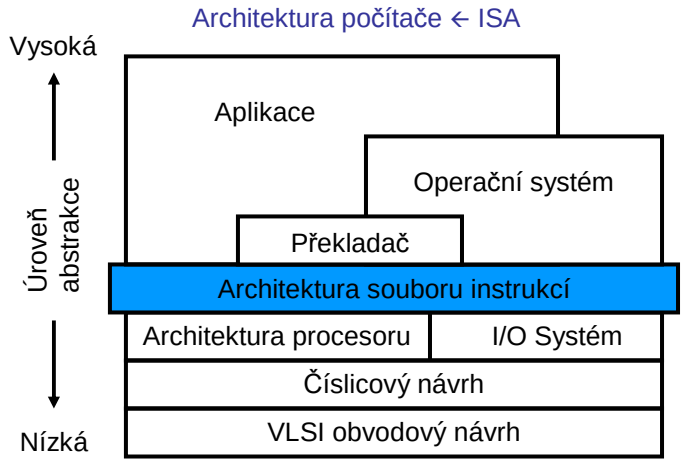
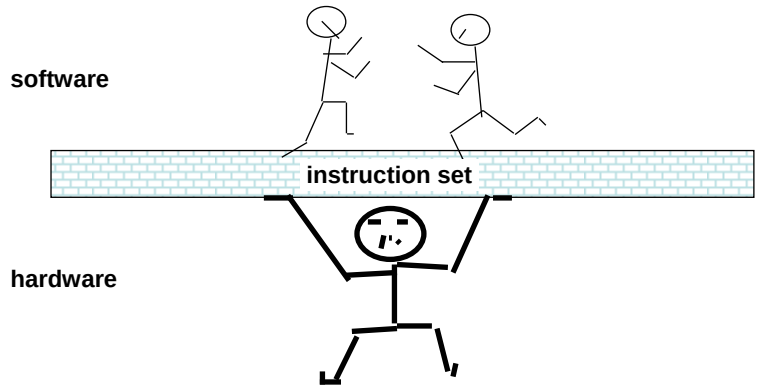
Y36SAP 9

Strojový kód
ISA – architektura souboru instrukcí
střadačově, zásobníkově
orientovaná, GPR

Architektura souboru instrukcí, ISA - Instruction Set Architecture



Instrukční soubor – kritický interface



Strojový kód

Co musí instrukce obsahovat

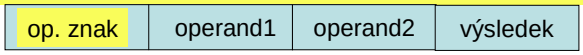
instrukce = příkaz, zakódovaný jako číslo

- co se má provést
- s čím se to má provést (operandy)
- kam se má uložit výsledek
- kde se má pokračovat

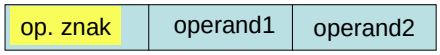
Tyto informace jsou obsaženy v instrukci ... tzn. **explicitně** např. počítač SAPO, tzv. 5 adresový

Zčásti v instrukci, zčásti dány architekturou počítače, tzn. **implicitně** (von Neumannova architektura)

Příklady strojových instrukcí



3 adresová instrukce



2 adresová instrukce

výsledek se ukládá na místo prvního operandu, zavedení operace **přesun**

$$\{x\} - \{y\} \rightarrow z \quad = \quad \begin{cases} \{x\} \rightarrow z \\ \{z\} - \{y\} \rightarrow z \end{cases}$$

1 adresová instrukce

zavedení „pracovního“ registru – STŘADAČ, ACCUMULATOR

$$\{x\} - \{y\} \rightarrow z \quad = \quad \begin{cases} \{x\} \rightarrow S \\ \{S\} - \{y\} \rightarrow S \\ \{S\} \rightarrow z \end{cases}$$

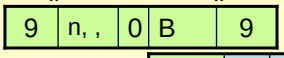
instrukce: *operační znak, OZ, opcode* | operand(y)

více „střadačů“ – více registrů k „obecnému“ použití

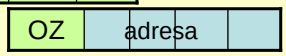
Příklad:

Motorola 68000
Datové registry $\mathbf{D_0, D_1, ..., D_7}$ á 32b
Odčítání 32b: $\mathbf{D_n - paměť \rightarrow D_n}$

OZ:



instrukce:



<D3> - <18FF20 ÷ 18FF23> → D3

96B90018FF20

zápis ve strojovém kódu je nepřehledný a špatně se programuje →

•vyšší programovací jazyk VPJ: Pascal, Java, C

•jazyk symbolických instrukcí JSI .... operační znak i adresy (operandy) jsou zapsány symbolicky .....asembler

