Die Geschichte von JAVA

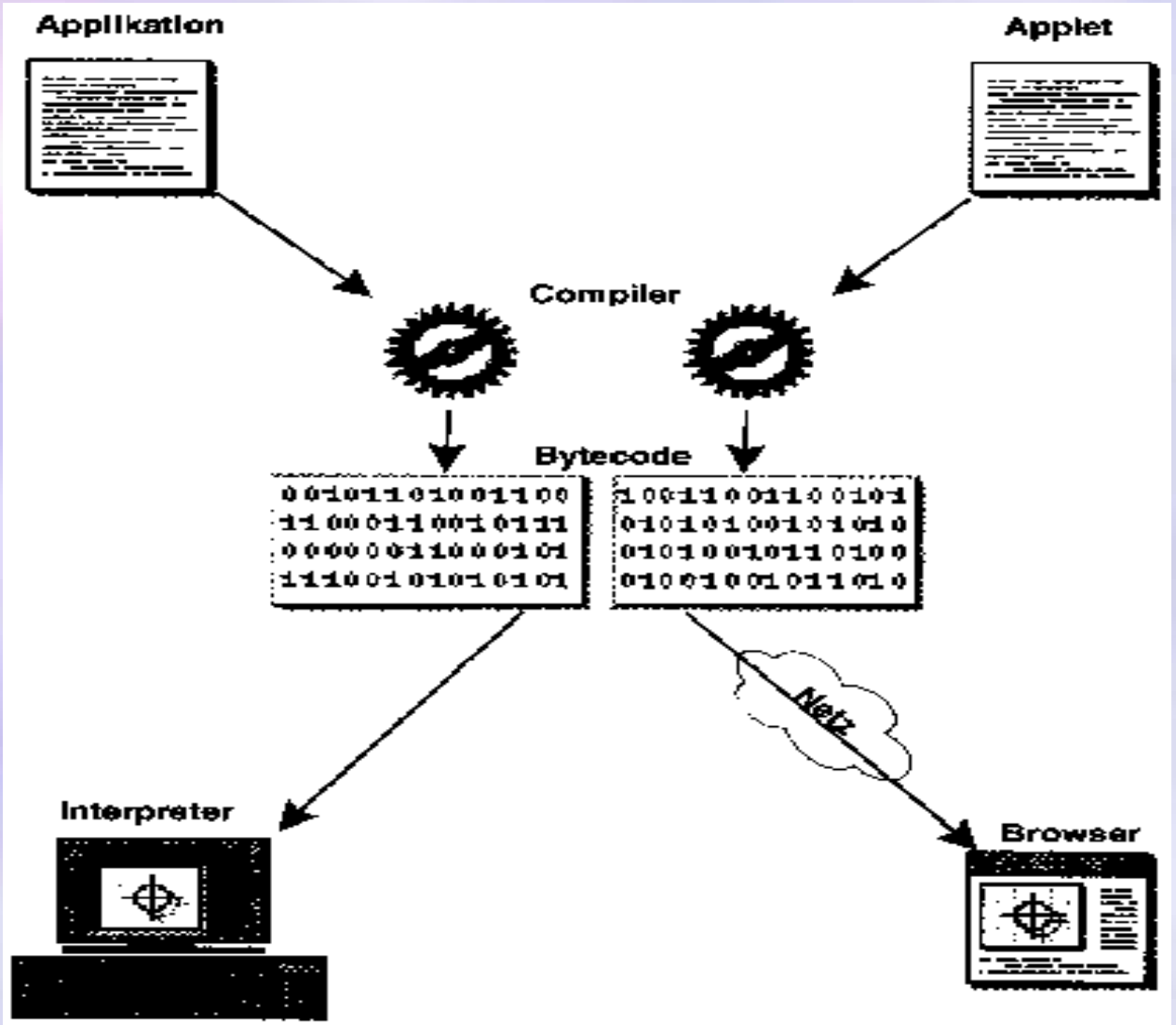
- 1991 Forschung von **Sun Microsystems** “Green Project”
- kleine Programme über TV Kanäle, interaktives Fernsehen
- dieser Markt entwickelte sich zu langsam
- Der Prototyp OAK (Object Application Kernel) viel somit in die Grundlagenforschung
- Ende 1995 erschien eine erste JAVA-Version

Was ist JAVA?

