

# **Principy počítačů a operačních systémů**

**Architektura a implementace  
zjednodušeného procesoru MIPS**

**Zimní semestr 2010/2011**

# Architektura procesoru MIPS (1)

## Registry

- 32 obecných 32-bitových registrů (*general-purpose*)
  - ♦ s0-s7, t0-t9, zero, a0-a3, v0-v1, gp, fp, sp, ra, at (r0-r31)
- registr PC s adresou následující instrukce
- speciální registry pro řízení
  - ♦ adresa instrukce, který vyvolala výjimku, apod.

## Paměť

- přístup pouze na zarovnané adresy (dělitelné 4)
  - ♦ granularita délky slova (32 bitů), tj. 4B
- nepřímá adresace s posunutím
  - ♦ *register indirect with displacement*
  - ♦  $R2 = \text{Mem}[R1 + \text{immediate}]$ ,  $\text{Mem}[R1 + \text{immediate}] = R2$



# Architektura procesoru MIPS (2)

---

## Operace

- operace registr/registr, registr/immediate
  - ♦ aritmetické a logické operace, přesun dat mezi registry
- přesuny dat registr/paměť
  - ♦ load/store architektura
- podmíněné skoky
  - ♦ při rovnosti/nerovnosti obsahu dvou registrů
- nepodmíněné skoky
  - ♦ včetně nepřímých skoků a skoků do podprogramu
- speciální instrukce
  - ♦ práce se speciálními registry



# Architektura procesoru MIPS (3)

---

## Zpracování instrukcí

- čtení instrukce z paměti na adrese v PC
- dekódování instrukce a čtení operandů z registrů
- vykonání operace odpovídající instrukčnímu kódu
  - ♦ operace s obsahem registrů, výpočet adresy a čtení z (zápis do) paměti, porovnání operandů pro podmíněný skok
- uložení výsledku do registru
  - ♦ výsledek operace s registry, data přečtená z paměti
- posun PC na následující instrukci
  - ♦ následující instrukce následuje bezprostředně za právě čtenou instrukcí, pokud není řečeno jinak (podmíněný/nepodmíněný skok, výjimka)



# Jednocyklová datová cesta

Uspořádání logických celků  
implementující operace procesoru

# Čtení instrukce (fetch)

## Registr PC

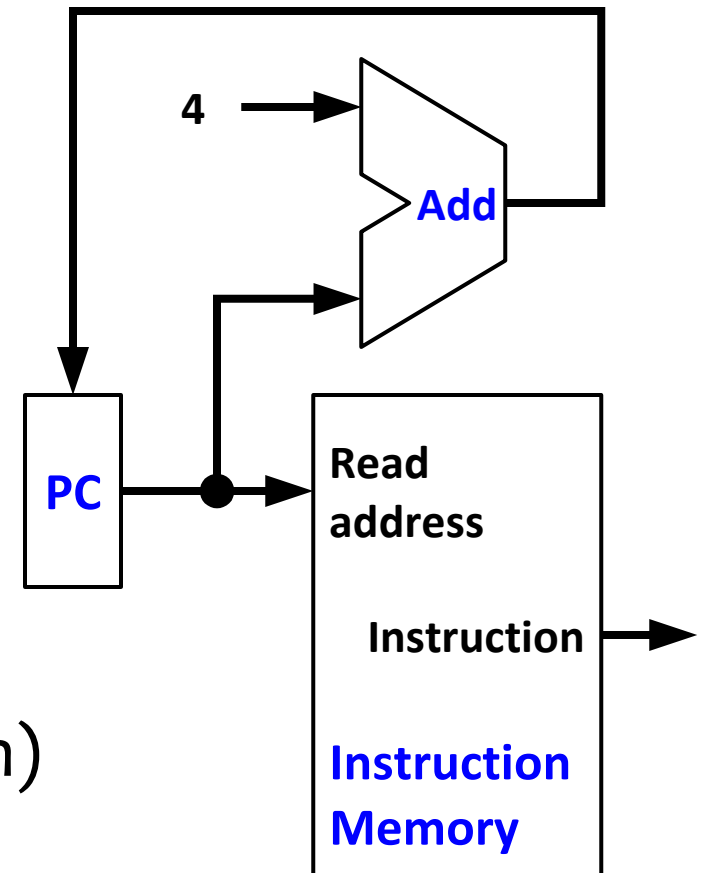
- obsahuje adresu instrukce v paměti
- pro programátora není přímo přístupný

## Paměť instrukcí

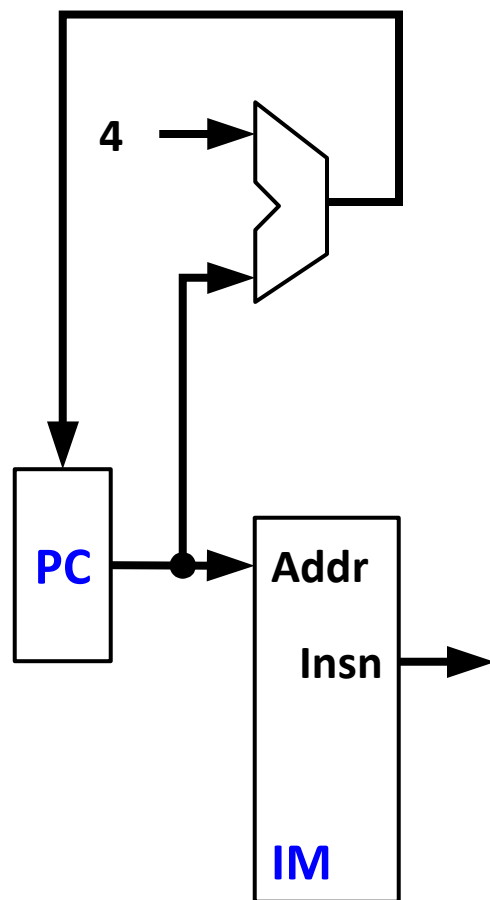
- oddělená paměť = Harvardská architektura (prozatím)

## Sčítačka

- inkrementuje obsah PC o 4  $\Rightarrow$  implicitní posun na následující instrukci (von Neumannovská koncepce)

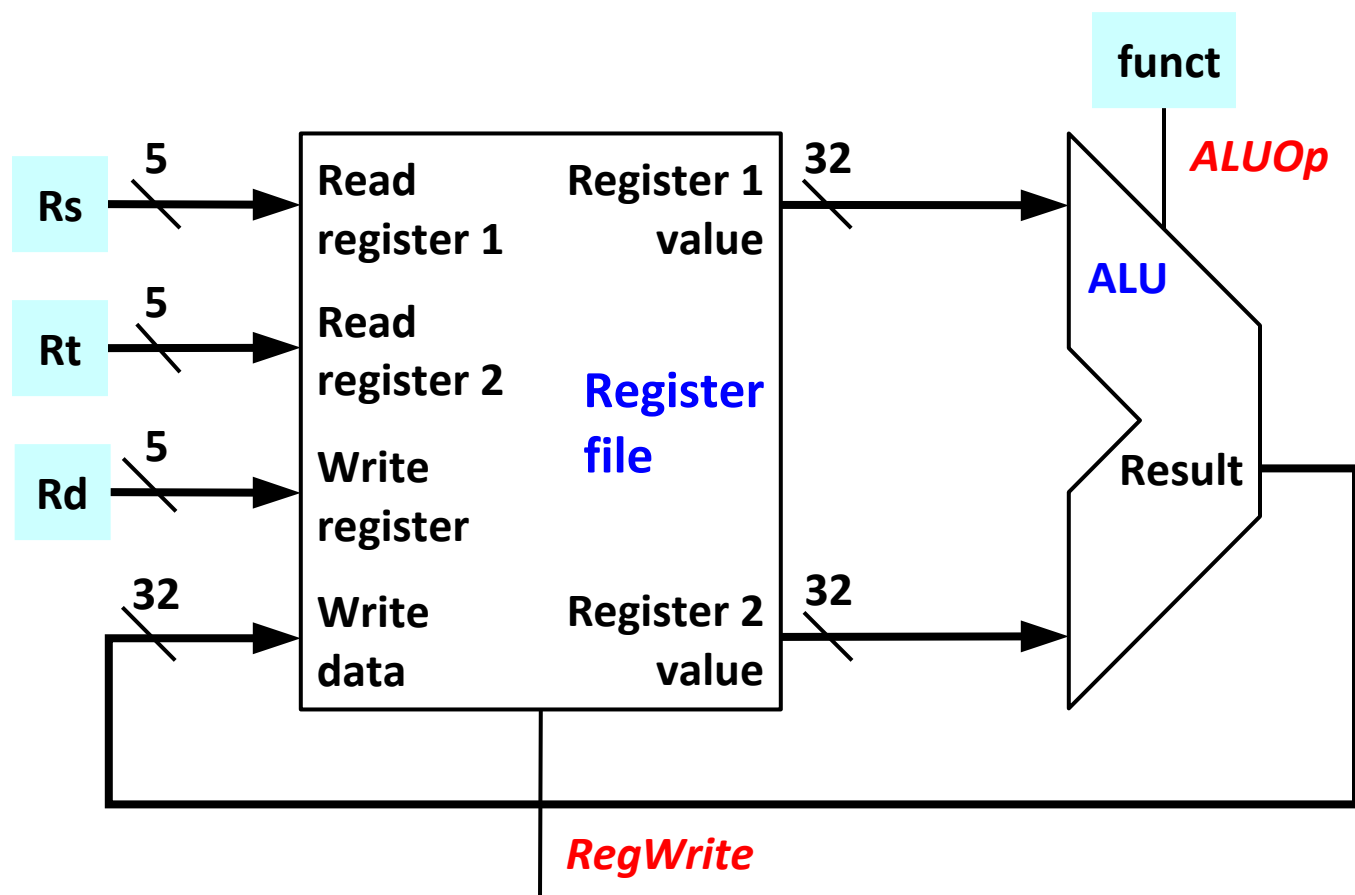


# Podpora čtení instrukce



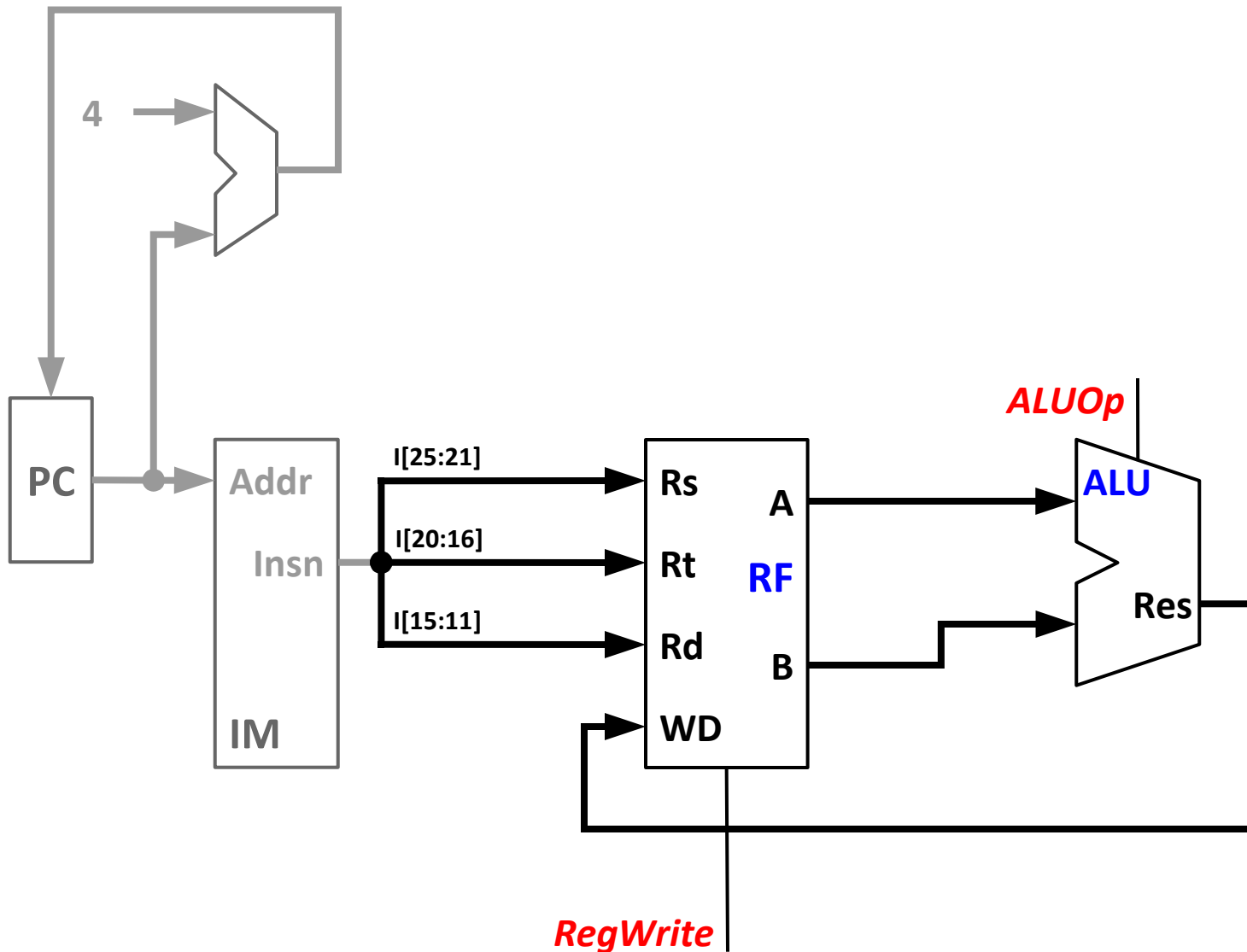
# Registrové operace (add, sub, ...)

Op (6)	Rs (5)	Rt (5)	Rd (5)	shamt (6)	Funct (6)
--------	--------	--------	--------	-----------	-----------

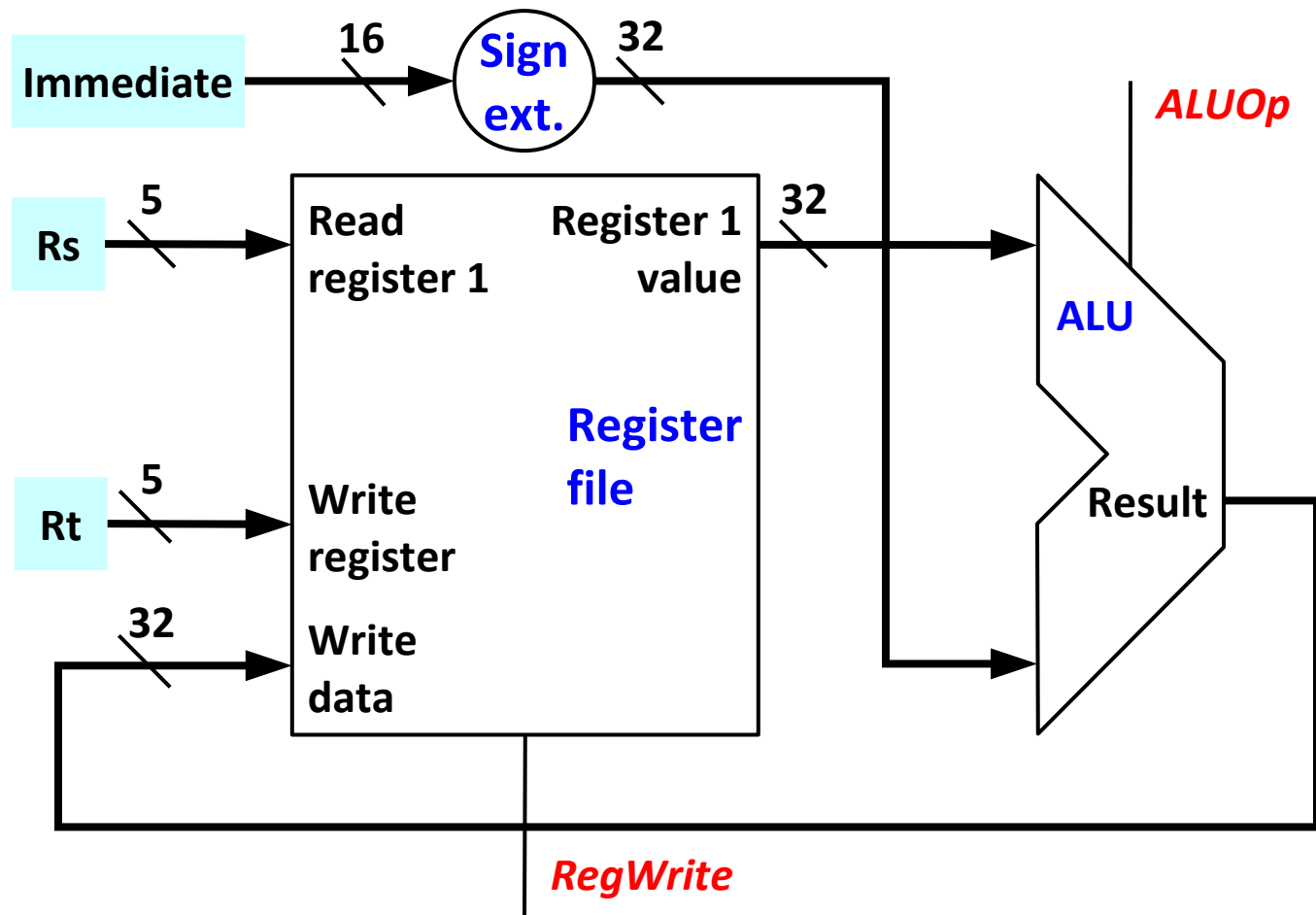




# Podpora registrových operací



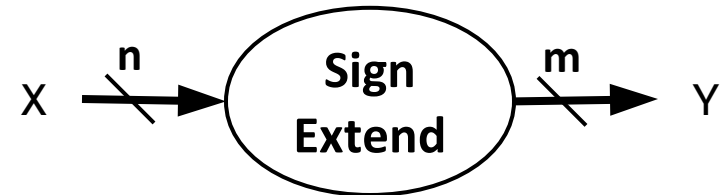
# Operace s přímými operandy (addi, andi, ...)