Architektura souboru instrukcí, ISA - Instruction Set Architecture

Zahrnuje

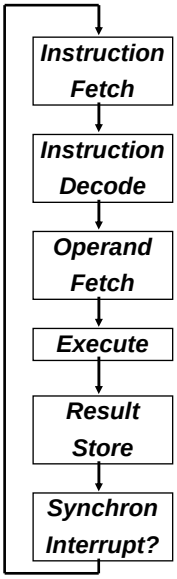
- Typy a formáty instrukcí, instrukční soubor
- Datové typy, kódovaní a reprezentace, způsob uložení dat v paměti
- Módy adresování paměti a přístup do paměti dat a instrukcí
- Mimořádné stavy

Umožňuje

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

2008-Kubátová Y36SAP-ISA 7

ISA: Co musí být definováno?



Formát a kódování instrukcí

– Jak se instrukce dekóduje?

Umístění operandů a výsledku

– Kolik explicitních operandů je v instrukci pro ALU?
– Jak jsou operandy umístěny v paměti nebo jinde?
– Který operand může být v paměti?

Typy dat a velikosti operandů

Operace v ISA

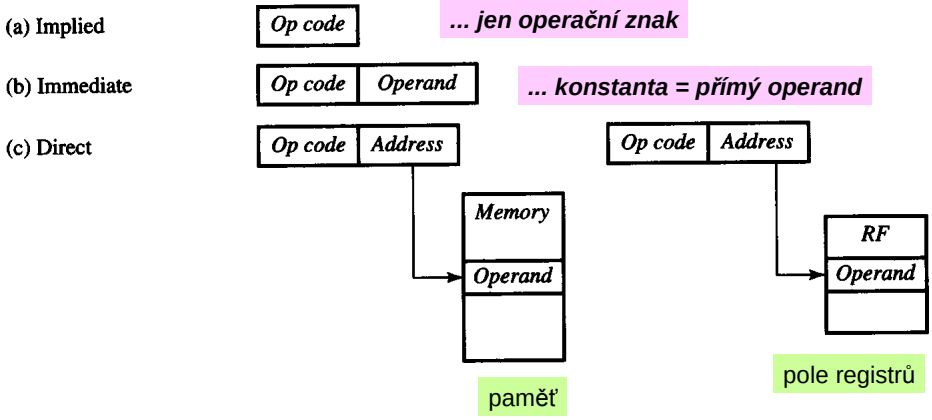
– Které jsou podporovány ?

Výběr další instrukce

– Skoky, větvení programu, volání podprogramů
– *„fetch-decode-execute“ je implicitní!*

