

Úvod do organizace počítače

Výpočetní struktura pro $CPI = 1$

Opakování

- Konstrukce datových cest
 - Specifické stavební bloky pro formáty (R, I a J)
 - Modulární návrh
 - ALU, registrový soubor, datová paměť
 - ALU nebo sčítačka pro výpočet adresy cíle skoku (BTA)
 - Datové cesty mezi výkonnými bloky, vytvořené podle požadavků jednotlivých instrukčních formátů
- Instrukční formáty a datové cesty
 - R: operace ALU
 - I: Load/store – vstup/výstup dat do/z registrového souboru nebo paměti
 - I: Branch (podmíněný skok) – vyhodnocení podmínky, výpočet BTA
 - J: Jump (nepodmíněný skok) – výpočet JTA

Přehled přednášky

- Lze postavit procesor tak, aby se operace provedly vždy v jednom taktu ?
 - Všechny instrukce provedeny tak, že $CPI = 1$
 - Zvýšení výkonu *software*
- Návrh realizovat z jednoduchých komponent
- Návrh provést iterativně (postupně „nabalováním“)
 - R-formát
 - I-formát
 - J-formát
- Problémy zpracování v jednom cyklu

MIPS procesor s jednoduchým cyklem

- Instruction Set Architecture je *interface* definující hardwarové operace, které má software k dispozici.
- Libovolný instrukční soubor může být implementován různými způsoby. V příštích hodinách porovnáme dvě důležité implementace.
 - V základní **implementaci s jednoduchým cyklem** trvají všechny operace stejnou dobu – jeden cykl.
 - V **implementaci s režimem pipeline** se v procesoru při provádění instrukce překrývají, což přináší potenciálně vyšší výkon.

Implementace s jednoduchým cyklem

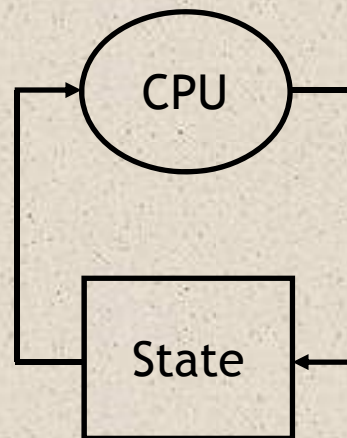
- Popíšeme implementaci instrukčního souboru procesoru na bázi MIPS s následujícími operacemi.

Aritmetické:	add	sub	and	or	slt
Přenos dat:	lw	sw			
Řídící:	beq				

- Použijeme architekturu MIPS, protože ji lze podstatně snáze implementovat než architekturu x86.
- Začneme s **implementací** instrukčního souboru **s jednoduchým cyklem**.
 - Doba provádění všech instrukcí bude stejná. Tím je určena doba cyklu pro rovnici výkonu.
 - Bude vysvětlena funkce datových cest a řídící jednotky.

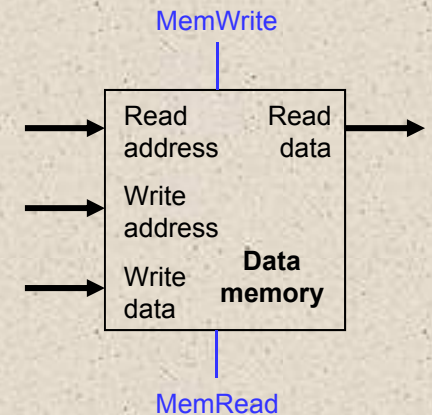
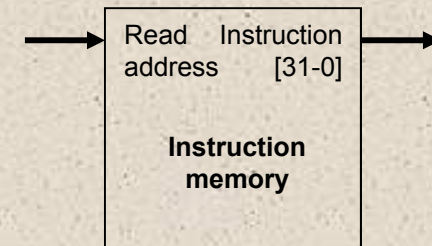
Počítače jsou automaty

- Počítač je vlastně „**velký automat**“ (state machine).
 - Registry, paměť, pevné disky a jiné paměťové prvky formují stav.
 - Procesor provádí „aktualizaci“ stavu podle instrukcí programu.
- Modelování chování počítačů se opírá o teorii automatů (konečných automatů).



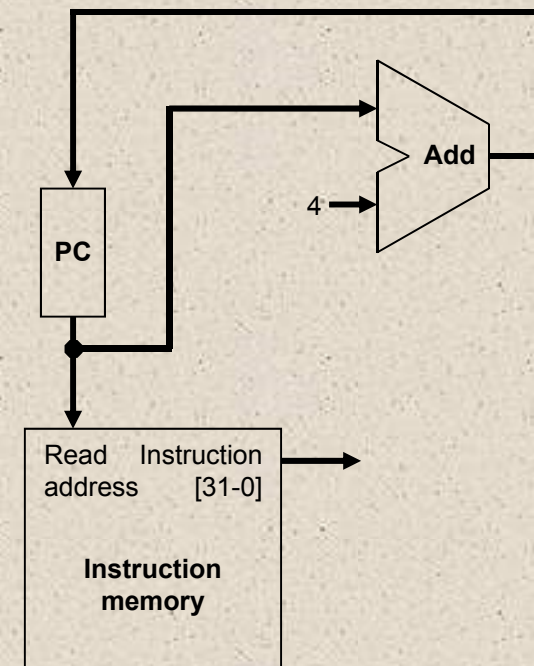
Paměti

- Začneme s jednodušší **Harvardskou architekturou**, kde data a instrukce leží v *oddělených* pamětech.
- Pro čtení instrukce a čtení & zápis slov, potřebujeme paměti široké 32-bitů (sběrnice reprezentovány tmavou tlustou čarou).
- Modré linky reprezentují řídicí signály. **MemRead** (**MemWrite**) se nastaví na 1 jestliže se provádí čtení (zápis) z (do) datové paměti (**data memory**). V opačném případě 0.
- Pro začátek se nebudeme zabývat zápisem do instrukční paměti (**instruction memory**).
 - Necht' instrukční paměť už obsahuje program a ten se během zpracování nemění.

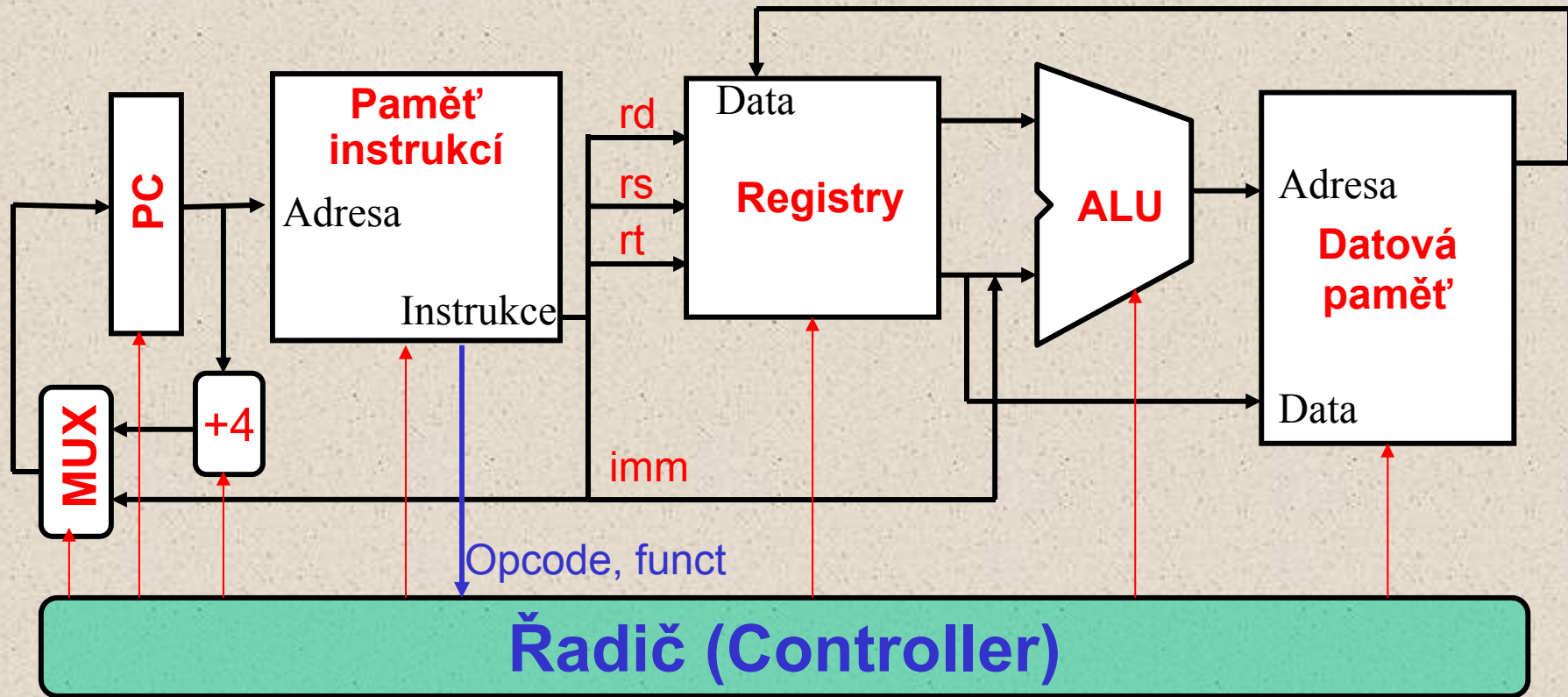


Čtení instrukce

- CPU pracuje v nekonečné smyčce – čte instrukce z paměti a provádí je.
- **Program counter** (register **PC**) obsahuje adresu aktuální instrukce.
- Instrukce MIPS jsou dlouhé 4 byte. Proto je PC inkrementován o 4, aby ukazoval na následující instrukci.



Přehled implementace



- **Datové cesty** umožňují přesuny obsahu registrů při provádění instrukcí
- Řízení zajišťuje provedení správných přesunů

Dekódování instrukcí (typu-R)

- Aritmetické instrukce typu Register-to-Register používají formát **typu-R**.
 - **op** je operační kód a **func** specifikuje blíže prováděnou aritmetickou operaci
 - **rs**, **rt** a **rd** jsou zdrojové a cílový registr.

op	rs	rt	rd	shamt	func
6 bitů	5 bitů	5 bitů	5 bitů	5 bitů	6 bitů

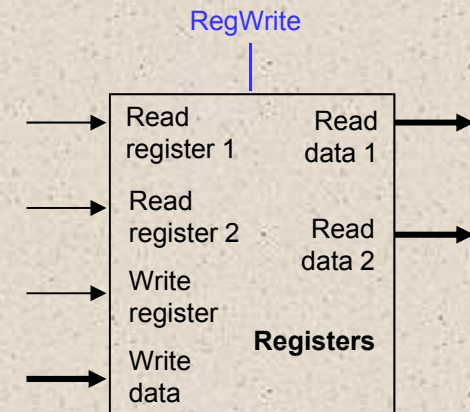
- Příklad instrukce a jejího dekodování:

add \$s4, \$t1, \$t2

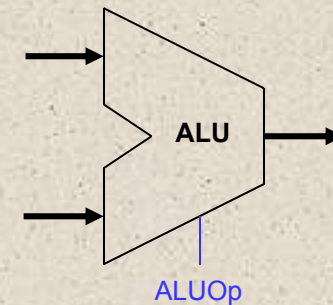
000000	01001	01010	10100	00000	1000000
--------	-------	-------	-------	-------	---------

Registry a ALU

- Instrukce typu R pracují s registry a s ALU.
- **Registrový soubor** obsahuje 32 32-bitových hodnot.
 - Každý specifikátor registru je dlouhý 5 bitů.
 - V jednom okamžiku lze číst dva registry.
 - **RegWrite** je 1, má-li být zapisováno do registru.
- Níže je jednoduchá **ALU** s pěti operacemi, výměr se provádí 3-bitovým polem (řídící signály) **ALUOp**.

