Strojový kód a data

4. Přednáška

ISA

J. Buček, R. Lórencz

Obsah přednášky

- Násobení a dělení v počítači
- Základní cyklus počítače
- Charakteristika třech základní typů ISA (střadačová, zásobníková a s univerzálními registry)
- Střadačově orientovaná ISA s absolutní a indexovou adresací
- Charakteristika dnešních střadačově orientovaných ISA (mikrořadiče atd.)
- Charakteristika zásobníkově orientované ISA (příklady, T800, JVM, .NET atd.)

Násobení v počítači (1)

$C = A \cdot B = 14 \cdot 11 = ?$

$\Rightarrow$ Sečteme postupně, posouváme

```
A=          1 1 1 0
B=          1 0 1 1
-----------------------
            1 1 1 0
          1 1 1 0
        0 0 0 0
      1 1 1 0
-----------------------
C=  1 0 0 1 1 0 1 0
```

```
A=          1 1 1 0
B=          1 0 1 1
-----------------------
C1=         0 0 0 0
          + 1 1 1 0
-----------------------
            0 1 1 1 | 0
        +   1 1 1 0
-----------------------
            1 0 1 0 | 1
      +     0 0 0 0
-----------------------
            0 1 0 1 | 0
    +       1 1 1 0
-----------------------
C1= 1 0 0 1 | 1
            C0= 1 0 1 0
```

Sečíst 4 čísla najednou? Neumíme!

„Shift – add“

Násobení v počítači (2)

nezáporná celá čísla, dvojková soustava

Máme spočítat C = A*B. Mějme registry A, B, C1, C0

1. Vynuluj C1 (C1 = 0)
2. Posuň B doprava
3. Vypadla 1?
   - ANO C1 = C1 + A
   - NE nic
4. Posuň C1 doprava, přitom zasuň přenos ze sčítání
5. Posuň C0 doprava, přitom zasuň, co vypadlo z C1
6. Všechny bity B zpracovány?
   - NE jdi na 2.
   - ANO výsledek je C = (C1, C0). Konec.

Kolik bitů pro A, B, C? Proč máme C1, C0? Proč nemusíme nulovat C0?

Dělení v počítači (1)

A : B = 27 : 5 = ?, zbytek ?

```
  1 1 0 1 1 : 0 0 1 0 1 =     1 0 1
- 1 0 1
-------
    1 1 1
-   1 0 1
---------
    0 1 0
```

Kde začneme odčítat?

```
  0 0 0 0 1 1 0 1 1 : 0 0 1 0 1 = 0 0 1 0 1
- 0 0 1 0 1
-----------
  - 0 0 1 0 1     ↓ ↓
    - 0 0 1 0 1
    -----------
      0 0 0 0 1 1 1
    -   0 0 1 0 1
    -------------
        - 0 0 1 0 1
        -----------
          0 0 0 1 0
```

			A
	B=	0 0 1 0 1	
0	R=	0 0 0 0 1	1 0 1 1
0	R=	0 0 0 1 1	0 1 1 0
1	R=	0 0 1 1 0	1 1 0 0
	R-B=	0 0 0 0 1	
0	R=	0 0 0 1 1	1 0 0 0
1	R=	0 0 1 1 1	0 0 0 0
	R-B=	0 0 0 1 0	

Dělení v počítači (2)

