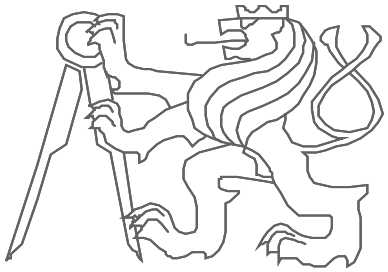
Architektura počítačů

Počítačová aritmetika a úvod

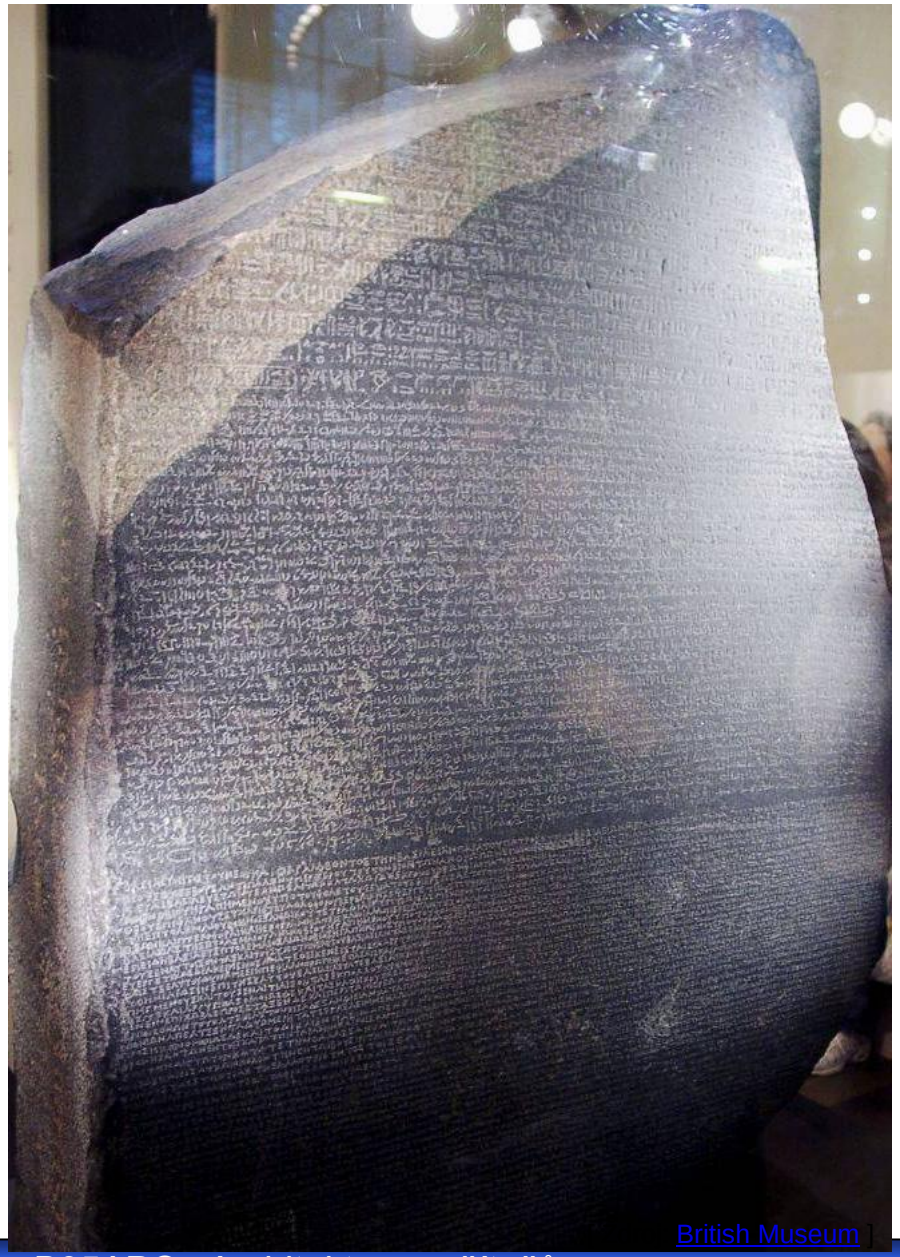
Richard Šusta, Pavel Píša



České vysoké učení technické, Fakulta elektrotechnická

Ver.3.1

Omluva

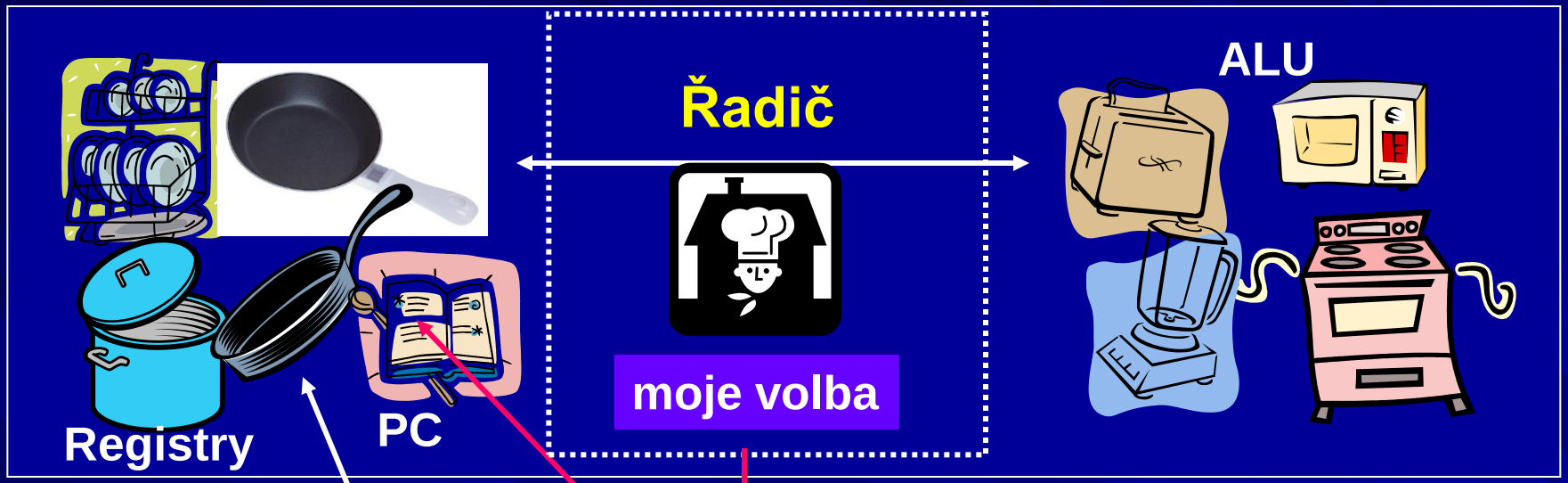


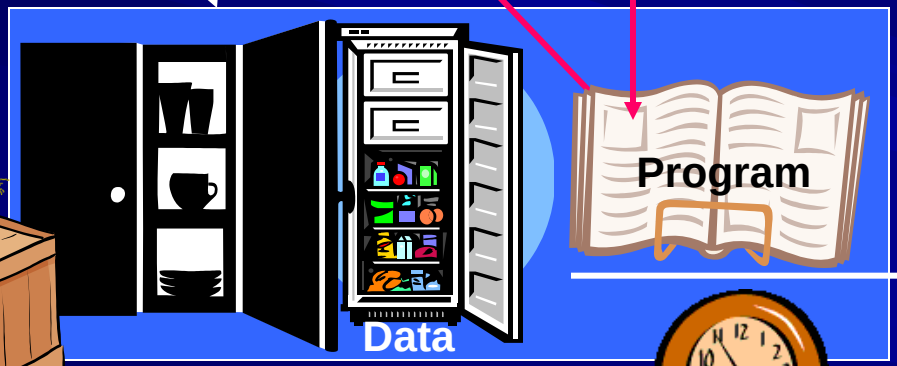
Na některých snímcích zůstane část v angličtině, protože vše nelze smysluplně nahradit českými termíny

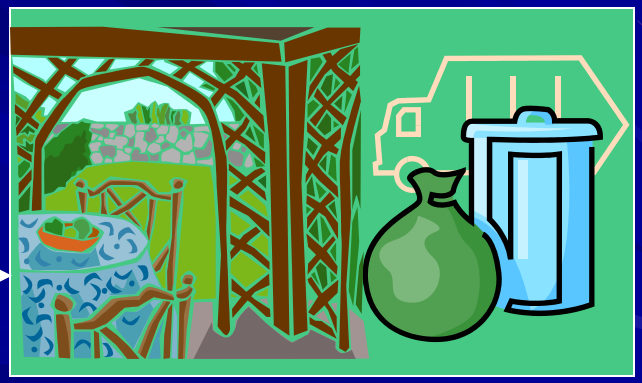
V procesoru

nepropadejte panice

Procesor se podobá kuchyni





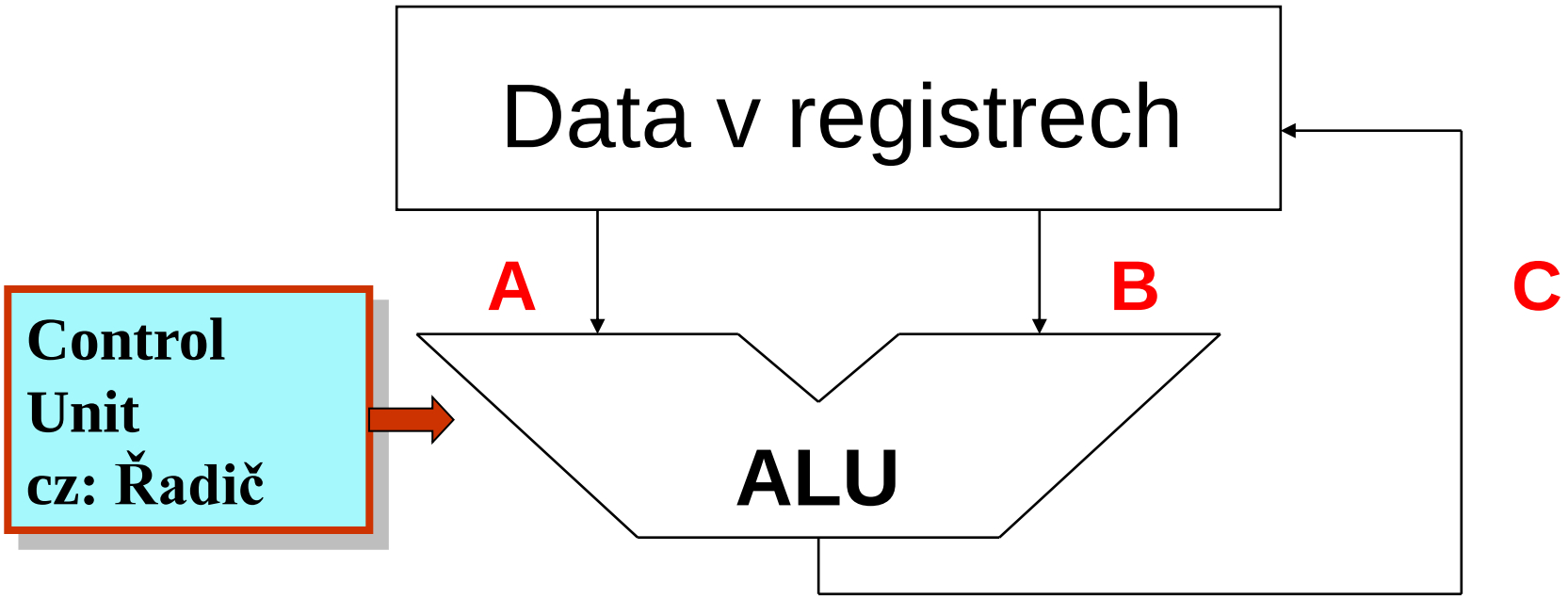


Paměť Časovač Výstup

Vstup

[Nelson: Computer Architecture and Design, Auburn 2008]

