

Úvod do organizace počítače

Alternativní architektury
Aritmeticko/logické operace

Aritmeticko-logická jednotka
(ALU) a podpora ALU u MIPS

Přehled koncepce (opakování z minulé hodiny)

- Přidělení paměti proměnným
 - Stackové rámce - frame (volající/volaný), statické, heap
- Pointery
 - Adresování paměti
 - Předávání argumentů: pole, struktury
 - Efektivní použití pointerů (aritmetika pointerů)
 - Problémy
 - Chybné paměťové reference (segmentation fault)
 - „Díry“ v paměti

Přehled

- Alternativní návrhy ISA
- Principy návrhu (konvence)
- Rovnice výkonu CPU
- RISC vs. CISC
- Historická perspektiva
- PowerPC a 80x86

Přehled (pokračování)

- Hardwarové komponenty (bloky)
- Návrh ALU
- Implementace ALU
- 1-bitová ALU
- 32-bitová ALU

Architektura & Mikroarchitektura (opakování)

- **Architektura:**
Souhrn vlastností procesoru (nebo systému) jak se jeví „uživateli“
 - Uživatel: „binární program“ běžící na procesoru nebo programátor na úrovni assembleru
- **Mikroarchitektura:**
Souhrn vlastností implementace (které uživatel na instrukční úrovni nevidí)
Vlastnosti, které se mění (časování, výkon, technologie), patří do mikroarchitektury
- Snahou každé firmy je navrhnout architekturu, která dokáže „přežít“ delší časové údobí. Mikroarchitektura se v tomto údobí může měnit.

Architektury počítačů

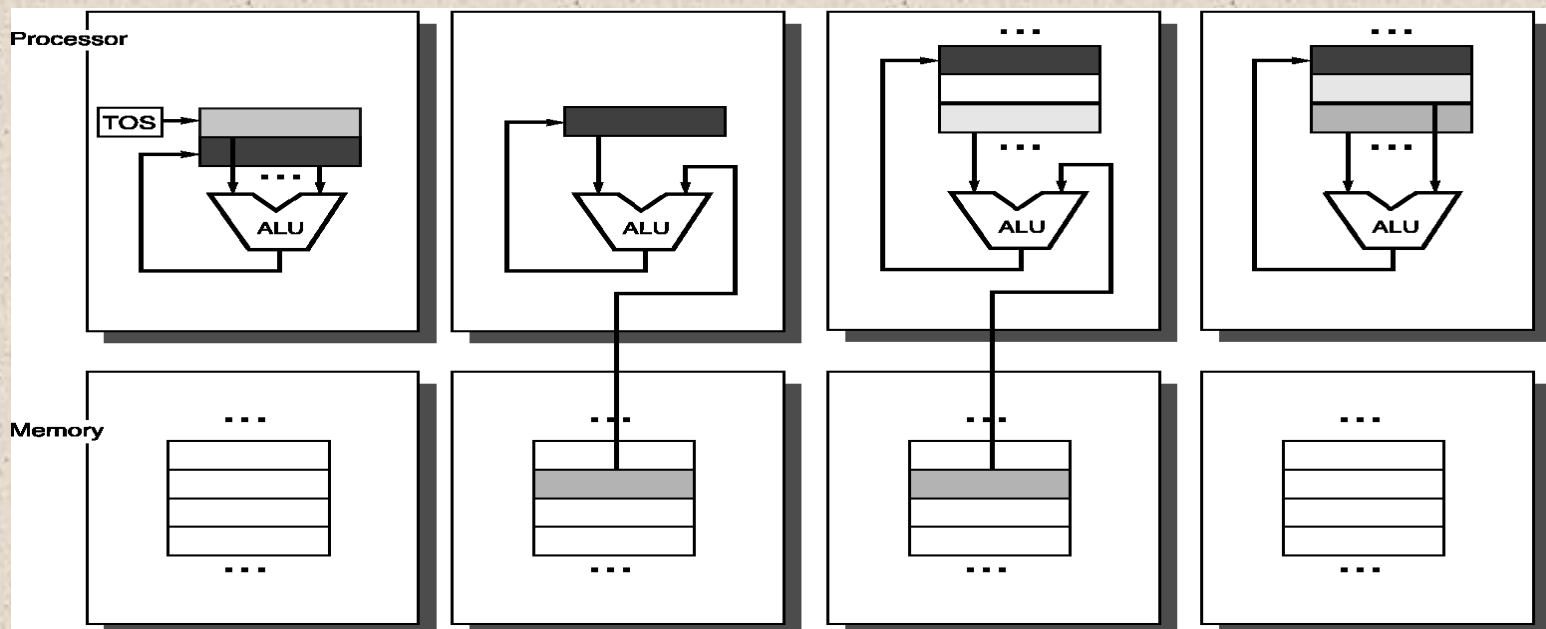
Orientace architektury na:

- Stack
 - Žádné registry (jednoduché kompilátory, kompaktní kódování)
- Akumulátor
 - Drahý hardware => pouze jeden registr
 - Akumulátor: jeden z operandů a výsledek
 - Adresní režimy vztažené na operand v hlavní paměti
- Registry se speciálním použitím (např. I8086)
 - Registr-paměť
 - Registr-registr (load-store)
- Architektura počítačů pro HLL

Ilustrace typů architektury (ISA)

Assembler pro **C:=A+B:**

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C,R1	Add R3,R1,R2
Pop C			Store C,R3



Illustrate typů architekt (ISA)

C = A + B;

Accumulator

Load AddressA
Add AddressC
Store AddressC

Stack

Push AddressA
Push AddressB
Add
Pop AddressC

Load-Store

Load R1, AddressA
Load R2, AddressB
Add R3, R1, R2
Store R3, AddressC

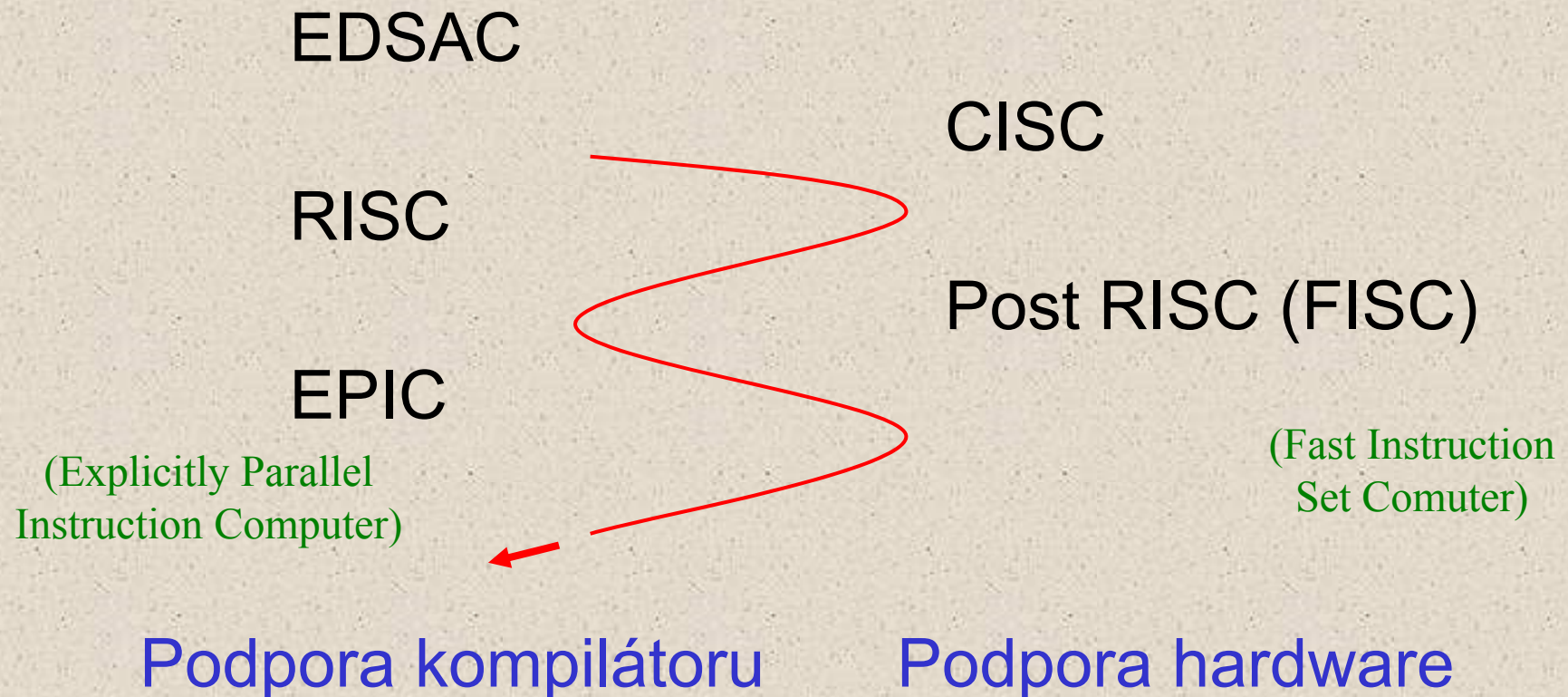
Memory-Memory

Add AddressC, AddressB, AddressA

$$\text{CPU}_{\text{time}} = \text{IC} * \text{CPI} * \text{Cycle_time}$$

Trendy ISA

- Technologické trendy HW a kompilátorů
 - Mez mezi hardware/software „osciluje“



Architektura RISC

- RISC - počítače s redukováným souborem instrukcí
- Filozofie návrhu
 - Load/store instrukce pro práci s pamětí
 - Instrukce pevné délky
 - Tříadresová architektura
 - Mnoho registrů
 - Jednoduché adresní módy
 - Instrukční pipelining
- Mnohé myšlenky, použité v moderních počítačích pocházejí z architektury CDC 6600 (1963)

PowerPC

- Podobné MIPS: 32 registrů, 32-bitové instrukce, RISC
- Rozdíly (porovnání: jednoduchost vs. common case)
 - Indexové adresování
 - Příklad: `lw $t1,$a0+$s3` # $t1$ =Memory[$a0+s3$]
totéž u MIPS: `add $t0, $a0, $s3;`
`lw $t1,0($t0)`
 - Adresní módy s aktualizací
 - Aktualizace registru jako část „load“ (průchod polem)
 - Příklad: `lwu $t0,4($s3)`
totéž u MIPS: `lw $t0,4($s3);`
`addi $s3,$s3,4`
 - Speciální instrukce
 - Load multiple/store multiple: až 32 slov v jedné instrukci
 - Speciální registr - čítač
`bc Loop, $ctr!=0` *#decrement counter, if not 0 goto loop*
totéž u MIPS: `addi, $t0, $t0, -1;`
`bne $t0, $zero, Loop`

Mezníky architektury 80x86

- 1978: 8086, 16 bit architektura (64KB), žádné GPR
- 1980: 8087 FP koprocessor, 60 + instrukcí, 80-bit stack, žádné GPR
- 1982: 80286, 24-bitová adresa, ochrany paměti
- 1985: 80386, 32 bitů, nové adresní režimy, 8 GPR
- 1989-1995: 80486, Pentium, Pentium Pro - přidáno několik nových instrukcí (pro zvýšení výkonu)
- 1997: MMX + 57 instrukcí (SIMD)
- 1999: PIII + 70 „multimediálních“ instrukcí
- 2000: P4 +144 „multimediálních“ instrukcí

Zlatá pouta kompatibility vzhůru => „tuto architekturu je těžké vysvětlit a nemožné mít rád“ (citát)