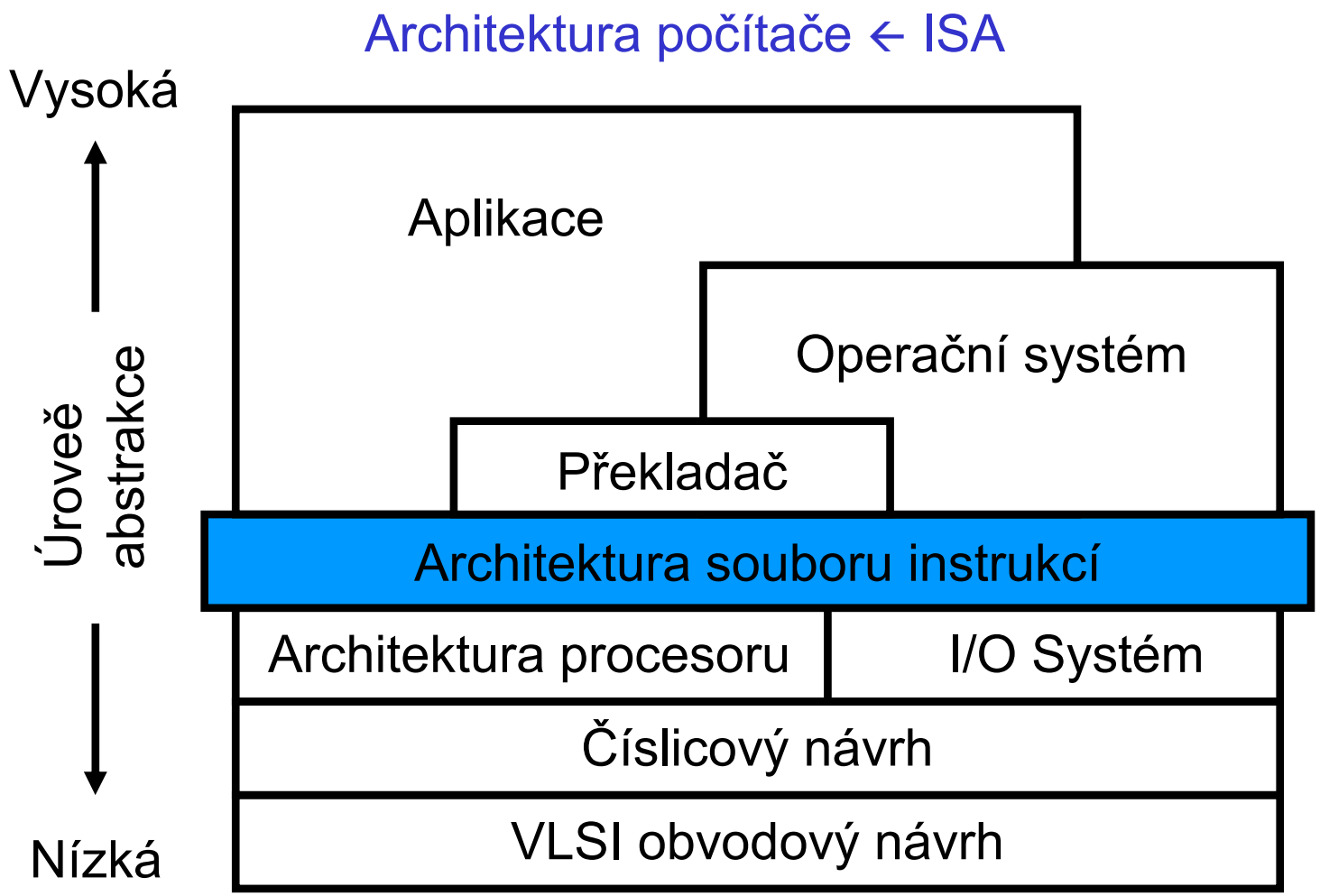
nezáporná celá čísla, dvojková soustava

Máme spočítat A : B = Q, zb. R. Mějme registry A, B, Q, R

1. Vynuluj R (R = 0)
2. Posuň A doleva
3. Posuň R doleva, přitom zasuň, co vypadlo z A
4. Porovnej: R >= B?
   - ANO — R = R – B, bit podílu = 1
   - NE — bit podílu = 0
5. Posuň Q doleva, přitom zasuň bit podílu z 4.
6. Všechny bity A zpracovány?
   - NE — jdi na 2.
   - ANO — podíl je Q, zbytek je R. Konec.

