Principy počítačů a operačních systémů

Architektura a implementace zjednodušeného procesoru MIPS

Zimní semestr 2011/2012

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 = Mem [R1 + immediate], Mem [R1 + 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)

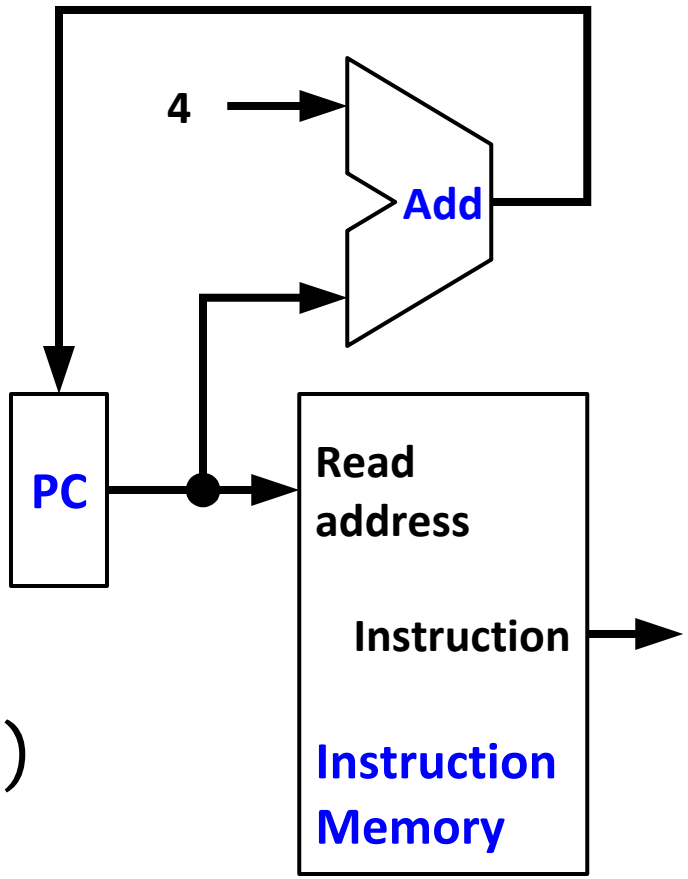
Jednocyklová datová cesta

Uspořádání logických celků implementující operace procesoru

Čtení instrukce (fetch)

Registr PC

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



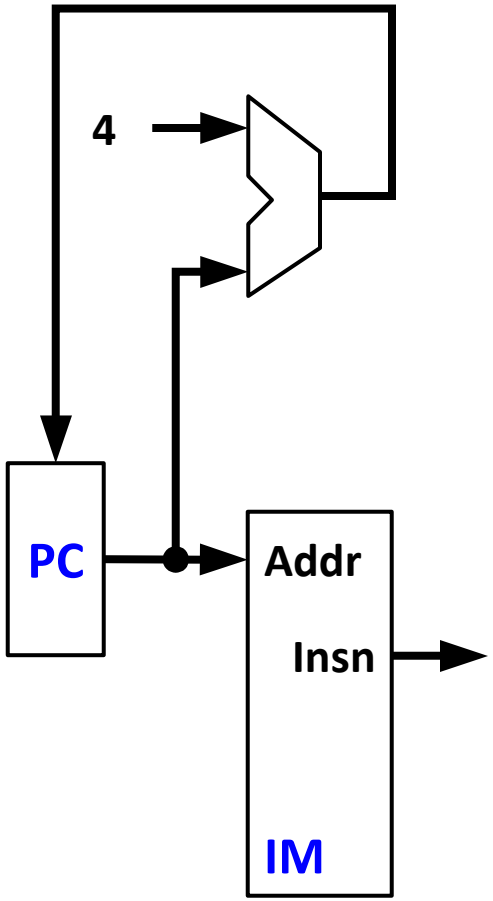
Paměť instrukcí

- oddělená paměť = Harvardská architektura (prozatím)

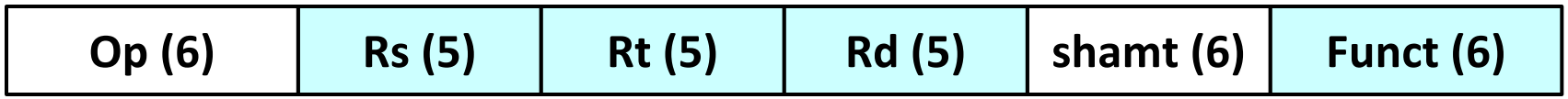
Sčítačka

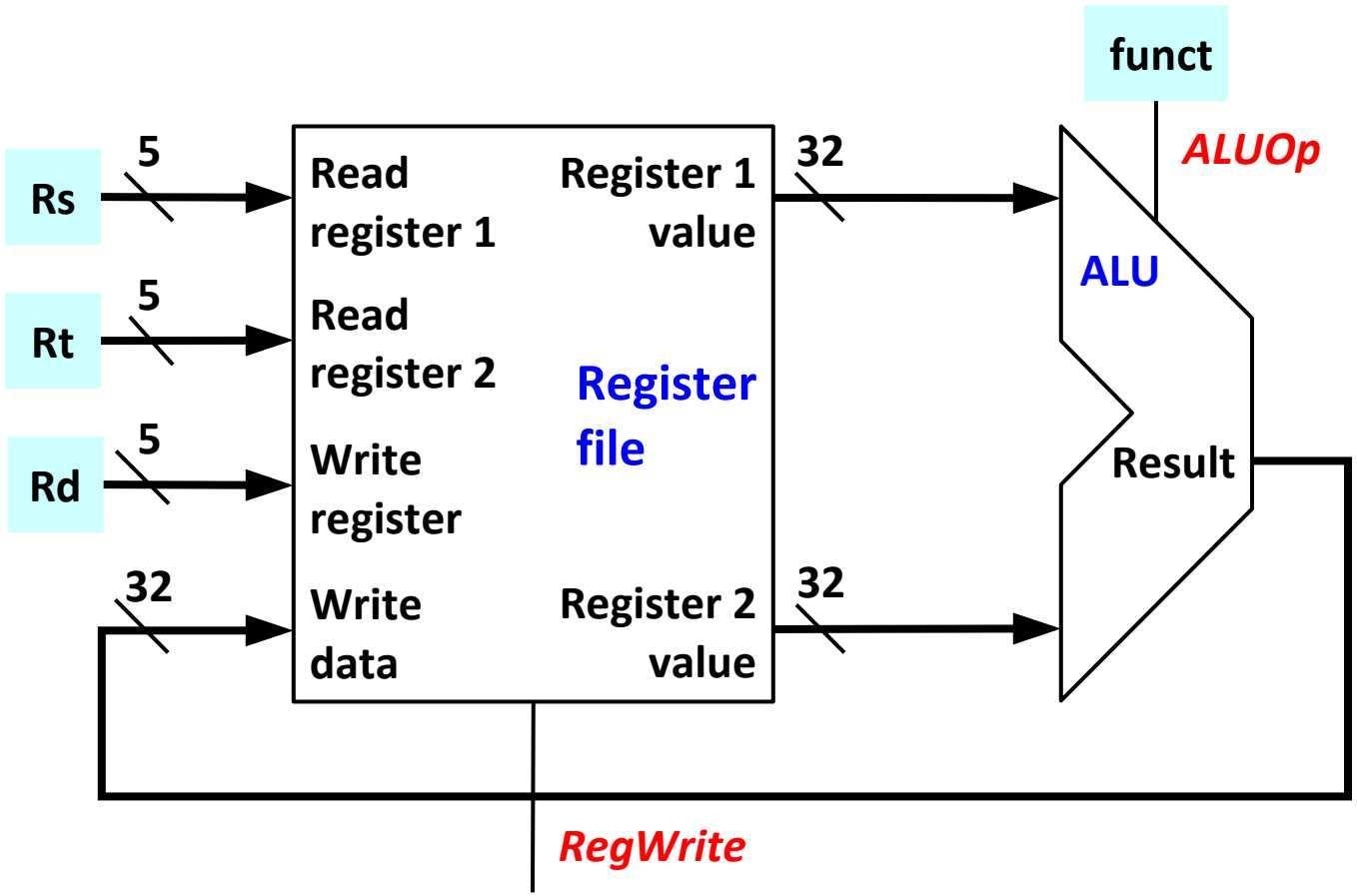
- inkrementuje obsah PC o 4 ⇒ implicitní posun na následující instrukci (von Neumannovská koncepce)

Podpora čtení instrukce

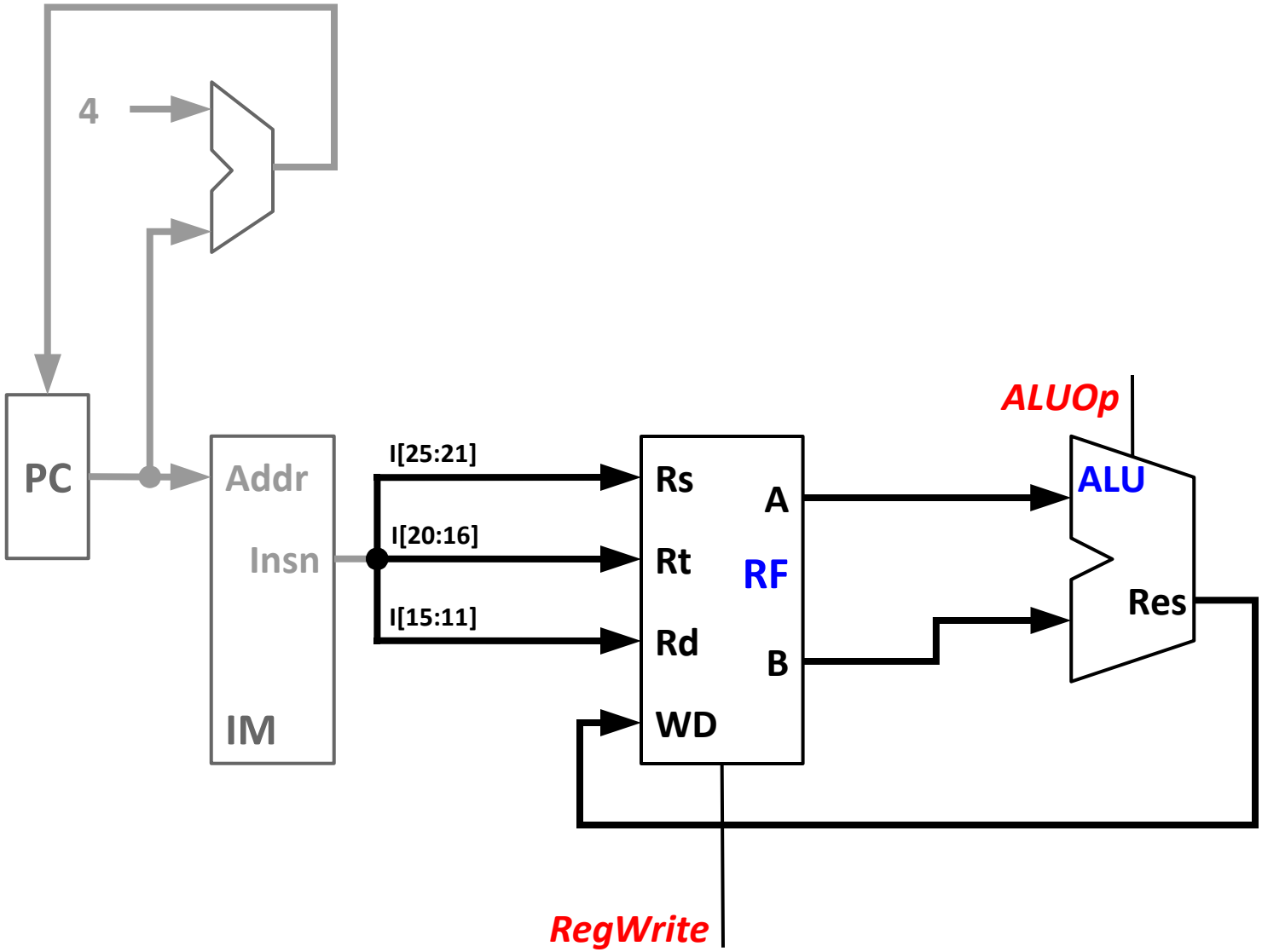


Registrové operace (add, sub, ...)

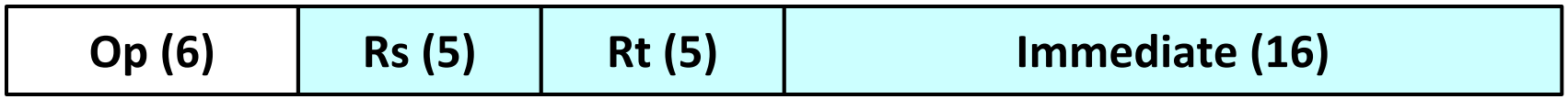


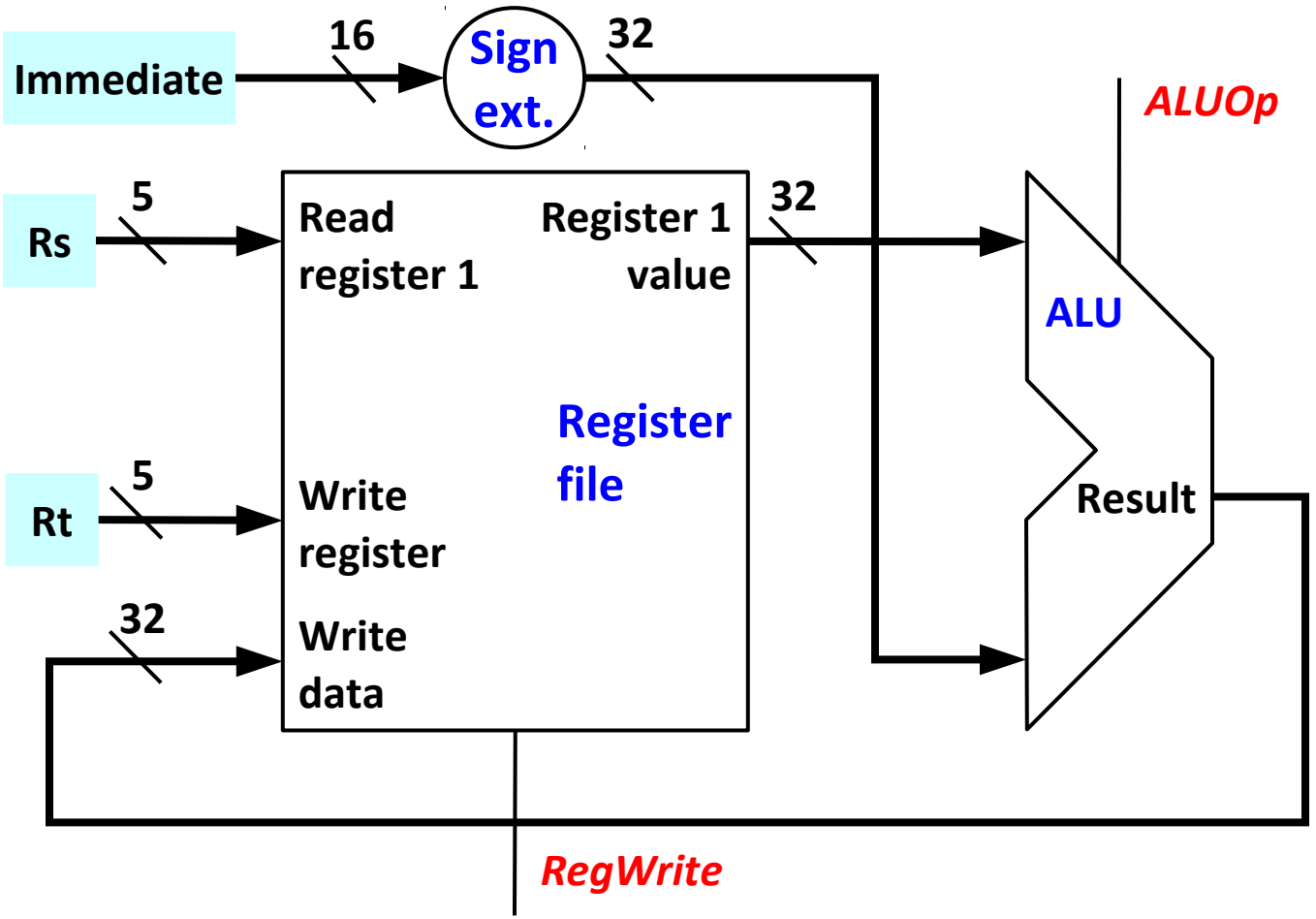


Podpora registrových operací



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

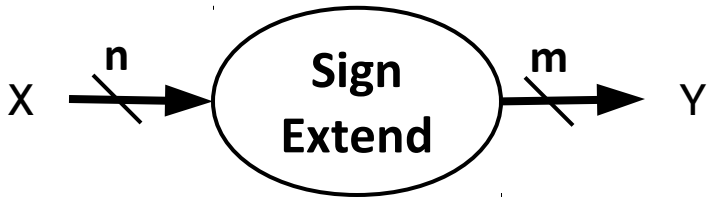




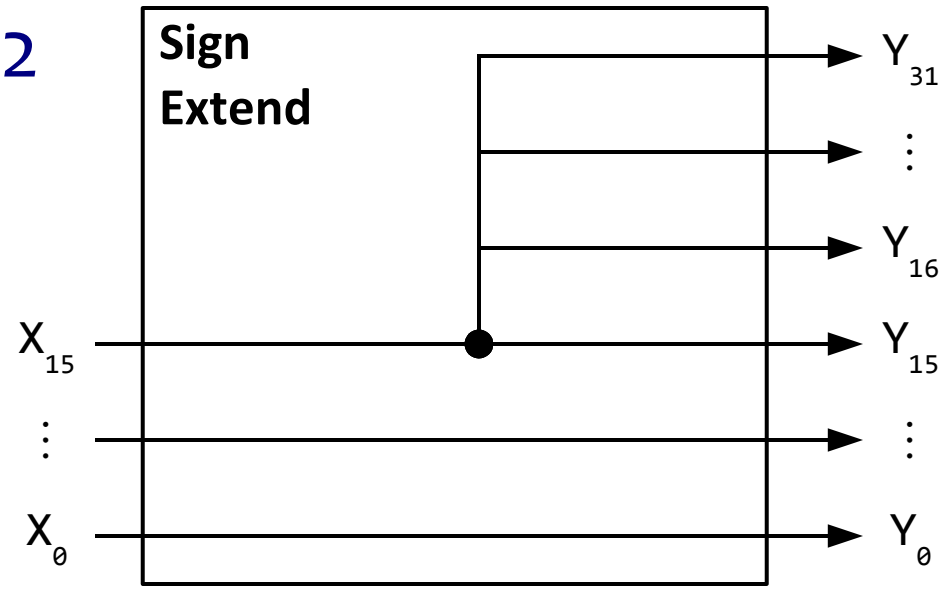
Znaménkové rozšíření (*sign extension*)