- Vereinigung mehrerer Sprachen, wie z.B C, C++, Smalltalk
- dabei wurde auf unötige Funktionalität verzichet, für das Zielgebiet eher unnötig
- plattformunabhängig, dank der **VM**, der Virtual Machine

Das Prinzip von JAVA

- Der JAVA-Compiler erzeugt **Bytecode**, ein hardware- und betriebsystem unabhängiger binärer Code, der nicht maschinenspezifisch ist, sondern von einer virtuellen Maschine, der JVM ausgeführt wird.
- Bytecode wird in ".class" Dateien gespeichert.
- unterstützt "threads" und "garbage-collection"


Applikation
Applet
Compiler
Bytecode
Netz
Interpreter
Browser

Die Virtuelle Maschine von Java besteht im Wesentlichen aus Folgendem:

- **Class Ladesystem** lädt die Class-Dateien und initialisiert Klassen für die Ausführung.
- **Method Area** hier wird der Byte-Code der Java-Klassen abgelegt.
- **Heap** Allgemeiner Speicher für die Datenstrukturen aller Objekte.
- **Stack** Speicher für lokale Ausführung der Methoden.

Die Class-Datei muss von allen JVM-en in gleicher Weise interpretiert werden. Es enthält unter anderen:

- **Magic** Wort 0xCAFFEBABE.
- **Constant Pool Area** zur Definition aller konstanten Werte, die von den Methoden gebraucht werden.
- **Fields** sind veränderliche Attribute der Instanzen bzw. der Klasse selbst.
- **Methods** - Methoden mit Byte-Code und ihren Attributen.
- weiteres ...

Datentypen in der JVM.

Datentyp	bit-Breite	impl.	Nutzung
byte	8 bit	32 bit	-128..127
short	16 bit	32 bit	-32768..32767
int	32 bit	32 bit	$-2^{31}..2^{31}-1$
long	64 bit	64 bit	$-2^{63}..2^{63}-1$
float	32 bit	32 bit	IEEE 754 single-precision
double	64 bit	64 bit	IEEE 754 double-precision
char	16 bit	32 bit	Unicode Zeichen
boolean	1 bit	32 bit	Boolscher Wert
returnValue		32 bit	Rücksprungaddresse
reference		32 bit	Zeiger auf einen Objekt

Befehle in der JVM.

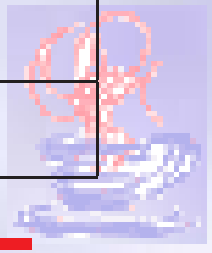
- Einfach (RISC)

Kurzel	Operanden	Opcode	Funktion
iadd		0x60	Ganzzahl Addition
iload	8 bit Offset	0x15	lade lokale Variable(Ganzzahl)
fload	8 bit Offset	0x17	lade lokale Variable(Fließkomma.)
bipush	8 bit Konst.	0x10	lege eine Byte-Konstante ab
ifeg	16 bit Offset	0x99	Springe, wenn 0

- Komplex (CISC)

  lookupswitch - Java “switch” Anweisung.

byte 1	byte 2	byte 3	byte 4
opc. 0xAB	0..3 bytes padding		
default - standard Sprungaddresse			
Anzahl der Vergleichswerte			
Vergleichswert 1			
Sprungaddresse für Wert 1			
Vergleichswert 2			
Sprungaddresse für Wert 2			
...			



- Java Spezifisch

Kurzel	Operanden	Opcode	Funktion
instanceof	Klasse 16b.	0xC1	Klassenzugehörigkeit
athrow		0xBF	Ausnahmebehandlung
new	Klasse 16b.	0xBB	ein neues Objekt
newarray	atype 8b.	0xBC	eine neue Tabelle
monitorenter		0xC2	Erwerben der Kontrolle über ein Objekt
monitorexit		0xC3	Abgeben der Kontrolle