Kolik bitů pro A, B, Q, R? Proč nemusíme nulovat Q? Jaký je rozdíl mezi porovnáním a odečtením? **Více viz X36JPO (příští sem.), X36ARI (mag.)**

Architektura souboru instrukcí - ISA opakování (1)



Architektura souboru instrukcí - ISA opakování (2)

Zahrnuje

- Typy a formáty instrukcí, instrukční set
- Datové typy, kódovaní a reprezentace, způsob uložení dat v paměti
- Módy adresování paměti a přístup do paměti dat a instrukci
- Mimořádní stavy

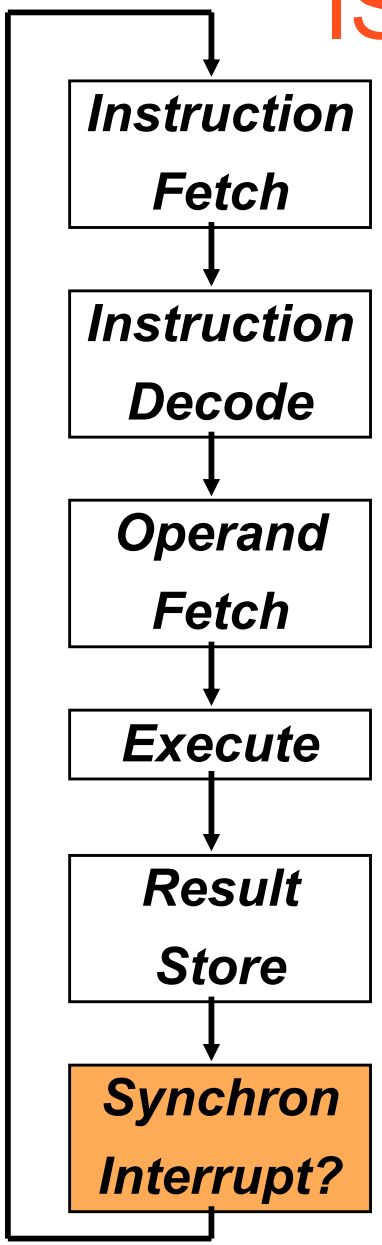
Umožňuje

- Abstrakce (výhoda – různé implementace stejné architektury)
- Definice rozhraní mezi nízko-úrovňovým SW a HW
- Standardizuje instrukce, bitové vzory strojového jazyka

Vlivy na vývoj architektur souboru instrukcí (ISA)

- Technologie výroby procesoru
- Operační systém
- Aplikace
- Programovací jazyky

ISA: Co musí být definováno? (3)



Kódování instrukcí

– Jak se instrukce dekóduje?

Umístění operandů

– Kolik explicitních operandů je v instrukci pro ALU?

– Jak jsou operandy umístěny v paměti?

– Který operand může být v paměti?

Typy dat a velikosti operandů

Operace v ISA

– Které jsou podporovány ?

Umístění výsledku

Výběr další instrukce

– Skoky, volání podprogramů, návraty

ISA: Základní třídy (4)

Akumulátorově orientovaná ISA (1 registr):

1 operand	ADD A	acc ← acc + mem[A]
	ADD (A + IX)	acc ← acc + mem[A + IX] IX je **indexovací registr**

Zásobníkově orientovaná ISA

0 operandů	ADD	stack(top –1) ← stack(top) + stack(top –1) top – –

ISA orientovaná na univerzální registry (reg. pro všeobecné použití) (GPR = General Purpose Registers)**:**

2 operandy	ADD A B	EA(A) ← EA(A) + EA(B)
3 operandy	ADD A B C	EA(A) ← EA(B) + EA(C)

EA … **Efektivní adresa** (určuje registr, nebo operand v paměti)

Akumulátorově orientovaná ISA s absolutní adresací

- **nejstarší ISA (1949-60) – vyvinula se z kalkulaček**

LOAD A	acc ← mem[A]
STORE A	mem[A] ← acc
ADD A	acc ← acc + mem[A]
SUB A	acc ← acc – mem[A]
…	
SHIFT LEFT	acc ← 2 x acc
SHIFT RIGHT	acc ← acc / 2
JUMP A	PC ← A
JGE A	if (0 ≤ acc) then PC ← A
LOAD ADDR X	načtení adresy operandu X do acc
STORE ADDR X	uložení adresy operandu X z acc

Typicky méně než 24 instrukcí! Hardware byl velmi drahý.

