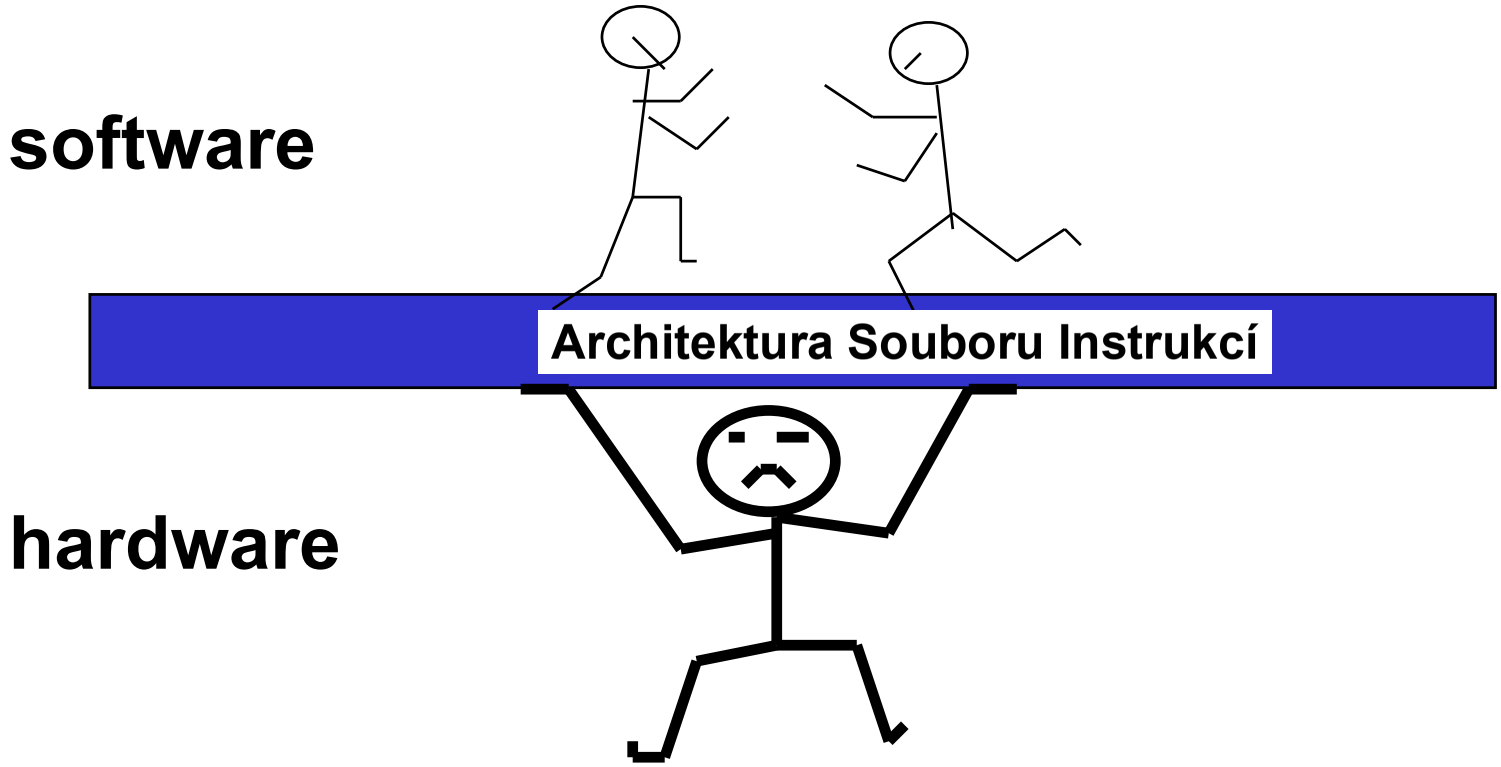
Architektura Souboru Instrukcí

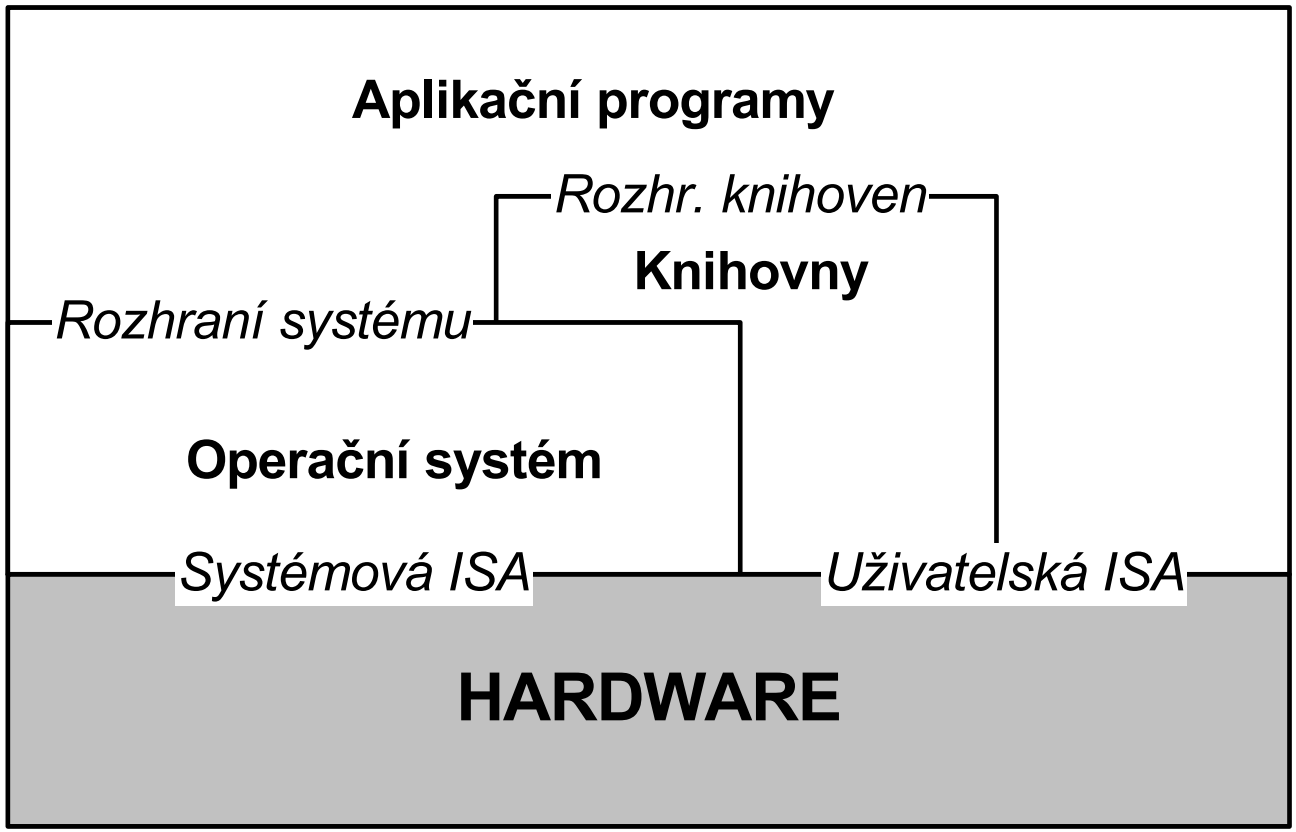
Ing. Miloš Bečvář

Architektura Souboru Instrukcí (ISA)



ISA je pohled na počítač z hlediska programátora ve strojovém jazyce .