Jednoduchý procesor



Řadič (eng. Control Unit) nastavuje datovou cestu (eng: Datapath)

Výpočet provede ALU – Arithmetic Logic Unit
cz: Aritmeticko-logická jednotka

Počítač podle von Neumanna tvoří

Řadič
ALU
Paměť
Vstup
Výstup

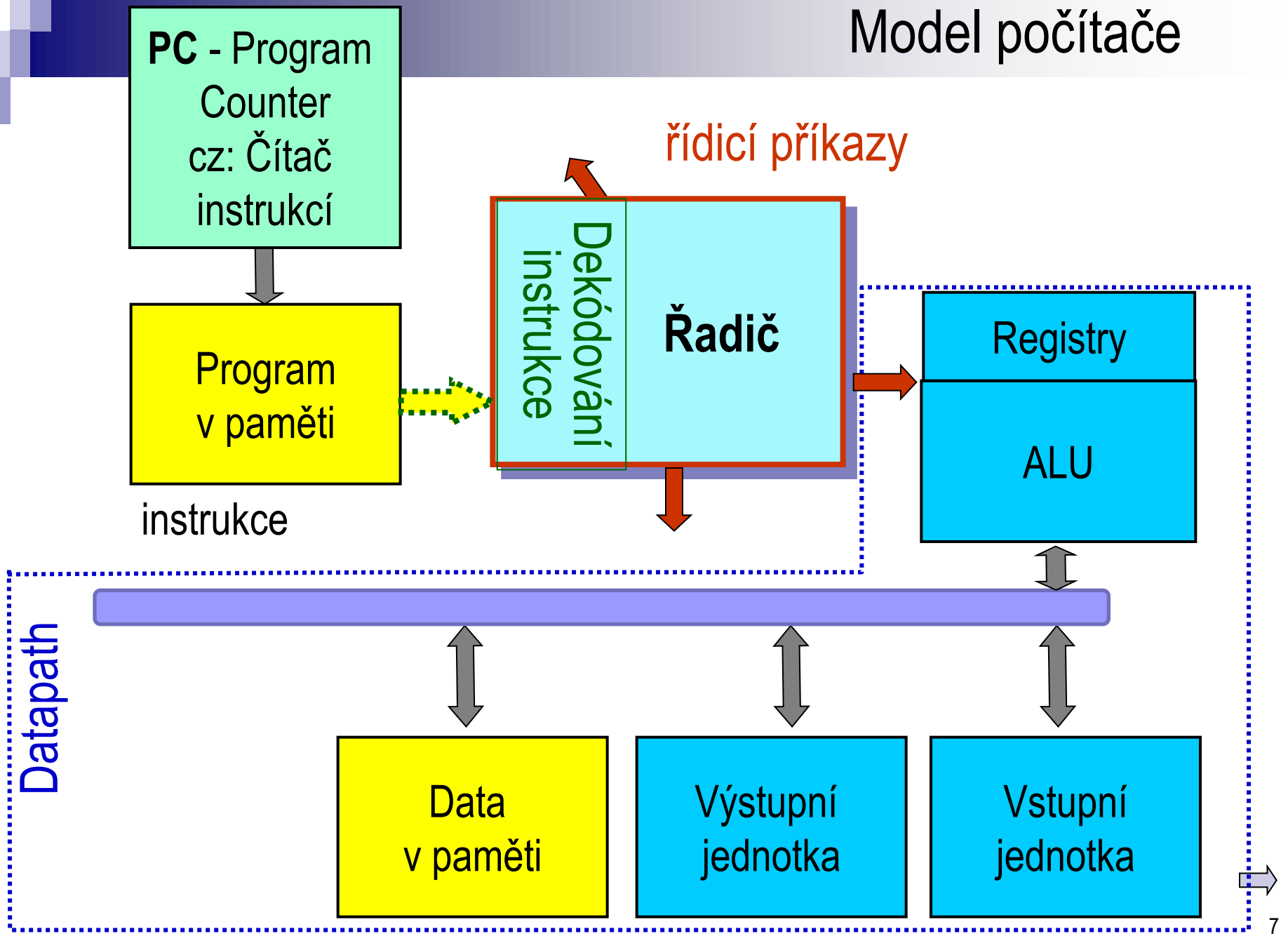
Procesor/mikroprocesor

I když je paměť programu společná s daty, instrukce a data se zpravidla čtou z jejích jiných částí.

V/V podsystém (V/V = I/O)

Řadič - součást (jednotka) počítače/procesoru, která jeho činnost řídí. Obsahuje registry potřebné k dekódování instrukce a vlastní řídicí část. Často se realizuje jako konečný automat – ten bude probrán v jiných předmětech.

Model počítače



RISC návrhová strategie

RISC - Reduced Instruction Set Computer

Zásada – vše se má držet jednoduché!

- **pevná délka instrukce** (obvykle 1 slovo - word);
- **load-store** instrukce nedělají nic jiného;
- **omezené možnosti adresace paměti;**
- **limitované sady instrukcí.**

Cíle návrhu:

rychlost ← *malý procesor, nízký příkon, spolehlivost*
nízká cena ← *návrh, výroba, testování, pouzdro*
malá spotřeba křemíku ← *vestavěné (embedded) systémy*

Příklady RISC procesorů: MIPS, NIOS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq), Alpha…

Zkratka MIPS v počítačích

MIPS procesor - **M**icroprocessor without **I**nterlocked **P**ipelined **S**tages

cz: Procesor nevyžadující prodlevy při zřetězeném zpracovaní instrukcí (eng:pipeline)

MIPS rychlost procesoru – **M**illion **I**nstructions **P**er **S**econd

Starší metoda měření rychlosti mající dnes význam jen u procesorů, které vykonají každou instrukci během pevného počtu cyklů hodin. Kritici MIPS údaje překládají zkratku jako Meaningless Indicator of Performance.

CISC návrhová strategie

Complex **I**nstructions **S**et **C**omputers

Příklady procesorů

Procesor	Instrukce	Činnost
Pentium	**MOVS**	*Přemístění pole dat na jinou adresu*
PowerPC	**cntlzd**	*Počítá délku bloku nul*
IBM 360-370	**CS**	*Porovná, a při splněné podmínce prohodí registry*
Digital VAX	**POLYD**	*Výpočet hodnoty polynomu podle tabulky koeficientů*

Registry

- **Registry tvoří hlavní operandy asembleru**
  - Registry se fyzicky umísťují co nejblíže ALU kvůli maximálně rychlému přístupu k nim a realizují se statickými pamětmi.
  - Operace lze provést pouze s registry! Některé CISC instrukce sice operují s hodnotami v paměti, ale ty musí stejně napřed načíst do registrů.
- **MIPS má 32 bitové registry**
  - číslované od $0 do $31
  - na každý registr lze odkázat číslem či jeho jménem.

MIPS: Registry a jejich ustálené použití

Reg	Name	Normal usage
$0	**$zero**	0x0000_0000 – pouze ke čtení!
$1	**$at**	Assembler Temporary
$2	**$v0**	Argumenty a návratové hodnoty funkcí
$3	**$v1**	
$4	**$a0**	
$5	**$a1**	
$6	**$a2**	
$7	**$a3**	
$8	**$t0**	Temporary - pomocné výpočty, není nutné zachovat jejich obsah
$9	**$t1**	
$10	**$t2**	
$11	**$t3**	
$12	**$t4**	
$14	**$t5**	
$14	**$t6**	
$15	**$t7**	

Reg	Name	Normal usage
$16	**$s0**	Chráněné hodnoty – nutno zachovat jejich obsah
$17	**$s1**	
$18	**$s2**	
$19	**$s3**	
$20	**$s4**	
$21	**$s5**	
$22	**$s6**	
$23	**$s7**	
$24	**$t8**	
$25	**$t9**	
$26	**$k0**	Přerušení
$27	**$k1**	
$28	**$gp**	Global Pointer
$29	**$sp**	Stack Pointer
$30	**$fp**	Frame Pointer
$31	**$ra**	návratová adresa

rezervování pro OS

Konstanta 0 v $0 ($zero)

MIPS registr $0 ($**zero**) má konstantní neměnnou hodnotu **0**

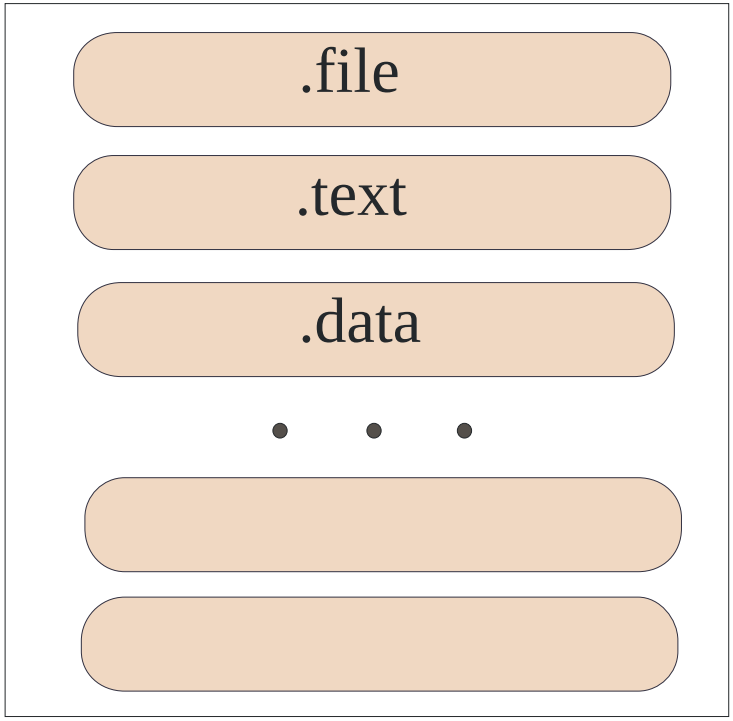
Používá se na vytvoření dalších operací jako třeba