Akumulátorově orientovaná ISA s absolutní adresací

Práce s polem: sebemodifikující program

$C_I \leftarrow A_I + B_I$ $\quad - N \leq I < 0$

```
LOOP:   LOAD I
        JGE DONE
        ADD ONE
        STORE I
F1:     LOAD A
F2:     ADD B
F3:     STORE C
        LOAD ADDR F1
        ADD ONE
        STORE ADDR F1
        LOAD ADDR F2
        ADD ONE
        STORE ADDR F2
        LOAD ADDR F3
        ADD ONE
        STORE ADDR F3
        JUMP LOOP
DONE:   HLT
```

Potřebuje společnou paměť pro program a data ⇒ von Neuman.

Všechny proměnné uloženy v paměti.
Konstanta “1” je také v paměti.

Jedna iterace :
17 instrukcí (14 režie cyklu)
10 načtení operandů (8 režie)
5 uložení (4 režie)

Veliká režie cyklu
(zejm. modifikace adres) !

Akumulátorově orientovaná ISA s indexovacími registry

Indexovací registr – jeden nebo více registrů pro zjednodušení adresace paměti

Úprava předchozích instrukcí :

LOAD	A, IX	acc ← mem[A + IX]
ADD	A, IX	acc ← acc + mem[A + IX]
atd.		

Nové instrukce pro manipulaci s indexovacími registry

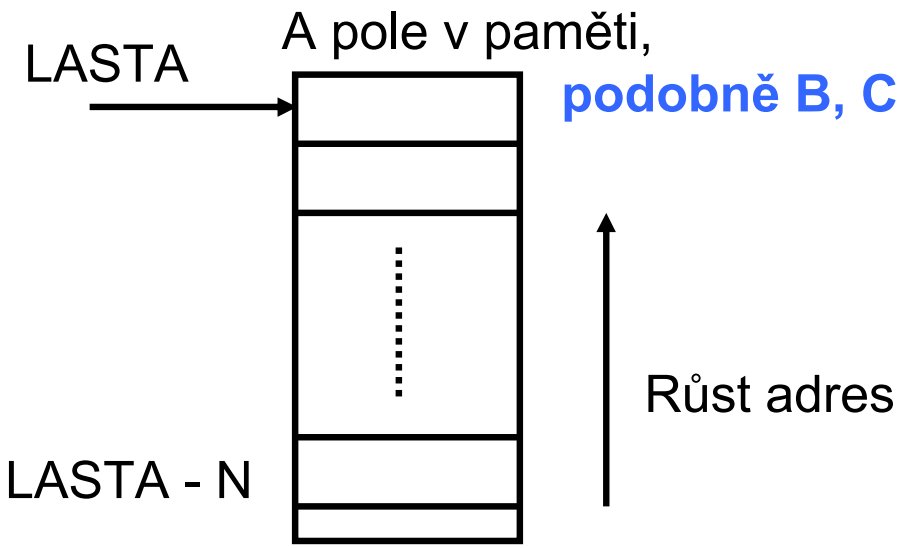
JZi	A,IX	if (IX = 0) then PC ← A
		else PC ← PC + 1, IX ← IX + 1
LOADi	A,IX	IX ← mem[A]

Cykly s indexovacími registry

CI ← AI + BI

- N ≤ I < 0

```
        LOADi  "–N", IX
LOOP:   JZi  DONE, IX
        LOAD LASTA, IX
        ADD LASTB, IX
        STORE LASTC, IX
        JUMP LOOP
DONE:   HLT
```



(Konstanta –N je uložena v paměti na adrese "–N"; IX obsahuje proměnnou I)

Program nemění sám sebe. Větší efektivita je zřejmá.
Cenou jsou delší instrukce (1-2 bity k určení indexregistru).

Akumulátorově orientovaná ISA - dnes

Z indexovacích registrů se vyvinuly **speciální registry pro nepřímou adresaci**, zvláštním typem je **stack pointer** (ukazatel na vrchol zásobníku).

Procesory také zahrnují **pracovní registry** (tzv. ***zápisníková paměť***). Toto pole snižuje četnost přístupů do paměti.

Implicitním operandem ALU je **vždy akumulátor** (druhý operand může být v registrech nebo v paměti).

Použita v prvních mikroprocesorech: 4004, 8008, 8080, 6502,…

Dnes použita v některých mikrokontrolérech: 8051, 68HC11, 68HC05, DOP …

ISA X86 představena jako „ISA s více akumulátory…“
=> s nástupem i386 upravena na GPR.

