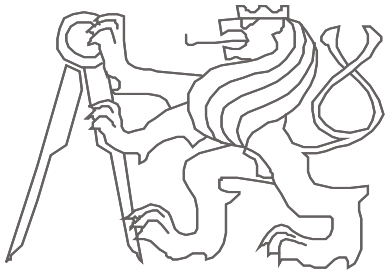
Architektura počítačů

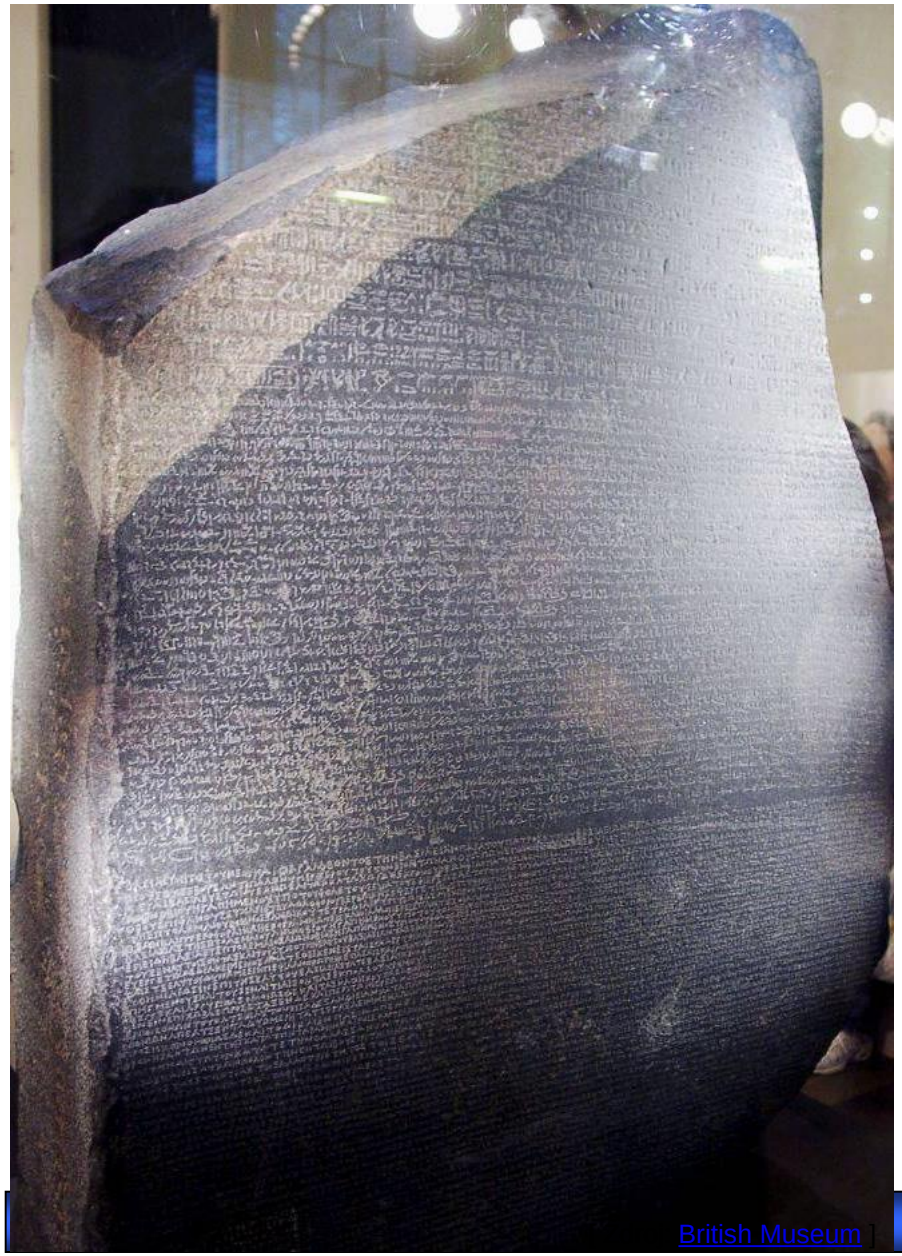
Počítačová aritmetika a úvod

Richard Šusta, Pavel Píša



České vysoké učení technické, Fakulta elektrotechnická

Omluva


British Museum]

Část snímků
nebude v češtině

** přednášky letos nově upravuji*
s cílem zpomalit výklad,
a tak měním lekce
i pro zahraniční studenty.
Žel pak nestíhám ještě
překlad z češtiny do angličtiny...

Zjednodušený přehled operací v plovoucí řádové čárce

Předvody: *na/z integer, double, float - pouhý posun mantisy podle exponentu*

Sčítání: $\mathbf{A \cdot 2^a , B \cdot 2^b , b < a}$

1. sjednocení exponentů posunem mantisy

$\mathbf{A \cdot 2^a + (B \cdot 2^{b-a}) \cdot z^a}$

2. sečtení + normalizace

$\mathbf{(A + (B >> (a-b))) \cdot z^a = [A + (B \cdot 2^{b-a})] \cdot 2^a}$

Odčítání: sjednocení exponentů, odečtení a normalizace

Násobení: $\mathbf{A \cdot 2^a \cdot B \cdot 2^b = A \cdot B \cdot 2^{a+b}}$

A·B +*normalizace posunem doleva, je-li třeba*

Dělení $\mathbf{A \cdot 2^a / B \cdot 2^b = A/B \cdot 2^{a-b}}$

A/B +*případná normalizace posunem doprava*

Příklad: a + b = 1000*pi + e/20

1.		**Float 6.5 číslic**	**na int**	<8 388 608;16 777 216) = <2^23;2^24)
af	1000*pi	~3141.593	* 2^12	**12867964**,928
bf	e/20	~0,1359141	* 2^26	**9121040**,8525824

2.	**Převedeme na binární číslo**		ale k němu
ax	**12867965**	1100 0100 0101 1001 0111 1101	. bude za 12.bitem <- * 2^12
bx	**9121041**	1000 1011 0010 1101 0001 0001	. bude za 26. bitem <- * 2^26

3.	**Binární číslo**	**exponent**
a	1100 0100 0101.1001 0111 1101	2^0
b	00.00 1000 1011 0010 1101 0001 0001	2^0

4.	**Normalizované binární číslo**	**exponent**
a	1.100 0100 0101 1001 0111 1101	* 2^11, 12+11=23
b	1.000 1011 0010 1101 0001 0001	* 2^-3, 26-3 = 23

Kontrola mezivýsledků

1.100 0100 0101 1001 0111 1101 * 2^11

= (12867965 * 2^-23) * 2^11 - *skutečně uložená hodnota*

= **1,533980**96561431884766 * 2^11

= **3141,593**01757812500000768

(1000*pi=**3141,592**6535897932384626 4)

1.000 1011 0010 1101 0001 0001 * 2^-3

= (9121041 * 2^-23) * 2^-3

= **1.087312**81757354736328 * 2^-3

= **0.1359141**0219669342041

(e/20=0,**1359140**9142295226177)

Výpočty provedené na kalkulátoru ***SpeedCrunch 0.12 - http://speedcrunch.org/***

Příprava na sčítání

5.	Normalizované binární číslo	exp.
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	1.000 1011 0010 1101 0001 0001	* 2^-3

6.	Na stejné exponenty	exp.
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	0.000 0000 0000 0010 0010 1100 ***10110100010001***	* 2^11

U čísla b je binární tečka posunutá o 14 míst vlevo, tj. rozdíl exponentů 11-(-3).
Červeně označené bity utíkají mimo rozsah -> ztráta přesnosti.