Logický prvek

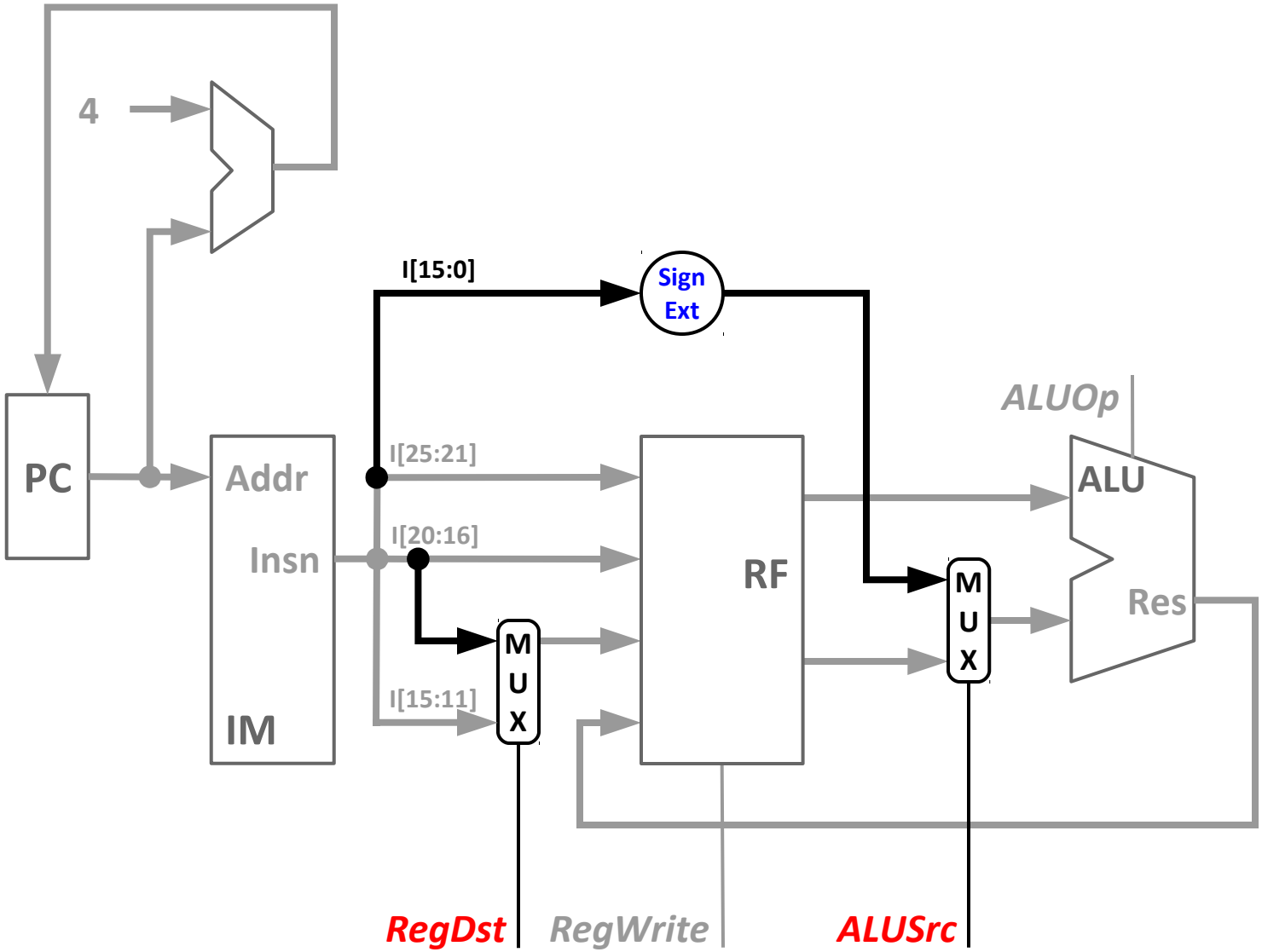
- vstup: ***n***-bitové číslo ***X***
  - předpokládá se reprezentace ve dvojkové doplňku
- výstup: ***m***-bitové číslo ***Y***
  - znaménkové rozšíření čísla ***X***



Příklad pro n=16, m=32



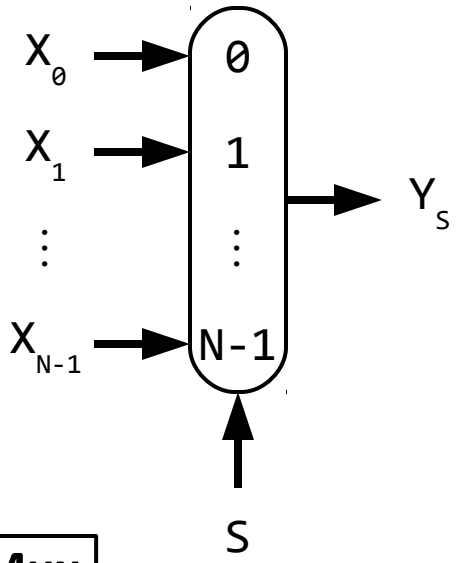
Podpora přímých operandů



Přepínač vstupů (*multiplexer, mux*)

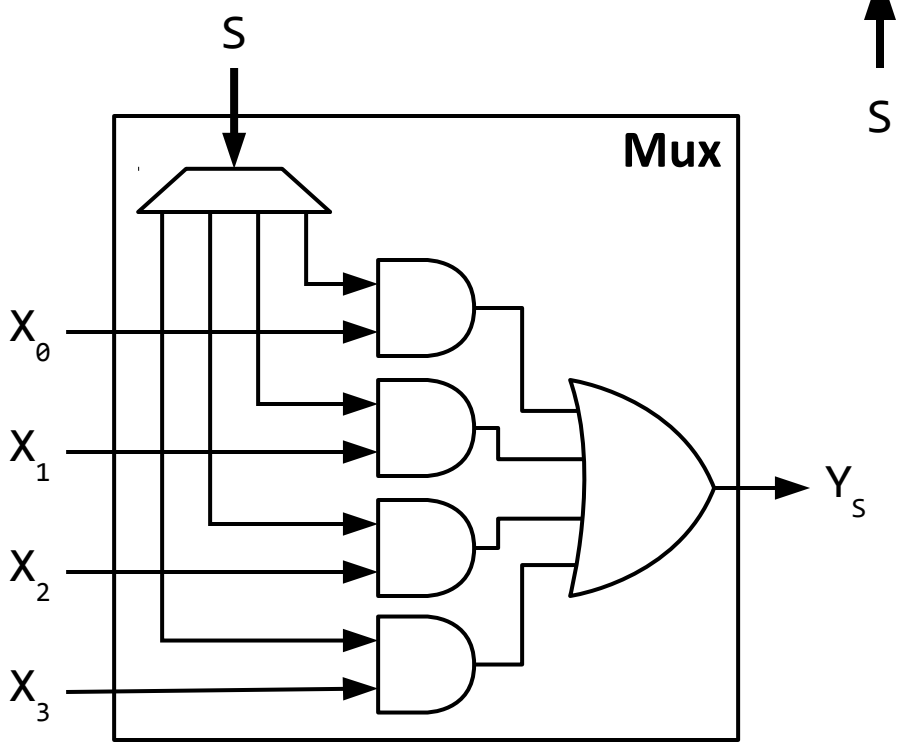
Logický prvek

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



Příklad pro N=4, m=1

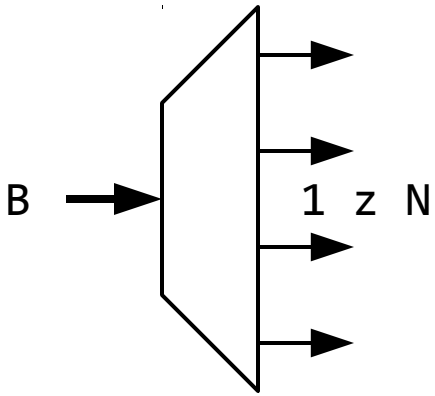
- pro výběr vstupu se používá dekodér do kódu ***"1 z N"***



Binární dekodér do "1 z N" (*binary to 1-hot*)

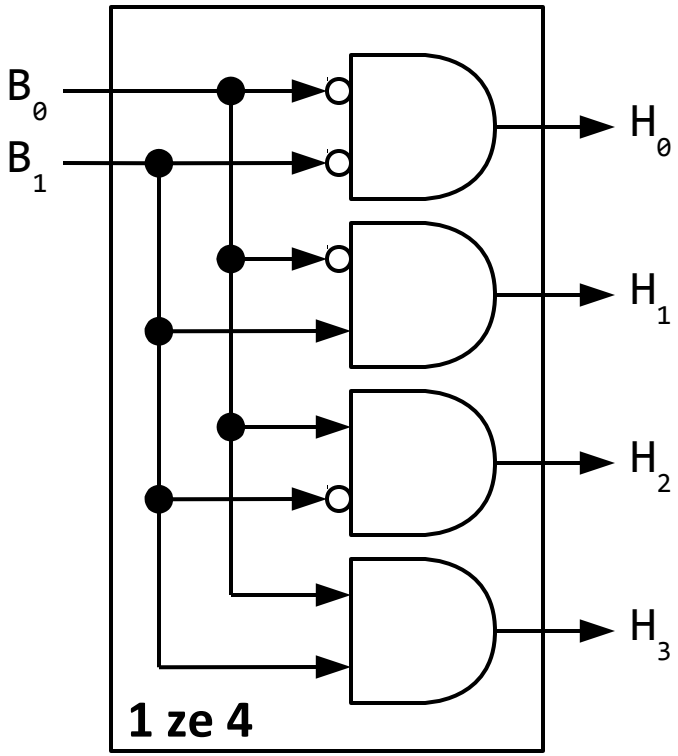
Logický prvek

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

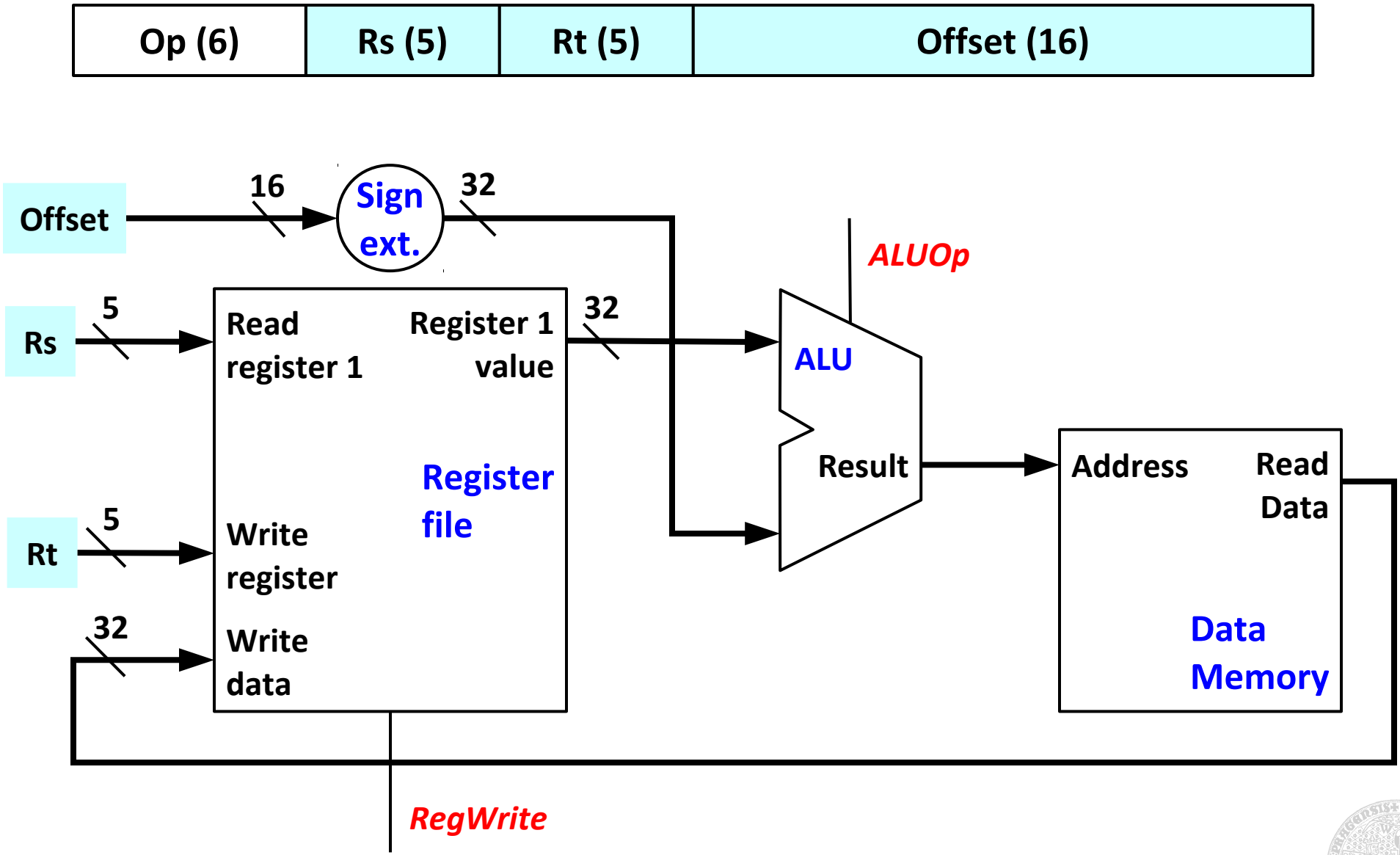


Příklad pro N=4

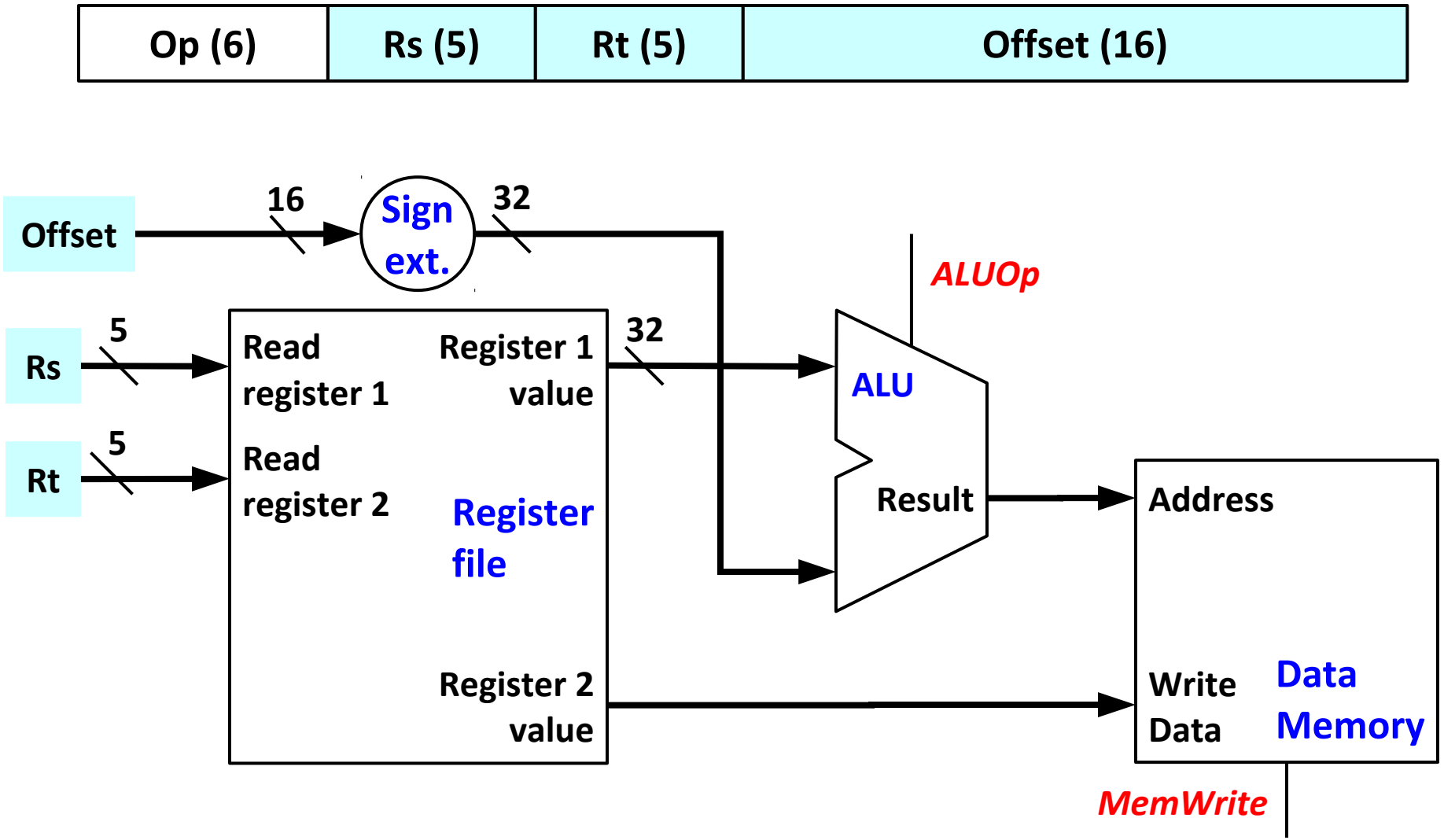
Vstupy		Výstupy			
B_1	B_0	H_3	H_2	H_1	H_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



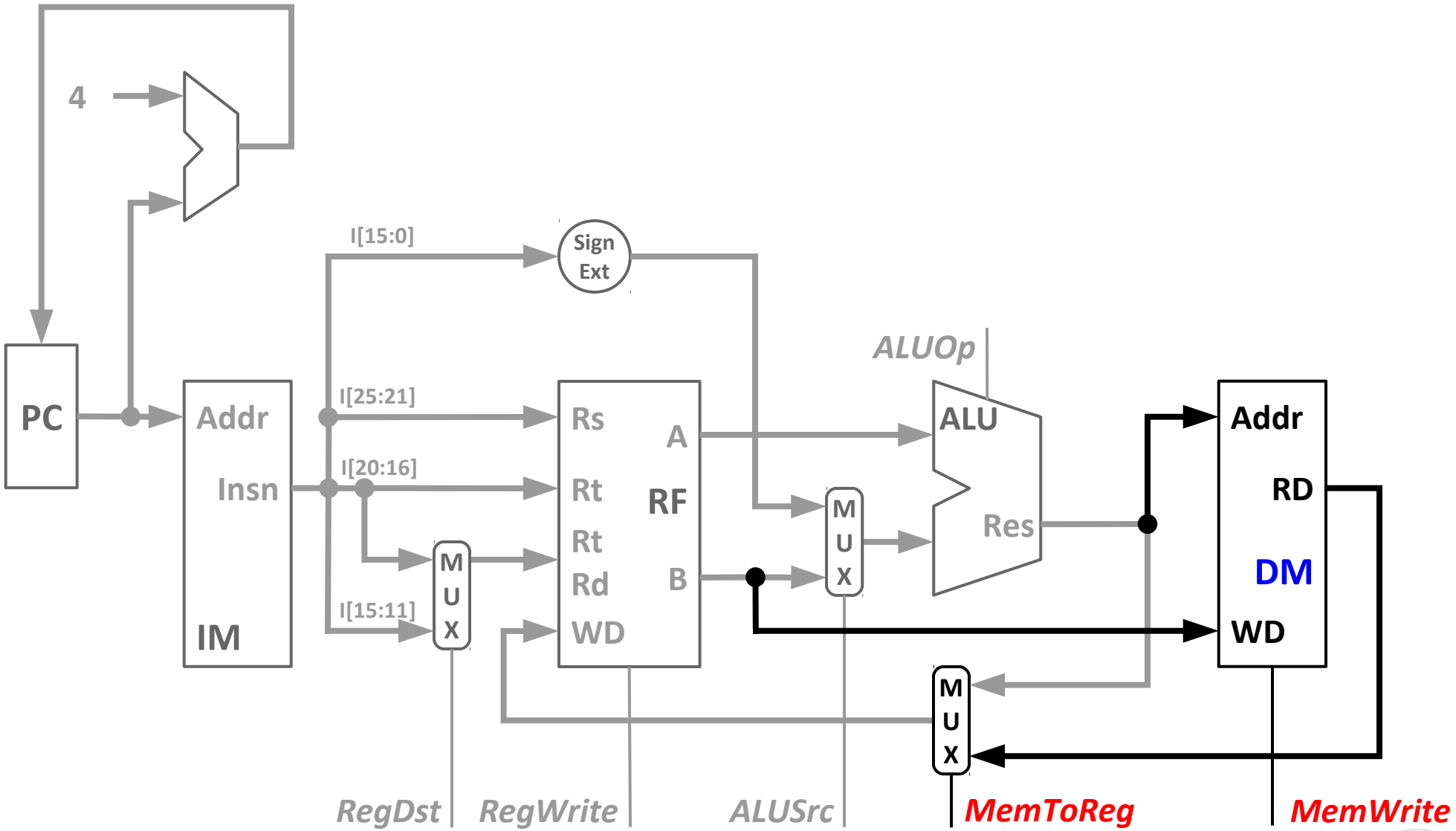
Čtení z paměti (load)