ISA jako jedno z důležitých rozhraní



ISA = Uživatelská ISA + Systémová ISA

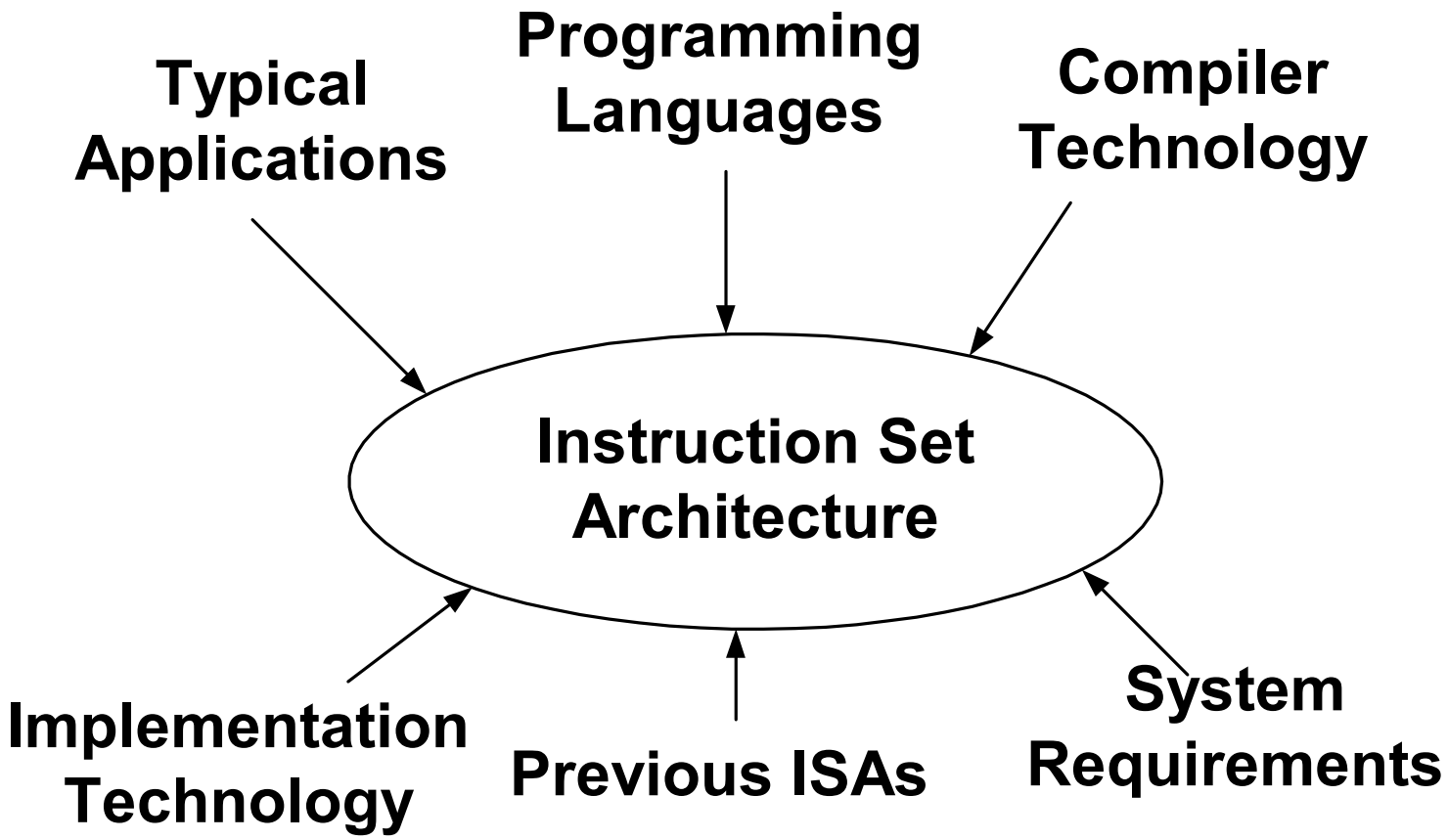
ABI* = Uživatelská ISA + Rozhraní systému

API+ = Rozhraní knihoven (na úrovni vyššího p.j.)

* Application Binary Interface

\+ Application Programming Interface

Faktory ovlivňující definici ISA



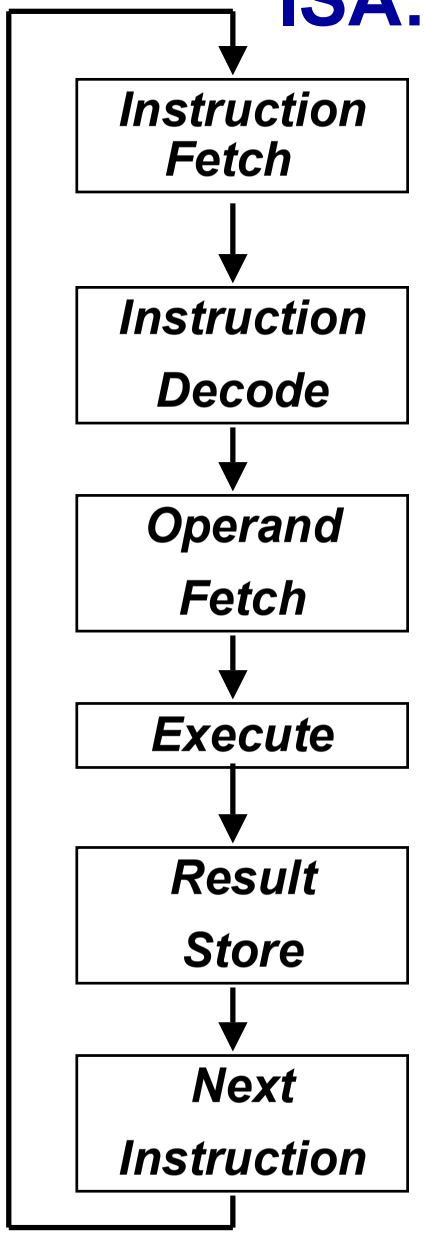
Výkonnostní rovnice CPU

$$T_{cpu} = IC * CPI * T_{clk}$$

	IC	CPI	T_{clk}
Překladač	*	*	
Architektura Souboru Instrukcí	*	*	*
Organizace Počítače		*	*
Technologie			*

ISA „žije dlouho“ díky principu zpětné binární kompatibility (tzv. legacy effect)

ISA: Co musí být definováno ?



- **Kódování instrukcí**
  - Jak jsou instrukce kódovány ?
- **Umístění operandů**
  - Kolik explicitních operandů je v ALU instrukci ?
  - Které operandy mohou být v paměti ?
  - Jak jsou operandy umístěny v paměti ?
- **Datové typy a velikosti operandů**
- **Operace v ISA**
  - Které operace jsou podporovány ?
- **Umístění výsledku**
- **Výběr následující instrukce**
  - Podmíněné skoky, volání a návraty z podprogramu, přerušovací model

Základní třídy ISA

Střadačově-orientovaná ISA:

1 adresní	ADD A	acc ← acc + mem[A]
	ADD (A + IX)	acc ← acc + mem[A + IX]

IX je **index registr**

Zásobníkově-orientovaná ISA:

0 adresní	ADD	stack(top-1) ← stack(top) + stack(top-1), top--

ISA s univerzálními registry (GPR ISA):

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

EA … **Efektivní Adresa** (specifikuje registr nebo paměťový operand)

Střadačově-orientovaná ISA dnes

Střadač je implicitním zdrojovým a cílovým operandem ALU instrukce.

Druhý (explicitní) operand může být:

- v paměti adresován absolutně příp. s **indexovým registrem**
- v paměti adresován nepřímo přes registr (např. **ukazatel zás.**)
- v poli pracovních registrů (tzv. ***zápisníková paměť***)

Střadačová ISA použita v prvních mikroprocesorech:
4004, 8008, 8080, 6502, …

Dnes je tato ISA použita u některých mikrořadičů:
8051, 68HC11, 68HC05, DOP …

X86 byla uvedena na trh jako „multiple accumulator ISA…“
=> S příchodem i386 byla doplněna na koncept GPR.