Vor- und Nachteile von Interpreter und JIT

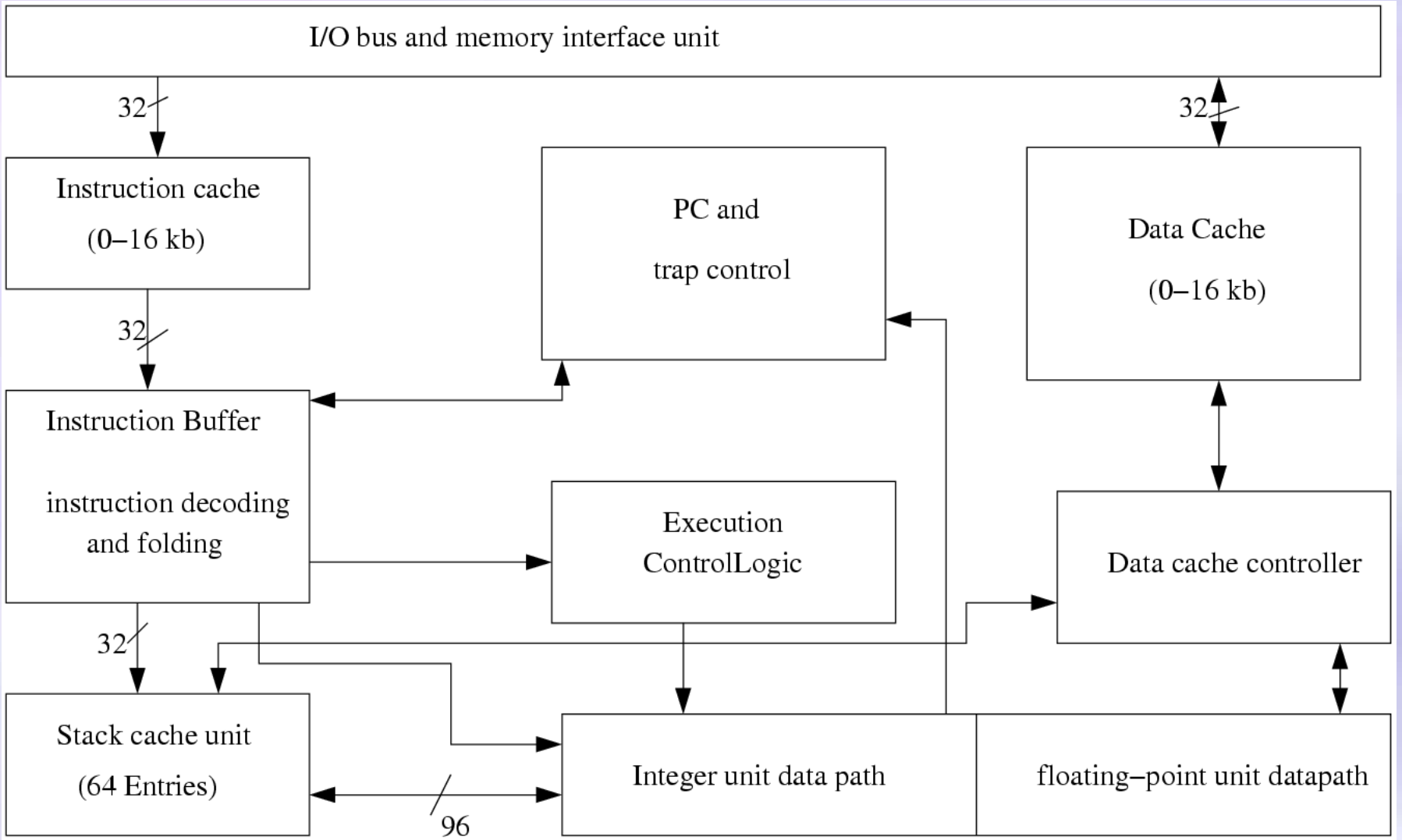
	Vorteile	Nachteile
Interpreter	leicht zu implementieren, wenig speicherfressend,	performance Verlust durch Emulation
JIT	wesentlich höhere Geschwindigkeit,	benötigt viel Speicher,verzögertes Ausführen von Programmen

Deshalb in Hardware!

