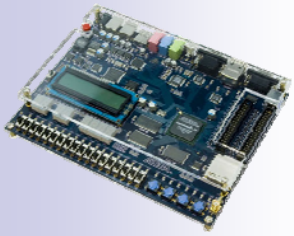
Struktury počítačových systémů

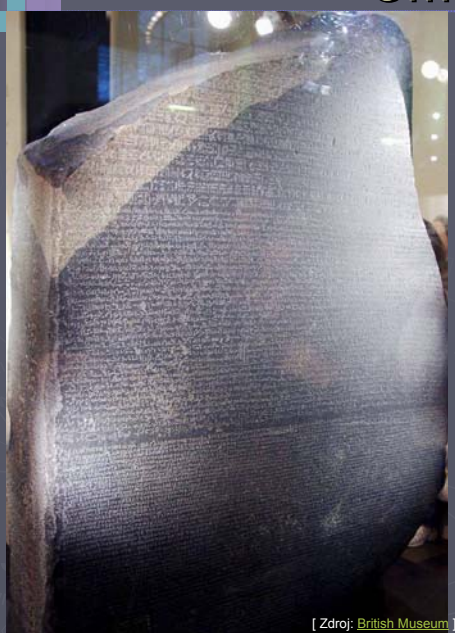
Přednáší: Richard Šusta



1. prosince 2011 verze 1.0

ČVUT-FEL v Praze – kód předmětu A0B35SPS

Omluva



[Zdroj: British Museum]

Dnešní dnešní přednáška bude zas česko-anglická

** žel nemám vše přeložené do angličtiny pro zahraniční studenty, ale ani vše v češtině*

** nicméně jsem upřednostňoval češtinu, pokud byla volba.*

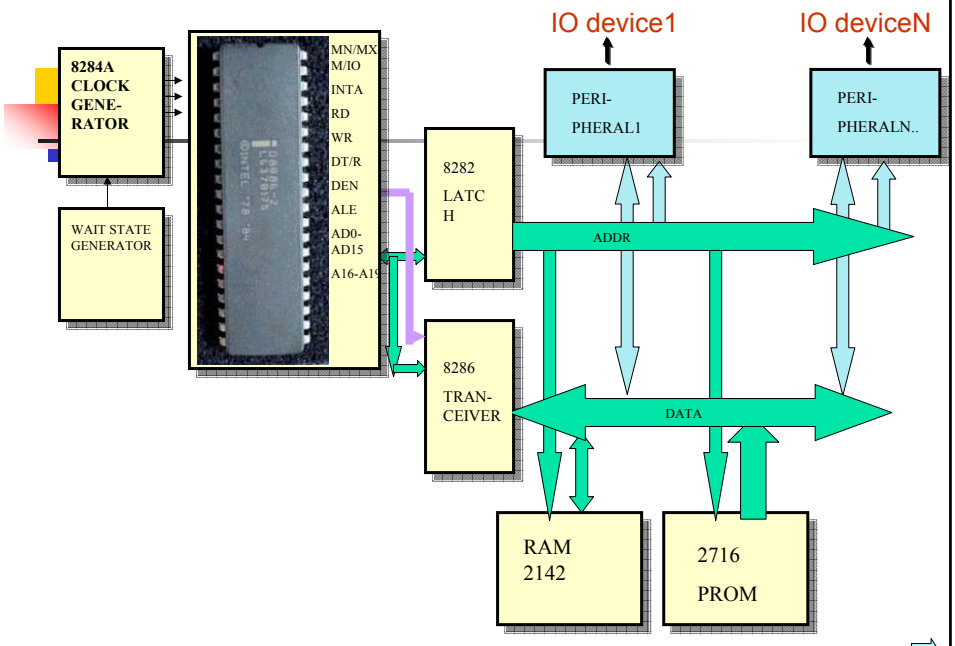
Definition: Microprocessor

- **A Microprocessor is a Single VLSI Chip that has a CPU and may also have some Other Units** (for example, caches, floating point processing arithmetic unit, pipelining, and super-scaling units)