Střadačově-orientovaná ISA - souhrn

Výhody:

- jednoduchý HW
- minimální vnitřní stav procesoru => rychlé přepínání kontextu
- krátké kódy instrukcí (záleží na způsobu adresace 2. operandu)
- jednoduché dekódování instrukcí a řadič

Nevýhody :

- častý přístup do paměti (dnes problém)
- limitovaný paralelismus mezi instrukcemi

Není náhoda, že tento typ ISA byl populární v 50. a 70. letech
- HW CPU byl drahý či omezený,
- Cyklus paměti byl kratší než instrukční cyklus.

Zásobníkově-orientovaná ISA

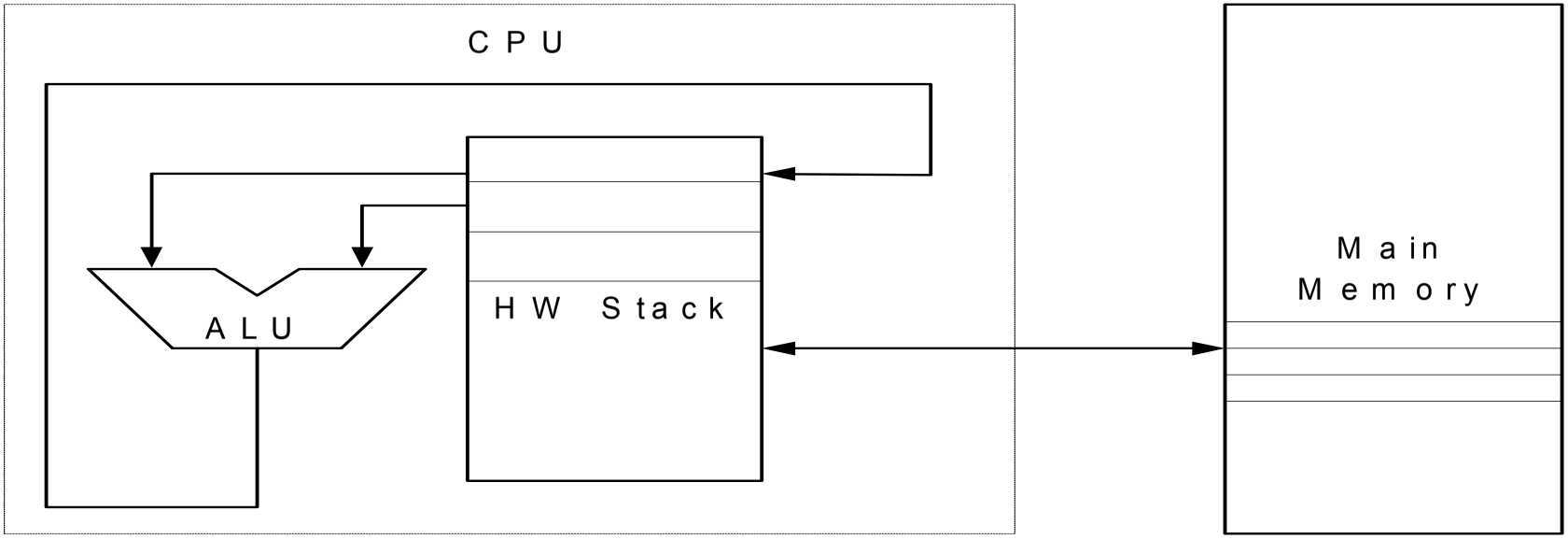
HW zásobník = registrové pole s ukazovátkem "Top" (uvnitř CPU)

Základní instrukce :

PUSH A Top++, Stack[Top] ← mem[A]

POP A mem[A] ← Stack[Top], Top--

ADD,SUB, … Stack[Top-1] ← Stack[Top] **OP** Stack[Top-1], Top--



Zásobníkově-orientovaná ISA

HW zásobník je část CPU => malý počet položek (*N* registrů)

ALE

Konceptuálně je zásobník **nekonečný** =>

- *N* nejvyšších položek zásobníku je v CPU (HW zásobník)
- Zbytek zásobníku je simulován v **hlavní paměti.**
- Přesuny mezi HW zásobníkem a hlavní pamětí řeší řadič procesoru automaticky (***Stack Spilling, Stack Fetching***)
- K přesunu dochází při přetečení a podtečení HW zásobníku

Zásobníkově-orientovaná ISA vs GPR ISA

Zásobníkově orientovaný procesor s N-položkovým HW zásobníkem má schopnost uschovat stejný objem dat uvnitř procesoru jako GPR procesor s N-registry.

Zásobníkově-orientovaný procesor

- **Omezený přístup k datům uvnitř procesoru**
- **Nutnost HW implementace *stack spillingu***

GPR procesor

- **Plně náhodný přístup k datům uvnitř procesoru (registry)**
- **Složitější implementace registrového pole**

=> GPR procesor je efektivnější v optimalizaci počtu přístupů do paměti neboť může s registry pracovat v libovolném režimu.

Zásobníkově-orientované ISA (téměř) vyhynuly před rokem 1980

Výhody :

- jednoduché a efektivní specifikace operandů v ALU instrukcích
- jednoduché a rychlé instrukce
- vysoká hustota kódování (programy zabírají málo paměti)
- napsání neoptimalizujícího překladače je snadné
- rychlá interpretace či emulace – vhodné pro *virtuální stroje*

Nevýhody :

- chybí možnost náhodného přístupu k lokálním proměnným
- zásobník je sekvenční (limituje paralelismus operací)
- obtížná minimalizace přístupů do paměti

Interpretace zásobníkově-orientované ISA

Hlavní paměť

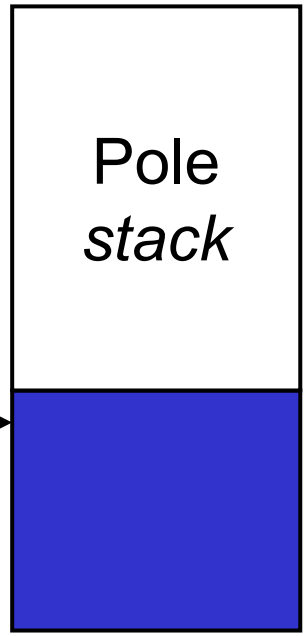
Pole
mem

PUSH A : *sp++; stack[sp]=mem[A];*
ADD : *stack[sp-1]=stack[sp] + stack[sp-1];*
sp--;
POP A : *mem[A]=stack[sp]; sp--;*

Interpretace je rychlá :

- Proměnná ***sp*** může být udržována v registru (implementace pomocí ukazatelů)
- Není třeba **extrahovat adresy operandů** u ALU instrukcí z kódu instrukce

Zásobník



Při interpretaci nemá rozdělení na HW zásobník a „zbytek“ smysl. Přesuny mezi nimi tak odpadnou.