Akumulátorově orientovaná ISA - shrnutí

Výhody:

– jednoduchý HW
– minimální vnitřní stav procesoru ⇒ rychlé přepínání kontextu
– krátké instrukce (záleží na typu druhého operandu)
– jednoduché dekódování instrukcí

Nevýhody:

– častá komunikace s pamětí (dnes problém)
– omezený paralelismus mezi instrukcemi

Není náhodou, že tento typ ISA byl populární v 50. a 70. letech - hardware byl drahý, paměť byla rychlejší než CPU.

Zásobníkově orientovaná ISA (1)

Využití „zásobníku“ při vykonávání programu :

- Vyhodnocení výrazů
- Vnořená volání podprogramů
  - předávání návratové adresy a parametrů
  - lokální proměnné

Známým příkladem byl Burroughs B5000, 1960
- počítač navržený k podpoře jazyka **ALGOL 60.**
- vyhodnocení výrazů podporoval **hardwarový zásobník**

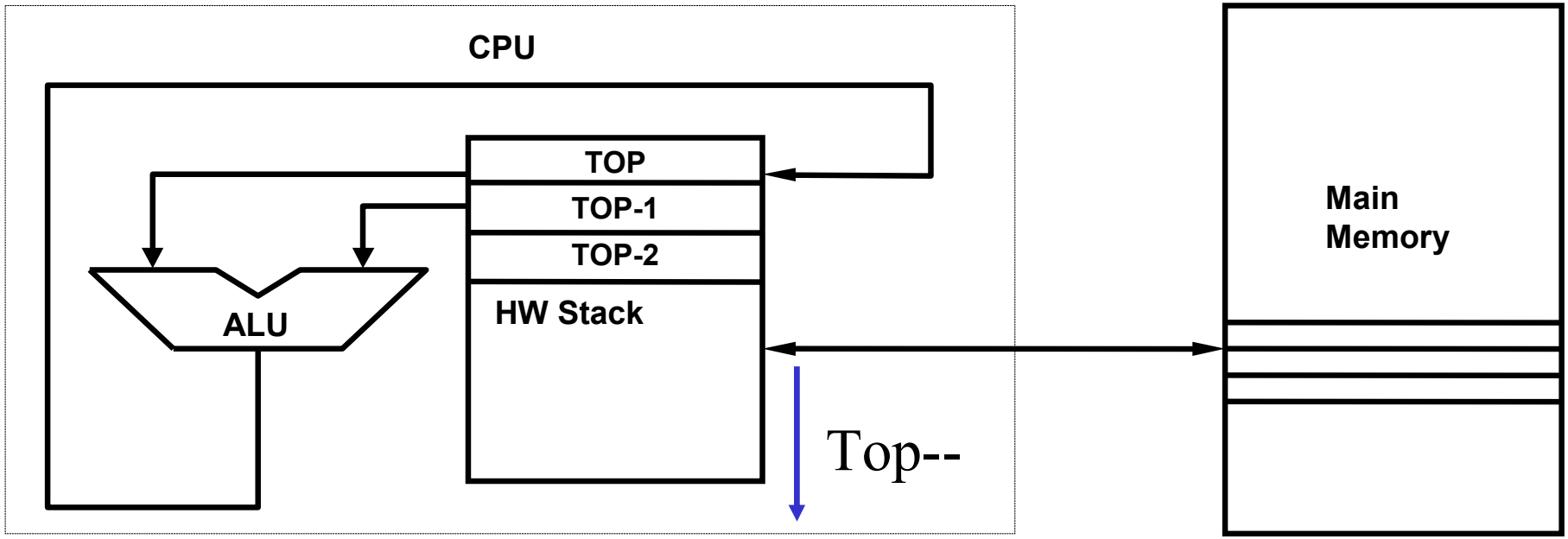
Zásobníkově orientovaná ISA (2)

HW zásobník = sada registrů s ukazatelem na vrchol (uvnitř CPU)

Instrukce :

PUSH A	Stack[++Top] ← mem[A]
POP A	mem[A] ← Stack[Top--]
ADD,SUB, …	Stack[Top-1] ← Stack[Top] **OP** Stack[Top-1]
	Top--

B5000 rovněž podporuje indexovací registry.