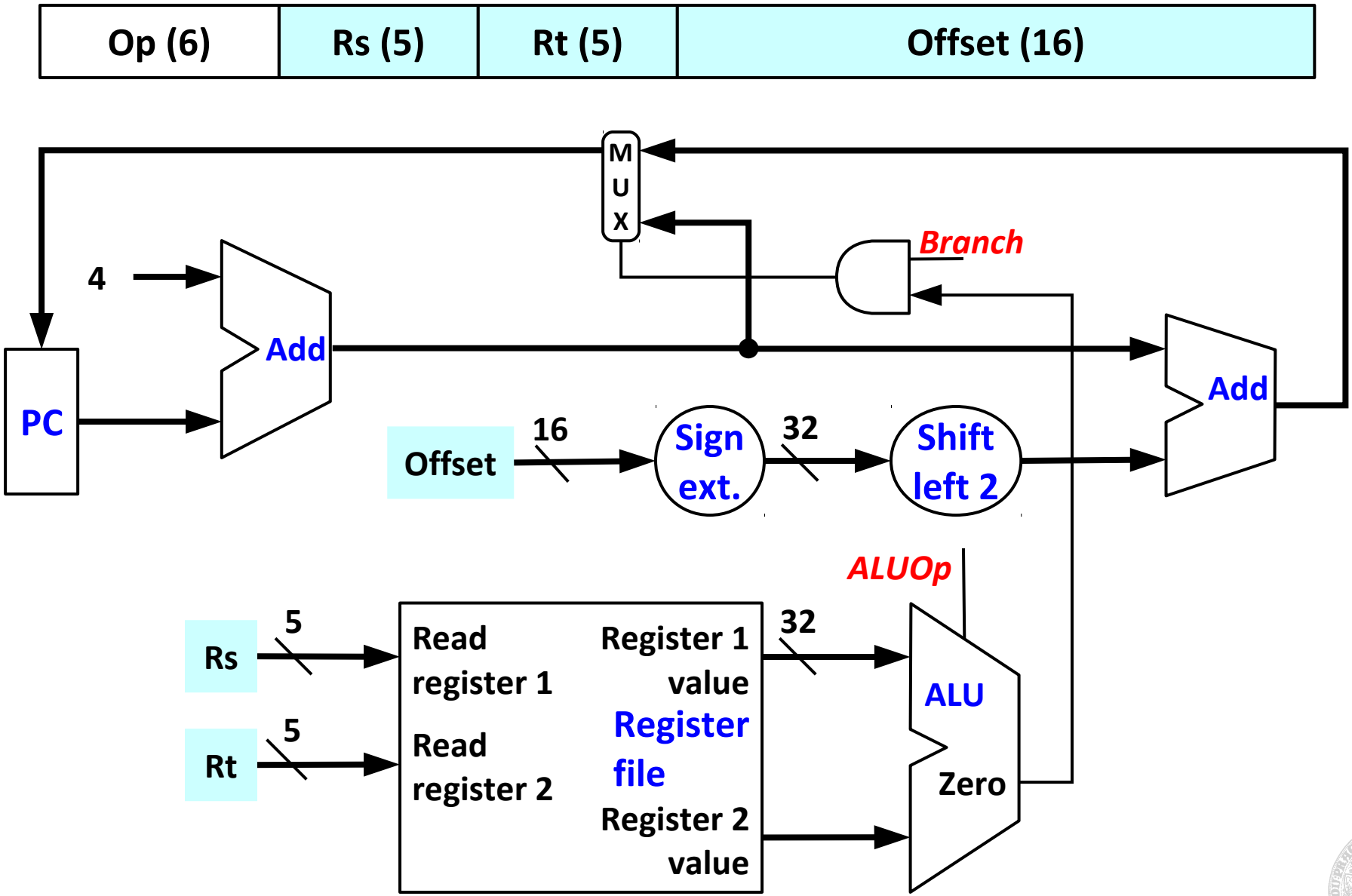
Zápis do paměti (store)



Podpora přístupu do paměti



Podmíněný skok s relativní adresou (branch)

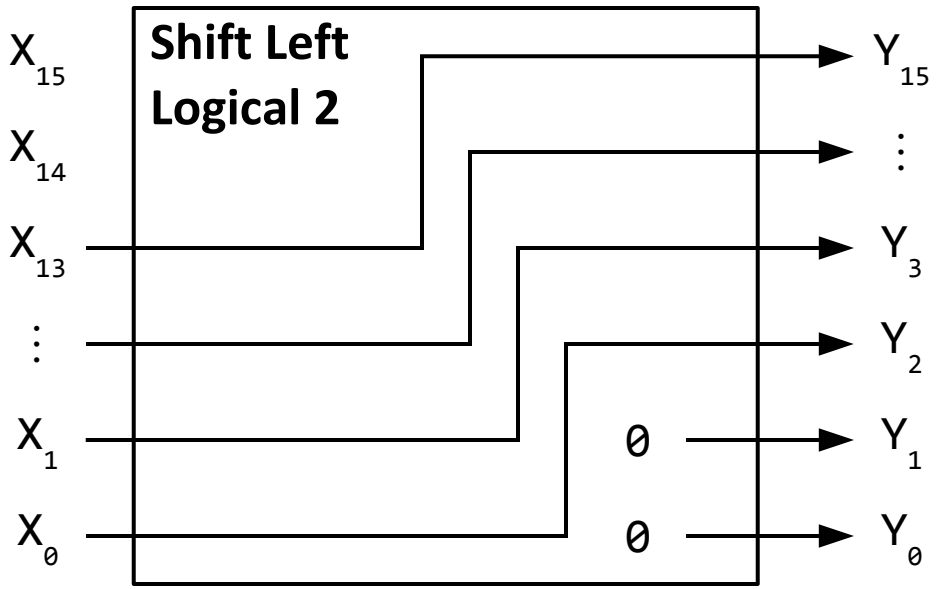


Logický posun vlevo o *k* bitů (shift logical left)

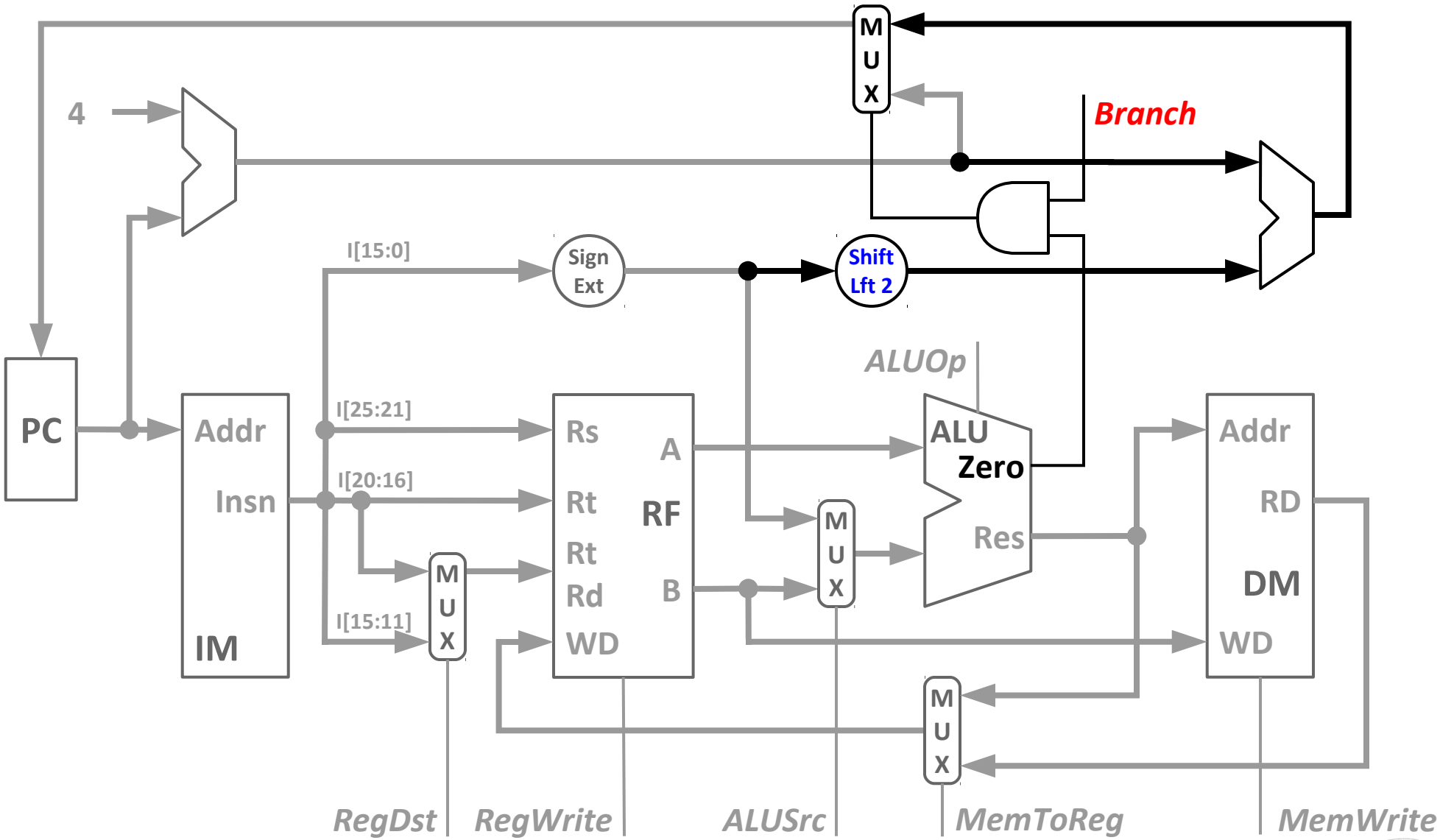
Logický prvek

- vstup: ***n***-bitové číslo ***X***
- výstup: ***n***-bitové číslo ***Y = X << k***

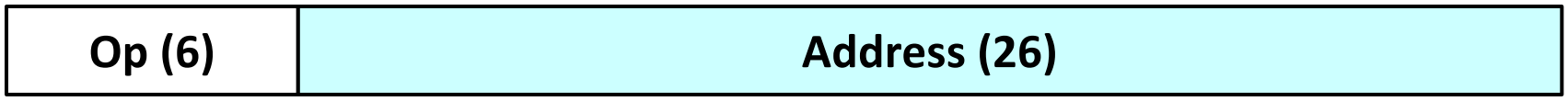
Příklad pro ***n=16, k=2***

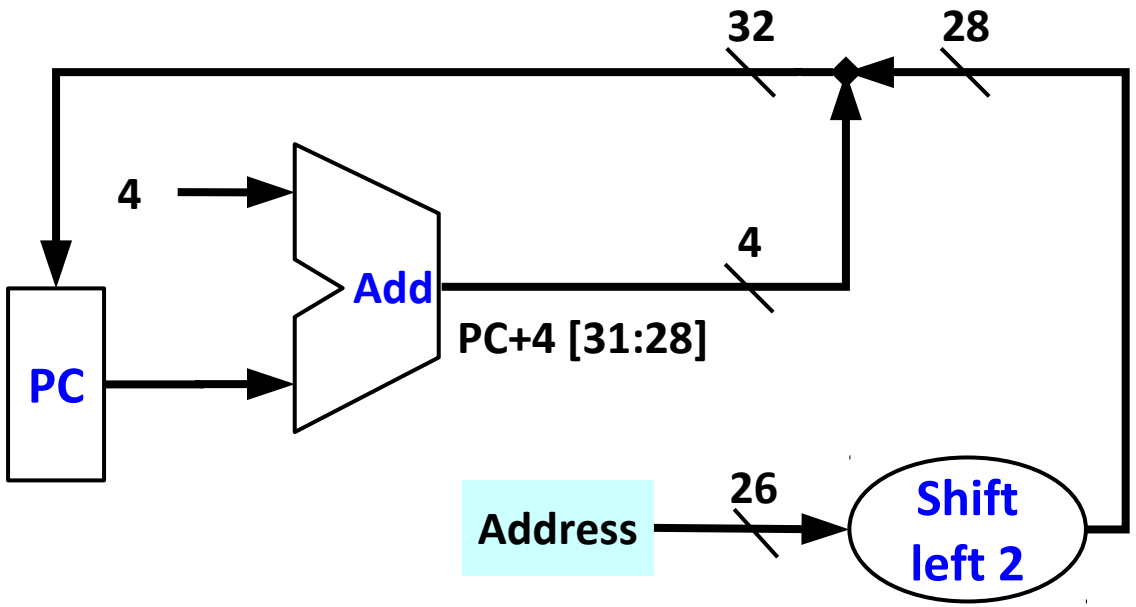


Podpora podmíněného skoku s relativní adresou

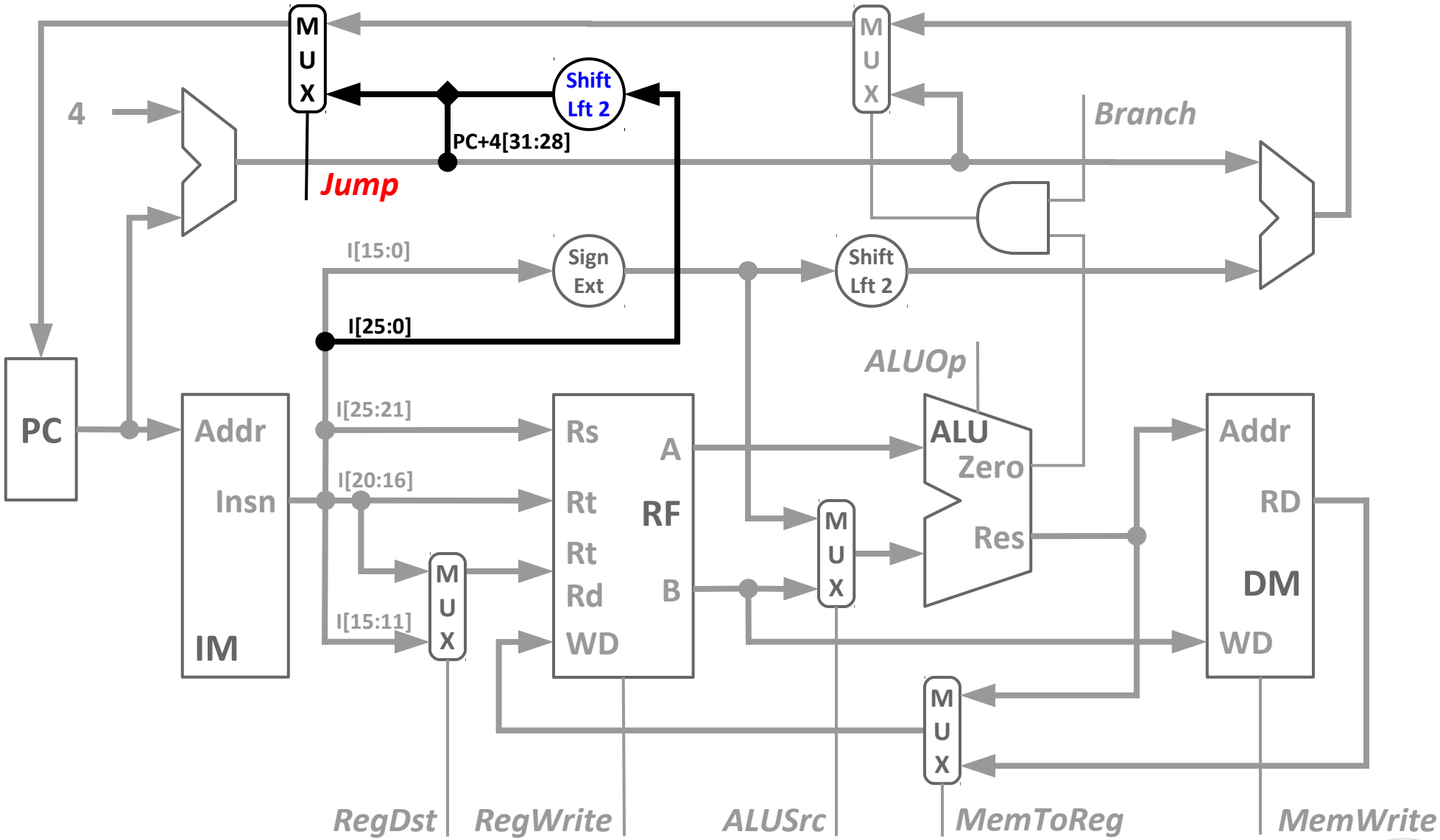


Nepodmíněný skok s absolutní adresou (jump)