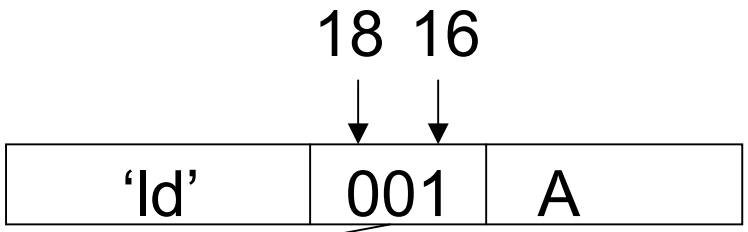
Interpretace GPR ISA

Hlavní paměť

Pole
mem

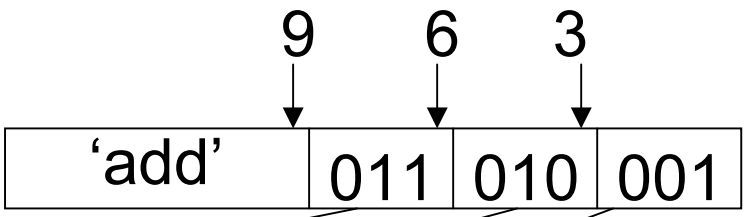
LD R1, A :

rs=extract(instr,16,18);
reg[rs]=mem[A];



ADD R3,R2,R1:

rs=extract(instr,0,2);
rt=extract(instr,3,5);
rd=extract(instr,6,8);
reg[rd]=reg[rs]+reg[rt];



Registry

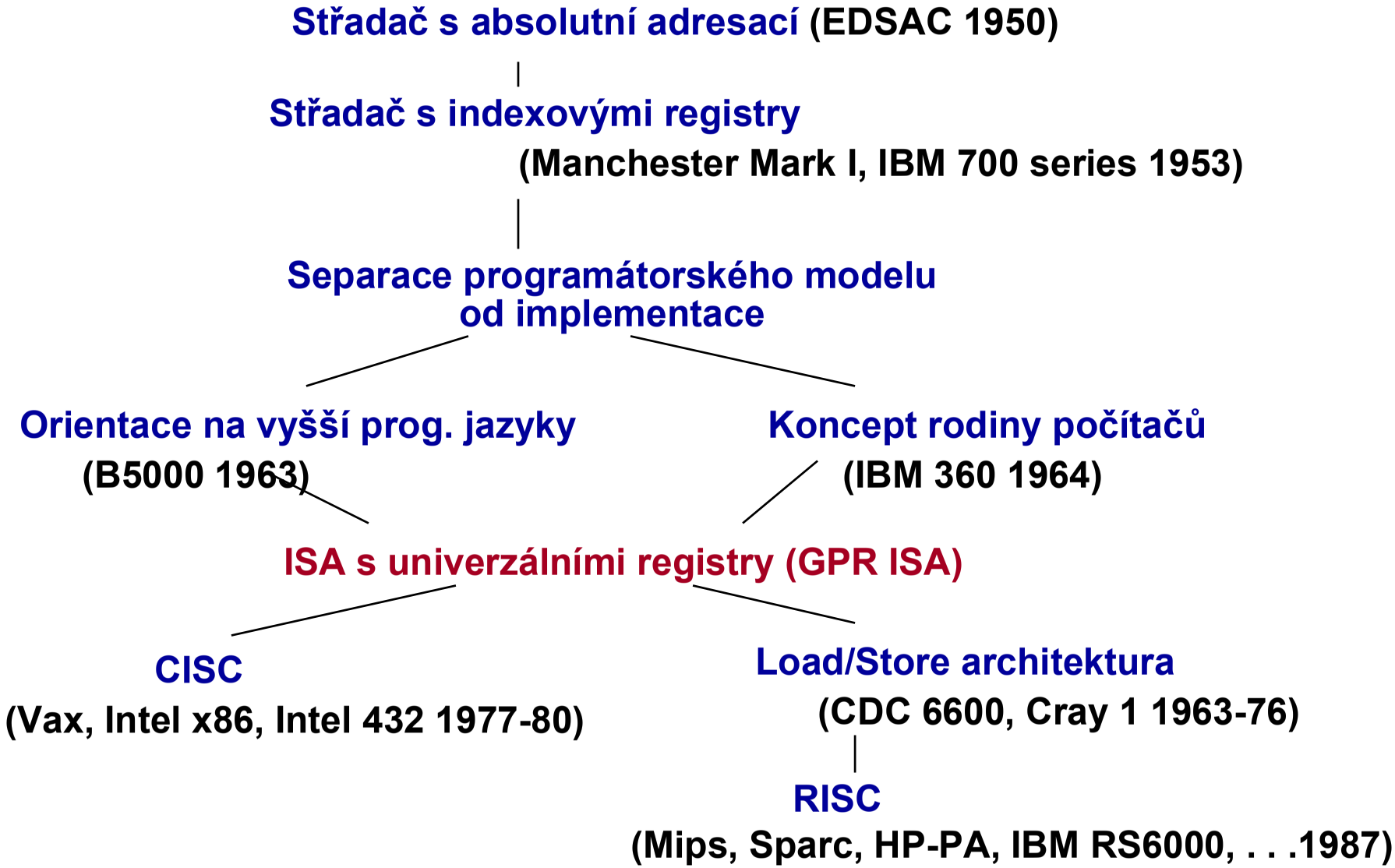
Pole
reg

Extrakce čísel registrů z kódu instrukce a výpočet jejich adresy v poli *reg* zpomaluje interpretaci oproti zásobníkové ISA.

Zásobníkově-orientovaná ISA po roce 1980

- Inmos Transputery (1985 – 1996)
  - navrženy pro paralelní systémy s podporou specializovaného paralelního jazyka **Occam**
  - **Inmos T800** byl nejrychlejším 32-bitovým mikroprocesorem ve druhé polovině 80. let
  - zásobníkově-orientovaná ISA zjednodušila implementaci
  - podpora rychlého přepínání kontextu
- Počítače s podporou jazyka Forth (Forth machines)
  - Forth je zásobníkově-orientovaný vyšší prog. jazyk
  - používán v řídicích aplikacích a robotice
  - řada výrobců mikrořadičů (Rockwell, Patriot Scientific)
- Intel x87 FPU stack
  - nepříliš dobře navržené rozhraní matematického koprocesoru 8087
  - dnes nahrazeno instrukcemi SSE2 (Pentium 4)
- Java Virtual Machine
  - navržena pro **SW interpretaci** (podobně Postscript Virtual Machine)
  - Sun PicoJava a další podpora přímého vykonávání „bytecode“

Vývoj ISA z pohledu historie



ISA s univerzálními registry dnes dominuje

• Po roce 1975 všechny nové procesory užívaly nějakou formu GPR

Výhody registrů

- Registry jsou rychlejší než paměť (včetně cache !!)
- Přístup k registrům může být náhodný **(X** zásobník je sekvenční struktura)
  Např.: (A*B) – (C*D) – (E*F) součiny mohou být vyhodnoceny v libovolném pořadí (případně i paralelně)
- Registry mohou uchovávat mezivýsledky, lokální proměnné a parametry
- Méně přístupů do paměti – potenciální zrychlení

Nevýhody (kritika) registrů

- je jich limitovaný počet
- složitější překladač (optimalizace použití registrů)
- dlouhé přepínání kontextu
- registry nemohou uchovávat složené datové struktury (záznamy, pole …)
- proměnné v registrech nelze adresovat pomocí ukazatelů
  (některé proměnné nemohou být uloženy v registrech)

Volby při návrhu GPR ISA

- Dva nebo tři operandy v instrukci ?

  ADD R2, R1 vs ADD R3, R2, R1

- Počet univerzálních registrů ?

  8, 16, 32, 64, 128

- Počet paměťových operandů v ALU instrukci ?

  0,1,2,3

- Počet a typ podporovaných adresních módů ?

- Kódování instrukcí pevné či proměnné délky ?

Tyto otázky představují obtížné kompromisy ovlivňující paměťovou náročnost, složitost procesoru a výkonnost.

Adresní módy – Které z nich má ISA podporovat ?