add $8, $9, $zero // $8 ← $9+$0

// provede operaci

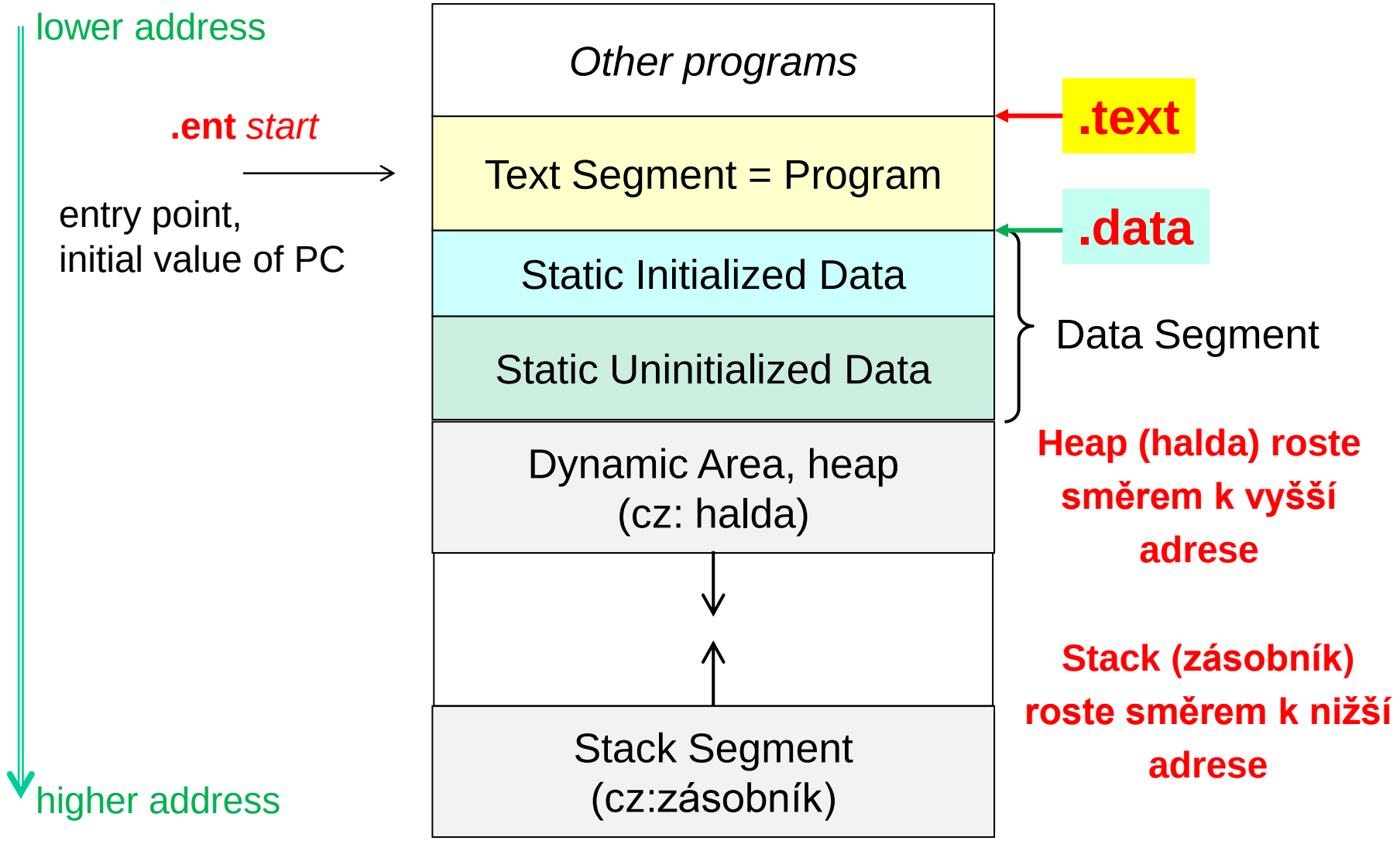
$8 ← $9 // do $8 se uloží $9

Zdrojový soubor assembleru



- se vnitřně dělí na několik sekcí;
- každá jeho sekce může obsahovat instrukce (direktiva .text) nebo data (direktiva .data)

Opakování: Uložení programu v adresovém prostoru



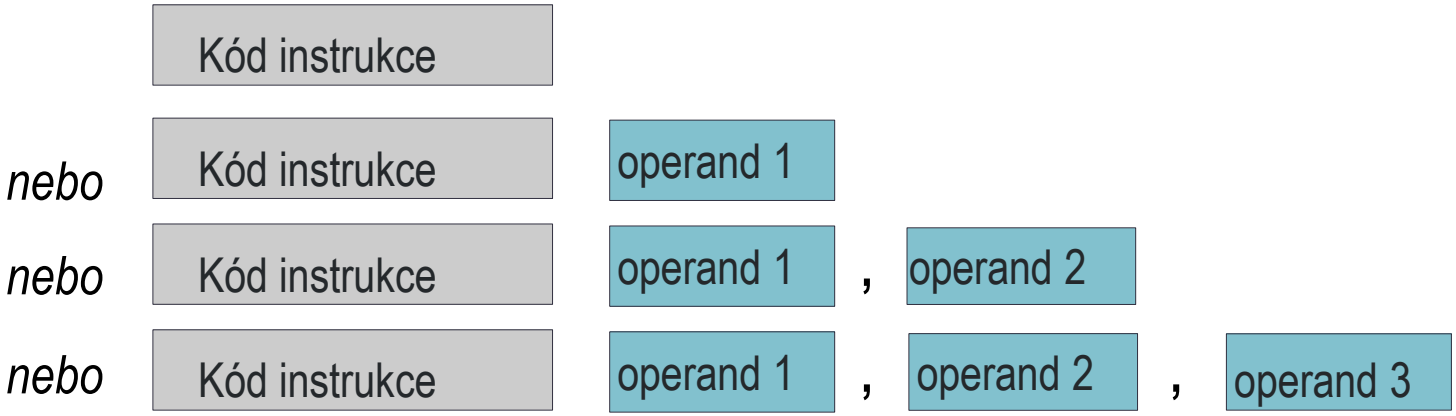
Kód assembleru

- Existuje mnoho assemblerů, jde o interní kód daného typu procesoru;
- Každý jeho příkaz zpravidla začíná na samostatné řádce;

Instrukce assembleru se dělí na tři kategorie

1. Executable Instructions – výkonné instrukce
   - generují strojový kód procesoru, tedy seznam povelů.
2. Pseudo-Instructions and Macros – pseudo-instrukce a makra
   - překládají se pomocí jedné či několika instrukcí assembleru,
   - zjednodušují psaní assembleru.
3. Assembler Directives – direktivy assembleru
   - jsou „non-executable“ (nevýkonné) – nepatří do strojového kódu procesoru
   - poskytují informaci překladači, modifikují způsob překladu.
   - definují segmenty a alokuji paměť proměnným, apod.

Struktura MIPS instrukcí



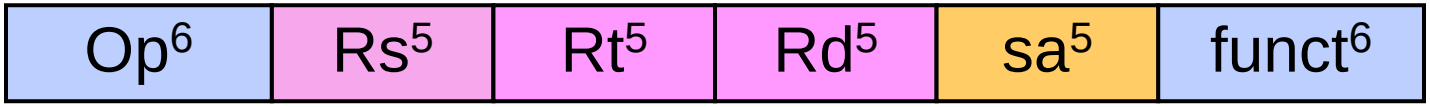
- kód - textový identifikátor instrukce
- operandy jsou typů
  - registr procesoru, tedy $0 až $31;
  - určení umístění v paměti;
  - konstanty (též označované jako „immediate“).

Formáty MIPS instrukcí

Všechny MIPS instrukce mají délku 4 byty (32 bitů).
V obrázcích určují horní indexy určují délku pole v bitech.

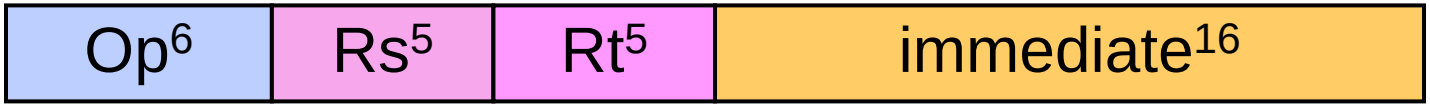
Register (R-Type) „Register-to-register“ formát

Op je operační kód definující typ instrukce,
Rs, Rt, Rd označují čísla registrů, Rs a Rt-zdrojové, a Rd-cílový
funct- sub-function - pomocné řídicí kódy
sa – rezervováno na specifikaci délky posunu (shift) a rotace.



Immediate (I-Type)

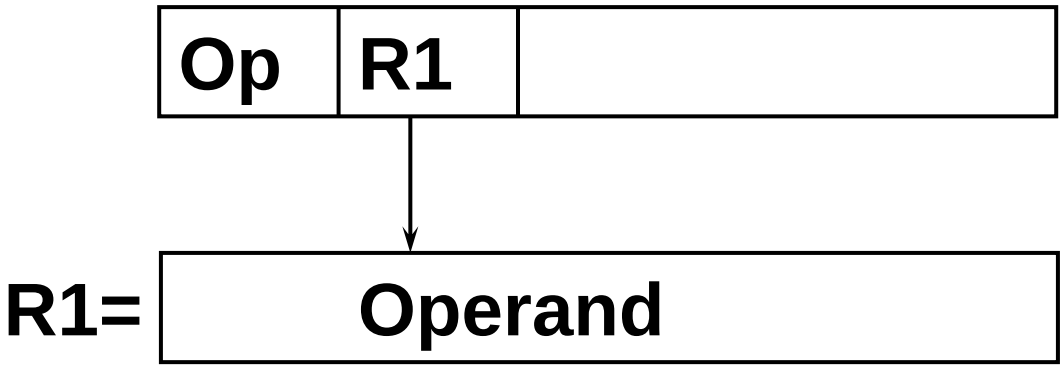
16-bitová konstanta uložená v instrukci, Rt – cílový registr



Jump (J-Type) používá pouze jediná skoková instrukce

MIPS adresovací módy 1/2

(a) Register direct addressing - *registr obsahuje hodnotu*



or $1, $2, $3

(b) Immediate addressing – *instrukce obsahuje číslo*

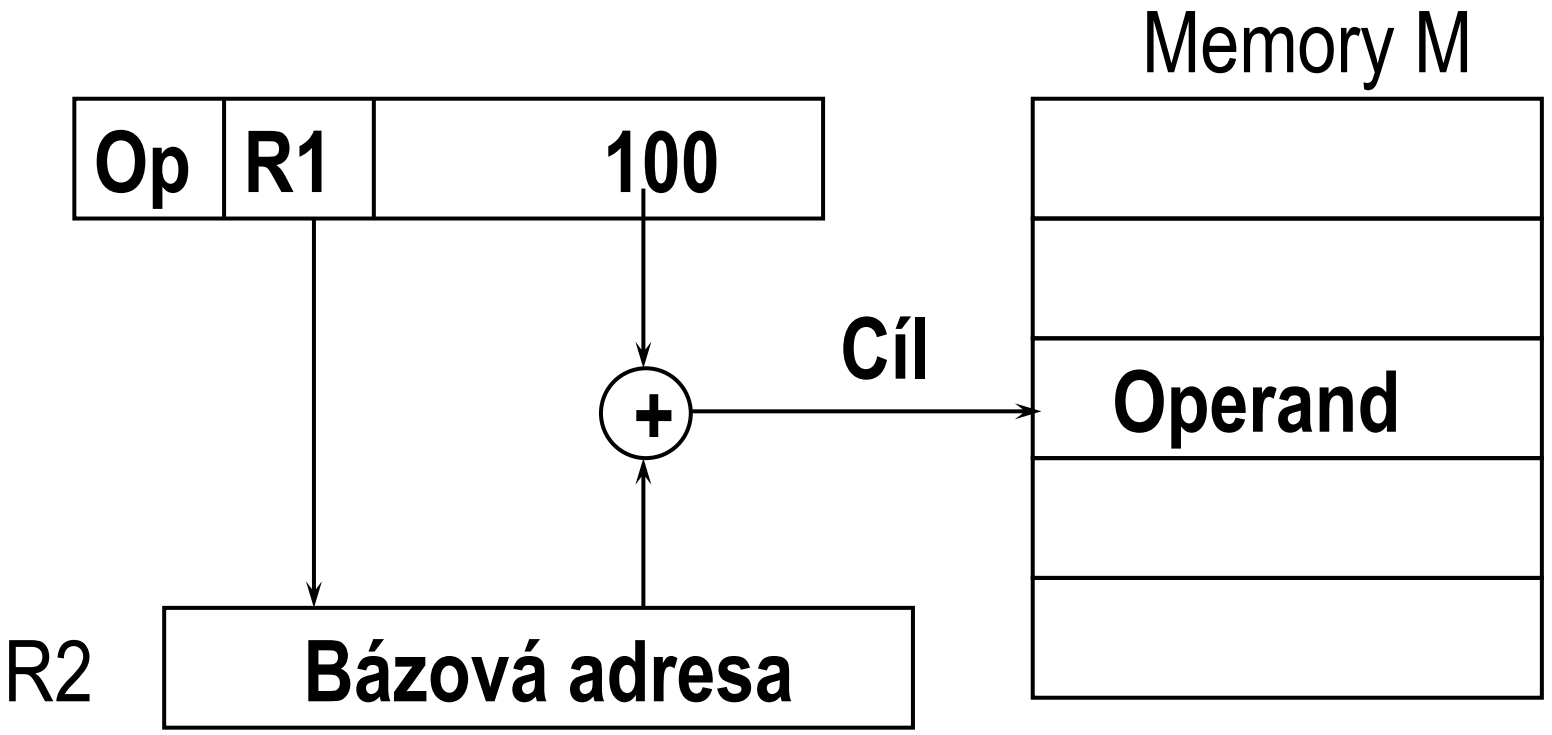
addi $1, $2, -20

MIPS adresovací módy 2/2

sw R1, byte_offset(R2)

sw $1, 100($2) má význam **$1** → Memory[**$2**+100]