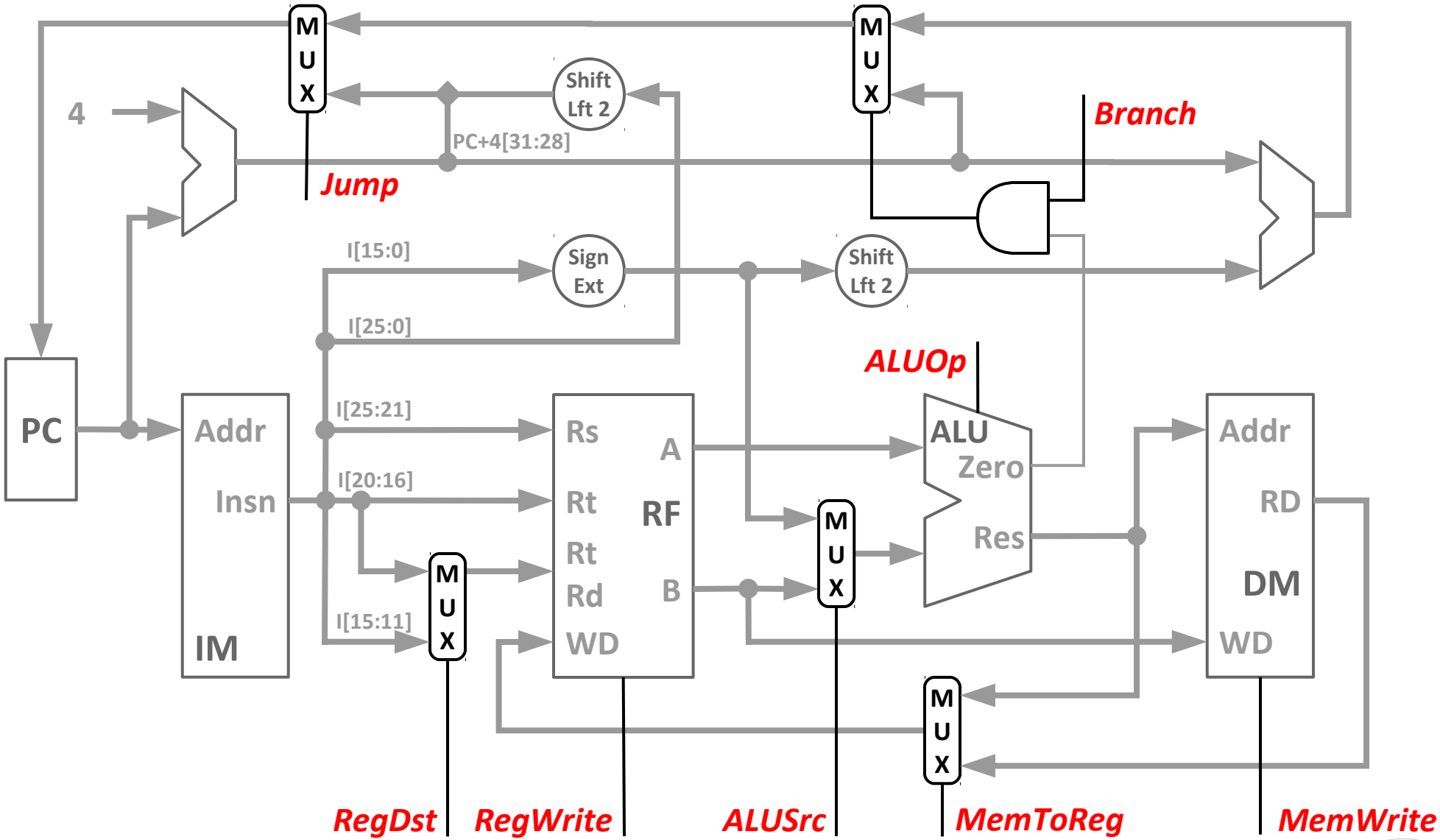
Podpora nepodmíněného skoku s absolutní adresou



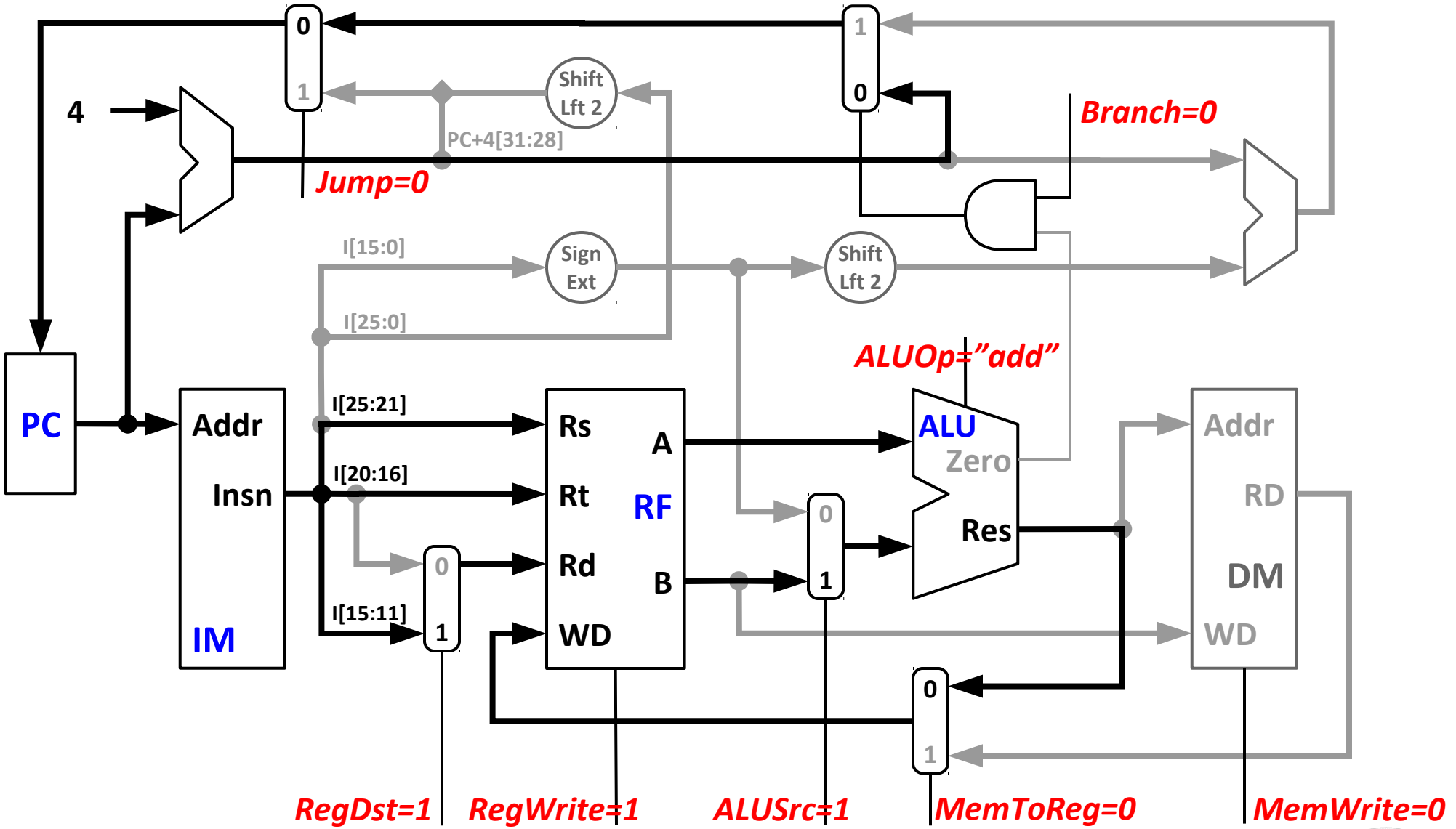
Řízení jednocyklové datové cesty

Řízení průchodu dat datovou cestou
v závislosti na typu operace

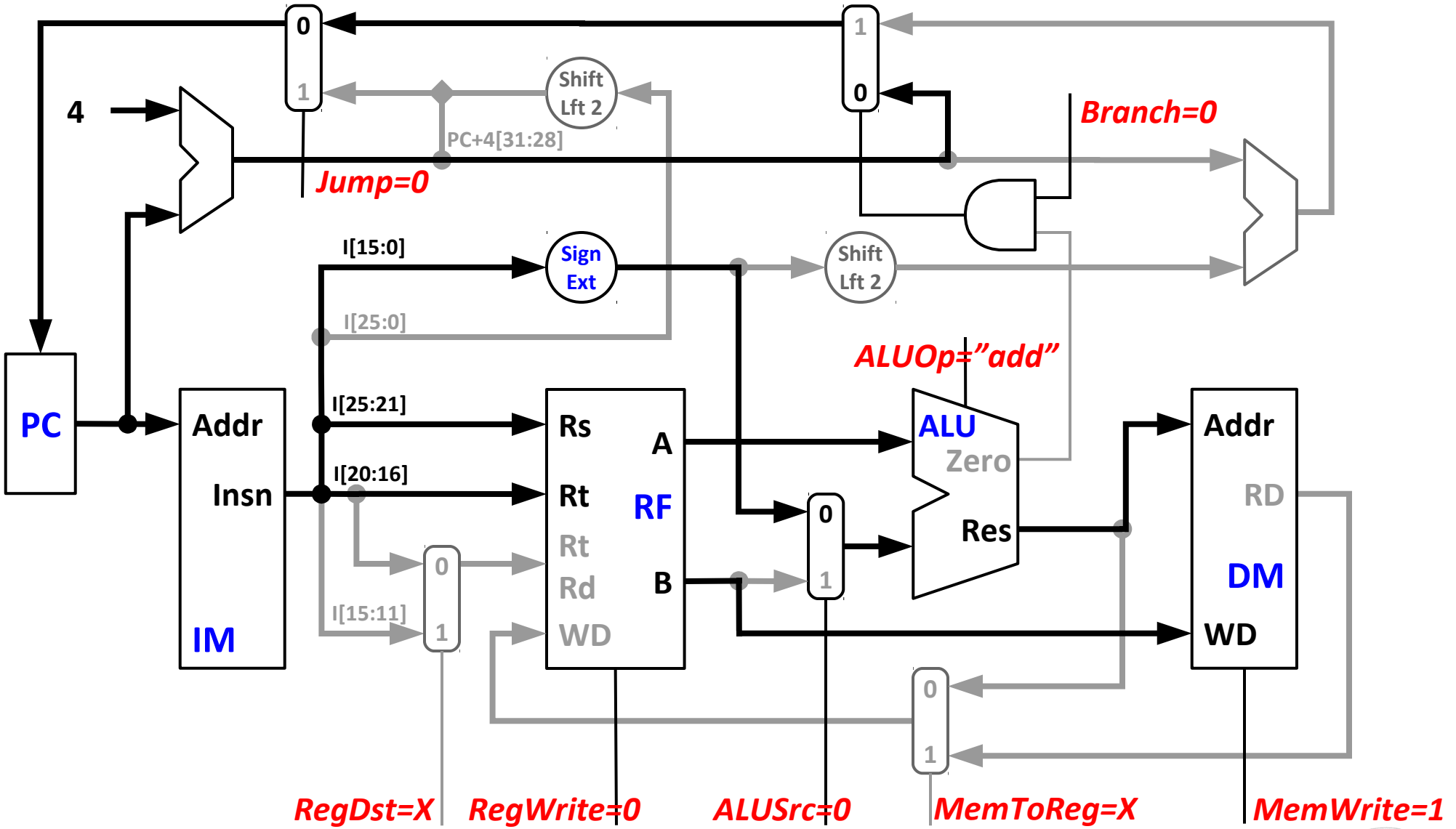
V čem spočívá řízení datové cesty?



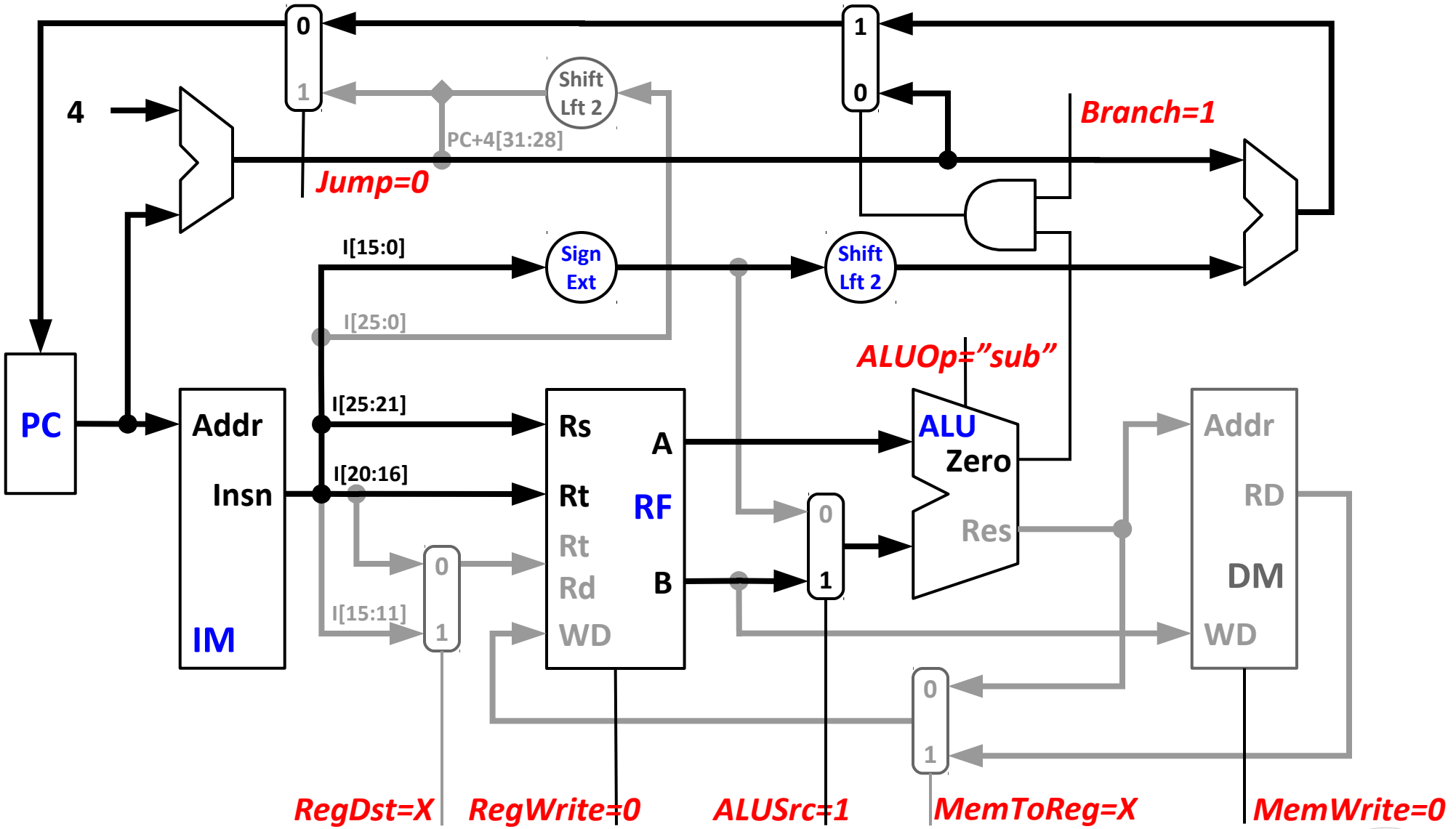
Příklad: řízení datové cesty pro instrukci “add”



Příklad: řízení datové cesty pro instrukci “sw”



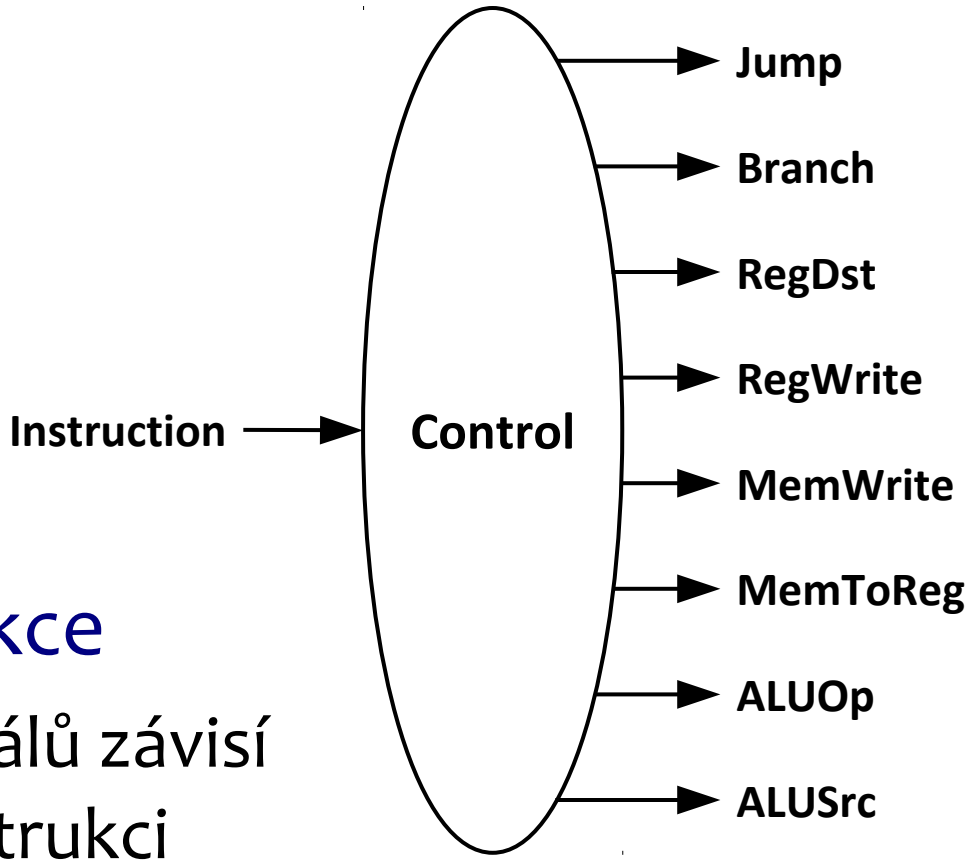
Příklad: řízení datové cesty pro instrukci “beq”



Radič datové cesty (*data path controller*)

Generuje řídící signály

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



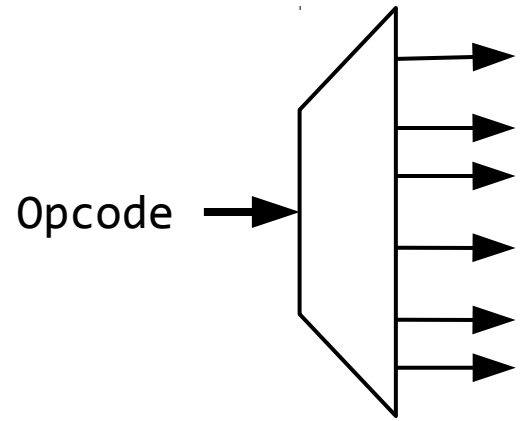
Hodnoty pro různé instrukce

- 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 řadiče

Implementace řadiče pomocí ROM

Řídící paměť

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



	Jump	Branch	RegDst	RegWrite	MemWrite	MemToReg	ALUOp	ALUSrc
add	0	0	1	1	0	0	0	1
addi	0	0	0	1	0	0	0	0
lw	0	0	0	1	0	1	0	0
sw	0	0	X	0	1	X	0	0
beq	0	1	X	0	0	X	1	1
j	1	X	X	0	0	X	X	X

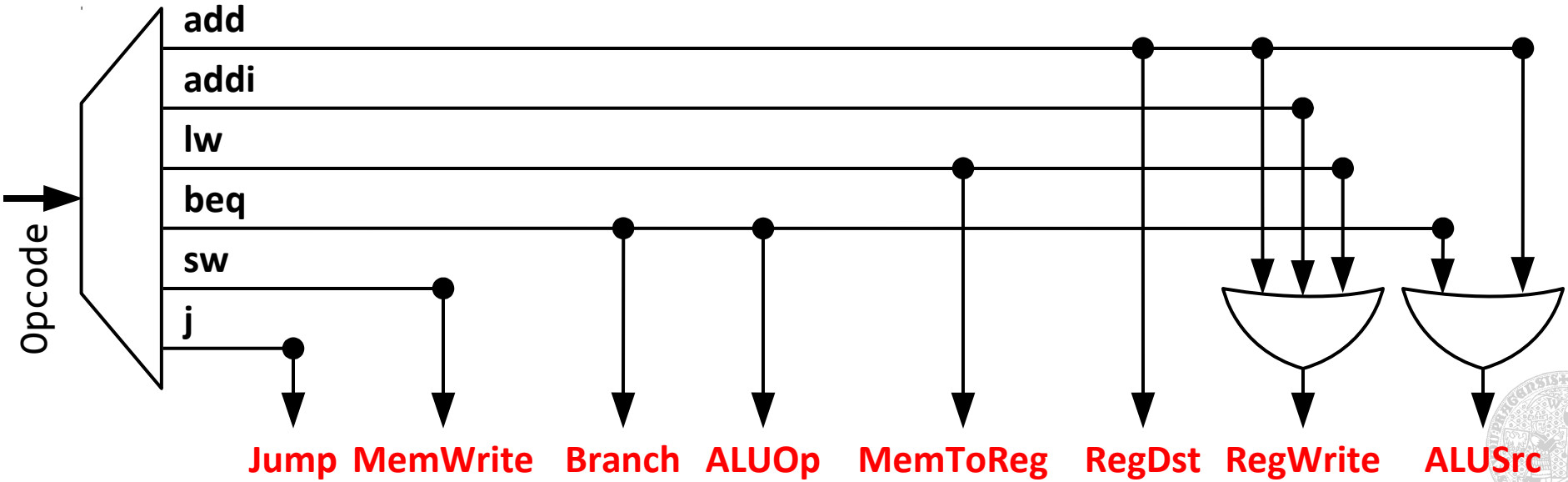
Implementace řadiče pomocí kombinačního obvodu