Source	Microprocessor Family
Motorola	68HCxxx
Intel	80x86 & i860
Sun	SPARC
IBM	PowerPC

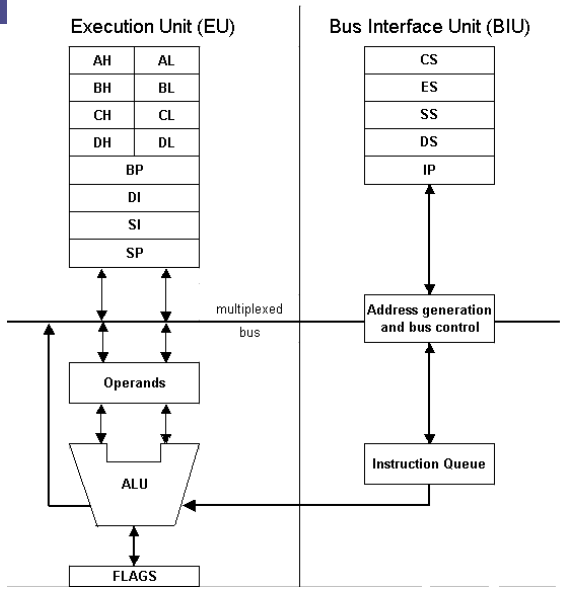
Source: Raj Kamal, Embedded Systems: Architecture, Programming and Design

BASIC 8086 MINIMUM MODE SYSTEM

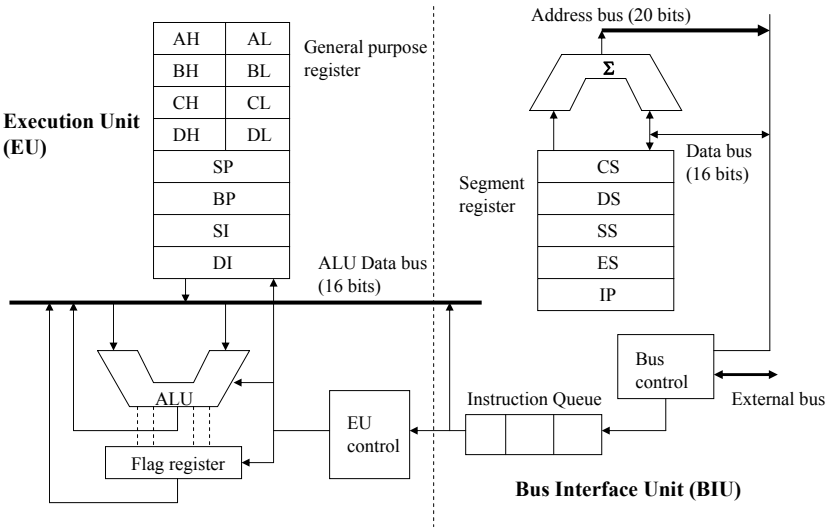


[Source: Intel, co.]

Internal Block Diagram of the 8088/86



Details Intel 8086/8088



[Source: Intel, co.]