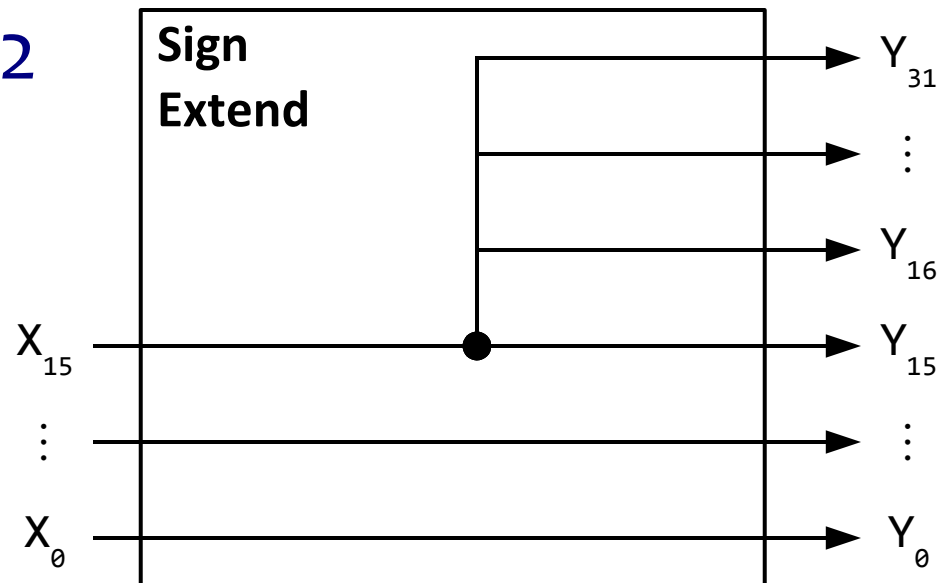
# Znaménkové rozšíření (sign extension)

## Logický prvek

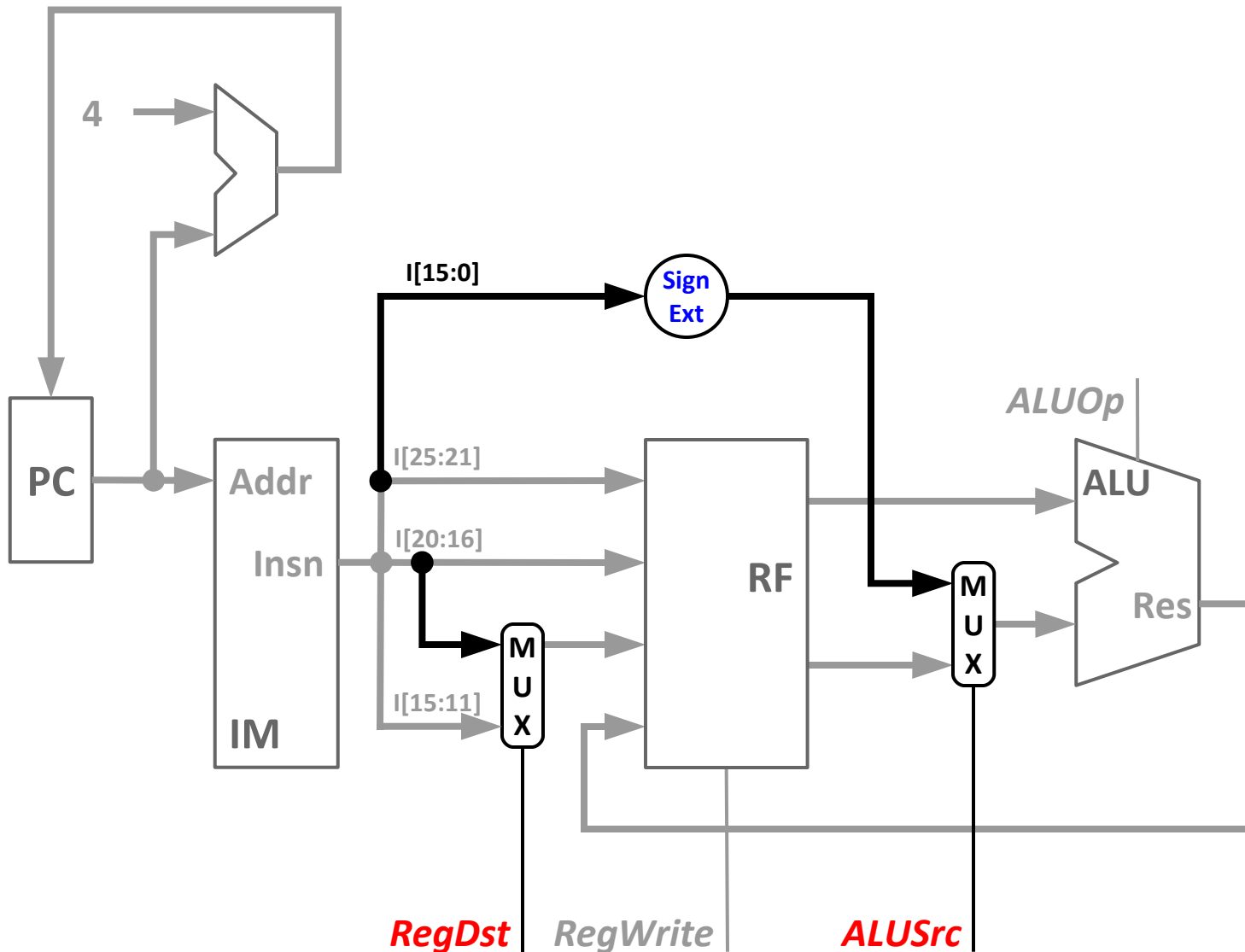
- vstup:  $n$ -bitové číslo  $X$ 
  - ♦ předpokládá se reprezentace ve dvojkové doplňku
- výstup:  $m$ -bitové číslo  $Y$ 
  - ♦ znaménkové rozšíření čísla  $X$



## Příklad pro $n=16$ , $m=32$



# Podpora přímých operandů



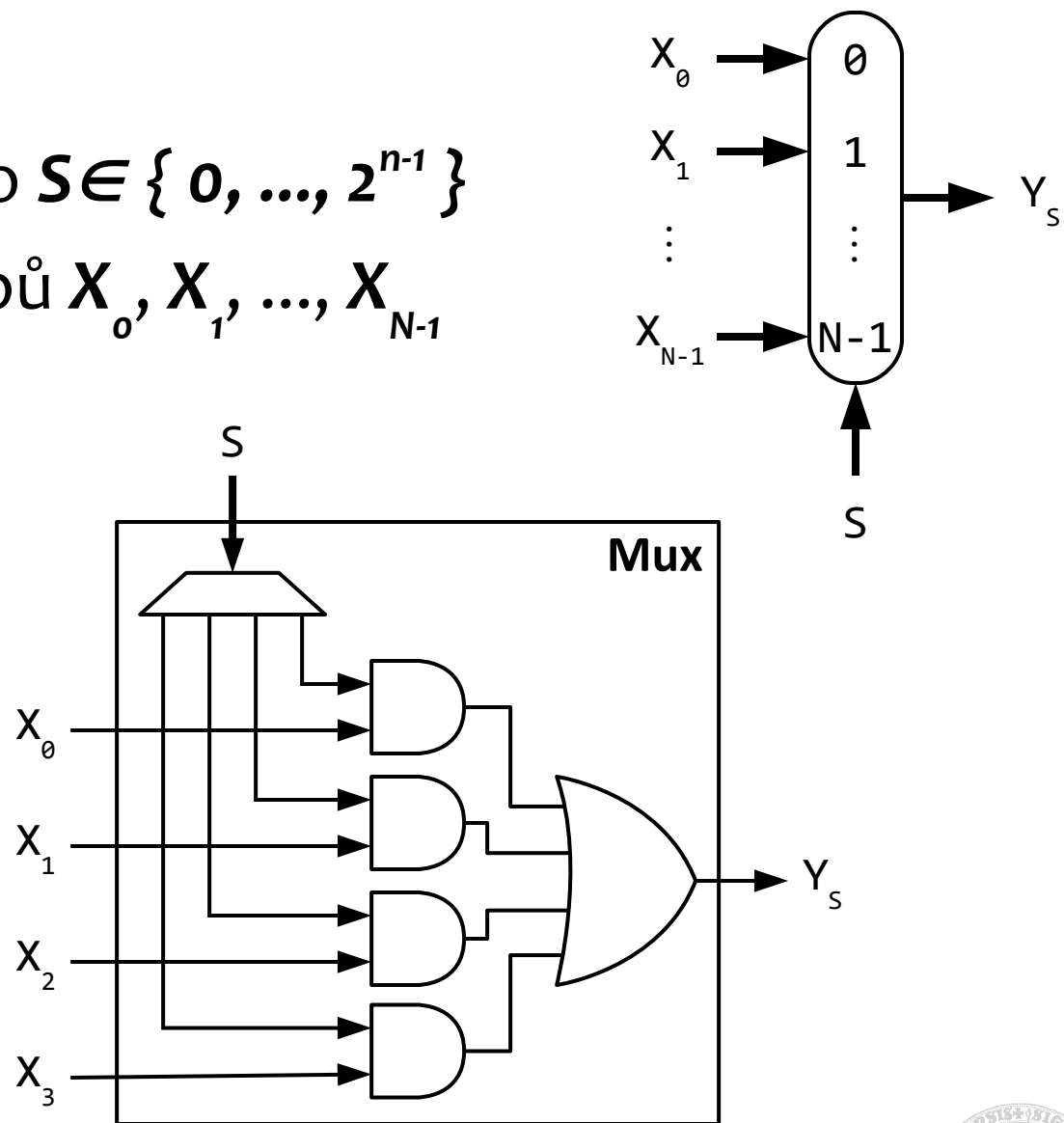
# Přepínač vstupů (multiplexer, mux)

## Logický prvek

- selektor:  $n$ -bitové číslo  $S \in \{0, \dots, 2^n - 1\}$
- $N=2^n$   $m$ -bitových vstupů  $X_0, X_1, \dots, X_{N-1}$
- $m$ -bitový výstup  $Y = X_S$

## Příklad pro $N=4, m=1$

- pro výběr vstupu se používá dekodér do kódu “1 z  $N$ ”



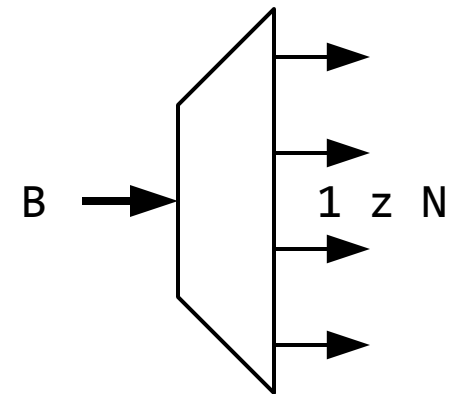
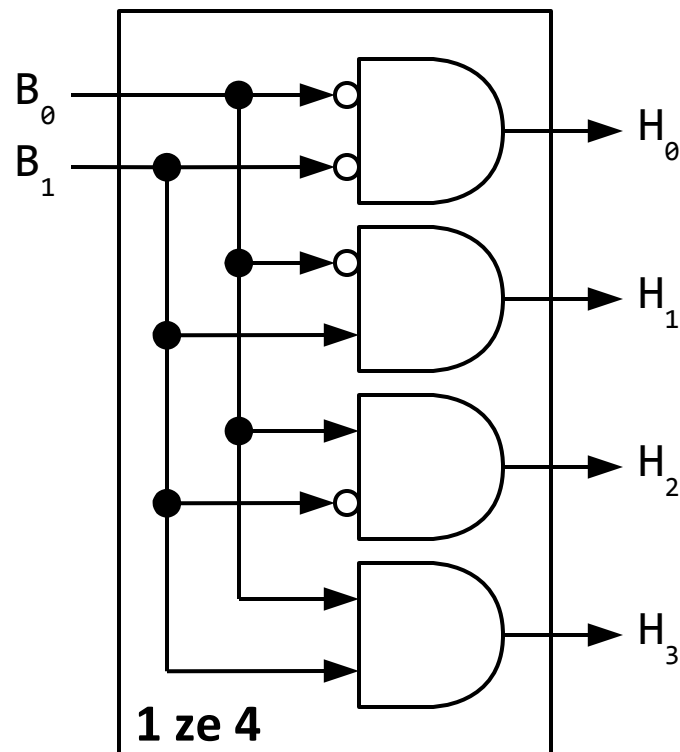
# Binární dekodér do “1 z N” (binary to 1-hot)

## Logický prvek

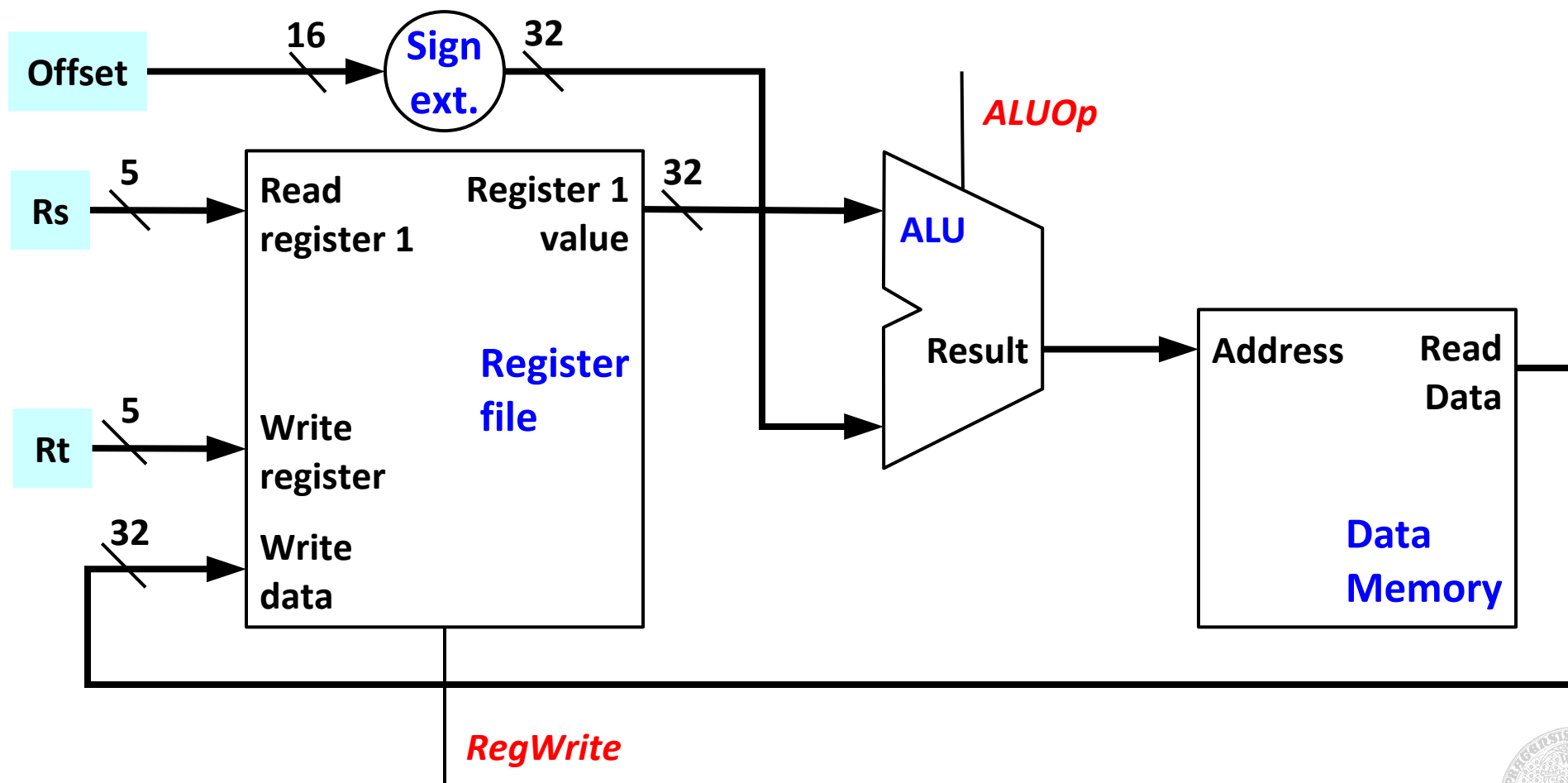
- $n$  vstupů:  $n$ -bitové dvojkové číslo  $B \in \{0, \dots, 2^{n-1}\}$
- $N=2^n$  výstupů: reprezentace  $B$  v kódu 1 z N
  - ♦  $B$ -tý bit 1, ostatní 0

## Příklad pro $N=4$

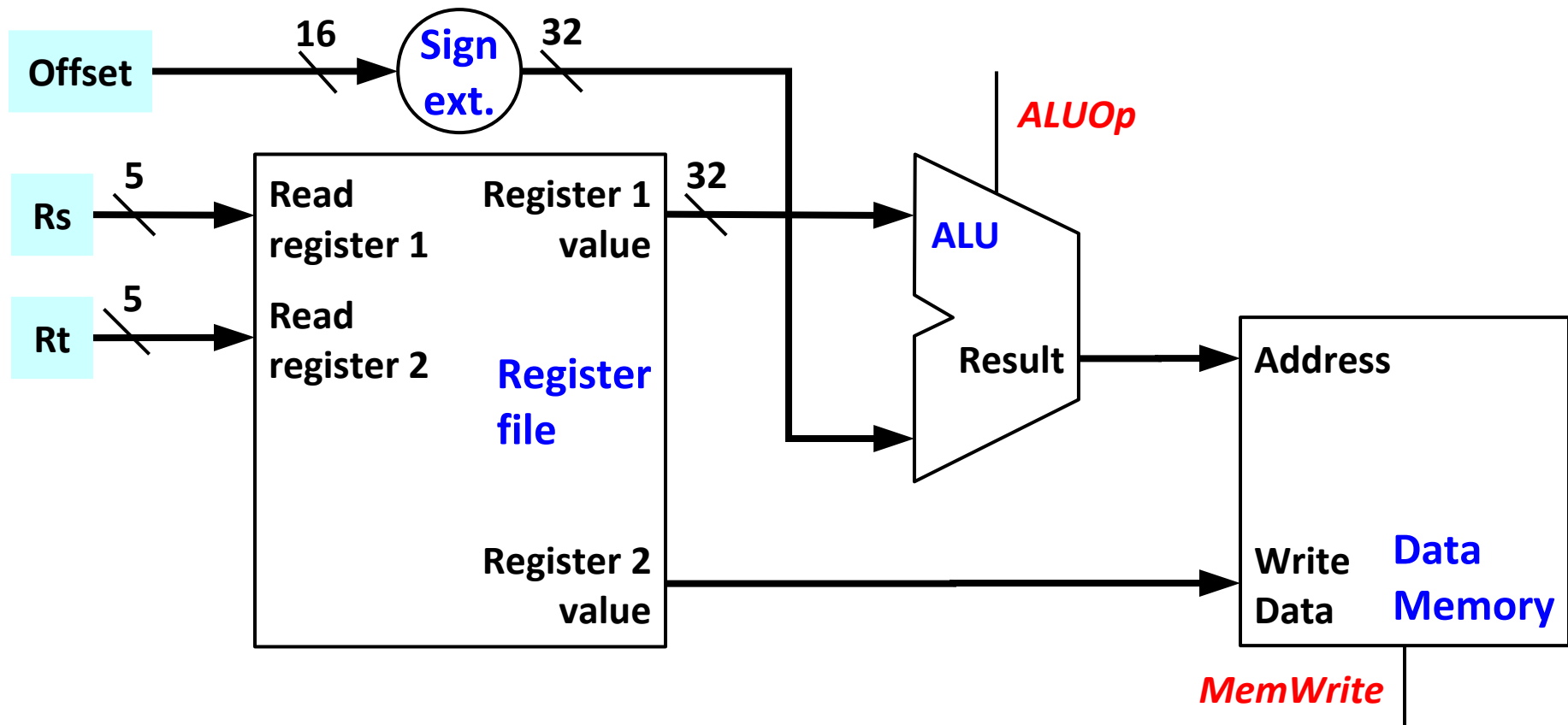
Vstupy		Výstupy			
$B_1$	$B_0$	$H_3$	$H_2$	$H_1$	$H_0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