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

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

Alternativa k ROM: kombinační obvod

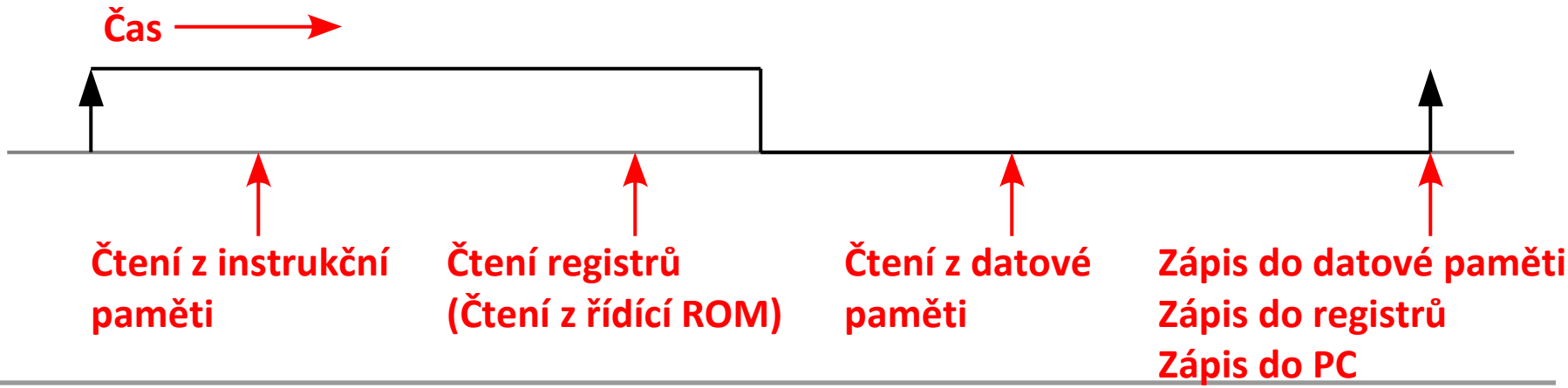
- mnoho signálů má málo jedniček nebo nul ⇒ kompaktní reprezentace log. funkcí pro řídící signály



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 řídící 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

Základní myšlenka

Proměnná doba zpracování instrukce

- jednoduché instrukce by neměly trvat stejně dlouho jako složité
- nelze měnit periodu hodinového signálu ⇒ 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. vícecyklová 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***

Vícecyklová datová cesta

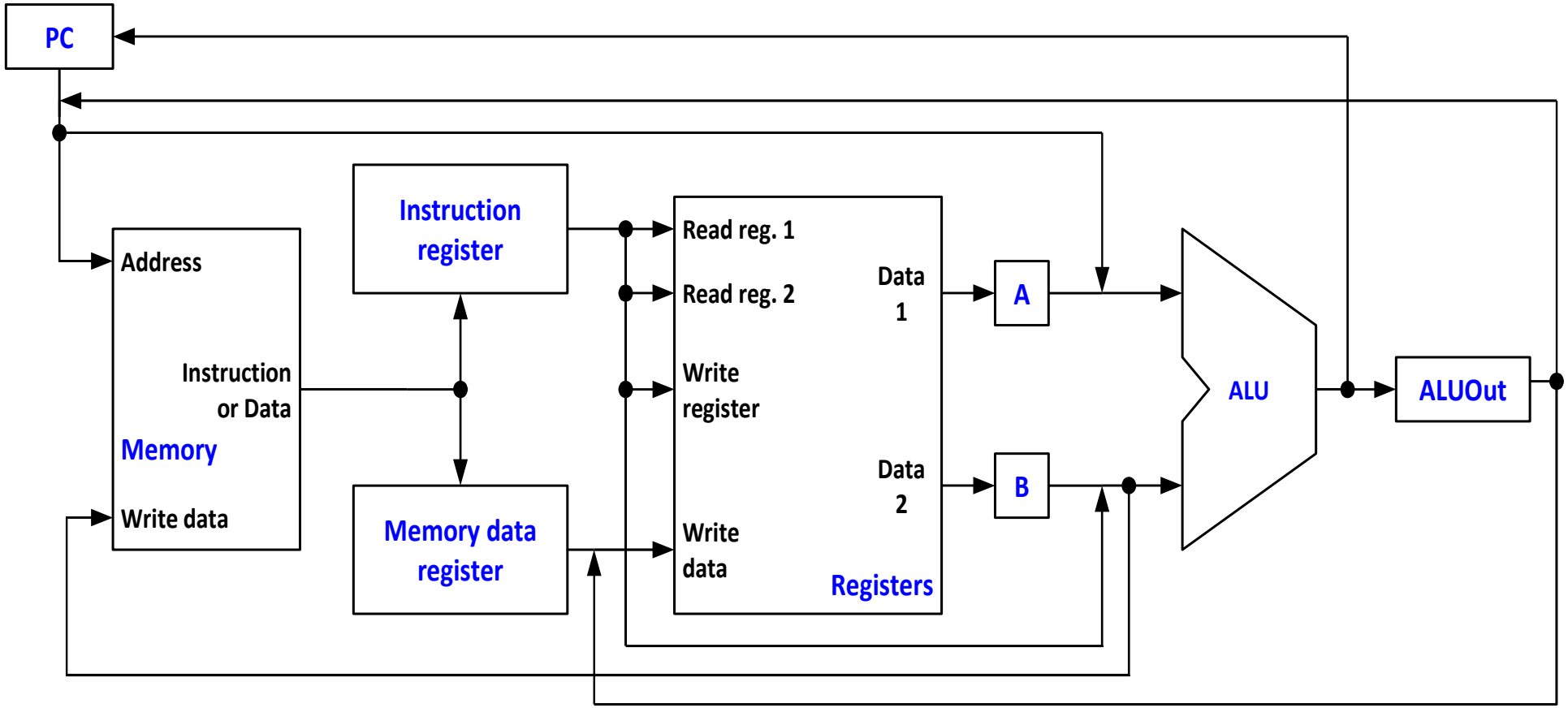
- perioda hodin 10ns, ***CPI*** $\approx (90\% \times 1) + (10\% \times 4) \approx 1.3$
- při (v průměru) 13ns na instrukci je výkon ***77 MIPS***
- vícecyklová datová cesta je 3× (o 200%) rychlejší

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 vícecyklové 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 <= Memory [PC]
  - přečtení instrukce do instrukčního registru
- PC <= 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 <= Reg [IR.rs]
  - přečtení obsahu zdrojového registru 1
- B <= Reg [IR.rt]
  - přečtení obsahu zdrojového registru 2
- ALUOut <= PC + (sign-extend (IR.addr) << 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 <= PC [31:28] + IR [25:0] << 2

Aritmeticko-logická operace

- ALUOut <= A funct B

Přístup do paměti

- ALUOut <= A + sign-extend (IR [15:0])
  - výpočet adresy pro přístup do paměti

Krok 4: přístup do paměti/zápis výsledku

Aritmeticko-logická operace (konec)

- Reg [IR.rd] <= ALUOut
  - výsledek operace zapsán do cílového registru

Zápis do paměti (konec)

- Memory [ALUOut] <= B
  - obsah registru zapsán do paměti

Čtení z paměti

- MDR <= Memory [ALUOut]
  - obsah paměti přečten do pomocného registru

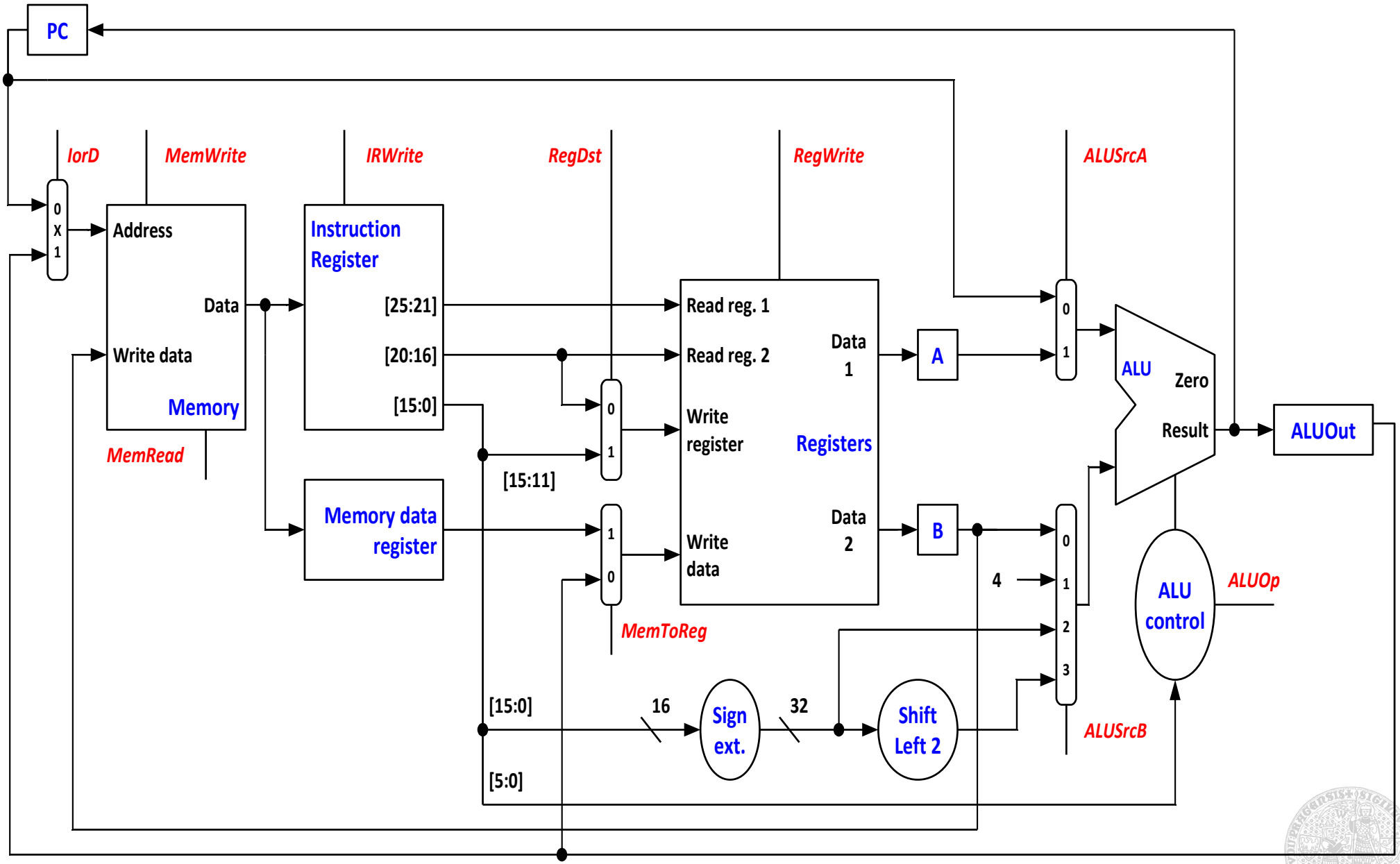
Krok 5: zápis dat z paměti do registru

Čtení z paměti (konec)

- Reg [IR.rt] <= MDR
  - zápis přečtené hodnoty do registru

Řízení vícecyklové datové cesty

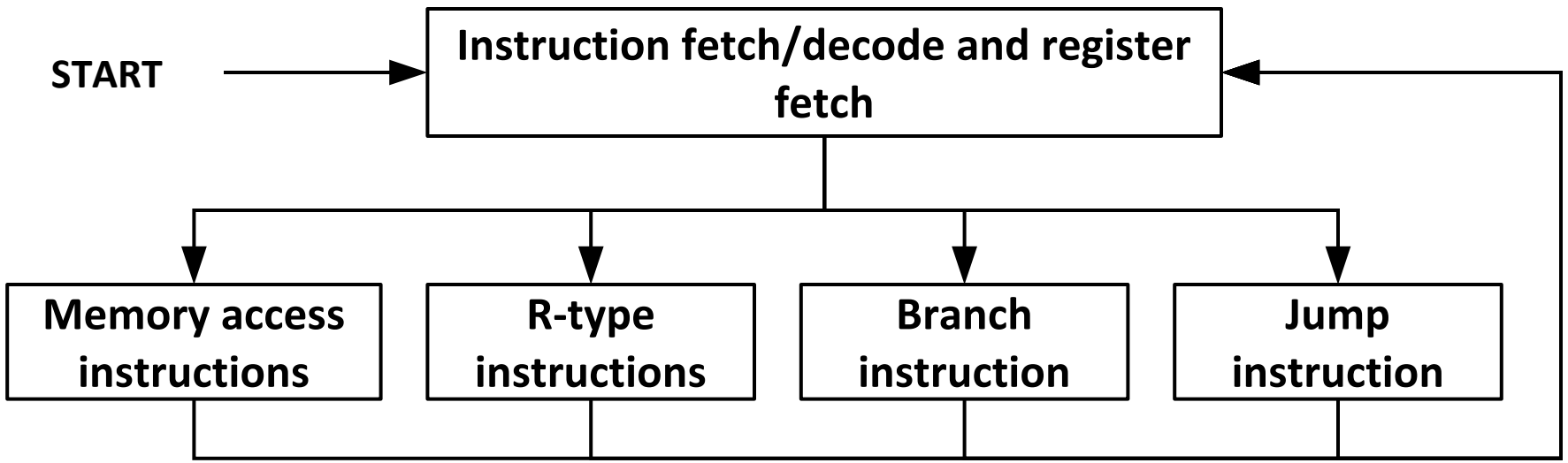
Implementace vícecyklové datové cesty



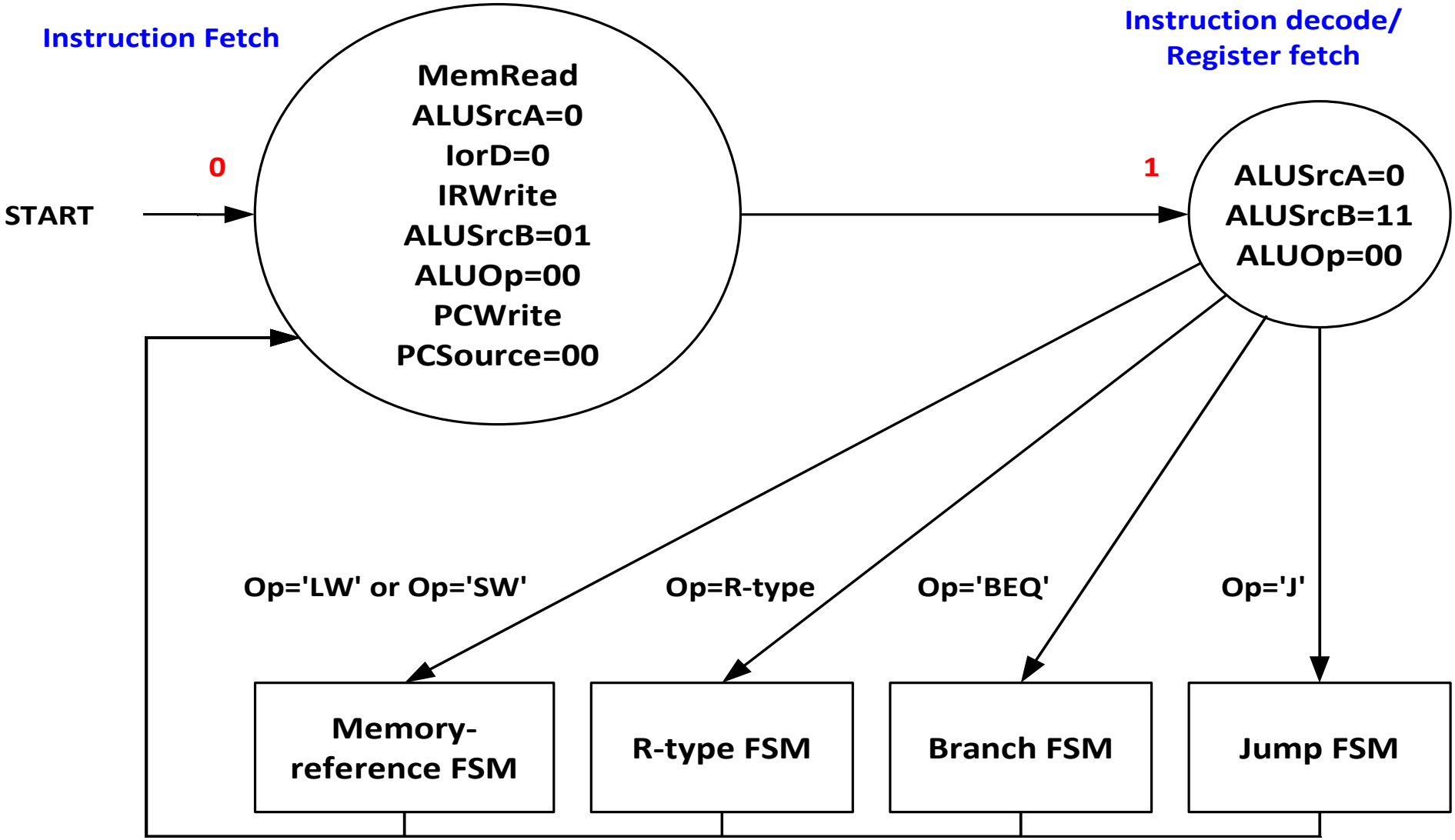
Řízení vícecyklové datové cesty je sekvenční