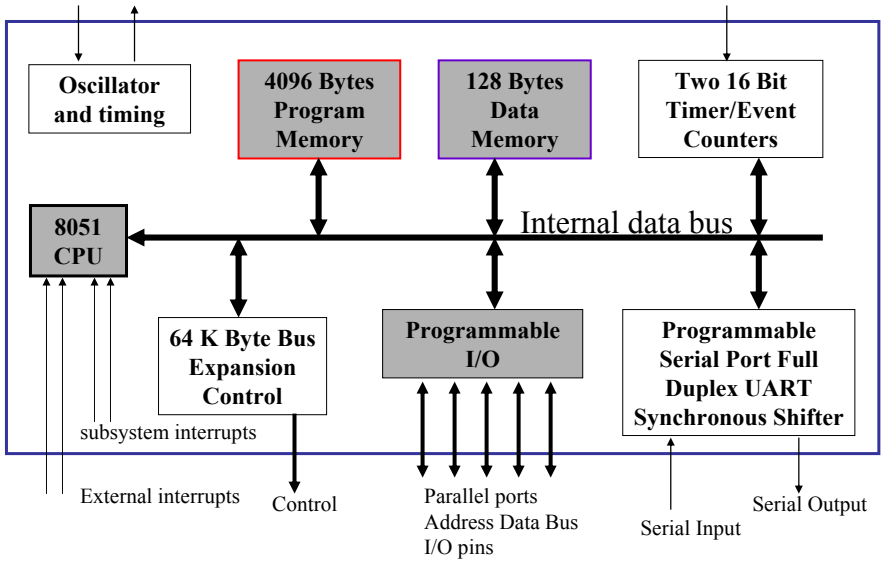
Definition: Microcontroller

- **A microcontroller is a single-chip VLSI unit which, though having limited computational capabilities possesses enhanced input-output capabilities and a number of on-chip functional units**

Source	Microcontroller Family
Motorola	68HC11xx, HC12xx, HC16xx
Intel	80x86, 8051, 80251
Microchip	PIC 16F84, 16F876, PIC18
ARM	ARM9, ARM7

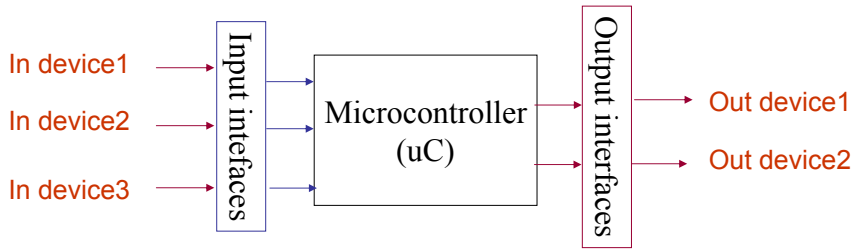
Source: Raj Kamal, Embedded Systems: Architecture, Programming and Design

Example: "Original" 8051 Microcontroller



Source: Raj Kamal, Embedded Systems: Architecture, Programming and Design

Microcontroller



RISC (Reduced Instruction Set Computer) CPU Design Stategy

- RISC philosophy (keep it simple!)
  - **fixed instruction length**(s) (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…

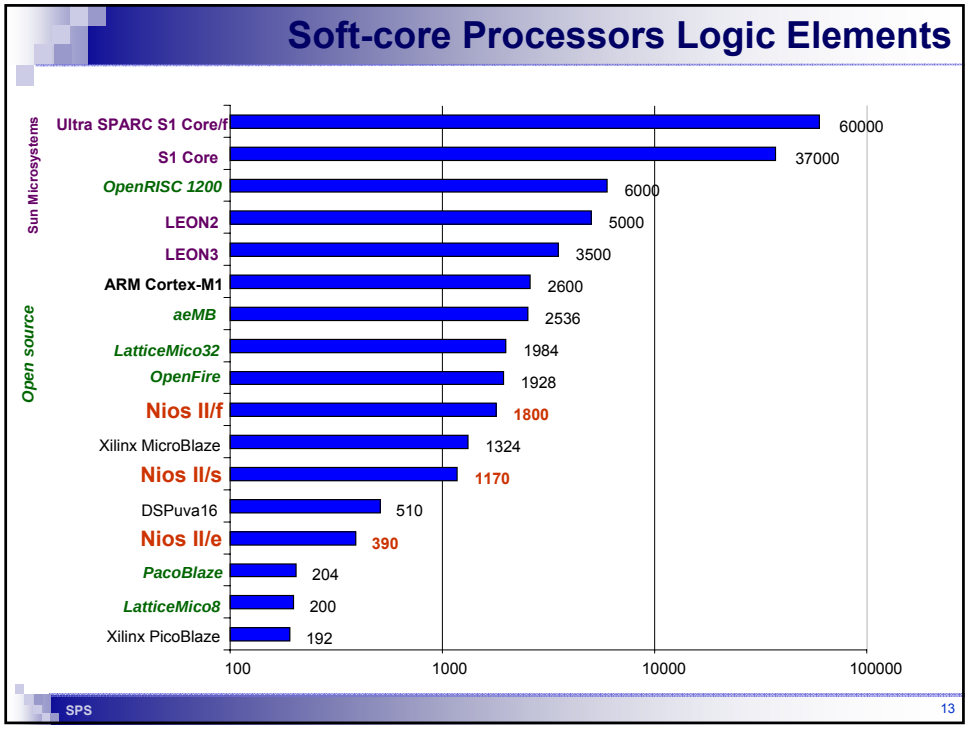
Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