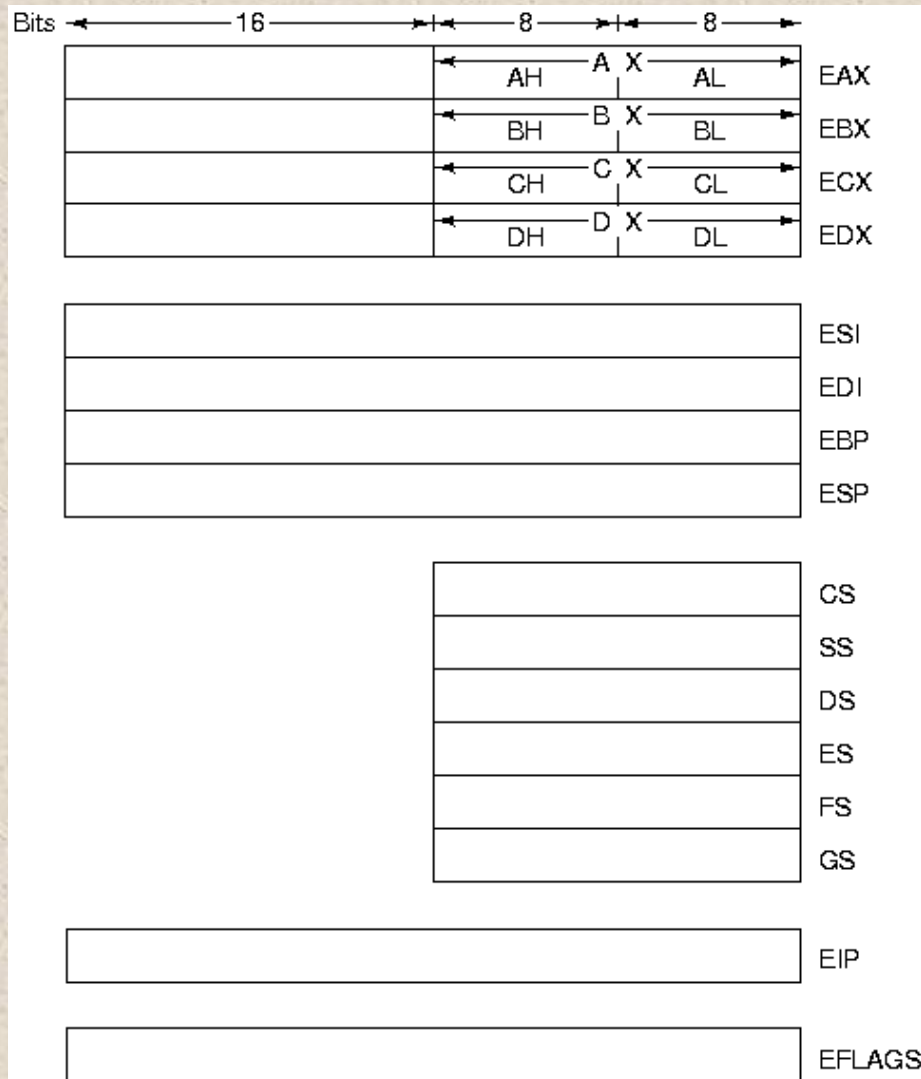
Architektura X86

- **Dvouadresní architektura**
 - Jeden z operandů je zároveň cílem
add \$s1,\$s0 # s0=s0+s1 (**C:** a += b;)
 - Výhoda: malé instrukce → malý kód → **rychlejší**
- **Architektura typu Register-Memory**
 - Jeden operand může být v paměti; druhý je v registru
add 12(%gp),%s0 # s0=s0+Mem[12+gp]
 - Výhoda : méně instrukcí → menší rozsah kódu
- **Instrukce proměnné délky** (1 až 17 bytů)
 - Malý rozsah kódu (o 30% menší)
 - Vyšší účinnost instrukční cache
 - Instrukce mohou zahrnovat 8- nebo 32-bitové přímé operandy

Vlastnosti X86

- Operační módy: reálný (8088), virtuální a chráněný
- Čtyři úrovně ochrany
- Paměť
 - Adresní prostor: 16,384 segmentů (4GB)
 - Uložení bytů ve slově - Little endian
- 8 32-bitových registrů (16-bit, 8086 jména, prefix **e**):
 - eax, ecx, edx, ebx, esp, ebp, esi, edi
- Datové typy
 - **S**igned/**U**nsigned integer (8, 16, a 32 bitů)
 - **B**inary **C**oded **D**ecimal integer čísla
 - **F**loating **P**oint (32 a 64 bitů)
- Floating point jednotka používá oddělený stack

Registry X86



Main arithmetic register

Pointers (memory addresses)

Loops

Multiplication and division

Pointer to source string

Pointer to destination string

Base of the current stack frame (\$fp)

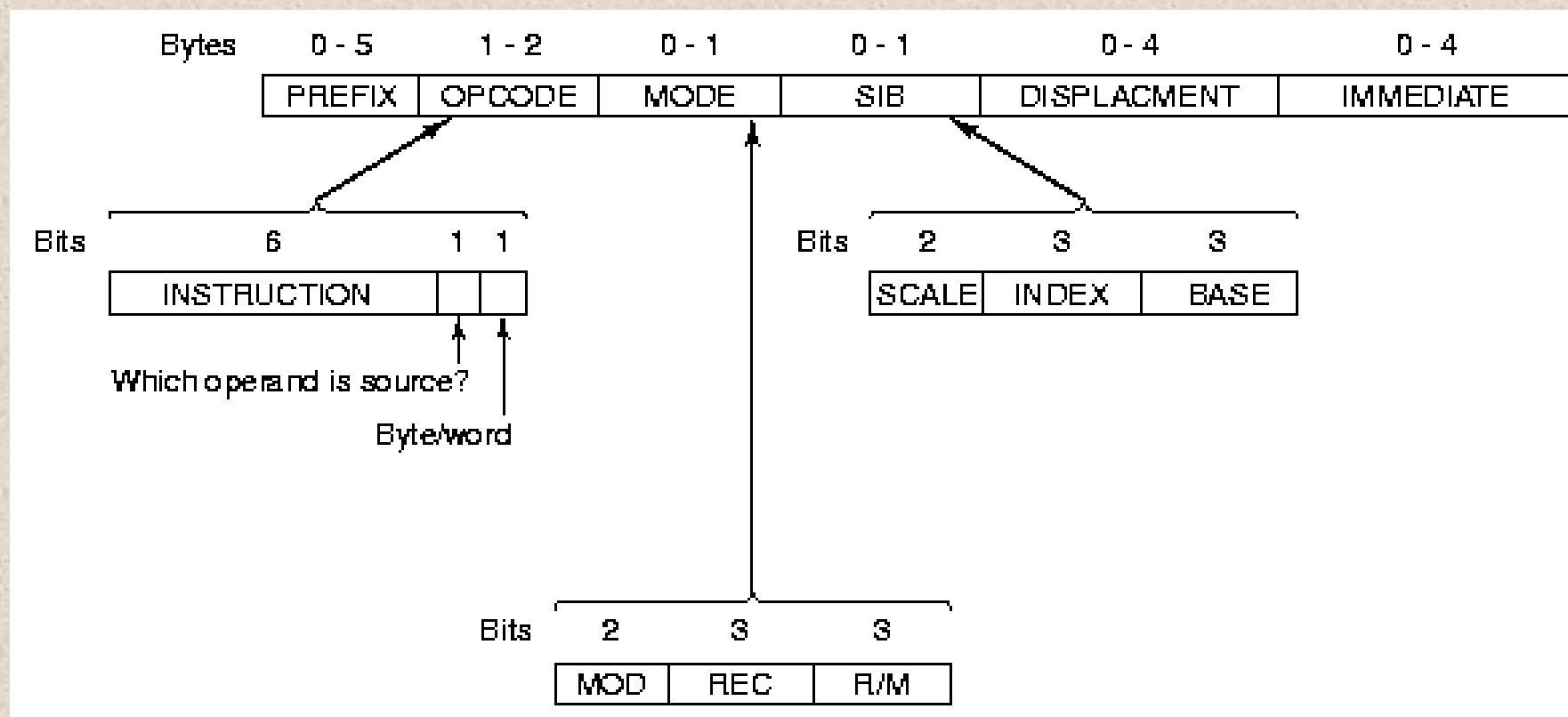
Stack pointer

Support for 8088 attempt to address 2^{20} bytes using 16-bit addresses

Program counter

Processor State Word

Instrukční formáty X86

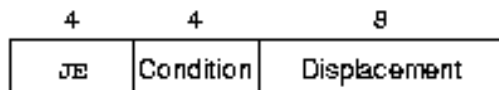


- **Velmi složité a neregulární**

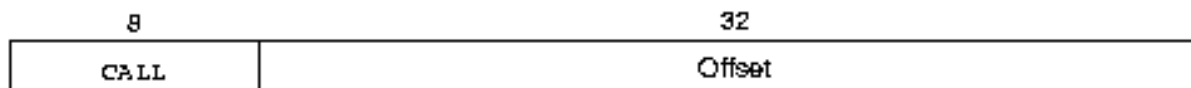
- Šest polí s proměnnou délkou
- Dalších pět polí se může vyskytnout

Příklady instrukčních formátů X86

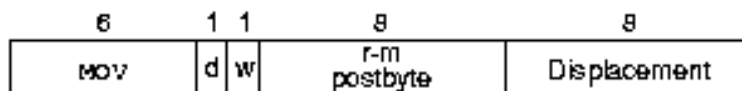
a. JE EIP + displacement



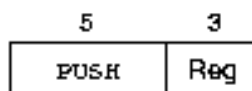
b. CALL



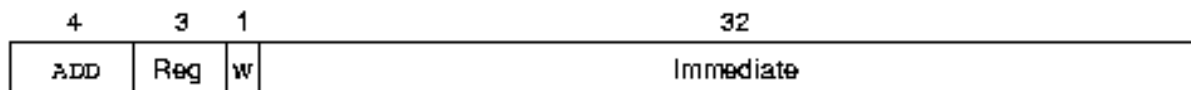
c. MOV EBX, [EDI + 45]



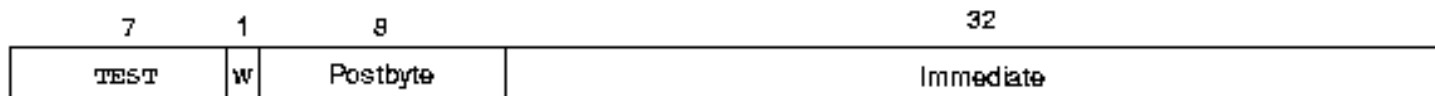
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



Instrukce pro čísla integer

- Řídící
 - JNZ, JZ
 - JMP
 - CALL
 - RET
 - LOOP
- Datové přenosy
 - MOV
 - PUSH, POP
 - LES
- Aritmetické
 - ADD, SUB
 - CMP
 - SHL, SHR, RCR
 - CBW
 - TEST
 - INC, DEC
 - OR, NOR
- Operace s řetězcí
 - MOVS
 - LODS

Příklady instrukcí X86

- `leal` (load effective address)
 - Vypočítá adresu podobně jako `load` ale zapíše **adresu** do registru
 - Určí 32-bit adresu:
`leal -4000000(%ebp),%esi #esi = ebp - 4000000`
- Paměťový stack je součástí instrukčního souboru
 - `call label` (`esp-=4; M[esp]=eip+5; eip = label`)
 - `push` uloží hodnotu do stacku, inkrementuje `esp`
 - `pop` vezme hodnotu ze stacku, dekrementuje `esp`
- `incl, decl` (increment, decrement)

`incl %edx # edx = edx + 1`

Dekódování adresních režimů

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

- **Vysoce neregulární, neortogonální adresní módy**
 - Instrukce v 16-bitovém nebo 32-bitovém módu?
 - Zdaleka ne všechny módy lze aplikovat na všechny instrukce
 - Zdaleka ne všechny registry mohou být užity ve všech módech

Adresní režimy (módy)

- **Base reg + offset (jako MIPS)**
 - `movl -80000044(%ebp), %eax`
- **Base reg + index reg (2 registry formují adresu)**
 - `movl (%eax,%ebx),%edi`
`edi = Mem[ebx + eax]`
- **Scaled reg + index (posuv registru o 1,2)**
 - `movl(%eax,%edx,4),%ebx`
`ebx = Mem[edx*4 + eax]`
- **Scaled reg + index + offset**
 - `movl 12(%eax,%edx,4),%ebx`
`ebx = Mem[edx*4 + eax + 12]`

Podpora větvení

- Namísto porovnání registrů používá x86 speciální 1-bitové registry nazývané “podmínkové kódy“, které vytvářejí **vedlejší efekty** operací ALU
 - S - Sign Bit
 - Z - Zero (výsledek je celý roven 0)
 - C - Carry Out
 - P - Parity: nastaven na 1, je-li počet jedniček v osmi bitech výsledku operace vpravo sudý
- Instrukce podmíněných skoků používají tyto podmínkové kódy pro všechna porovnání: <, <=, >, >=, ==, !=

Smyčka While

```
while (save[i]==k)
    i = i + j;
```

X86

(i,j,k => %edx, %esi, %ebx)

```
    leal    -400(%ebp),%eax
.Loop: cmpl  %ebx,(%eax,%edx,4)
      jne   .Exit
      addl  %esi,%edx
      j     .Loop
.Exit:
```

MIPS

(i,j,k => \$s3, \$s4, \$s5)

```
Loop: { sll   $t1, $s3, 2
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        add   $s3, $s3, $s4
        j     Loop
Exit:
```


PIII, P4 a AMD

- PC World magazine, Nov. 20, 2000
 - WorldBench 2000 benchmark (aplikace obchodního charakteru)
 - P4 score @ 1.5 GHz: 164 (větší hodnota znamená lepší)
 - PIII score @ 1.0 GHz: 167
 - AMD Athlon @ 1.2 GHz: 180
 - (Mediální aplikace vycházejí lépe na P4 vs. PIII)
- Proč? => rovnice výkonu CPU
 - $\text{Čas} = \text{Počet instrukcí} \times \text{CPI} \times 1/\text{Frekvence_hodin}$
 - Počet instrukcí je stejný jako pro x86
 - Frekvence_hodin : P4 > Athlon > PIII
 - Proč může být P4 pomalejší?
 - Střední CPI procesoru P4 musí být horší než Athlonu a PIII

Souhrn

- Složitost instrukcí je pouze jeden faktor
 - menší počet instrukcí vs. vyšší CPI / nižší frekvence hodin
- Principy návrhu:
 - jednoduchost vyžaduje regularitu
 - menší je rychlejší
 - dobrý návrh vyžaduje dobré kompromisy
 - společné části stavět rychle
- Instruction Set Architecture
 - velmi důležitá abstrakce!