Adresní mód	Syntaxe	Sémantika
Registrový	ADD R4,R3	R4← R4+R3
Přímý operand	ADD R4,#3	R4 ← R4+3
Bázovaný s konstantou	ADD R4,100(R1)	R4 ← R4+Mem[100+R1]
Registrový nepřímý	ADD R4,(R1)	R4 ← R4+Mem[R1]
Bázovaný/indexovaný	ADD R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Přímá/absolutní adresa	ADD R1,(1001)	R1 ← R1+Mem[1001]
Paměťový nepřímý	ADD R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
S post-inkrementací	ADD R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
S pre-dekrementací	ADD R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Škálovaný	ADD R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

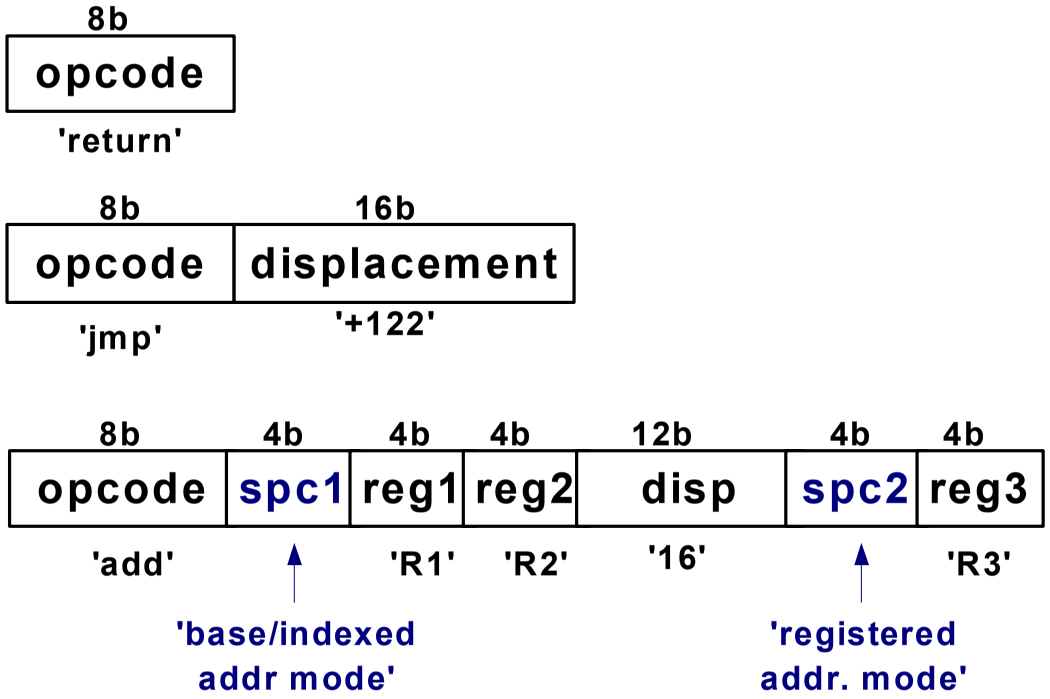
Formáty kódování instrukcí (1)

Proměnná délka kódu instrukce

+ lepší hustota kódování (aplikace kritické na kapacitu paměti)

+ každý operand může mít svůj specifikátor adresního módu (umožňuje komplexní adresní módy a ortogonalitu ISA)

- složité dekódování (vyžaduje čas nebo plochu čipu či oboje)



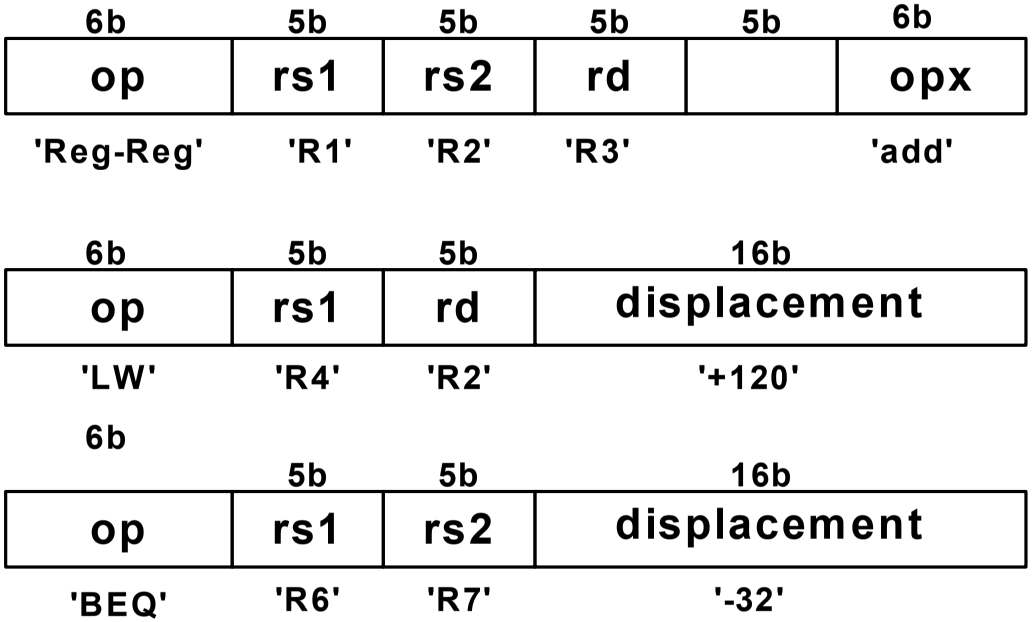
Délka kódu IA32 (X86) instrukce je od 1B do 13B

Formáty kódování instrukcí(2)

Pevná délka kódu instrukce

+ rychlé a jednoduché dekódování instrukcí
+ snazší implementace proudového zpracování (pipeline)
+/- adresní mód je součást kódu operace
(méně ortogonální ale snazší na dekódování)

- programy zabírají více paměti (plýtvání)
- implementace komplexních adresních módů je obtížná



ADD R3, R2, R1

LW R2, (R4)+120

BEQ R6,R7,-32

Podrobná klasifikace GPR ISA

Počet operandů v ALU instrukci	Max. počet paměťových operandů v ALU ins.	Příklady
2 **3**	**0**	IBM RT PC SPARC, MIPS, HP PA, ALPHA, PowerPC
2 **3**	**1**	PDP10, M6800, IBM360, Intel x86 IBM360RS
2 **3**	**2**	PDP11, National 32x32, IBM 360SS, VAX NEC S1
3	**3**	VAX

Registr–Registr (3,0) též Load-Store varianta

- (2,0) varianta je méně běžná (např. jednočipy)
- typická ISA RISC procesorů

Výhody :

- jednoduché kódování instrukcí (pevná délka instr. kódu)
- dekódování a čtení operandu je jednoduché a rychlé
- regulární pravidla generování instrukcí překladačem
- cpi_i závisí pouze na typu operace v instrukci
- snadná implementace proudového zpracování (pipeline)

Nevýhody :

- malá hustota kódování (více instrukcí, delší programy)
- některé instrukce by bylo možné zakódovat méně bity (např. ADD R1, R1, R2 jako ADD R1, R2) – plýtvání místem

Register–Paměť (2,1) varianta

• (3,1) varianta je méně běžná
• typická ISA CISC procesorů
• někdy nazývaná ***vícestřadačová ISA*** *(multiple-accumulator)*

Výhody :

- přímý přístup do paměti bez předchozí „load“ instrukce
- lepší hustota kódování (kratší programy)

Nevýhody :

- (2,1) operandy nejsou ekvivalentní (jeden zdrojový operand je ztracen) => vyžaduje dodatečné MOVE instrukce
- počet registrů může být limitován neboť specifikace adresy paměťového operandu může zabírat hodně bitů
- cpi_i má velkou variaci v závislosti na umístění operandů
- proudové zpracování instrukcí je obtížnější

Paměť–Paměť (2,2), (3,3) varianty

• „Pravý“ komplexní CISC model, momentálně „vyhynulý“ druh ISA a pravděpodobně to tak zůstane.