Examples of CISC Instruction (CISC = Complex Instructions Set Computers Design Stategy)

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

Hardcore/Softcore Processors

- Hard-core Processors
  - A Fabricated Integrated Circuit that may or may not be Embedded into Additional Logic
- Soft-core Processors
  - A Processor Described in a HDL that is implemented in Reconfigurable Logic (ie, in an FPGA), e.g NIOS, MicroBlaze, OpenRISC


Soft-core Processors Logic Elements
Sun Microsystems
Open source
Ultra SPARC S1 Core/f
S1 Core
OpenRISC 1200
LEON2
LEON3
ARM Cortex-M1
aeMB
LatticeMico32
OpenFire
Nios II/f
Xilinx MicroBlaze
Nios II/s
DSPuva16
Nios II/e
PacoBlaze
LatticeMico8
Xilinx PicoBlaze
60000
37000
6000
5000
3500
2600
2536
1984
1928
1800
1324
1170
510
390
204
200
192
100
1000
10000
100000
SPS
13


ALTERA.
Nios II: RISC soft-core processor
Copyright © 2005 Altera Corporation

Nios II Versions

- **Nios II Processor Comes In Three ISA (Instruction Set Architecture) Compatible Versions**

– **FAST: Optimized for Speed**

– **STANDARD: Balanced for Speed and Size**

– **ECONOMY: Optimized for Size**

- **Software**
  - Code is Binary Compatible
    - No Changes Required When CPU is Changed

Processor Core Variations

	Nios II /f Fast	Nios II /s Standard	Nios II /e Economy
Pipeline	6 Stage	5 Stage	None
H/W Multiplier & Barrel Shifter	1 Cycle	3 Cycle	Emulated In Software
Branch Prediction	Dynamic	Static	None
Instruction Cache	Configurable	Configurable	None
Data Cache	Configurable	None	None
Logic Requirements (Typical LEs)	1800 w/o MMU 3200 w/ MMU	1200	600
Custom Instructions	Up to 256		

(MMU - Memory Management Unit)

Nios II: Hard Numbers

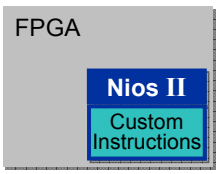
	Nios II/f	Nios II/s	Nios II/e
Stratix II	200 DMIPS @ 175MHz 1180 LEs	90 DMIPS @ 175MHz 800 LEs	28 DMIPS @ 190MHz 400 LEs
Cyclone	100 DMIPS @ 125MHz 1800 LEs	62 DMIPS @ 125MHz 1200 LEs	20 DMIPS @ 140MHz 550 LEs

DMIPS = Dhrystone benchmark MIPS (million instructions per second)
Dhrystone is without float point operations
(Note: Whetstone [*cz brousek*] has float points)

Comparing with Pentium (1996) 224 DMIPS @ 200MHz
cca **196 DMIPS@175MHz** - cca 140 DMIPS@125 MHz

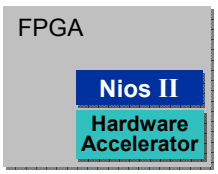
3 Ways to Increase Performance

Custom Instructions



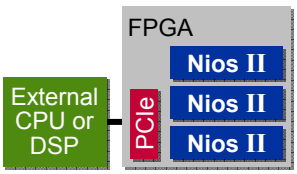
- Accelerate CPU processing performance with application-specific hardware