Nové – Aritmeticko/logické operace

- Aritmeticko-logická jednotka (ALU)
 - Jádro počítače
 - Provádí aritmetické a logické operace nad daty
- Problémy počítačové aritmetiky
 - Reprezentace čísel
 - Integer a floating point
 - Omezená přesnost (overflow / underflow)
 - Algoritmy použité pro základní operace
- Vlastnosti reprezentace čísel
 - Jedna nula
 - Číslo rozloženo symetricky kolem nuly
 - Efektivní hardwarová implementace algoritmů
- Dvojkový doplněk (algoritmus): negace čísla a přičtení jedničky

Souhrn

N decimal	(+N) Positive	(-N) Sign/magnitude	(-N) 1's complement	(-N) 2's complement
0	00000000	10000000	11111111	00000000
1	00000001	10000001	11111110	11111111
2	00000010	10000010	11111101	11111110
3	00000011	10000011	11111100	11111101
4	00000100	10000100	11111011	11111100
5	00000101	10000101	11111010	11111011
6	00000110	10000110	11111001	11111010
7	00000111	10000111	11111000	11111001
8	00001000	10001000	11110111	11111000
9	00001001	10001001	11110110	11110111
10	00001010	10001010	11110101	11110110
20	00010100	10010100	11101011	11101100
50	00110010	10110010	11001101	11001110
100	01100100	11100100	10011011	10011100
127	01111111	11111111	10000000	10000001
128	NA	NA	NA	10000000

Sčítání

- $5_{10} + 6_{10}$

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ (5_{10}) \\
 +\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110\ (6_{10}) \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011\ (11_{10})
 \end{array}$$

Přenosy

$$\begin{array}{r}
 \dots (0) \quad (1) \quad (0) \quad (0) \quad (0) \\
 \dots 0 \quad 0 \quad 1 \quad 0 \quad 1 \\
 + \dots 0 \quad 0 \quad 1 \quad 1 \quad 0 \\
 \hline
 \dots 0 \quad (0)1 \quad (1)0 \quad (0)1 \quad (0)1
 \end{array}$$

Odčítání

- $12_{10} - 5_{10}$

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ (12_{10}) \\ -\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ (5_{10}) \\ \hline =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ (7_{10}) \end{array}$$

- $12_{10} - 5_{10} = 12_{10} + (-5_{10})$

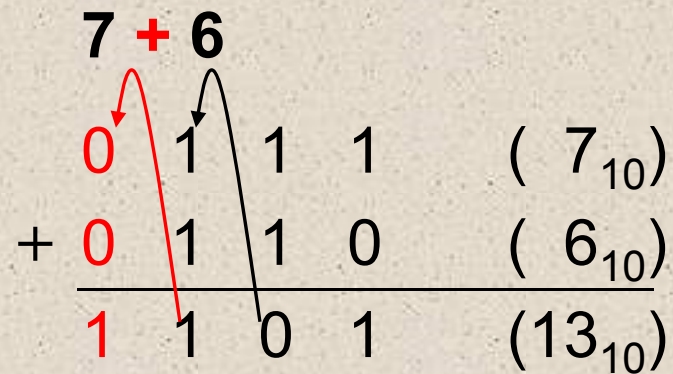
$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ (12_{10}) \\ +\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011\ (-5_{10}) \\ \hline =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ (7_{10}) \end{array}$$

Přetečení

- Množina čísel vzhledem k operacím $+$ $-$ $*$ $/$ není uzavřená
- Přetečení
 - Výsledek operace nelze zobrazit v původním rozsahu (na 32 bitech)
- Přetečení nemůže nastat
 - Sčítání: mají-li sčítanci opačná znaménka
 - Odčítání: mají-li operandy stejná znaménka
- Detekce přetečení
 - Výsledek potřebuje 33 bitů
 - Sčítání: liší se C_{n+1} a C_n
 - Odčítání: promyslete sami

Příklady

- 4 bity (na rozdíl od 32 u MIPS) => lze zobrazit čísla $[-8 : 7]$

$$\begin{array}{rcccccl} & \mathbf{7 + 6} & & & & \\ & \textcolor{red}{0} & 1 & 1 & 1 & (7_{10}) \\ + & \textcolor{red}{0} & 1 & 1 & 0 & (6_{10}) \\ \hline & \textcolor{red}{1} & 1 & 0 & 1 & (13_{10}) \end{array}$$


$$\begin{array}{rcccccl} & \mathbf{-7 + -6} & & & & \\ & \textcolor{red}{1} & 0 & 0 & 1 & (-7_{10}) \\ + & \textcolor{red}{1} & 0 & 1 & 0 & (-6_{10}) \\ \hline & \textcolor{red}{0} & 0 & 1 & 1 & (-13_{10}) \end{array}$$

$$\begin{array}{rcccccl} & \mathbf{-7 - 6 = -7 + -6} & & & & \\ & \textcolor{red}{1} & 0 & 0 & 1 & (-7_{10}) \\ + & \textcolor{red}{1} & 0 & 1 & 0 & (-6_{10}) \\ \hline & \textcolor{red}{0} & 0 & 1 & 1 & (-13_{10}) \end{array}$$

Podmínky přetečení

Operace	Operand A	Operand B	Výsledek
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Podpora MIPS

- U MIPS nastane výjimka, vznikne-li přetečení
 - Výjimky (**interrupty**) pracují jako volání procedury
 - Registr **EPC** uloží adresu „závadné“ instrukce
 - **mfc0 \$t1, \$epc** # moves contents of EPC to \$t1
 - **Neexistuje podmíněný skok s testem přetečení**
- Aritmetika dvojkového doplňku (add, addi a sub)
 - Výjimka při přetečení
- Aritmetika bez znaménka (addu a addiu)
 - Při přetečení nevznikne výjimka
 - Používáno při výpočtu adres
- Kompilátory
 - C ignoruje přetečení (vždy používá addu, addiu, subu)
 - Fortran používá vhodné instrukce

Detekce přetečení

C_{n-1}	A_{n-1}	B_{n-1}	S_{n-1}	C_n	OF
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	0

Discarded

- Test **MSB**
- P: kladné; N: záporné
- $N + N = N$
 - $P + P = P$
 - $P+N$ nebo $N+P$ vždy „spadne“ do zobrazovaného intervalu
 - Např. $-128+P$ nemůže být menší než -128 nebo větší než 127
- Problém nastává pro:
 - $N+N = P$
 - $P+P = N$

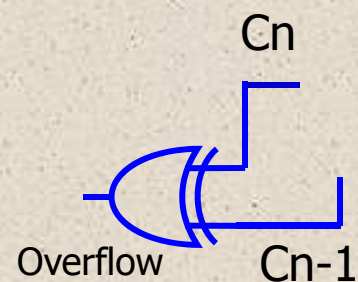
Detekce přetečení

C_{n-1}	A_{n-1}	B_{n-1}	S_{n-1}	C_n	OF
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	0

$$OF = \overline{S_{n-1}} A_{n-1} B_{n-1} + S_{n-1} \overline{A_{n-1}} \overline{B_{n-1}}$$

nebo

$$OF = C_{n-1} \oplus C_n$$



**n-bitová sčítačka/
odčítačka**

Co způsobí přetečení (Overflow)

- Nastane výjimka (interrupt)
 - Řadič způsobí skok na předem stanovenou adresu obsluhy výjimky
 - Adresa přerušení (kde bylo přerušeno) se uloží kvůli možnému dalšímu pokračování
- Detaily závisejí na software (role OS)
- Detekce přetečení není požadována vždy
 - nové instrukce MIPS: addu, addiu, subu

Pozn.: addiu pracuje s rozšířením znaménka!

- stejně tak addi, vyjma *no overflow exception*

Pozn.: sltu, sltiu pro porovnání bez znaménka

Podmíněné skoky při přetečení

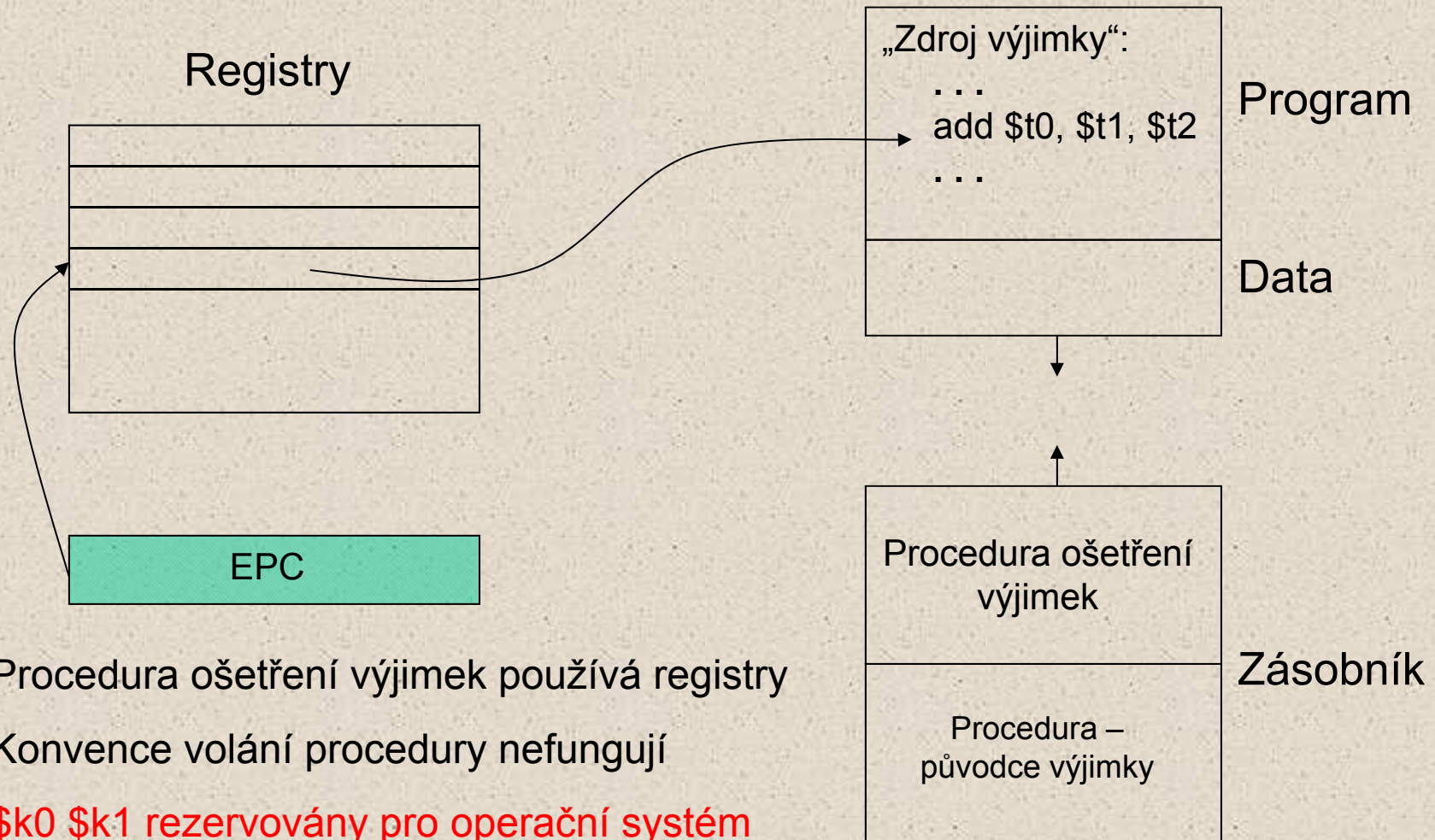
Sčítání se znaménkem – softwarová detekce přetečení

```
addu    $t0, $t1, $t2      # add but do not trap
xor      $t3, $t1, $t2      # check if sign differ
slt      $t3, $t3, $0        # $t3 = 1 if signs differ
bne      $t3, $0, NO_OVFL    # signs of t1, t2 different
xor      $t3, $t0, $t1      # sign of sum (t0) different?
slt      $t3, $t3, $0        # $t3 = 1 if sum has different sign
bne      $t3, $0, OVFL       # go to overflow
```

Sčítání bez znaménka (range = $[0 : 2^{32} - 1]$ => $\$t1 + \$t2 \leq 2^{32} - 1$)

```
addu    $t0, $t1, $t2      # $t0 contains the sum
nor      $t3, $t1, $0        # negate $t1 ($t3 = NOT $t1)
sltu     $t3, $t3, $t2      #  $2^{32} - 1 - t1 < t2$ ?
bne      $t3, $0, OVFL       #  $t1 + t2 > 2^{32} - 1$  => overflow
```


Registry \$k0 a \$k1



Logické operace

- Operace nad poli bitů v rámci 32-bitových slov
 - Znaky (8 bitů)
 - Bitová pole (v C)
- Logické operace
 - **sll** posuv vlevo
 - **srl** posuv vpravo
 - **and, andi** bitové AND
 - **or, ori** bitové OR
- Bitové operátory - operandy jsou *bitové vektory*

Bitová pole v C

```
struct {
    unsigned int ready:      1;
    unsigned int enable:     1;
    unsigned int receivedByte: 8;
} receiver;
int data = receiver.receiverByte;
receiver.ready = 0;
receiver.enable = 1;
```

```
#$s0: data; $s1: receiver

sll    $s0, $s1, 22
srl    $s0, $s0, 24
andi   $s1, $s1, 0xfffe
ori    $s1, $s1, 0x0002
```



Hardwarové komponenty (bloky)

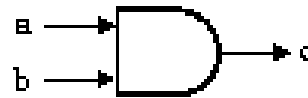
- ALU se staví z komponent nízké úrovně (implementace)
 - z logických hradel
- **Hradlo** (přehled)
 - Hardwarový prvek, který zpracovává několik vstupů a generuje jeden výstup
 - Může být reprezentován pravdivostní tabulkou a nebo logickou rovnicí
 - Hradla se vytvářejí z tranzistorů na křemíku
- Různé hardwarové komponenty:
 - Hradlo And
 - Hradlo Or
 - Invertor (not)
 - Multiplexor (mux)

Poznámka:

Označení „hradlo“ není zcela korektní, protože všechny jeho vstupy jsou rovnocenné.