7.	Součet	expt
a	1.100 0100 0101 1001 0111 1101	* 2^11
+ b	0.000 0000 0000 0010 0010 1100 ~~***10110100010001***~~	* 2^11
a+b	1.100 0100 0101 1011 1010 1001	* 2^11

Výsledek a kontrola v double

a+b = 1.100 0100 0101 1011 1010 1001 * 2^11

= (12868521 * 2^-23) * 2^11

= **1.534047**24597930908203 * 2^11

= **3141,728**7597656249999744

Původní čísla a+b sečtená v double

=3141.593 + 0,1359141 = **3141,728**9141

= 1.10001000101101110101**1010** * 2^11 a jeho skutečná hodnota

= 3141,72900390625

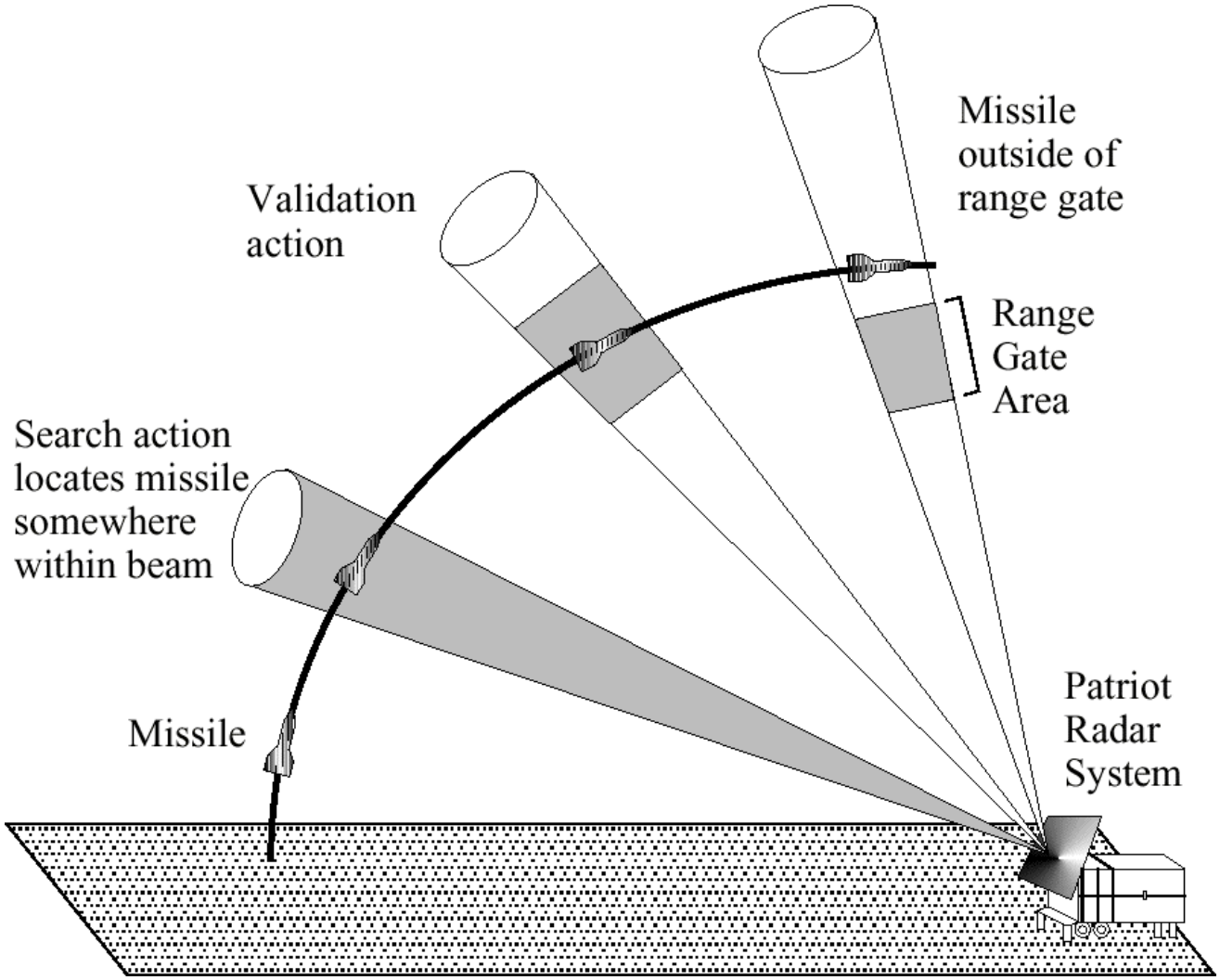
Výpočet v double 1000*pi+e/20 a následný převod na float

1.100 0100 0101 1011 1010 1000* 2^11 = 3141,728515625

*Výpočty provedené na kalkulátoru **SpeedCrunch 0.12** - http://speedcrunch.org/*

Effect of Loss of Precision

According to the General Accounting Office of the U.S. Government, a loss of precision in converting 24-bit integers into 24-bit floating point numbers was responsible for the failure of a Patriot anti-missile battery.



Slide source: UIUC

Effect of Loss of Precision

- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- This doesn't seem like much, but after 100 hours the time ends up being off by 0.34 seconds—enough time for a Scud to travel 500 meters!
- UIUC Emeritus Professor Skeel wrote a short article about this.

Roundoff Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.

Slide source: UIUC

Jak správně sčítat ?

Chceme napsat program pro zjištění součtu:

$$\sum_{i=1}^{N} \frac{1}{i^2}$$

Najdete všechny chyby v programu?

- Který způsob je nejvýhodnější?

$$\sum_{i=1}^{N} \frac{1}{i^2} = \sum_{i=N}^{1} \frac{1}{i^2} = \sum_{i=1}^{N} \frac{1}{(N-i+1)^2}$$

$$\sum_{i=1}^{10^{10}} \frac{1}{i^2} \approx 1.644934057834186,5$$

$$\sum_{i=10^{10}}^{1} \frac{1}{i^2} \approx 1.644934066848226 4$$

typ double pro oba případy...
Proč se liší? Vysvětlete!

Násobení čísel v pohyblivé řádové čárce

- Exponenty sečteme.
- Mantisy vynásobíme.
- Normalizujeme.
- Zaokrouhlíme.

HW FP násobičky je srovnatelně složitý, jako FP sčítačky.
Jen má namísto sčítačky násobičku.