Hardware Accelerators



- Add external co-processing hardware to accelerate data functions

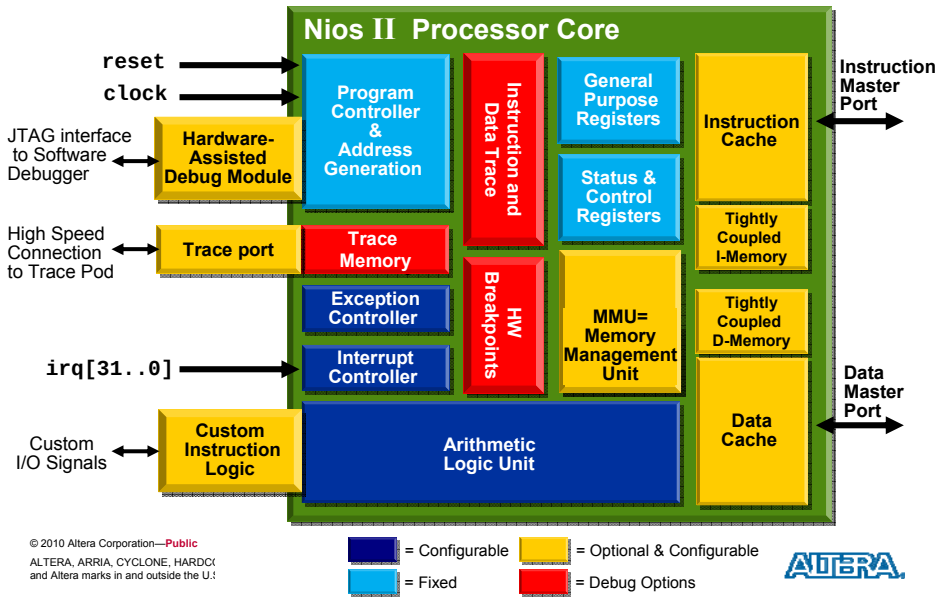
Multi-Processor System



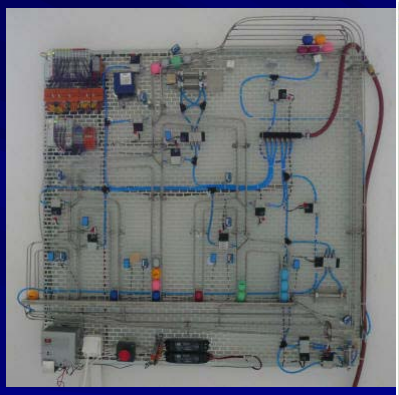
- Add more processors (internal &/or external) to increase processing power

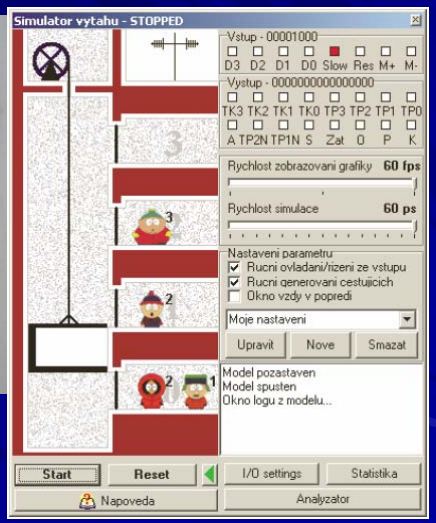
Next lecture

Nios II Processor Configuration



Automat / hardwarový akcelerátor: Kdy se hodí a nehodí?





Aneb jinak: Co umí řešit RTL automaty?

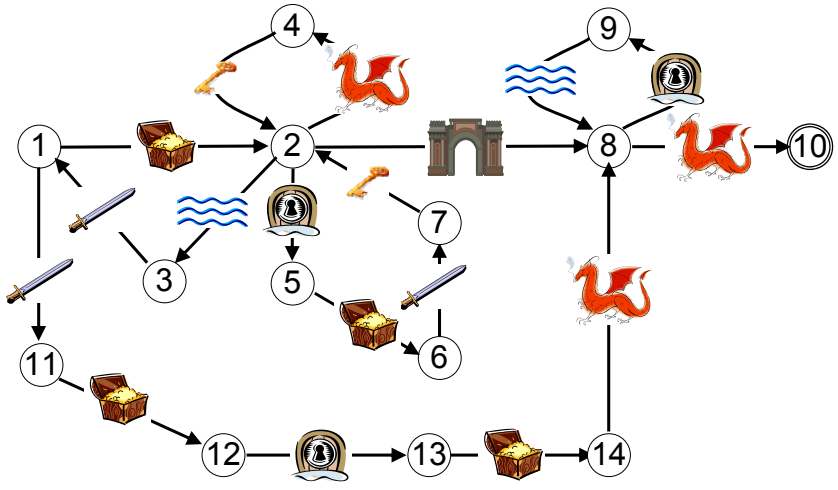
***Jorge Luis Borges, Library of Babel**, 1941*
varianta na Jonathan Swift's Word Machine
v akademii Lagado, kniha III. Gulliverových cest,