Výhody :

- zcela ortogonální (teoreticky nejlepší)
- registry nemusí být vůbec použity pro data proudového charakteru (grafika, multimédia)
- krátké programy (odpadají Load/Move/Store instrukce)

Nevýhody :

- proměnná délka kódu instrukce (VAX 1B až 36B)
- velká variabilita v cpi_i – „práce na instrukci“ není balancovaná => vyžaduje mikroprogramovaný řadič
- paměť může být úzkým hrdlem systému
- proudové zpracování na úrovni instrukcí téměř nemožné
- složitý překladač (mnoho možností překladu)

Modely přístupu k paměti - otázky

- Společná nebo oddělená paměť dat a programu ?
- Nejmenší adresovatelná položka v paměti ?
- Big Endian nebo Little Endian ?
- Podpora nezarovnaných dat ?

Modely přístupu k paměti - otázky

- **Harvardská architektura**
  - oddělená paměť dat a instrukcí (programu)
  - možnost současného přístupu k datům a instrukcím
  - datová a programová paměť mohou mít odlišnou organizaci
- **von Neumannova architektura (též Princetonská arch.)**
  - společná (unifikovaná) paměť pro data a instrukce
  - snadná manipulace s programy (nahrátí z disku, …)
  - možnost psát *sebemodifikující programy* (podpora není zaručena u současných procesorů)

Nejmenší adresovatelná jednotka v paměti

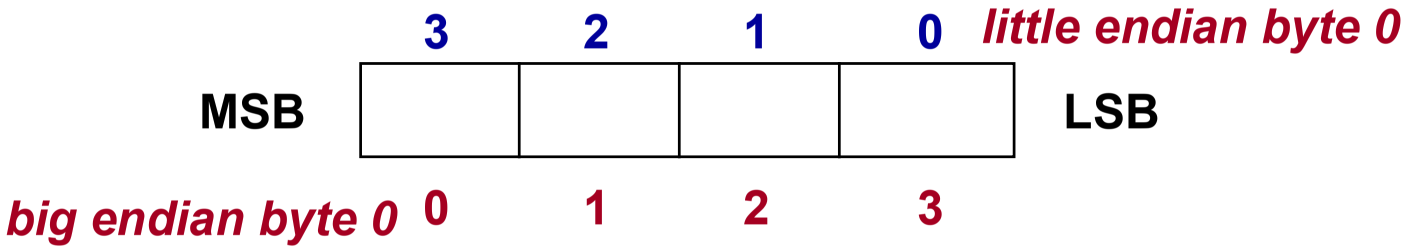
- Byte (slabika) = 8 bitů – dnes standard u univerz. počítačů

- Word (slovo) = více slabik či libovolný počet bitů – dnes zřídka

- Instrukční slovo – např. v Harvardské architektuře u paměti programu (jednočipové RISC mikrořadiče)

Problémy s adresací dlouhých operandů ve slabikově organizované paměti

- Big Endian: adresa MSB = adresa slova
  (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: adresa LSB = adresa slova
  (xx00 = Little End of word)
  - Intel 80x86, DEC VAX, DEC Alpha (Windows NT)



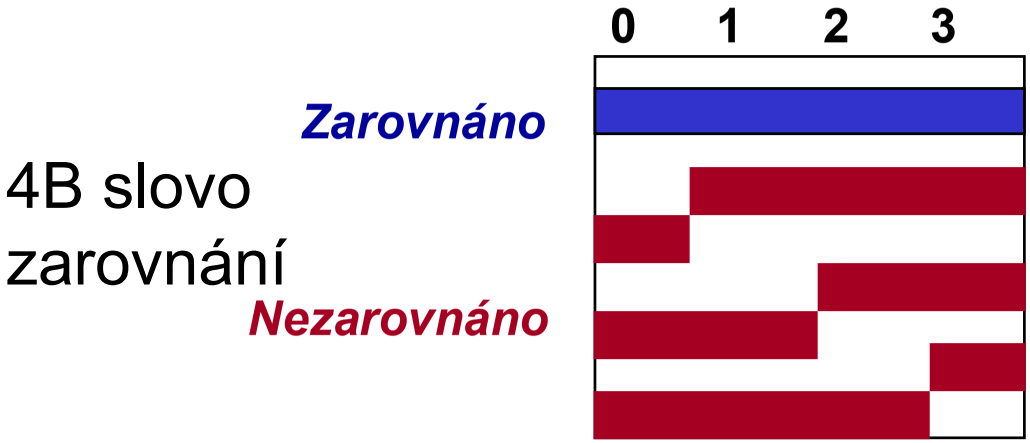
Viz X36SKD, X36JPO

Zarovnaná a nezarovnaná data

- ***Datový element je zarovnán (aligned), je-li uložen na adrese, která je dělitelná jeho velikostí.***

32-bitový procesor:

- **4B slovo – zarovnáno pokud je addresa dělitelná 4**
- **2B půlslovo – zarovnáno pokud je adresa dělitelná 2**
- **1B slabika – vždy zarovnána**



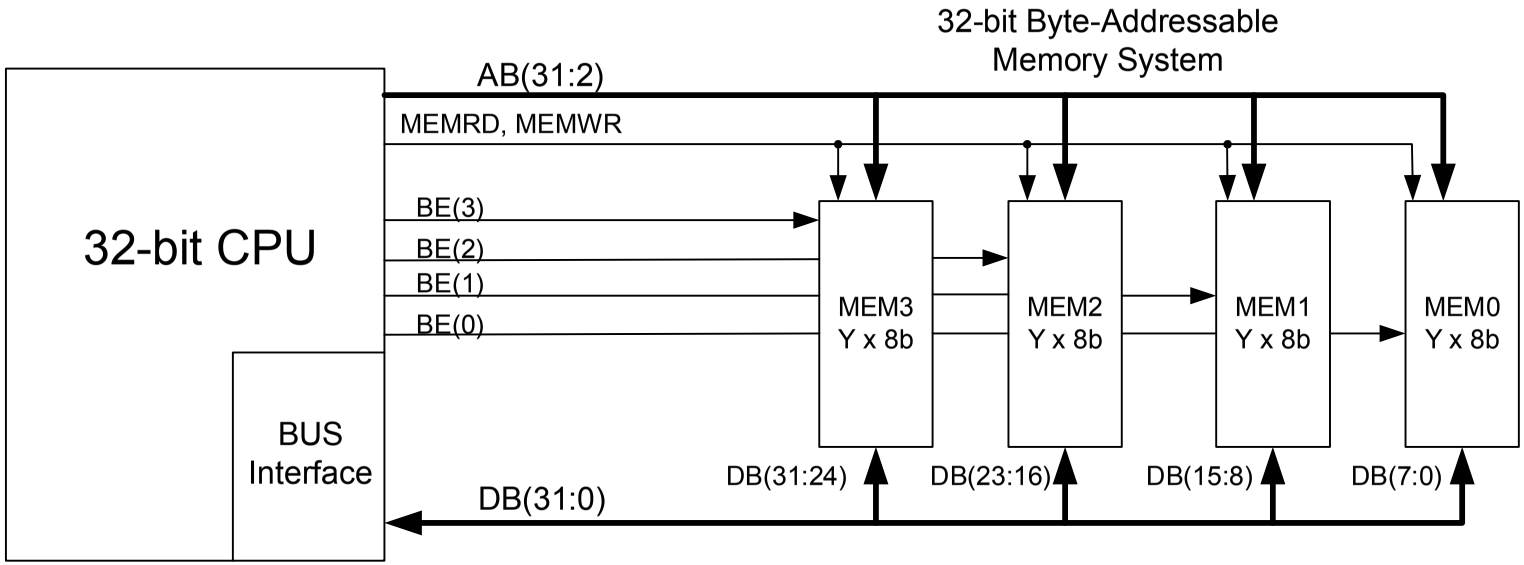
Má procesor podporovat nezarovnaná data ?