Rychlost integer operací

Operace	Operace
Negace čísla	negace MSB (Main Scale Bit)
Porovnání	a) znaménko -> b) abs. hodnota
Násobení či dělení 2^n	změna exponentu
Převody mezi int, float, double	posun mantisy dle exponentu
Sčítání, Odčítání, Increment, decrement	Srovnání mantis na stejné exponenty, +/-, zaokrouhlení, normalizace
Násobení na hardwarové násobičce	Součet exponentů, součin mantis, zaokrouhlení, normalizace
Násobení na sekvenční násobičce	
Dělení	Rozdíl exponentů, podíl mantis, zaokrouhlení, normalizace

Iterační dělička - Goldschmidt

$$Q = \frac{N}{B} = \frac{m_N 2^{e_N}}{m_B 2^{e_B}} = \frac{m_N}{m_B} 2^{e_N - e_b}$$

Pokud jsou čísla v normalizovaném tvaru, pak platí:

m_N = 1.???????...? a m_B = 1.???????...?

tzn. $1 \leq m_N, m_B < 2$ pokud uvažujeme celou mantisu, nebo
$0{,}5 \leq m_N, m_B < 1$ pokud bereme pouze zlomkovou část.

Uvažujme pouze zlomkovou část (za desetinnou čárkou).

Goldschmidt Division

* Let us compute the reciprocal of B (1/B)
  * Then, we can use the standard floating point multiplication algorithm
* Ignoring the exponent
  * Let us compute $(1/P_B)$, where P_B is mantissa
* If B is a normal floating point number
  * $1 <= P_B < 2$
  * $P_B = 1 + X$ where $(X < 1)$

Source: IIT Delhi, McGrawHill

Goldschmidt Division - II

$$\frac{1}{P_B} = \frac{1}{1+X} \; (P_B = 1 + X, 0 < X < 1)$$

$$= \frac{1}{1+1-X'} \; (X' = 1 - X, X' < 1)$$

$$= \frac{1}{2 - X'}$$

$$= \frac{1}{2} * \frac{1}{1 - \frac{X'}{2}}$$

$$= \frac{1}{2} * \frac{1}{1 - Y} \; (Y = \frac{X'}{2} = (1 - X)/2, Y < \frac{1}{2})$$

Source: IIT Delhi, McGrawHill

Goldschmidt Division - III

$$\frac{1}{1-Y} = \frac{1+Y}{1-Y^2}$$
$$= \frac{(1+Y)(1+Y^2)}{1-Y^4}$$
$$= \ldots$$
$$= \frac{(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})}{1-Y^{32}}$$
$$\approx (1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})$$

* There is no point considering Y^{32} because it cannot be represented in our format!

Source: IIT Delhi, McGrawHill

Generating the 1/(1-Y)

$$(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})$$

* We can compute Y^2 using a FP multiplier.
  * Again square it to obtain Y^4, Y^8, and Y^{16}
  * Takes 4 multiplications, and 5 additions, to generate all the terms
  * Need 4 more multiplications to generate the final result (1/1-Y)
* Compute $1/P_B$ by a single right shift

Source: IIT Delhi, McGrawHill

Iterační dělička – Goldschmidt – vylepšená verze

- S rostoucím x ($0 < x \leq 0{,}5$) se konvergence zhoršuje. Pokud x == 0.5 je nejhorší. Jinými slovy, pro fixní počet iterací se snižuje přesnost výsledku.

- Modifikace Goldschmidtova algoritmu spočívá v odhadu (nepřesném) převrácené hodnoty K hodnoty m_D z look-up tabulky – podle několika málo prvních bitů (~10).
- Místo původního x=1- m_D počítáme s x=1- Km_B

$$\frac{m_N}{m_B} \approx m_N K\left(1+x\right)\left(1+x^2\right)\ldots\left(1+x^{2^i}\right)$$

- Tato dělička se používá v moderních CPU.
- Kontrolní otázka: Můžeme tuto děličku použít i pro INTEGER?

*Program

k zamyšlení

Kvíz: Rozhodněte o platnosti vztahů

```
int x = …;
float f = …;
double d = …;
```

Předpokládejme, že `d` a `f` nejsou NAN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Odpovědi na kvíz

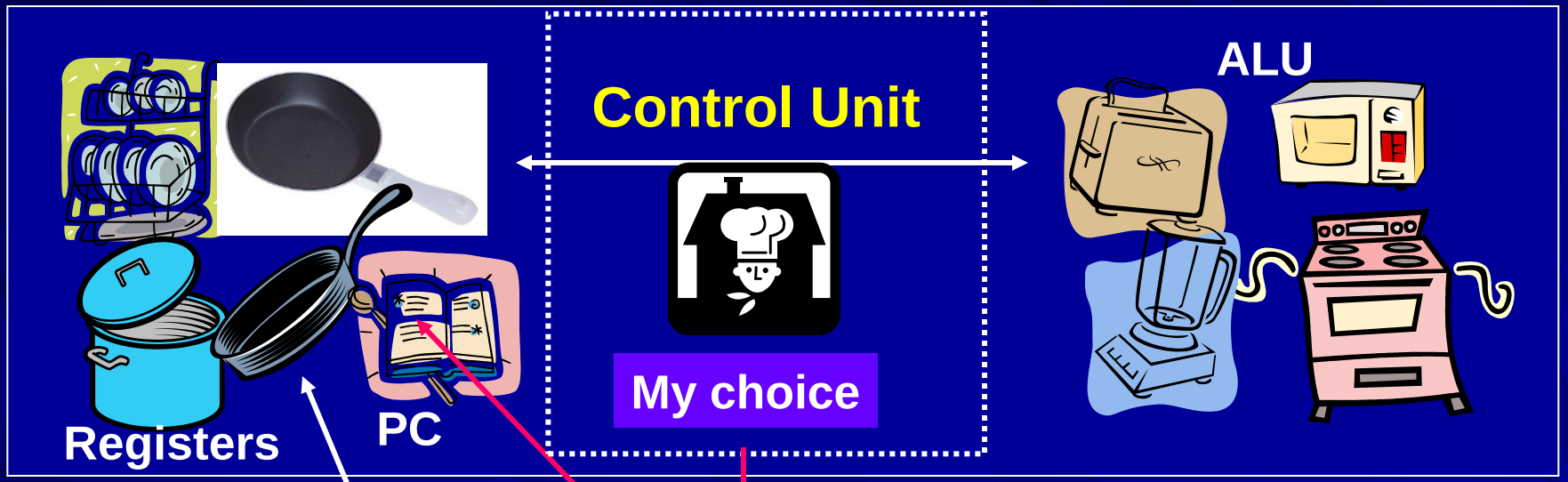
```
int x = …;
float f = …;
double d = …;
```