Základní elektronické prvky

1. AND gate ($c = a \cdot b$)



a	b	$c = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate ($c = a + b$)



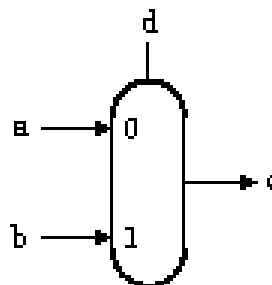
a	b	$c = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)



a	$c = \bar{a}$
0	1
1	0

4. Multiplexor (MUX)
 (if $d = 0$
 $c = a$;
 else
 $c = b$)



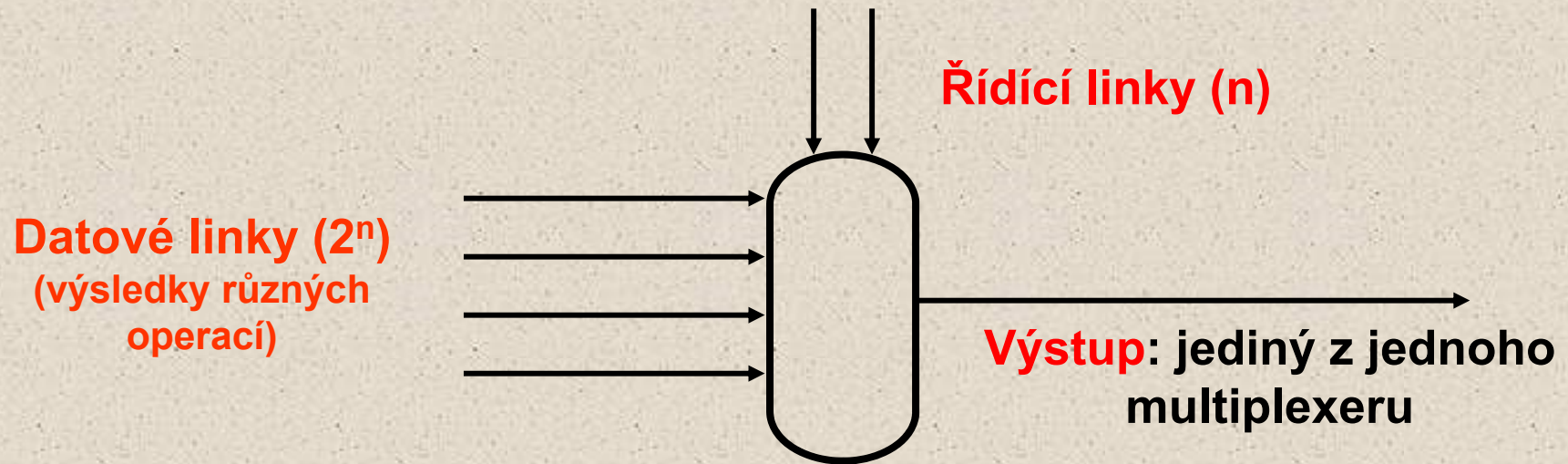
d	c
0	a
1	b

n control bits select
 between 2^n data bits

Modulární návrh ALU

- Fakta
 - Stavební bloky pracují s **individuálními** (I/O) bity
 - ALU pracuje se 32-bitovými registry
 - ALU provádí množinu základních operací (+, -, *, /, posuvy, atd.)
- Principy
 - Sestavit 32 samostatných 1-bitových ALU
 - Sestavit samostatné hardwarové bloky pro každou úlohu
 - Všechny operace se provádí paralelně
 - Pro výběr aktuální operace se použije multiplexor
- Výhody
 - Snadné připojení další operace (instrukce)
 - Připojení nových datových linek k multiplexoru; informovat „řízení“ o změně

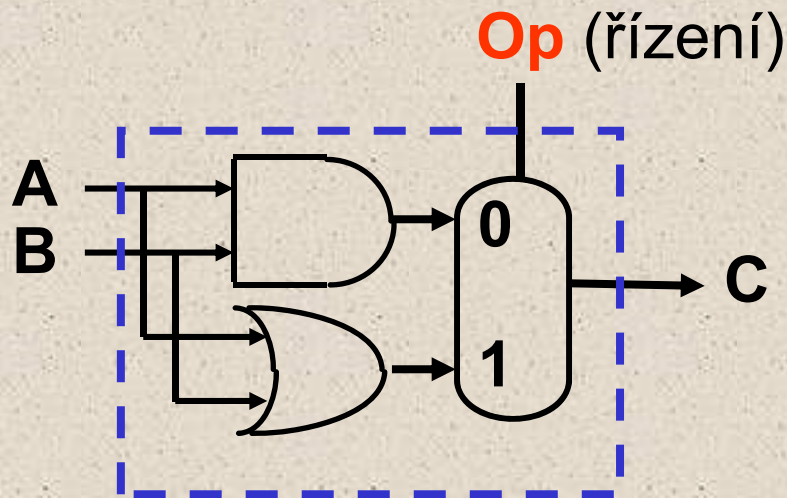
Implementace ALU



1. 32-bitová ALU použije 32 multiplexerů (pro každý výstupní signál jeden)
2. Projít instrukční soubor a pro implementaci odpovídajících operací přidat datové (a řídící) linky.

Jednobitové logické instrukce

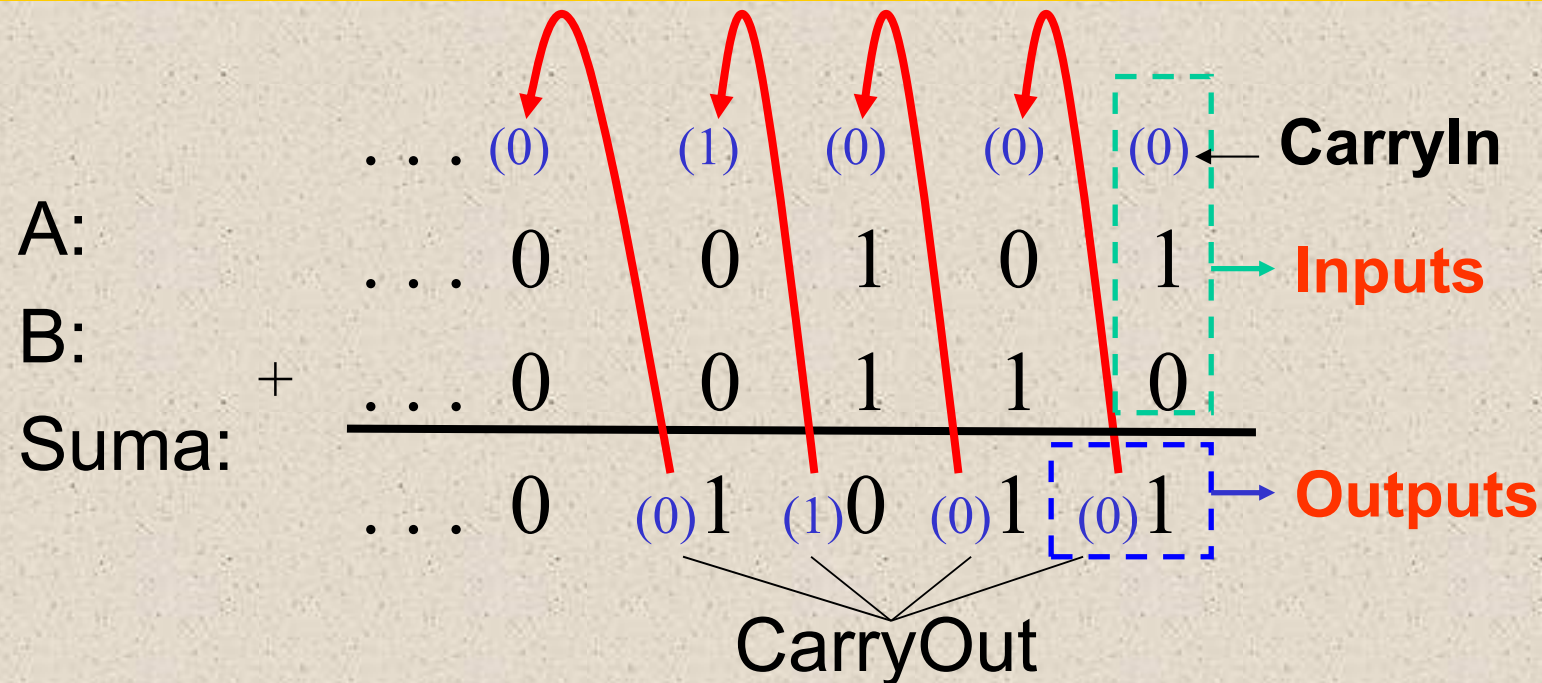
- Přímé mapování na hardwarové komponenty
 - instrukce AND
 - Jedna datová linka pochází z pouhého hradla AND
 - instrukce OR
 - Další datová linka pochází z pouhého hradla OR



Definice

Op	C
0	A and B
1	A or B

Jednobitová úplná sčítačka



- Každý „bit“ sčítačky má:

- Tři vstupní signály:

$A_i, B_i, \text{CarryIn}_i$

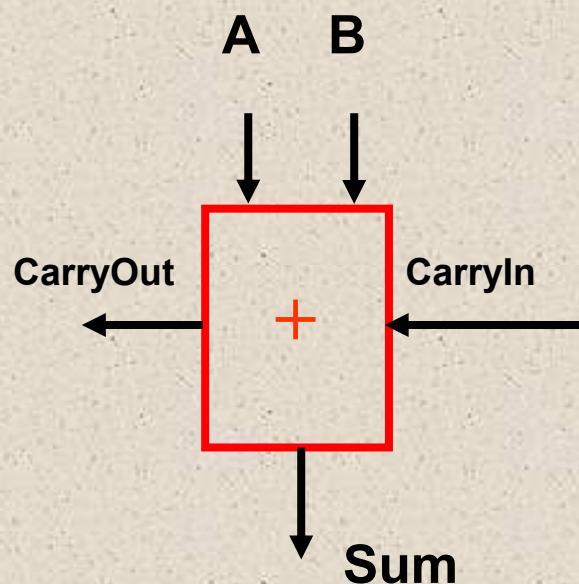
- Dva výstupní signály:

$\text{Sum}_i, \text{CarryOut}_i$

($\text{CarryIn}_{i+1} = \text{CarryOut}_i$)

Pravdivostní tabulka úplné sčítačky

Symbol



Definice

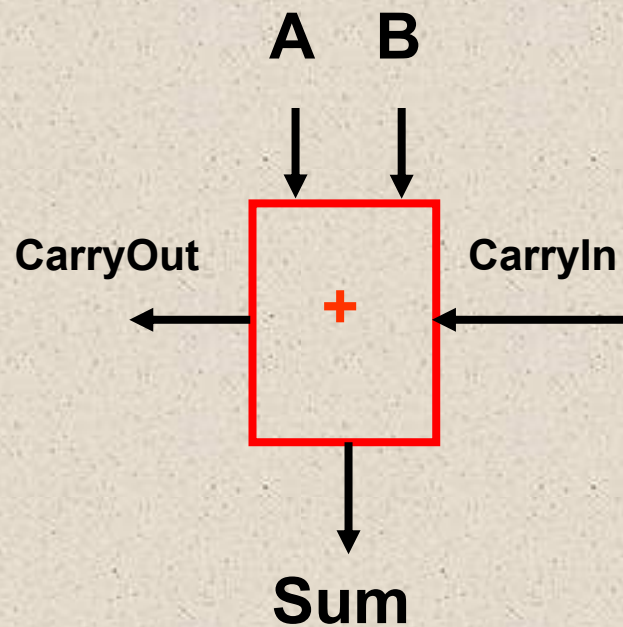
A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\text{CarryOut} = (A' \cdot B \cdot \text{CarryIn}) + (A \cdot B' \cdot \text{CarryIn}) + (A \cdot B \cdot \text{CarryIn}') + (A \cdot B \cdot \text{CarryIn}) = (B \cdot \text{CarryIn}) + (A \cdot \text{CarryIn}) + (A \cdot B)$$

$$\text{Sum} = (A' \cdot B' \cdot \text{CarryIn}) + (A' \cdot B \cdot \text{CarryIn}') + (A \cdot B' \cdot \text{CarryIn}') + (A \cdot B \cdot \text{CarryIn})$$