+ Šetří paměť
(element může začínat na libovolné adrese)

- Režie plochy na „zarovnávací“ obvody

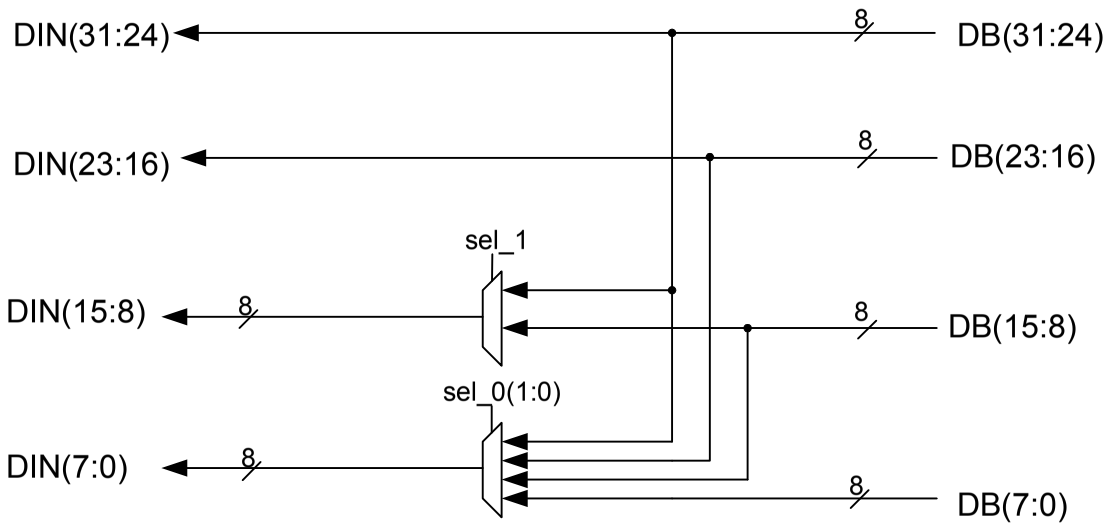
- Nezarovnaná data vyžadují více přístupů do paměti na realizaci čtení/zápisu (trvá déle)

Organizace paměťového subsystému (32b CPU)



- Pouze bity AB(31:2) adresují 32-bitovou paměť
- Byte-enable BE(3:0) signály řídí čtení/zápis elementů kratších než 32b (4B)

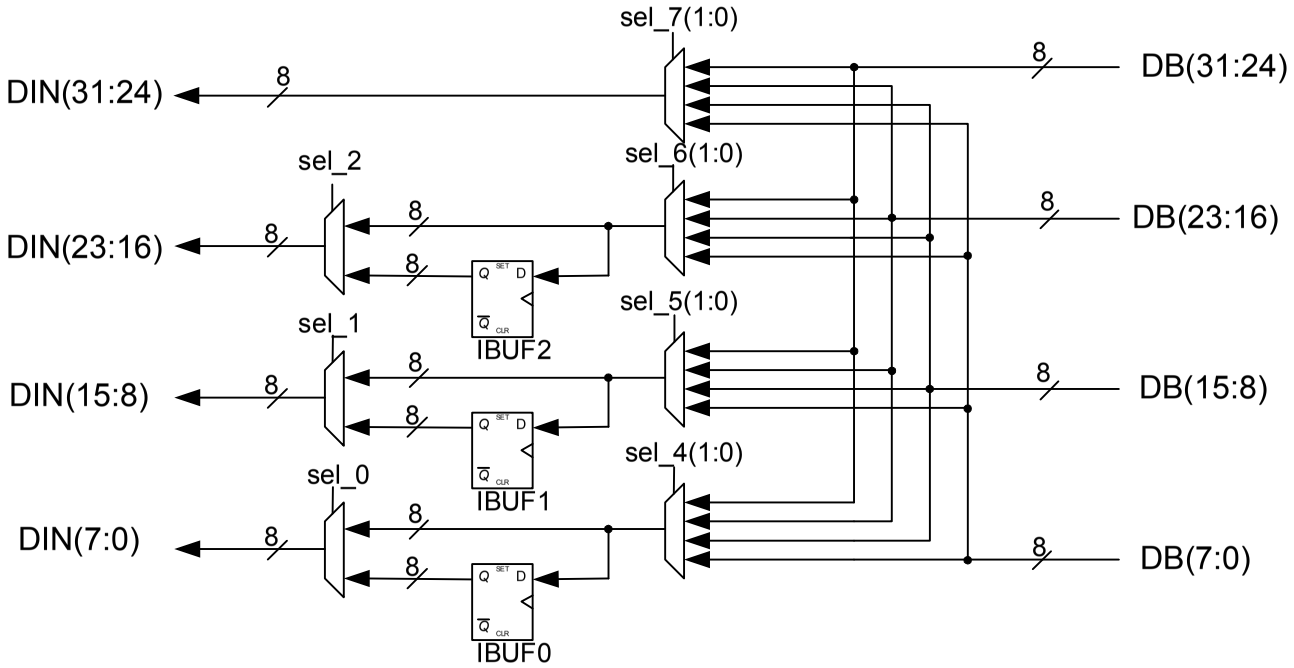
Rozhraní pro čtení dat CPU (jen zarovnaná data)



- Všechny přístupy do paměti vyžadují jen jedno čtení
- Dva multiplexery umožňují číst slabiku a zarovnané půlslovo