- (c) Displacement (or offset) addressing – též zvané bázové adresování, kdy adresa je součet registru a +/- offsetu



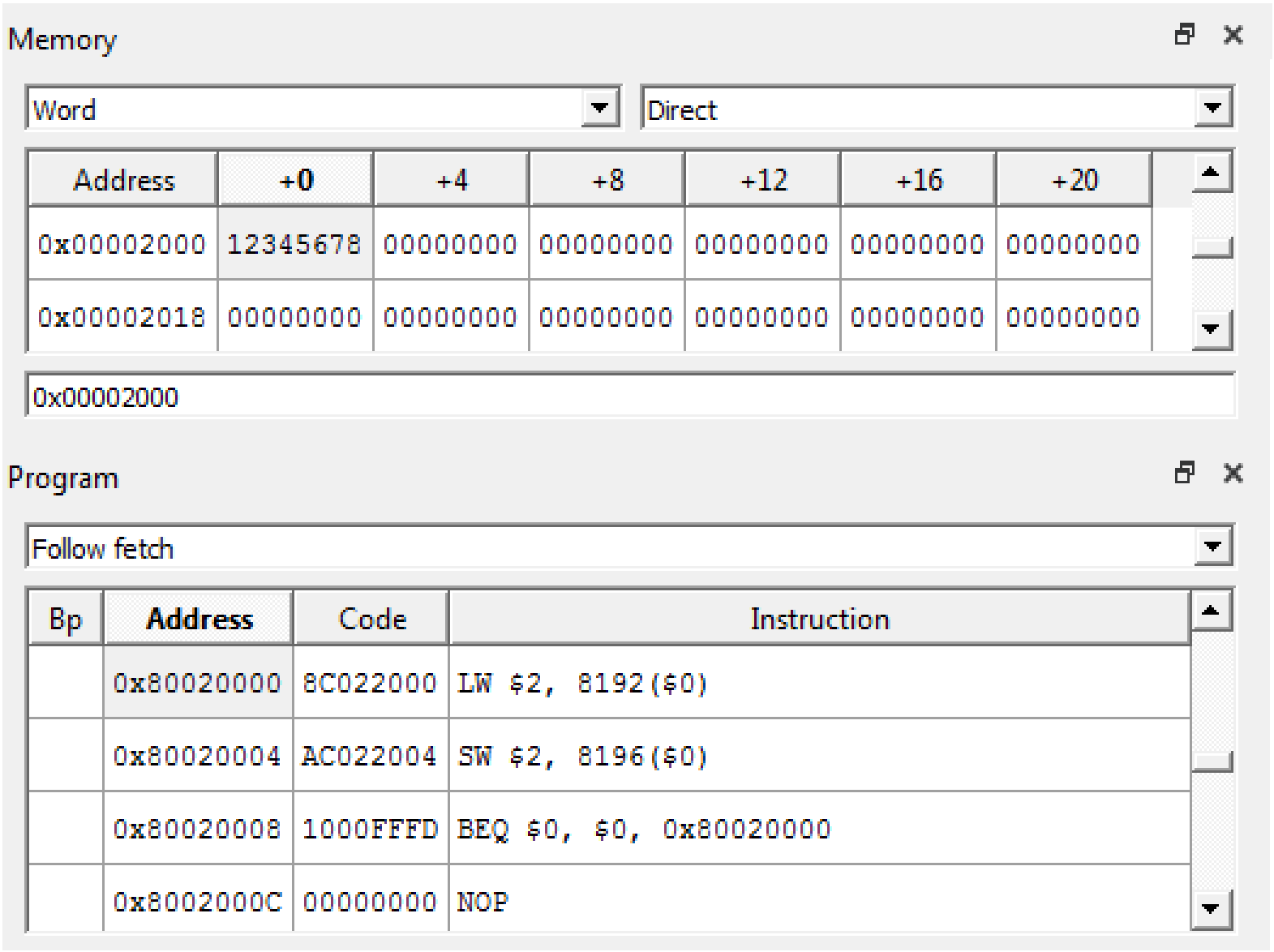
Instrukce lw a sw

Syntax	Operace	Význam
lw $t,C($s)	$t = Memory[$s + C]	Load word: Načte slovo z paměti a uloží jej do registru $t
sw $t,C($s)	Memory[$s + C] = $t	Store word: Uloží obsah registru $t do paměti

QtMips příklad simple-lw-sw-ia.S

```
#pragma qtmips show registers // pomocné direktivy QtMips simulátoru
#pragma qtmips show memory

.globl _start     // direktiva učiní symbol viditelný pro linker
.set noat         // potlačí varování, že uživatel měnil $at ($1) registr
.set noreorder    // potlačí změny pořadí instrukcí překladačem, důvod viz další přednášky

.text // začátek bloku instrukcí assembleru
_start:     // návěští začátku programu

loop:     // návěští cyklu v programu
      lw $2, 0x2000($0)  // i lw $v0, 0x2000($0) načti 4-bytové slovo z abs. adresy 0x2000
      sw $2, 0x2004($0) // i sw $v0, 0x2004($0) ulož 4-bytové registr na abs. adr. 0x2004
      beq $0, $0, loop // vždy skoč na návěští loop (skok při podmínce $0==$0)
      nop                  // No OPeration – prázdná instrukce

.data
. org 0x2000   // QtMIPS začne datový segment od adresy 0x2000
src_val:          // orientační návěští na adr 0x2000, v programu nepoužité, lze vynechat
      .word 0x12345678   // 4-bytové slovo
dst_val:         // orientační návěští na adr. 0x2004, v programu nepoužité, lze vynechat
      .word 0x0              // 4-bytové slovo 0
.end _start  // konec zdrojového kódu, QtMips nevyžaduje
```

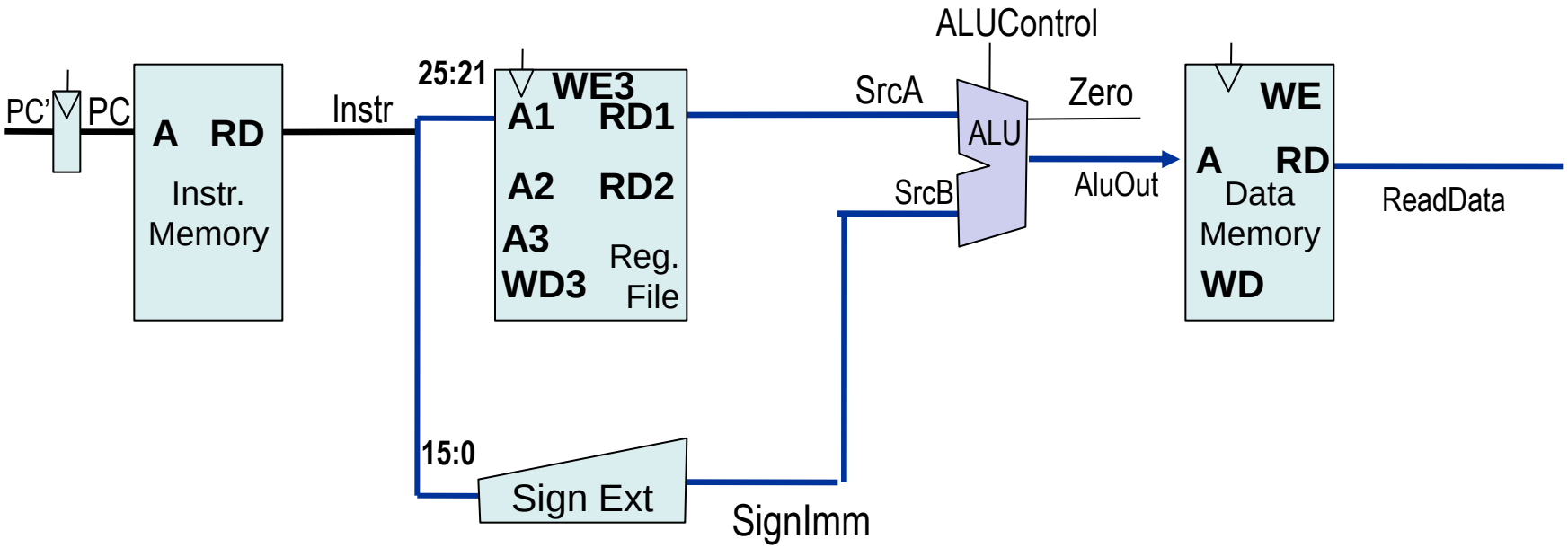
QtMips – výsledek překladu



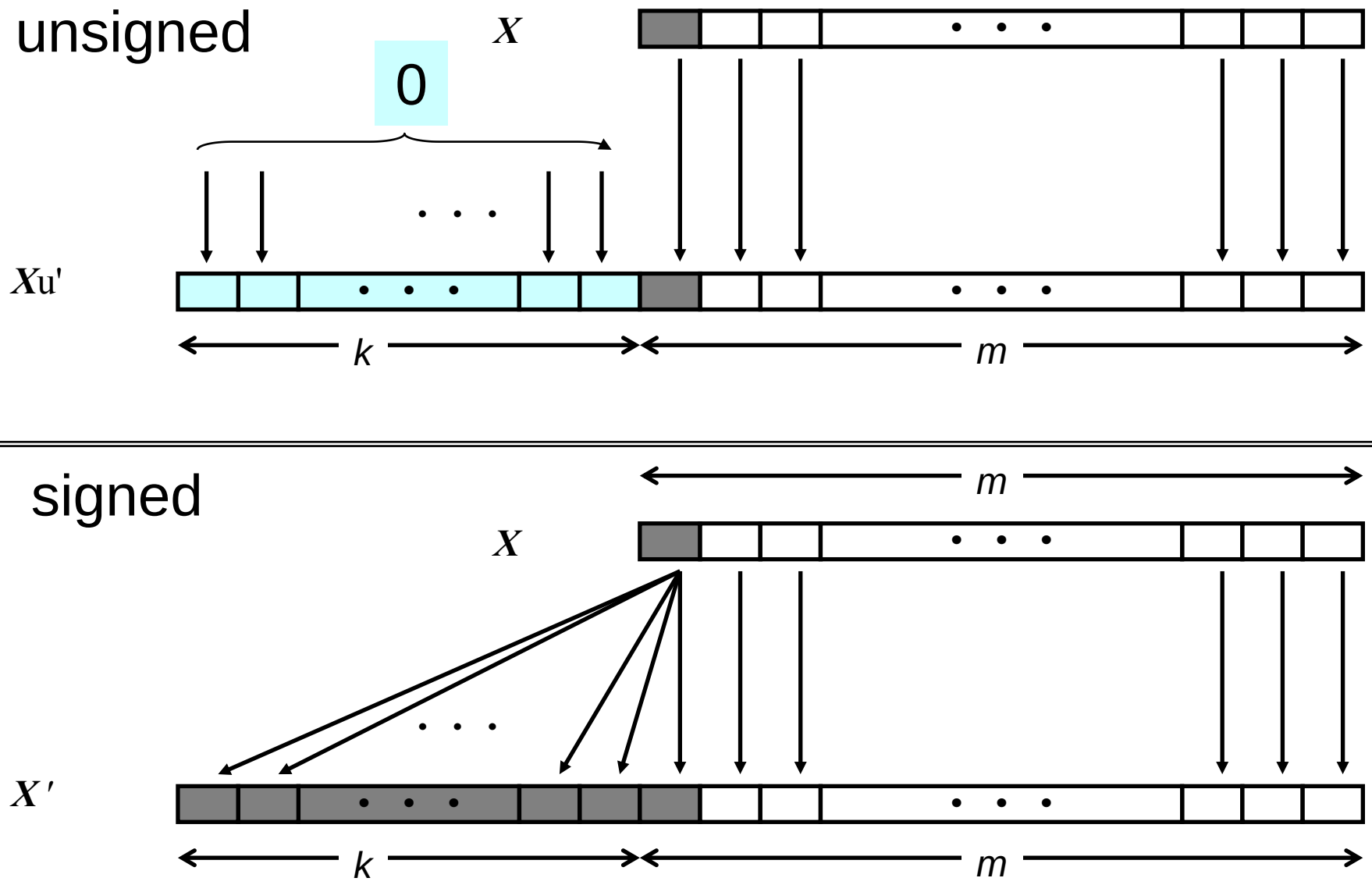
Jedno-cyklový procesor – návrh čtení z paměti

