Abstrakte Maschinen Ausarbeitung - 2025S

2025-05-26 22:07

Contents

Aufbau von Interpretern

Aus dem Compilerbau-Skriptum:

> Ein **Interpreter** ist ein Programm, das Programme ausführen kann, die in seiner **Interpretersprache** I formuliert sind. Ein Interpreter verhält sich wie eine Maschine bzw. ein Prozessor. Man bezeichnet ihn daher auch als **virtuelle** oder **abstrakte Maschine**.

Zwischencodes - Intermediate Language (IL)

Compiler und Interpreter laufen durch verschieden Phasen (”passes”) (lexing, parsing, scope analysis, ...) - dabei wird üblicherweise der Code in verschiedene Datenstrukturen zum weiteren Verwenden gepackt (e.g. AST - abstract syntax **tree**). Da die semantic des ursprünglichen Codes (ein kontinuierlicher String) nicht verloren geht, kann man die neue Struktur als IL bezeichnen.

Meist bezeichnet man aber erst die letzteren Datenstrukturen als IL: Java Byte-Code, Microsoft IL, etc. Sprich: Von einer IL spricht man im Regelfall erst, sobald ein AST *linearisiert* wurde.

Prozessorarchitekturen

Ein Prozessor ”besteht” aus 2 verschiedenen Arten von Architekturen: **Befehlsarchitektur - instruction set architecture (ISA)** und die **Mikroarchitektur - microarchitecure**

ISA

References:

- ISA - Wikipedia

Was die ISA macht steckt bereits im Namen: **instruction set** architecture. Hier wird also definiert, welche Instruktionen eine CPU versteht (im weiteren Sinne braucht die CPU dafür dann auch natürlich entsprechende Register, memory access, etc.), für RISC-V gibt es hier z. B. `addi` **...** `li` **...** `ret` (Das ist natürlich nur assembly-code, was die ISA ”nichts” angeht, die ISA definiert hier wirklich nur die Befehle in bits)

Eine **RISC-V** CPU ist also jede beliebige CPU, welche die RISC-V ISA laut Spezifikation entsprechend implementiert. Das erklärt dann im weiteren auch leicht den Unterschied zwischen ISA und Microarchitecture.

CISC, RISC, MISC, OISC, ...

References:

- CISC - Wikipedia
- RISC - Wikipedia
- MISC - Wikipedia
- OISC - Wikipedia

Es gibt in Bezug zur ISA hier verschiedene ISA-”Gruppen” (”Design Philosophien”), z. B.:

- CISC: Complex instruction set computer
  - variable Befehlslänge, komplexe Adressierungsarten, komplexe Befehle
  - Ein Befehl kann viele kleine Operationen ausführen (e.g. String compare)
  - Beispiele:
    - VAX
    - 68K
    - x86
- RISC: Reduced instruction set computer
  - fixe Befehlslänge, wenig Adressierungsarten, ein Speicherzugriff pro Befehl, einfache Befehle, viele Register
  - Ein Befehl führt einen kleinen Task aus (zb *nicht* zwei loads in einem Befehl)
  - Sagt nichts über die **Anzahl** der Befehle aus, RISCV hat trotzdem viele verschiedene Befehle
  - Beispiele:
    - MIPS
    - Precision Architecture
    - Sparc
    - ARM
    - PowerPC
    - Alpha
- MISC: Minimal instruction set computer
  - Nur sehr wenige Befehle, meist für Mikroprozessoren
  - Sind oft als Stack-Maschinen implementiert, dadurch können die Befehle deutlich simpler und kürzer gestaltet werden, da Operanden einfach vom Stack gelesen werden.
- OISC: One instruction set computer
  - Auch bekannt als ***ULTIMATE REDUCED INSTRUCTION SET COMPUTER***
  - Nur eine einizige instruction (ja, nur eine)

- Die fixe Befehlslänge ist damit natürlich trivial (daher auch U-RISC)
- Siehe Instruction-Types für Beispiele von solchen instructions

Microarchitecture

References:

- Microarchitecture - Wikipedia

(Hat nichts mit Mikroprozessoren zu tun)

Die Microarchitecture ist jetzt die **konkrete** Implementation einer ISA in einer CPU. Wie bereits gesagt, können mehrere verschiedene CPUs dieselbe ISA implementieren - es gibt viele RISC-V CPUs, diese sind aber natürlich nicht alle ident: Der Unterschied zwischen denen ist hierbei die unterschiedliche Microarchitecture.