# Čtení z paměti (load)

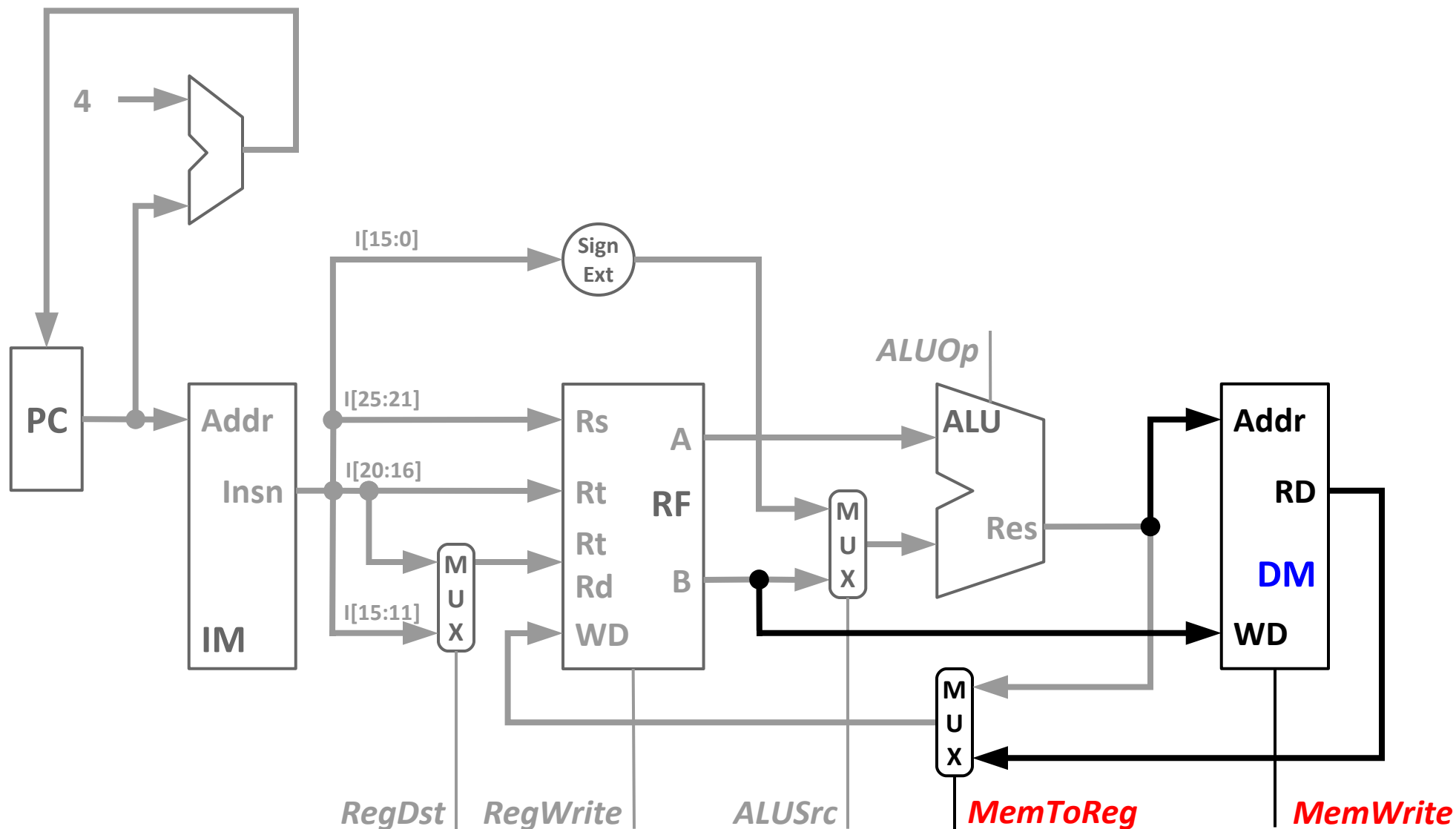


# Zápis do paměti (store)

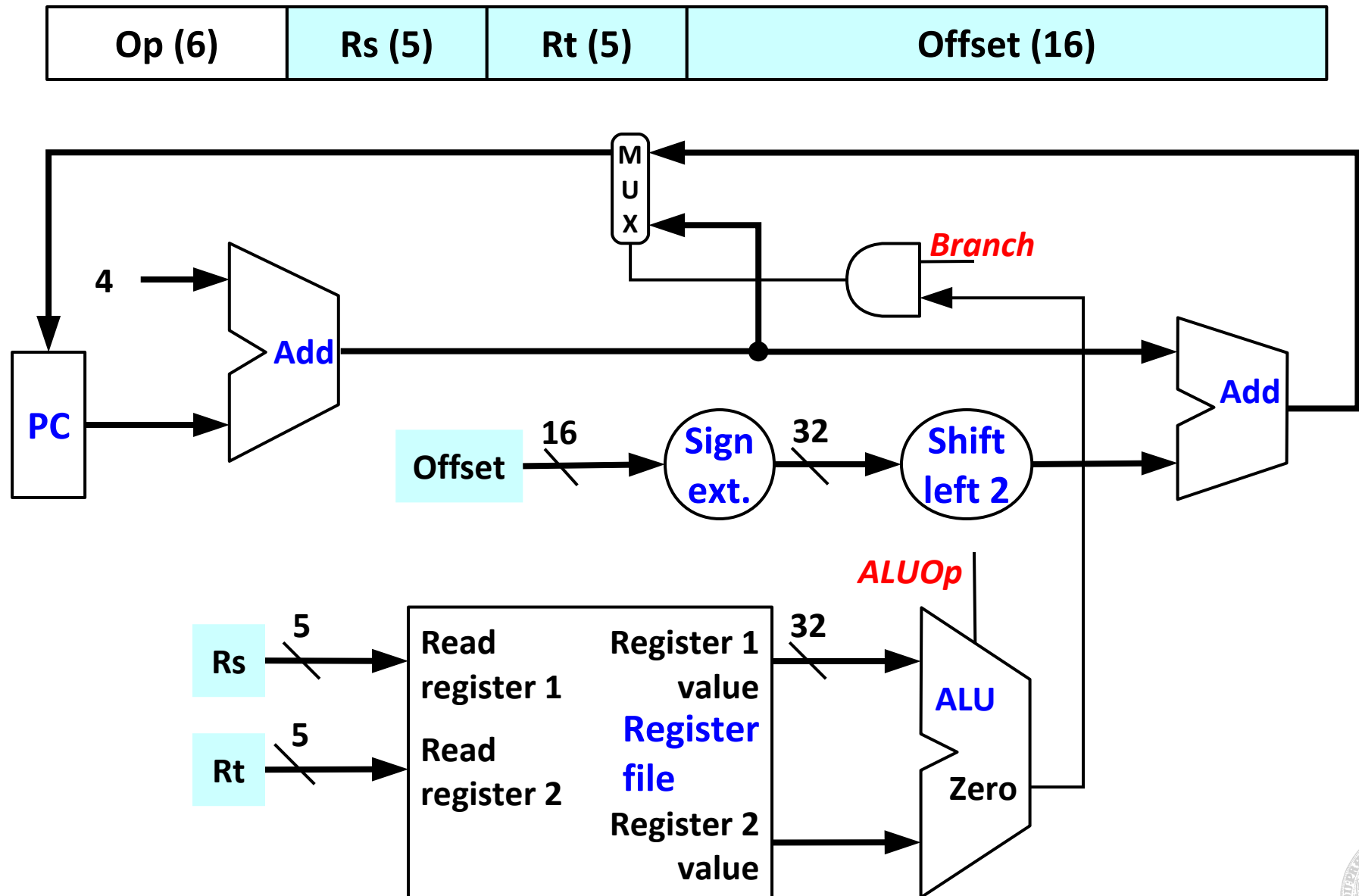




# Podpora přístupu do paměti



# Podmíněný skok s relativní adresou (branch)



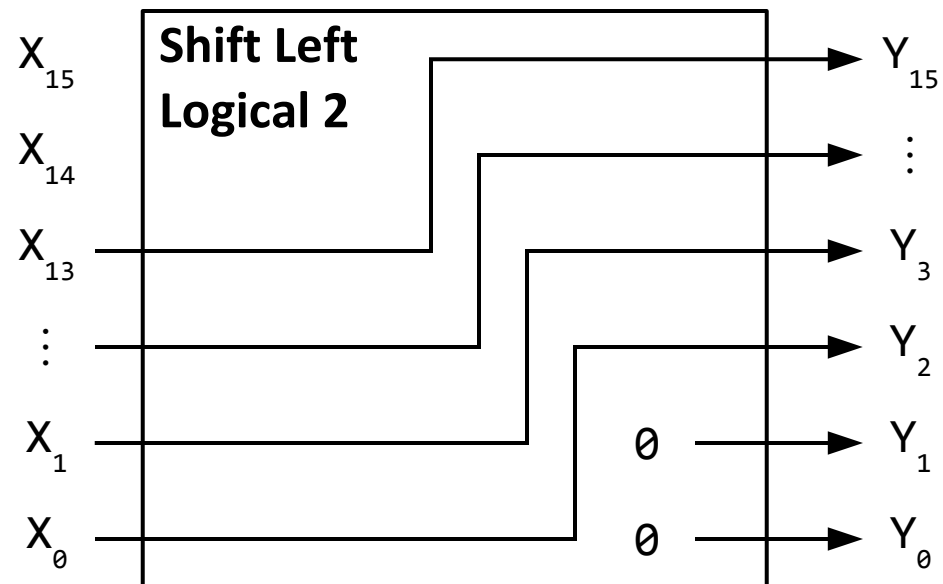
# Logický posun vlevo o $k$ bitů (shift logical left)

## Logický prvek

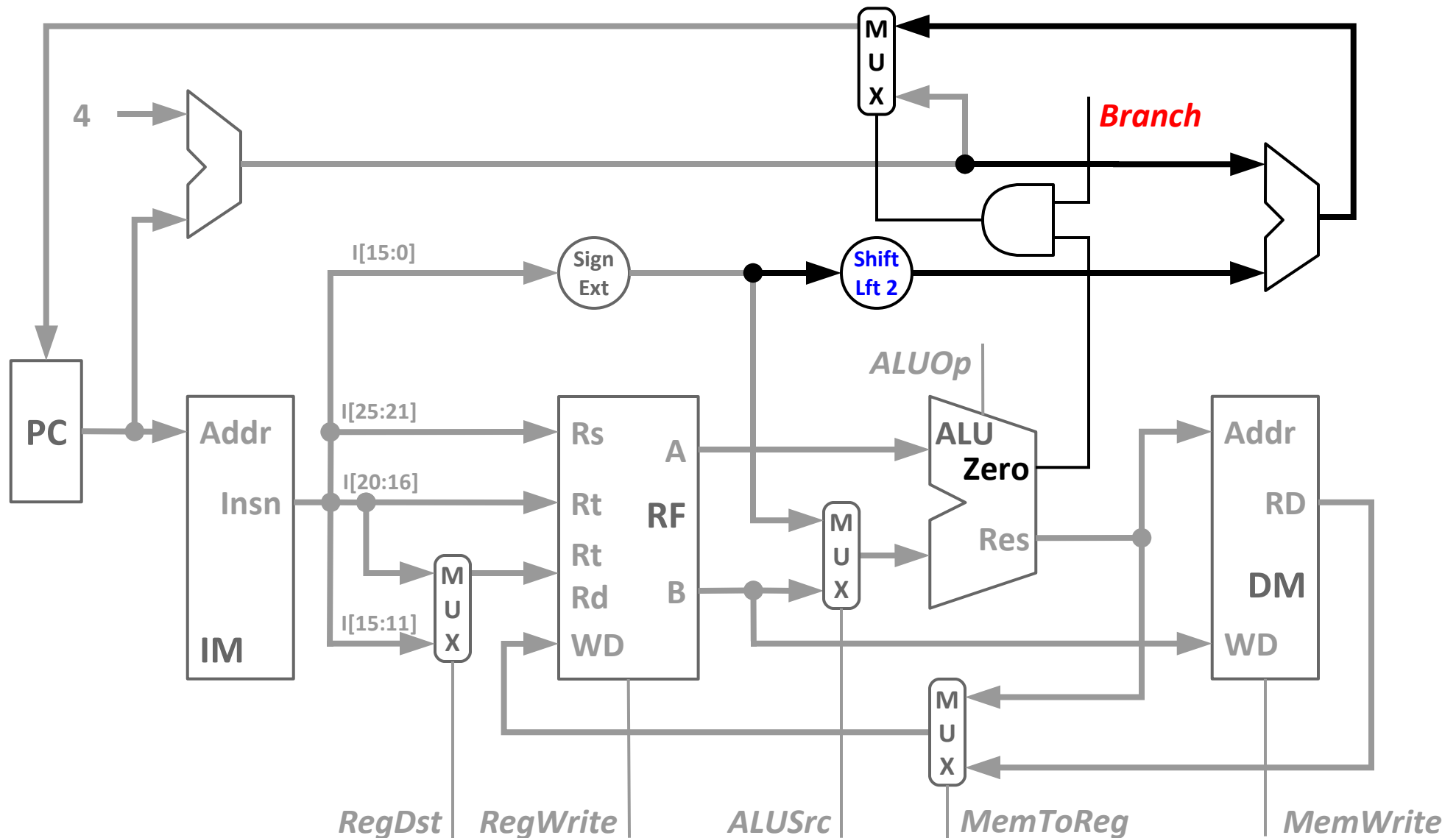
- vstup:  $n$ -bitové číslo  $X$
- výstup:  $n$ -bitové číslo  $Y = X \ll k$



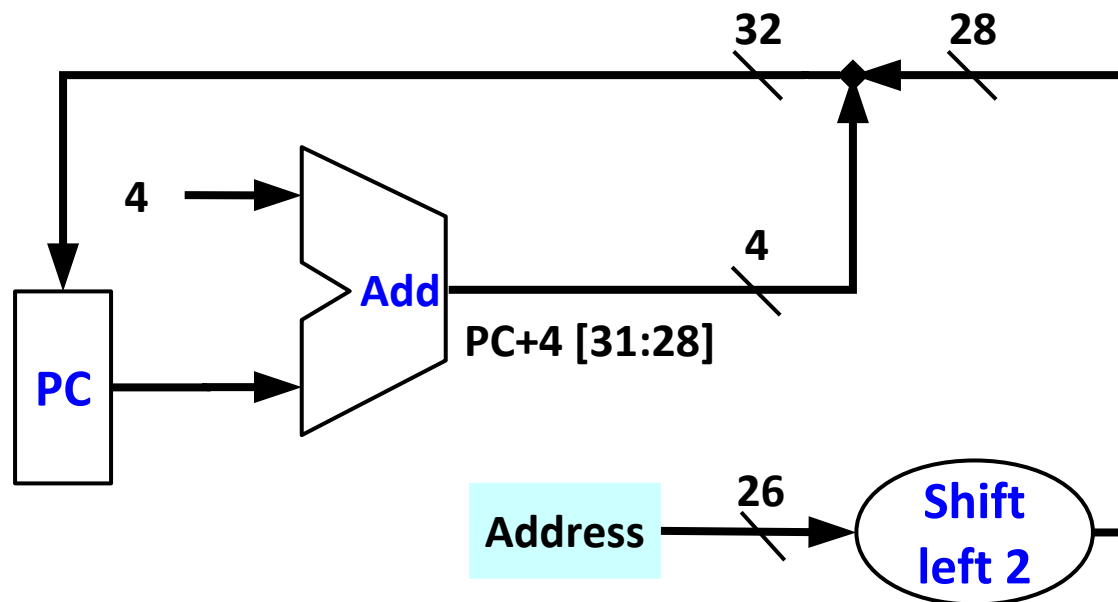
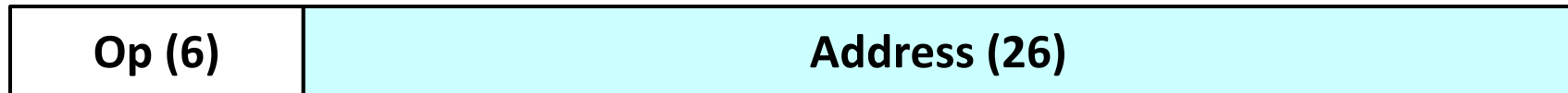
## Příklad pro $n=16, k=2$



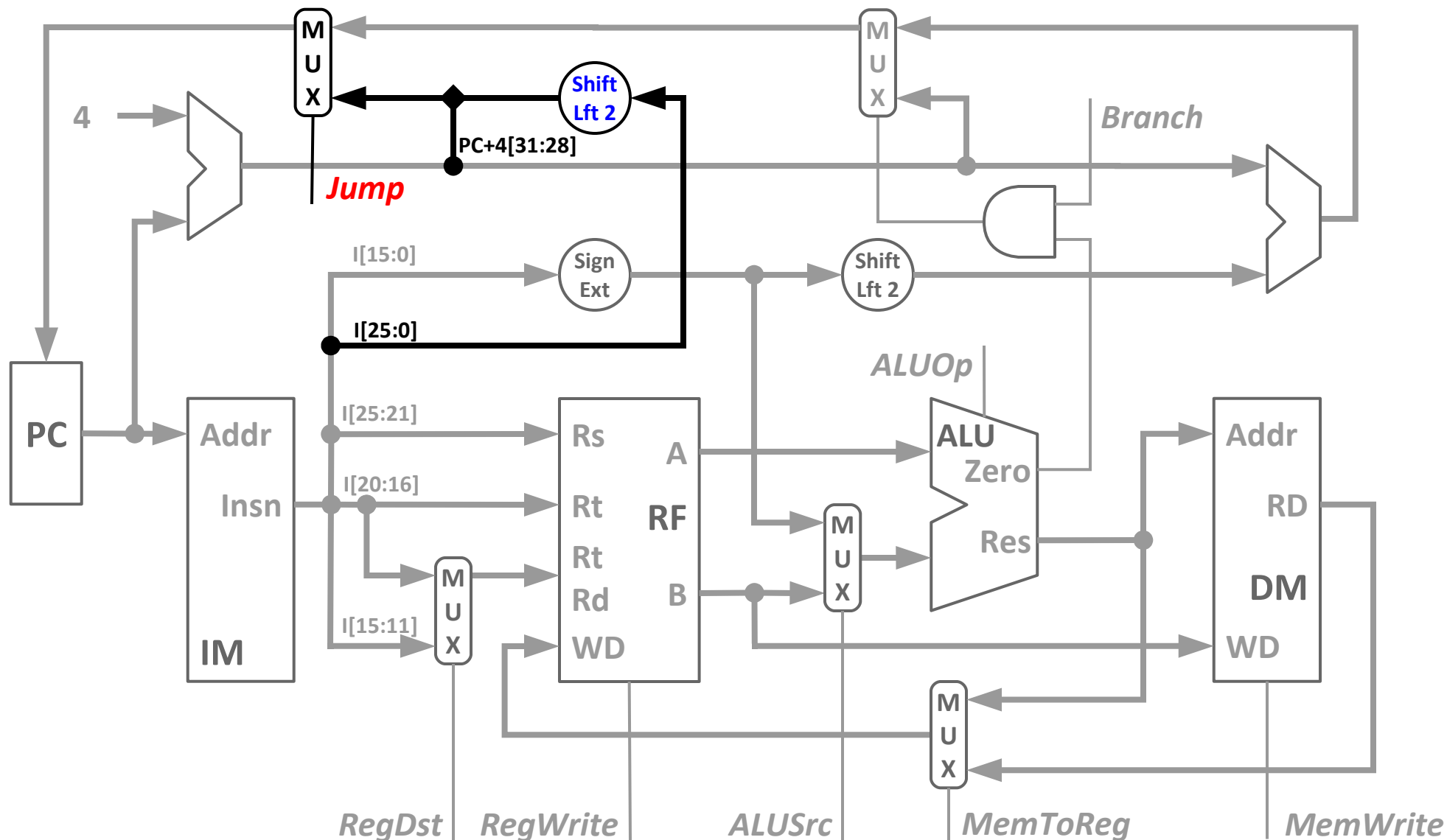
# Podpora podmíněného skoku s relativní adresou



# Nepodmíněný skok s absolutní adresou (jump)



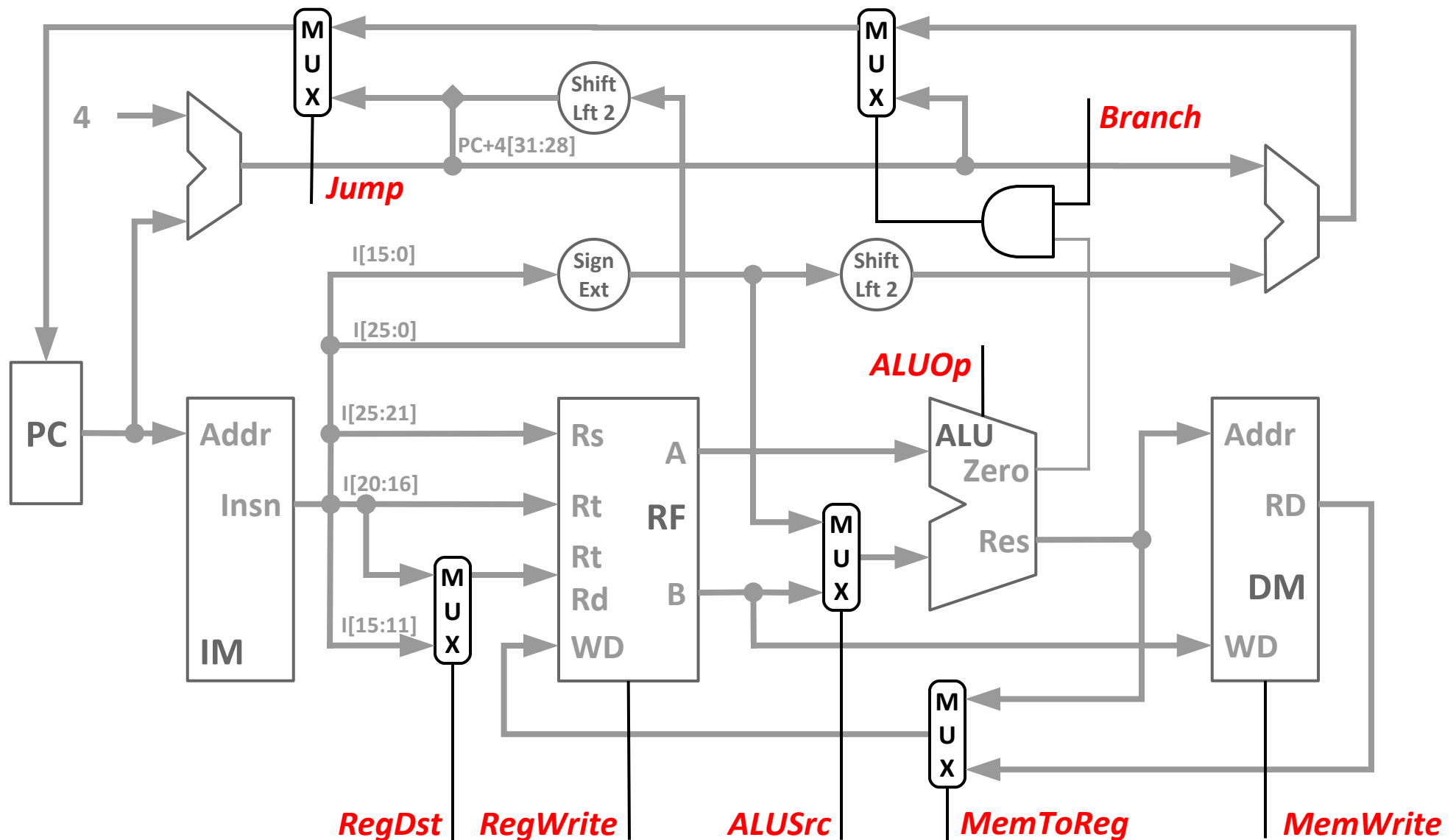
# Podpora nepodmíněného skoku s absolutní adresou