- **`lw`**: typ I, rs – bázová adresa, imm – offset, rt – kde uložit

I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



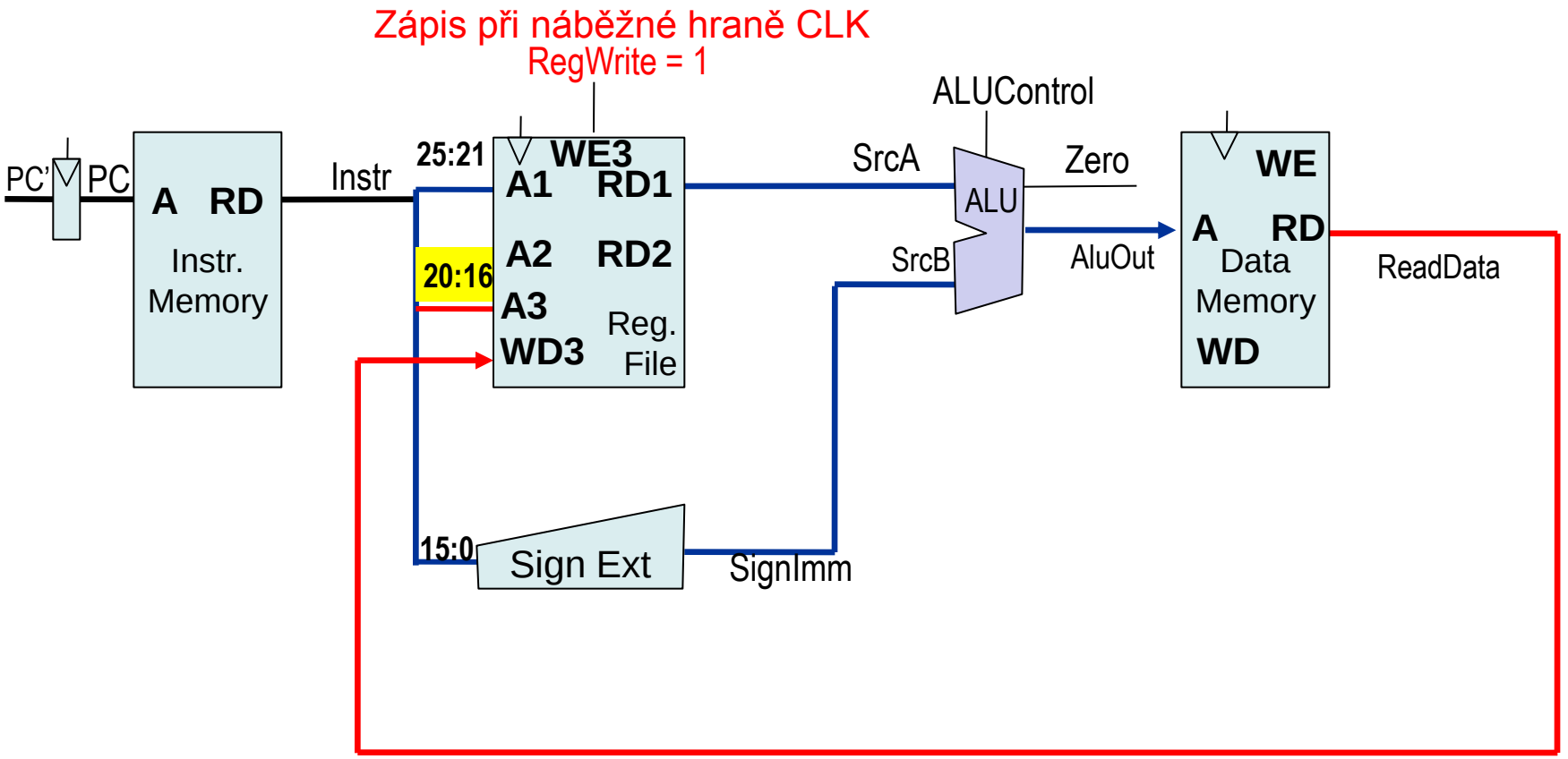
Unsigned/Signed Extension



Jedno-cyklový procesor – návrh čtení z paměti

- **`lw`**: typ I, rs – bázová adresa, imm – offset, rt – kde uložit

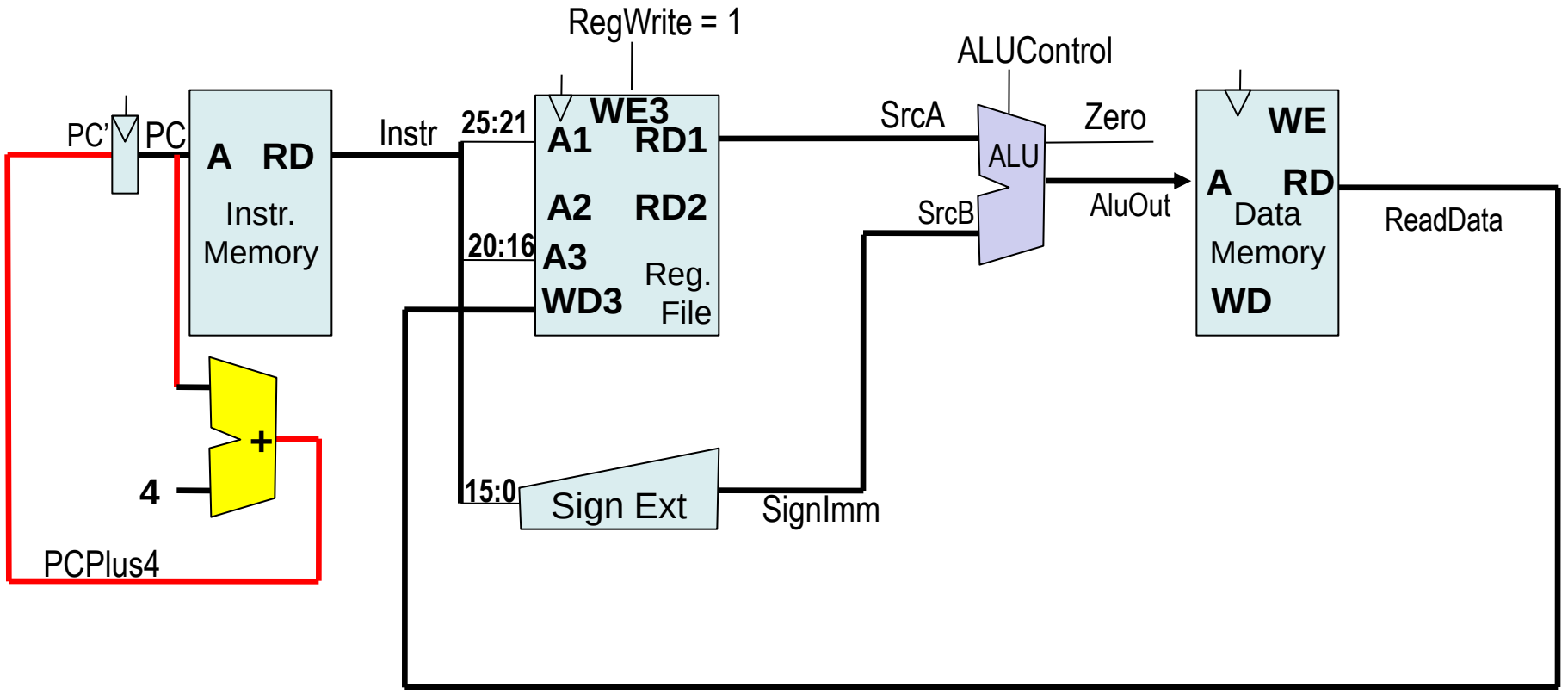
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Jedno-cyklový procesor – návrh čtení z paměti

- **`lw`**: typ I, rs – bázová adresa, imm – offset, rt – kde uložit

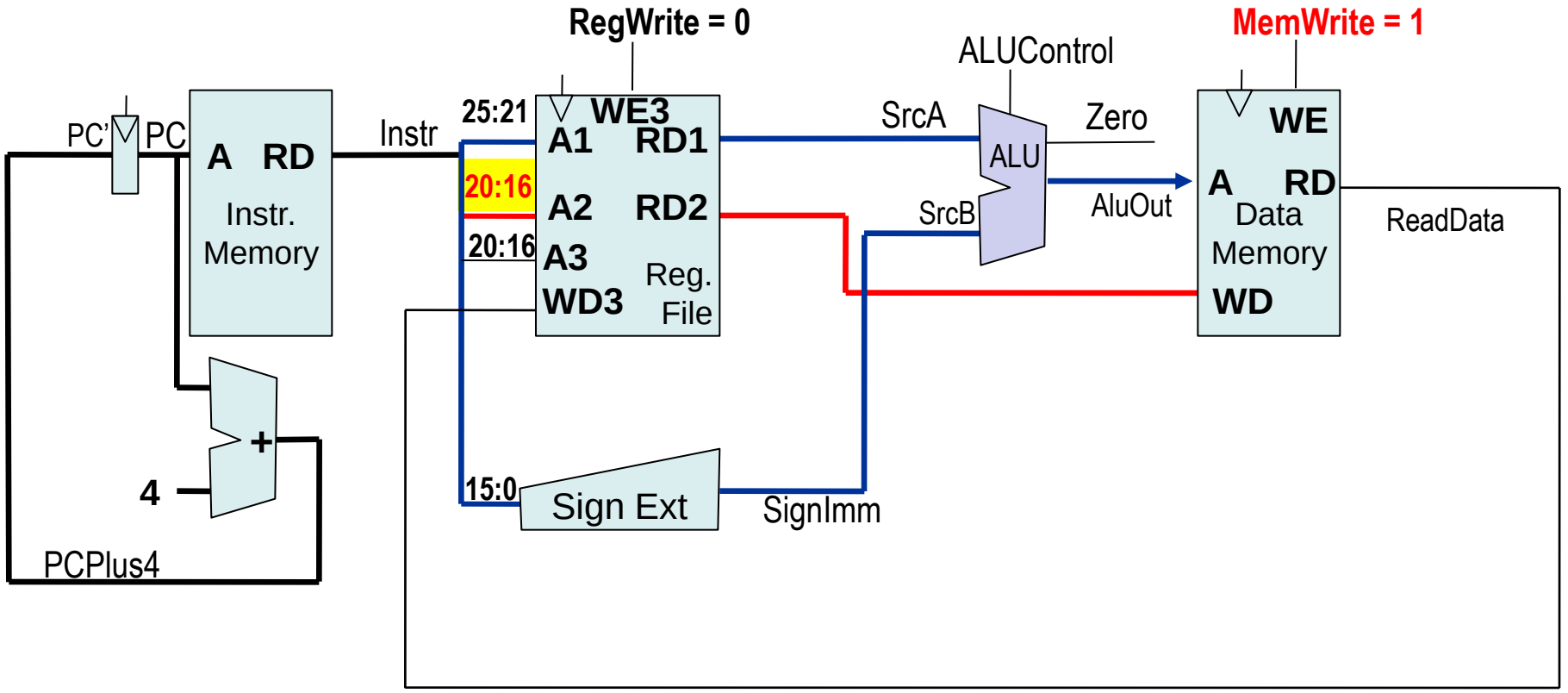
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