2008-Kubátová Y36SAP-ISA 8

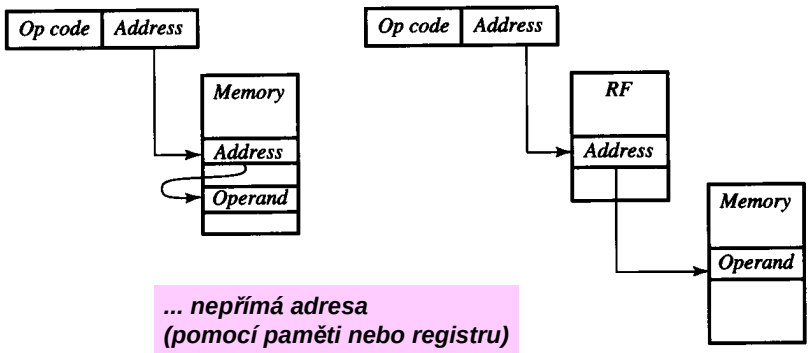
Adresace operandů - přímá



2008-Kubátová Y36SAP-ISA 9

Adresace operandů - nepřímá

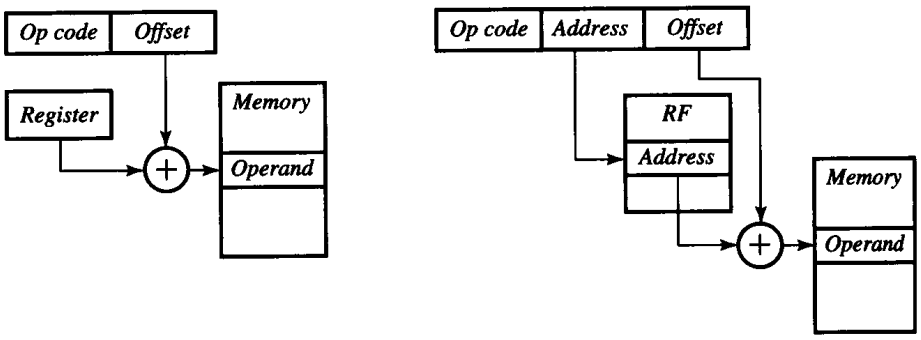
d) **Indirect**



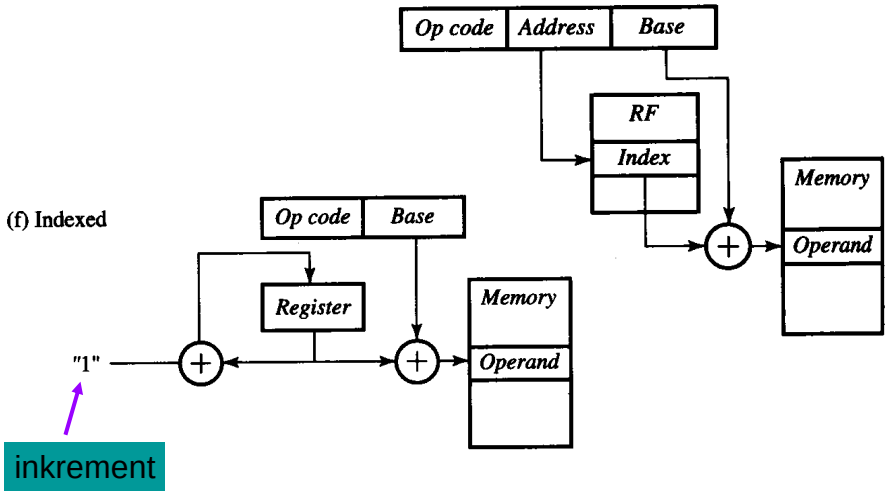
2008-Kubátová Y36SAP-ISA 10

Adresace operandů - relativní

d) **relative**



Adresace operandů - indexová



ISA: Základní třídy

Střadačově (akumulátorově) orientovaná ISA (1 registr=střadač):

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 registry 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)

Střadačově 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.

Střadačově orientovaná ISA - dnes

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

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

Implicitním operandem ALU je **vždy střadač** (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, …

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

Srřadačově 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

Využití „zásobníku“ při vykonávání programu :

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

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

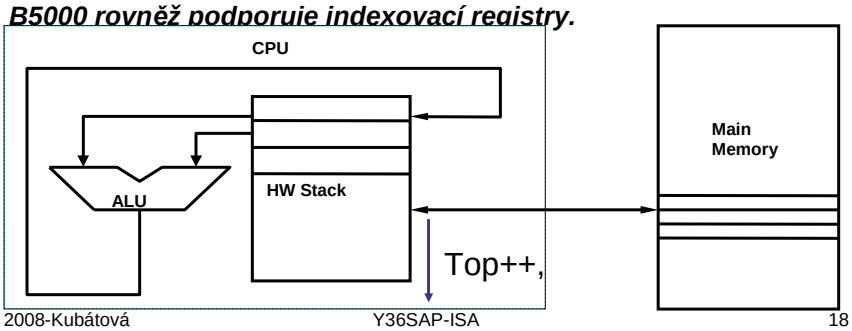
Zásobníkově orientovaná ISA

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 orientované ISA
… 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 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

GPR ISA … dnes převládá

- Po roce 1975 používají všechny nové procesory nějakou podobu GPR (registry pro všeobecné použití .. *general purpose registers*)

Výhody GPR ISA

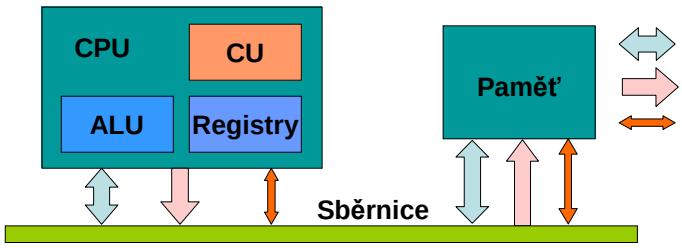
- Registry jsou rychlejší než paměť (včetně cache !!)
- K registrům lze přistupovat náhodně (X zásobník je přísně sekvenční)
- Registry mohou obsahovat mezivýsledky a lokální proměnné
- Méně častý přístup do paměti – potenciální urychlování