# Řízení jednocyklové datové cesty

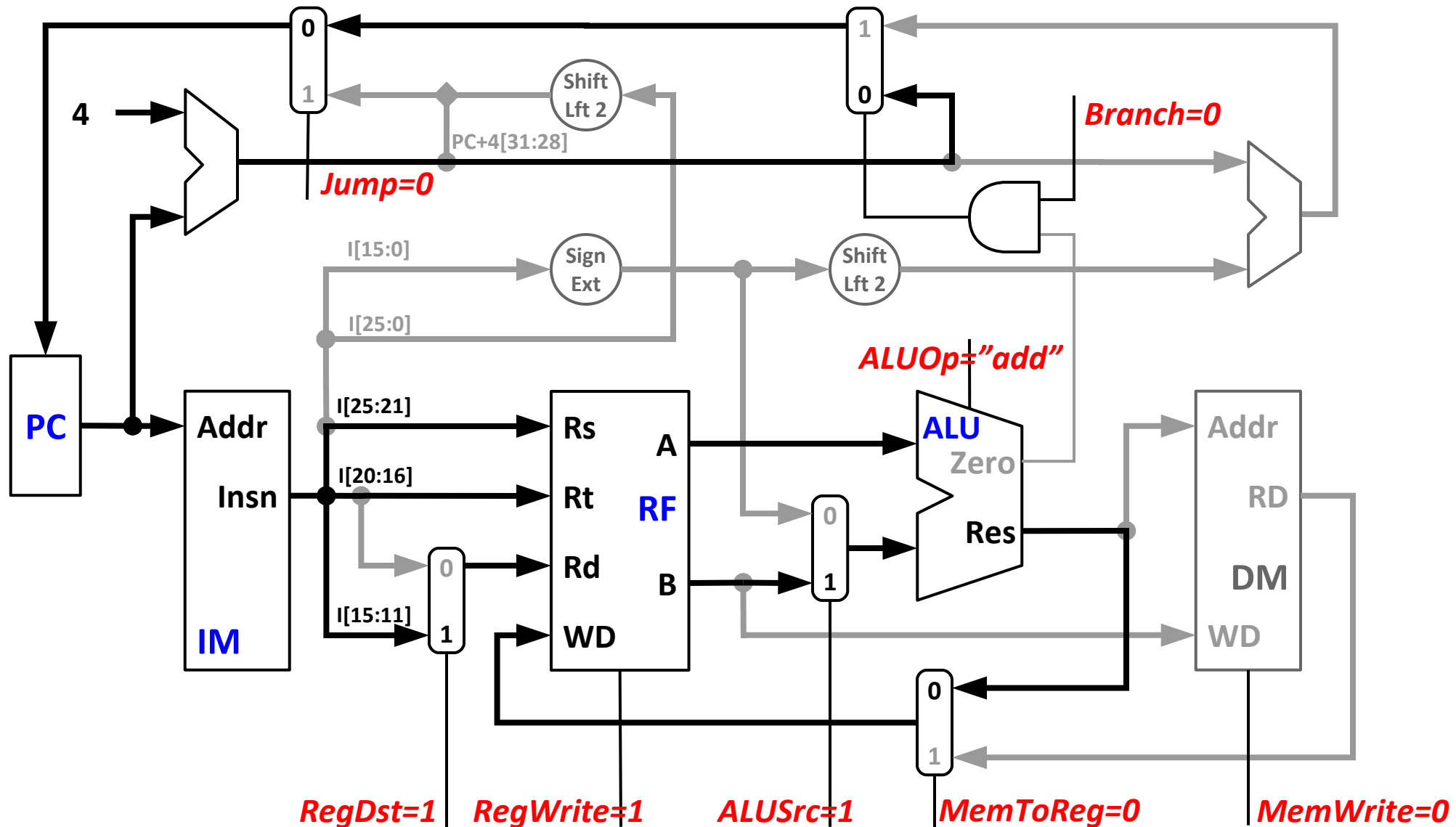
Řízení průchodu dat datovou cestou  
v závislosti na typu operace

# V čem spočívá řízení datové cesty?

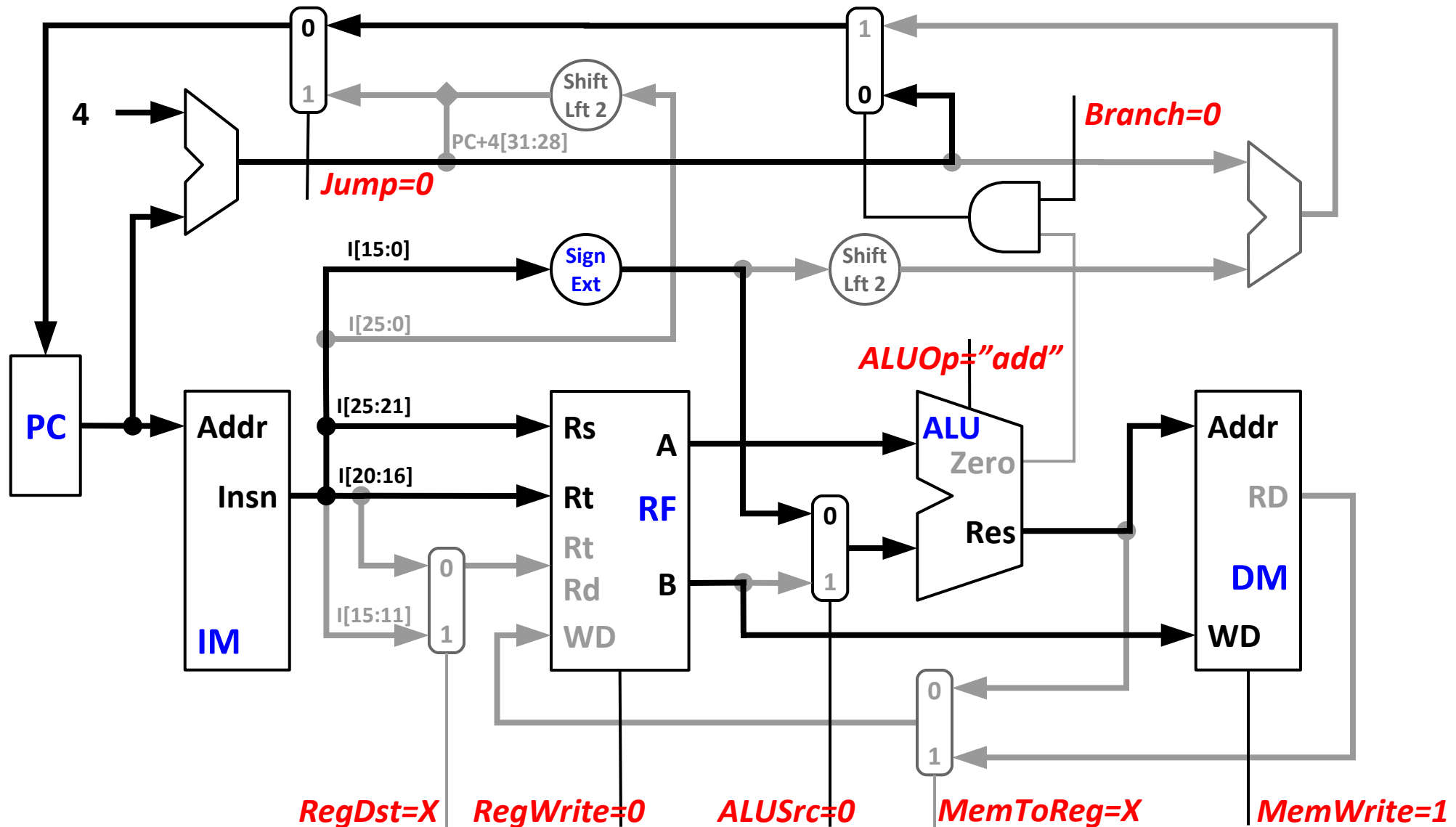




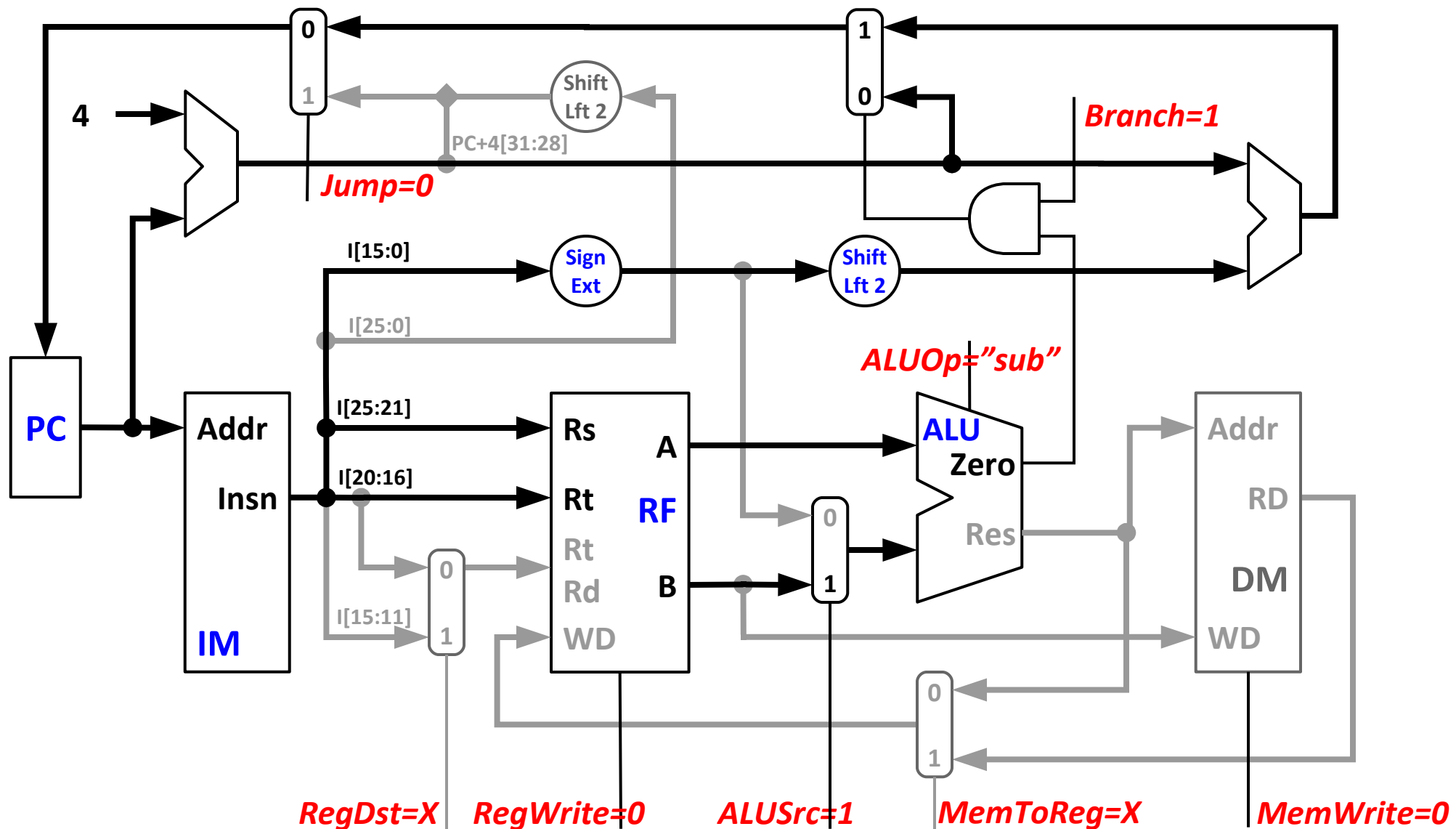
# Příklad: řízení datové cesty pro instrukci "add"



# Příklad: řízení datové cesty pro instrukci "sw"



# Příklad: řízení datové cesty pro instrukci "beq"



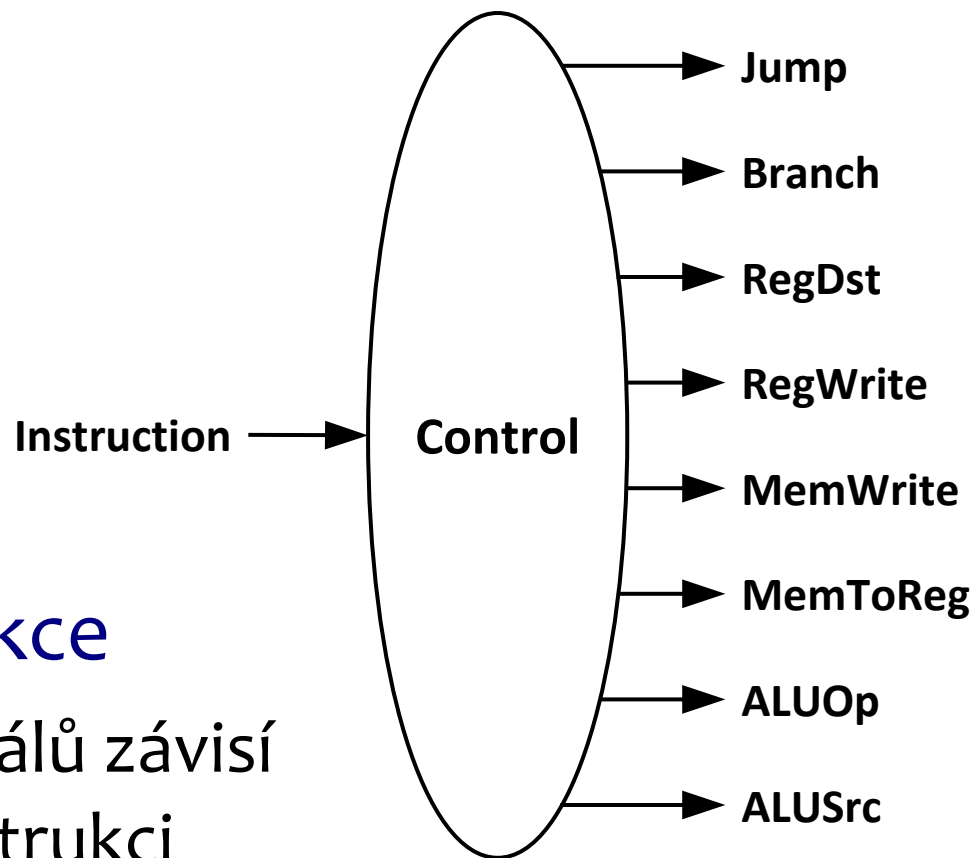
# Radič datové cesty (data path controller)

## Generuje řídicí signály

- zdroj hodnoty PC
- zápis do registrů
- čtení/zápis dat
- operace ALU

## Hodnoty pro různé instrukce

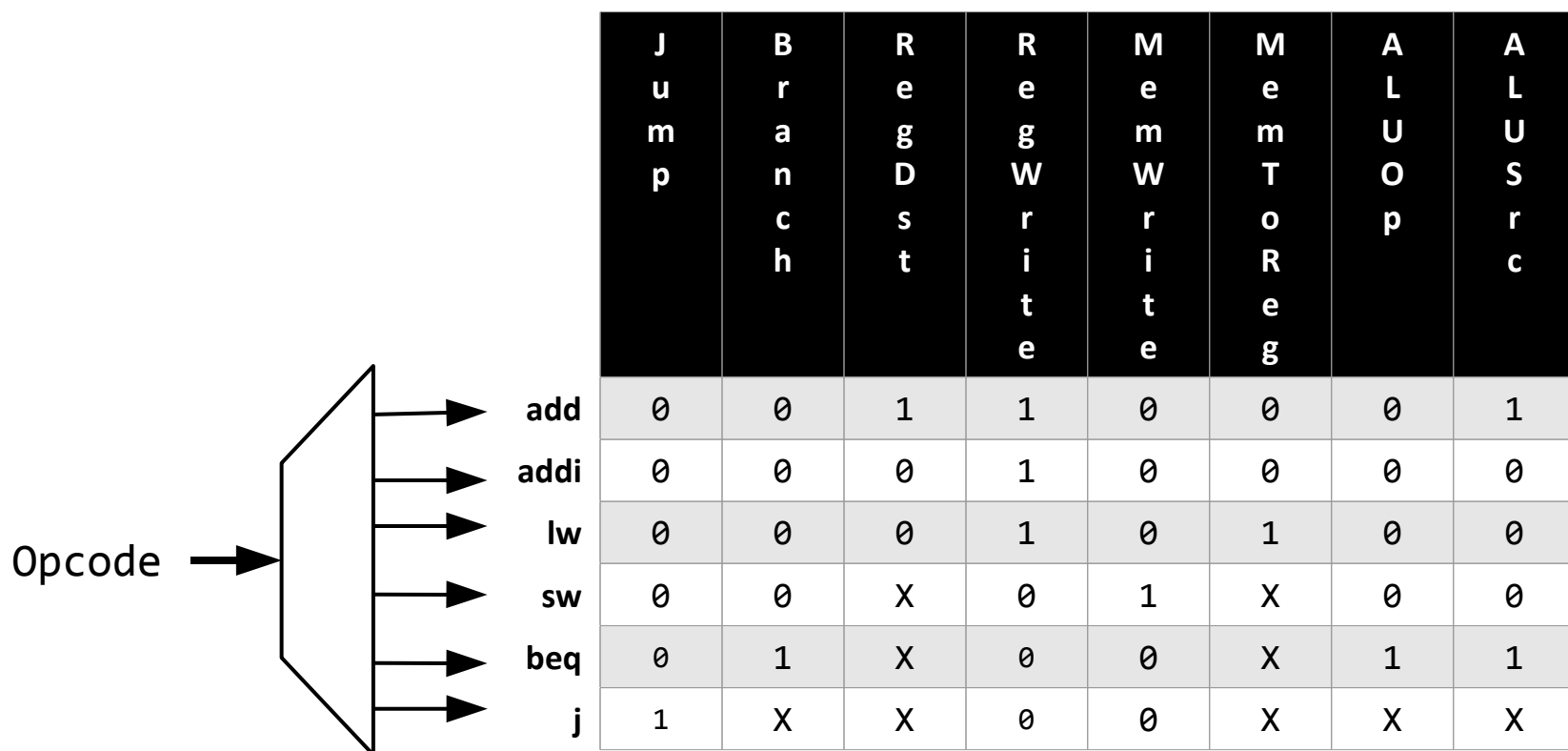
- ohodnocení většiny signálů závisí na operačním kódu v instrukci
- některé mohou být přímo součástí instrukčního kódu
  - ♦ část signálů ALUOp odpovídá bitům v poli “funct”
  - ♦ zjednodušuje implementaci radiče



# Implementace řadiče pomocí ROM

## Řídící paměť

- ROM (*Read Only Memory*) – jako RAM, ale jen pro čtení
- slova v paměti reprezentují hodnoty řídicích signálů
- hodnota operačního kódu adresuje řádky paměti



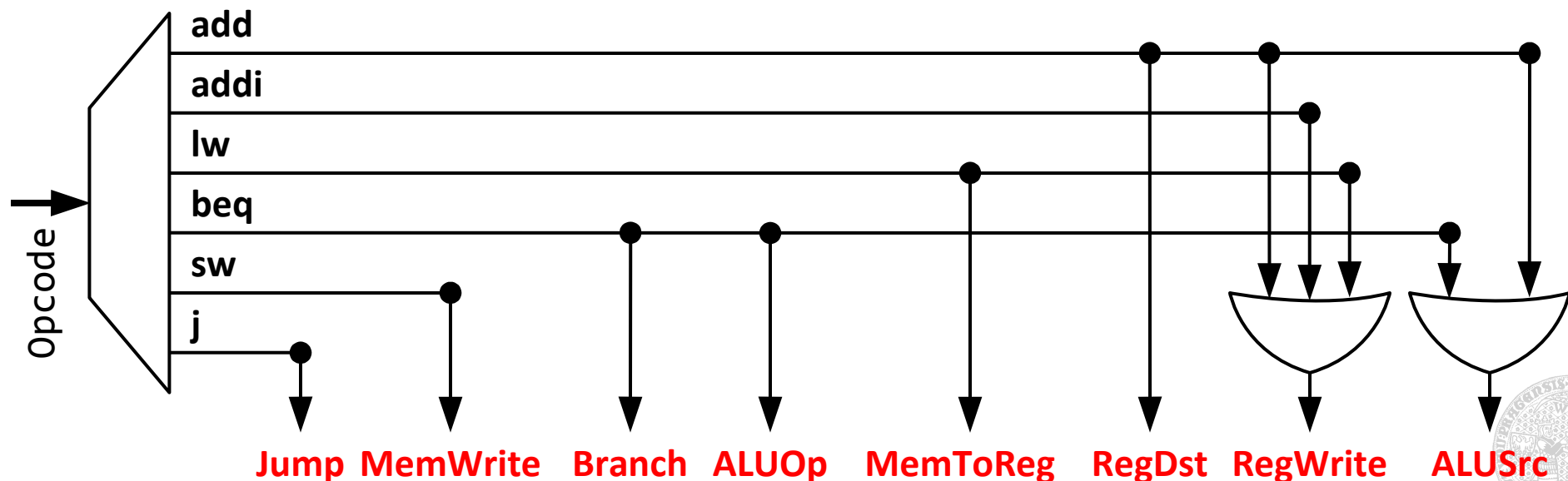
# Implementace řadiče pomocí kombinačního obvodu

Reálný procesor: 100+ instrukcí a 300+ ř. signálů

- kapacita řídicí ROM 30000+ bitů (~4 KiB)
- problém je udělat ROM **rychlejší** než datovou cestu

Alternativa k ROM: kombinační obvod

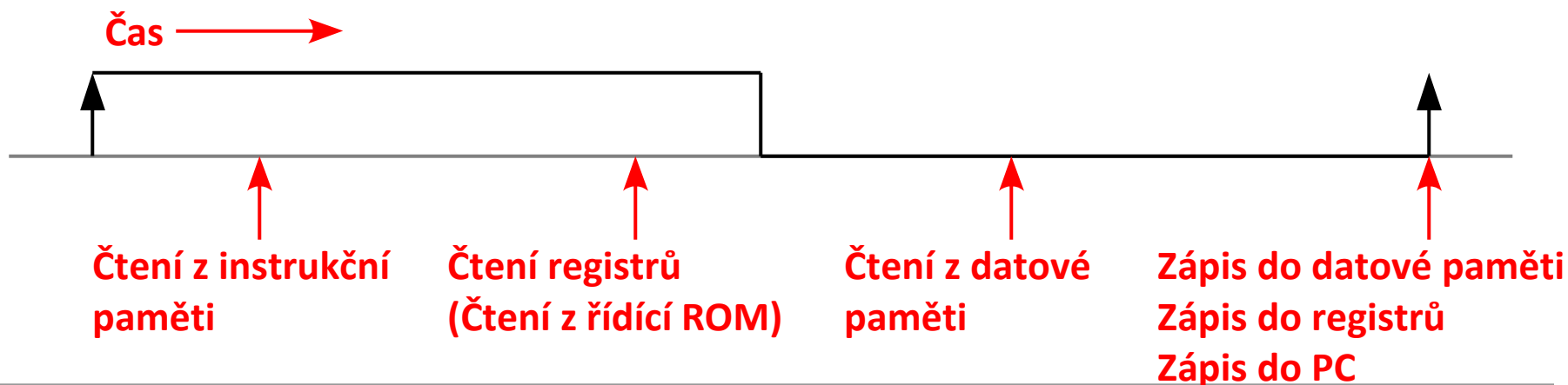
- mnoho signálů má málo jedniček nebo nul  $\Rightarrow$  kompaktní reprezentace log. funkcí pro řídicí signály



# Průběh hodinového cyklu

## Datová cesta s nepřetržitým čtením

- nevadí, protože zápisy (PC, RF, DM) jsou nezávislé
- v rámci cyklu žádné čtení nenásleduje po zápisu
- čtení instrukce (fetch) nepotřebuje řízení
  - ♦ po přečtení instrukce radič dekóduje operační kód na řídicí signály pro zbývající část datové cesty
  - ♦ při změně obsahu PC (adresa instrukce) se začne zpracovávat další instrukce



# Výkon jednocyklového počítače

---

Každá instrukce trvá 1 takt ( $CPI=1$ )

- jednocyklový radič = kombinační obvod nebo řídící ROM
- obecně nižší frekvence hodinového signálu

Délka cyklu odpovídá délce nejdelší instrukce

- v tomto případě “lw”, ale typicky násobení, dělení či floating-point operace
- v rozporu s “*optimize for the common case*”

Datová cesta obsahuje duplicitní prvky

- instrukční a datová paměť, 2 sčítačky navíc

Jde to i lépe?