1. Vereinigt alle Vorteile des Interpreters mit denen der VM, eleminiert die Nachteile
2. Der Garbage Collector der VM, sowie “thread synchronization” in einer Hardwareimplementation stigern weiter die Performance
3. Ideal für “embedded systems”(Handys, Palms, Mikrowellen, Fernseher , Taschenrechnern, Hörgeräte, Videokameras,...u.v.a), da relativ klein, optimiert und wenig speicherfressend, bei hoher Leistung
4. gutes Preis/Leistungsverhältnis
5. traditionell wurden eingebettete Prozessoren in Assembler programmiert ... der Vorteil eines JAVA-Prozessors liegt hier wohl klar auf der Hand!

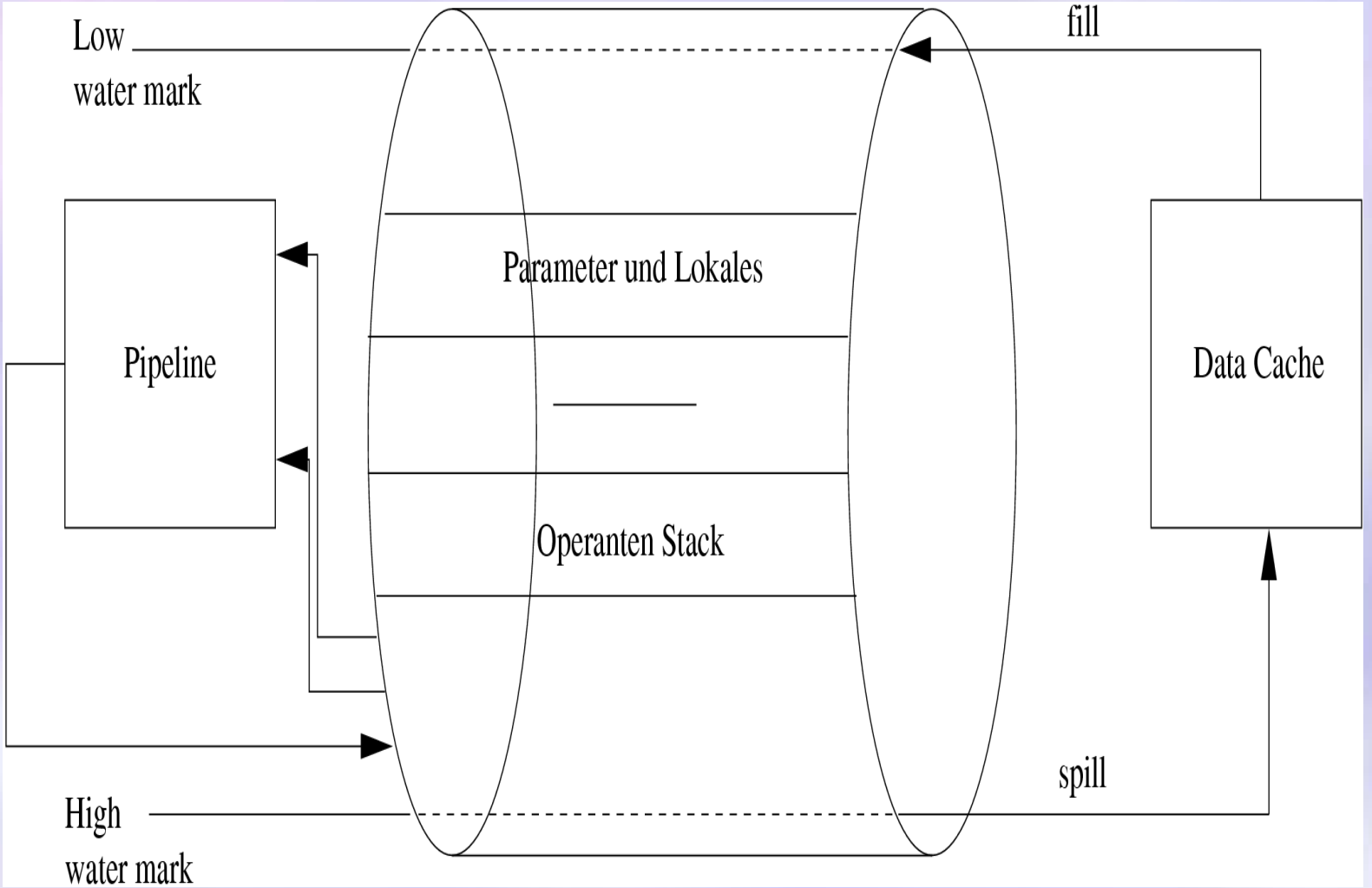
Verschiedene Prozessoren- picoJava I Hierbei handelt es sich um ein Core-Design der Firma SUN Microsystems