Zpracování instrukce trvá více taktů

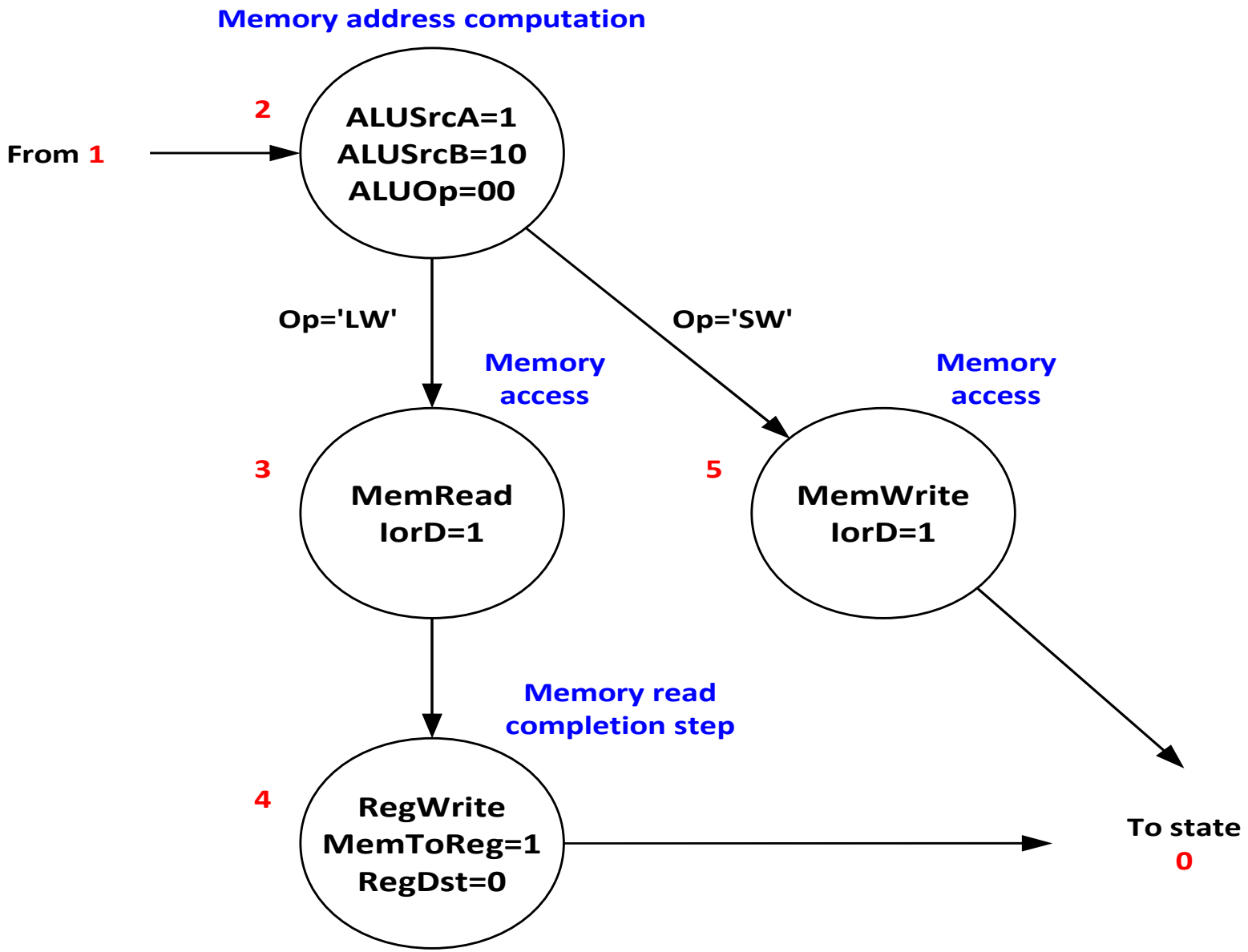
- řadič musí vědět, který krok zpracování instrukce má provést ⇒ sekvenční obvod ⇒ konečný automat
  - nachází se v jednom z množiny možných stavů, stav uchován v paměťovém prvku (registru)
  - kombinační logika určí následující stav, který se do stavového registru zapíše s náběžnou hranou hod. signálu



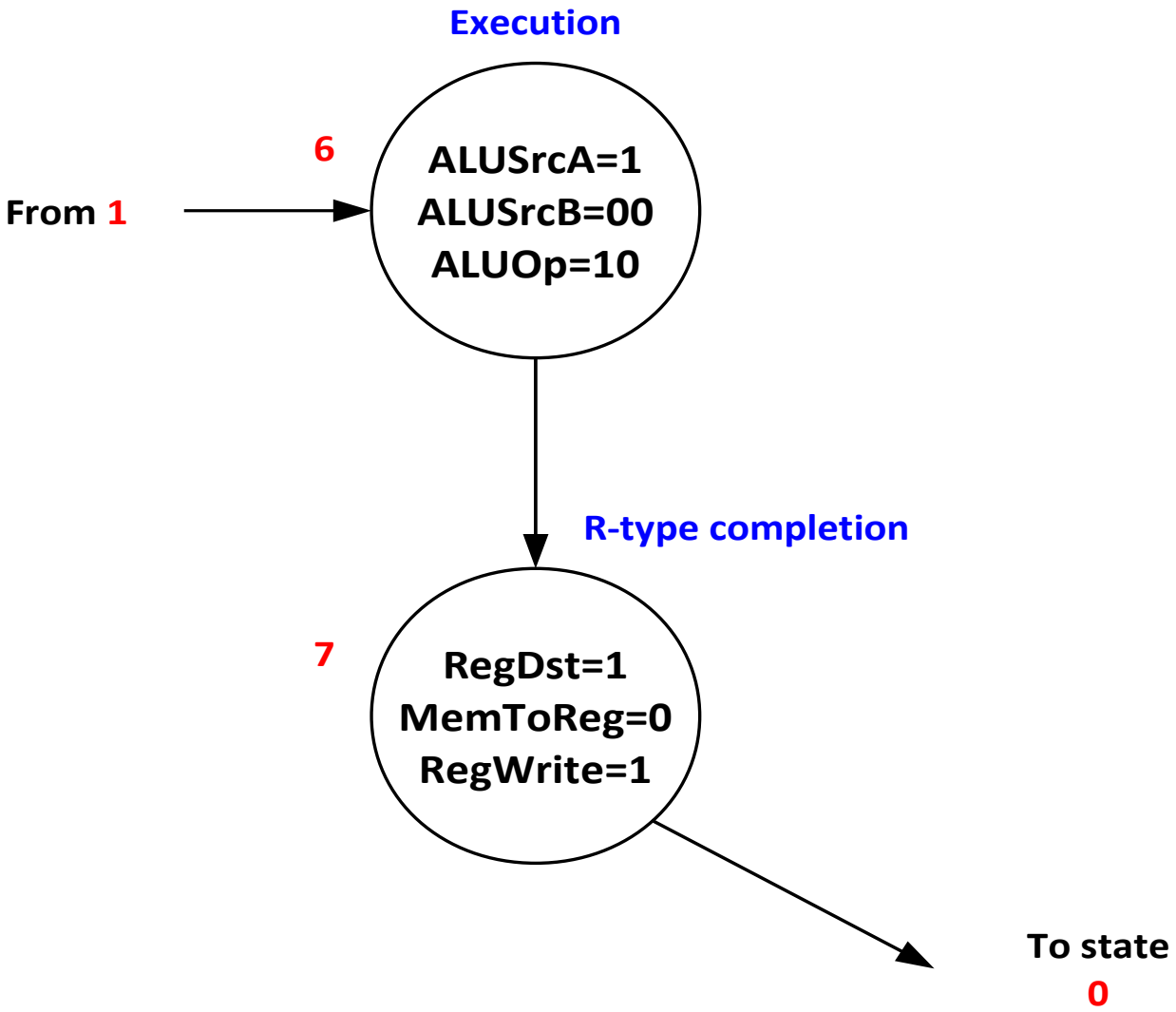
Instruction fetch/decode and register fetch



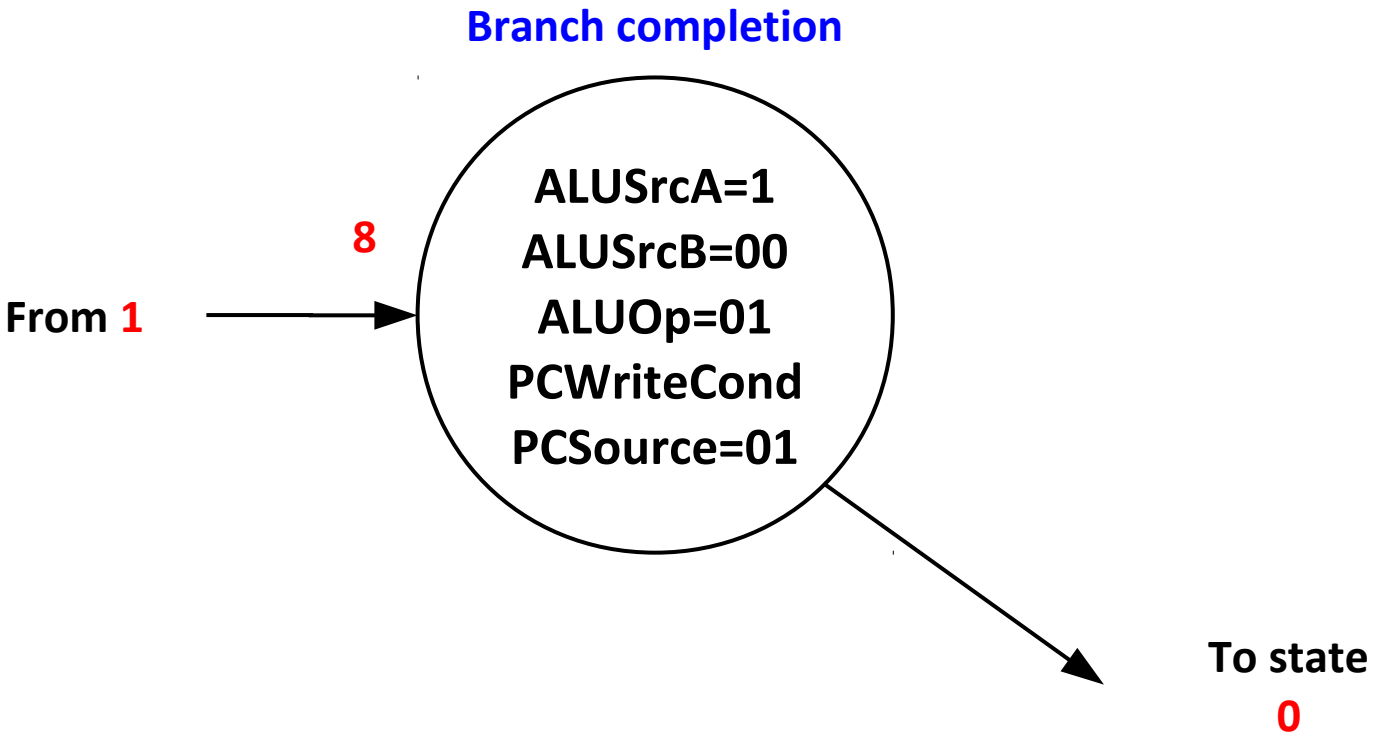
Memory reference FSM



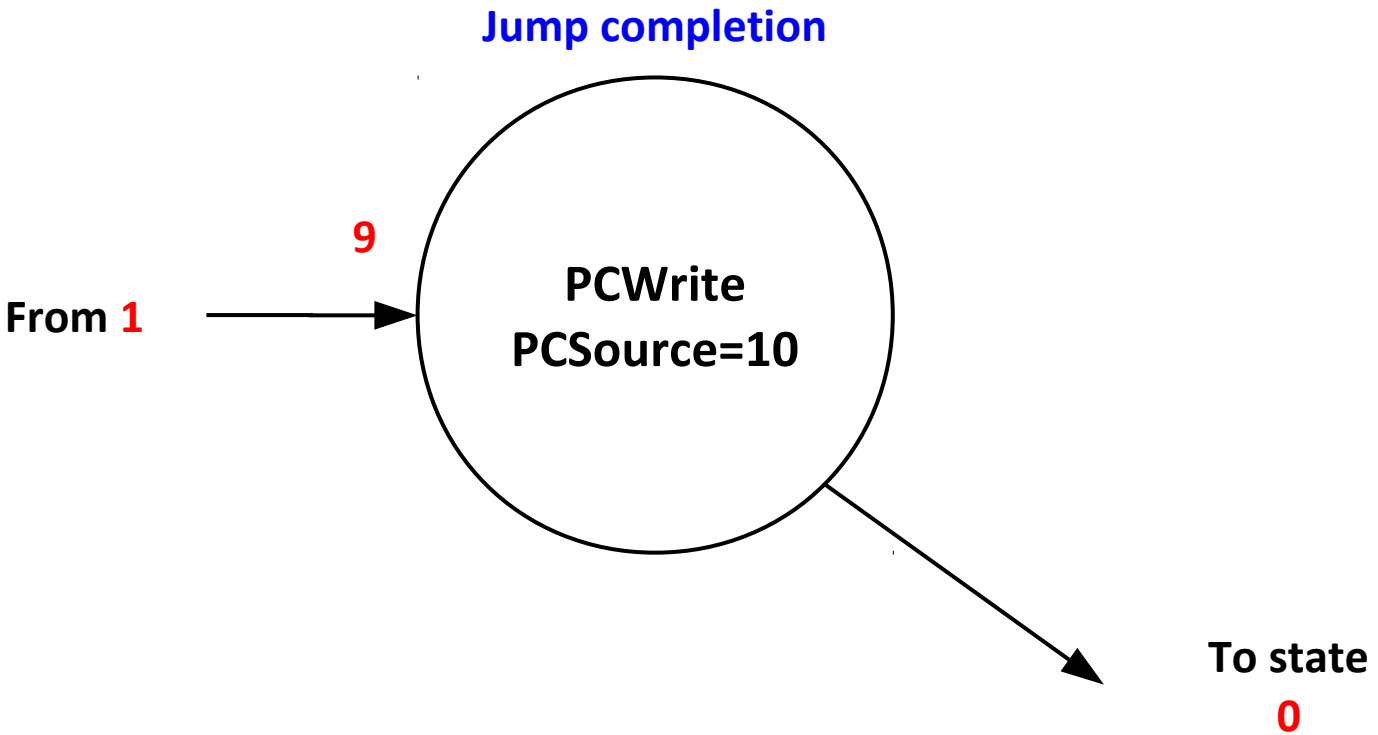
R-type FSM


Execution
6
From 1
ALUSrcA=1
ALUSrcB=00
ALUOp=10
R-type completion
7
RegDst=1
MemToReg=0
RegWrite=1
To state
0

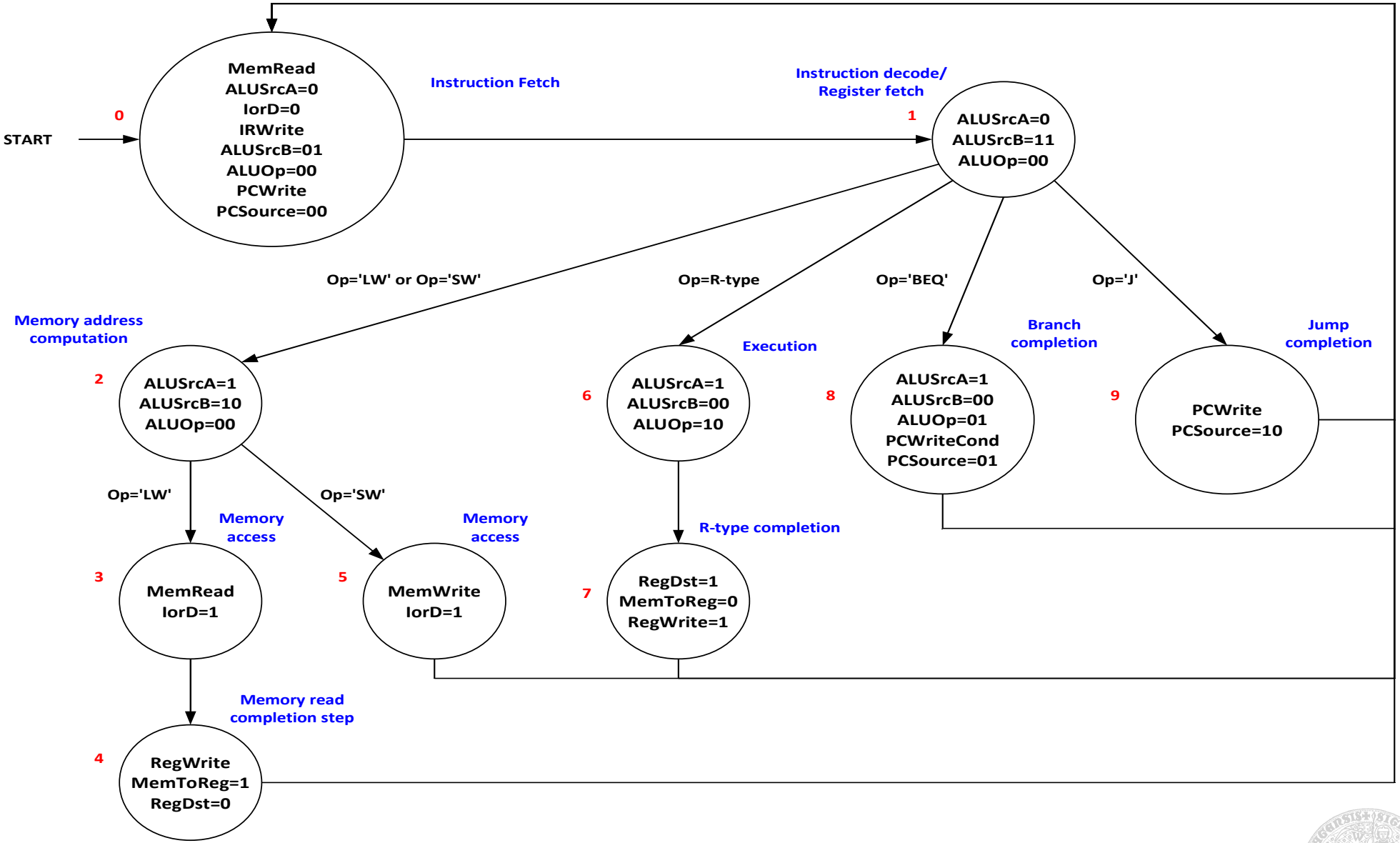
Branch FSM


Branch completion
8
From 1
ALUSrcA=1
ALUSrcB=00
ALUOp=01
PCWriteCond
PCSource=01
To state
0

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)