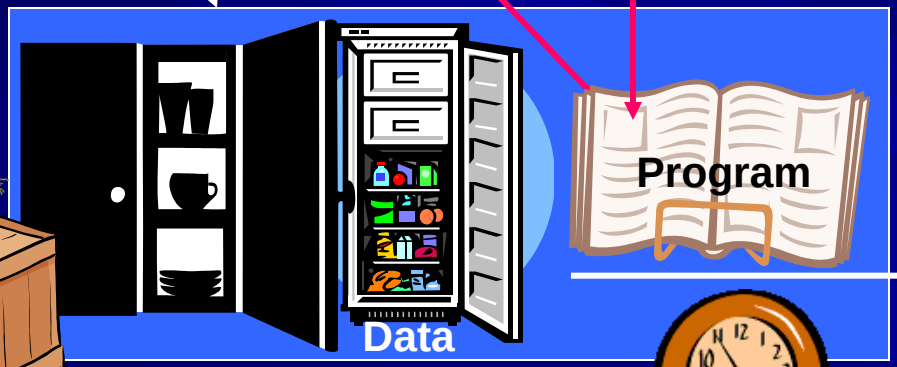
Předpokládejme, že `d` a `f` nejsou NAN

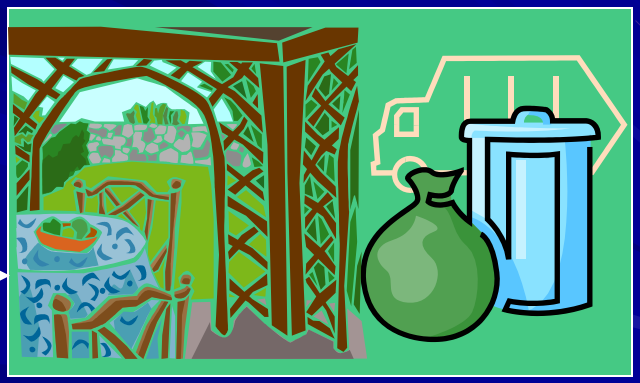
- `x == (int)(float) x` — Ne: 24 významných bitů
- `x == (int)(double) x` — Ano: 53 významných bitů
- `f == (float)(double) f` — Ano : zvýšení přesnosti
- `d == (float) d` — Ne: ztráta přesnosti
- `f == -(-f);` — Ano : pouhá změna znaménka
- `2/3 == 2/3.0` — Ne: 2/3 == 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` — Ano!
- `d > f ⇒ -f > -d` — Ano!
- `d * d >= 0.0` — Ano!
- `(d+f)-d == f` — Ne: Není asociativní

Microprocessors

Processor is something à la Kitchen







Memory

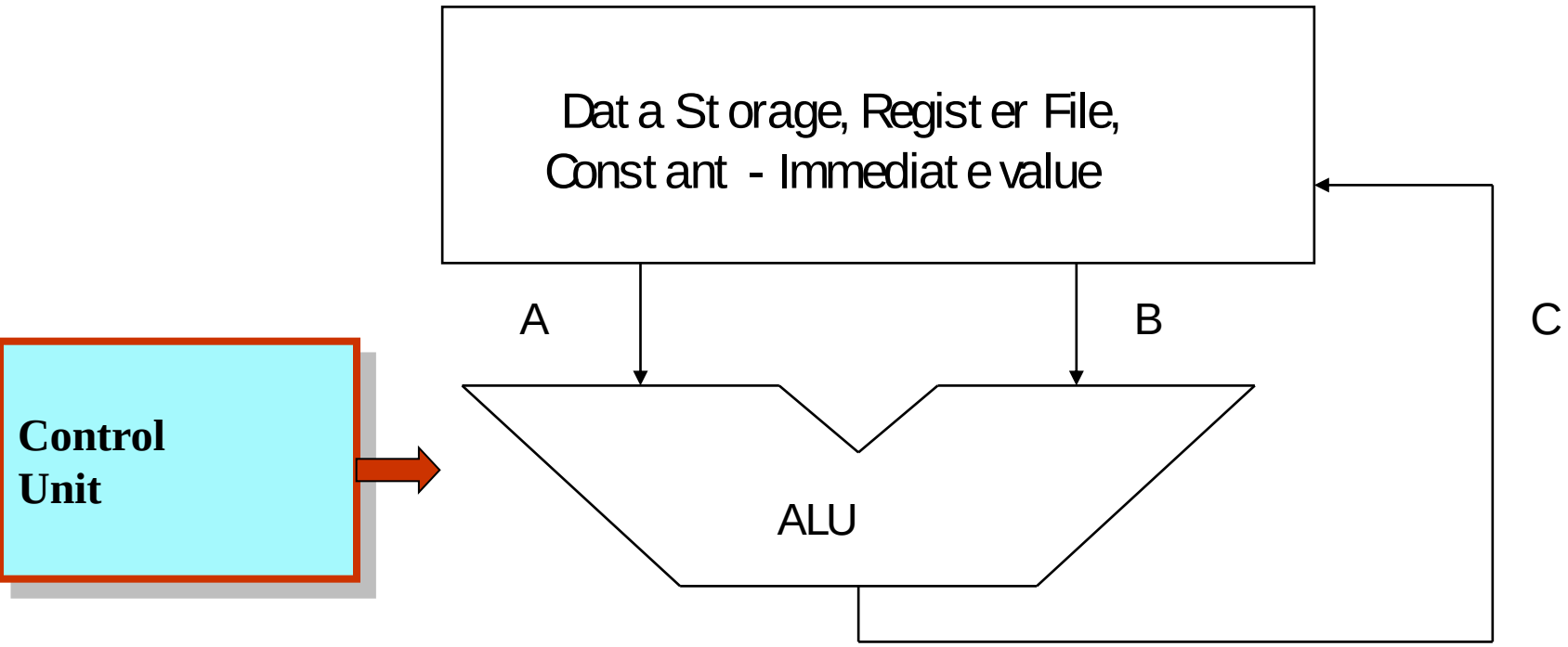
Timer Interrupt

Output

Input

[Nelson: Computer Architecture and Design, Auburn 2008]

