Abstrakte Maschinen Ausarbeitung - 2025S

2025-05-29 01:26

Contents

Aufbau von Interpretern

Aus dem Compilerbau-Skriptum:

> Ein **Interpreter** ist ein Programm, das Programme ausführen kann, die in seiner **Interpretersprache** l 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 (MiA)**

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
- History buffer - Random Slides von einer Uni
- Future files - Random Slides von einer Uni

- 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. Das bedeutet dass das fetchen nicht stallen muss wenn eine Komponente gerade blockiert ist sondern so lange weiter fetchen kann bis die Reservation Station voll ist.
- Das ermöglicht eine gleichmäßigere Auslastung der Komponenten.
- History buffers: Speichert die Werte des Zielregisters von Instruktionen vor der ausführung und löscht diese falls die Instruktion die älterste ist und kein interrupt/trap aufgetreten ist.
- Future files: Ist ein seperates Register File welches mit Werten von ausgeführten Instruktionen befüllt wird. Das vermeidet dass neue Instruktionen Werte aus dem Reorder Buffer lesen müssen, sondern gleich die spekulativen Werte lesen können. Wenn eine exception auftritt muss das architekturelle (reale) Register file auf das Future file übertragen werden.

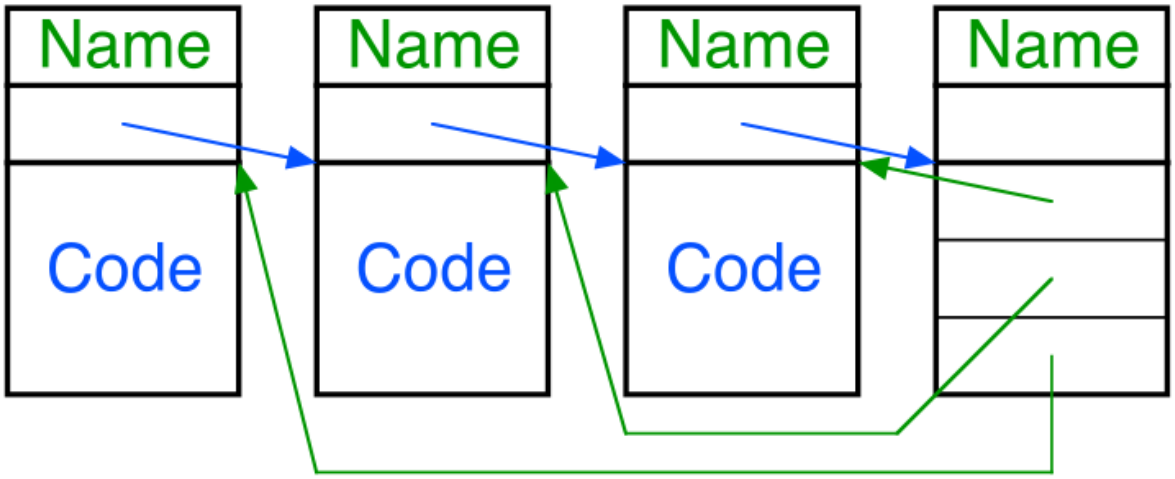
Threaded Code

References:

- ThreadedCode - Skriptum Slides
- Threaded Code - Wikipedia

Aus dem Skriptum:

> Die interne Darstellung eines threaded Interpreters ist eine Liste von Adressen vorher definierter interner Darstellungen (Unterprogrammen). Diese Darstellungen sind in einer linearen Liste aufgefädelt. Die einzelnen Elemente werden übersetzt. Die abstrakte Maschine ist meistens eine Stackmaschine.



Mit anderen Worten: Threaded Code besteht im Grunde aus zwei verschiedenen ”Code”-Stücken:

- Zwischencode
- Maschinencode

Der Maschinencode ist der ”tatsächliche” Code der ausgeführt wird. Das muss natürlich nicht klassisches (Dis-)assembly sein, sondern ein beliebiger Code der von einer gegebenen (abstrakten) Maschine ausgeführt werden kann.

Zwischencode ist lediglich für das managen des Maschinencodes zuständig. Auf Assembly-Ebene würde man hier zb eine Aneinanderreihung von jump instructions finden. Man kann sich das aber auch wie eine Main-Function vorstellen, die nacheinander function-calls macht.