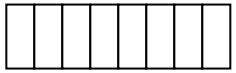
Jedno-cyklový procesor – návrh zápis do paměti

- **`sw`**: typ I, rs – bázová adresa, imm – offset, rt – co zapsat

I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



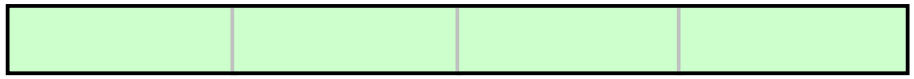
MIPS Load/Store instrukce



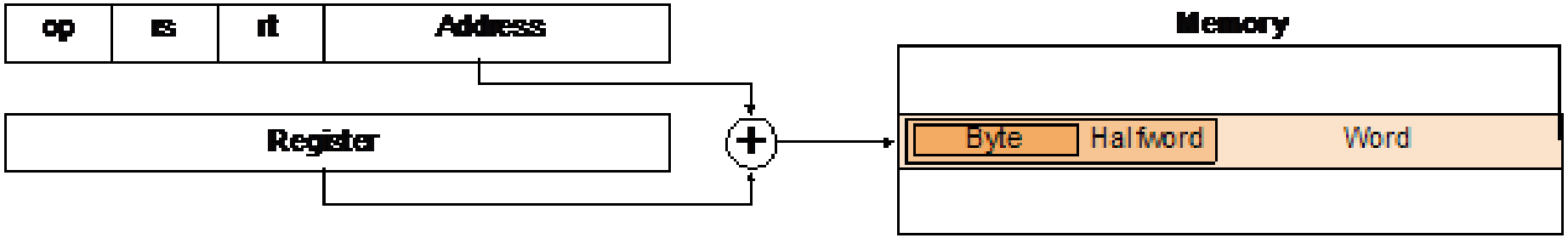
Load Byte = 8 bitů - **ldb / ldbu R1, offset(R2)**
- byte se znaménkovým rozšířením a bez něho

Load Halfword = 2 byty: **ldh / ldhu R1, offset(R2)**
- 2 byty se znaménkovým rozšířením a bez něho

Word = 4 byty - **ldw R1, offset(R2)** – Load Word



Store: **sw R1, offset(R2)** – ulož word,
sh R1, offset(R2) – ulož halfword,
sb R1, offset(R2) – ulož byte



Datové direktivy v sekci .data

Definují inicializovaná data v paměti a včetně jejich nepovinného návěští (label) ***name***

Syntaxe:

[***name***:] *directive* *initializer* [, *initializer*] . . .

```
var1:     .word   10
myarray:  .half   5, 3, 4, 1, 15
mojebyty: .byte   48, 49, 50, 51
```

```
mips-elf-gcc -Wl,-Ttext,0x80020000 -Wl,-Tdata,0x2000
        -nostdlib -nodefaultlibs -nostartfiles
        -o simple-lw-sw simple-lw-sw.S
```

Memory Alignment (cz:zarovnání paměti?)

`.align n` directive

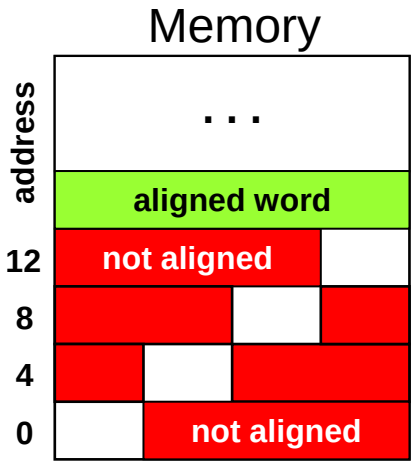
- další definice dat začíná na 2^n bytovém rozhraní

Příklad .align 2

- poslední dva bity začátku jsou **00**

Paměť se adresuje jako pole bytů

U 32-bitového procesoru je slovo (word) složeno ze 4 bytů



Poznámka: Direktiva align se v různých asemblerech píše jinak. Může mít více parametrů, někde n znamená počet bitů, jinde mocninu dvou. Vyskytuje se i tvar ***palign****. Někde direktiva ovlivní jen následující definici a další již ne.*
Nutno vždy prostudovat manuál.

Align v datovém segment assembleru

```
.DATA
.ALIGN  2
   var1:   .BYTE   3, 5,'A','P','O'
   var2:   .WORD   0x12345678
.ALIGN  3
   var3:   .HALF   1000
```

var1 ↓ var2 ↓

BIG ENDIAN	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2000	3	5	41	50	4F				12	34	56	78				
0x2010	10	00														

var3 ↑

var1 ↓ var2 ↓

LITTLE ENDIAN	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x2000	3	5	41	50	4F				78	56	34	12				
0x2010	00	10														

var3 ↑

ALU Instructions

operace	R-format	I-format
add	**add addu**	**addi addiu**
subtract	**sub**	**-**
multiply divide	**mult / multu** **div / divu**	**-**
AND	**and**	**andi**
OR	**or**	**ori**
XOR	**xor**	**xori**
NOR	**nor**	**-**

addi, addiu **rB ← rA + se(*number*)**,
addu, addiu - no overflow trap

Logické instrukce ANDI, ORI, XORI nemají sign-extension!

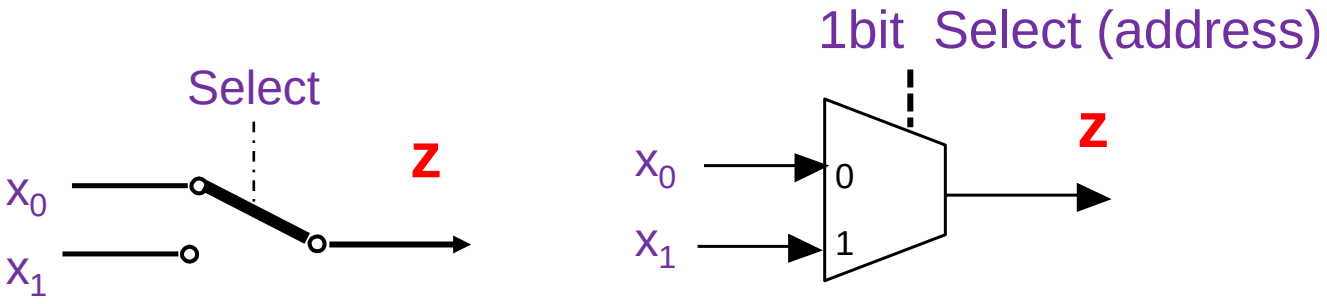
Jak dostanu hodnotu do registru ?

ori $1, $0, 1000	$1← 1000
addi $2, $0, 1000	$2 ← 1000
lui $3, 0x1234	$3 ← 0x12345678
ori $3, 0x5678	
la $3, 0x12345678	la - pseudo-instrukce

Instr.	Syntax	Operace
Load upper immediate: Uloží předanou přímou hodnotu C do horní části registru. Registr je 32-bitový, C je 16-bitová.		
lui	lui $t,C	$t = C << 16
Load Address: 32-bitové návěstí uloží do registru $at. Jedná se o pseudoinstrukci - tzn. při překladu se rozloží na dílčí instrukce.		
la	la $at, LabelAddr	lui $at, LabelAddr[31:16]; ori $at,$at, LabelAddr[15:0]

Připomenutí přepínačové analogie multiplexoru

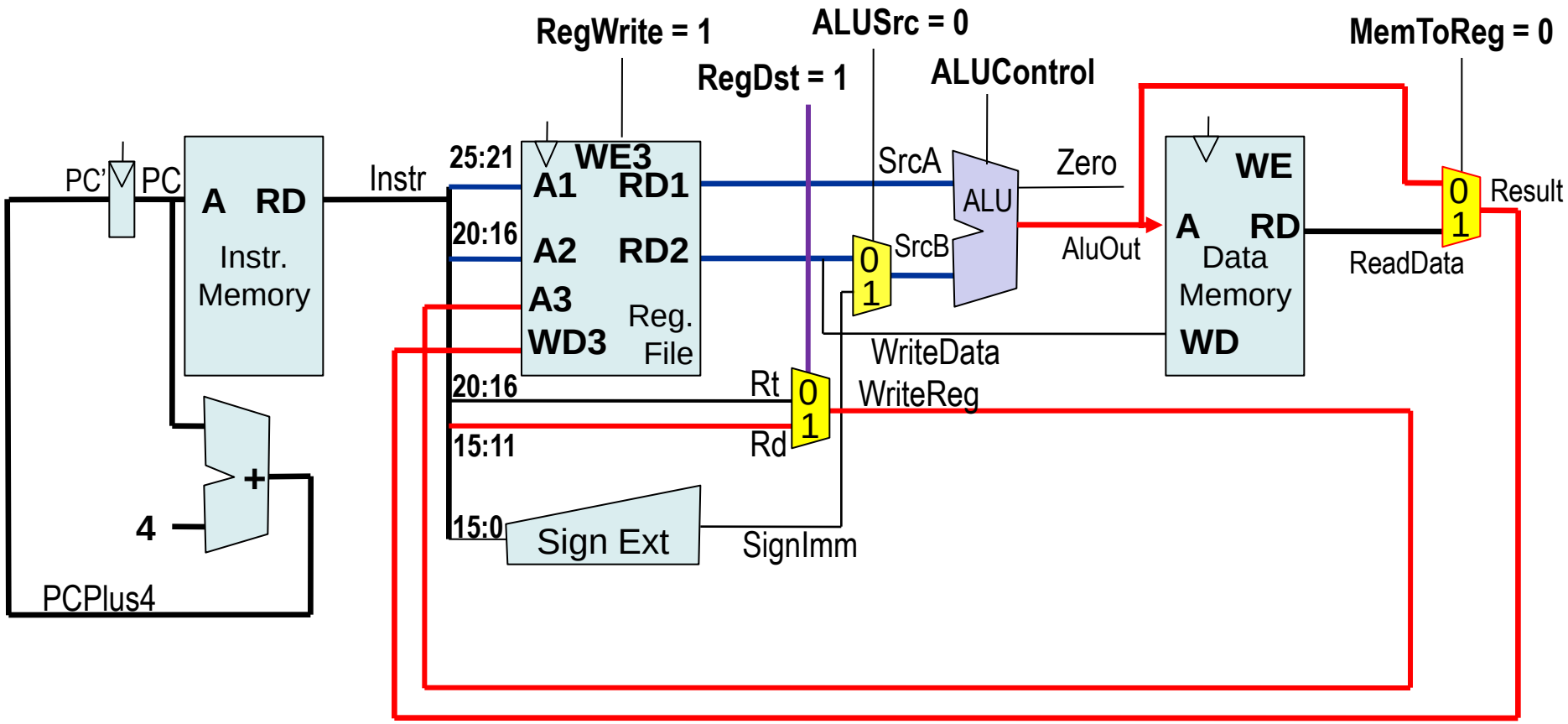
Multiplexer 2 to 1 *cz :2-kanálový (2-vstupový) multiplexor*