# **Vícecyklová datová cesta**

## Proměnná doba zpracování instrukce

- jednoduché instrukce by neměly trvat stejně dlouho jako složité
- nelze měnit periodu hodinového signálu  $\Rightarrow$  instrukce rozdělena do menších kroků
  - ♦ perioda hodinového signálu odpovídá **délce kroku**
- v každém taktu proveden 1 krok, počet taktů nutných pro zpracování se pro různé instrukce liší
  - ♦ instrukční cyklus vs. strojový cyklus
  - ♦ aproximace proměnné délky cyklu

Jak dobře to funguje?



# Jednocyklová vs. vícecyclová datová cesta

## Předpokládejme...

- většina instrukcí trvá 10ns, ale násobení trvá 40ns a v programech je (v průměru) 10% instrukcí násobení

## Jednocyklová datová cesta

- perioda hodin 40ns,  **$CPI=1$**
- při 40ns na instrukci je výkon **25 MIPS**

## Vícecyclová datová cesta

- perioda hodin 10ns,  **$CPI \approx (90\% \times 1) + (10\% \times 4) \approx 1.3$**
- při (v průměru) 13ns na instrukci je výkon **77 MIPS**
- vícecyclová datová cesta je  $3\times$  (o 200%) rychlejší



# Rozdělení instrukcí do kroků

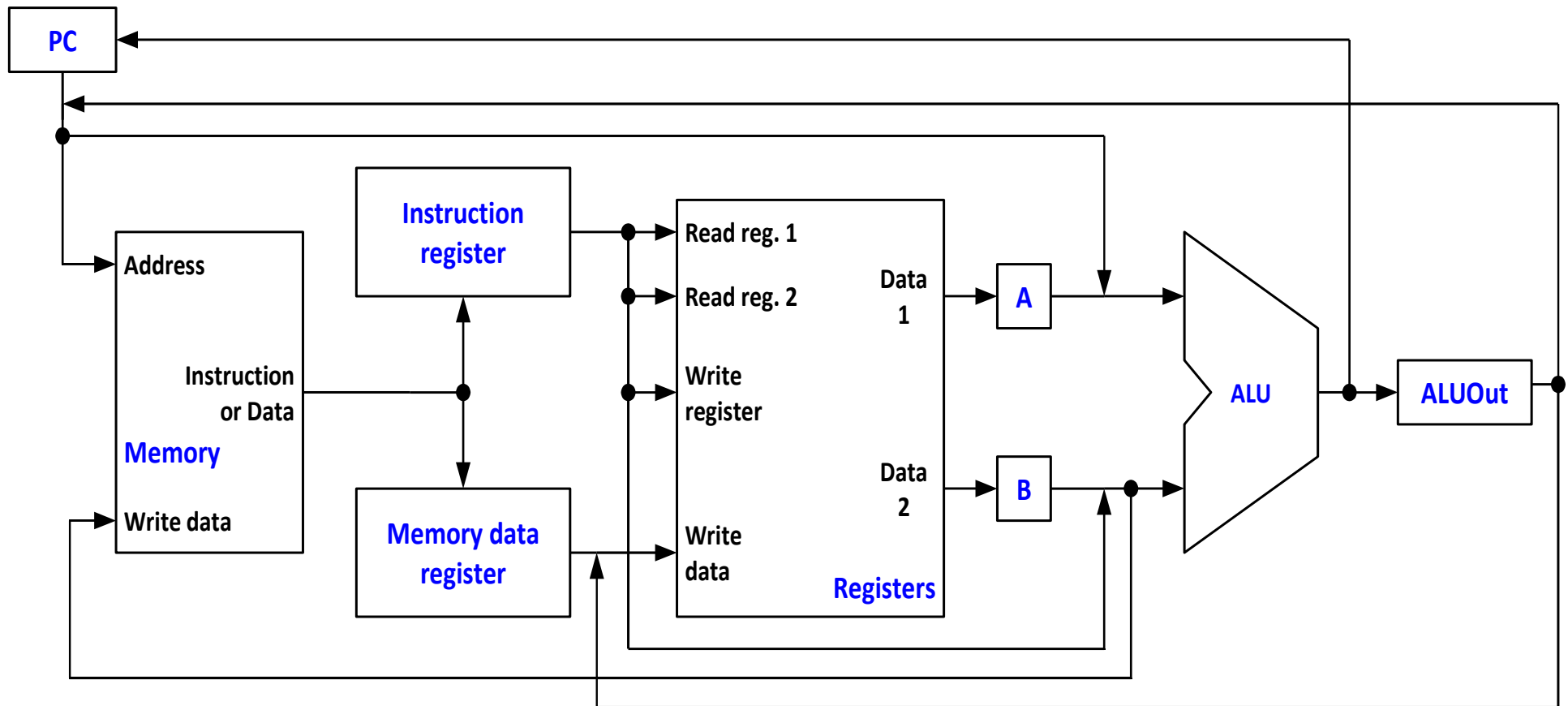
---

## Instrukční cyklus

- načtení instrukce
- dekódování instrukce a přečtení registrů
- vykonání instrukce, výpočet adresy, dokončení větvení
- přístup do paměti a zapsání výsledku
- dokončení čtení z paměti



# Princip implementace vícecyclové datové cesty



- rozdělit na části nutné pro vykonání jednotlivých kroků
  - ♦ části nutno izolovat pomocí registrů pro mezivýsledky
- řadič musí “provést” instrukci datovou cestou
  - ♦ instrukce mohou některé kroky přeskočit a skončit dříve



# Krok 1: čtení instrukce

---

## Současně probíhá

- $IR \leq \text{Memory}[PC]$ 
  - ♦ přečtení instrukce do instrukčního registru
- $PC \leq PC + 4$ 
  - ♦ posun PC na adresu další instrukce
  - ♦ změna hodnoty PC nevadí, protože přečtená instrukce je již v instrukčním registru



## Krok 2: dekódování instrukce a čtení registrů

### Současně probíhá

- $A \leq \text{Reg}[\text{IR.rs}]$ 
  - ♦ přečtení obsahu zdrojového registru 1
- $B \leq \text{Reg}[\text{IR.rt}]$ 
  - ♦ přečtení obsahu zdrojového registru 2
- $\text{ALUOut} \leq \text{PC} + (\text{sign-extend}(\text{IR.addr}) \ll 2)$ 
  - ♦ výpočet adresy podmíněného skoku
  - ♦ pokud instrukce není skok, výsledek se nepoužije

Další kroky se liší podle typu instrukce...



# Krok 3: provedení operace/výpočet adresy v paměti

## Podmíněný skok (konec)

- if (A == B) then PC <= ALUOut
  - ♦ adresu skoku máme v ALUOut z předchozího kroku

## Nepodmíněný skok (konec)

- $PC \leq PC[31:28] + IR[25:0] \ll 2$

## Aritmeticko-logická operace

- $ALUOut \leq A \text{ funct } B$

## Přístup do paměti

- $ALUOut \leq A + \text{sign-extend}(IR[15:0])$ 
  - ♦ výpočet adresy pro přístup do paměti





## Krok 4: přístup do paměti/zápis výsledku

---

### Aritmeticko-logická operace (konec)

- $\text{Reg}[\text{IR.rd}] \leq \text{ALUOut}$ 
  - ♦ výsledek operace zapsán do cílového registru

### Zápis do paměti (konec)

- $\text{Memory}[\text{ALUOut}] \leq B$ 
  - ♦ obsah registru zapsán do paměti

### Čtení z paměti

- $\text{MDR} \leq \text{Memory}[\text{ALUOut}]$ 
  - ♦ obsah paměti přečten do pomocného registru



# Krok 5: zápis dat z paměti do registru

---

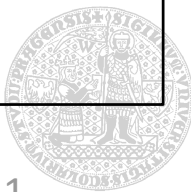
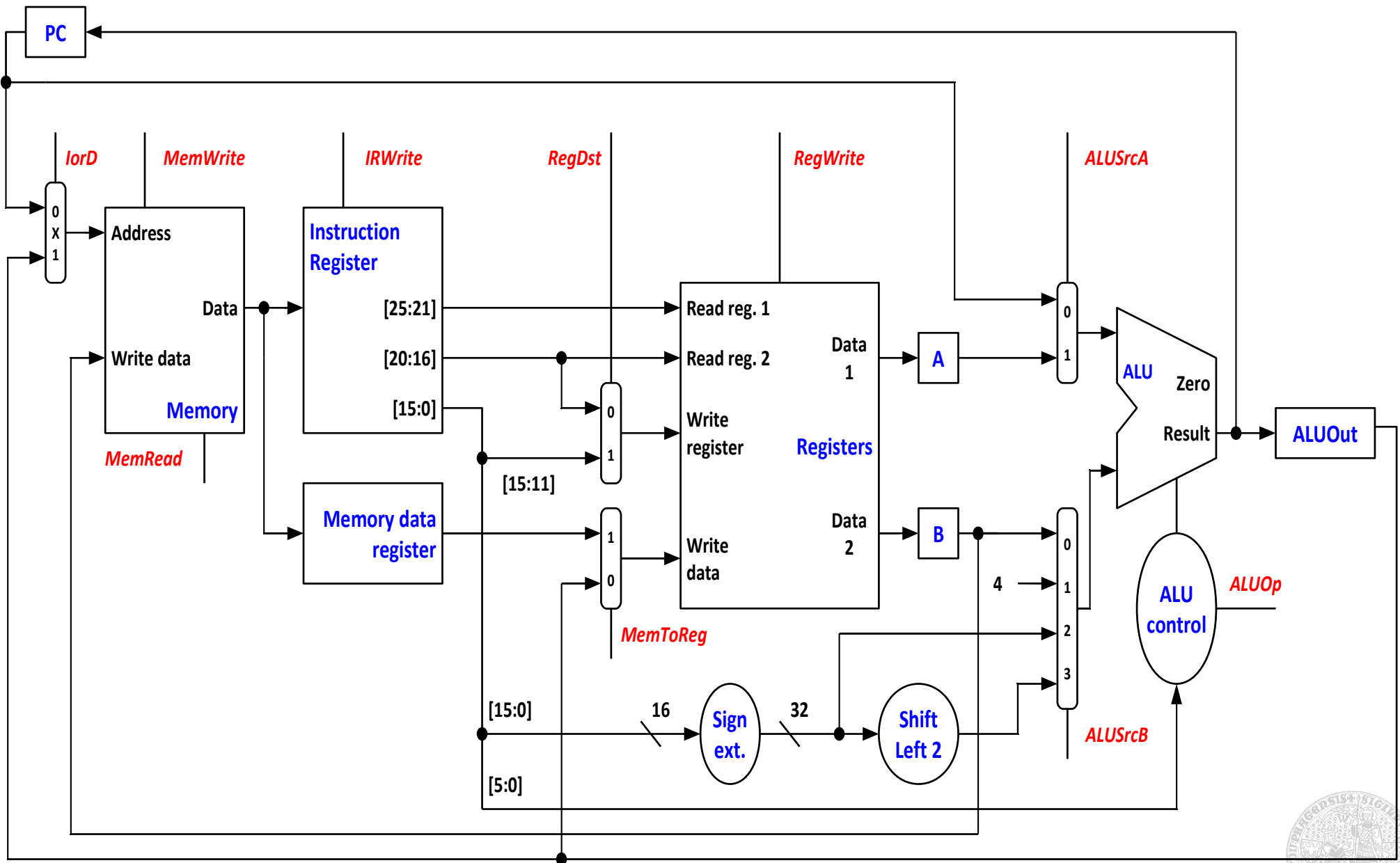
## Čtení z paměti (konec)

- $\text{Reg}[\text{IR.rt}] \leftarrow \text{MDR}$ 
  - ♦ zápis přečtené hodnoty do registru



# **Řízení vícecyclové datové cesty**

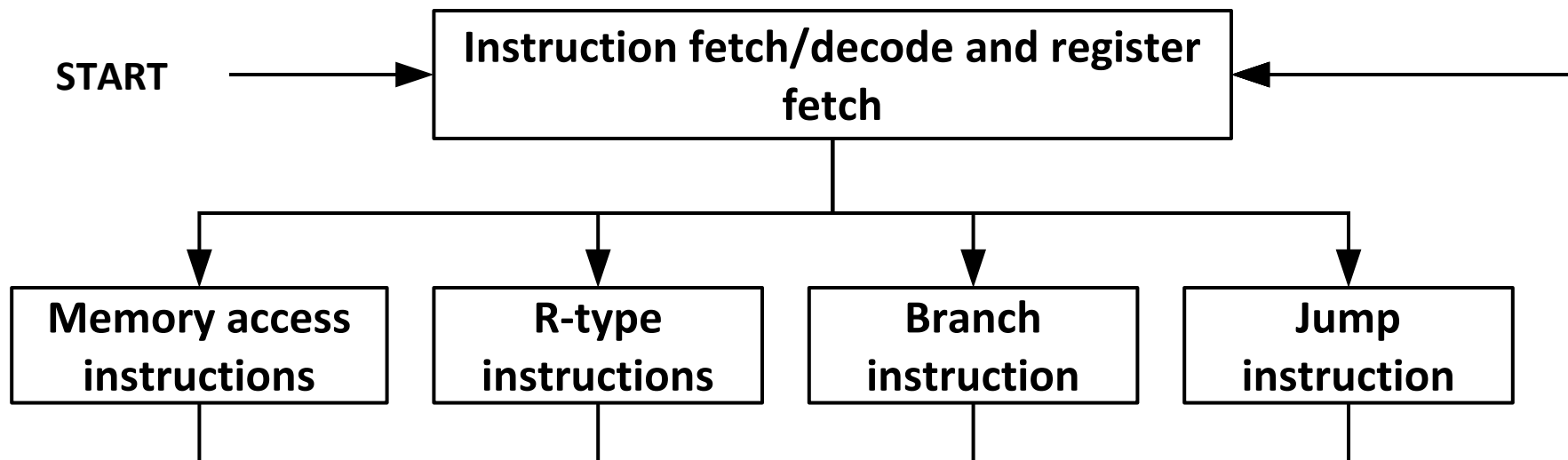
# Implementace vícecyclové datové cesty



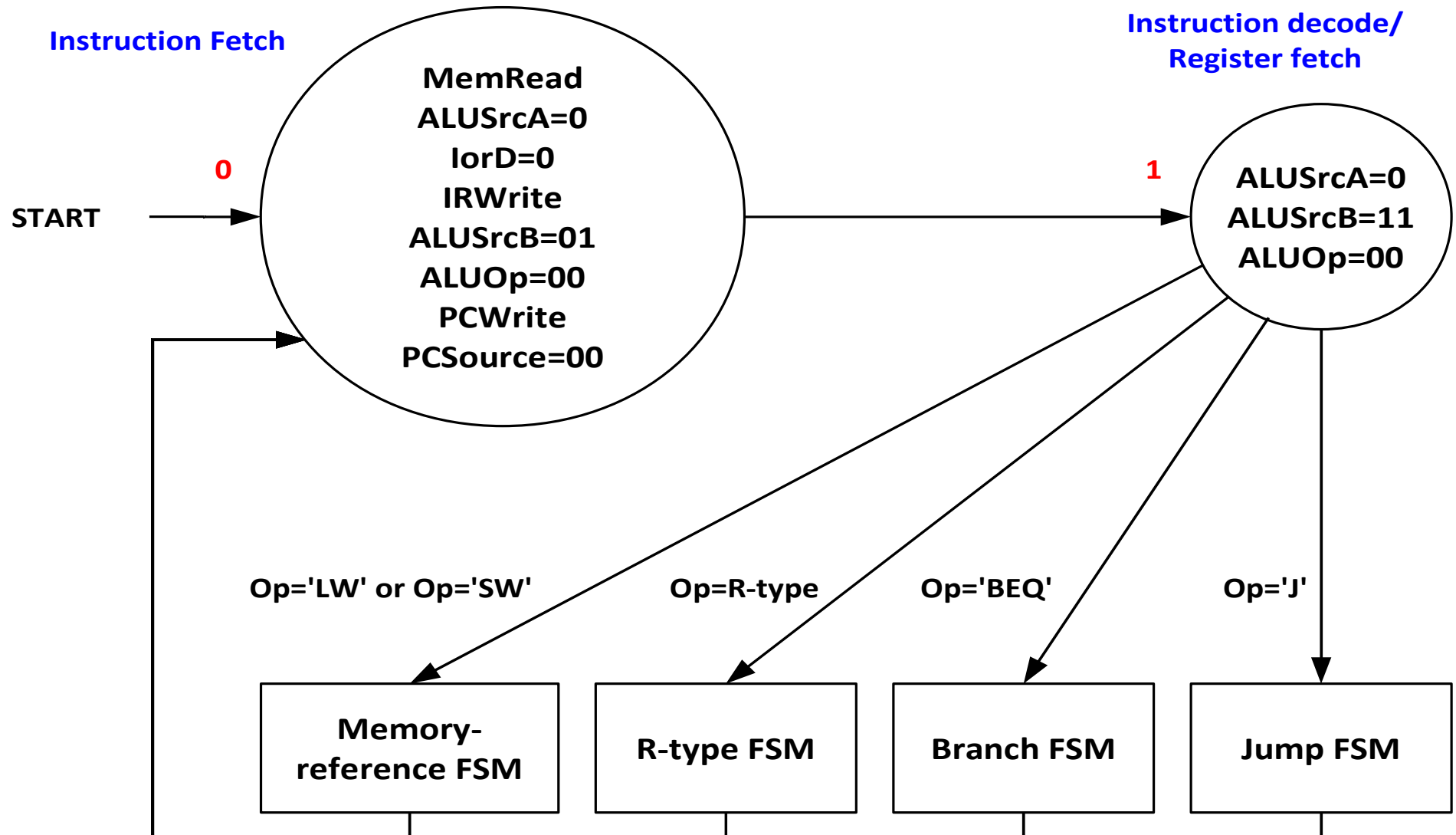
# Řízení vícecyclové datové cesty je sekvenční