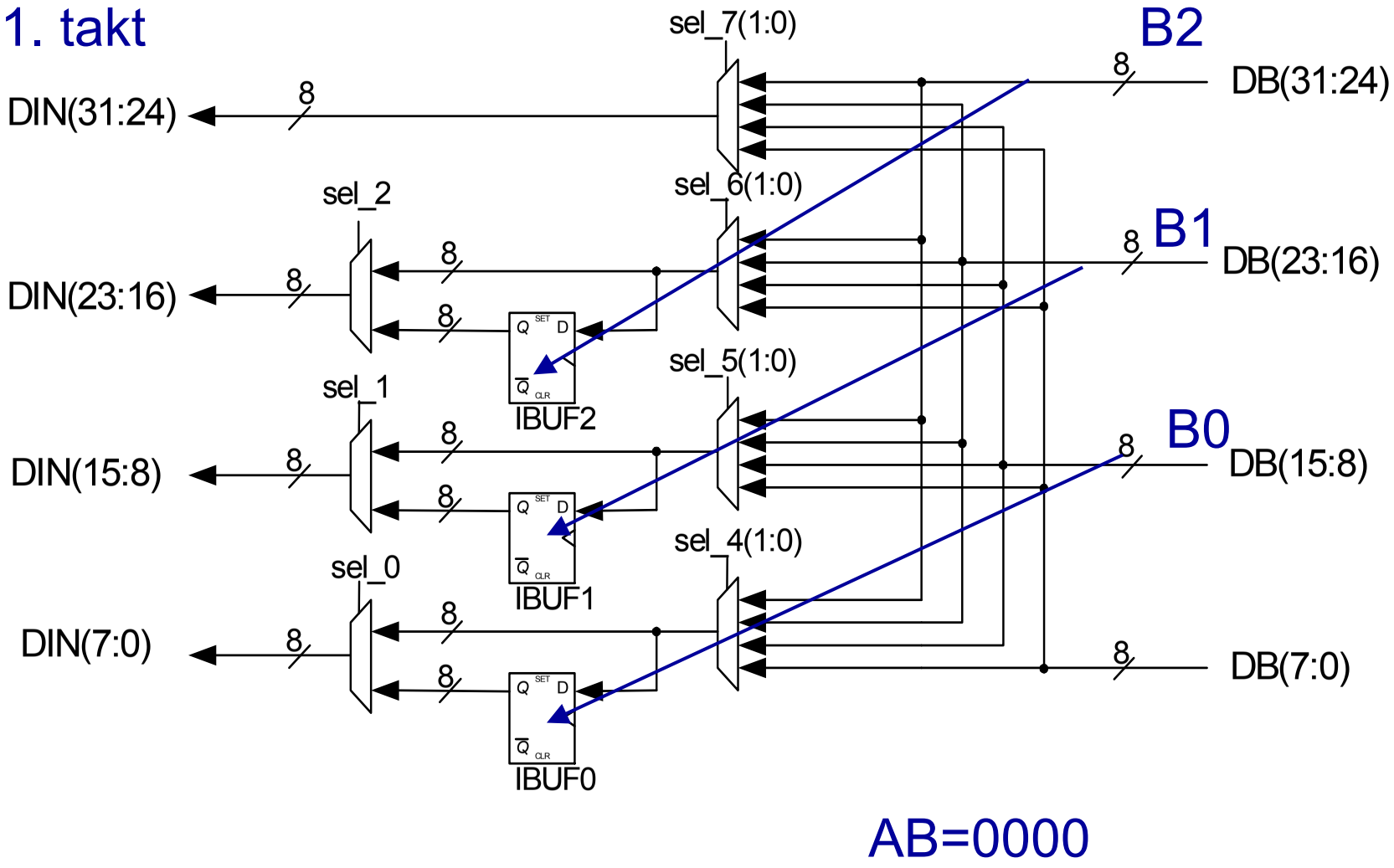
Některé procesory vyžadují od všech dat zarovnání na hranici slova, pak oba multiplexery nejsou potřeba.

Rozhraní pro čtení dat CPU (i nezarovnaná data)

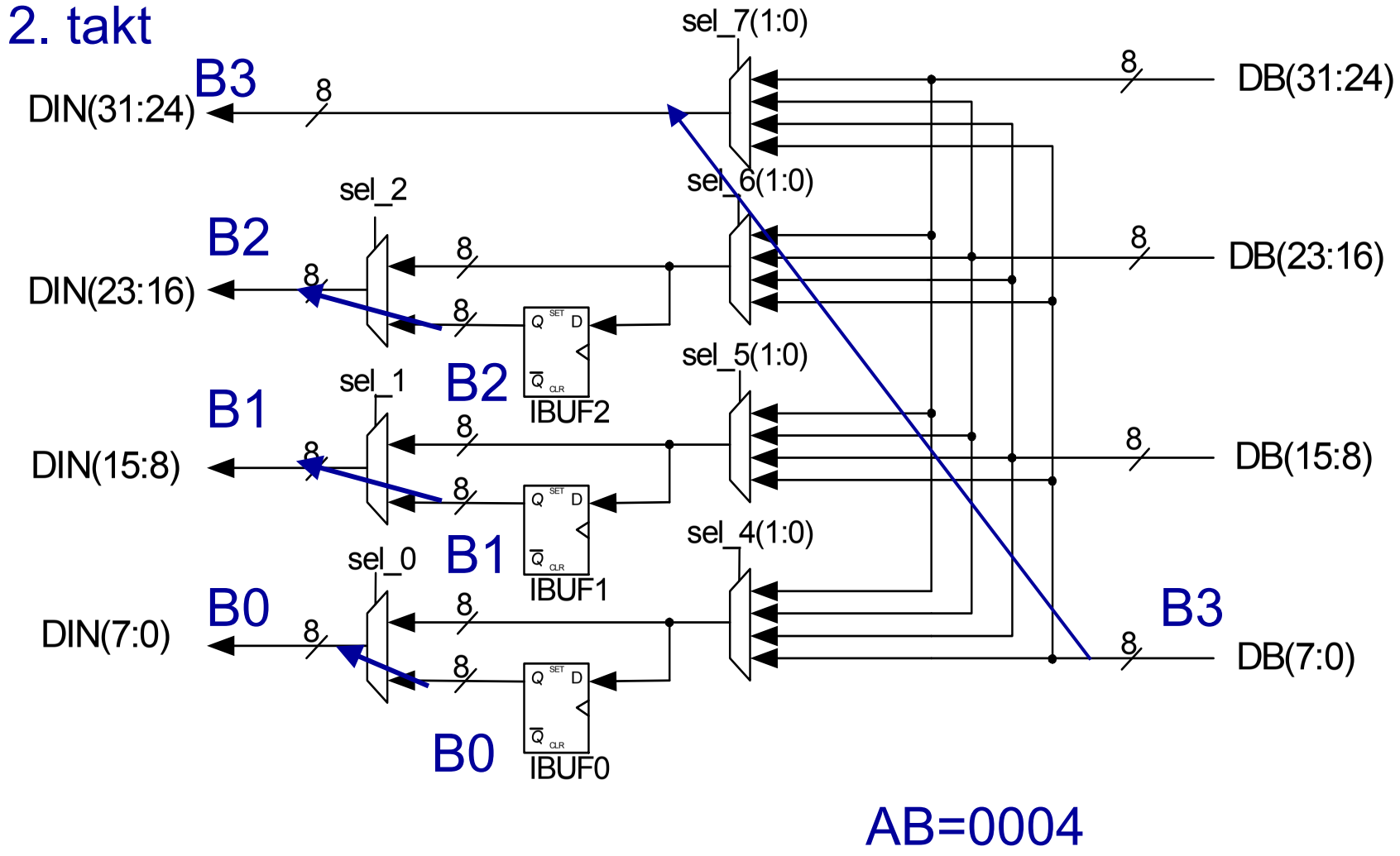


- Přístup k nezarovaným datům vyžaduje dva čtecí cykly (všimněte si též registrů IBUF)
- Multiplexery implementují úplnou permutační síť => velká plocha čipu (představte si též 64-bitovou verzi tohoto diagramu…)

Příklad – čtení slova z adresy 0001



Příklad – čtení slova z adresy 0001



Jak současné procesory řeší otázku zarovnání

X86 architektura:

- HW plně podporuje nezarovnaná data
- Překladače přesto většinou data zarovnávají z důvodů rychlosti

RISC architectury (např. DLX, Sparc, MIPS, Alpha)

- Přístup k nezarovnaným datům vede na výjimku
- SW obsluha může načíst nezarovnaná data pomocí několika instrukcí, typicky ale jen nahlásí chybu uživateli

=> Nezapomínejte používat direktivu *.align při psaní programu v assembleru DLX nebo jiného RISC procesoru.*

Shrnutí

Střadačově-orientovaná ISA je nejstarší, ale stále užívaná v některých vestavných aplikacích

Zásobníkově-orientovaná ISA umožňuje psaní jednoduchých překladačů. Využívána je v některých řídicích aplikacích a v emulačních systémech jako ISA virtuálních procesorů.

ISA s univerzálními registry (GPR ISA) je dnes nejběžnější.

Registr-Registr (3,0) GPR ISA je typická pro RISC

Registr-Paměť (2,1) GPR ISA je typická pro CISC

Typický RISC má Harvardskou architekturu, pevnou délku kódu instrukce a podporuje pouze zarovnaná data.