Ekvivalentní zápis rovnic úplné sčítačky

Symbol



$$\text{CarryOut} = \underline{A*B} + \underline{\text{CarryIn} * (A+B)}$$

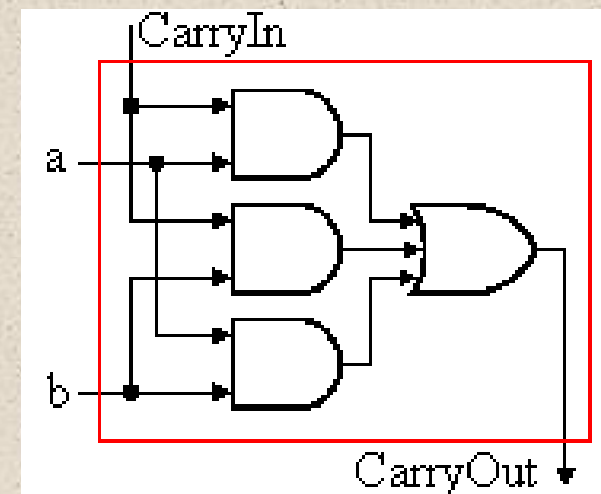
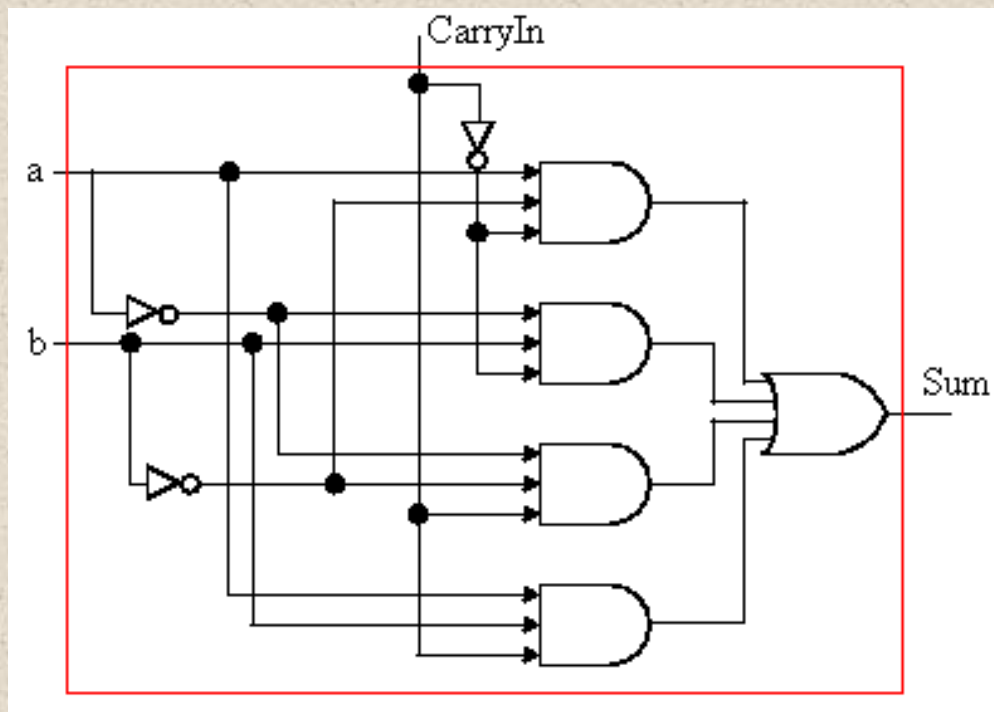
Generační funkce G

Přenosová funkce P

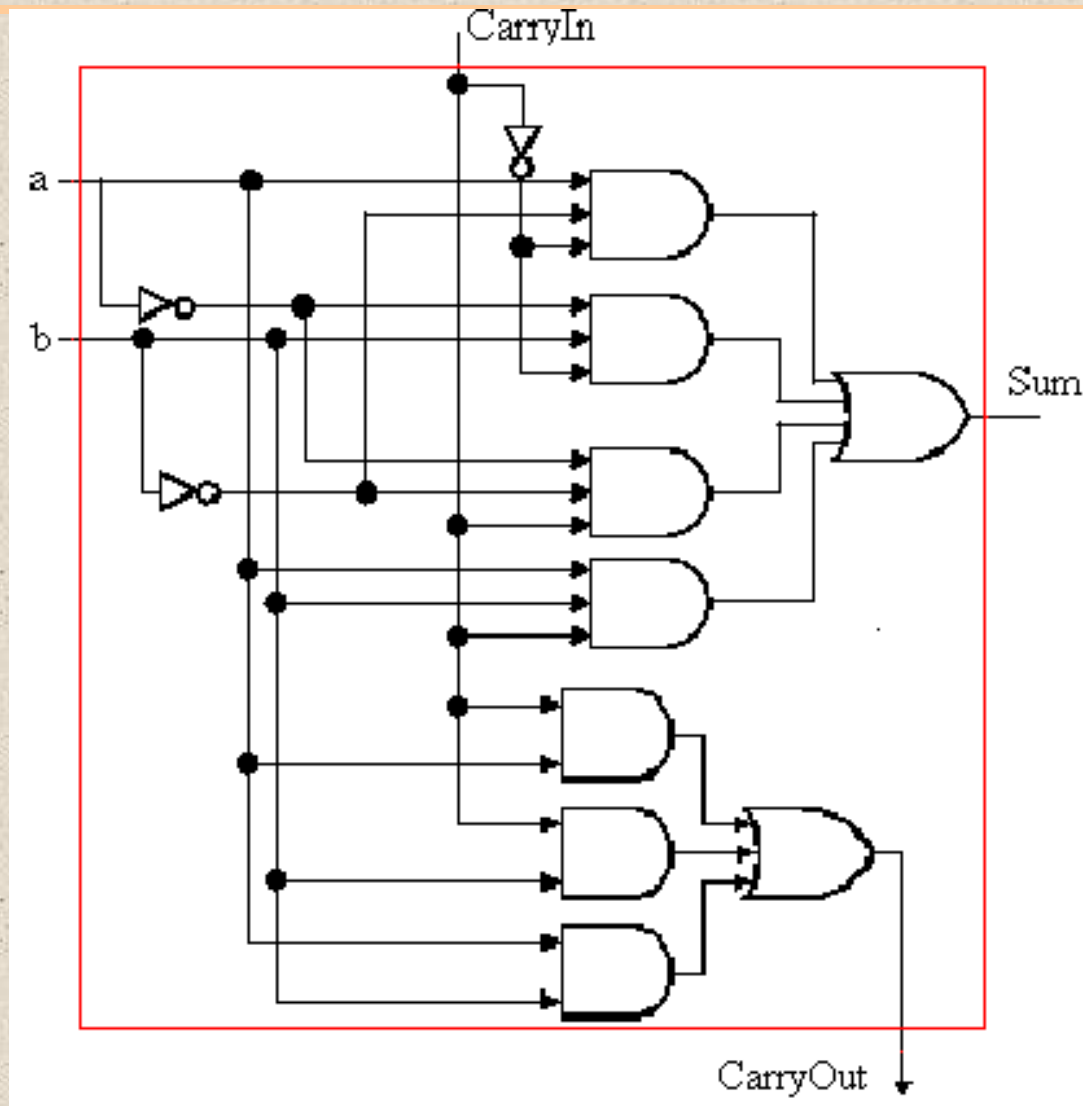
$$\text{Sum} = (A'*B'*\text{CarryIn}) + (A'*B*\text{CarryIn}') + (A*B'*\text{CarryIn}') + (A*B*\text{CarryIn})$$

Obvody úplné sčítačky (1/2)

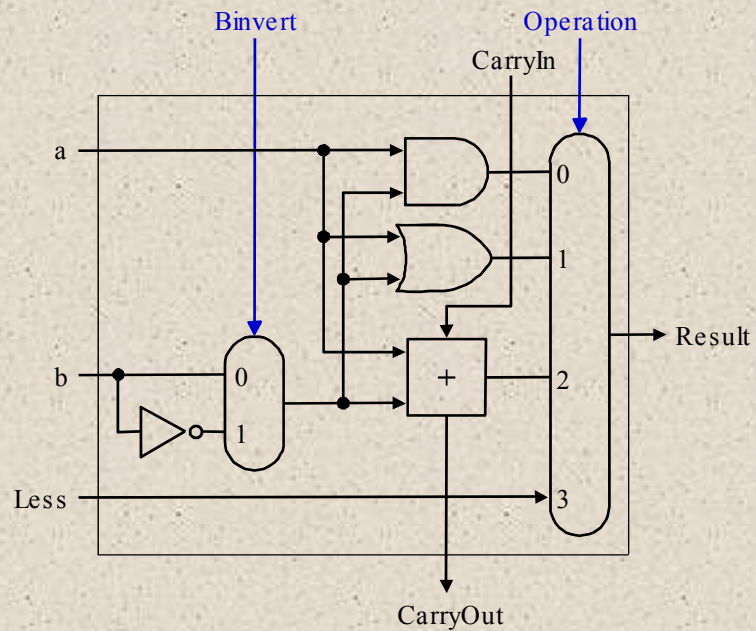
1. Sestavení funkce Sum
2. Sestavení funkce CarryOut
3. Propojení signálů se stejným jménem



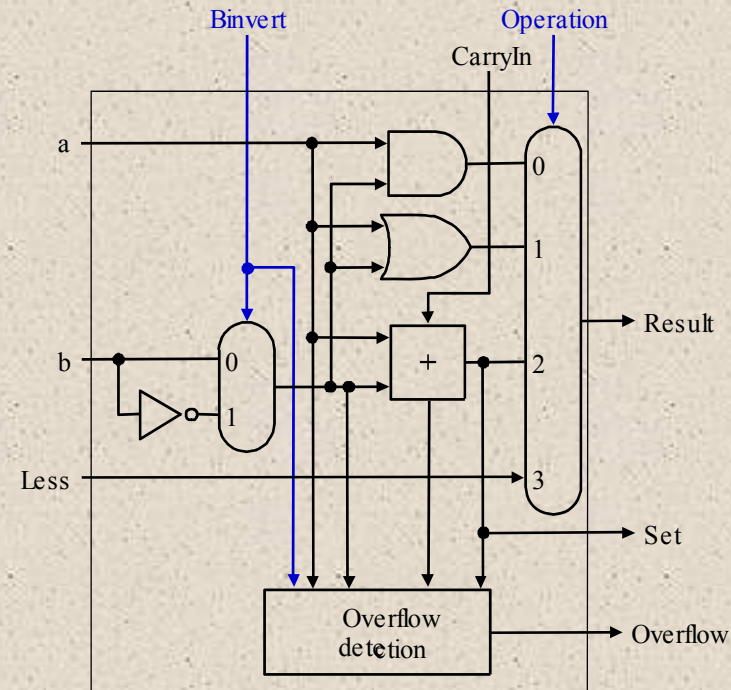
Obvody úplné sčítačky (2/2)