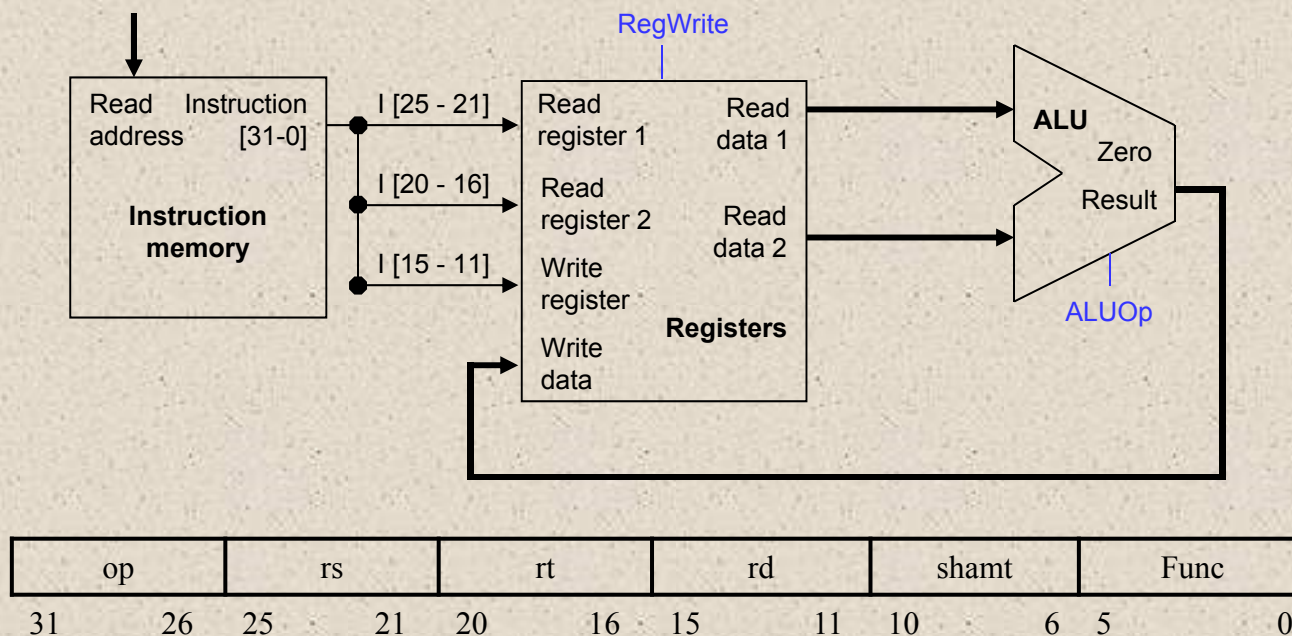


ALUOp	Funkce
000	and
001	or
010	add
110	subtract
111	slt



Provedení instrukce (typu-R)

1. Čtení instrukce z instrukční paměti.
2. Zdrojové registry, určené v instrukci poli **rs** a **rt**, se čtou z registrového souboru.
3. ALU provede požadovanou operaci.
4. Výsledek se uloží do cílového registru, určeného polem **rd** v instrukci.



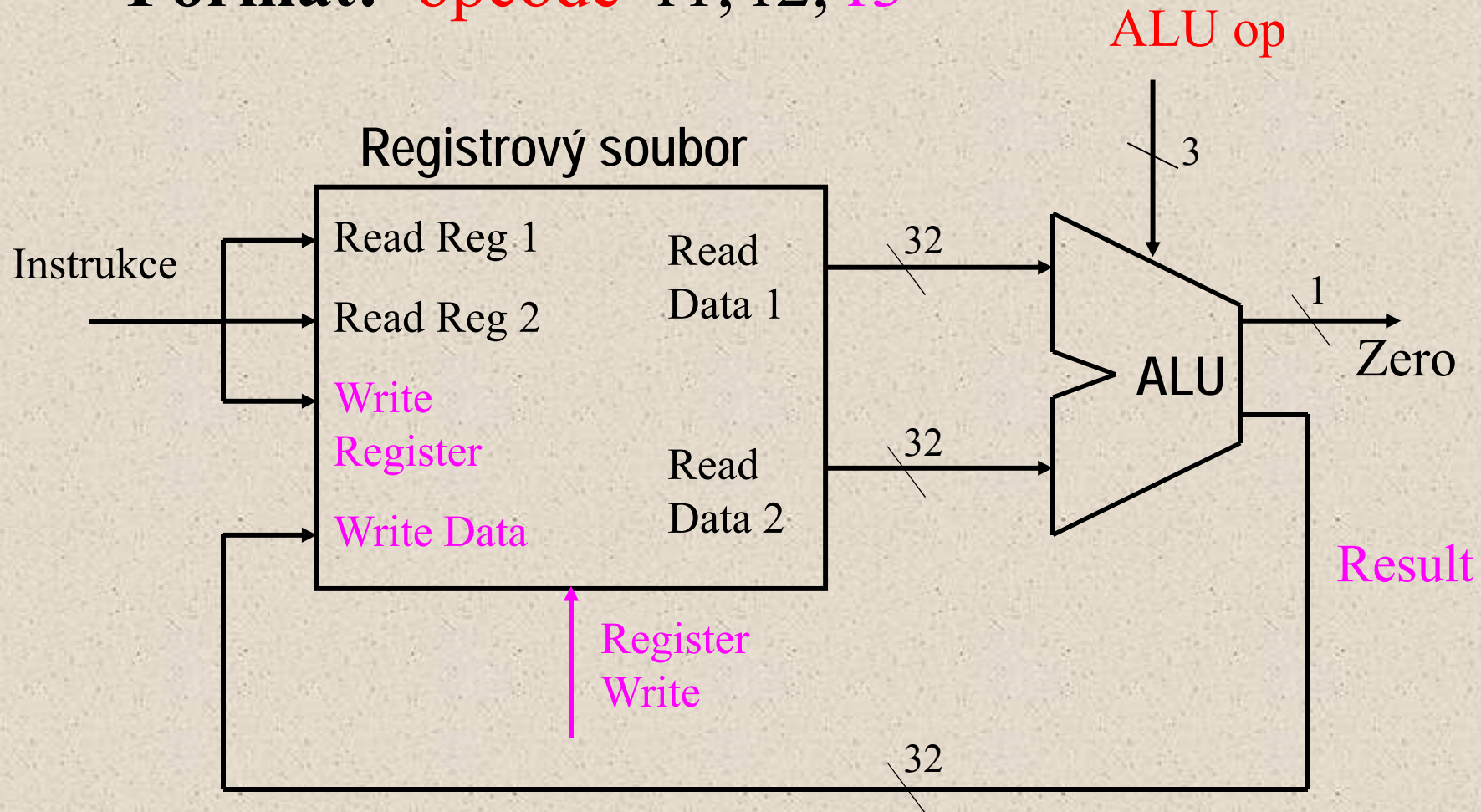
R-formát - provedení

Instrukce: `add $t0, $t1, $t2`

1. Načtení instrukce a inkrement PC
2. Vstup \$t0 a \$t1 z registrové sady
3. ALU zpracovává \$t0 a \$t1 podle pole *funct* instrukce MIPS (bity 5-0)
4. Výsledek z ALU je zapsán do registrové sady, bity 15-11 instrukce vybírají cílový registr (např, \$t0).

Komponenty: R-formát

- **Formát:** **opcode** r1, r2, r3



Dekódování instrukcí typu - I

- Instrukce lw, sw a beq jsou kódovány ve formátu **typu-I**.
 - **rt** je *cílový registr* pro lw, ale *zdrojový registr* pro beq a sw.
 - **adresa** je 16-bitová konstanta se znaménkem.

op	rs	rt	adresa
6 bitů	5 bitů	5 bitů	16 bitů

- Dva příklady instrukcí:

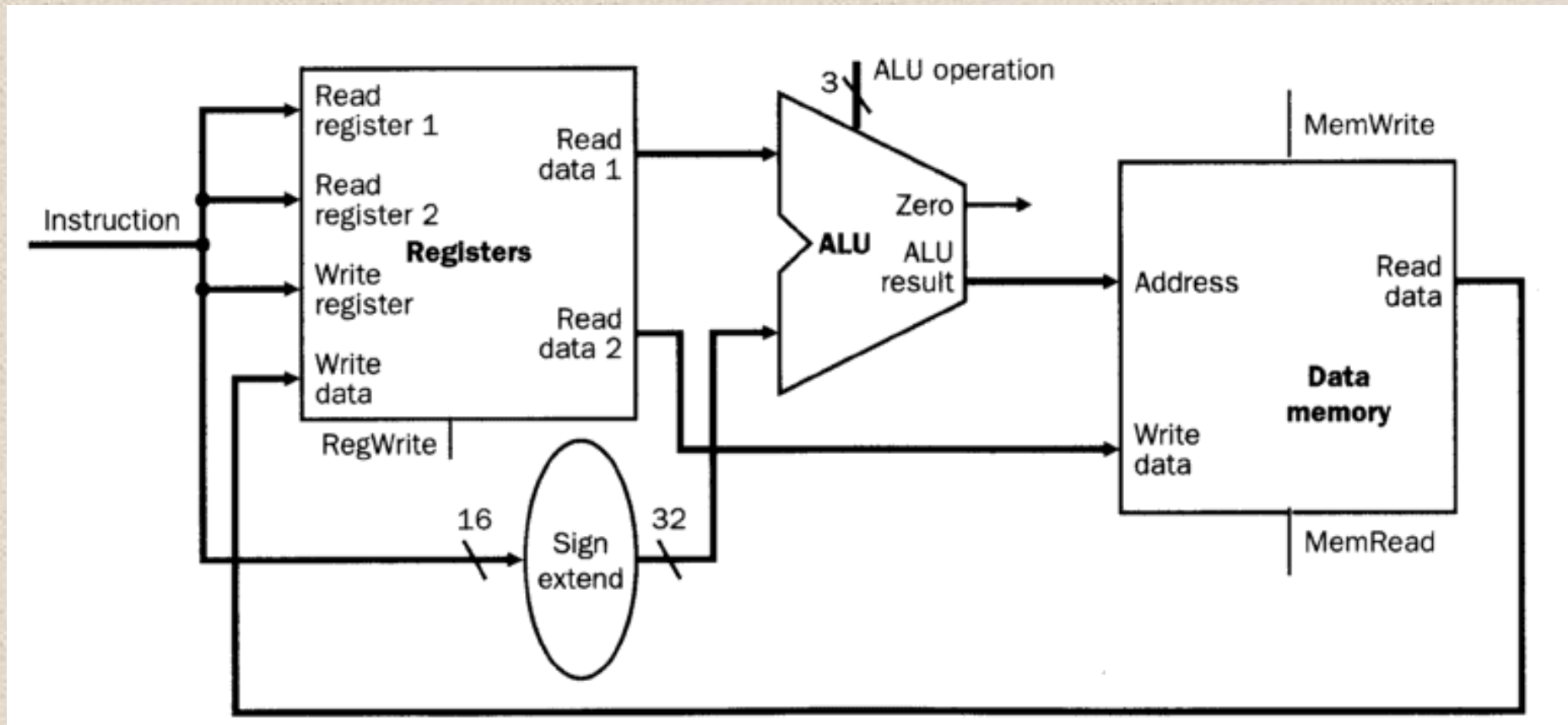
lw	\$t0, -4(\$sp)	100011	11101	01000	1111 1111 1111 1100
sw	\$a0, 16(\$sp)	101011	11101	00100	0000 0000 0001 0000

Load/Store - provedení

Instrukce: `lw $t1, offset($t2)`

1. Načtení instrukce a inkrement PC
2. Čtení obsahu registru (např., bázeová adresa v \$t2) z registrového souboru
3. ALU přičte hodnotu z \$t2 ke (znaménkem rozšířeným) dolním 16 bitům instrukce (`offset`)
4. Výsledek z ALU = adresa do datové paměti
5. Čtení dat z paměti, zápis do registrové sady, podle čísla registru, uvedeného v \$t1 (bity 20-16)