Zásobníkově orientovaná ISA (3)

HW zásobník je součástí CPU ⇒ malý (*N* registrů)

ALE

Teoreticky je zásobník **nekonečný** ⇒

- *N* položek nejvýše na zásobníku je v CPU (HW zásobník)
- Zbytek se emuluje v **hlavní paměti.**
- Přenos mezi HW zásobníkem a pamětí řeší automaticky HW **(*Stack Spilling* – přelévání zásobníku)**
- Přenos se provádí při **přetečení** nebo **podtečení** HW zásobníku.

Zásobníkově orientovaná ISA – příklad (4)

Mějme 2-registrový HW zásobník

Příklad : **(a + b * c) / ((a + d * c) - e)**

Program v postfixovém zápisu: **a b c * + a d c * + e - /**

Program	Zásobník	Přístupy do paměti
PUSH a	**R0**	**a**
PUSH b	**R1 R0**	**b**
PUSH c	**R2 R1 R0**	**c, ss**
MUL	**R1 R0**	**sf**
ADD	**R0**	
PUSH a	**R1 R0**	**a**
PUSH d	**R2 R1 R0**	**d, ss**
PUSH c	**R3 R2 R1 R0**	**c, ss**
MUL	**R2 R1 R0**	**sf**
ADD	**R1 R0**	**sf**
PUSH e	**R2 R1 R0**	**e, ss**
SUB	**R1 R0**	**sf**
DIV	**R0**	

Celkem: 7 načtení operandu
+ 4 přelévání zásobníku (ss)
+ 4 čtení ze zásobníku (sf)

Proměnné a a c se vždy čtou dvakrát (nezávisle na velikosti HW zásobníku) !

Stejný příklad pro GPR (Load/Store) ISA

4 registry by měly stačit

Výraz: **(a + b * c) / ((a + d * c) - e)**

```
                a c b d e
LOAD R0, a      |
LOAD R1, c      | |
LOAD R2, b      | | |
MUL  R2, R1     | | |
ADD  R2, R0     | | |
LOAD R3, d      | | | |
MUL  R3, R1     | | | |
ADD  R3, R0     | | | |
LOAD R0, e      |   |   |
SUB  R3, R0     |   |   |
DIV  R2, R3     |   |   |
```

Potřeba pouze 5 čtení operandů
(porovnejte se 7)

- Kódování instrukcí je delší
- Kompilátor je složitější
- Alokace registrů eliminuje zásobník
  pro vyhodnocení výrazu
  (je potřeba analýza využití proměnných,
  záleží na pořadí instrukcí)
- HW zásobník není potřeba
- Alokace registrů je NP-těžký problém

Zásobníkově orientované ISA (5)

Závěr – většinou vyhynuly před rokem 1980

Výhody:

- jednoduchá a efektivní adresace operandů
- krátké instrukce
- vysoká hustota kódu (krátké programy)
- jednoduché dekódování instrukcí
- neoptimalizující překladač se dá snadno napsat

Nevýhody:

- nelze náhodně přistupovat k lokálním datům
- zásobník je sekvenční (omezuje paralelismus)
- přístupy do paměti je těžké minimalizovat

Zásobníkově orientované ISA (6)

Po roce 1980

Inmos Transputers (1985 – 1996)

- navrženy k podpoře efektivního paralelního programování pomocí paralelního programovacího jazyka **Occam**
- **Inmos T800** byl v druhé polovině 80. let nejrychlejší 32-bitový CPU
- zásobníkově orientovaná ISA zjednodušila implementaci
- podpora pro rychlé přepínání kontextu

Forth machines

- Forth je zásobníkově orientovaný jazyk
- používá se v řídicích a kybernetických aplikacích
- několik výrobců (Rockwell, Patriot Scientific)

Intel x87 FPU

- nepříliš dobře navržený zásobník pro vyhodnocování FP výrazů
- překonán architekturou SSE2 FP v Pentiu 4

Java Virtual Machine, .NET

- navržen pro **SW emulaci** (podobně jako PostScript)
- Sun PicoJava a další HW implementace

Vývoj ISA – Historický pohled