… GPR ISA

Nevýhody GPR ISA

- omezený počet registrů
- složitější překladač (optimalizace použití registrů)
- přepnutí kontextu trvá déle
- registry nemohou obsahovat složitější datové struktury (records …)
- k objektům v registrech nelze přistupovat přes ukazatele (omezuje alokaci registrů)

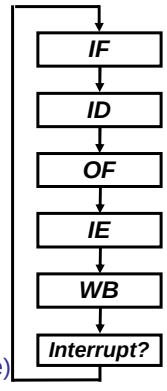
Jednoduchý procesor von Neumannova typu



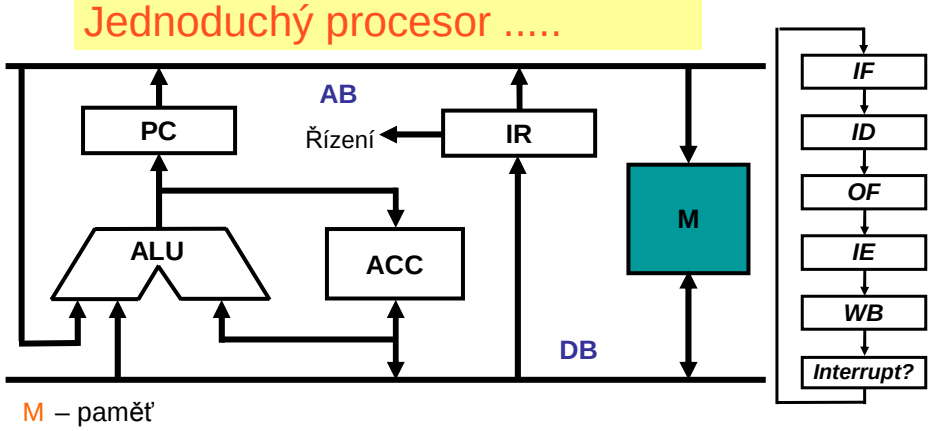
CPU musí obsahovat ALU, CU a registry pro dočasné uchování dat, instrukcí a pomocných proměnných

CPU pro běh programu prochází základním cyklem počítače:

1. Instruction Fetch — IF – načtení instrukce
2. Instruction Decode — ID – dekódování instrukce
3. Operand Fetch — OF – načtení operandu(ů)
4. Instruction Execution — IE – vykonání instrukce
5. Write Back — WB – zapsání výsledku (také Result Store)



Jednoduchý procesor .....



M – paměť

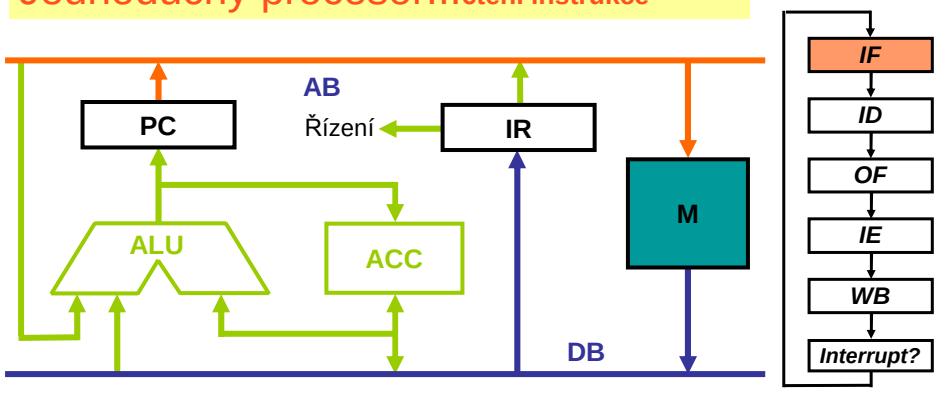
PC – Program Counter, prog. čítač, obsahuje adresu instrukce, která bude vykonána v dalším cyklu

ACC – Accumulator, střadač, obsahuje zpracovávaná data

ALU – Arithmetic Logic Unit, aritmeticko-logická jednotka, vykonává operace s daty

IR – Instruction Register, instrukční registr, obsahuje kód aktuálně prováděné instrukce – *pro programátora je skrytý*

