimated to be roughly 20% faster than regular JavaScript and 20% slower than native code for most tasks.

Today WASM is also being used outside of browsers – including for Node. Today we have projects like Doppio [3], which allows for programs written in general purpose languages to run inside a browser (e.g. JVM), and Browsix [2], a browser-based UNIX layer.

WASM code can be generated from C, C++, FORTRAN, Rust, and pretty much anything that is supported by LLVM. All these languages are statically typed and have no garbage collection. Garbage collected languages like JavaScript and Java are hard to compile to WASM because it has no garbage collector yet. There are many proposals for adding garbage collection to WASM, but this is controversial.

3.4 Digression on intermediate representations

Compiler writers often compile high-level code into an "intermediate representation," then compile *that* into machine code, instead of doing the compilation from source to machine code directly. This makes targeting multiple architectures easier, since you can reuse the source-code-to-IR compiler for every architecture. Some common IRs include:

1. *abstract syntax trees (ASTs):* tree data structure created by parsing source code.
2. *bytecode:* like from Java or Python

An IR is in static single assignment (SSA) form iff every variable is defined exactly once and every variable is declared before being used.

4 RISC

RISC stands for Reduced Instruction Set Computer (as opposed to CISC/Reduced Instruction Set Computer). The goal of RISC architectures is to reduce complexity of the processor's microarchitecture and the codegen portion of compilers that target the processor. RISC uses fixed-length instructions rather than complex variable-length instructions, which simplifies the fetch-decode-execute cycle and makes it faster because it can assume specific constraints.

One drawback of CISC is that the complexity enables a wide breadth of arbitrary code execution. By finding desirable sequences of bytes within a program (gadgets), exploit developers can hop around the memory without injecting too much of their own assembly. RISC is less susceptible to this issue, as fixed-length instructions mean you can make certain assumptions about the alignment of the instruction pointer. Examples of RISC architectures include:

- ARM
  - ARM is actually the most popular architecture in the world because it is used in a large number of embedded devices and phones.
- μ-ops in CISC architectures
  - At the level where instructions are actually executed, many CISC architectures use micro-ops (μ-ops).
  - CISC instructions act as a layer of abstraction over the μ-ops
  - This abstraction layer is highly complicated, and prone to bugs, leading to the development of microcode, which defines the layer of abstraction between CISC ops and internal μ-ops.

- RISC-V
- MIPS

Despite being highly ubiquitous, RISC is "dead" because a prevailing trend is to add as much specialized hardware and as many instructions as possible:

- FPUs (floating point units)
- GPUs (graphic processing units)
- cryptography instructions
- vector instructions
  - SIMD (single instruction multiple data)
  - MIMD (multiple instruction multiple data)
- SMT
  - Simultaneous multithreading
    - Developed by Susan Eggers and Dean Tullsen
  - Processor aware of threads
  - May die? due to security.
- FPGAs built into processors

Regularity is an important aspect of the machine code.

Processors implemented in systems like

- VHDL
- HLS (high level synthesis)
  - Can compile to FPGA (field programmable gate array)

One major architecture component in processor design is pipelining, which is equivalent to parallelizing an assembly line. One subset of this is called "instruction-level parallelism" or ILP. ILP works well when there isn't a lot of branching, but if there is, you need speculative execution to keep the pipeline full. If speculation fails, causing a re-fetch, that's known as a "bubble", which can also cause the pipeline to "run dry". Pipelining started with the Pentium Pro, where bubbles caused extremely high latency. It continued as a trend with longer and longer pipelines for a while, but Ahmdals law struck, and there's only so much you can parallelize a pipeline.

Moore's Law

As processors attained higher and higher density due to Moore's law, decisions had to be made about how to utilize the space on a die. For a while, a prevailing trend was to add more and more cache, but eventually there was diminishing returns. So, processor manufacturers started splitting processors into "cores", where they just put identical copies of a processor on a die with interconnects and cache. Now, the prevailing trend is to add more cores.

References

[1] Abhinav Jangda et al. "Not So Fast: Analyzing the Performance of {WebAssembly} vs. Native Code". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 107–120.

[2] Bobby Powers, John Vilk, and Emery D Berger. "Browsix: Bridging the gap between unix and the browser". In: *ACM SIGPLAN Notices* 52.4 (2017), pp. 253–266.

[3] John Vilk and Emery D Berger. "Doppio: breaking the browser language barrier". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pp. 508–518.