Ein simples Beispiel für einen Unterschied in der Microarchitektur: Es gibt verschiedene implementationen für einen n-bit-Volladdierer (zb Ripple-Carry-Adder und Carry-lookahead-Adder). Da sich diese nur in der Performance (und Preis...) Unterscheiden, kann man diese beliebig austauschen. Die ISA bleibt also weiterhin gleich, aber die Microarchitektur hat sich geändert!

(Moderne) Implementierungs-Techniken

References:

- Microarchitecture#Microarchitectural_concepts - Wikipedia

Pipelining Wenn ein Befehl ausgeführt wird (zb `addi x5, 3`) werden viele Komponenten angesprochen. Ein Befehl muss zuerst geladen werden, dann decodiert, dann wird irgendwann die ALU angesprochen und irgendwann wird das Ergebnis irgendwo abgespeichert.

Pipelining nutzt dieses Verhalten aus: Während eine Berechnung gerade in der ALU (also nachdem der Befehl schon gefetched und decodiert wurde) stattfindet, kann der nächste Befehl bereits ausgeführt werden.

Oft nimmt man hier anfänglich zum Lernen des Konzepts diese Pipeline-Stages:

- Fetch: nächsten Befehl laden
- Decode: den Befehl decodieren
- Execute: den Befehl ausführen (meist über eine Berechnung in der ALU)
- Write back: Ergebnis returnen (entweder in ein Register oder Memory)

Pipelining hängt meist stark mit den anderen Techniken zusammen (zb result forwarding).

Superscalar Pipelining erlaubt es, mehrere (potentiell voneinander abhängige) Instructions ”verzahnt” auszuführen. Superscalar beschreibt hierbei ein **gleichzeitiges** Ausführen von Instructions (also z.B. zwei Instructions auf einmal fetchen). Dafür benötigt man zum einen natürlich entsprechend mehr Hardware, zum anderen aber auch ein handeln von dependencies zwischen diesen instructions.

Cache (decoded instruction cache) Kleine caches auf der CPU für schnelleren Zugriff als auf main RAM, heutzutage meist ~2MB groß.

Result forwarding (bypass) References:

- Operand forwarding - Wikipedia

Theoretisch ist ein Ergebnis erst nach der `Write back` stage verfügbar. Allerdings ”kennt” man das Ergebnis ja schon bereits nach der `Execute` stage, wo es in der ALU ausgerechnet wurde.

Damit eine instruction also nicht eine stage länger warten muss, kann (simplified) ein extra Wire in der CPU ”platziert” werden, dass direkt vom output der ALU in den Input der ALU für die nächste Instruktion führt.

Result forwarding bezieht sich aber nicht nur auf die ALU / auf die Execute Stage, das ist implementationsabhängig.

Register renaming References:

- Register renaming - Wikipedia

Register können umbenannt werden um gewisse dependencies zwischen Instruktionen aufzulösen, was die Parallälität erhöht.

Branch prediction Beim Pipelining ging es darum, den nächsten Befehl so früh es geht zu starten während der vorherige noch abgearbeitet wird. Ist der vorherige jetzt allerdings ein (conditional-)jump Befehl, stellt sich die Frage, welchen Befehl man nun fetchen soll.

Dafür gibt es branch prediction (mit verschiedenen, immer komplexer werdenden Implementationen), welche anhand von dem vergangenen Verhalten des Programms versuchen zu vorhersagen, welcher branch genommen wird.

Falls richtig geraten wurde, hat man einen performance benefit, falls falsch eben einen performance hit (falsche instruction muss geflushed werden und andere gestartet).

Reservation stations, history buffers, future files References:

- Reservation Station - Wikipedia

- Reservation Station: Kümmert sich um das handeln von dependencies zwischen operands und checkt ob eine Komponente frei ist. Dadurch kann ”buffered” gefetched und decoded werden und erst verzögert dann die execute stage ausgeführt werden. (? - bin mir hier etwas unsicher)
- (TODO) History buffers: nichts dazu gefunden
- (TODO) Future files: nichts dazu gefunden

Threaded Code

TODO

Forth

TODO

Pascal P4

TODO

JVM

TODO

Microsoft IL

TODO

Registermaschinen

TODO

DalvikVM

TODO

Syntaxgesteuerte Editoren

TODO

Baummaschinen

TODO

Prologmaschinen

TODO

Funktionale Sprachen

TODO