Komponenty: Load/Store

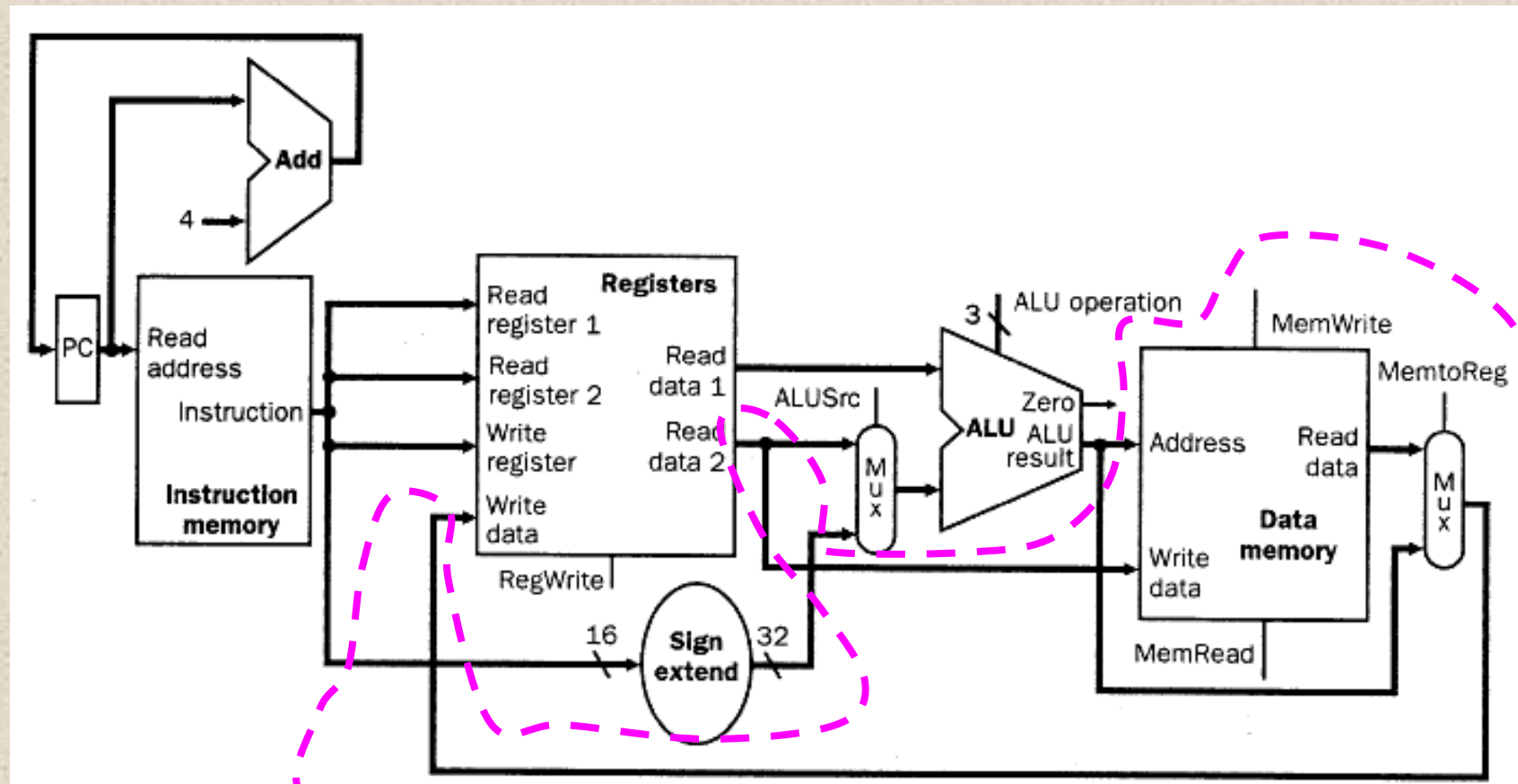


Fetch

Decode

Execute

R-format + Load/Store HW



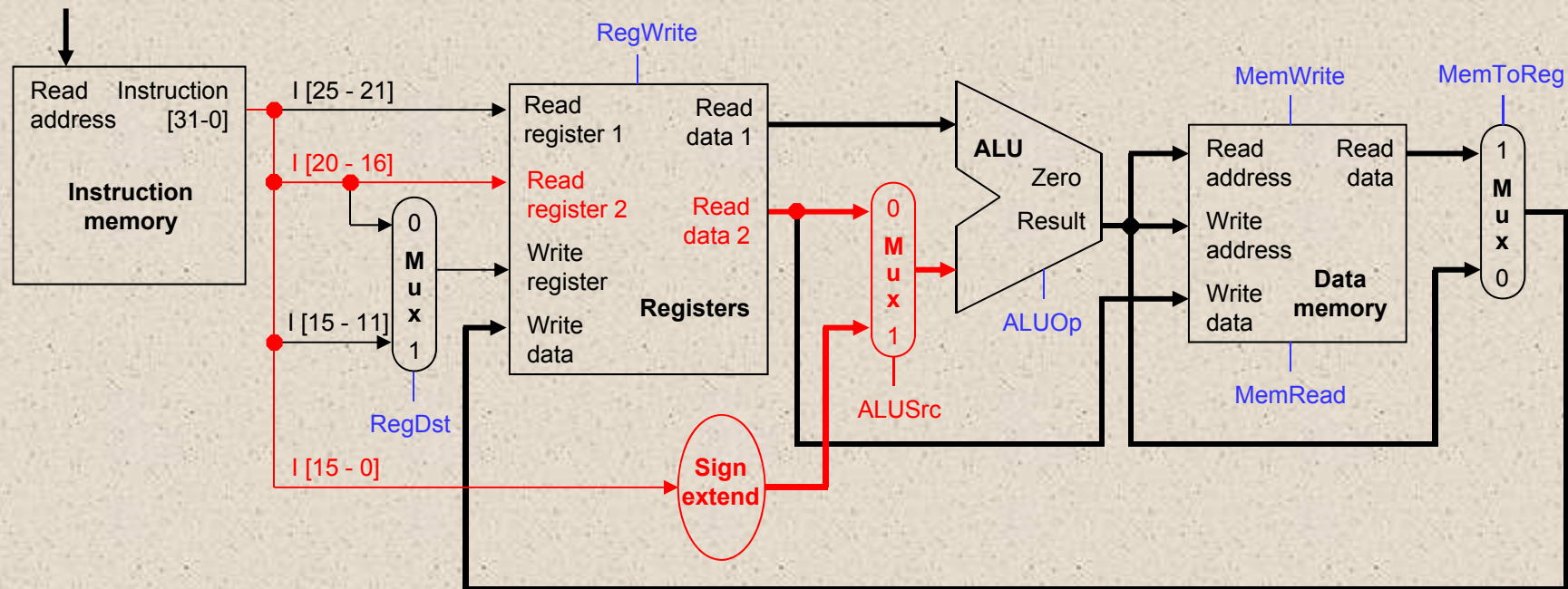
Fetch

Decode

Execute

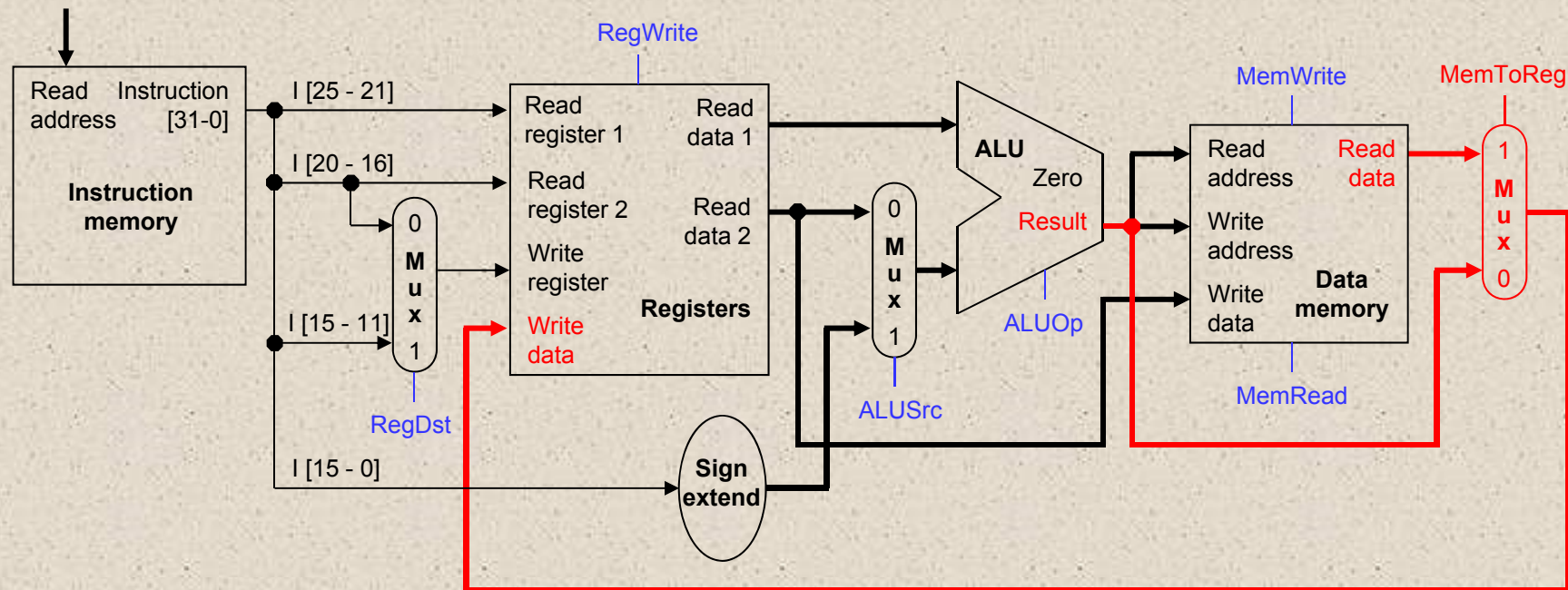
Přístup do datové paměti

- Pro instrukci typu jako např. `lw $t0, -4($sp)` je bazový registr `$sp` přičten ke konstantě, rozšířené na 32 bitů (se znaménkem). Tak vznikne adresa do paměti.
- To znamená, že ALU musí mít na vstupu buď registrový operand pro aritmetickou instrukci a nebo *přímý operand (immediate)* pro `lw` a `sw`.
- Doplníme multiplexer, řízený polem `ALUSrc`, který vybere registrový operand (0) nebo konstantu (1).



Přesun z paměti do registru

- Vstup do registrového souboru “Write data” také musí nabýt hodnotu výstupu ALU pro instrukce typu-R *nebo* hodnotu dat na výstupu paměti pro lw.
- Doplníme multiplexer řízený signálem **MemToReg**, který vybere výsledek ALU (0) nebo výstup datové paměti (1).



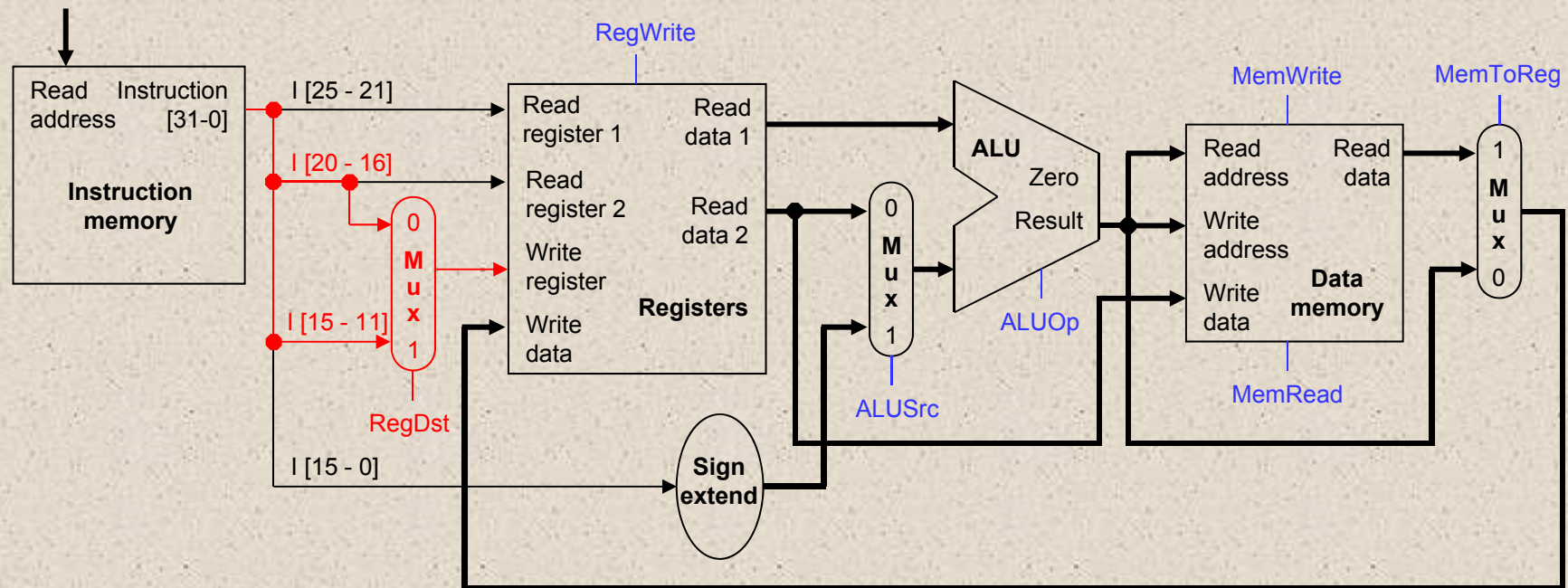
Cílový registr

- Zbývá vyřešit výběr cílového registru, pro instrukci lw je to pole *rt* namísto *rd*.

op	rs	rt	adresa
----	----	----	--------

lw \$rt, address(\$rs)

- Doplníme další multiplexer řízený signálem *RegDst* pro výběr cílového registru podle instrukčního pole *rt* (0) nebo pole *rd* (1).