## Zpracování instrukce trvá více taktů

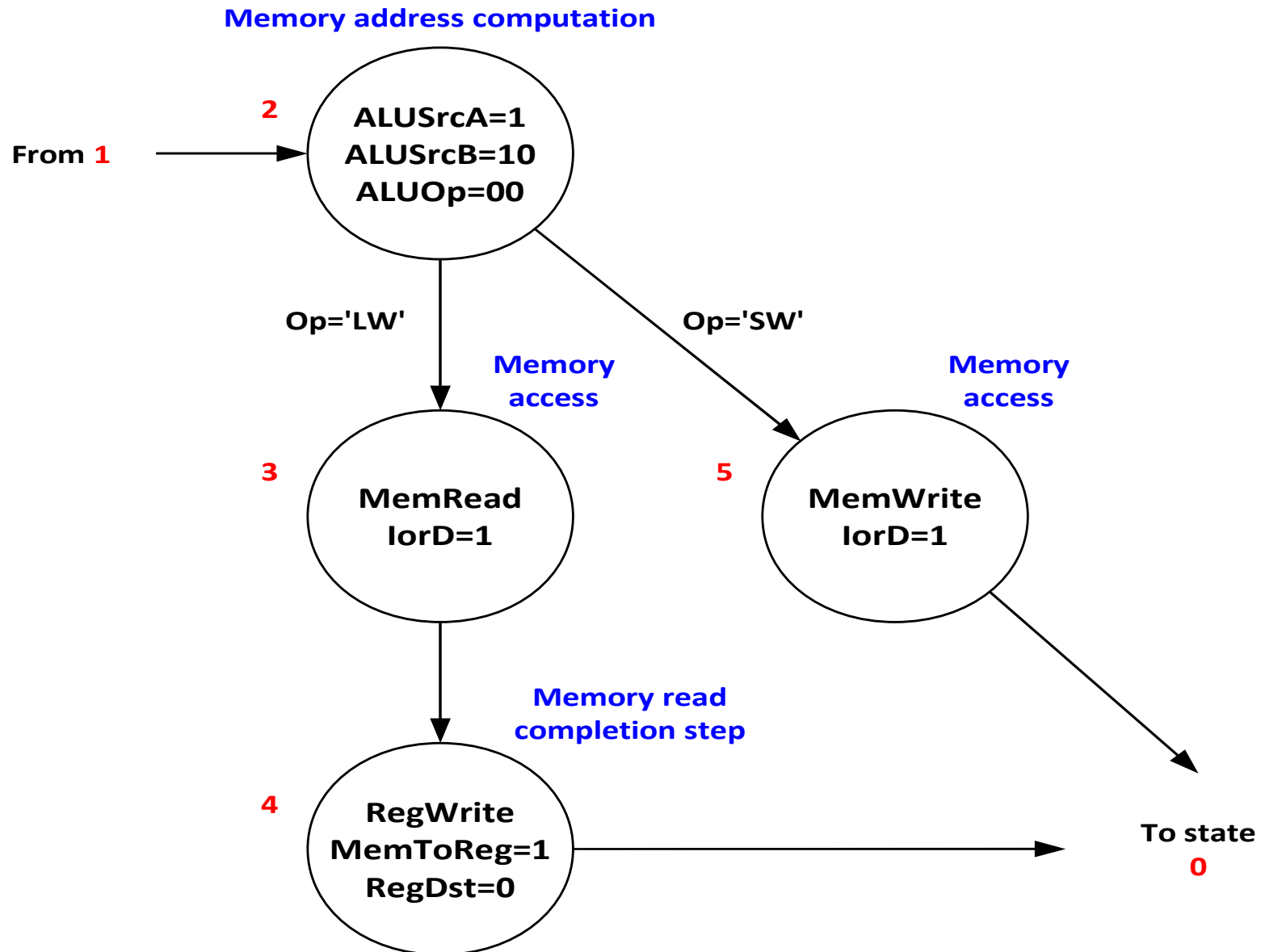
- řadič musí vědět, který krok zpracování instrukce má provést  $\Rightarrow$  sekvenční obvod  $\Rightarrow$  konečný automat
  - ♦ nachází se v jednom z množiny možných stavů, stav uchován v paměťovém prvku (registru)
  - ♦ kombinační logika určí následující stav, který se do stavového registru zapíše s náběžnou hranou hod. signálu



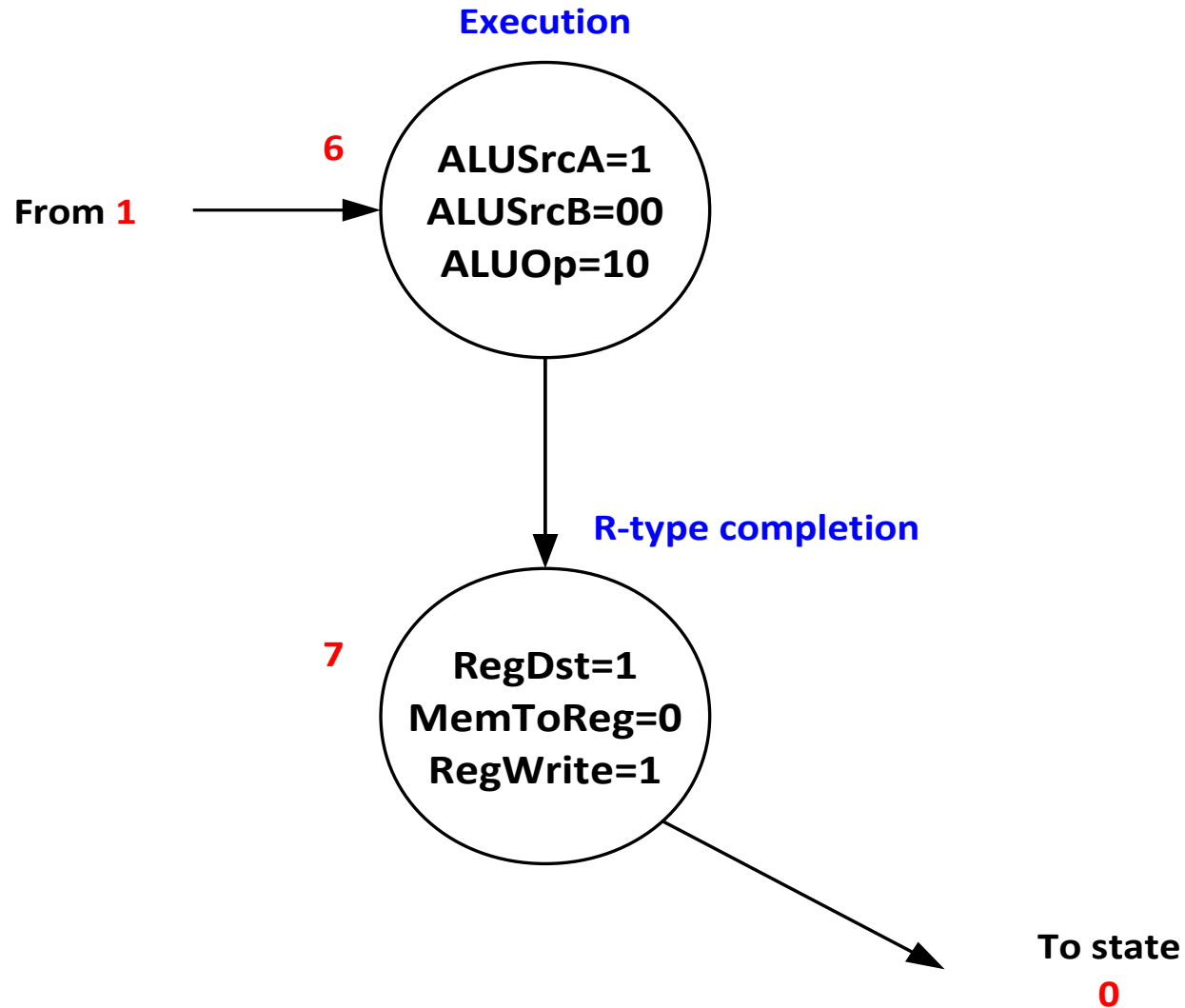
# Instruction fetch/decode and register fetch



# Memory reference FSM

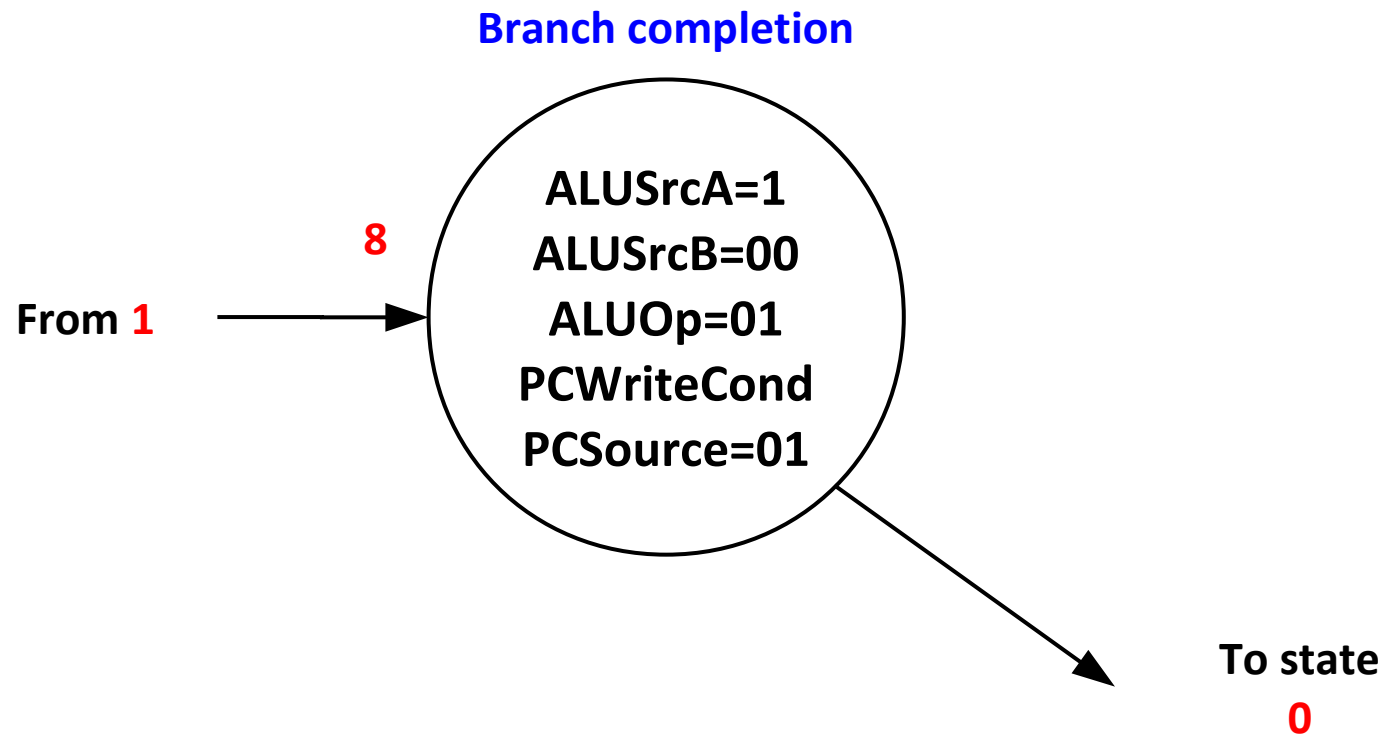


# R-type FSM

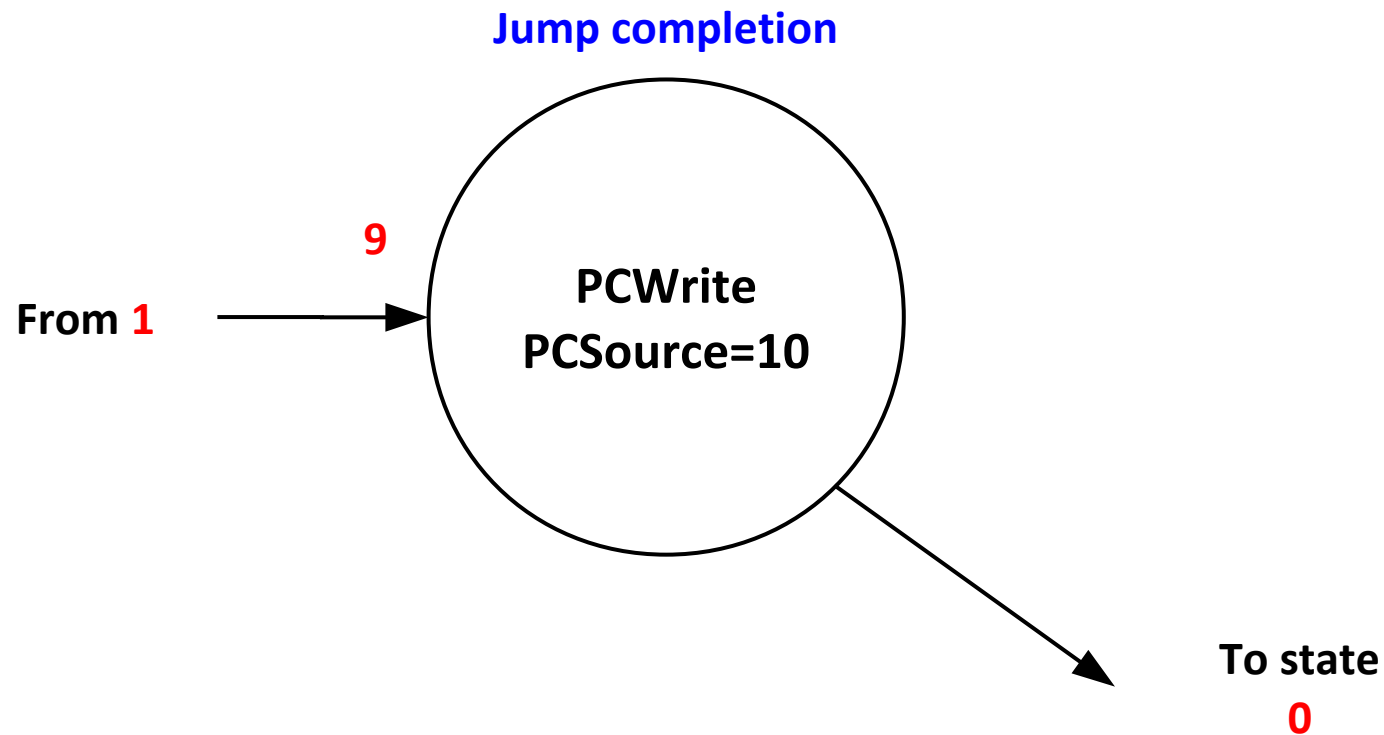




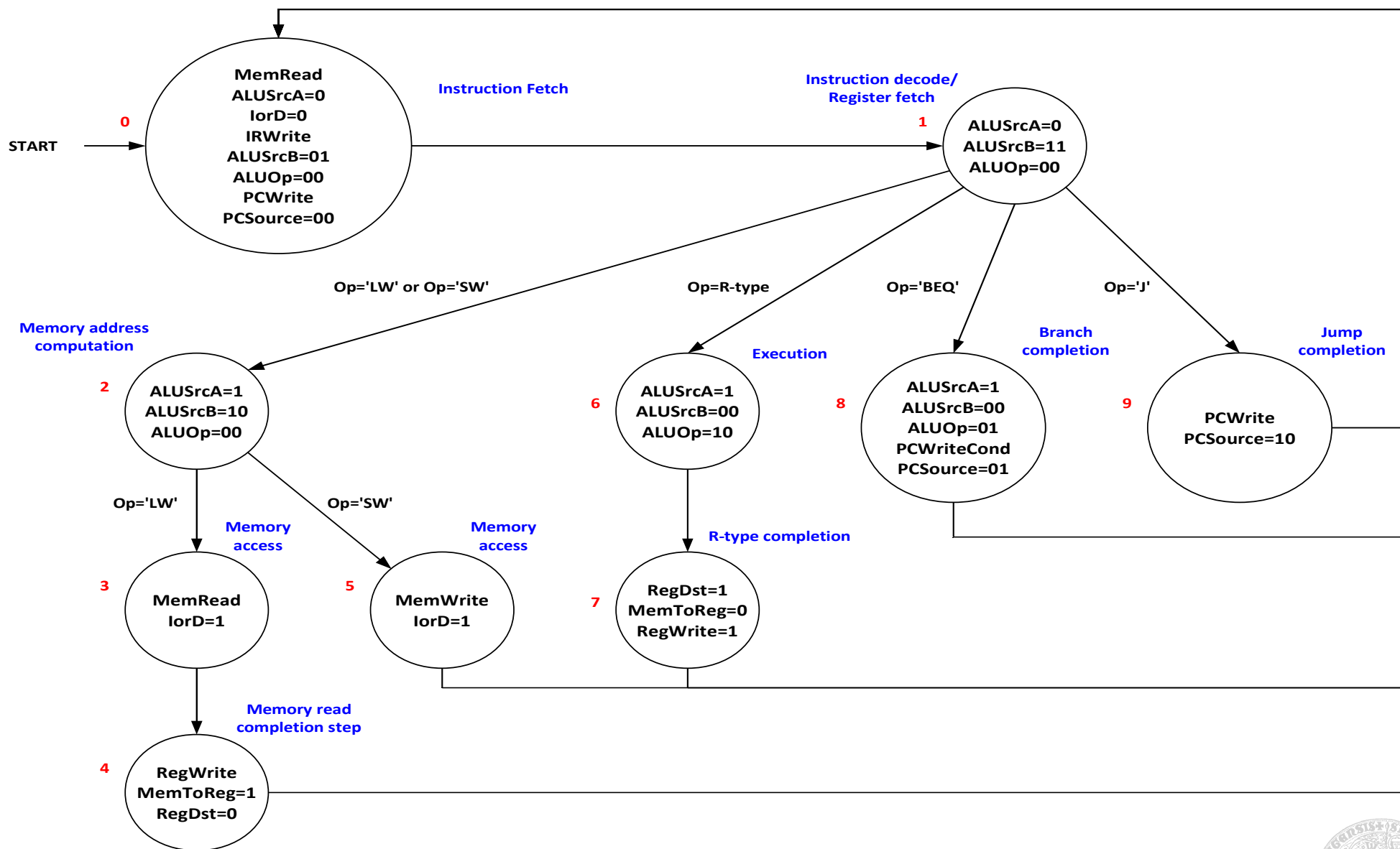
# Branch FSM



# Jump FSM



# Konečný automat pro řadič vícecyklové datové cesty



# Výjimky a přerušení

---

## Neočekávaná změna toku provádění instrukcí

- jiná příčina než jump/branch

## Vnitřní (exception, trap)

- aritmetické přetečení
- nedefinovaná instrukce
- vyvolání služby operačního systému
- selhání hardware