Jedno-cyklový procesor – návrh – podpora add

- **add**: typ R; rs, rt – zdroje, rd – cíl, funct – operace součtu

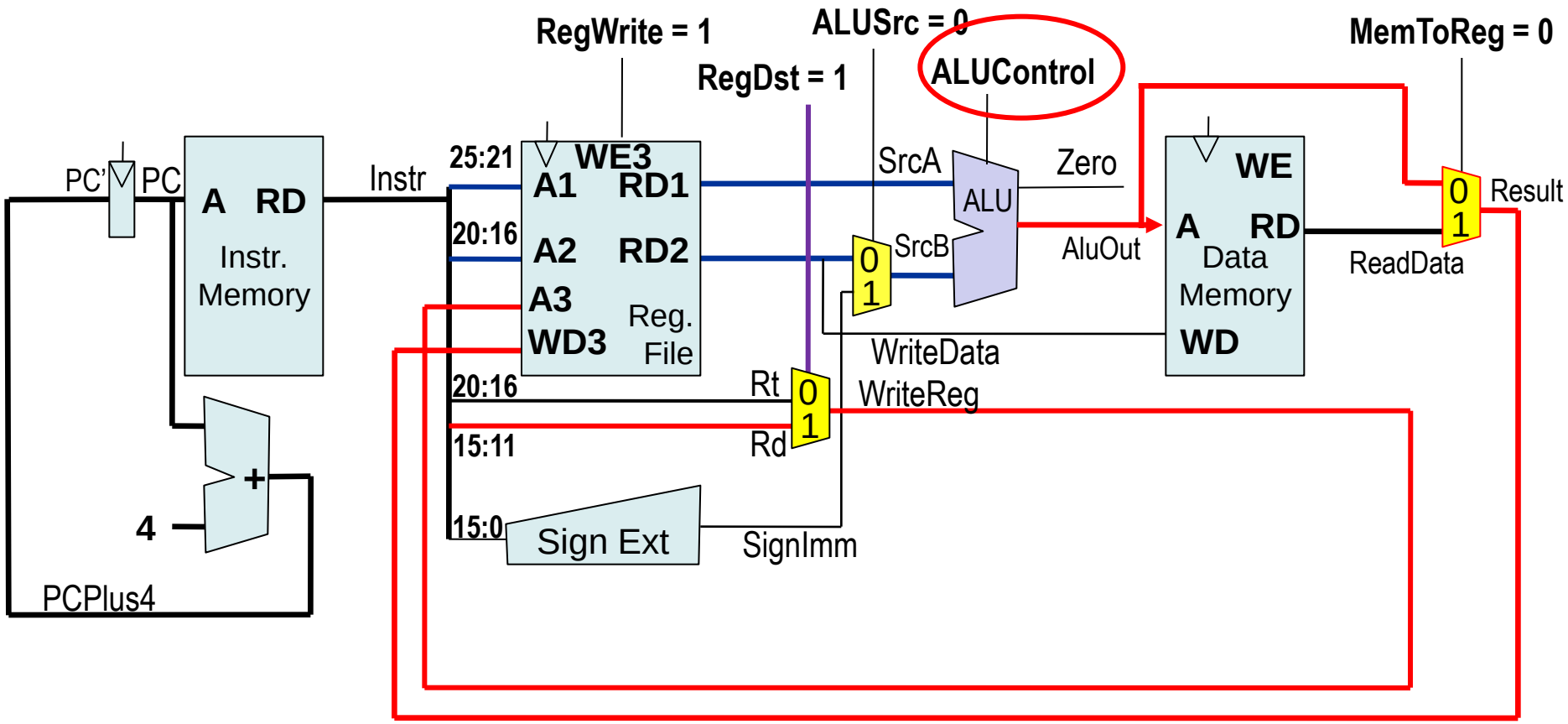
R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0



Jedno-cyklový procesor – návrh – podpora *sub, and, or, slt*

Cesta dat je beze změny; rozdíl od add jen v ALUControl

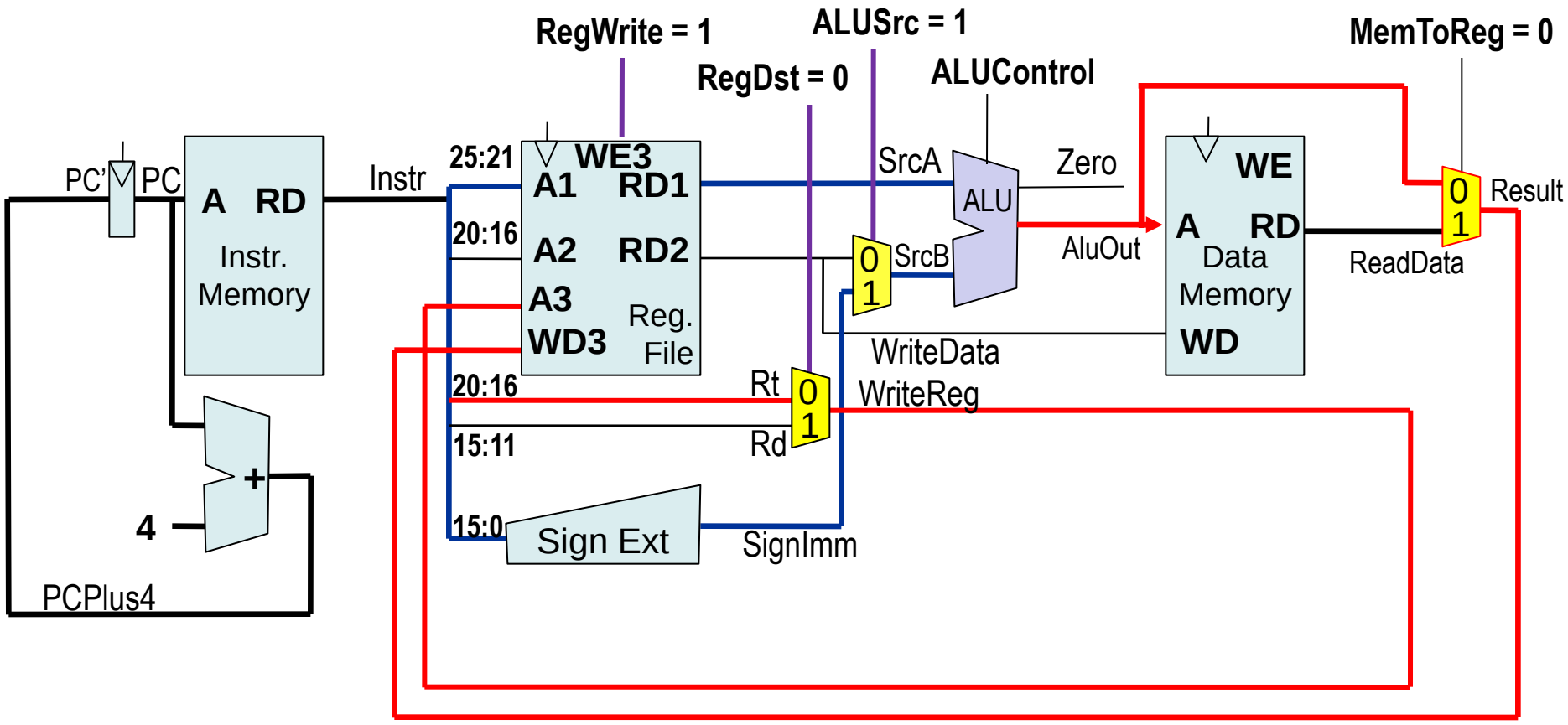
R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0



Jedno-cyklový procesor – návrh – podpora `addi`

- **`addi`**: typ R; rs – zdroj, rt – cíl, funct – operace součtu

R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**15:0** konstanta



Některé shift operace

Instr.	Syntax	Operace	Význam
sll	sll $d,$s,C	$d = $s << C	***Shift Logical Left**: Posune hodnotu v registru o C bitu doleva (ekvivalentní k operaci nasobeni konstantou 2^C)*
srl	srl $d,$s,C	$d = $s >> C *unsigned*	***Shift Logical Right**: Posune hodnotu v registru o C bitu doprava (ekvivalentní dělení konstantou 2^C)*
sra	sra $d,$s,C	$d = $s >> C *signed*	***Shift Logical Right**: Posune hodnotu v registru o C bitu aritmeticky doprava (ekvivalentní dělení konstantou 2^C)*
nop	nop	sll $0,$0,0	*pseudoinstrukce - **n**o **op**eration*

NOP binární kód

000000 00000 00000 00000 00000 000000 -- pole instrukce

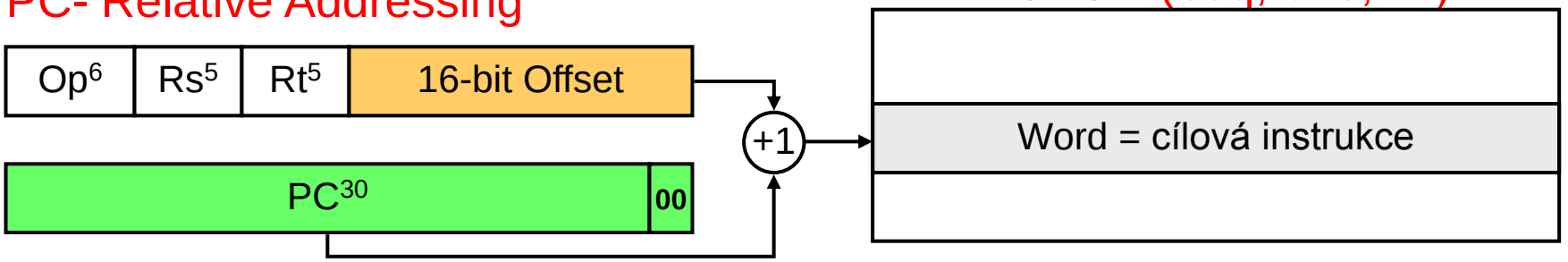
opcode *$0* *$0* *0* *funct* -- význam

sll source dest shft sll sll $0, $0, 0

MIPS skoky

PC- Relative Addressing

Větvení (beq, bne, …)



Cíl větvení

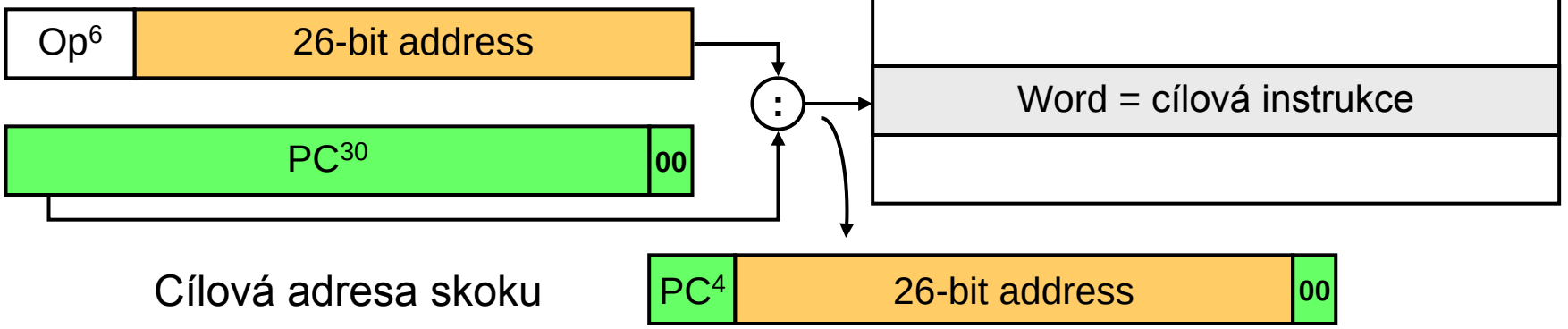
PC = PC + 4 × (1 + Offset)

= PC+4+4*offset

$PC^{30} + Offset^{16} + 1$ | 00

Pseudo-direct Addressing

používá jediná skoková instrukce



Cílová adresa skoku

Source: Dr. M. Mudawar, COE 301, KFUPM

MIPS Jump Instruction

Branch on not equal: *Skáče, pokud registry $s a $t obsahují různé hodnoty*		
bne	bne $s, $t, offset	if $s != $t goto PC+4+4*offset; else goto PC+4
Branch on equal: *Skáče, pokud registry $s a $t obsahují stejné hodnoty*		
beq	beq $s, $t, offset	if $s == $t goto PC+4+4*offset; else goto PC+4
Jump: *Skáče bezpodmínečně na návěstí C*		
jump	j C	