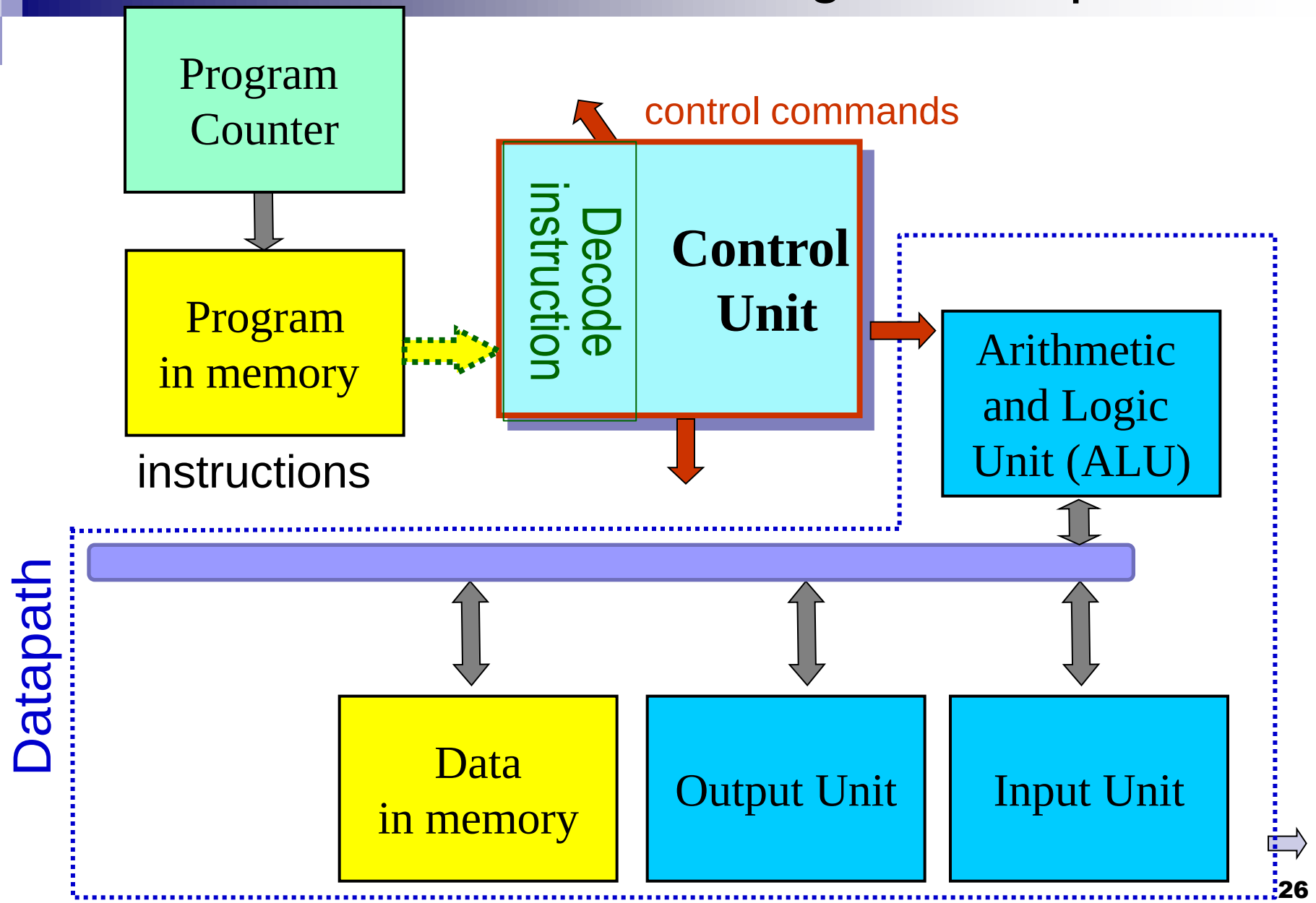
Simplified Processor



Control unit controls datapath

Inherently sequential.

Model of Digital Computer



RISC CPU Design Stategy

RISC - Reduced Instruction Set Computer

Its philosophy - keep it simple!

- **fixed instruction length**(s) (usually one word)
- **load-store** instruction sets (don't do anything else)
- **limited addressing modes**
- **limited operations**

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

Design goals:
speed, size, power consumption, reliability,
cost ← design, fabrication, test, packaging,
space on chip ← embedded systems

CISC Design Strategy

CISC = Complex Instructions Set Computers

Examples of CISC Instruction

Machine	Instruction	Effect
Pentium	`MOVS`	Move string of bytes, words, or double words
PowerPC	`cntlzd`	Count the number of consecutive 0s
IBM 360-370	`CS`	Compare and swap register if a condition is satisfied
Digital VAX	`POLYD`	Evaluation of polynomial using a coefficient table

Počítač podle von Neumanna tvoří

- Řadič
- ALU
- Paměť
- Vstup
- Výstup

Procesor/mikroprocesor

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

V/V podsystém (V/V = I/O)

Řadič - součást (jednotka) počítače/procesoru, která jeho činnost řídí.
Sestává ze dvou částí:

- datové
  - registry,
  - další potřebné obvody,
- vlastní řídicí části, z tzv. jádra řadiče.

Registers

- **Assembly operands are registers**
  - registers are special memory elements inside CPU that allow fast access
  - operations can only be performed on them!
- **MIPS registers are 32 bit wide**
  - they are numbered from $0 to $31
  - Each register can be referred to by its number or defined name:
    - number references: $0, $1, $2, … $30, $31
    - named references: zero, at, v0, ..., fp, ra

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
```

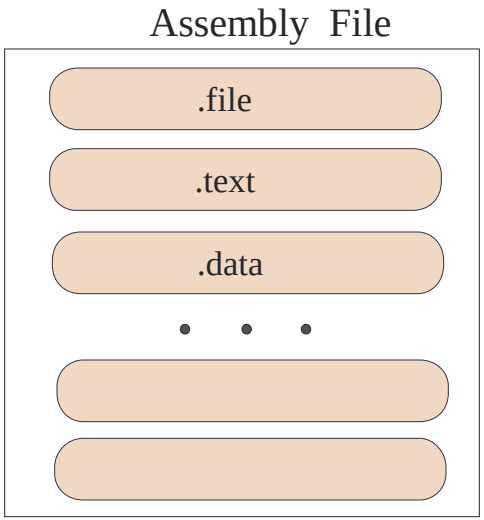
MIPS: Common Register Usage

Reg	Name	Normal usage
$0	**zero**	0x0000_0000 - read only!
$1	**at**	Assembler Temporary
$2	**v0**	Unsaved function arguments and return value
$3	**v1**	
$4	**a0**	
$5	**a1**	
$6	**a2**	
$7	**a3**	
$8	**t0**	Unsaved temporaries
$9	**t1**	
$10	**t2**	
$11	**t3**	
$12	**t4**	
$14	**t5**	
$14	**t6**	
$15	**t7**	

Reg	Name	Normal usage
$16	**s0**	Saved Temporaries
$17	**s1**	
$18	**s2**	
$19	**s3**	
$20	**s4**	
$21	**s5**	
$22	**s6**	
$23	**s7**	
$24	**t8**	
$25	**t9**	
$26	**k0**	Interrupt
$27	**k1**	
$28	**gp**	Global Pointer
$29	**sp**	Stack Pointer
$30	**fp**	Frame Pointer
$31	**ra**	return Address

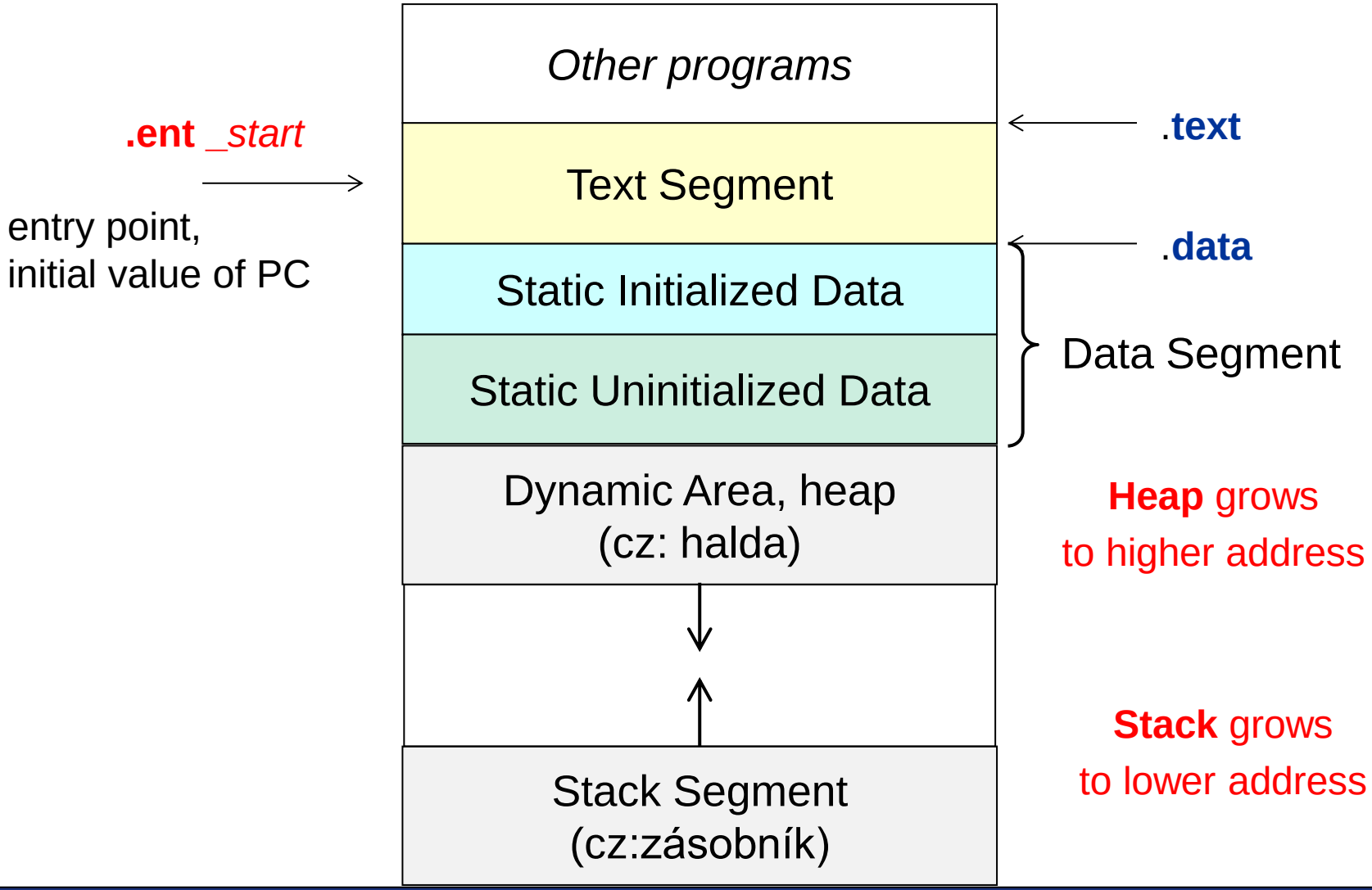
Reserved

Assembly File



- Divided into different sections
- Each section contains some data, or assembly instructions

Layout of a Program in Memory



Assembly code - preview