Betrachtet man das Bild vom Skriptum, dann wären die ersten 3 Tabellen (mit `Code` geschrieben) die Maschinencode-Blöcke, und die 4te Tabelle enthält den Zwischencode-Block mit den Pointern zu den Maschinencode-Blöcken. Das wichtige Detail ist hier eben, dass die Maschinencode-Blöcke nichts voneinander wissen, sondern nur einen Pointer (`instruction counter` / `instruction pointer`) zurück auf den Zwischencode haben (genauer dann bei Threaded Code Arten beschrieben).

Es gibt jetzt aber mehrere Möglichkeiten, wie der Zwischencode und die ”return logic” vom Maschinencode implementiert ist. Als simples Beispiel könnte der Zwischencode entweder wirklich eins nach dem anderen eine Funktion aufrufen, oder es gibt eine globale Liste mit einem index-counter welcher direkt nach jedem Function-Call sofort erhöht wird und somit kann die ”return logic” vom Maschinen-Code selbst gleich den nächsten Maschinen-Code Block aufrufen.

Die Unterschiede hier geben dann je nach ISA und Microarch dann unterschiedlich gute performance.

Konkret unterscheiden wir zwischen diesen threaded code Arten:

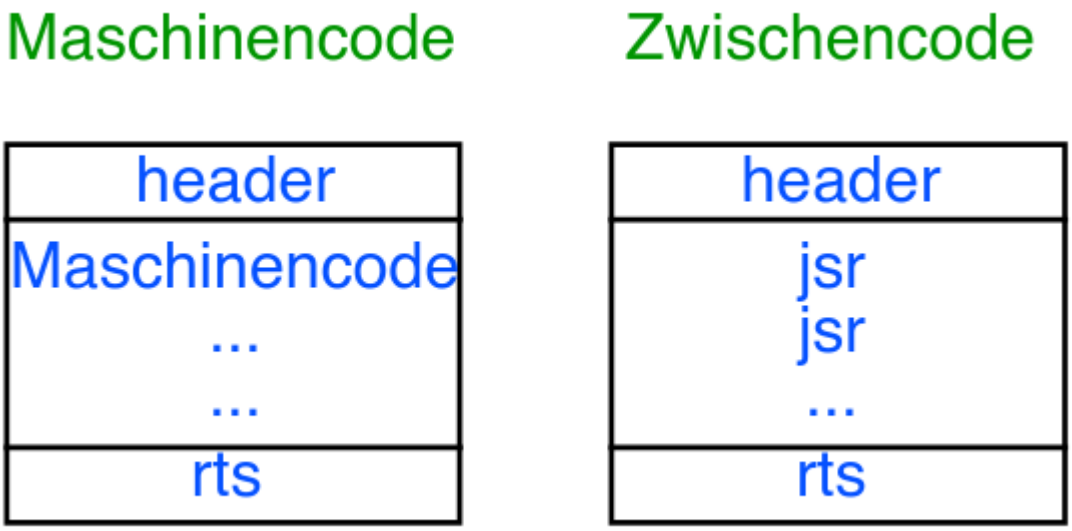
- subroutine threaded code
- direct threaded code
- indirect threaded code
- token threaded code
- indirect token threaded code

Threaded Code Arten

subroutine threaded code

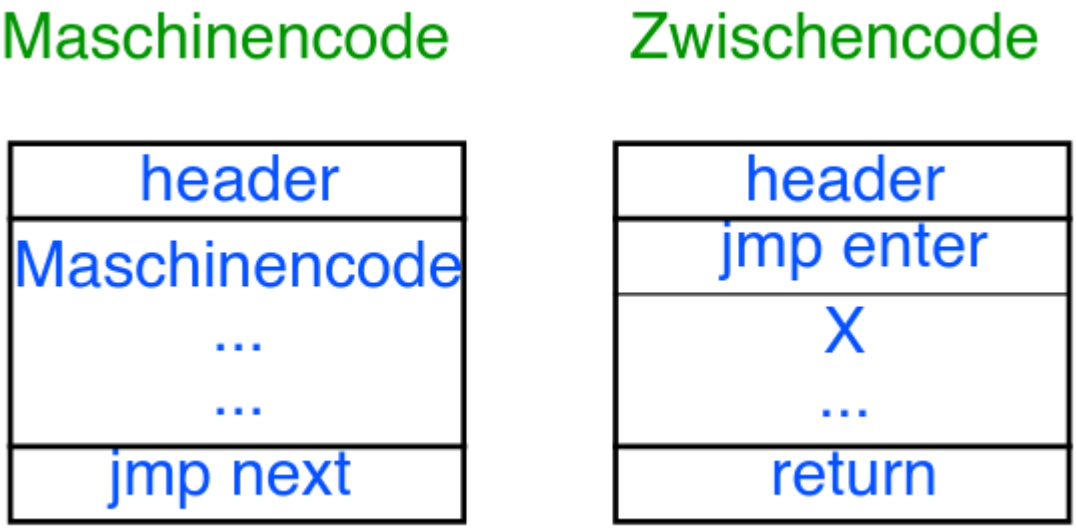
Die (meiner Meinung nach) einfachste Art von threaded code. Im Grunde genau die konkrete Abfolge von Function-Calls in einer main-function.