- klein, verschiedene Cache-Größen,wahlweise FPU
- ein Design, welches sich bestens dem Bedarf anpasst
- RISC-Style Architektur
- aufwendigere Instruktionen werden durch microcode emuliert
- die meißten Befehle benötigen 1-3 Taktzyklen
- zusätzliche Befehle(**extended Instructionset**) zur VM (anstatt library)


I/O bus and memory interface unit
32
32
Instruction cache
(0–16 kb)
PC and
trap control
Data Cache
(0–16 kb)
32
Instruction Buffer
instruction decoding
and folding
Execution
ControlLogic
Data cache controller
32
Stack cache unit
(64 Entries)
96
Integer unit data path
floating–point unit datapath

Front end

- 12-byte instruction buffer trennt den instruction cache vom Rest der pipeline
- Prozessor kann 4 bytes gleichzeitig schreiben, aber 5 bytes aufeinmal lesen.
- mehrere Instruktionen / Takt (durchschnittlich 1.8 bytes/Instruktion
- kleinere Instr. als 5 byte können "gefaltet" werden. ($\rightarrow$ folding)
- keine Sprungvorhersage-Einheit, hätte schlechtes Preis/Leistungsverhältnis, höhere Stromaufnahme


Low
water mark
fill
Parameter und Lokales
Pipeline
Data Cache
Operanten Stack
spill
High
water mark