```
/* 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
```

Assembly code

- Three types of statements in assembly language
  - Typically, one statement should appear on a line

1. Executable Instructions
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do
2. Pseudo-Instructions and Macros
   - Translated by the assembler into real instructions
   - Simplify the programmer task
3. Assembler Directives
   - Provide information to the assembler while translating a program
   - Used to define segments, allocate memory variables, etc.
   - Non-executable: directives are not part of the instruction set

.data directive - Definition Directives

*Sets aside storage in memory for a variable and optionally assigns a **name** (label) to the data*

Syntax:

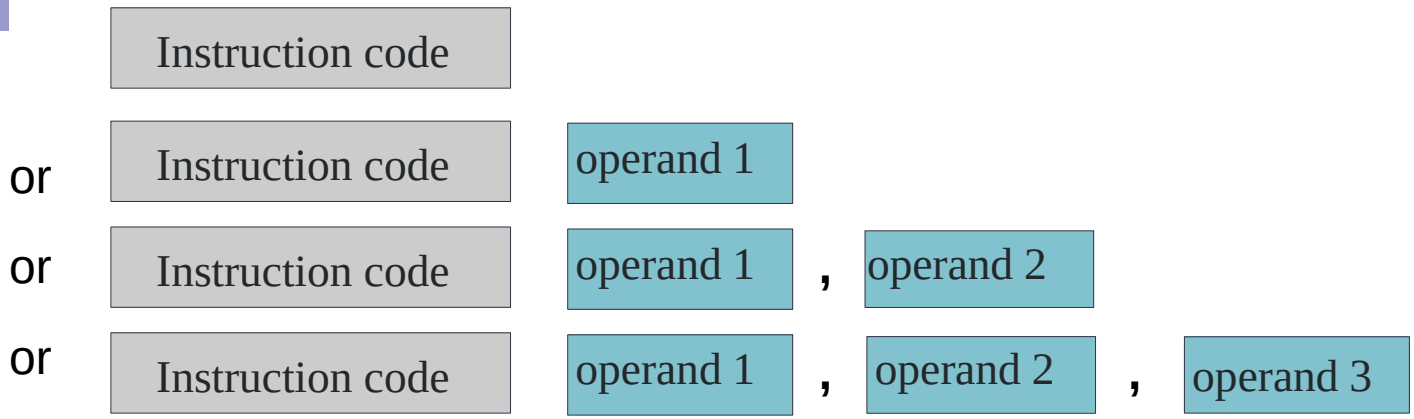
[***name**:*] *directive* *initializer* [, *initializer*] . . .

```
var1:     .word  10
myarray:  .word  5, 3, 4, 1, 15
```

All initializers become binary data in the initialized memory,
we will discuss this topic more in the next lecture. The loacation of text and data can be specified as compiller parameters, e.g.