Auf Assembly-Ebene sieht das dann z. B. so aus:



Wie man sieht enthält der Zwischencode also wirklich subroutine calls, und der Maschinencode macht am Ende einfach ein return, wodurch der Zwischencode automatisch den nächsten subroutine call dann macht.

direct threaded code




```
enter:   push   ic
         move   ic = instr + jmp_len
next:    move   instr = (ic)+
         jmp    (instr)

return:  pop    ic
         jmp    next
```

Beim direct threaded code springt man nicht mehr die ganze Zeit zwischen Maschinencode und Zwischencode hin und her. Der Zwischencode ist zwar noch immer der main entrypoint, aber enhält jetzt nur noch mehr eine Liste von Adressen / Pointern zu den einzelnen Maschinencode-Blöcken (also *keine* call instructions mehr).

Stattdessen wird jetzt am Anfang nur ein instruction-counter (ic) initialisiert (i.e. ein pointer zu der Address-Liste), welcher einfach am Anfang der Address-Liste beginnt. Nachdem der erste (oder ein beliebiger) Maschinencode-Block fertig ausgeführt wurde, steht am Ende von diesem ein `jmp next`. Schaut man sich im Bild den Pseudo-Assembly-Code an sieht man, dass hier einfach der `ic` incremented und dann auf die Addresse der nächsten instruction `instr` gesprungen wird `jmp (instr)`.

Das `push ic` und `pop ic` ist die exit-logic vom gesamten ”Programm”.

Hier auch ein Code-Beispiel von Wikipedia:

```
#define PUSH(x) (*sp++ = (x))
#define POP() (*--sp)
start:
  ip = &thread  // ip points to &pushA (which points to the first instruction of pushA)
  jump *ip++  // send control to first instruction of pushA and advance ip to &pushB
thread:
  &pushA
```

```
  &pushB
  &add
   ...
pushA:
  PUSH(A)
  jump *ip++ // send control where ip says to (i.e. to pushB) and advance ip
pushB:
  PUSH(B)
  jump *ip++
add:
  result = POP() + POP()
  PUSH(result)
  jump *ip++
```

indirect threaded code

Der Unterschied zwischen `direct` und `indirect` (unten dann auch bei `token`) threaded code ist, dass es eine weitere *Indirektion*, also ein extra Pointer gibt. Am besten vergleicht man den direct threaded code von oben mit den folgenden äquivalenten indirect threaded code:

```
start:
  ip = &thread  // points to '&i_pushA'
  jump *(*ip)  // follow pointers to 1st instruction of 'push', DO NOT advance ip yet
thread:
  &i_pushA
  &i_pushB
  &i_add
   ...
i_pushA:
  &push
  &A
i_pushB:
  &push
  &B
i_add:
  &add
push:
  // look 1 past start of indirect block for operand address
  *sp++ = *(*ip + 1)
  // advance ip in thread, jump through next indirect block to next subroutine
  jump *(*++ip)
add:
  addend1 = *--sp
  addend2 = *--sp
  *sp++ = addend1 + addend2
  jump *(*++ip)
```

Man sieht, dass hier eine extra indirection ist (deswegen auch `jump *(*++ip)`, mit 2mal deref `*`). Performance ist dadurch natürlich zwangsweise schlechter als bei direct threaded code, da schließlich eine pointer-deref mehr berechnet werden muss.