## Vnější (interrupt)

- periferní zařízení
- selhání hardware



# Podpora výjimek a přerušení

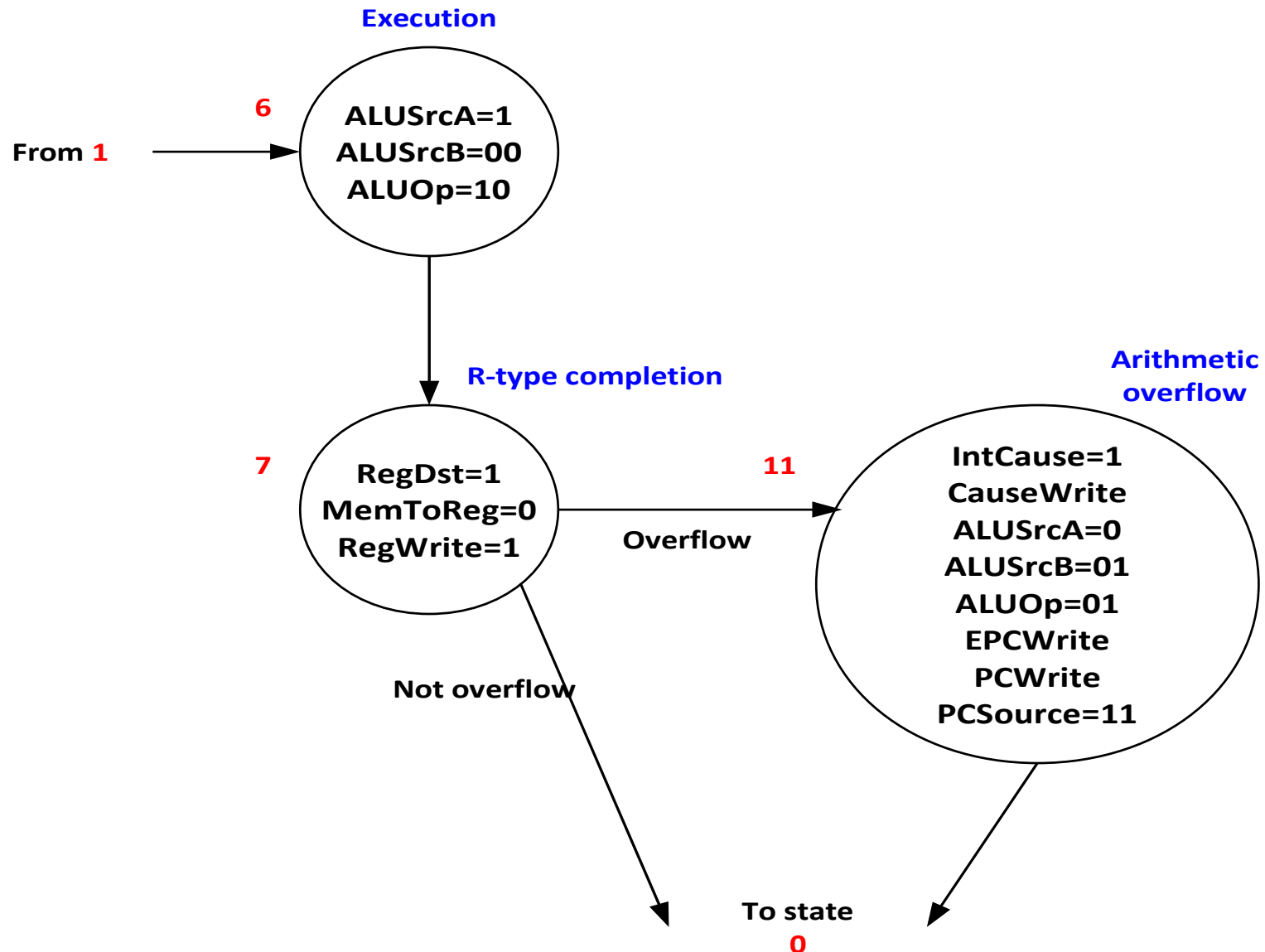
---

## Hardware

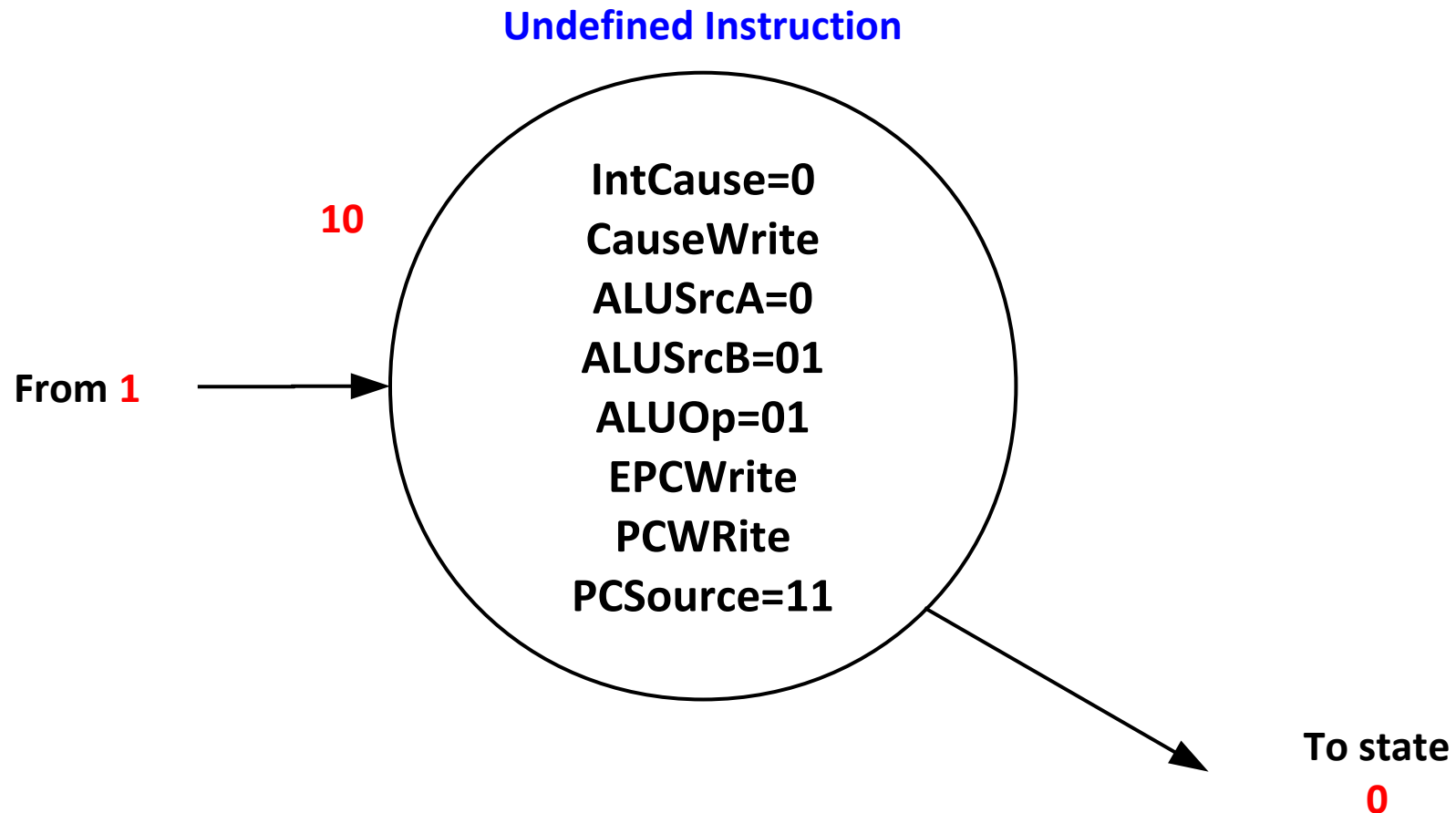
- zastavení vykonávání instrukce
  - ♦ důležité je zachovat korektní stav procesoru
- zajistit možnost identifikace příčiny
  - ♦ příznakové bity ve speciálním registru
  - ♦ případně další upřesňující informace
- uschovat adresu instrukce, při které výjimka nastala
  - ♦ umožňuje pokračovat v běhu (přerušeného) programu
- skok na adresu obslužné rutiny
  - ♦ stejná adresa pro všechny typy výjimek (MIPS)
  - ♦ různé adresy pro různé výjimky (Intel)



# Podpora pro výjimku při aritmetickém přetečení



# Podpora pro výjimku při neplatné instrukci



# Obsluha výjimek a přerušení

---

## Realizuje software

- uschování stavu původního výpočtu
- zjištění příčiny výjimky/přerušení
- obsluha příslušného typu výjimky
  - ♦ může dojít k ukončení výpočtu
- obnovení stavu původního výpočtu
- návrat do původního programu
  - ♦ pokračovat následující instrukcí
  - ♦ restartovat instrukci, která výjimku vyvolala





# Výkon vícecyclové datové cesty

## Instrukční mix

- 30% load (5ns), 10% store (5ns)
- 50% add (4ns), 10% mul (20ns)

## Jednocyklová datová cesta (takt 20ns, CPI=1)

- **20ns** na instrukci  $\Rightarrow$  výkon **25 MIPS**

## Jednoduchá vícecyclová datová cesta (takt 5ns)

- $CPI \approx (90\% \times 1) + (10\% \times 4) = 1.3$
- **6.5ns** na instrukci  $\Rightarrow$  výkon **153 MIPS**

## Jemně členěná vícecyclová datová cesta (takt 1ns)

- $CPI \approx (30\% \times 5) + (10\% \times 5) + (50\% \times 4) + (10\% \times 10) = 6$
- **6ns** na instrukci  $\Rightarrow$  výkon **166 MIPS**



# **Realizace vícecyklového řadiče**

# Více-cyklový řadič = konečný automat

---

## Konečný automat

- množina platných stavů, vnitřní stav, počáteční stav
- přechody mezi stavy v závislosti na stavu a vstupech

## Realizace konečného automatu

- stav + podmínky = paměť + logika = sekvenční obvod
  - ♦ realizace závisí na reprezentaci vnitřního stavu
- obvodové řešení
  - ♦ stavový registr, kombinační logika
  - ♦ posuvný řetězec klopných obvodů
- paměť + jednoduchý sekvenční obvod (místo dekodéru)
  - ♦ mikroprogramování, nanoprogramování



## Obvodová realizace konečného automatu

- přechody stavovým diagramem
- standardní metody sekvenční logiky

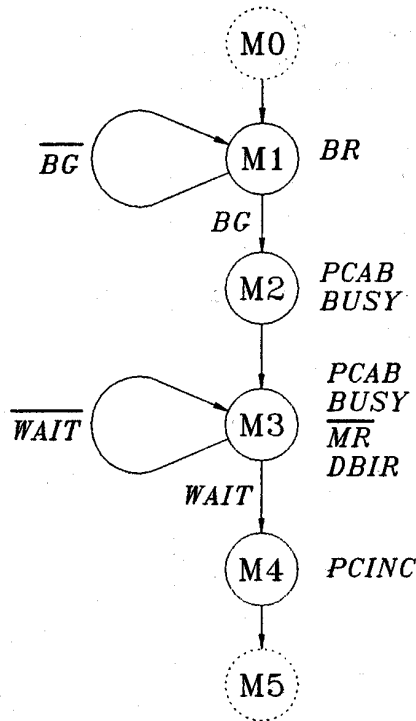


Figure 7.9. Instruction fetch  
— state diagram

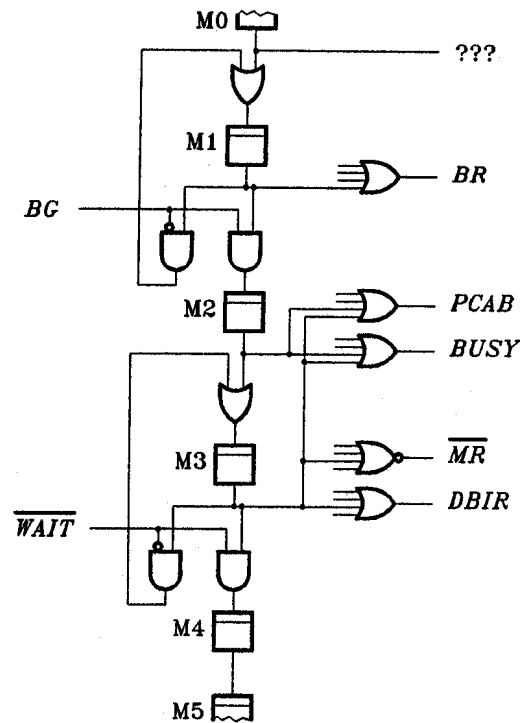
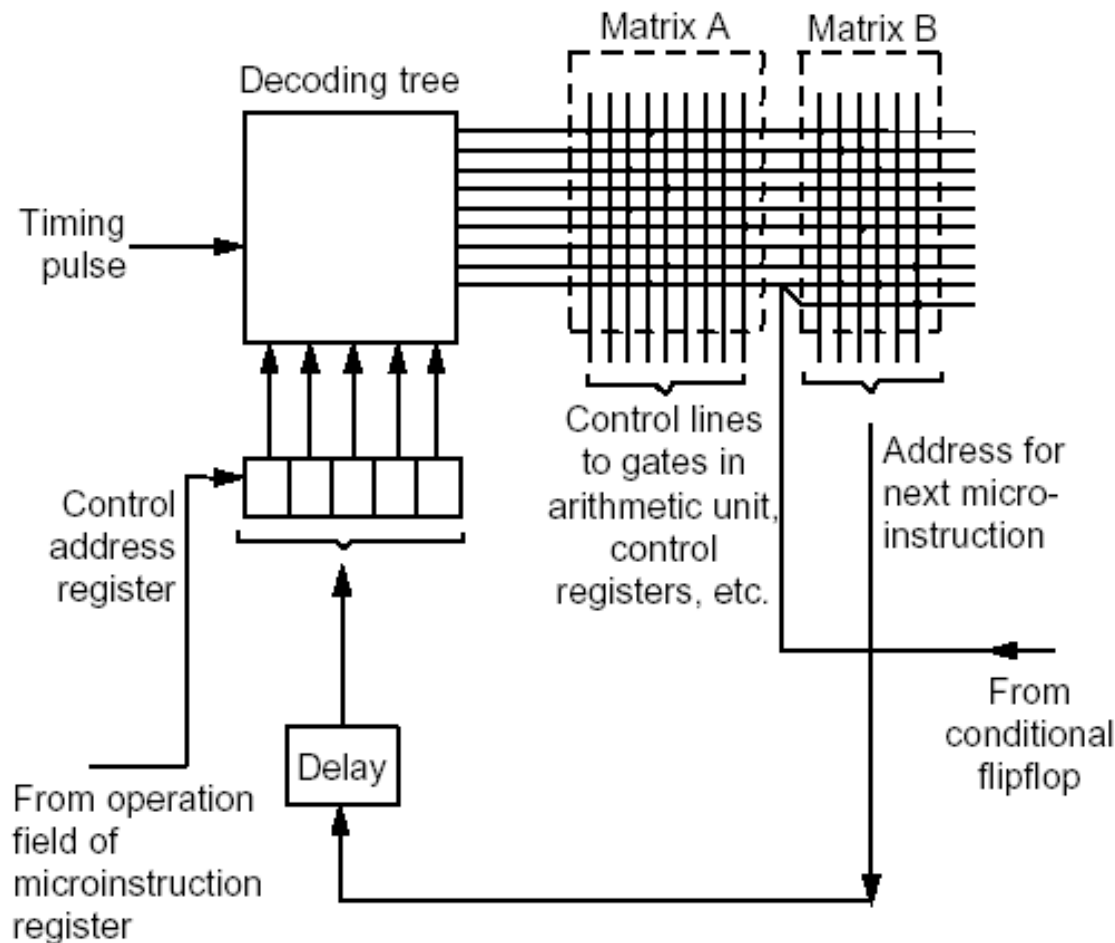


Figure 7.10. Part of hard-wired controller

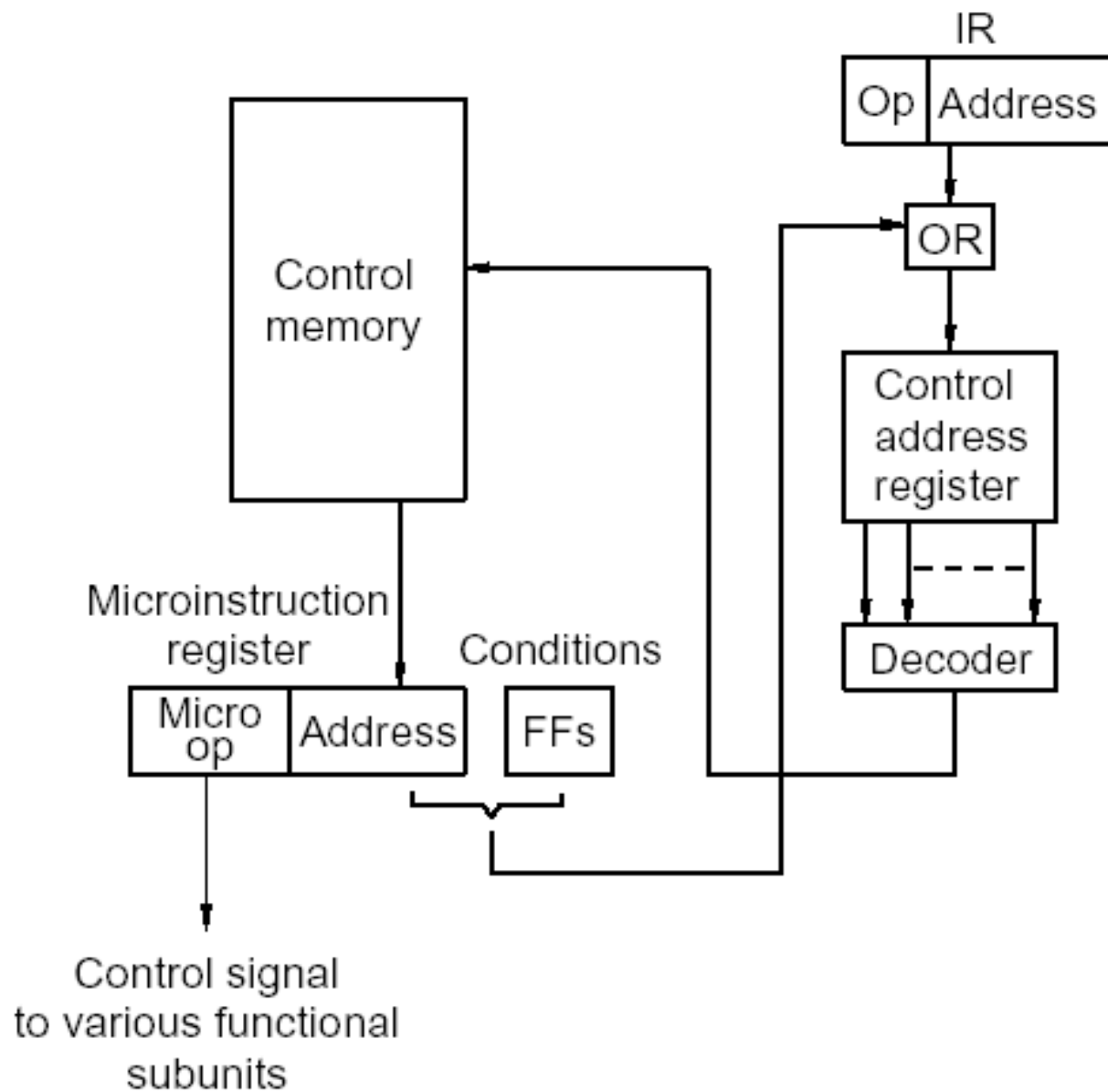


# Mikroprogramový řadič

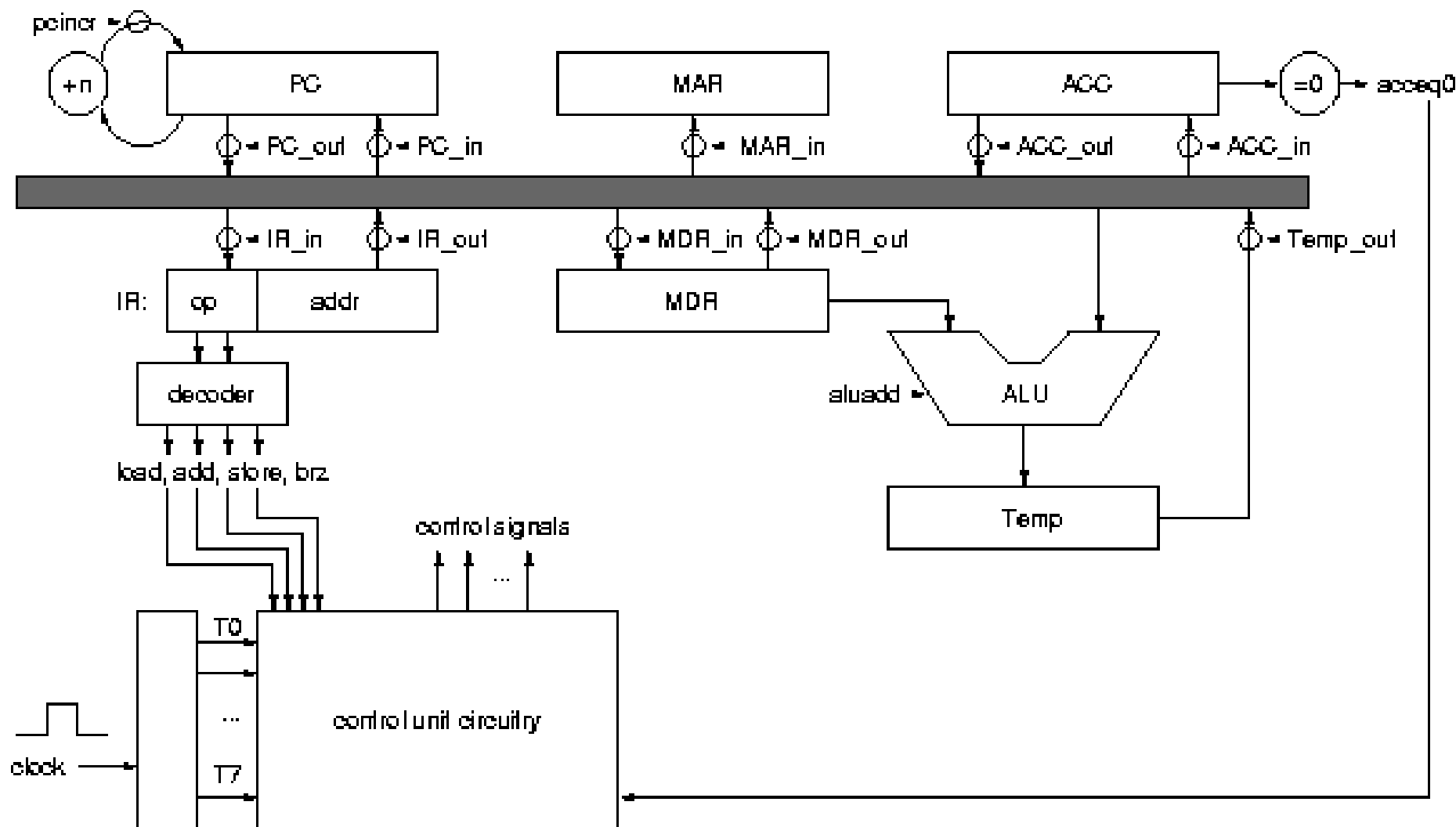
Maurice V. Wilkes, 1951



# Princip vykonávání mikrokódu



# Mikroprogramování na jednoduchém CPU



Zdroj: Mark Smotherman - A Brief History of Microprogramming



# Architektura (akumulátorová)

## Registry

- ACC (akumulátor)

## Instrukční sada

- instrukce 8 bitů (2 bity operační kód, 6 bitů adresa)
- opcode=00: **load** (*ACC from memory*)  
 $ACC \leq \text{memory}[\text{address}]$
- opcode=01: **add** (*to ACC from memory*)  
 $ACC \leq ACC + \text{memory}[\text{address}]$
- opcode=10: **store** (*ACC to memory*)  
 $\text{memory}[\text{address}] \leq ACC$
- opcode=11: **branch** (*to address if ACC zero*)  
if ( $ACC == 0$ )  $PC \leq \text{address}$