Jednoduchý procesor…čtení instrukce

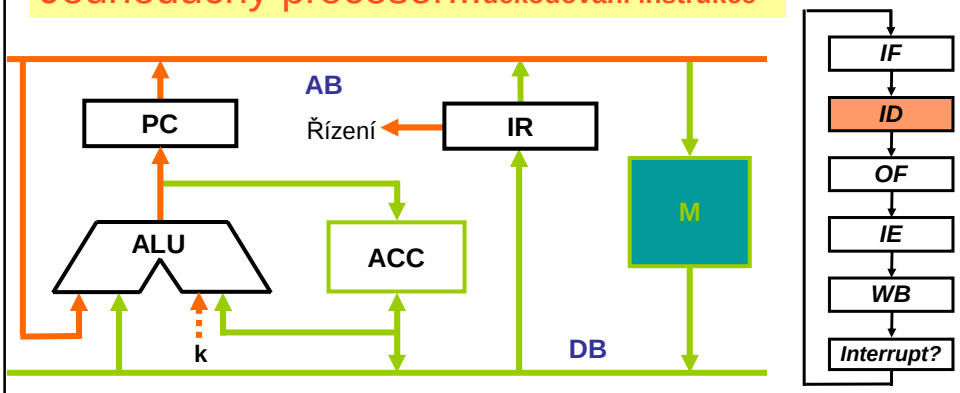


Procesor – pošle adresu instrukce z PC na AB

M – vystaví obsah paměťového místa instrukce z PC na DB

IR – kód instrukce z DB je zapsán do IR

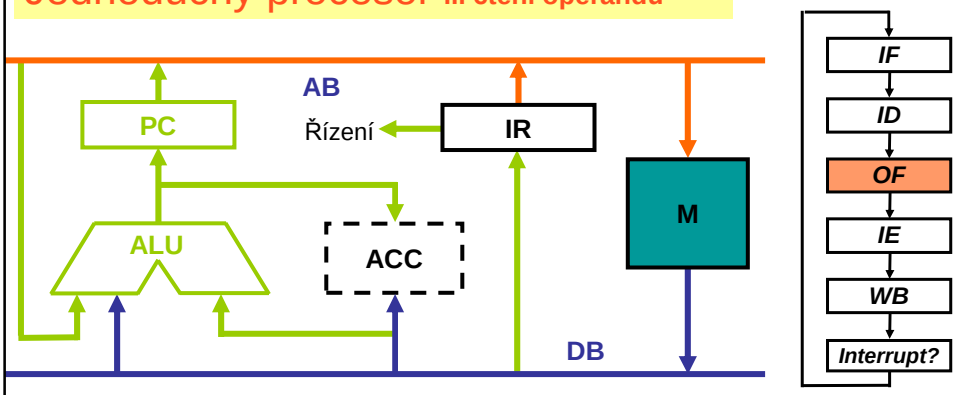
Jednoduchý procesor…dekódování instrukce



IR – instrukční kód z IR je dekódován interní logikou (řadičem) a současně jsou generovány řídící signály pro ALU a další interní obvody procesoru.

PC – programový čítač vystaví hodnotu na AB, ALU zvětší tuto hodnotu o k a zapíše nazpět do PC (když pc je původní obsah PC, ⇒ pc ← pc+k), hodnota k je určena dekódováním instrukce, sekvenční zpracování ⇒k=1, skok ⇒ k nějaké celé číslo s omezeného intervalu. 27

Jednoduchý procesor ... čtení operandů

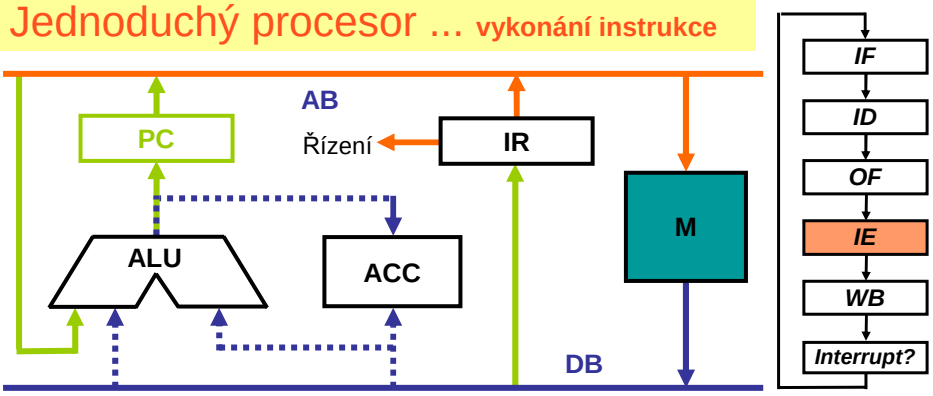


IR – zapsání adresy operandu, který má být při vykonávání instrukce použit, na AB