Větvení

- Pro instrukce větvení nepředstavuje konstanta adresu, ale *offset* registru PC od cílové adresy.

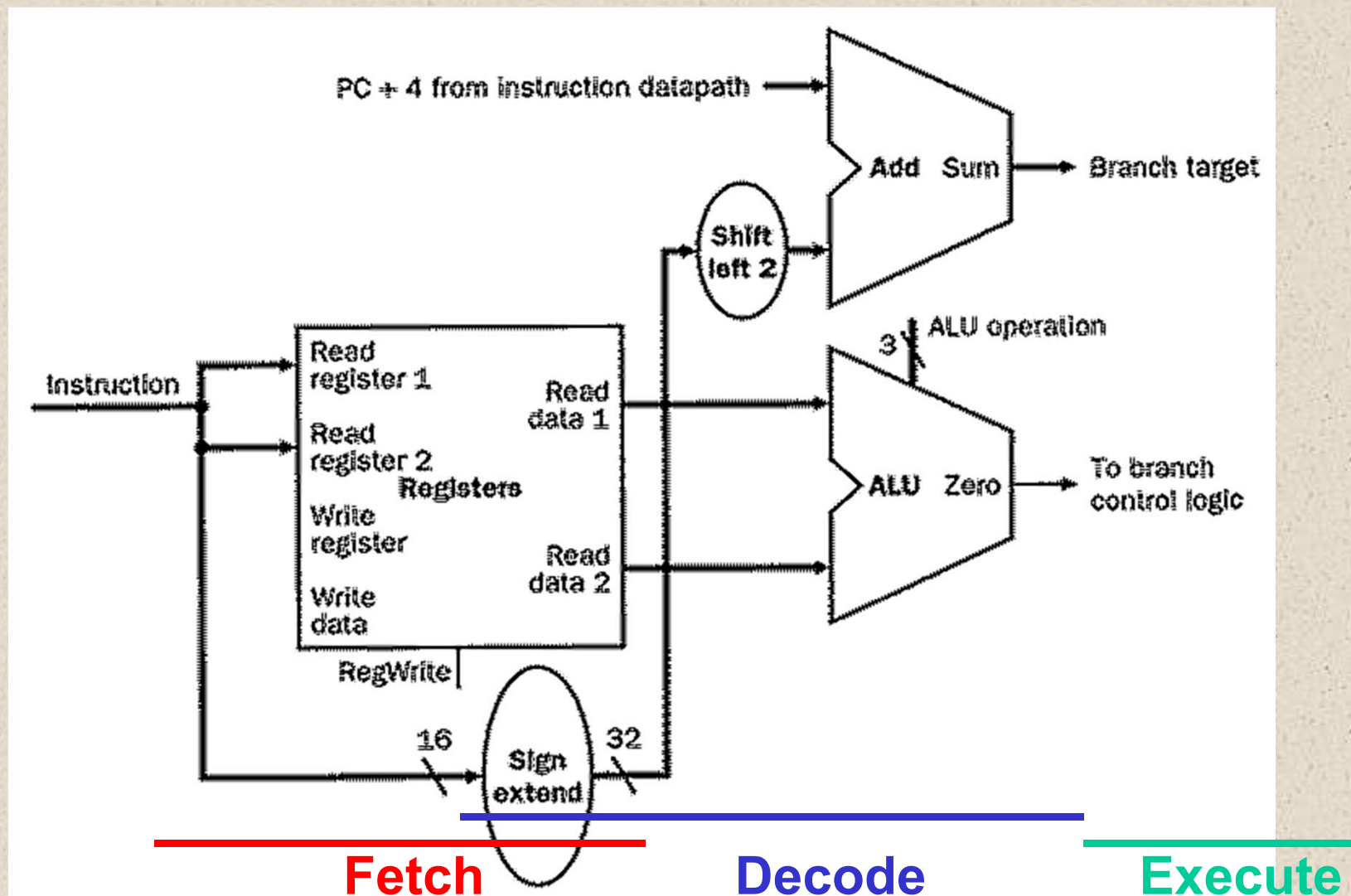
```
beq    $a1, $0, L
add    $v1, $v0, $0
add    $v1, $v1, $v1
j      Somewhere
L:     add    $v1, $v0, $v0
```

- Cílová adresa L leží tři *instrukce* za *beq*, z toho vyplývá obsah adresního pole (offsetu) 0000 0000 0000 0011.



- Délka instrukcí je čtyři byte. Proto je aktuální offset do paměti 12 bytů.

Komponenty: instrukce větvení

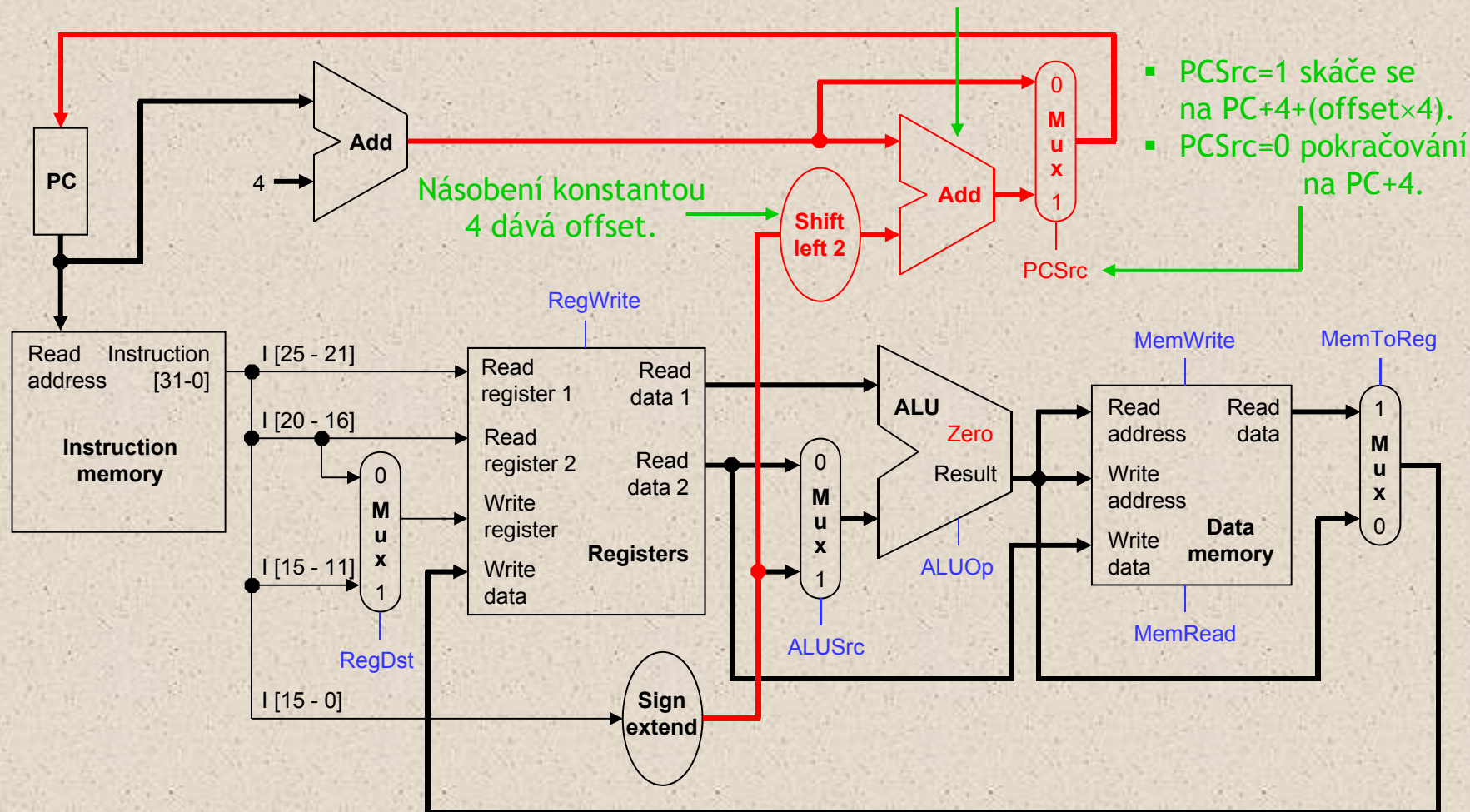


Provádění instrukce beq

1. Čtení instrukce, např. `beq $at, $0, offset` z paměti.
2. Čtení zdrojových registrů, `$at` a `$0`, z registrového souboru.
3. Odečtením v ALU se obsahy porovnají.
4. Je-li výsledek rozdílu 0 (signál ALU **zero**), zdrojové operandy byly stejné a PC musí být naplněn cílovou adresou $PC + 4 + (\text{offset} \times 4)$.
5. V opačném případě se skok nekoná a PC je pouze inkrementován na $PC + 4$ a čte se následující instrukce.

Hardware pro provádění skoků

Potřebujeme další sčítačku, protože ALU právě odčítá pro instrukci beq.



Řídící kódy ALU

ALU má dva řídící kódy (celkem = 5 bitů):

- 1) *ALUop* – vybírá specifickou operaci ALU
- 2) *Control Input* – vybírá funkci ALU

ALUop Input	Operation
00	load/store
01	beq
10	determined by opcode

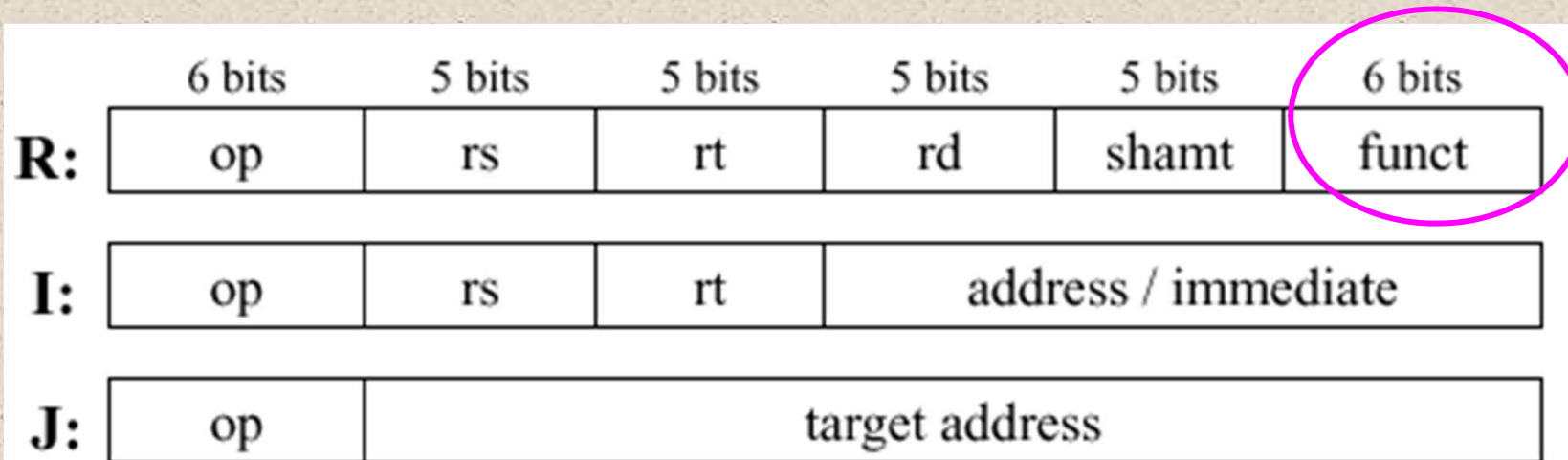
ALU Control Input	Function
000	and
001	or
010	add
110	sub
111	slt

Řídící bity ALU

ALU má následující řídící bity:

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Opakování: Instrukční formáty MIPS



op: basic operation of the instruction (opcode)

rs: first source operand register

rt: second source operand register

rd: destination operand register

shamt: shift amount

funct: selects the specific variant of the opcode (function code)

address: offset for load/store instructions ($\pm 2^{15}$)

immediate: constants for immediate instructions

MIPS instrukční pole - pravidla

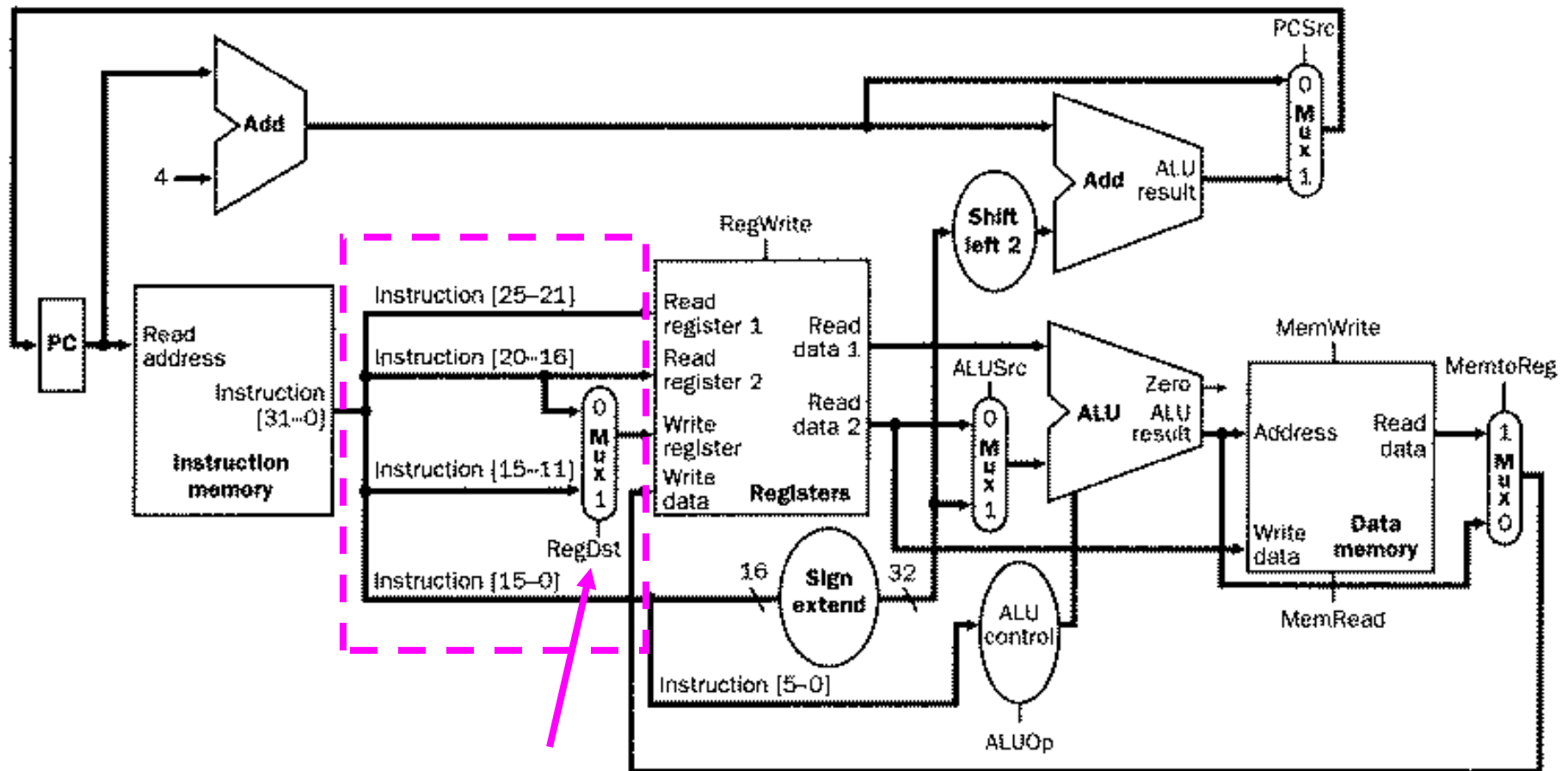
Všimněte si, že vždy platí:

- **Bity 31-26:** *opcode* – vždy v tomto místě
- **Bity 25-21 a 20-16:** *specifikace vstupů* – vždy v tomto místě

Dále podle typu instrukce platí:

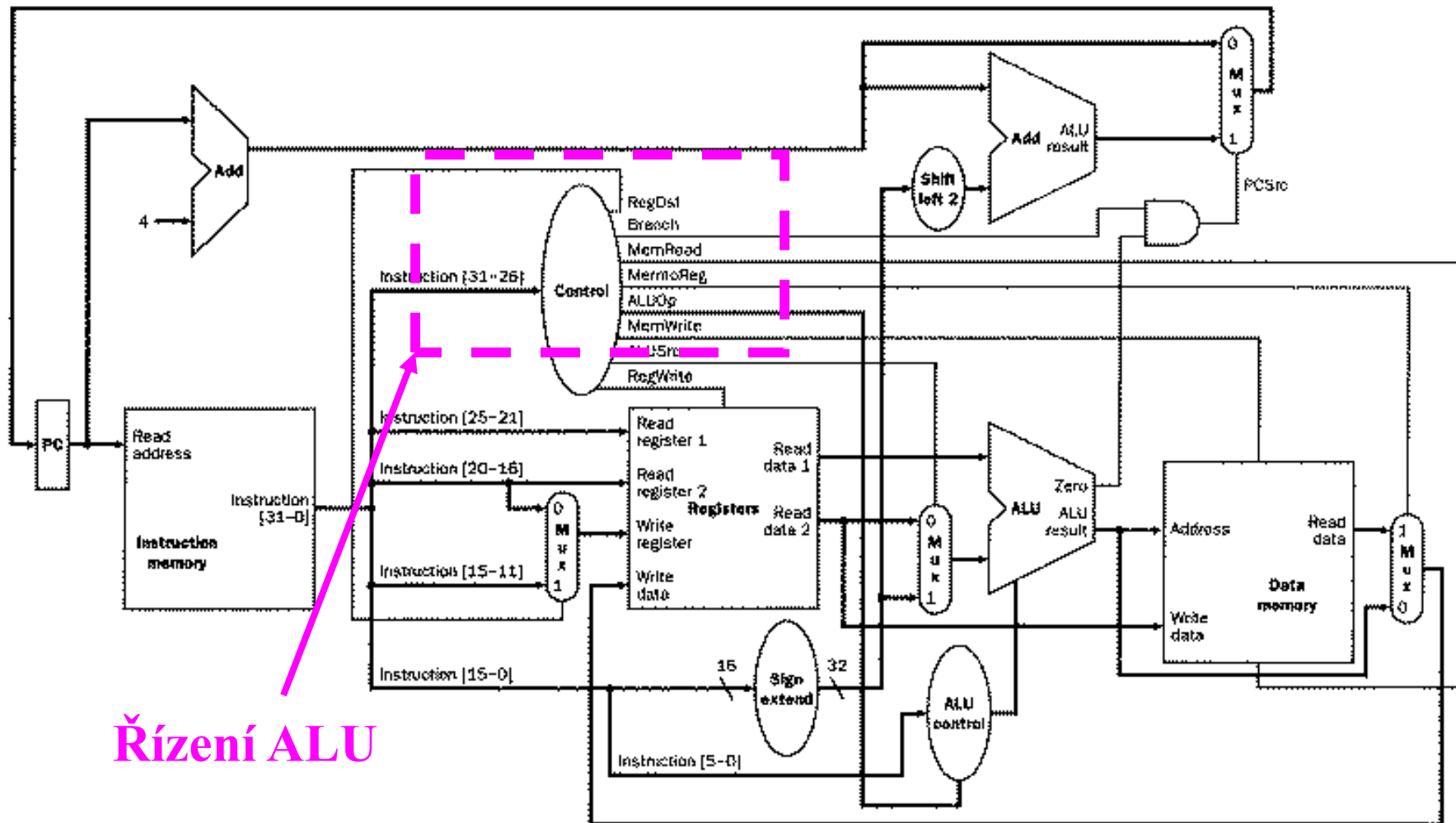
- **Bity 25-21:** *bázový registr* pro operace Load/Store – vždy v tomto místě
- **Bity 15-0:** *16-bitový offset* pro podmíněné skoky – vždy v tomto místě
- **Bity 15-11:** *registr určení* pro R-formát instrukcí – vždy v tomto místě
- **Bity 20-16:** *registr určení* pro operace Load/Store – vždy v tomto místě

Datové cesty - řízení zápisu do registrů



MUX navíc

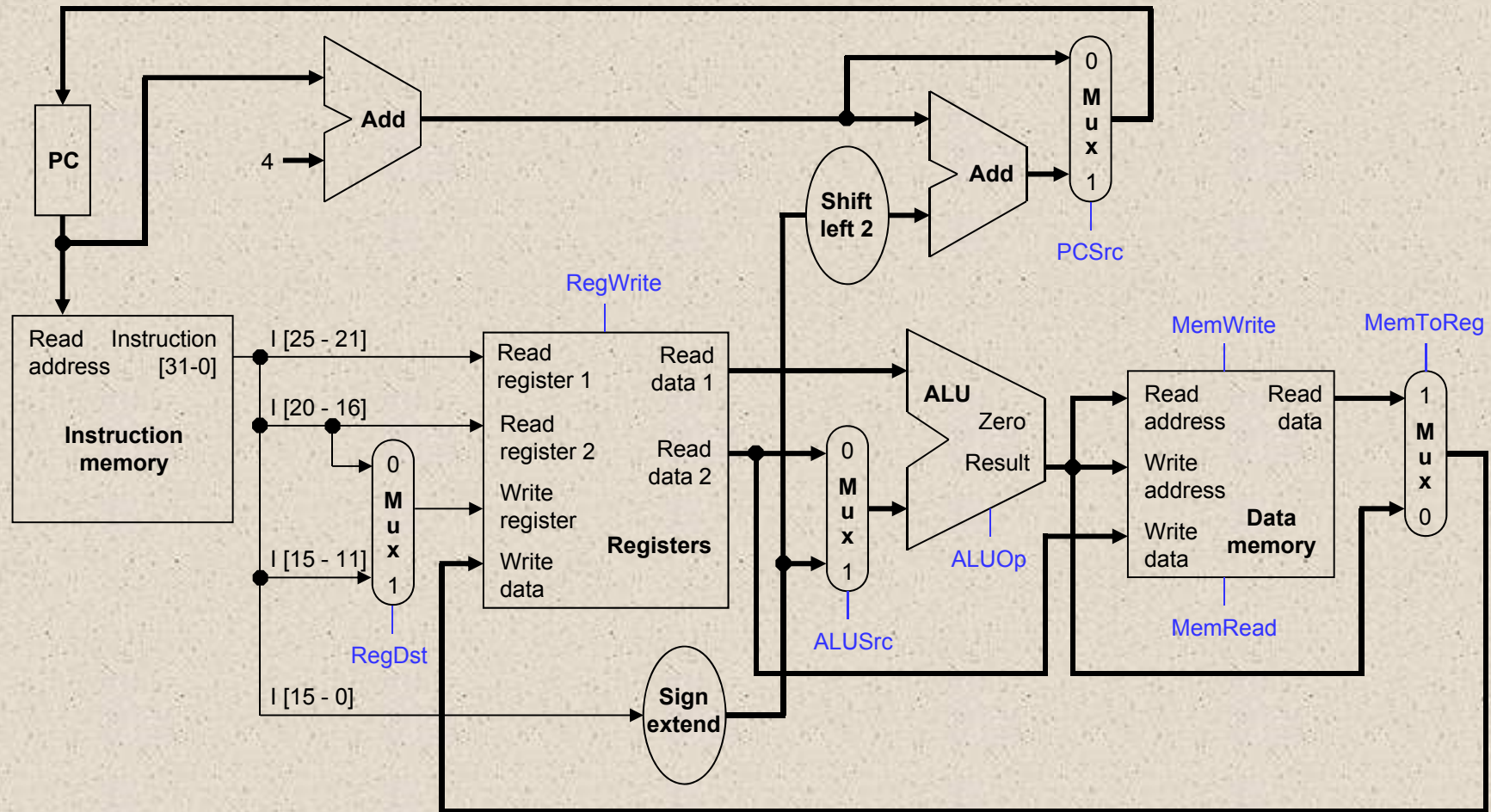
Datové cesty – řídicí signály



Řízení datových cest (souhrn)