Jednabitová ALU



Středové bity ALU

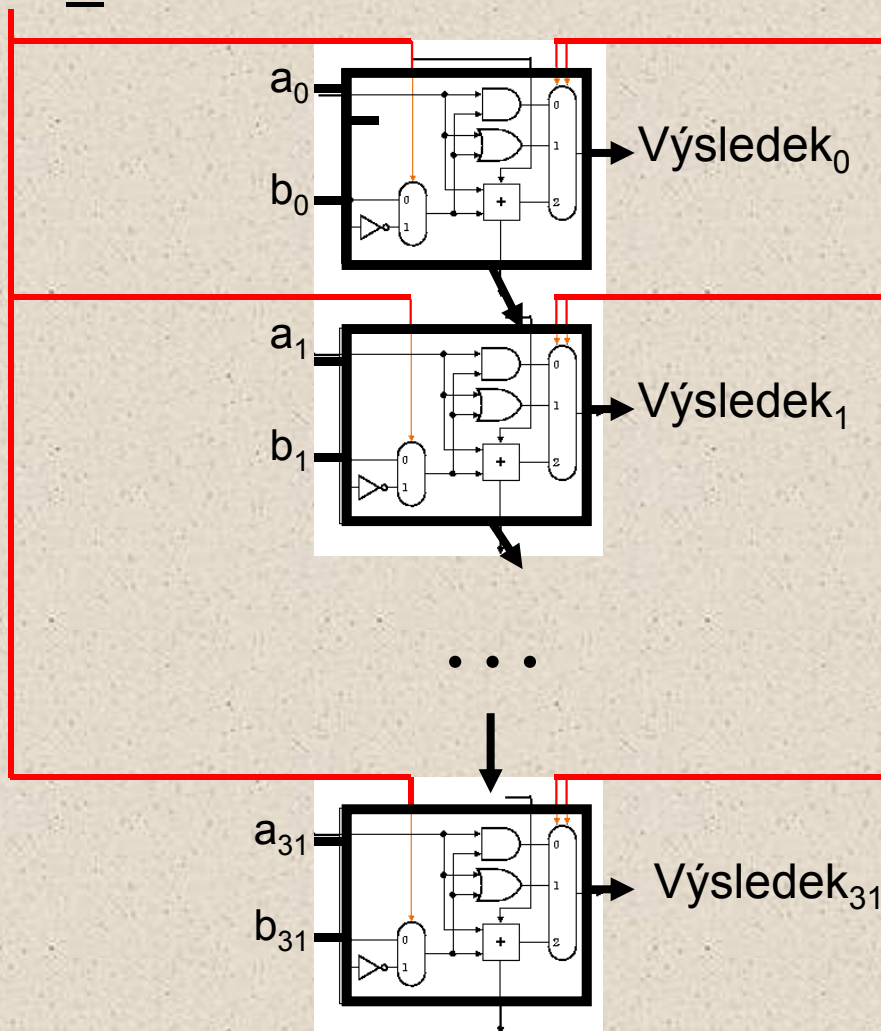


Nejvyšší bit ALU

32-bitová ALU

signál B_invert

Typ operace



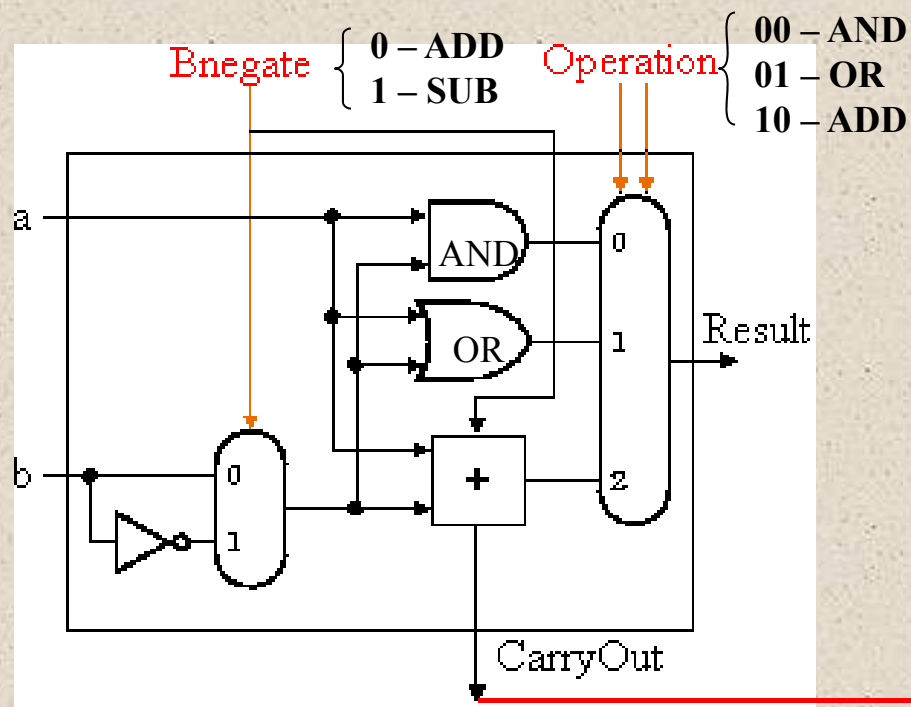
Souhrn

- Stavební bloky: základní hradla (AND, OR a NOT)
- Modulární návrh a implementace
 - Hradla mají větší počet vstupů a jen jeden výstup
 - ALU zpracovává 32-bitová slova (integer)
 - v ALU je implementována řada operací *paralelně*
 - => Nejdříve konstrukce 1-bitové ALU**
 - *Mux* vybere jednu z mnoha různých operací ALU
 - Podle instrukčního souboru se doplní základní operace ALU tak, aby bylo možno implementovat všechny instrukce
 - *Reprezentace dvojkového doplňku dovoluje použít tentýž hardware pro sčítání i odčítání*

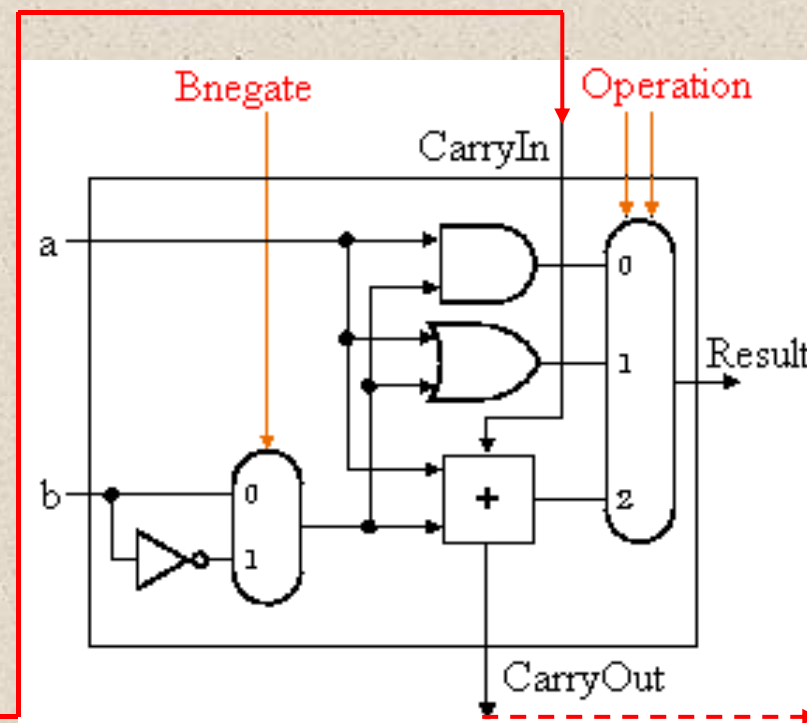
Aplikace na ALU - MIPS

- *Rozšíření MIPS ALU*
- Detekce přetečení
- Instrukce Slt
- Instrukce větvení
- Posuvové instrukce
- Instrukce s přímými operandy
- Výkonnost ALU
 - Výkon vs. cena
 - Sčítačka s urychlením typu „Carry lookahead“
- Alternativy implementace

Opakování: Generická jednobitová ALU



První bit (LSB)



Ostatní bity

Operace: AND, OR, ADD, SUB

Řídící linky: 000 001 010 110

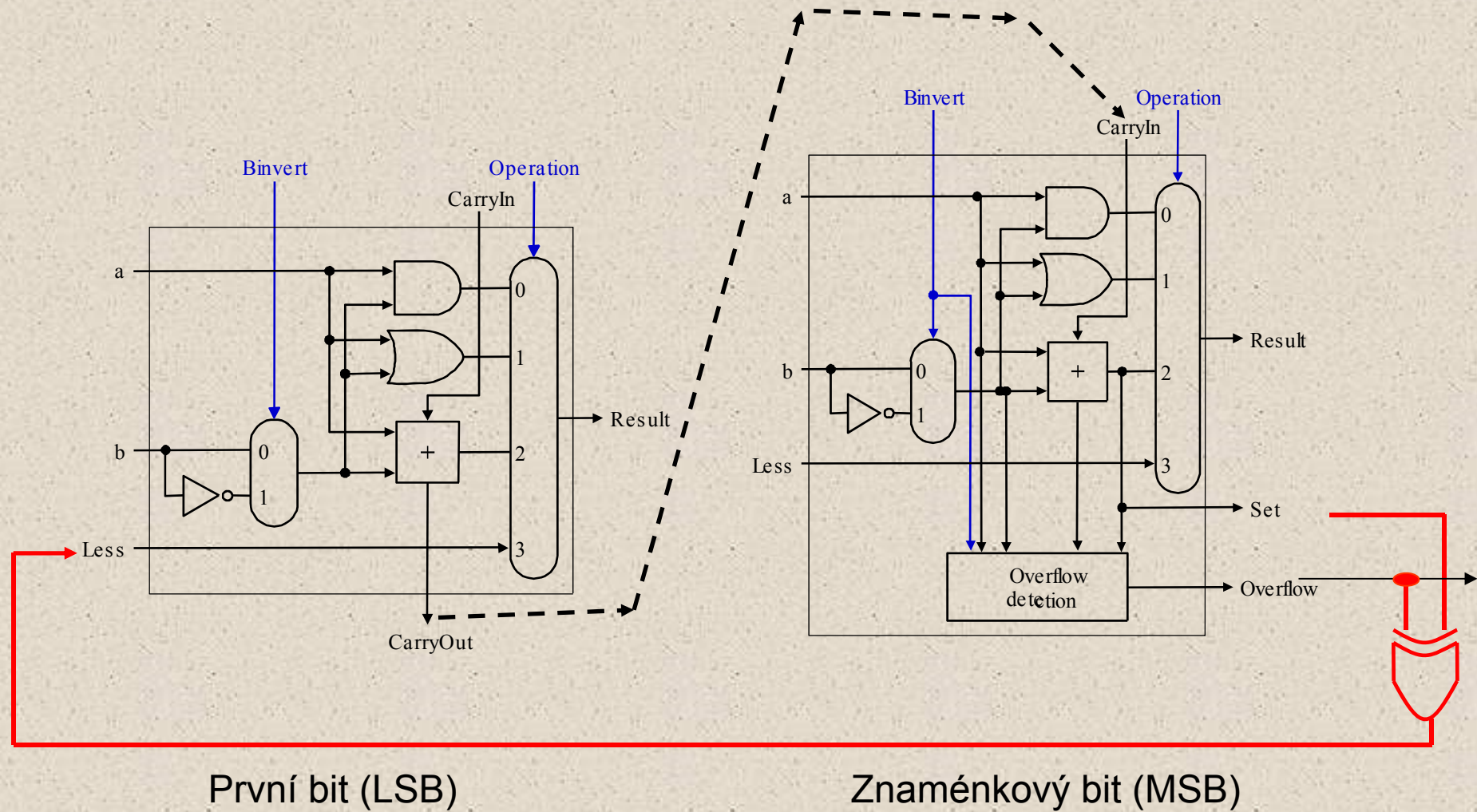
Instrukce Slt

- **Slt rd, rs, rt**

rd: 0000 0000 0000 0000 0000 0000 0000 000**r** → $\begin{cases} 1 & \text{if } (rs < rt) \\ 0 & \text{else} \end{cases}$

- $A < B \Rightarrow A - B < 0$
 1. Vytvoření rozdílu použitím úplné sčítačky
 2. Test bitu s nejvyšší vahou (znaménkový bit)
 3. Znaménkový bit říká, zda $A < B$
- Nový vstupní signál (**Less**) jde přímo na mux
- Nová řídící linka (111) pro slt
- Výsledek pro slt není výstupem z ALU
 - Je třeba další 1-bitová ALU pro bit s nejvyšší vahou
 - Má novou výstupní linku (**Set**) použitou pouze pro slt
 - (S tímto bitem je také spojena logika detekce přetečení)

HW podpora pro Slt



Instrukce větvení programu

- beq \$t5, \$t6, L
 - Použití rozdílu: $(a-b) = 0 \Rightarrow a = b$
 - Pro test výsledku na rovnost 0 - přidat HW
 - operace OR s 32 bity výsledku a následná inverze výstupu OR

$$\text{Zero} = \overline{(\text{Result}_1 + \text{Result}_2 + \dots + \text{Result}_{31})}$$

- Uvažujme operace $A + B$ a $A - B$
 - Přetečení nastane, je-li
 - $A = 0$?
 - $B = 0$?

HW podpora větvení

- Řídící linky:

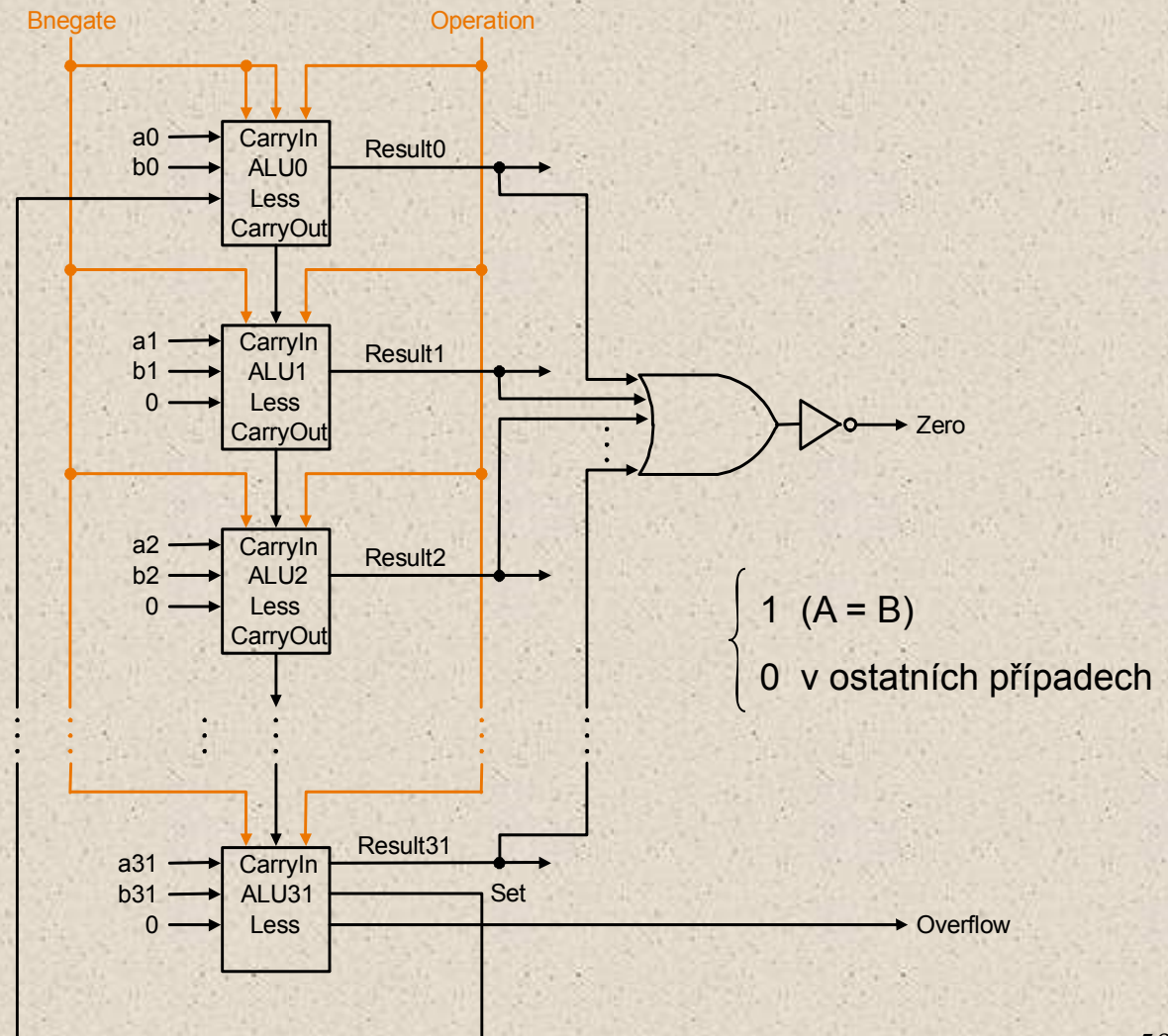
000 = and

001 = or

010 = add

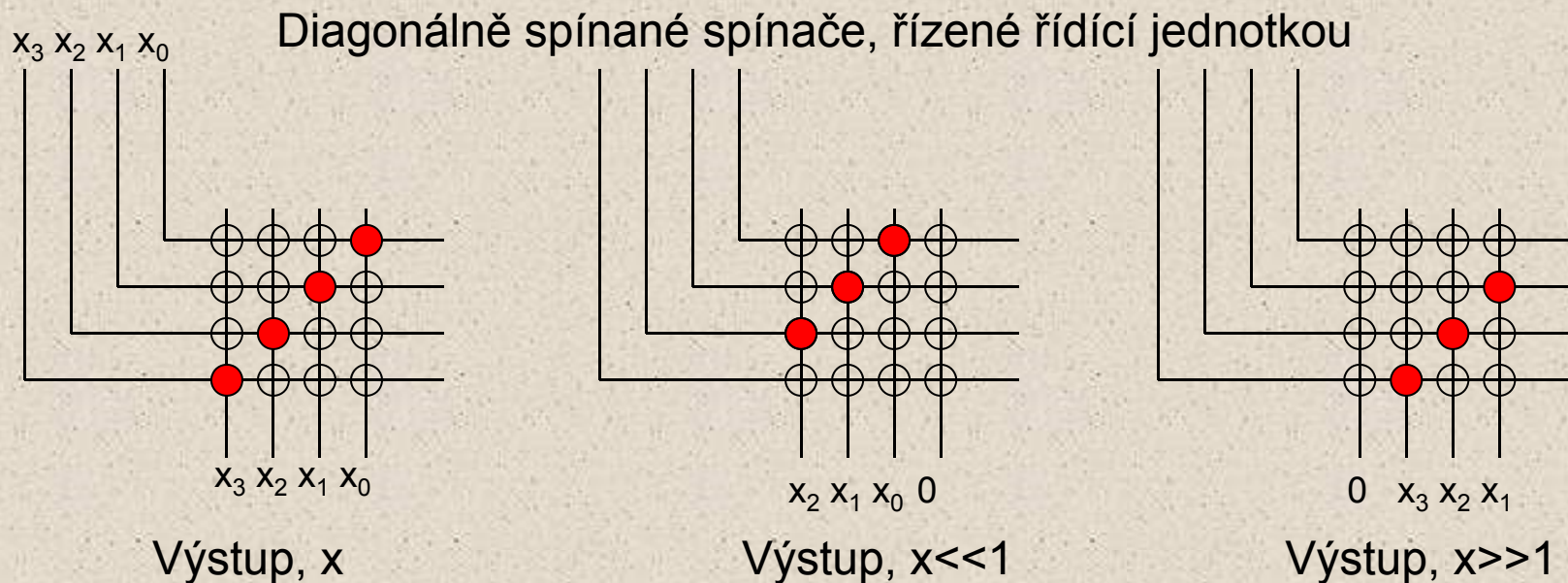
110 = subtract

111 = slt

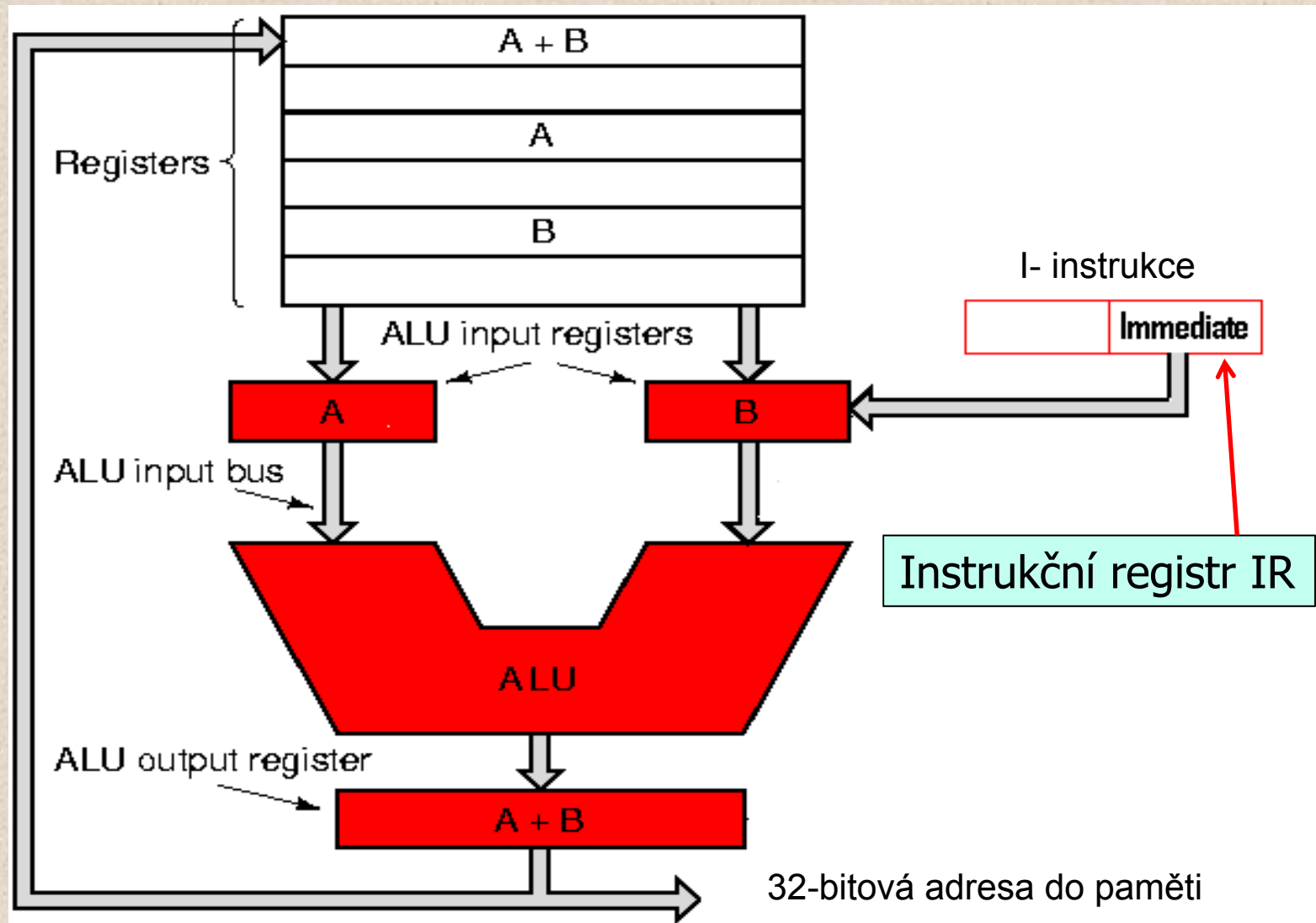


Instrukce posuvu

- SLL, SRL a SRA
- Potřebujeme datovou linku pro posuvy (\overleftarrow{L} a \overrightarrow{R})
- Nicméně posuvové jednotky se snáze implementují na úrovni tranzistorů (mimo ALU)
- Posuvové jednotky typu „Barrel shifter“

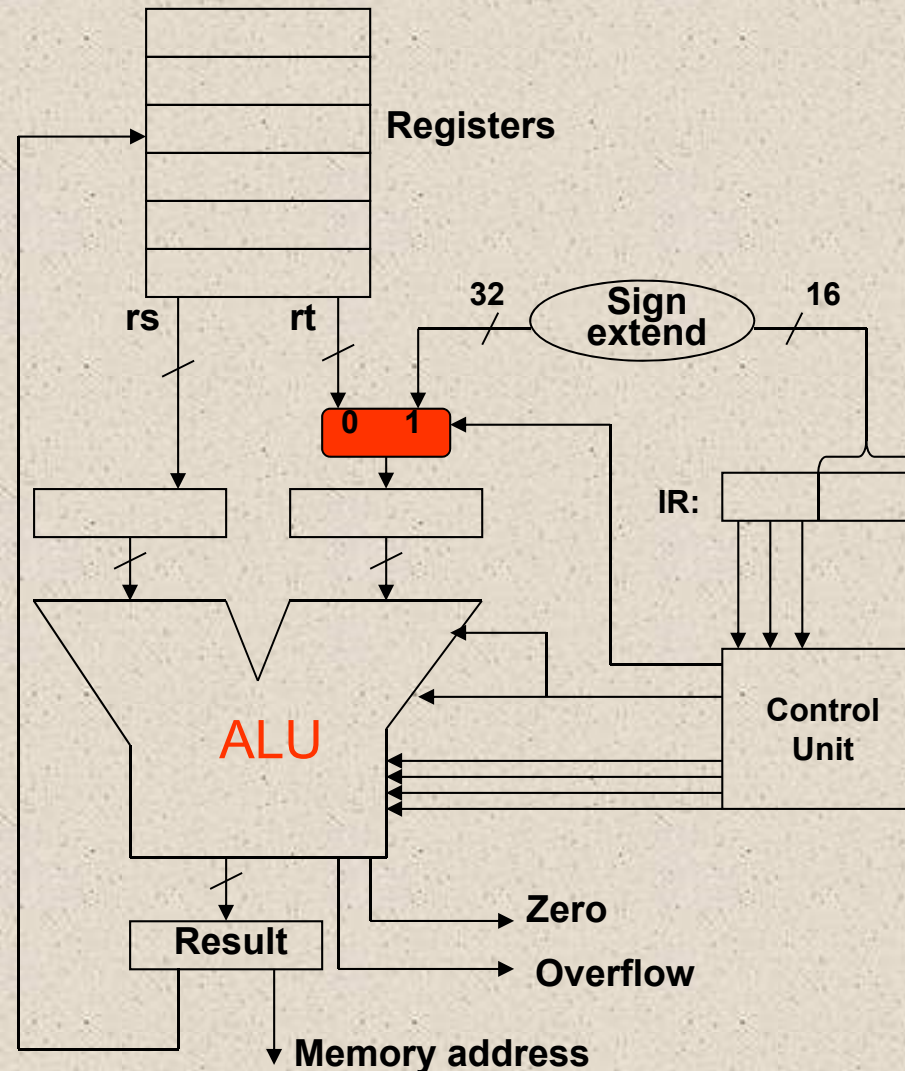


Přehled

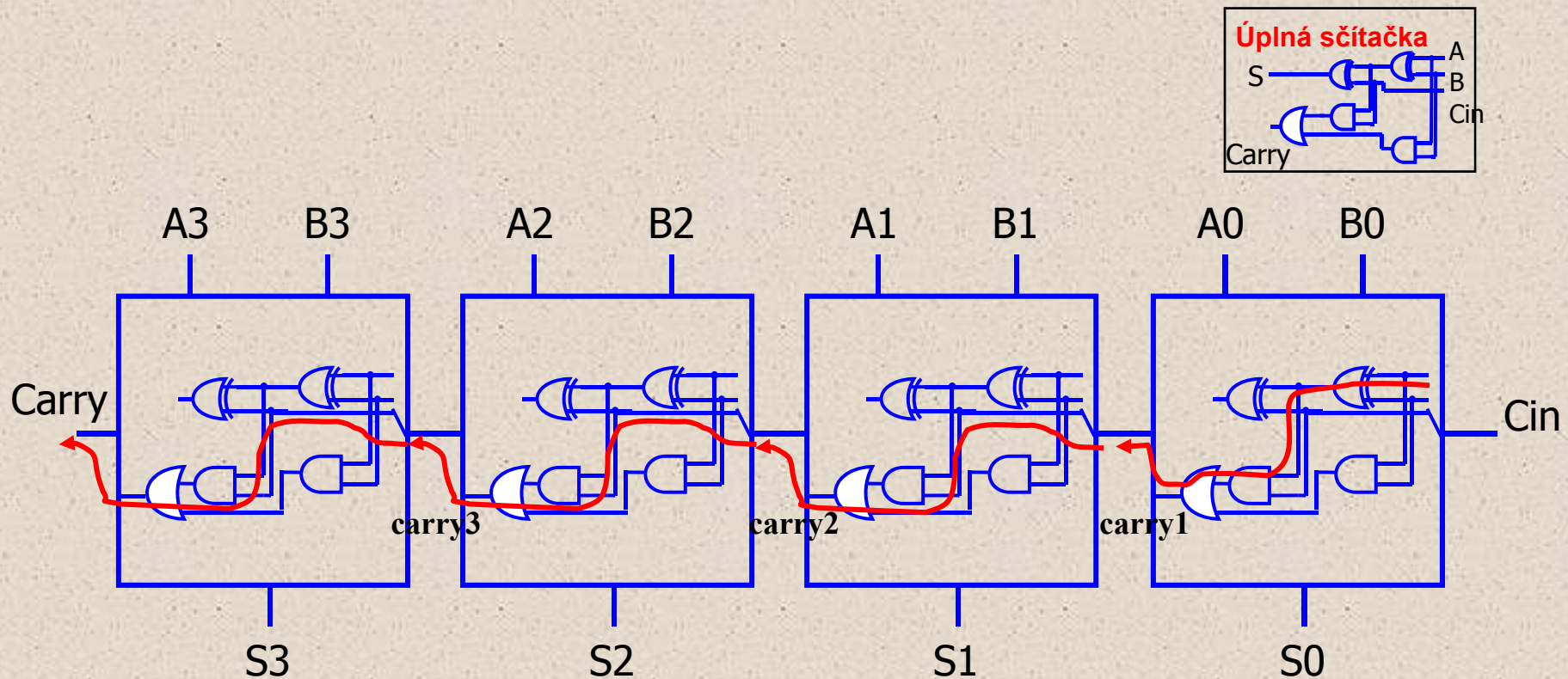


Instrukce s přímým operandem

- Prvý vstup do ALU tvoří první registr (rs)
- Druhý vstup
 - Data z registru (rt)
 - Nula nebo **immediate** s rozšířením znaménka
- Přidáme multiplexer na druhý vstup ALU