Der Vorteil ist allerdings ein kompakterer Code: Wenn beispielsweise `push` eine Implementation von 10 Zeilen hätte, dann würde das beim **direct** threaded code insgesamt **20** Zeilen hinzufügen (2 * 10), bei **indirect** aber nur 10, da die Implementation "ausgelagert" wurde. Das funktioniert deswegen, da man jetzt Maschinencode selbst Parameter handeln kann, das sieht man z. B. bei `i_pushA`:

```
i_pushA:
    &push
    &A
```

`&A` wird hier ausschließlich als Parameter für `push` verwendet. Da `&push` am Anfang von `i_pushA` steht, wird auch dorthin gesprungen: `*ip` → `i_pushA`, `**ip` → `*i_pushA` → `push`. Bedeutet es wird nie

(sinnvoller weise) auf die Adresse von `A` gesprungen, sondern nur in der Implementation von `push` dann selbst dereferenziert und als **Wert** (in dem Fall für den Stack) verwendet:

```
push:
    *sp++ = *(*ip + 1)
```

`*ip + 1 → i_pushA + 1 → &A, *(*ip + 1) → *(i_pushA + 1) → *(&A) → A`

token threaded code

Im Grunde noch eine zweite Art von indirection. Jetzt geht es im Grunde um folgenden Fall:

```
thread:
    &pushA
    &pushA
    &add
```

In diesem Code wurde 2mal `pushA` referenziert. Ein Pointer hat natürlich selbst auch eine gewisse Größe (z. B. 8byte für 64-Bit Systeme). Wir verbrauchen also in dem code-snippet oben insgesamt `3 * 8 = 3byte` (zur runtime). ***DAS IST VIEL ZU VIEL OMG***

Als Lösung für dieses tragische Problem kann man also nun eine extra Tabelle erstellen, welche einfach alle Adressen aller verwendeten Funktionen auflistet. Greift man nun auf diese Tabelle per ”index” zu, kann man sich demnach Platz sparen, jenachdem wie groß der Datentyp für den Index ist.

Bsp: Index hat 1bit Größe (i.e. 2 Einträge in der Tabelle möglich)

```
thread:
    1
    1
    0

table:
    &add
    &pushA
```

Hiermit verbrauchen wir also nur noch mehr `(2 * 8) + (3 * 1) = 19bit < 3 byte`. Eine unfassbare `~20%` runtime code-size reduction.

Der Rest bleibt im Grunde gleich, ein vollständiges Bsp wäre:

```
start:
  vpc = &thread
dispatch:
  // Convert the next bytecode operation
  // to a pointer to machine code that implements it
  addr = decode(&vpc)
  // Any inter-instruction operations are performed here
  // (e.g. updating global state, event processing, etc)
  jump addr
CODE_PTR decode(BYTE_CODE **p) {
  // In a more complex encoding,
  // there may be multiple tables to choose between or control/mode flags
  return table[*(*p)++];
}
thread:  /* Contains bytecode, not machine addresses.  Hence it is more compact. */
  1 /*pushA*/
  2 /*pushB*/
  0 /*add*/
table:
  &add    /* table[0] = address of machine code that implements bytecode 0 */
  &pushA  /* table[1] ... */
  &pushB  /* table[2] ... */
pushA:
  *sp++ = A
  jump dispatch
```

```
pushB:
  *sp++ = B
  jump dispatch
add:
  addend1 = *--sp
  addend2 = *--sp
  *sp++ = addend1 + addend2
  jump dispatch
```

Wir haben also noch immer einen instruction pointer, allerdings zeigt dieser jetzt nicht mehr auf ”echte” Adressen, sondern auf Indexe → und diese können wir gemeinsam mit der Tabelle verwenden um dann doch den Pointer vom Maschinencode zu bekommen.

indirect token threaded code

Einfach eine Kombination aus token threaded code und der vorherigen Beschreibung von `indirect` threaded code.

- `token` → Reduziert size beim referenzieren im call-thread
- `indirect` → Reduziert size indem Maschinencode nicht (weniger) dupliziert werden muss

Threaded Code Vergleich

Äquivalenter Code kann auf die verschiedenen threading Arten compiled werden. Allerdings macht es ja einen Unterschied auf welcher MiA ein Code ausgeführt wird. Diese Unterschiede in der MiA können dann sogar dazu führen, dass derselbe Code mit zb:

- `direct threaded` **schneller** als `subroutine threaded` auf Maschine `A` ist, aber
- `direct threaded` **langsamer** als `subroutine threaded` auf Maschine `B` ist

Man kann daher nicht per-se sagen, welche Threading-Art schneller ist (mit Außname, dass indirect natürlich immer langsamer als direct threading ist).

TODO: Brauchen wir hier genauer die Beispiele aus den Folien mit den konkreten CPUs wissen?

Forth

TODO

Pascal P4

TODO

JVM

TODO

Microsoft IL

TODO

Registermaschinen

TODO

DalvikVM

TODO

Syntaxgesteuerte Editoren

TODO

Baummaschinen

TODO

Prologmaschinen

TODO

Funktionale Sprachen

TODO