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Výsledné datové cesty

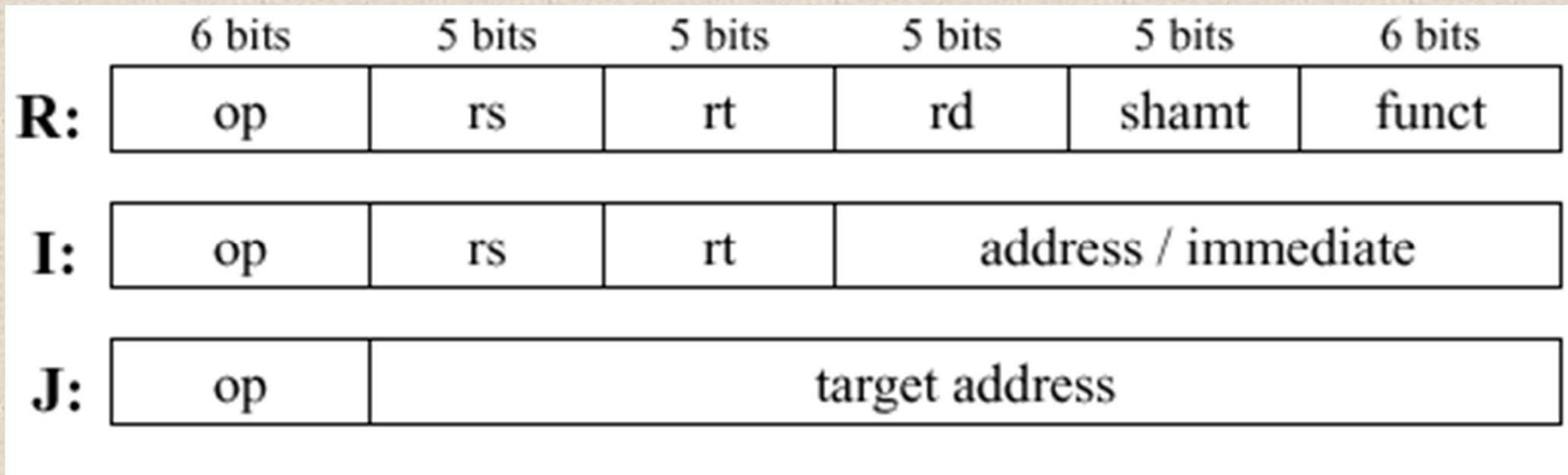


Rozšíření: instrukce Jump

Instrukce: j address

1. Načtení instrukce a inkrement PC
2. Čtení *adresy* z pole instrukce immediate
3. Cílová adresa skoku (JTA) má tyto bity:
 - **Bity 31-28:** horní čtyři bity PC+4
 - **Bity 27-02:** pole *immediate* instrukce skoku
 - **Bity 01-00:** Zero (00_2)
4. Mux řízený signálem **Jump** vybere JTA nebo cílovou adresu skoku jako nový obsah PC

Rozšíření: instrukce Jump

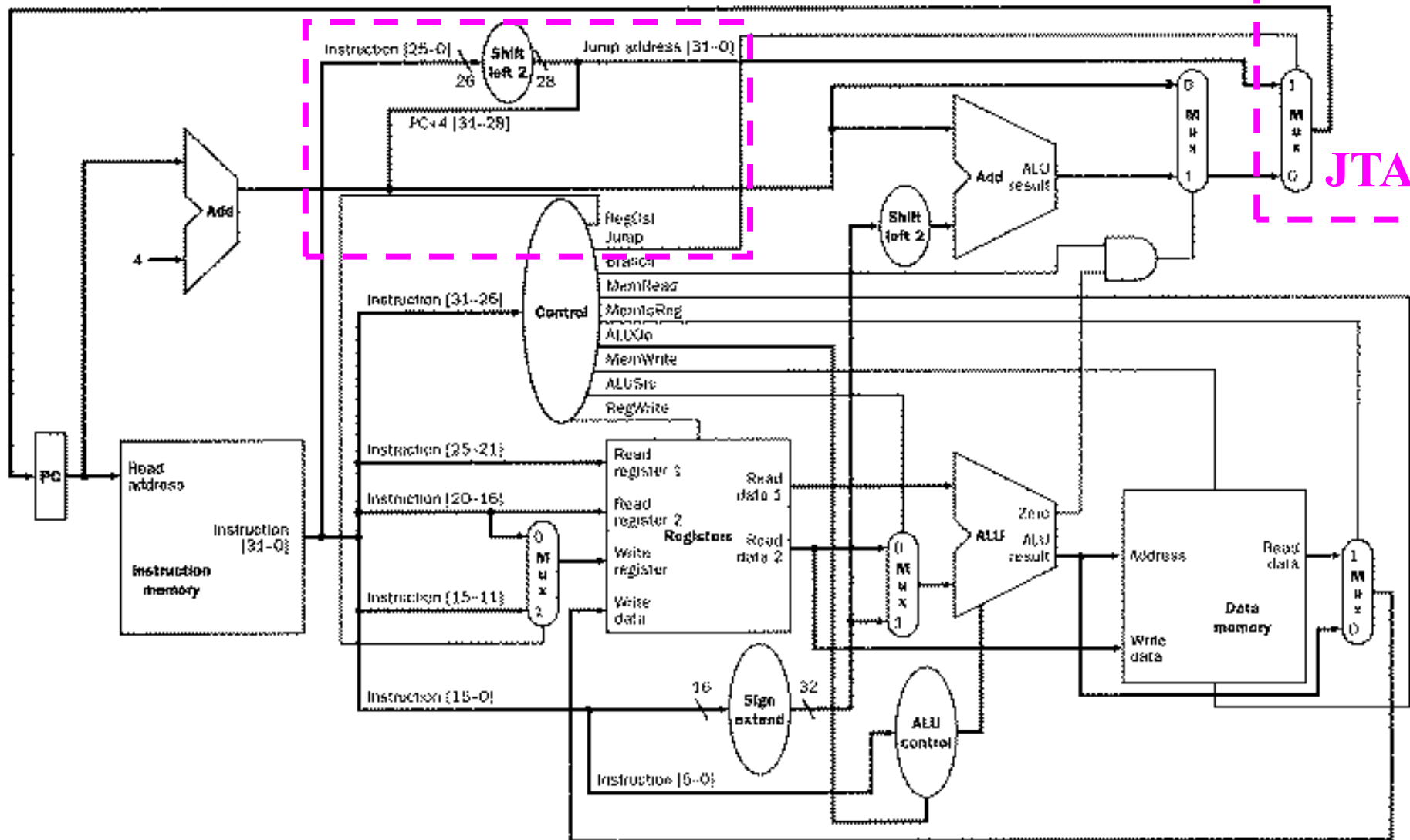


- **Bity 31-28:** Horní čtyři bity (PC + 4)
- **Bity 27-02:** pole **immediate** instrukce skoku
- **Bity 01-00:** Zero (002) - zarovnání slov

Rozšíření: instrukce **Jump**

Řízení skoků

Jump Target Address



Řízení

- **Řídící jednotka** zajišťuje generování řídicích signálů tak, že všechny instrukce se správně provádějí.
 - Vstupem řídicí jednotky je 32-bitové instrukční slovo.
 - Výstupem jsou řídicí signály v datových cestách (označené modře).
- Většina signálů je odvozována pouze z operačního kódu instrukce, nikoliv z celého 32-bitového slova.

Shrnutí jednocyklové implementace

- **Datové cesty** obsahují všechny funkční jednotky a spoje, potřebné pro implementaci dané ISA (Instruction Set Architecture).
 - Pro **implementaci s jednoduchým cyklem** jsme použili dvě oddělené paměti, ALU, několik sčítaček a řadu multiplexerů.
 - MIPS je 32-bitový procesor, a proto má většina sběrnic šířku 32-bitů.
- **Řídící jednotka** určuje činnost podle toho, jaká instrukce se právě vykonává.
 - Náš procesor má 10 **řídících signálů**, které ovládají datové cesty.
 - Řídící signály mohou být generovány kombinačními obvody, odvozenými od 32-bitového instrukčního slova.
- Dále se budeme zabývat omezením výkonu, které uvedená implementace s jednoduchým cyklem přináší.

Problémy

- Lze postavit jednotku operující v jednom cyklu?
 - Ano – “Návrh lze provést”
 - Všechny instrukce prováděny s CPI = 1 😊
 - Doba cyklu je diktována dobou ustálení všech obvodů 😞
 - Všechny operace trvají jako nejdelší operace, obvykle (load) 😞
 - *Vyšší efektivita software u architektury MIPS*

😞 Problémy s jednocyklovou jednotkou

- Zpoždění signálu odpovídá průchodu 1-5 prvků
- Není rozloženo do fází: Dokončení během 1 hodin taktu
- Maximální zpoždění = instrukce Load (5 komponent)
- **Zvětšuje dobu cyklu hodin**
- **Pokles výkonu** $t_{\text{cpu}} = IC * CPI * t_{\text{cyc}}$

Úvod do organizace počítače

Multicyklové jednotky

Přehled – vícecyklové zpracování

- Každá instrukce je zpracována ve více stupních
- Každý stupeň vykoná operaci během jednoho cyklu

1. Načtení instrukce
2. Dekódování instrukce / Načtení dat
3. Operace ALU / provedení instrukcí R-formátu
4. Dokončení provedení R-formátu
5. Provedení paměťového cyklu