# Řídící signály

**pcincr** :  $PC \leq PC + 1$

**PC\_in** :  $PC \leq \text{CPU internal bus}$

**PC\_out** :  $\text{CPU internal bus} \leq PC$

**IR\_in** :  $IR \leq \text{CPU internal bus}$

**IR\_out** :  $\text{CPU internal bus} \leq \text{address portion of IR}$

**MAR\_in** :  $MAR \leq \text{CPU internal bus}$

**MDR\_in** :  $MDR \leq \text{CPU internal bus}$

**MDR\_out** :  $\text{CPU internal bus} \leq MDR$

**read** :  $MDR \leq \text{memory}[MAR]$

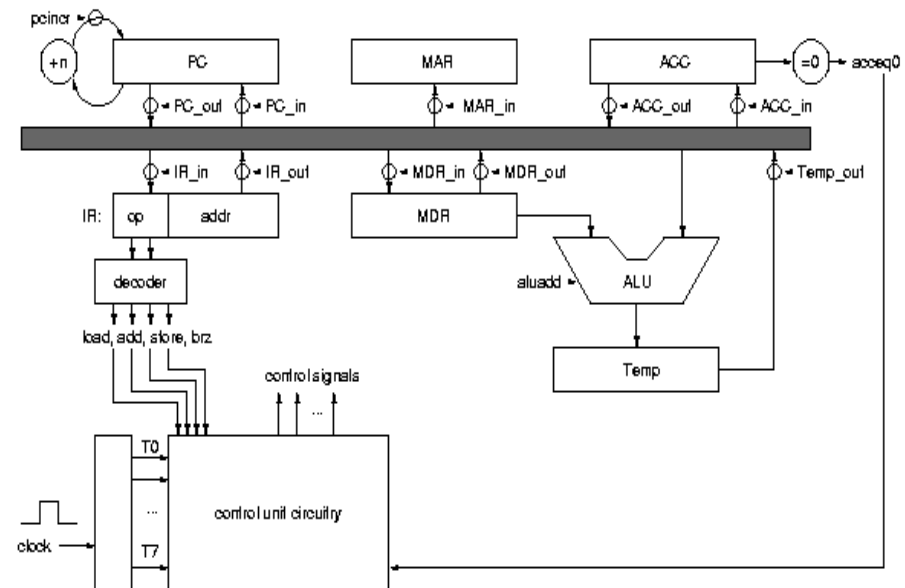
**write** :  $\text{memory}[MAR] \leq MDR$

**ACC\_in** :  $ACC \leq \text{CPU internal bus}$

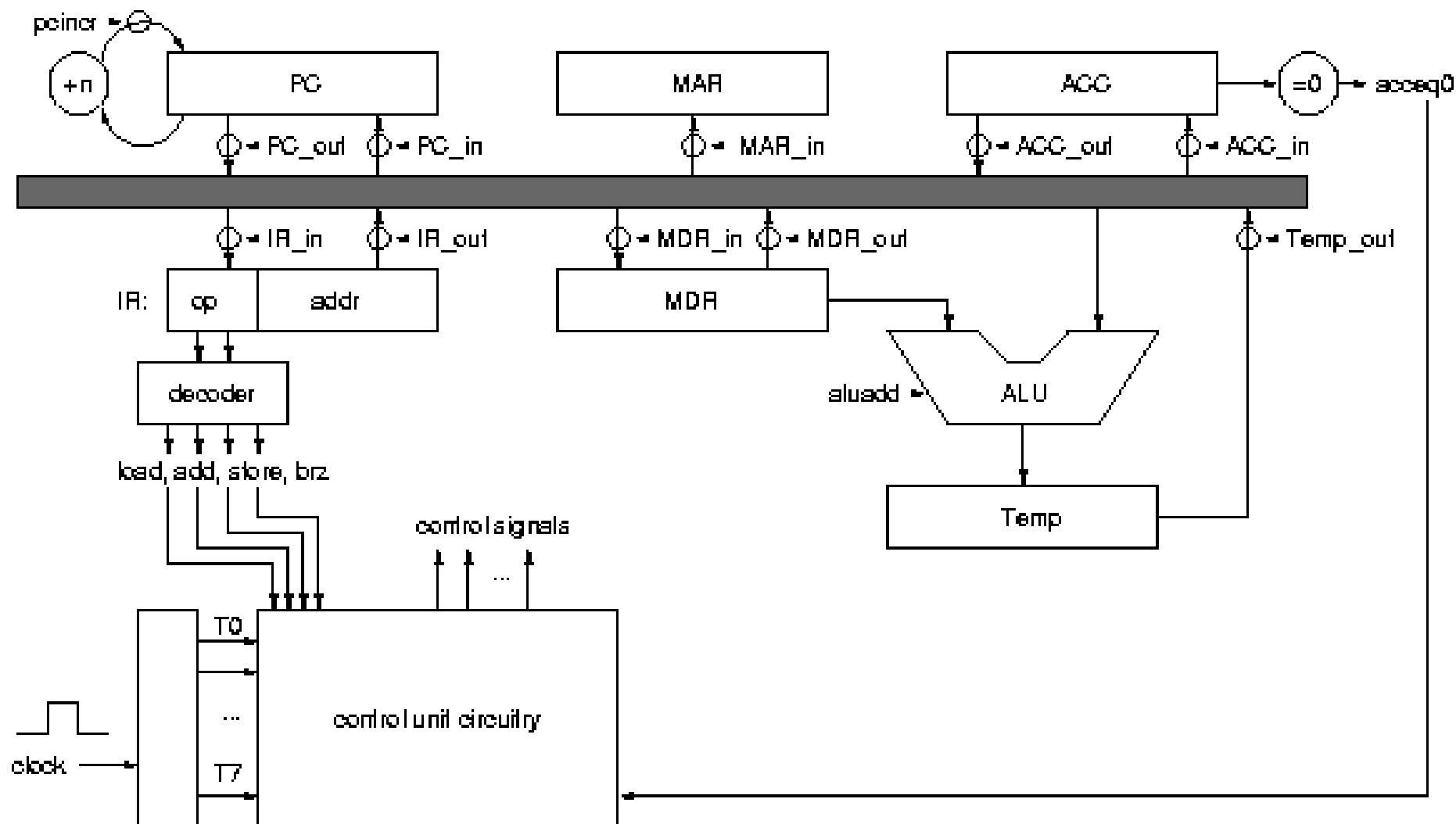
**ACC\_out** :  $\text{CPU internal bus} \leq ACC$

**TEMP\_out** :  $\text{CPU internal bus} \leq TEMP$

**aluadd** : addition is selected as the ALU operation



# Mikroprogram pro instrukci load



# Mikroprogram pro instrukci load

---

- fetch

T0: PC\_out, MAR\_in

T1: read, pcincr

T2: MDR\_out, IR\_in

- decode

T3: decode opcode in IR

- execute

T4: IR\_out(addr part), MAR\_in

T5: read

T6: MDR\_out, ACC\_in, reset to T0



# Realizace řídicí logiky: logické funkce

$ACC\_in = (load \ \& \ T6) + (add \ \& \ T7)$

$ACC\_out = (store \ \& \ T5) + (add \ \& \ T6)$

$aluadd = add \ \& \ T6$

$TEMP\_out = add \ \& \ T7$

$IR\_in = T2$

$IR\_out = (load \ \& \ T4) + (add \ \& \ T4) + (store \ \& \ T4) \\ + (brz \ \& \ acceq0 \ \& \ T4)$

$MAR\_in = T0 + (load \ \& \ T4) + (add \ \& \ T4) + (store \ \& \ T4)$

$MDR\_in = store \ \& \ T5$

$MDR\_out = T2 + (load \ \& \ T6)$

$PC\_in = brz \ \& \ acceq0 \ \& \ T4$

$PC\_out = T0$

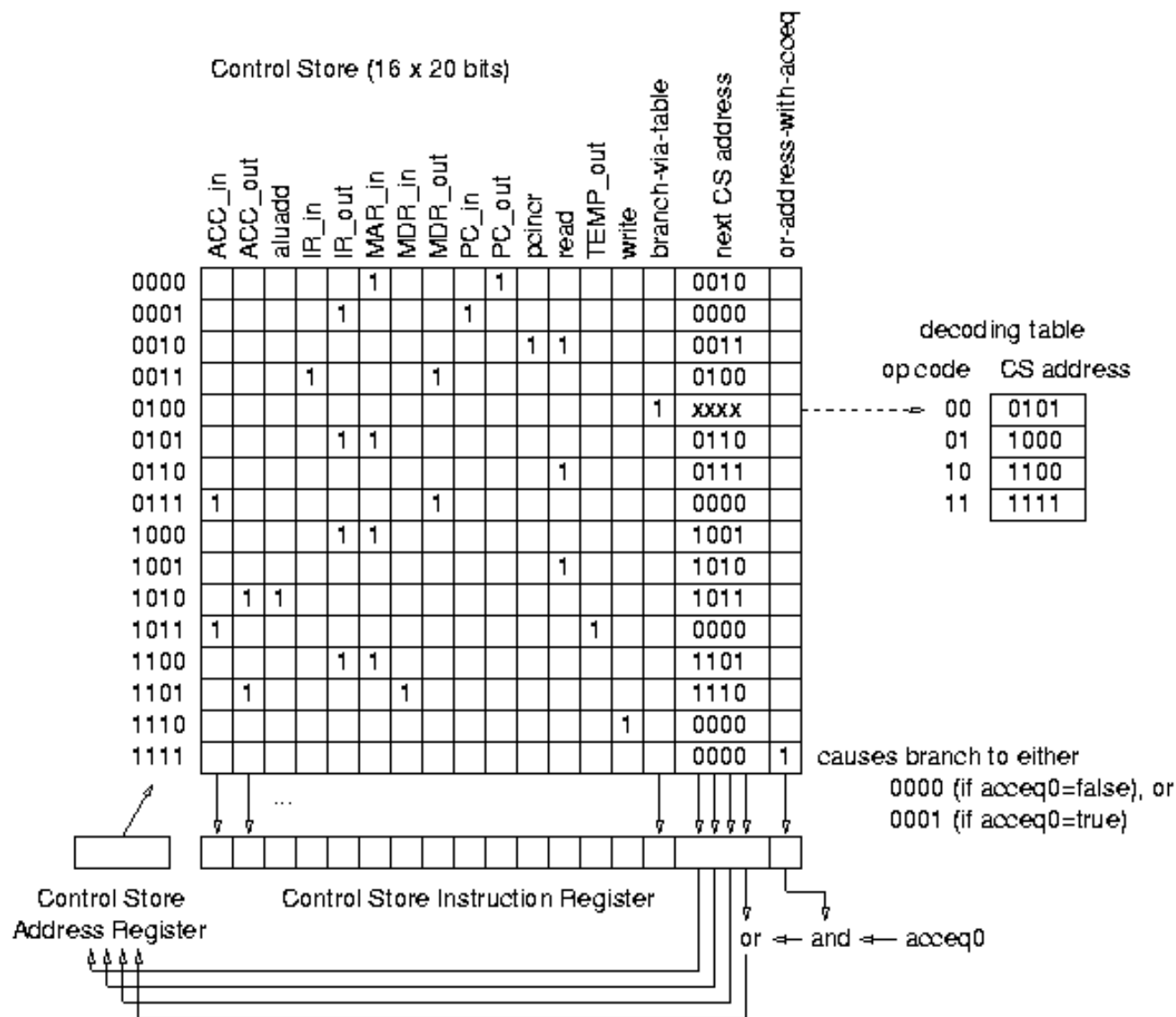
$pcincr = T1$

$read = T1 + (load \ \& \ T5) + (add \ \& \ T5)$

$write = store \ \& \ T6$



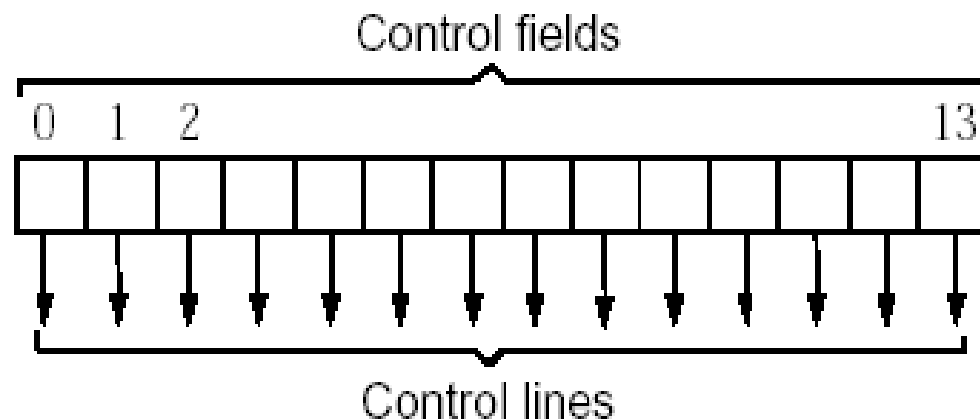
# Realizace řídicí logiky: mikrořadič s řídicí pamětí



# Horizontální formát mikroinstrukcí

## Přímá reprezentace řídicích signálů

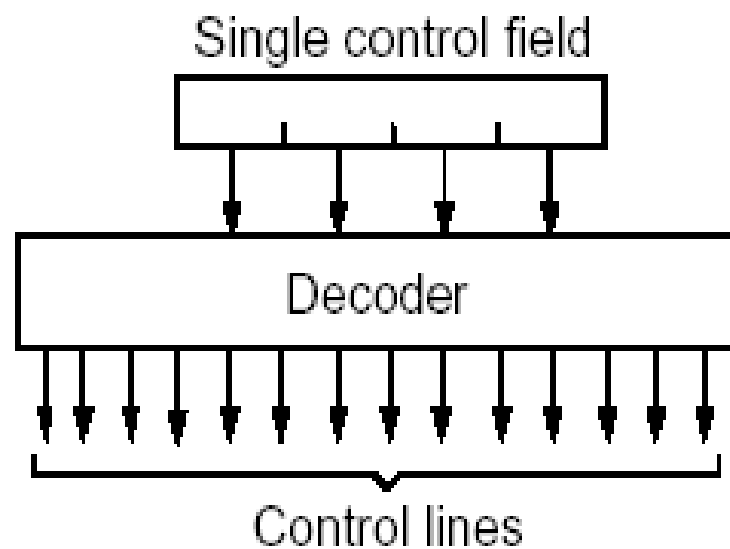
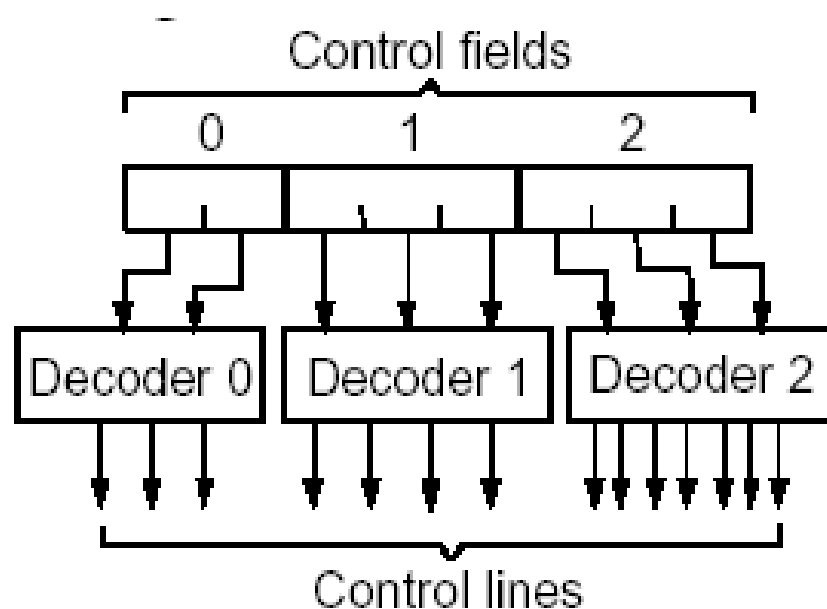
- mikroinstrukce obsahuje přímo hodnoty řídicích signálů
- není třeba dekodovat → rychlost
- libovolná kombinace → pružnost
- velké nároky na prostor



# Vertikální formát mikroinstrukcí

## Kódovaná reprezentace řídicích signálů

- některé kombinace se vylučují navzájem → možno zakódovat → menší objem
- nutno dekodovat → zpomalení, zesložitění
- pevný návrh → méně pružné



# Nanoprogramování

---

## Základní myšlenka

- jen některé kombinace řídicích signálů dávají smysl
- smysluplné kombinace očíslováme a uložíme do paměti

## Dvojúrovňová řídicí paměť

- první úroveň použije vertikální formát (tj. čísla smysluplných kombinací) pro indexaci druhé úrovně
- vybraný řádek ve druhé úrovni je již v horizontálním formátu (tj. bity odpovídají řídicím signálům)

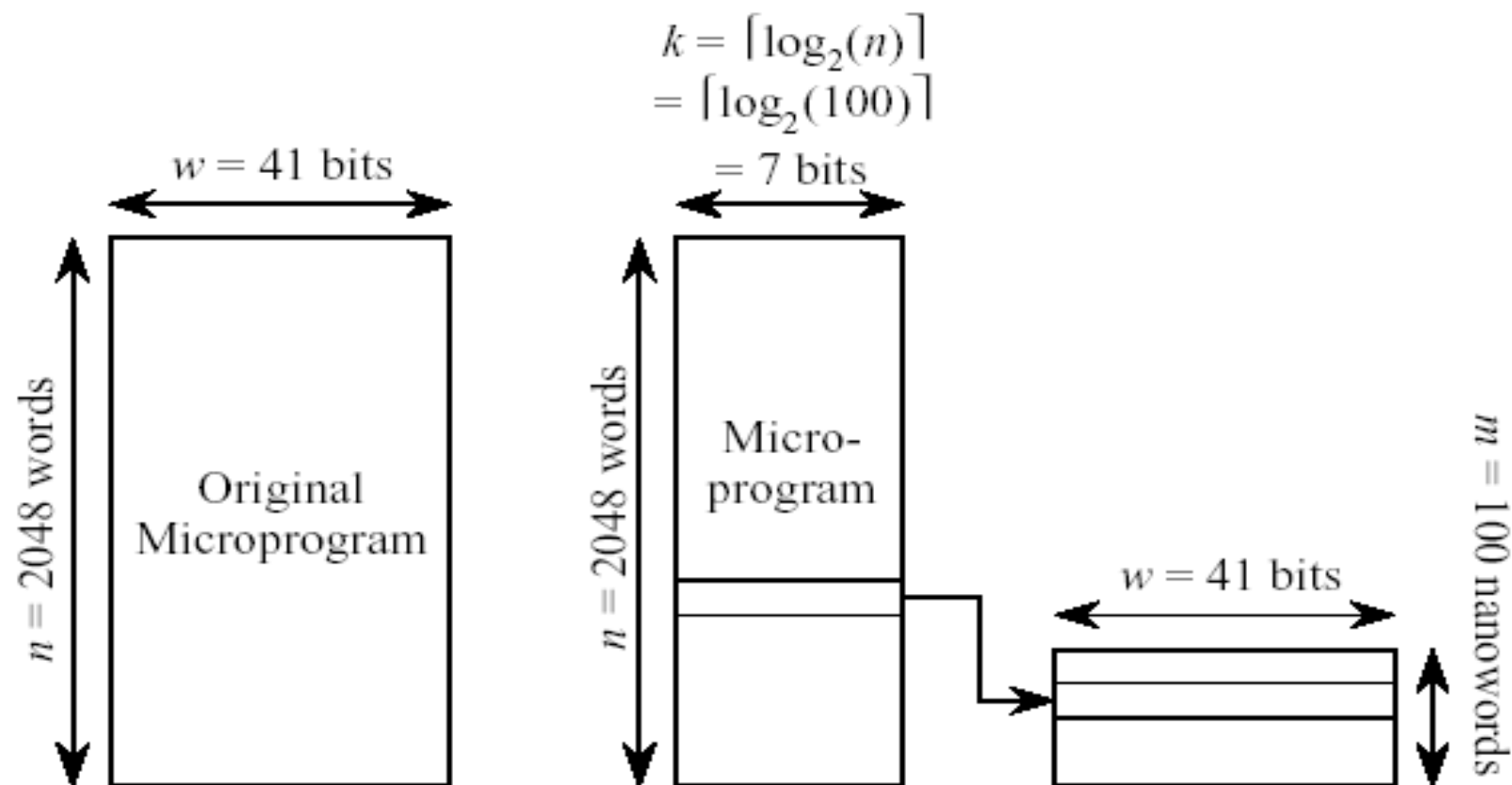
## Důsledky

- silná redukce velikosti
- zpomalení





# Srovnání mikro- a nanoprogramování



Total Area =  $n \times w =$   
 $2048 \times 41 = 83,968$  bits

Microprogram Area =  $n \times k = 2048 \times 7$   
 $= 14,336$  bits

Nanoprogram Area =  $m \times w = 100 \times 41$   
 $= 4100$  bits

Total Area =  $14,336 + 4100 = 18,436$  bits