```
mips-elf-gcc -Wl,-Ttext,0x1000 -Wl,-Tdata,0x2000 -nostdlib -nodefaultlibs -
                nostartfiles -o simple-lw-sw simple-lw-sw.S
```

Structure of instructions



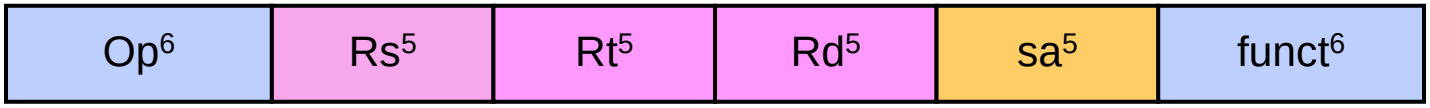
- instruction textual identifier of a machine instruction
- operands
  - register
  - memory location
  - constant (also known as an immediate)

Instruction Formats

All instructions are 32-bit wide.

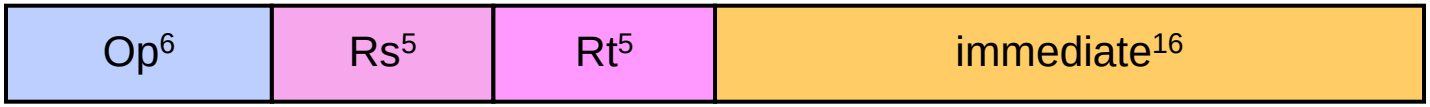
Register (R-Type)

Register-to-register instructions, Rx are numbers of registers
Op: operation code specifies the format of the instruction,
funct- sub-function, control codes
sa - used with the shift and rotate instructions,



Immediate (I-Type)

16-bit immediate constant is a part of the instruction



Jump (J-Type)

Used by jump instructions only

Upper indexes specify a bit width of fields .

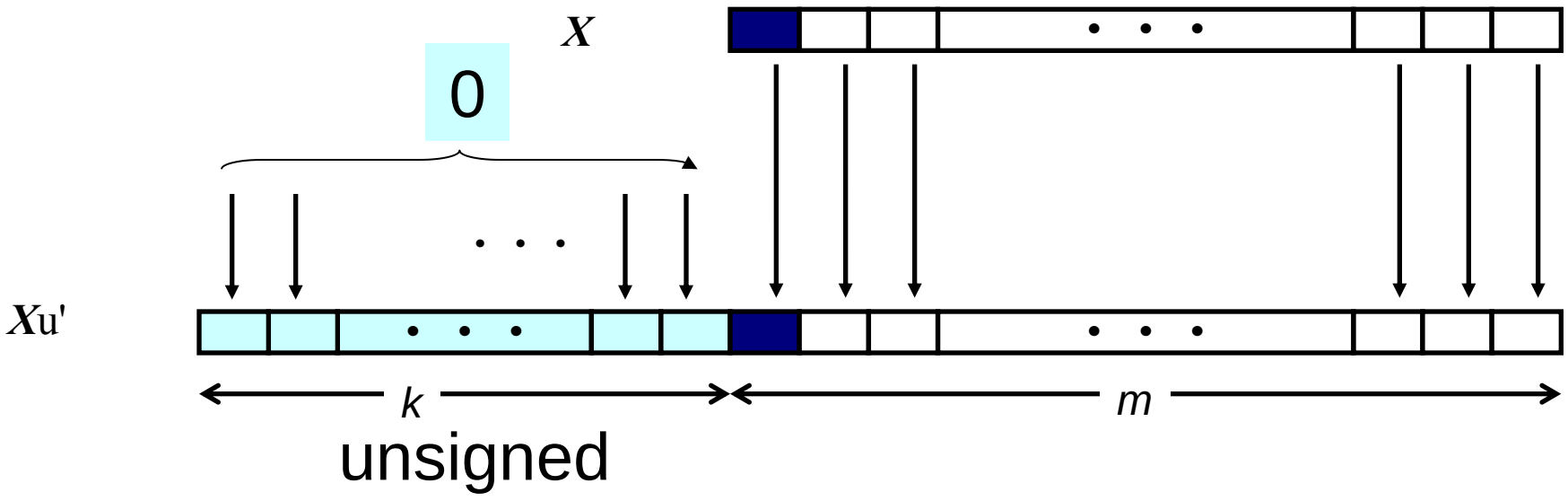
ALU Instructions

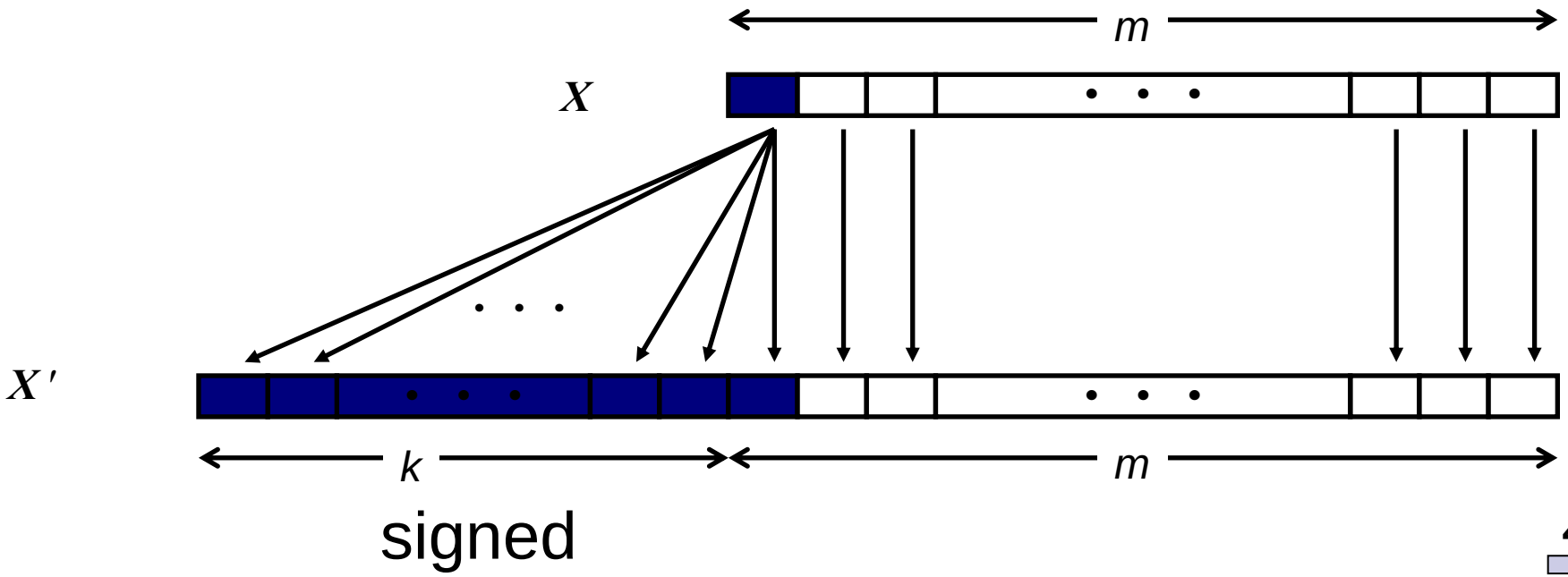
operation	R-format	I-format
add	**add addu**	**addi addiu**
subtract	**sub**	**-**
multiply divide	**mult / multu** **div / divu**	**-**
AND	**and**	**andi**
OR	**or**	**ori**
XOR	**xor**	**xori**
NOR	**nor**	**-**

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

Logic instructions AND, OR, XOR, NOR do not use se = sign-extension

Unsigned/Signed Extension





The Constant Zero

- MIPS register $0 (**zero**) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $8, $9, $zero
```

$8 ← $9

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]