Používají všechny instrukce

☺ Každý stupeň může opět použít hardware předchozího stupně

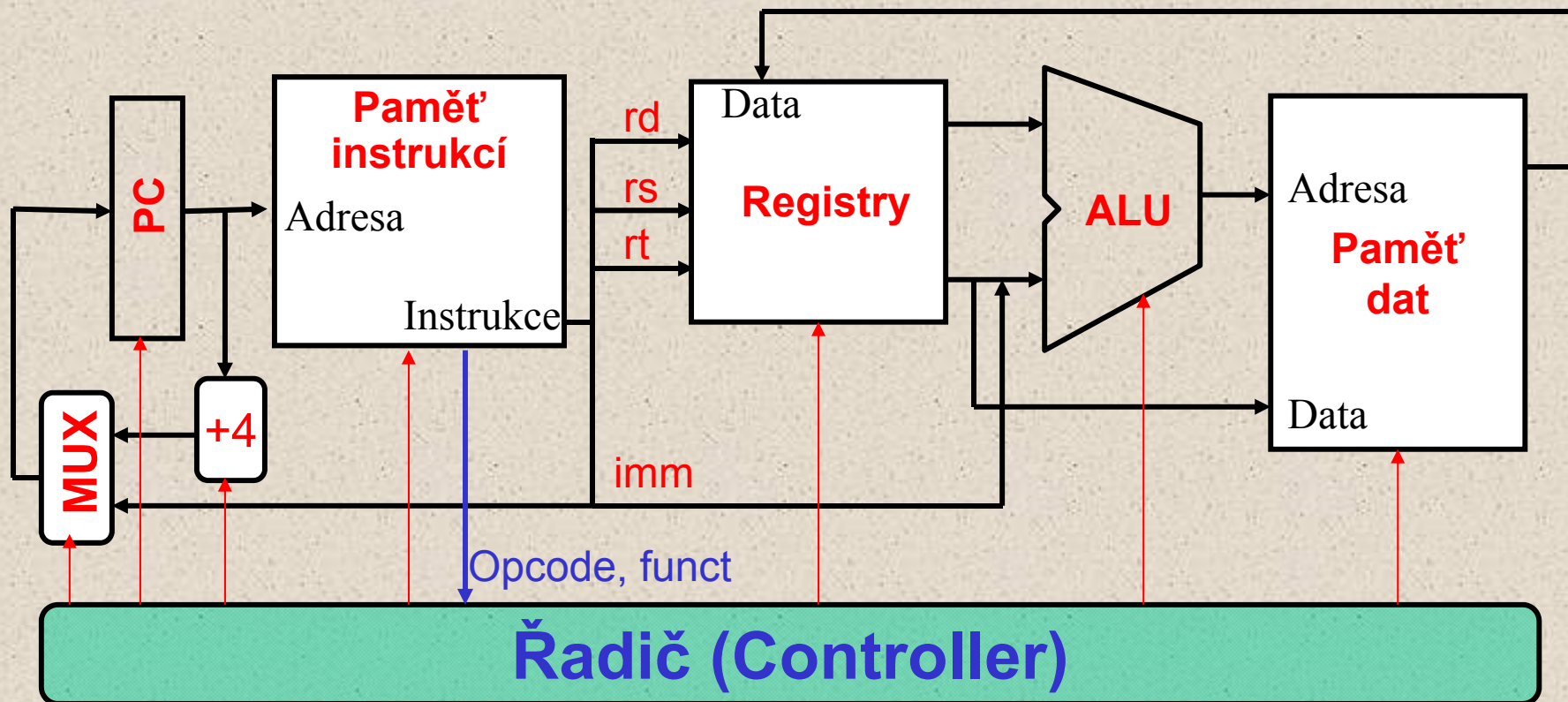
☺ Efektivnější využití hardware a času

>> **Nový hardware** vyžaduje registrový výstup

>> **Nové multiplexery (MUX)** pro opětné využití hardware

>> **Rozšíření řídicí jednotky (řadiče)** pro nový hardware

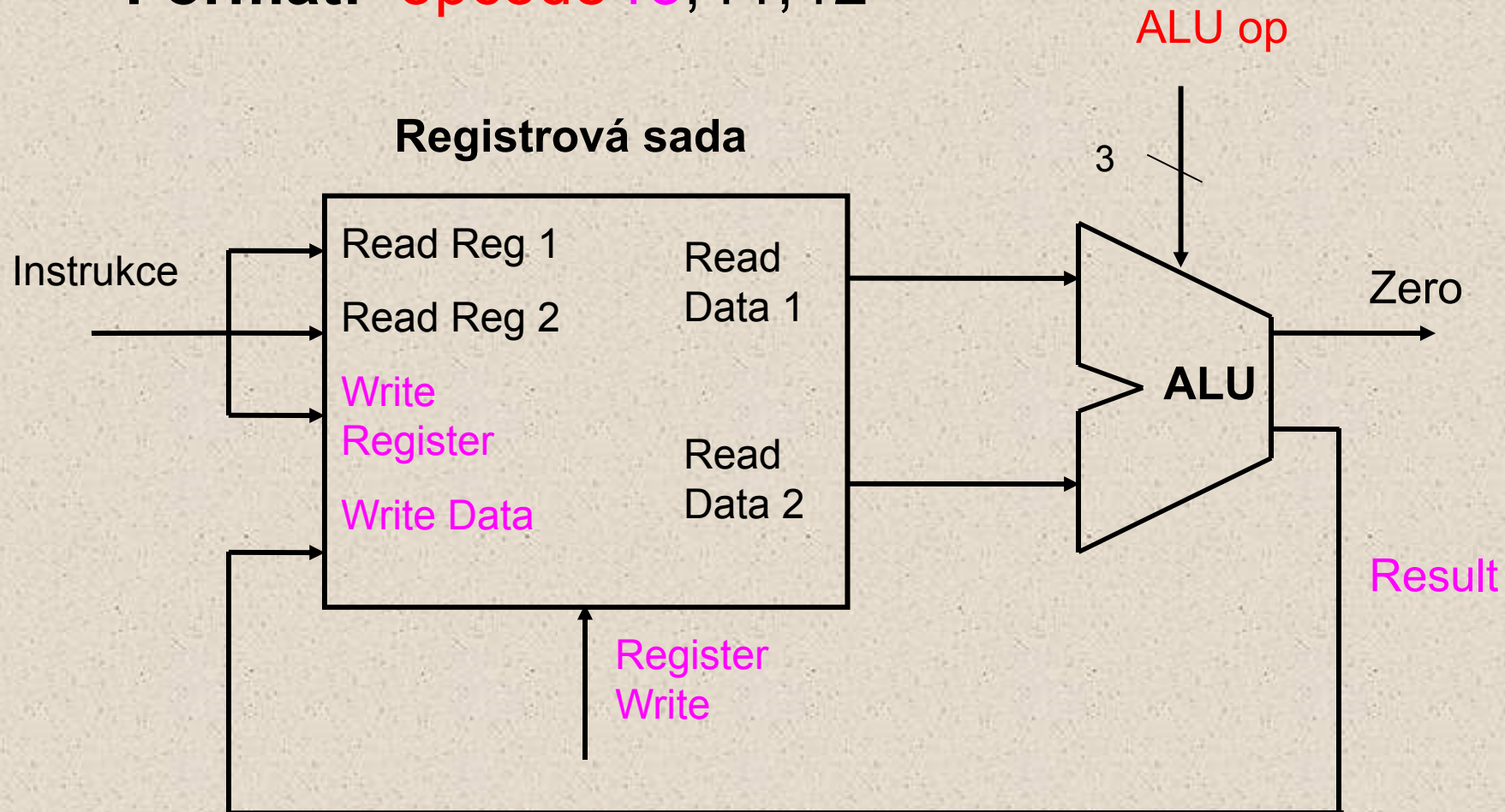
Opakování: Jednoduché datové cesty



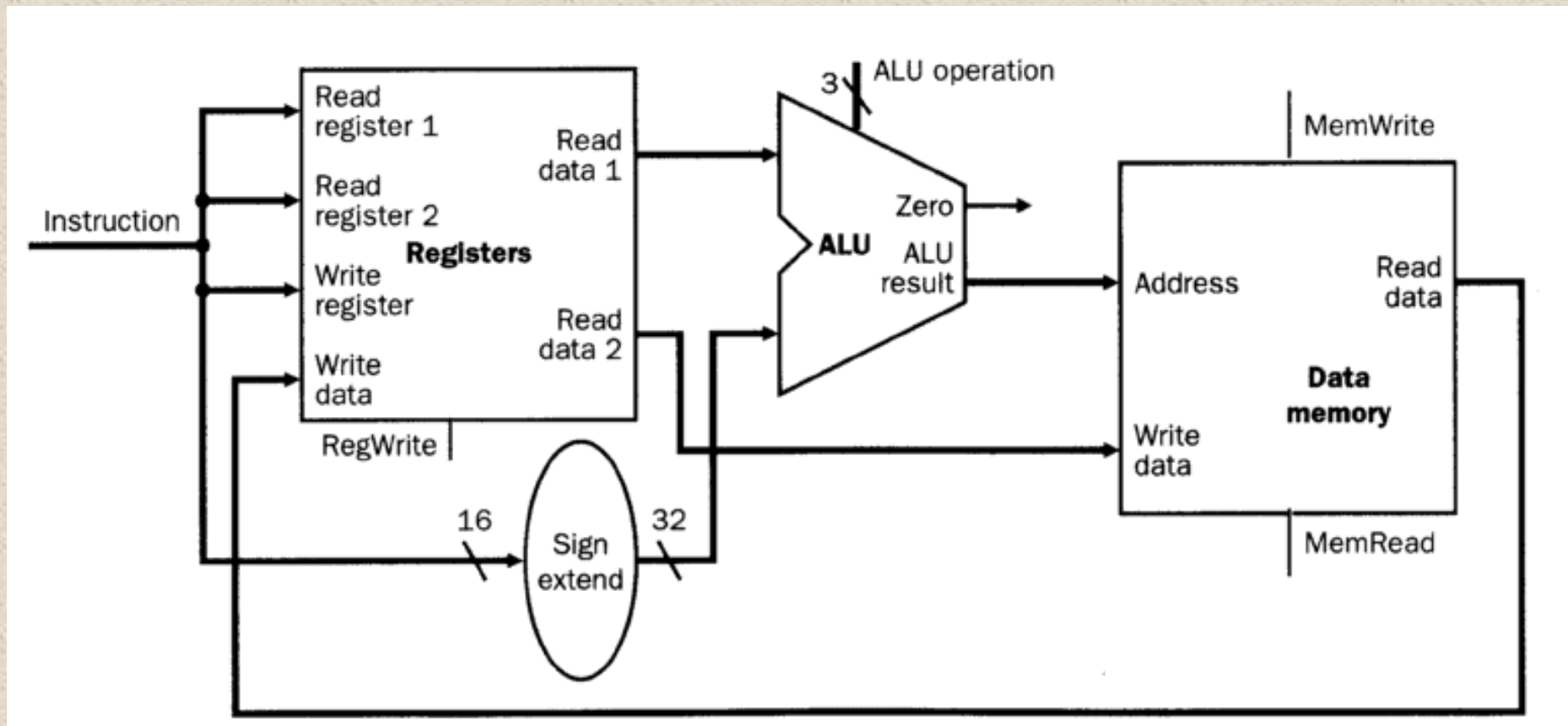
- **Datové cesty** umožňují přesuny obsahu registrů při provádění instrukcí
- Řízení zajišťuje provedení správných přesunů