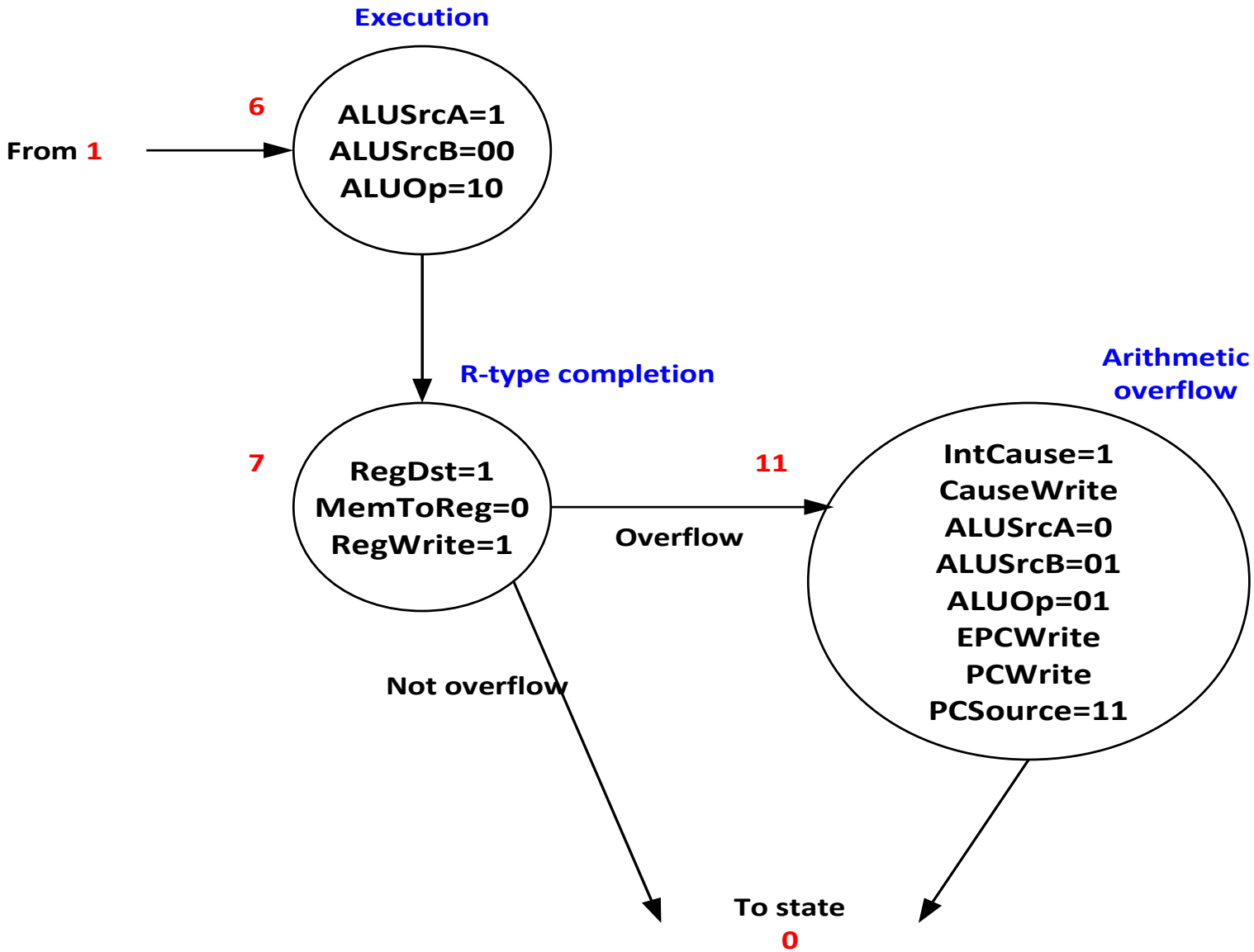
- periferní zařízení
- selhání hardware

Podpora výjimek a přerušení

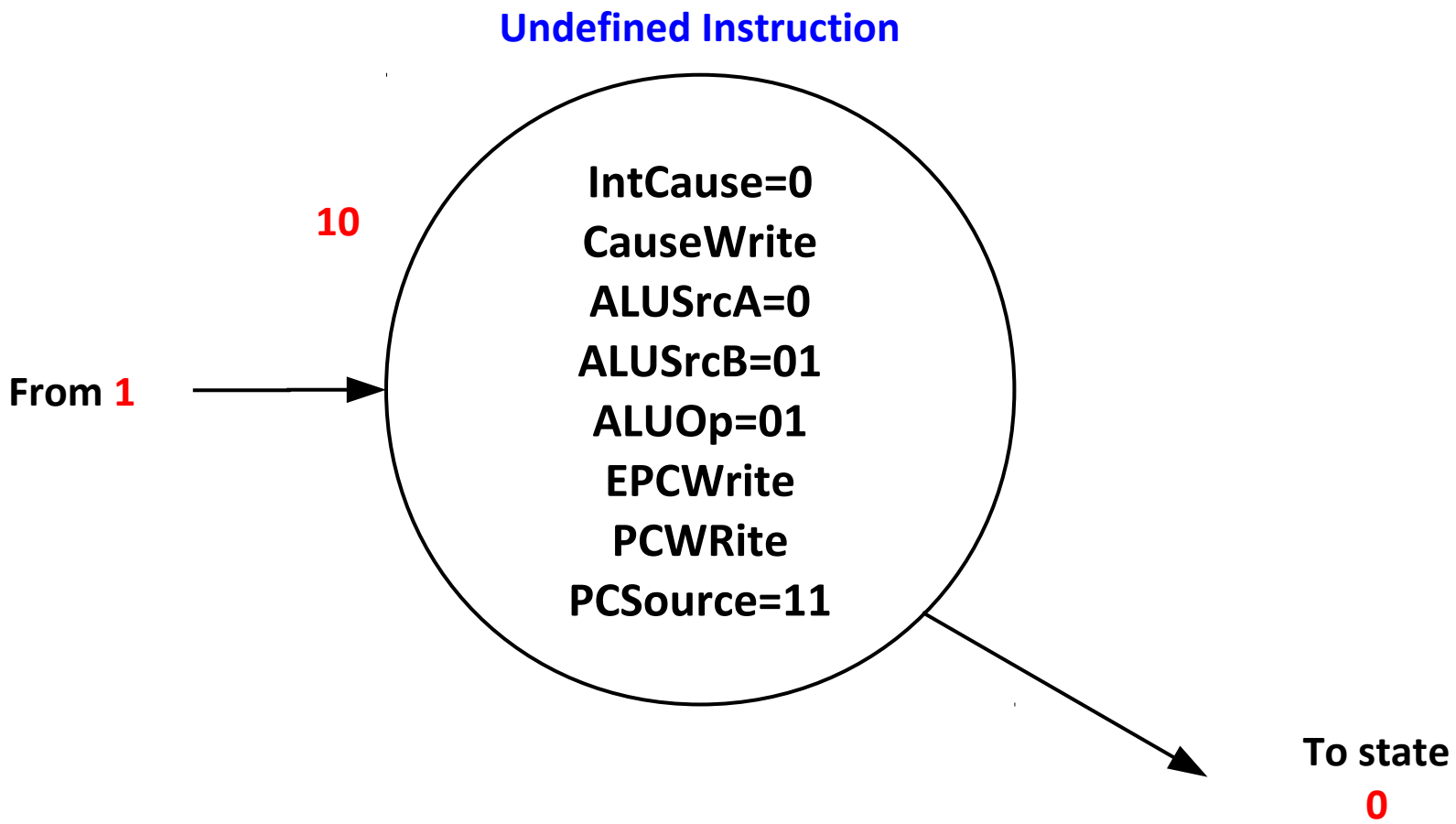
Hardware

- 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 vypočtu
- obnovení stavu původního výpočtu
- návrat do původního programu
  - pokračovat následující instrukcí
  - restartovat instrukci, která výjimku vyvolala

Výkon vícecyklové 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 ⇒ výkon ***25 MIPS***

Jednoduchá vícecyklová datová cesta (takt 5ns)

- ***CPI*** ≈ *(90% × 1) + (10% × 4) = 1.3*
- ***6.5ns*** na instrukci ⇒ výkon ***153 MIPS***

Jemně členěná vícecyklová datová cesta (takt 1ns)

- ***CPI*** ≈ *(30% × 5) + (10% × 5) + (50% × 4) + (10% × 20) = 6*
- ***6ns*** na instrukci ⇒ 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

Obvodový řadič

Obvodová realizace konečného automatu

- přechody stavovým diagramem
- standardní metody sekvenční logiky

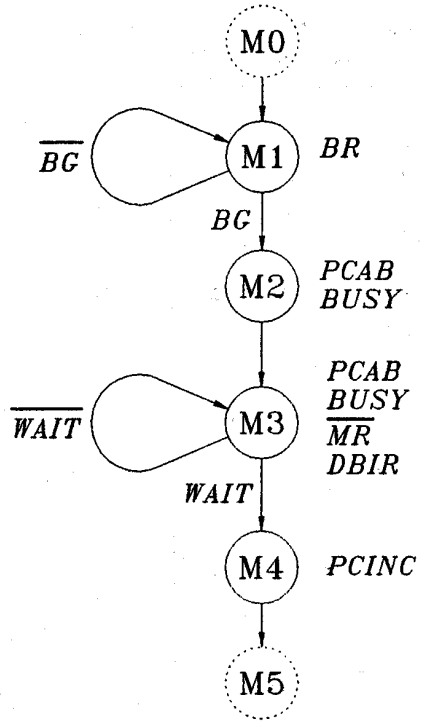


Figure 7.9. *Instruction fetch — state diagram*

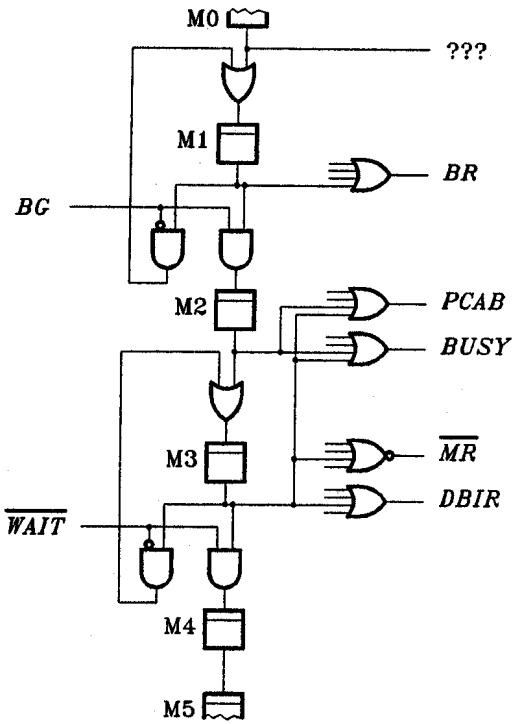
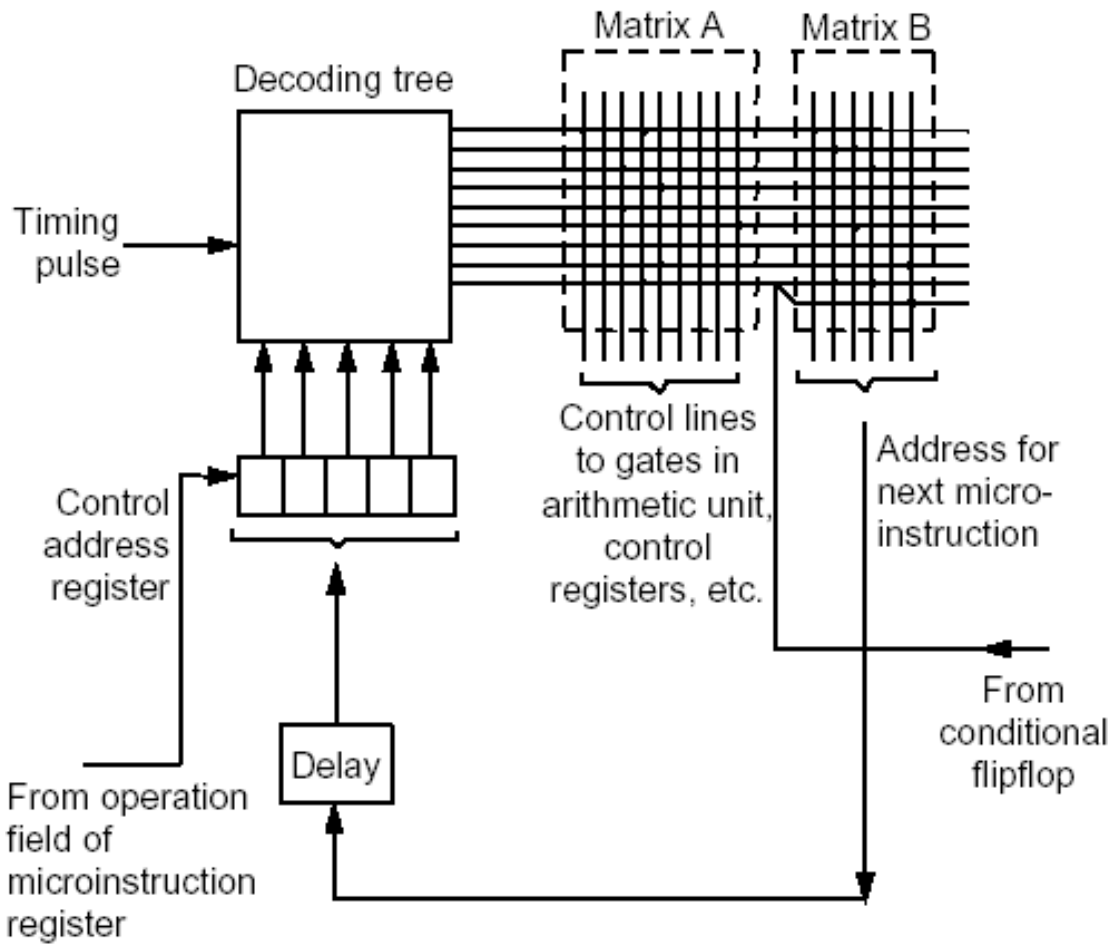


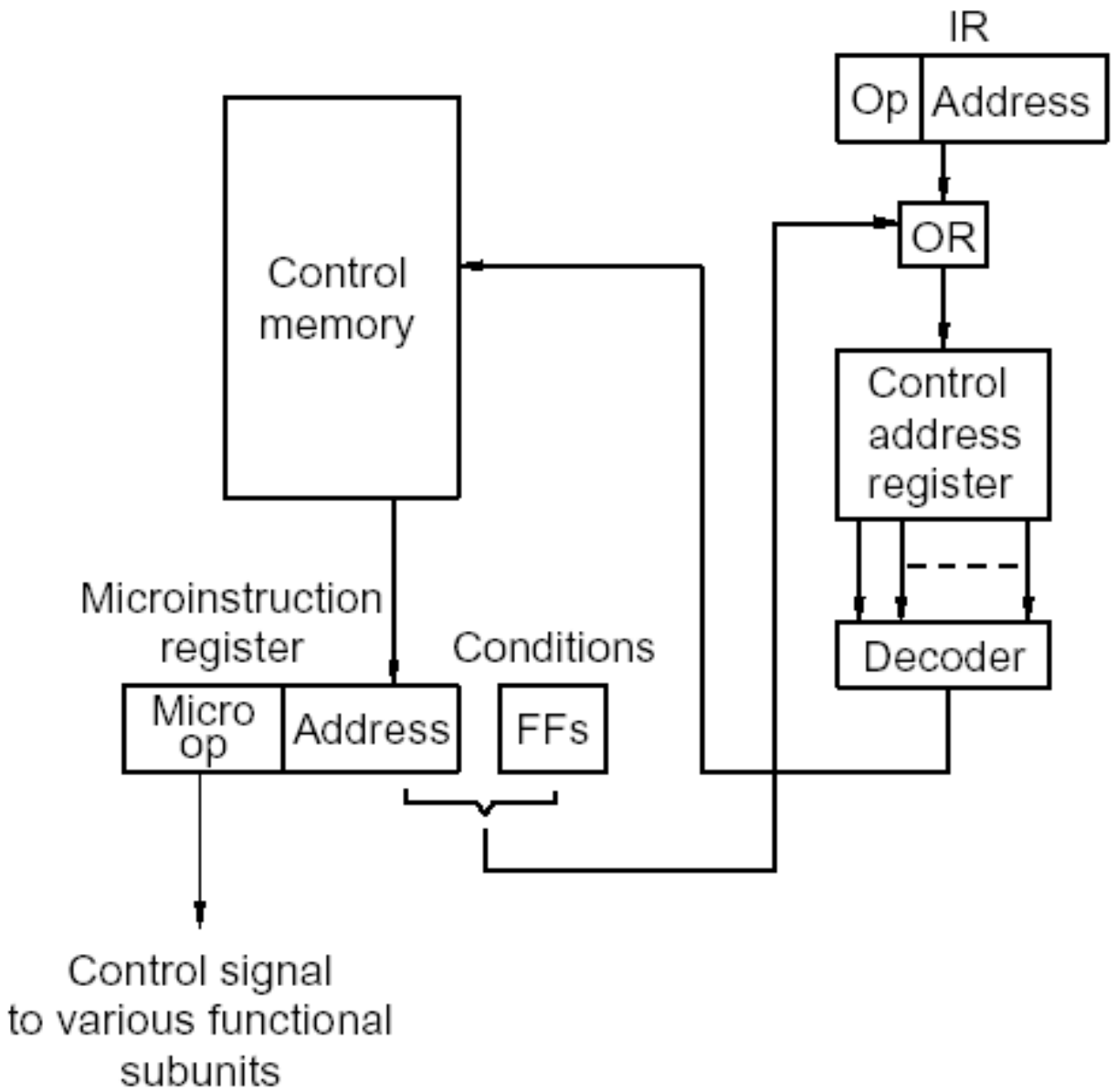
Figure 7.10. *Part of hard-wired controller*

Mikroprogramový řadič

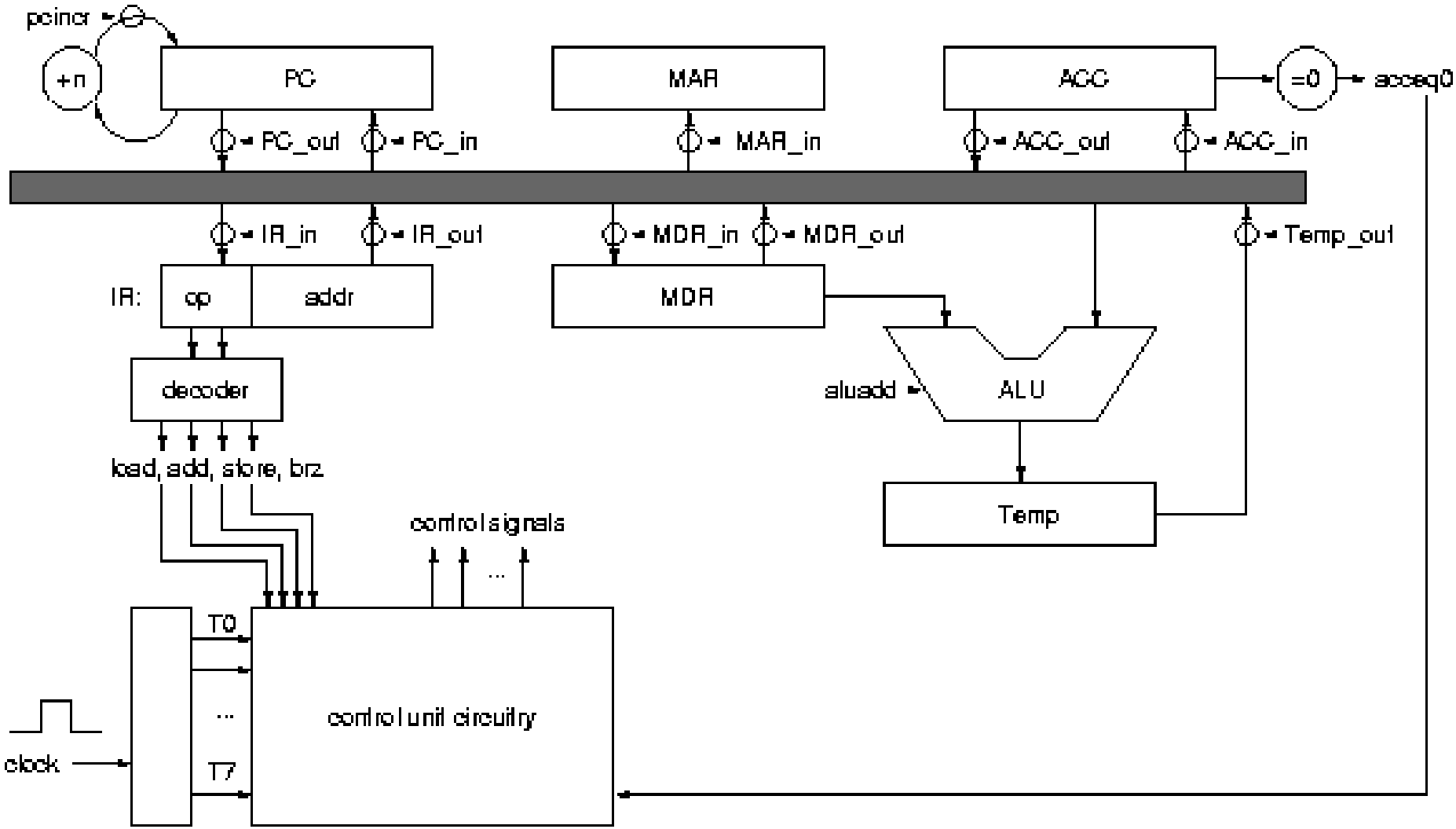
Maurice V. Wilkes, 1951



Princip vykonávání mikrokódu



Mikroprogramování na jednoduchém CPU



Zdroj: Mark Smotherman - A Brief History of Microprogramming

Architektura (akumulátorová)