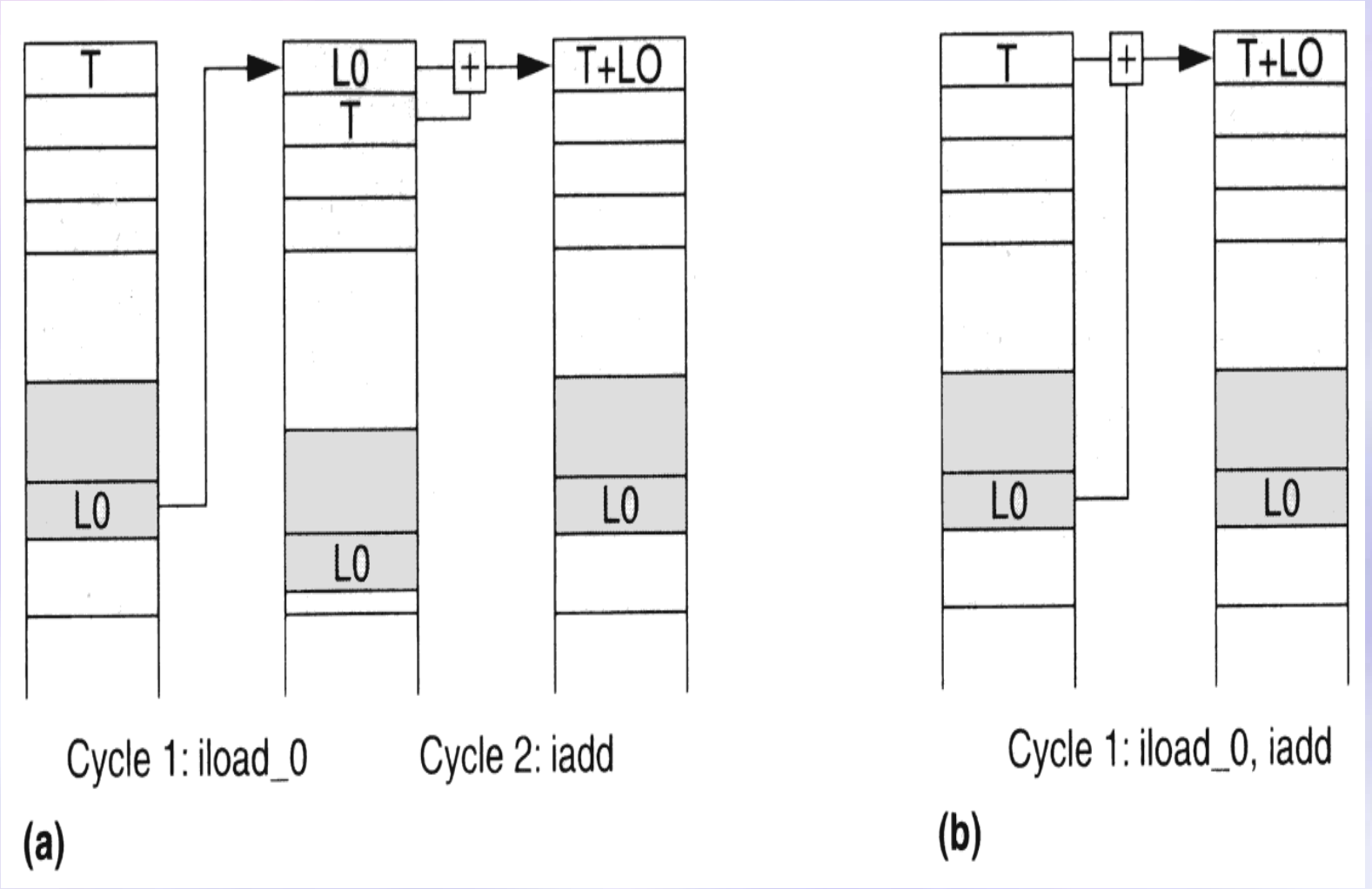
Stack cache

- 64 Einträge können gecached werden
- als Register implementiert, wie Kreispuffer verwaltet
- Stack wächst abwärts. Draufpushen eines Elementes dec. den Pointer
- Dieser kann “überspringen”.
- Um Gleichgewicht zu halten, alte Daten in den Cache, und zurück

folding

- beschleunigt JAVA Bytecode
- ohne folding: eine lokale Variable wird auf den Stack gepushed,anschliessend eine Operation ausgeführt.
- mit folding: Der instruction-decoder erkennt den Zusammenhang und führt diese beiden Instruktionen aufeinmal aus, spart somit einen Takt.
- normalerweise ist die lokale Variable bereits im Stack. Falls nicht, wird "folding" unterdrückt, ebenfalls bei ein-Schritt-Instruktionen.
- Einfache Konstanten, können "gefaltet" werden, weil sie früh beim Deocdieren erkannt werden. $\Rightarrow$ direkt zur jew. Instruktion
- ca. 15 % weniger Instruktionen nötig, durch "folding"

Folding-Schema



Data Cache

- 0-16kb, optional
- “two-way, set-associative, write-back cache”
- DataPath zwischen Cache und Pipeline ist 32bit breit
- “line_zero instruction” verbessert performance,weniger bus-traffic

FPU

- single und double precision FP Arithmetik (IEEE 754)
- Rundung: “round-to-nearest” für Ergebnisse und “round-towards-zero” für $float \rightarrow int$
- nur eine Instruktion zur Zeit.
- kompakte FPU durch “overlapping”
- eine Instruktion benötigt im Schnitt 3 Zyklen

Speicher und I/O Schnittstelle

- Speicher Interface kompatibel zu SDRAM,EDO,SRAM, DRAM,FLASH,..
- I/O unterstützt PCI, USB, PCMCIA,..
- 32bit Speicherbus, durchexterne Logik kanndieses leicht verändert werden.

Pipeline

Holen	Dekodieren	Ausführen Cachen	Zurück-schreiben
4-byte cache lines in instruction-buffer	Dekodieren von bis zu 2 Instruktionen (Folding logic)	ein bis mehrere Zyklen zum Ausführen	Ergebnisse zurückauf den operand(en) stack

Performance

Methode	System	Javac	Raytraycer
Native	picoJava-I	15.2	19.6
JIT	Pentium	2.9	3.9
	486	2.6	2.3
Interpreter	Pentium	1.3	1.5
	486	1.0	1.0