M – paměť vystaví hodnotu operandu na DB, takto je operand připraven pro zpracování buď v ALU nebo v ACC

2008-Kubátová Y36SAP-ISA 28

Jednoduchý procesor … vykonání instrukce



IR – vystavuje na AB adresu hodnoty operandu, který má být při vykonání instrukce použit

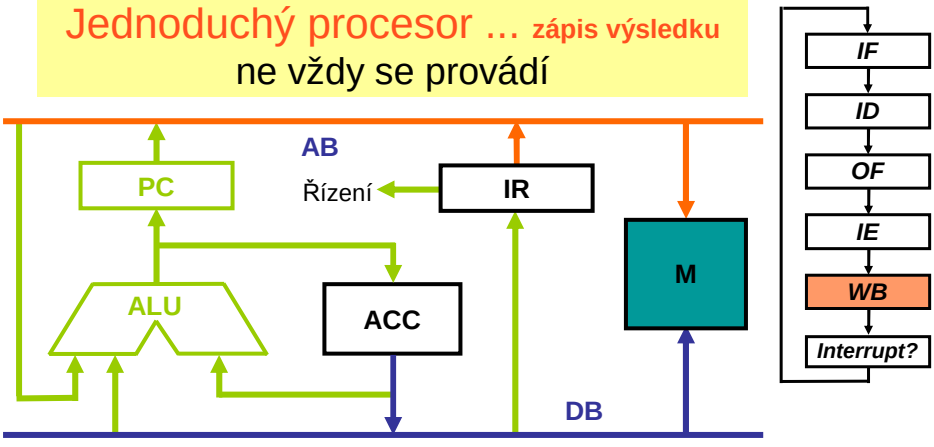
ALU – operace je vykonána v ALU podle instrukce, jejíž kód je v IR a generuje pomocí interní logiky řídící signály

M – paměť může mít nadále vystavenou hodnotu operandu na DB

ACC – slouží jako cílový registr, může být ale také zdrojovým

2008-Kubátová Y36SAP-ISA 29

Jednoduchý procesor … zápis výsledku
ne vždy se provádí



IR – vystaví hodnotu adresy paměti, kam má být hodnota obsažená v ACC zapsána

ACC – vystaví hodnotu obsaženou v něm na DB

M – hodnota z DB je zapsána do paměti

Táto fáze základního cyklu může být, nebo nemusí být vykonána. V některých procesorech je táto fáze zákl. cyklu vykonávaná jako separátní instrukce.

Jaké instrukce?

- ?? bitový procesor
- Registry – kolik, jakých, jak přístupných, PC, IR, PSW, adresace?, datové?
- Paměť ?? 16 bitů adresuje 64 Kslov.... slovo??
- příznaky Z (Zero), S (Sign), C (Carry), O (Overflow) a další?
- instrukce v JSI:
  - aritmetické **ADD, SUB, AND, OR, XOR, NEG, NOT, MOV, CMP**
    - **Přímý operand, nepřímý operand**
  - řídící **JMP, JO, JNO, JS, JNS, JC, JNC, JZ, JNZ, JE, JNE, JG, JL, JGE, JLE, JA, JB, JAE, JBE**
  - uložení do paměti – **ST**
  - **HALT, OUT, IND**
- reprezentace hodnot – přímý kód, doplňkový kód, nezáporná čísla
- syntaxe jazyka symbolických instrukcí

2008-Kubátová Y36SAP-ISA 31

Návrh instrukcí procesoru ADOP

- jaké
- jak kódované
- jak dlouhé
- jaká adresace operandů
- kolik registrů

2008-Kubátová Y36SAP-ISA 32

Architektura ADOP

Registry … 16 registrů dostupných programátorovi:

- R0 – R11 universálních (datových) registrů
- SP – ukazatel zásobníku
- PC – programový čítač
- PSW – stavový registr,
  - Příznaky Z … zero, C … carry, S … sign, O … overflow, ES … extended sign (znaménko 2. operandu v binárních operacích)
- ZR – obsahuje konstantní nulu