4-bitová „Ripple Carry“ sčítačka



Kritická cesta = $D_{\text{XOR}} + 4 \cdot (D_{\text{AND}} + D_{\text{OR}})$ pro 4-bitovou ripple carry sčítačku (9 úrovní hradel)

Pro N-bitovou ripple carry sčítačku:

Zpoždění kritické cesty $\sim 2 \cdot (N-1) + 3 = (2N+1) \cdot \text{zpoždění hradla}$

Výkonnost ALU

- Je 32-bitová ALU stejně rychlá jako 1-bitová ALU?
 - Šíření signálu při vyhodnocování přenosů?
- Hardware provádí vyhodnocení paralelně (???)
- Rychlost vs. cena
 - Méně sekvenčních hradel vs. celkový počet hradel
- Dva extrémy jak provést součet
 - „Ripple carry“ a součet součinů
- Jak se zbavit problému s přenosem
 - Dvě úrovně logiky

$$c_1 = b_0 c_0 + a_0 c_0 + a_0 b_0$$

$$c_2 = b_1 c_1 + a_1 c_1 + a_1 b_1$$

$$c_3 = b_2 c_2 + a_2 c_2 + a_2 b_2$$

$$c_4 = b_3 c_3 + a_3 c_3 + a_3 b_3$$

$$c_2 =$$

$$c_3 =$$

$$c_4 =$$

Sčítačka Carry Lookahead

$$C_{i+1} = g_i + p_i C_i$$

$$C_{i+1} = A_i B_i + C_i (A_i + B_i)$$

$$g_i = A_i B_i \quad (\text{generace})$$

$$p_i = A_i + B_i \quad (\text{propuštění})$$

$$C_1 = g_0 + p_0 C_0$$

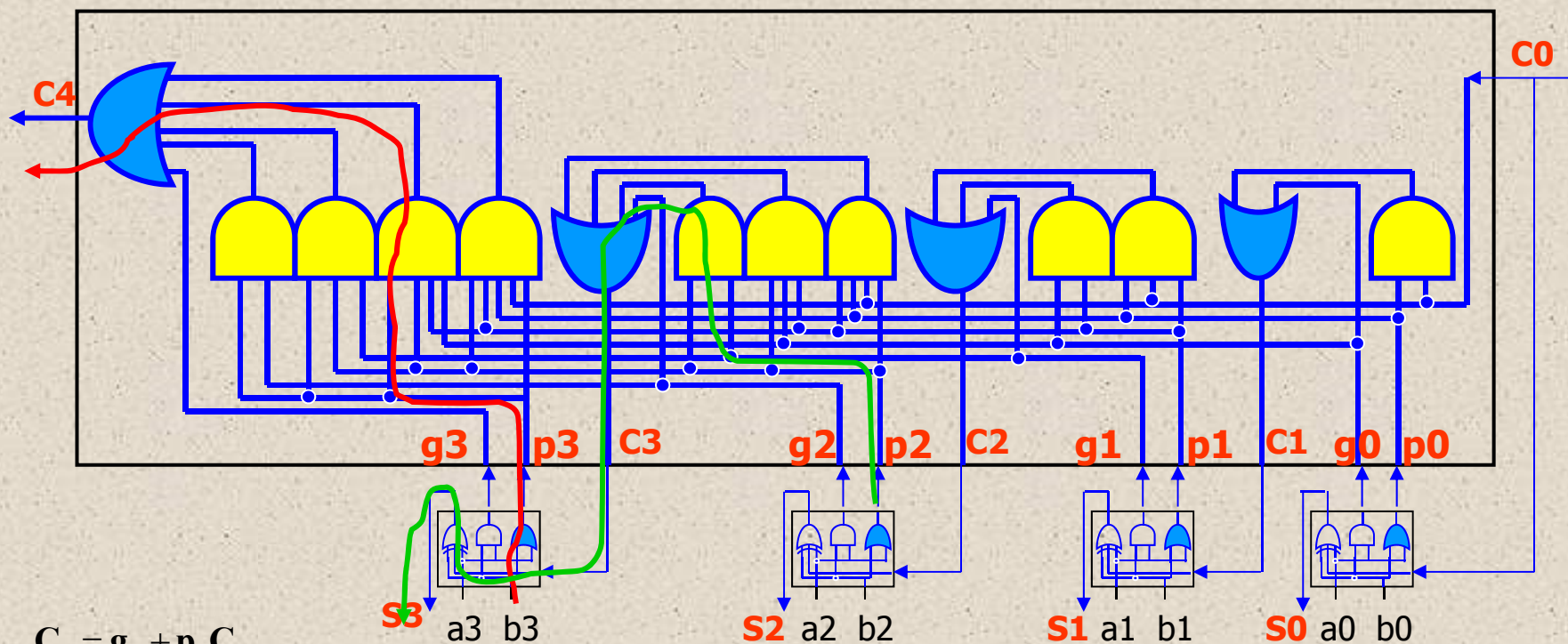
$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2 C_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

$$C_4 = g_3 + p_3 C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

Poznámka: Přenosy závisí pouze
na vstupech A, B a C !!!

4-bitová sčítačka Carry Lookahead



$$C_1 = g_0 + p_0 C_0$$

$$C_2 = g_1 + p_1 C_1 = g_1 + p_1 g_0 + p_1 p_0 C_0$$

$$C_3 = g_2 + p_2 C_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$$

$$C_4 = g_3 + p_3 C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$$

Zpoždění jen 3 hradel pro každý C_i
 $= D_{AND} + 2 \cdot D_{OR}$

4 zpoždění pro každý S_i
 $= D_{AND} + 2 \cdot D_{OR} + D_{XOR}$

Sčítačka typu Carry-Lookahead (1/2)

- Řešení - kompromis mezi dvěma extrémy
- Motivace:
 - Co můžeme dělat, neznáme-li hodnotu carry-in?
 - Kdy budeme vždy generovat přenos? $g_i = a_i b_i$
 - Kdy jej budeme předávat dál? $p_i = a_i + b_i$
- Zbavili jsme se šíření vlny v cestě přenosu?

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1$$

$$c_3 = g_2 + p_2 c_2$$

$$c_4 = g_3 + p_3 c_3$$

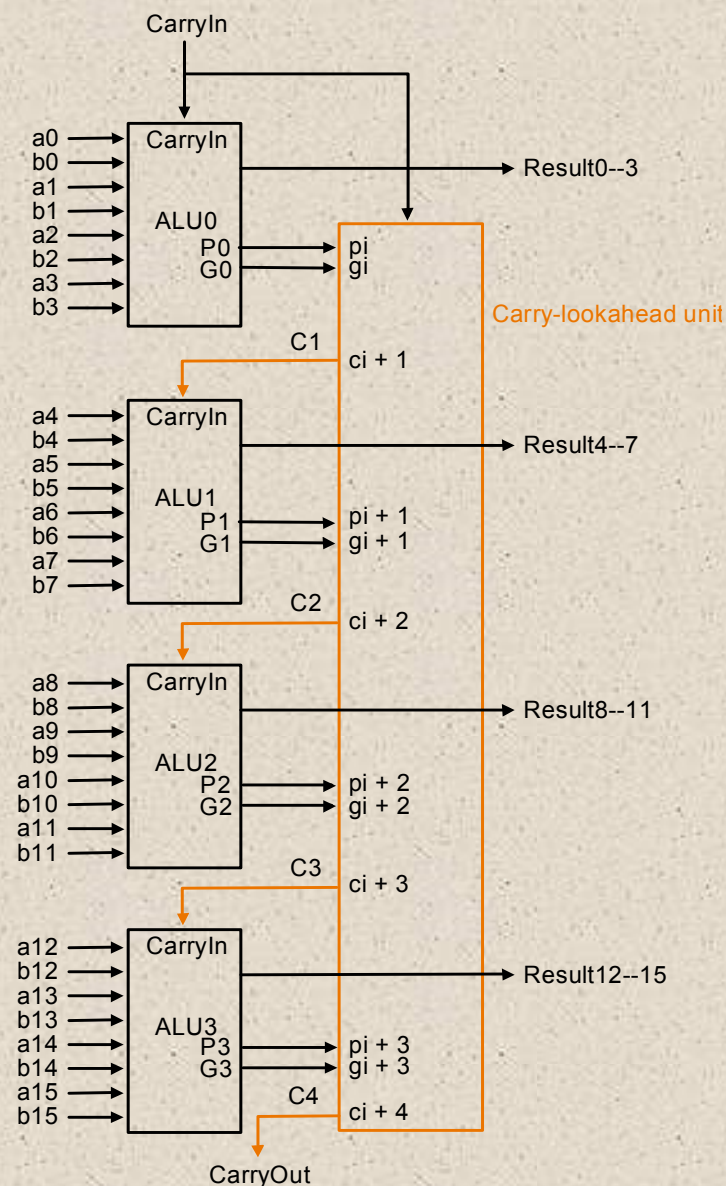
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 g_1 + \dots$$

$$c_4 =$$

Hierarchická stavba sčítaček typu Carry-Lookahead

- Uvedeným způsobem nelze postavit 16-ti bitovou sčítačku (příliš velká a složitá)
- Lze postavit „ripple carry“ sčítačku s použitím 4-bitových CLA sčítaček
- Lépe: použít princip CLA opakovaně!



Funkce G a P druhé úrovně

$$P_0 = p_3 + p_2 + p_1 + p_0$$

$$P_1 = p_7 + p_6 + p_5 + p_4$$

$$P_2 = p_{11} + p_{10} + p_9 + p_8$$

$$P_3 = p_{15} + p_{14} + p_{13} + p_{12}$$

$$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$

$$G_1 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4$$

$$G_2 = g_{11} + p_{11}g_{10} + p_{11}p_{10}g_9 + p_{11}p_{10}p_9g_8$$

$$G_3 = g_{15} + p_{15}g_{14} + p_{15}p_{14}g_{13} + p_{15}p_{14}p_{13}g_{12}$$

Ripple Carry vs. Carry Lookahead

- Budeme uvažovat stejné zpoždění pro průchod signálu hradlem (AND nebo OR)
- Celková doba = dána počtem hradel v nejdelší cestě
- Předpokládejme 16-bitovou sčítačku
- Signály **CarryOut** c_{16} a c_4 určují nejdelší cestu
 - **Ripple carry**: $2 * 16 = 32$
 - **Carry lookahead**: $2 + 2 + 1 = 5$
 - 2 úrovně logiky pro vytvoření P_i a G_i
 - P_i je určen v jedné úrovni (AND) z jednotlivých p_i
 - G_i se vytváří ve dvoustupňové logice ze signálů p_i a g_i
 - p_i a g_i lze vytvořit v jedné úrovni ze signálů a_i a b_i
- „Carry lookahead adder“ je zde **šestkrát** rychlejší

Alternativy implementace

- Logická rovnice pro součet může být zapsána mnohem jednodušeji použitím hradel XOR
$$\text{Sum} = a \text{ XOR } b \text{ XOR } \text{CarryIn}$$
- Pro některé technologie vychází XOR efektivněji než dvě úrovně AND a OR
- Procesory se nyní implementují pomocí CMOS tranzistorů (spínače)
- CMOS ALU a posuvové jednotky typu „barrel shifter“ mají méně multiplexorů, než obsahoval náš návrh.
- Principy návrhu jsou stejné

Závěr

- ISA podporuje rozvoj architektury
- Hardware/Software, důraz na RISC/CISC
- Technologie je hnacím motorem rozvoje
- ALU = jádro procesoru
- ALU problém = přetečení
- Ošetření výjimek (přerušení) 😊

Závěr

- Můžeme postavit ALU tak, aby vyhověla MIPS ISA
 - **Klíčová myšlenka:** Použít **multiplexer** pro **výběr výstupu ALU**
 - Pro **odčítání** se používá **sčítání dvojkového doplňku**
 - Pro sestavení 32-bitové ALU **opakovaně použít** 1-bitovou ALU
- Důležité poznámky k hardware
 - Všechna **hradla** v ALU **pracují paralelně**
 - **Rychlost hradel** závisí na **počtu vstupů**
 - **Rychlost obvodu** závisí na počtu **sériově řazených hradel**
(v **kritické cestě**, nebo-li na **počtu úrovní logiky**)
- Primární cíl: (koncepční)
 - **Promyšlené změny organizace** mohou zlepšit výkon
(podobně jako použití lepšího algoritmu při programování)