Registry

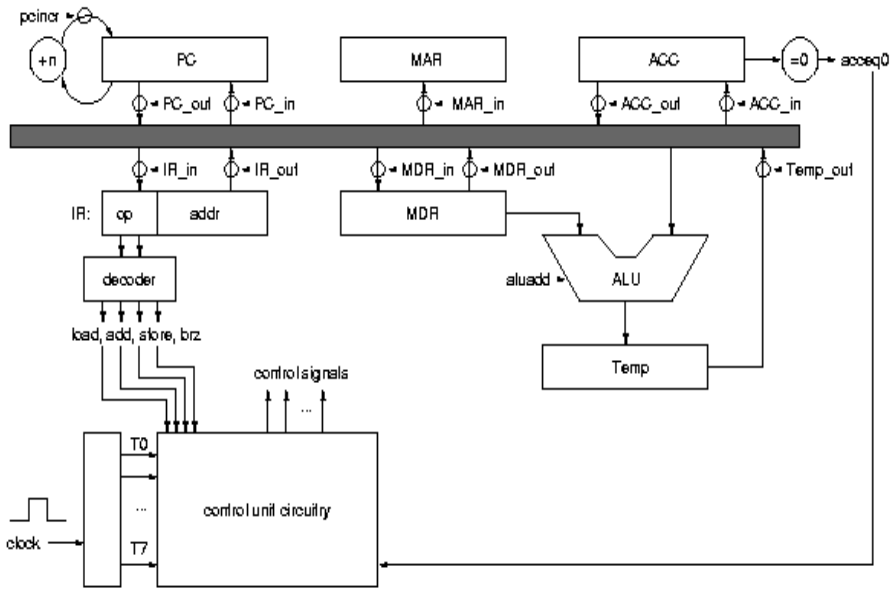
- ACC (akumulátor)

Instrukční sada

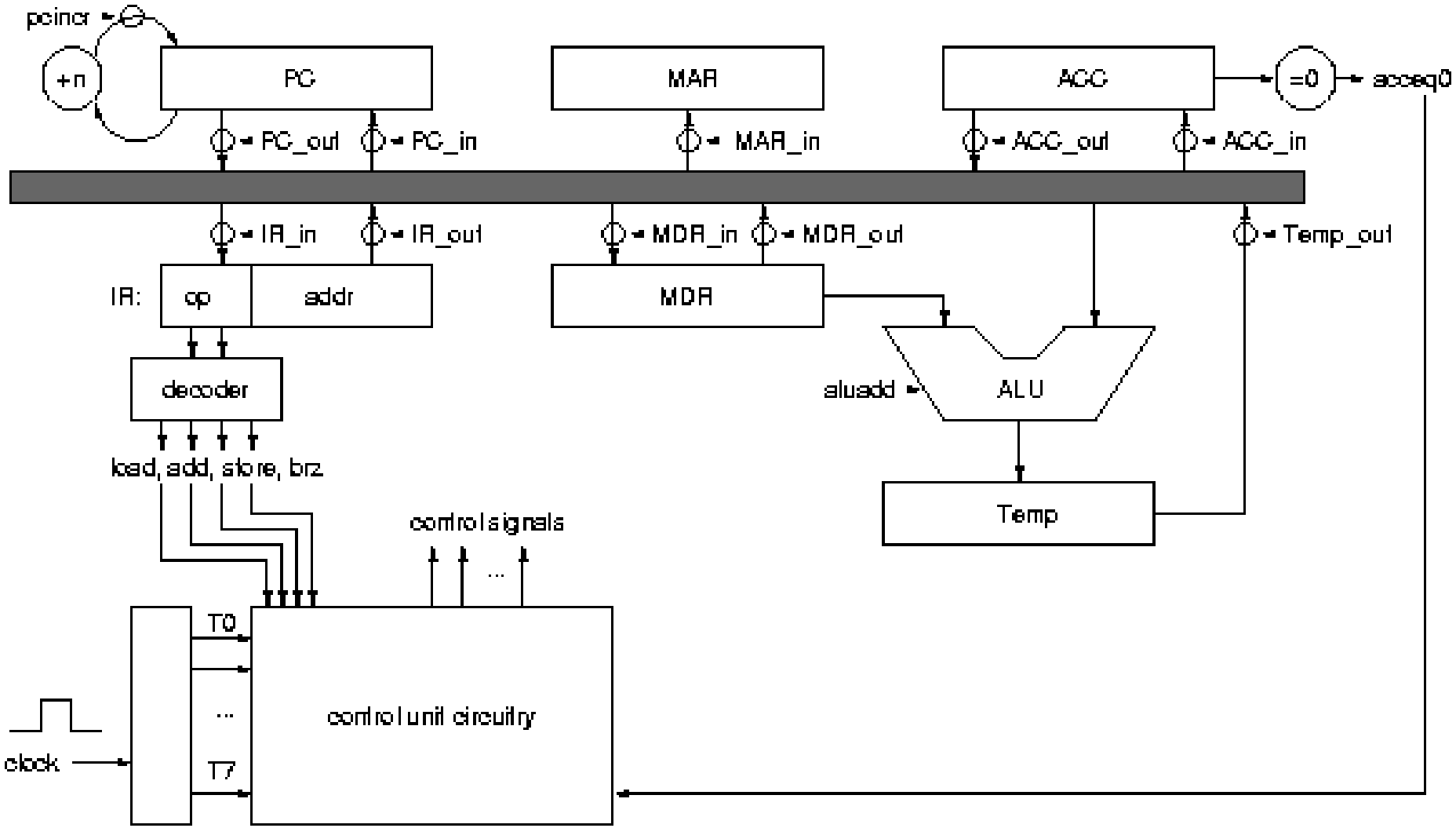
- instrukce 8 bitů (2 bity operační kód, 6 bitů adresa)
- opcode=00: ***load*** (*ACC from memory*)
  ACC <= memory [address]
- opcode=01: ***add*** (*to ACC from memory*)
  ACC <= ACC + memory [address]
- opcode=10: ***store*** (*ACC to memory*)
  memory [address] <= ACC
- opcode=11: ***branch*** (*to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

Řídící signály

```
pcincr    : PC <= PC + 1
PC_in     : PC <= CPU internal bus
PC_out    : CPU internal bus <= PC

IR_in     : IR <= CPU internal bus
IR_out    : CPU internal bus <= address portion of IR

MAR_in    : MAR <= CPU internal bus
MDR_in    : MDR <= CPU internal bus
MDR_out   : CPU internal bus <= MDR

read      : MDR <= memory[ MAR ]
write     : memory[ MAR ] <= MDR

ACC_in    : ACC <= CPU internal bus
ACC_out   : CPU internal bus <= ACC

TEMP_out  : CPU internal bus <= 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
```

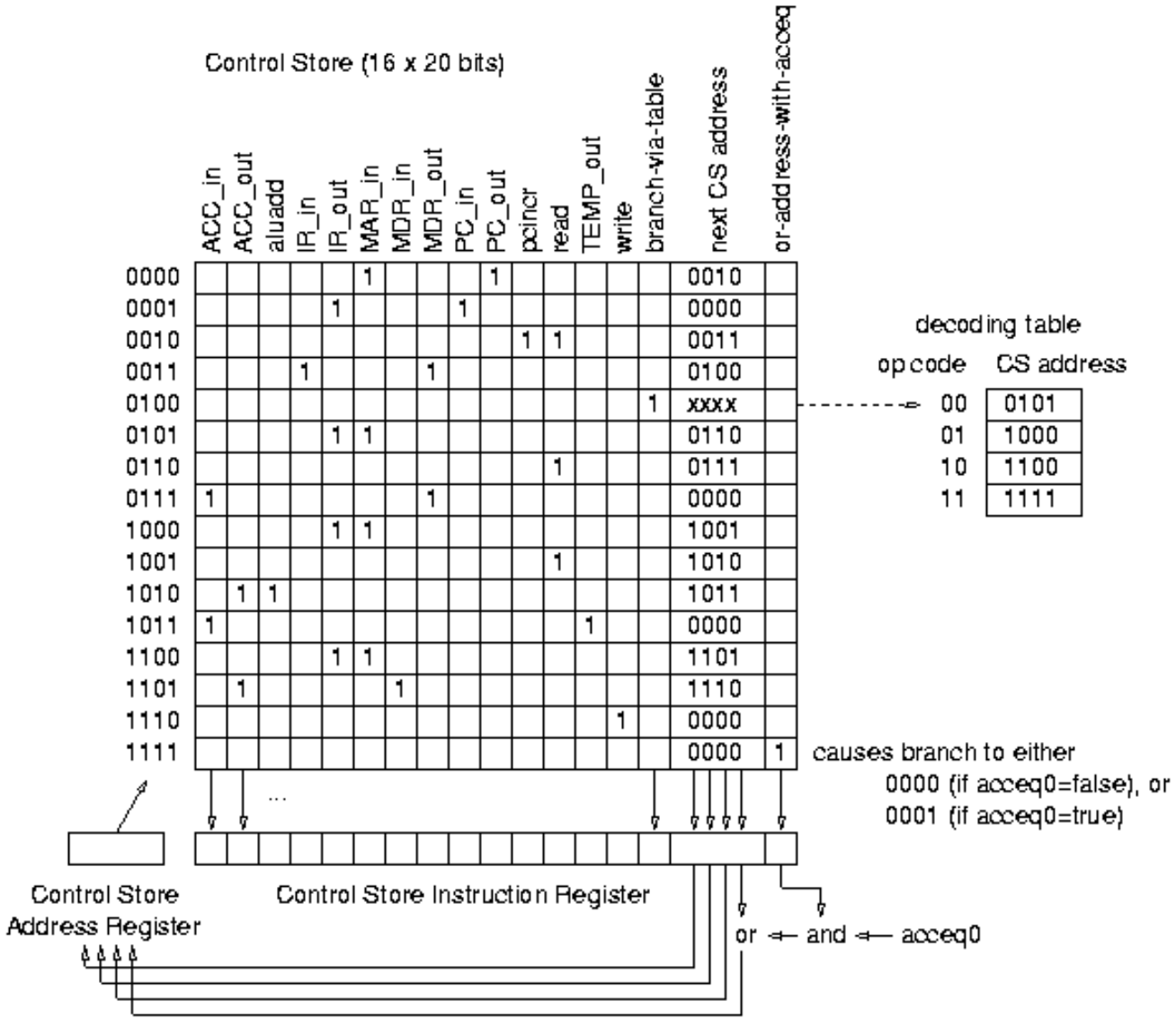
- execute

```
T4:  IR_out(addr part), MAR_in
T5:  read
T6:  MDR_out, ACC_in, reset to T0
```

Realizace řídící 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
```

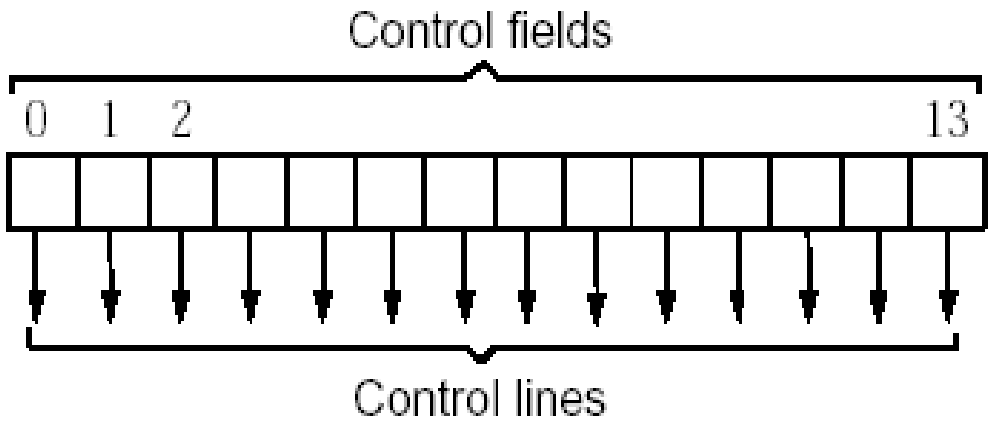
Realizace řídící logiky: mikrořadič s řídící pamětí



Horizontální formát mikroinstrukcí

Přímá reprezentace řídících signálů

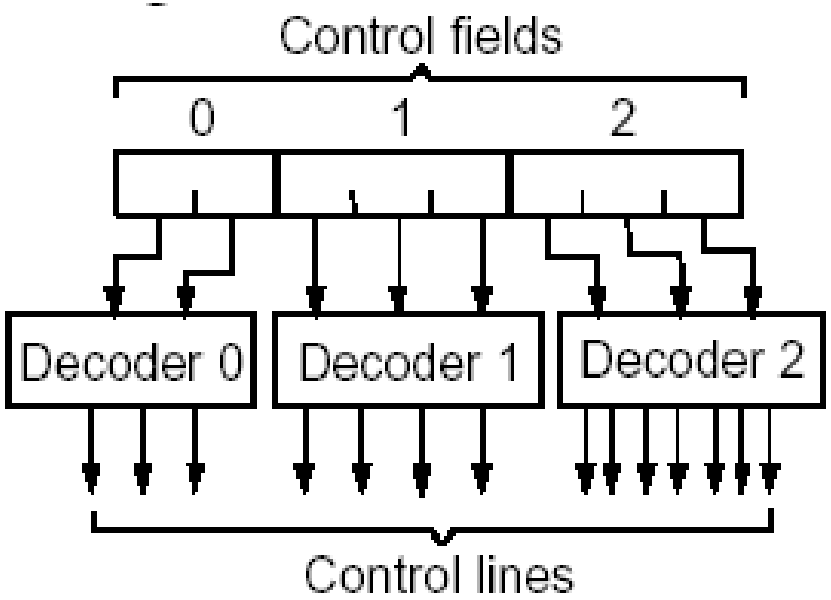
- mikroinstrukce obsahuje přímo hodnoty řídících signálů
- není třeba dekódovat → rychlost
- libovolná kombinace → pružnost
- velké nároky na prostor

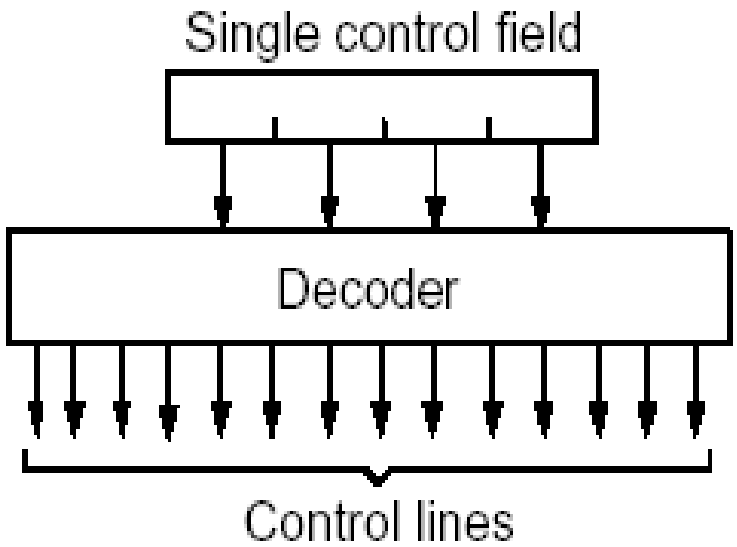


Vertikální formát mikroinstrukcí

Kódovaná reprezentace řídících signálů

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





Nanoprogramování

Základní myšlenka

- jen některé kombinace řídících signálů dávájí smysl
- smysluplné kombinace očíslujeme a uložíme do paměti

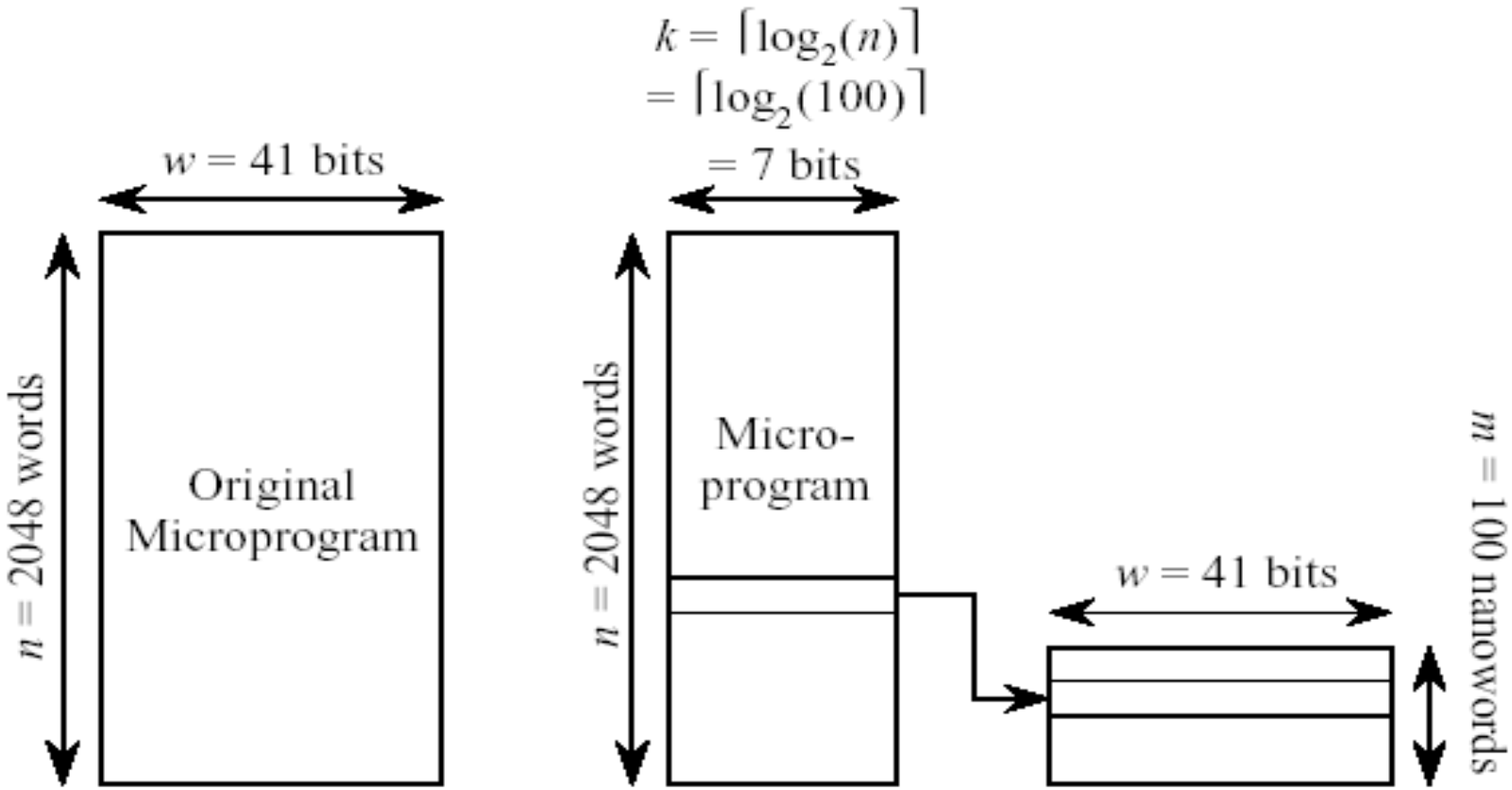
Dvojúrovňová řídící paměť

- 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 řídícím 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