Paměť …big endian, kapacita 2^{16} B – 64KB společná pro data i instrukce

2008-Kubátová Y36SAP-ISA 33

Přesuny dat

- MOV *kam, co*

kam .... registr

co ... registr (obsah registru), přímý operand 4, 8, 16 bitový, operand nepřímo adresovaný

- ST *co, kam*
- PUSH, POP

2008-Kubátová Y36SAP-ISA 34

```
// nalezení maximálního prvku v poli
        mov r0, 0x8000          // doposud max hodnota do R0
        mov r2, pole    // pointer na začátek pole do R2
rep:    cmp r0, [r2]    // porovnání dosavadní maximální hodnoty
        jge next
        mov r0, [r2]    // nalezení nové maximální hodnoty
next:   add r2, 2       // přesun na další prvek v poli
        cmp r2, endPole // otestování zda je dosaženo konce pole
        jeq end                 // skok na konec
        jmp rep                 // opakování
end:    st r0, [max]            // uložení výsledku
        halt
max: dw 0
pole:dw 1, -13, 33, 0x7777
endPole:
```

Deklarace proměnných

Deklarace proměnných

- pseudoinstrukce
  - vyhrazení místa v paměti (pro výsledek)
  - zadání vstupních dat

data:

SHORT - jednobytový operand se znaménkem

BYTE - jednobytový operand bez znaménka

WORD - dvoubytový operand

deklarace ... DS, DB, DW

2008-Kubátová Y36SAP-ISA 36

```
// nalezení maximálního prvku v poli
        mov r0, 0x8000          // doposud max hodnota do R0
        mov r2, pole    // pointer na začátek pole do R2
rep:    cmp r0, [r2]    // porovnání dosavadní maximální hodnoty
        jge next
        mov r0, [r2]    // nalezení nové maximální hodnoty
next:   add r2, 2       // přesun na další prvek v poli
        cmp r2, endPole  // otestování zda je dosaženo konce pole
        jeq end                         // skok na konec
        jmp rep                         // opakování
end:    st r0, [max]            // uložení výsledku
        halt
max: dw 0
pole:dw 1, -13, 33, 0x7777
endPole:
```

Aritmetické

- binární
  - ADD
  - ADC
  - SUB
  - SBB
  - CMP
- unární
  - NOT
  - INC
  - DEC

```
// nalezení maximálního prvku v poli
        mov r0, 0x8000          // doposud max hodnota do R0
        mov r2, pole    // pointer na začátek pole do R2
rep:    cmp r0, [r2]    // porovnání dosavadní maximální hodnoty
        jge next
        mov r0, [r2]    // nalezení nové maximální hodnoty
next:   add r2, 2       // přesun na další prvek v poli
        cmp r2, endPole  // otestování zda je dosaženo konce pole
        jeq end                         // skok na konec
        jmp rep                         // opakování
end:    st r0, [max]            // uložení výsledku
        halt
max: dw 0
pole:dw 1, -13, 33, 0x7777
endPole:
```

paměť slabikově organizovaná, data jsou slova, proto posuv o 2 adresy

Nezáporná x záporná čísla interpretace

Př.: ADD R0, R1

Nezáporná č.			Doplňkový kód			příznaky			
R0	R1	R0+R1	R0	R1	R0+R1	CF	OF	SF	ZF
7A	78	0F2	~7A	~78	~(-E)	0	1	1	0
7A	FF	179	~7A	~(-1)	~79	1	0	0	0

2008-Kubátová Y36SAP-ISA 40

CMP - porovnání

jako SUB, ale neuloží výsledek, jen příznaky


```
// nalezení maximálního prvku v poli
        mov r0, 0x8000          // doposud max hodnota do R0
        mov r2, pole    // pointer na začátek pole do R2
rep:    cmp r0, [r2]    // porovnání dosavadní maximální hodnoty
        jge next
        mov r0, [r2]    // nalezení nové maximální hodnoty
next:   add r2, 2       // přesun na další prvek v poli
        cmp r2, endPole // otestování zda je dosaženo konce pole
        jeq end                         // skok na konec
        jmp rep                         // opakování
end:    st r0, [max]            // uložení výsledku
        halt
max: dw 0
pole:dw 1, -13, 33, 0x7777
endPole:
```