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 - no operation*

NOP binární kód

000000 00000 00000 00000 00000 000000 -- fields of the instruction

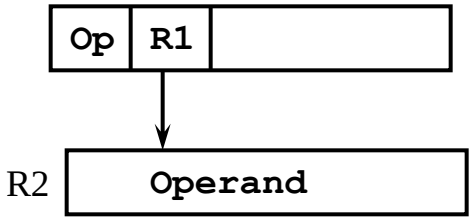
opcode *$0* *$0* *0* *funct* -- meaning of the fields

sll source dest shft sll

MIPS Addressing Modes

(a) Register direct addressing

Register contains the operand




```
or $1, $2, $3
```

(b) Immediate addressing

Instruction contains the operand


```
addi $1, $2, -20
```

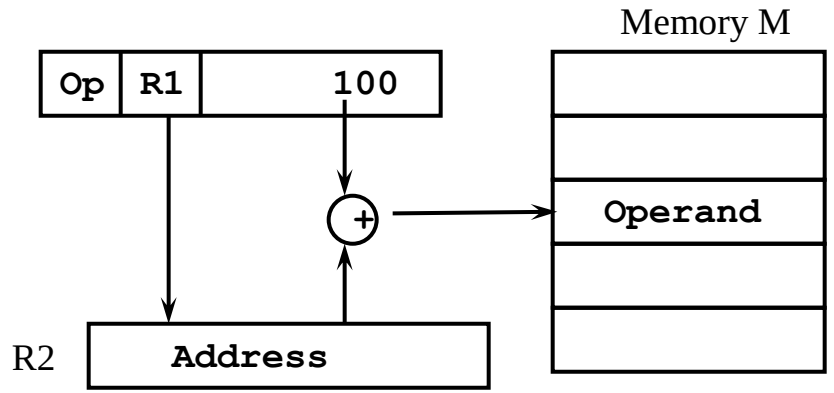
Memory addressing

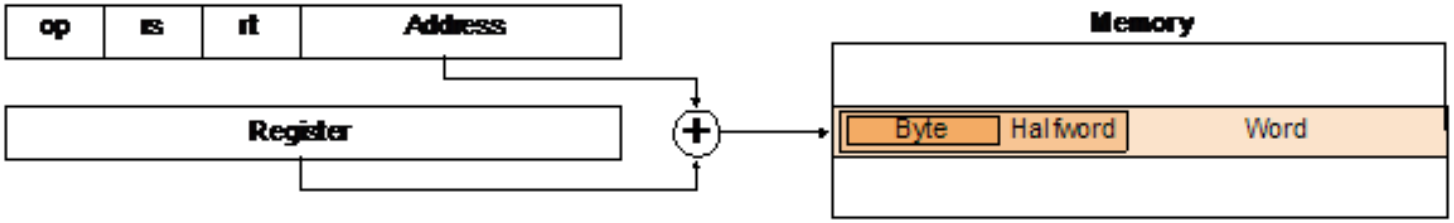
- (c) Displacement (or offset) addressing, it is also called base addressing
  - Address of operand = register + constant

  Memory address in load and store instructions is specified by a base register and offset

```
sw R1, byte_offset(R2)
```

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





Some of MIPS Memory Instruction

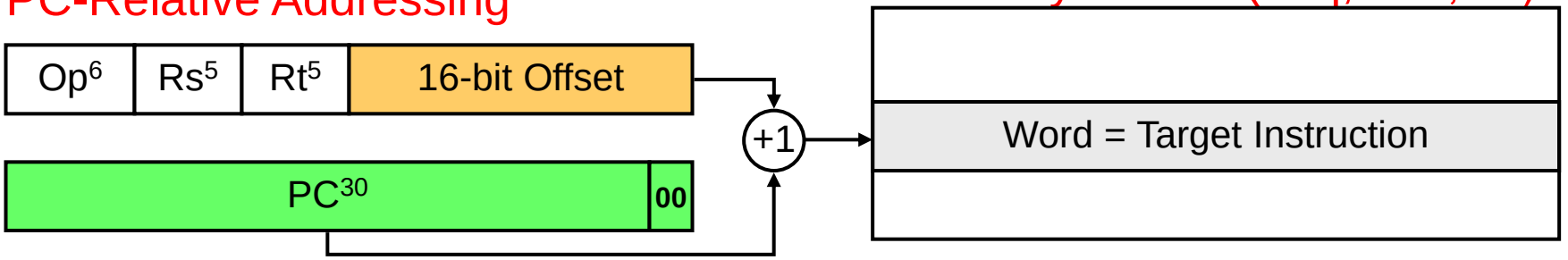
Instr	Syntax	Operace	Význam
lw	lw $t,C($s)	$t = Memory[$s + C]	Load word: Načte slovo z paměti a uloží jej do registru $t
sw	sw $t,C($s)	Memory[$s + C] = $t	Store word: Uloží obsah registru $t do paměti

We will discuss this topic more in the next lecture.

MIPS Jumps

PC-Relative Addressing

Used by branch (beq, bne, …)



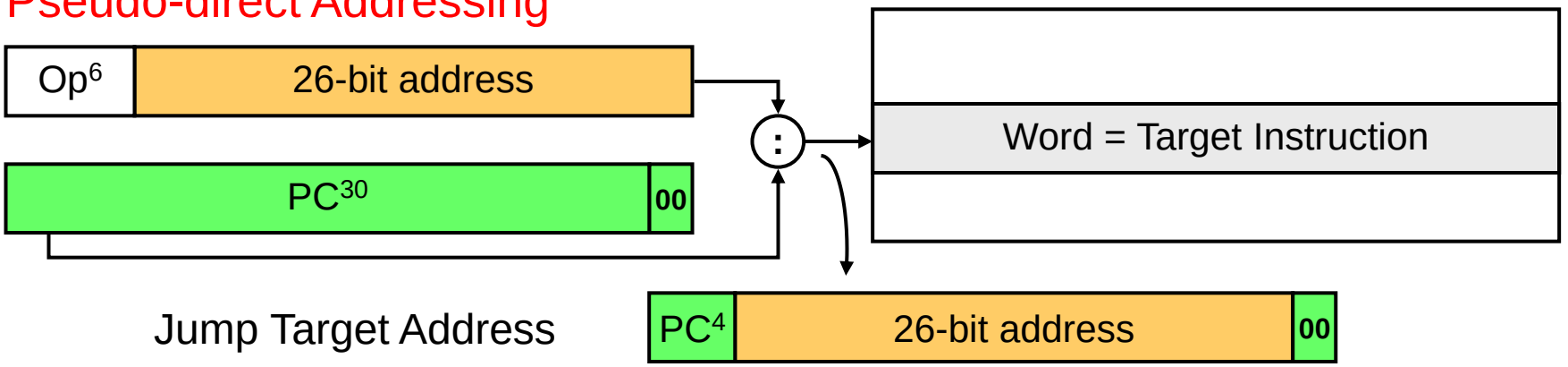
Branch Target Address

PC = PC + 4 × (1 + Offset)

= PC+4+4*offset

Pseudo-direct Addressing

Used by jump instruction



Jump Target Address

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

MIPS Jump Instruction

Instrukce	Syntax	Operace
Branch on not equal: *Skáče, pokud si registry $s a $t nejsou rovny*		
bne	bne $s, $t, offset	if $s != $t goto PC+4+4*offset; else goto PC+4
Branch on equal: *Skáče pokud si registry $s a $t jsou rovny*		
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 Jump Instruction

Instrukce	Syntax	Operace
Set on less than: *Nastaví registr $d=1, pokud platí podmínka $s < $t nebo $s<imm jako signed*		
slt, slti	slt $d,$s,$t slti $d, $s, imm	$d = ($s < $t) $d = ($s < imm)
Set on less than: *Nastaví registr $d=1, pokud platí podmínka $s < $t nebo $s<imm jako unsigned*		
sltu, sltiu	sltu $d,$s,$t sltu $d, $s, imm	$d = ($s < $t) $d = ($s < imm)

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
```