Opakování: R-formát - cesty

- **Formát:** opcode r3, r1, r2



Opakování: Load/Store -cesty

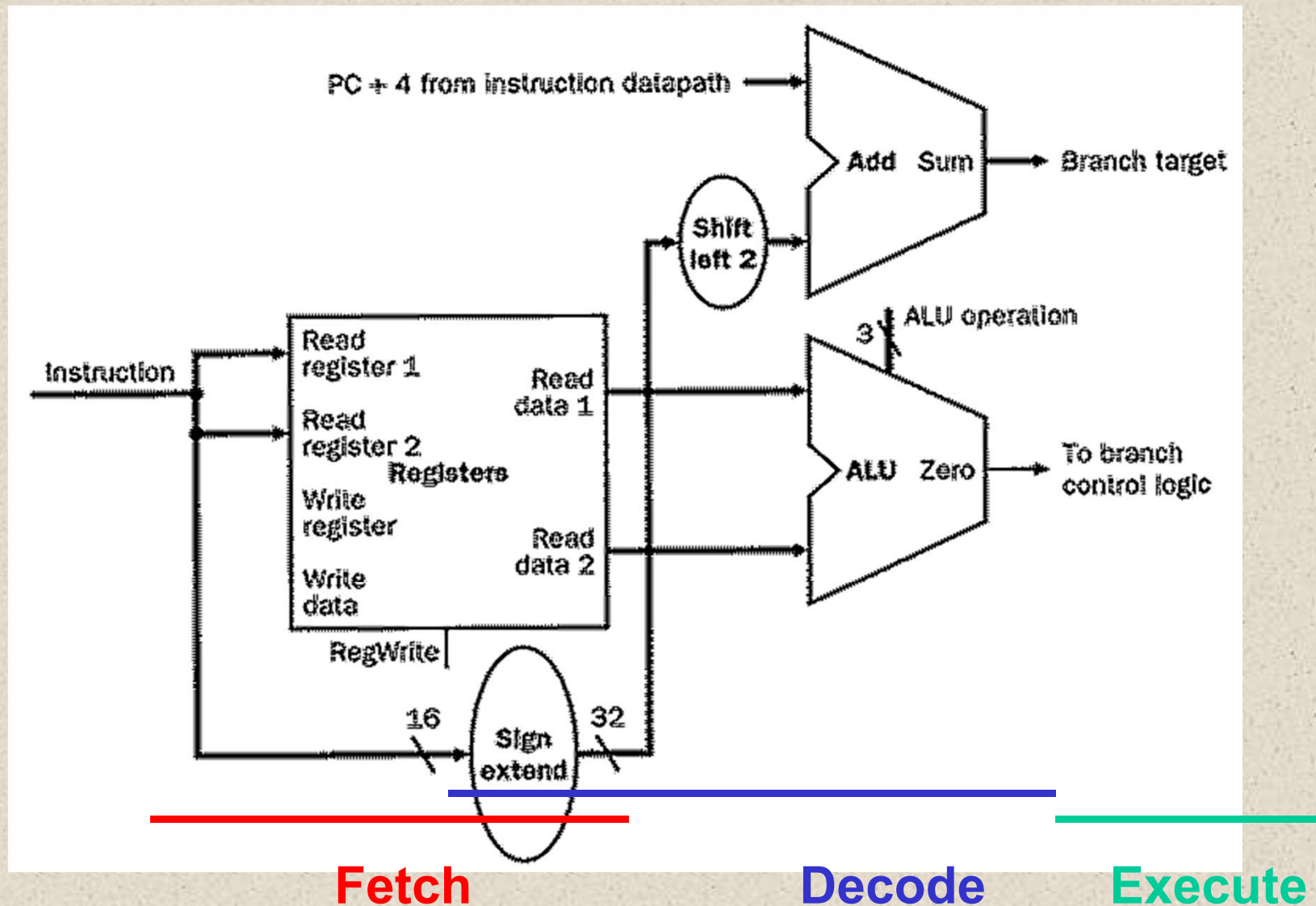


Fetch

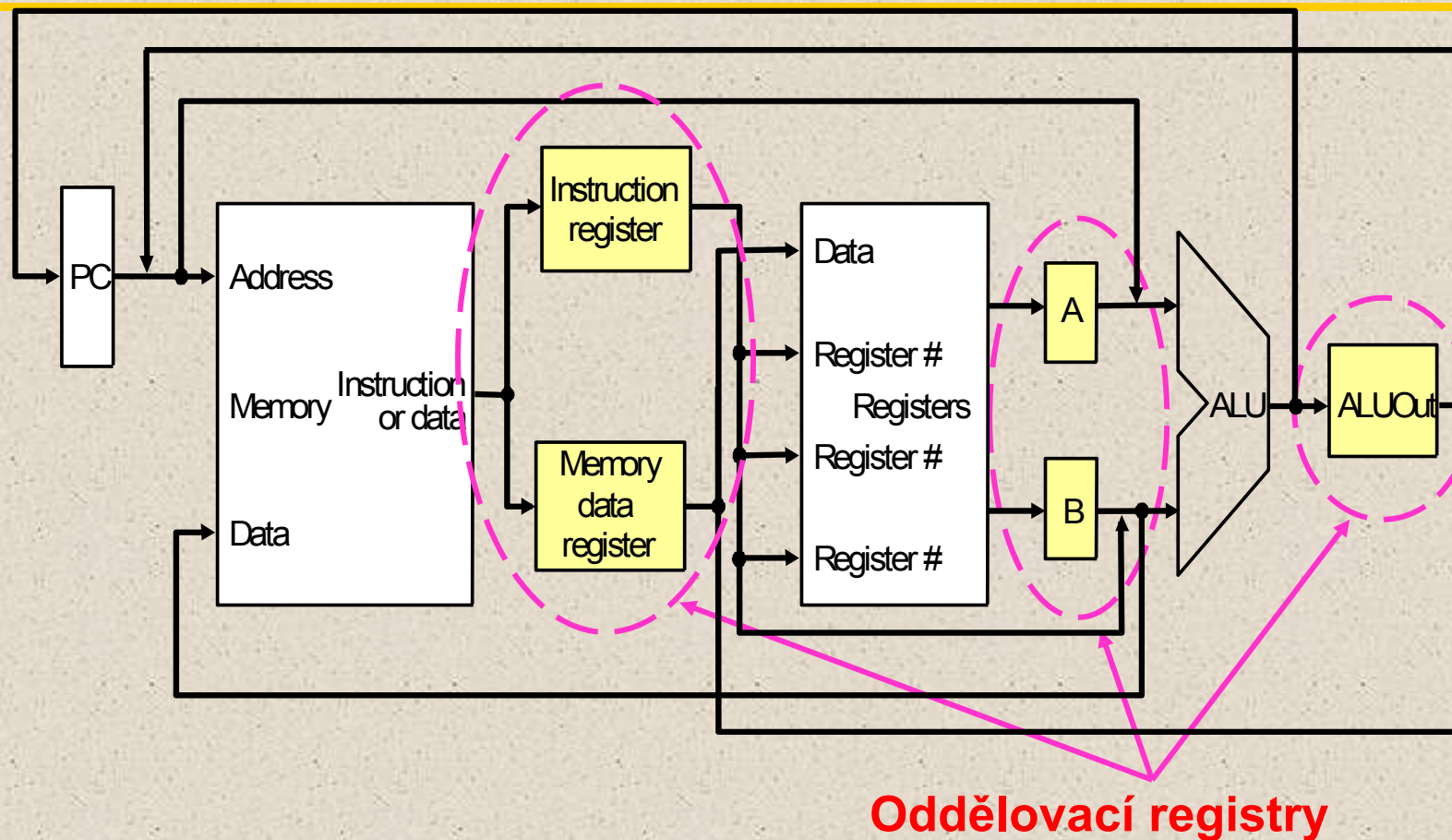
Decode

Execute

Opakování: Podmíněný skok - cesty



Princip: Vícecyklové datové cesty



Načtení instrukce

Dekódování/Načtení dat

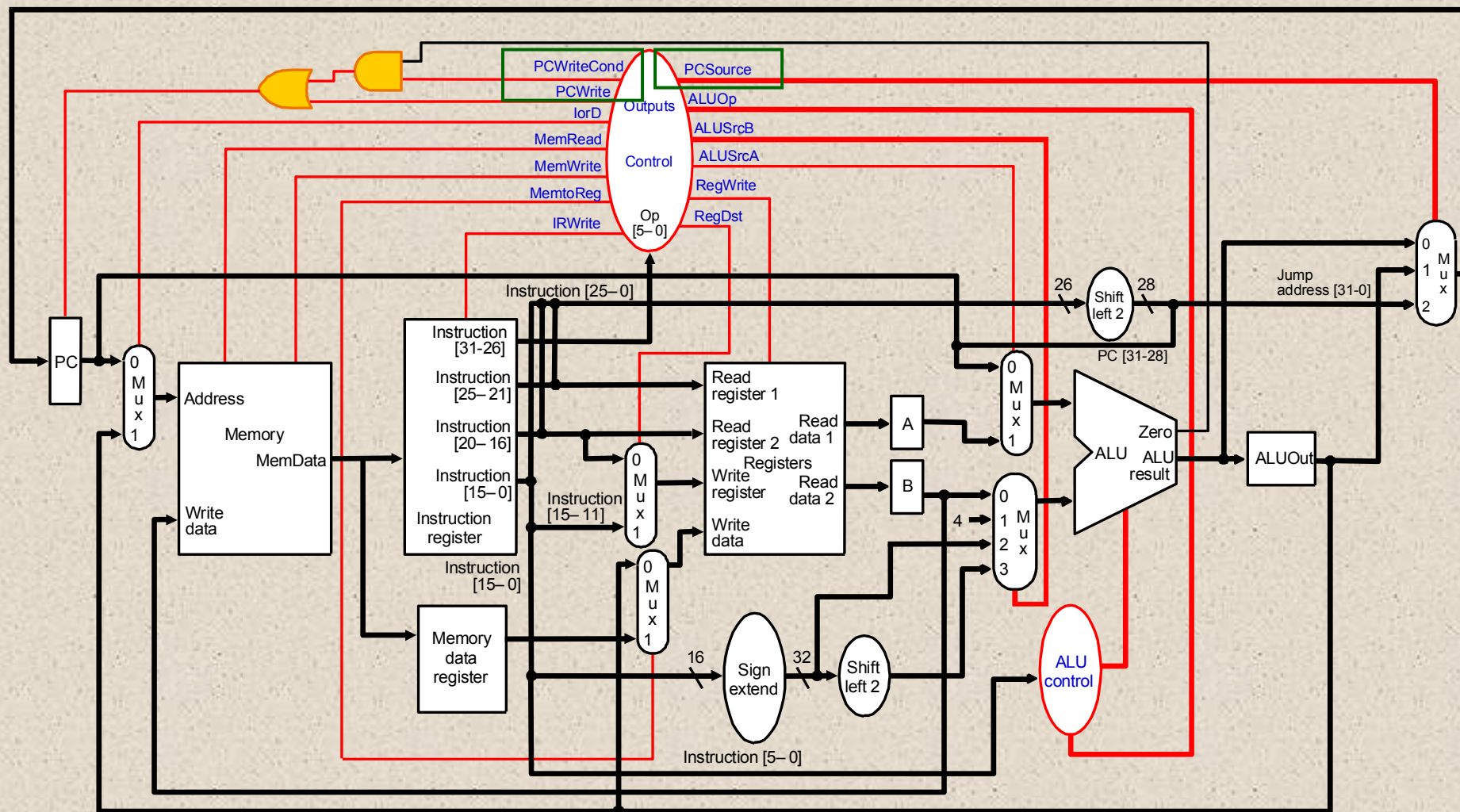
Provedení

Princip: Multicyklové datové cesty

Jak realizovat multicyklové jednotky (DP)???

1. Nahradíme **3 ALU** ve struktuře jednou ALU
2. Přidáme **jeden multiplexer** pro výběr vstupu ALU
3. Přidáme **jednu řídící linku** pro vstupní multiplexer ALU
 - Nové vstupy: Konstanta = 4 [PC + 4]
 - Sign-ext., posunutý offset [výpočet BTA]
4. Přidáme **temporary (buffer) registry**: (oddělovací registry)
 - MDR: Datový registr paměti
 - IR: Instrukční registr
 - A, B: Registr operandů ALU
 - ALUout: Výstupní registr ALU

Multicyklová jednotka (komplet)



Multicyklová jednotka: 1-bitové řídící signály

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lrd	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.




















Multicyklová jednotka: 2-bitové řídící signály

Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the Instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU ($PC + 4$) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address ($IR[25-0]$ shifted left 2 bits and concatenated with $PC + 4[31-28]$) is sent to the PC for writing.

Plánování činnosti multicyklové jednotky

Krok 1: Dekompozice provedení MC/DP na cykly

Krok 2: Ověřit, na které instrukce lze které cykly aplikovat

Jednocyklové kroky (akce)	R-fmt	lw	sw	beq	j
1. Načtení instrukce					
2. Dekódování instrukce / čtení dat					
3. Operace ALU/ provedení R-formátu					
4. Dokončení R-formátu					
5. Dokončení přístupu do paměti					

Multicyklová jednotka: R-formát

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole `opcode`, `rd`, `rs`, `rt`, `funct`

Načtení dat: Výběr registrů podle `rs`, `rt`

Data načtena do pomocných registrů `A`, `B` (vstup ALU)

Krok 3: Operace ALU (`ALUsrcA`, `ALUsrcB`, `ALUop`)

Výstup z ALU je zapsán do registru `ALUout`

Krok 4: Obsah registru `ALUout` jde na zápisový port registrové sady

V `rd` je zapsáno číslo registru

Aktivovány signály: `RegWrite`, `RegDst`

CPI pro R-formát = 4 cykly

Multicyklová jednotka: Store Word (sw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole *opcode*, *rs*, *rt*, *offset*

Načtení dat: *rt* adresuje registrovou sadu => Bázová adresa

Data načtena do buffer registru *A* (báze)

SignExt, Shift pole *offsetu* do buffer registru *B*

Krok 3: Operace ALU (*ALUsrcB*, *ALUop*) => Báze + Offset

výstup ALU jde do registru *ALUout*

Krok 4: Obsah registru *ALUout* je použit jako adresa do paměti

Aktivován signál: *MemWrite* [*ALUout* => reg. sadu]

CPI pro Store = 4 cykly

Multicyklová jednotka: Load Word (lw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole **opcode**, **rs**, **rt**, **offset**

Načtení dat: **rt** – pointer do registrové sady => Bázová adresa

Data načtena do buffer registru **A** (báze)

SignExt, Shift pole **offsetu** do buffer registru **B**

Krok 3: Operace ALU (**ALUsrcB**, **ALUop**) => Báze + Offset

výstup ALU jde do registru **ALUout**

Krok 4: Obsah registru **ALUout** je použit jako adresa do paměti

Aktivován signál: **MemRead**

Krok 5: Data z paměti jdou na zápisový port registrové sady,

adresa určena obsahem **rd** - proveden zápis

CPI pro Load = 5 cyklů

Multicyklová jednotka: Podmíněný skok

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole `opcode`, `rs`, `rt`, `offset`

Načtení dat: `rs` a `rt` určují adresy do sady registrů

Výpočet BTA: SignExt, Shift pole `offsetu` do buffer registru `B`

ALU určí PC, `offset` => BTA

Krok 3: Operace **ALU** (`ALUsrcA`, `ALUsrcB`, `ALUop`) = komparace
podle výstupu z ALU (registr `Zero`)
dojde k výběru BTA nebo PC+4

CPI pro podmíněný skok = 3 cykly

Multicyklová jednotka: Skok (Jump)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: Pole *opcode*, *address*

Výpočet JTA: Pole SignExt, Shift *offset* [Bity 27-0]

sestaveny z části PC [Bity 31-28] => JTA












Krok 3: Obsah PC nahrazen cílovou adresou skoku (JTA)

PCsource = 10, *Aktivován signál: PCWrite*

CPI pro Jump = 3 cykly

Závěr

- **MIPS ISA:** Tři instrukční formáty (R, I, J)
- Jeden cykl/stupeň, různé stupně na formát
- **Jednocyklové kroky (akce)**

	R-fmt	lw	sw	beq	j
1. Načtení instrukce					
2. Dekódování instrukce / čtení dat					
3. Operace ALU/provedení R-formátu					
4. Dokončení R-formátu					
5. Dokončení přístupu do paměti					

Cena: Komplikovanější návrh řízení