Logické

- binární
  - AND
  - OR
  - XOR
- unární
  - NEG

Skoky ... nepodmíněný skok

JMP label // label .... návěští

Př.		**MOV**	**r0, 1**	**1**
		JMP	**navesti1**	**2**
	cosi:	**ADD**	**r0, r5**	**-**
	navesti1:	**rrc**	**r0**	**3**

Skoky ... podmíněné skoky

JC *náv*	…	Je-li C=1, skok na *náv (jinak nic)*
JNC *náv*	…	Je-li C=0, skok na *náv (jinak nic)*

Př.: AX:=max(BX,CX) – nezáporná čísla

```
        MOV AX,BX
        CMP CX,BX
        JC OK
        MOV AX,CX
OK:
```

Analogicky: **JZ,JNZ,JS,JNS,JO,JNO,** JP, JNP

2008-Kubátová Y36SAP-ISA 45

Podmínky ve skocích

- Z, NZ, ... je možné použít JEG i JZ, je to to samé
- C, NC,
- S, NS
- O, NO,

Složené podmínky se používají pro vyhodnocení porovnání (CMP) resp. odečtení dvou operandů a testují logický výrazy nad příznaky:

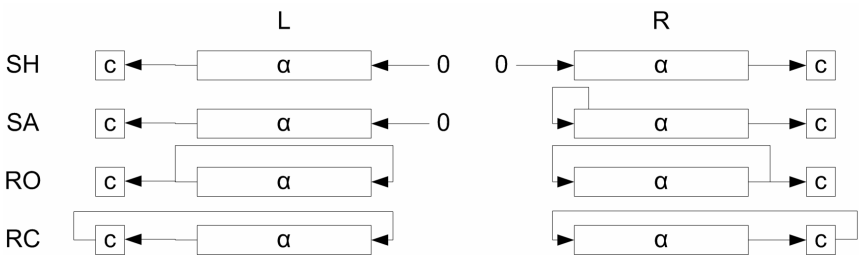
- GE (greater or equal) – !SF && !OF || SF && OF (výraz je ekvivalentní s SF == OF) ... *pro čísla v doplňovém kódu*
- LT (less then) – negace podmínky GE ... *pro čísla v doplňovém kódu*
- GT (greater then) – GE && !Z
- LE (less or equal) – negace podmínky GT,
- AT (above then) – C && !Z ... *pro čísla nezáporná*
- BE, (below equal) – negace podmínky AT ... *pro čísla nezáporná*
- TRUE – podmínka vždy splněna

```
// nalezení maximálního prvku v poli
        mov r0, 0x8000          // doposud max hodnota do R0
        mov r2, pole    // pointer na začátek pole do R2
rep:    cmp r0, [r2]    // porovnání dosavadní maximální hodnoty
        jge next
        mov r0, [r2]    // nalezení nové maximální hodnoty
next:   add r2, 2       // přesun na další prvek v poli
        cmp r2, endPole // otestování zda je dosaženo konce pole
        jeq end                 // skok na konec
        jmp rep                 // opakování
end:    st r0, [max]            // uložení výsledku
        halt
max: dw 0
pole:dw 1, -13, 33, 0x7777
endPole:
```

Posuvy

- SHL,
- SHR,
- ASR – aritmetický posun vpravo,
- RRC,
- RLC

Např. rotace (RC) nebo posuv (SH) vybraného bitu do C a následný skok podle hodnoty C



2008-Kubátová Y36SAP-ISA 48

Příklad

Napište program v JSI ADOP, který nalezne největší číslo v poli.


```
// nalezení maximálního prvku v poli
        mov r0, 0x8000          // doposud max hodnota do R0
        mov r2, pole    // pointer na začátek pole do R2
rep:    cmp r0, [r2]    // porovnání dosavadní maximální hodnoty
        jge next
        mov r0, [r2]    // nalezení nové maximální hodnoty
next:   add r2, 2       // přesun na další prvek v poli
        cmp r2, endPole  // otestování zda je dosaženo konce pole
        jeq end                         // skok na konec
        jmp rep                         // opakování
end:    st r0, [max]            // uložení výsledku
        halt
max: dw 0
pole:dw 1, -13, 33, 0x7777
endPole:
```

Simulátor

http://service.felk.cvut.cz/jws/proc/procwww/

umožňuje psát programy v JSI, překládat, krokovat, spouštět a sledovat změny obsahu registrů a paměti