MIPS podpora skoků

Set on less than

*Platí-li podmínka $s < $t nebo $s<imm jako **signed**, pak registr $d=1, jinak $d=0*

slt, slti	slt $d,$s,$t slti $d, $s, imm	$d = ($s < $t) $d = ($s < imm)

Set on less than

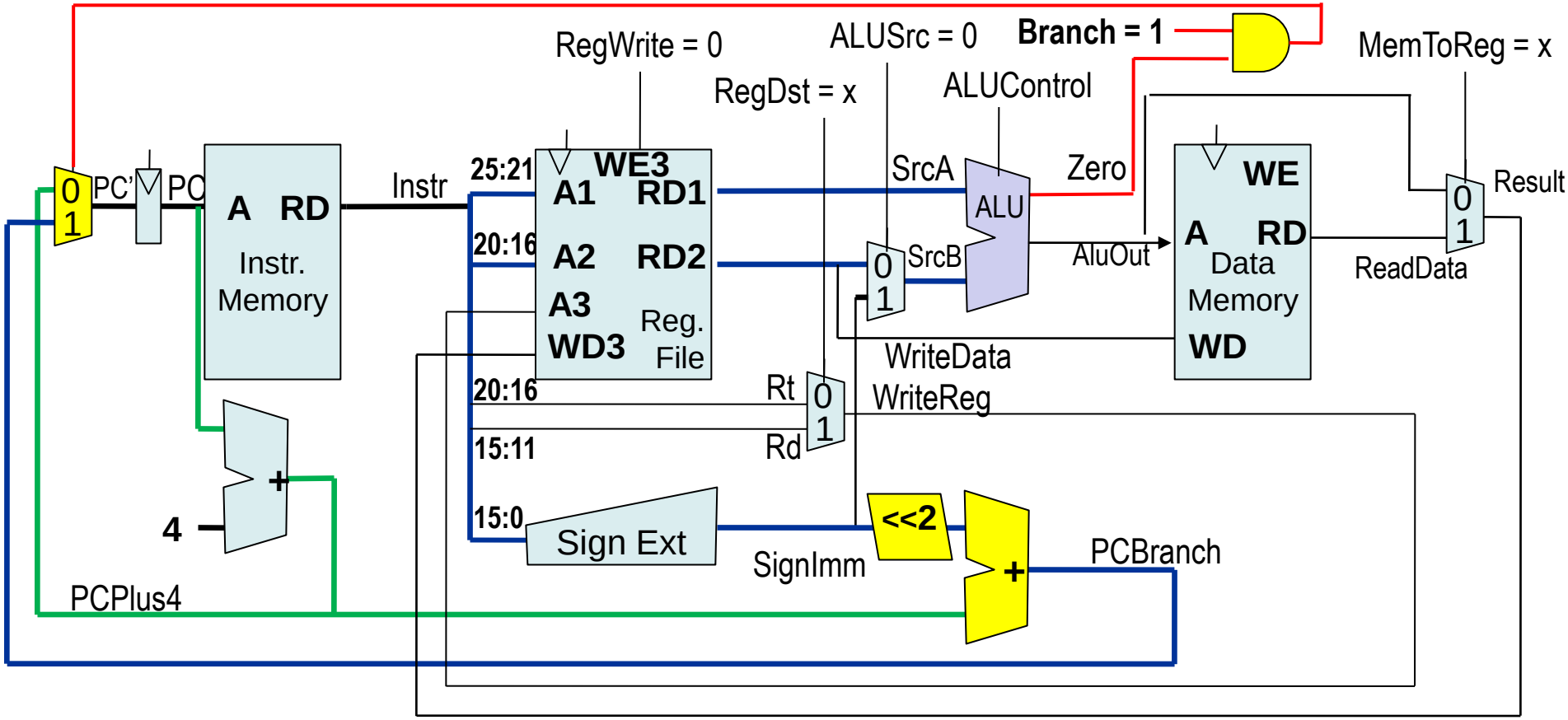
*Platí-li podmínka $s < $t nebo $s<imm jako **unsigned**, pak registr $d=1, jinak $d=0*

sltu, **sltiu**	sltu $d,$s,$t sltu $d, $s, imm	$d = ($s < $t) $d = ($s < imm)

Jedno-cyklový procesor – návrh – podpora beq

- **beq** – branch if equal; imm–offset; PC´ = PC+4 + SignImm*4

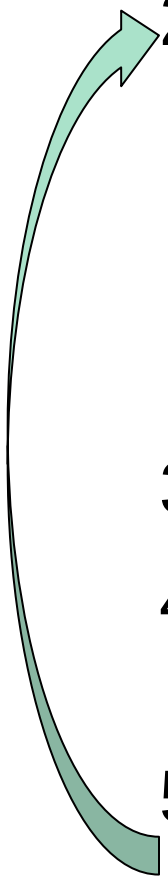
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Assembly code - again

```
/* template for own QtMips program development */
.globl _start    // .globl makes the symbol visible to linker
.set noat        // disables warning when $at register is used by user.
.set noreorder // prevents the assembler from reordering machine-language instructions
                 // See later lectures
.ent _start
.text
_start:
      lw $2, 0x2000($0)     // load the word from absolute address
      sw $2, 0x2004($0) // store the word to absolute address

loop:
      break // stop execution wait for debugger/user
      beq $0, $0, loop // endless loop
      // it ensures that continuation does interpret random data
      nop
.data
src_val:
      .word 0x12345678
dst_val:
.end _start
```

Základní cyklus počítače – sekvenční postup vykonávání instrukcí



1. Počáteční nastavení, zejména např. PC.
2. Čtení instrukce

   PC → adresa hlavní paměti,

   Čtení obsahu,

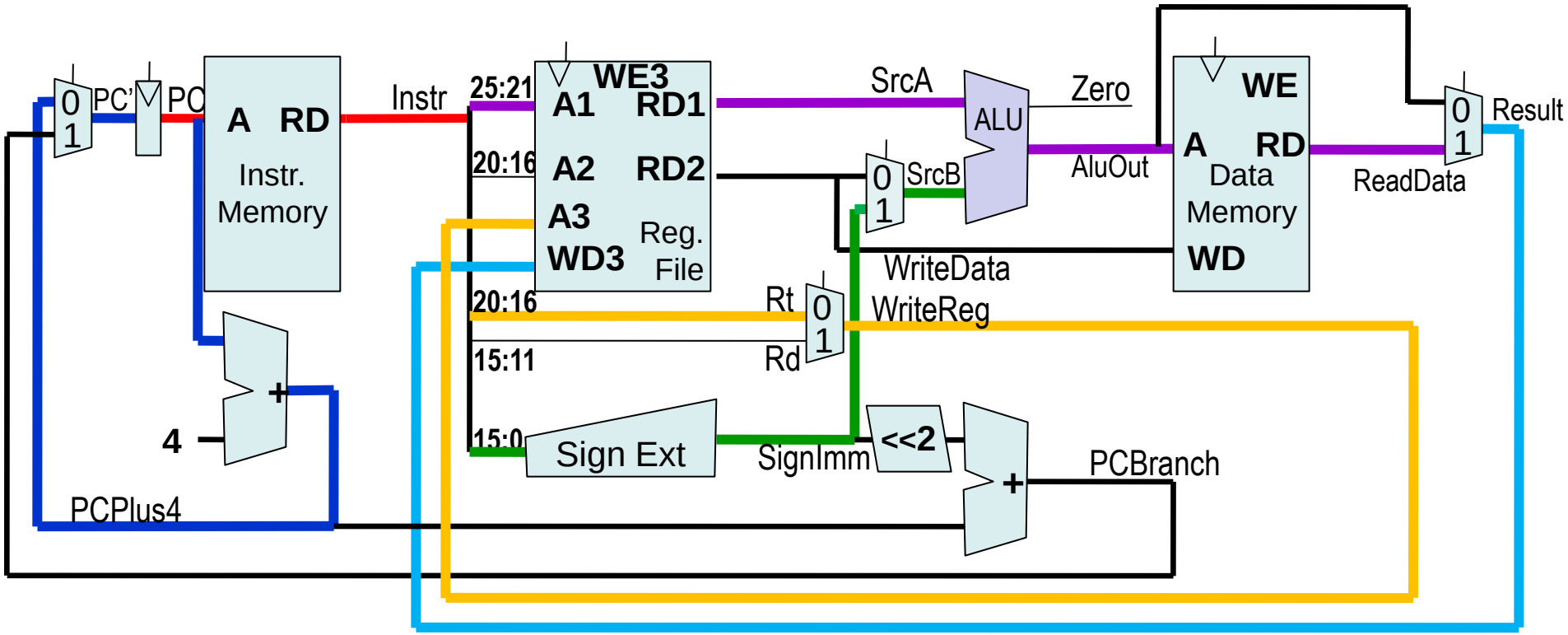
   Přečtená data → IR (Instruction Register),

   PC+n → PC, kde *n* je délka instrukce.
3. Dekódování operačního znaku (OZ),
4. Provedení operace (včetně vyhodnocení efektivních adres, čtení operandů, apod.).
5. Dotaz na možné přerušení. Ano-li, obsluha.
6. Ne-li, opakování od bodu 2.

Jedno-cyklový procesor – výkon: IPS = IC / T = $IPC_{str}.f_{CLK}$

- Jaká může být maximální frekvence procesoru?
- Zpoždění na kritické cestě – instrukce `lw`:

$$Tc = t_{PC} + t_{Mem} + \mathbf{t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}}$$



Jedno-cyklový procesor – výkon: IPS = IC / T = $IPC_{str}.f_{CLK}$

- $Tc = Tc_{instr} + Tc_{proc}$
  $= (t_{PC} + t_{Mem}) + (t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup})$
- Předpokládejme:

t_{PC}	= 30 ns	t_{Mem}	= 300 ns
t_{RFread}	= 50 ns	t_{ALU}	= 200 ns
t_{Mux}	= 20 ns	$t_{RFsetup}$	= 20 ns

Pak Tc = 920 ns --> $f_{CLK\ max}$ = 1,08 MHz,
IPS = 1 080 000 [instrukce za sekundu]

Ale při Tc_{instr} prováděném paralelně s Tc_{proc},
jelikož je vždy $Tc_i < Tc_p$, pak Tc_p = 50+200+300+20+20
= 590 ns = 1.69 MHz -> **IPS = 1 690 000**

Důležitá poznámka

- Tenhle výsledek si, prosím, zapamatujte.
- Budeme s ním pracovat na pozdější přednášce.

Kompilace a kódování programu

```
int pow = 1;
int x = 0;

while(pow != 128)
{
  pow = pow*2;
  x = x + 1;
}
```

```
addi s0, $0, 1        // pow = 1
addi s1, $0, 0        // x = 0
addi t0, $0, 128      // t0 = 128

while:
  beq  s0, t0, done  // if pow==128, go to done
  sll  s0, s0, 1      // pow = pow*2
  addi s1, s1, 1      // x = x+1
  j    while
done:
```

```
8001FFF4  00 00 00 00          NOP
8001FFF8  00 00 00 00          NOP
8001FFFC  00 00 00 00          NOP
80020000  20 10 00 01  start() ADDI   $16, $00, 0x1
80020004  20 11 00 00          ADDI   $17, $00, 0x0
80020008  20 08 00 80          ADDI   $08, $00, 0x80
8002000C  12 08 00 04  while:  BEQ    $08, $16, 0x4
80020010  00 00 00 00          NOP
80020014  00 10 80 40          SLL    $16, $16, 1
80020018  08 00 80 03          J      0x8003
8002001C  22 31 00 01          ADDI   $17, $17, 0x1
80020020  00 00 00 00  done:   NOP
80020024  00 00 00 00          NOP
80020028  00 00 00 00          NOP
```