Svět je větší, než si člověk dokáže představit.

Počet možných situací je stejně tak veliký.

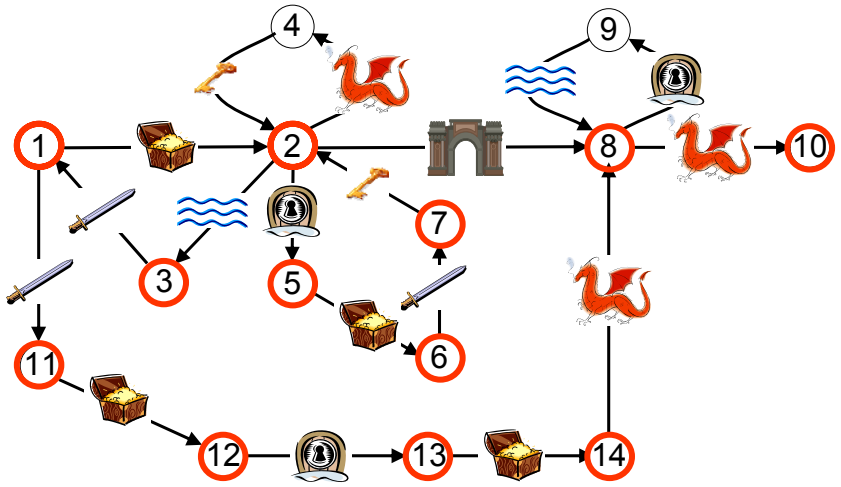
Každá lidská teorie řeší jen část problémů.

Modely regulární, bezkontextové a kontextové úlohy



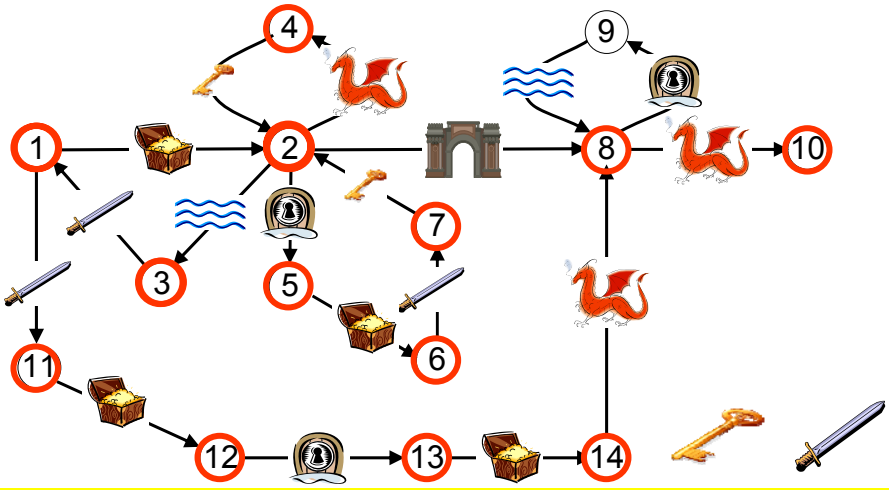
Varianta na motivy Brauer, Holzer1, Konig, Schwoon: The Theory of Finite-State Adventures,
(http://www.fmi.uni-stuttgart.de/szs/publications/koenigba/eatcs79.pdf)

Animace řešení na úrovni automatu



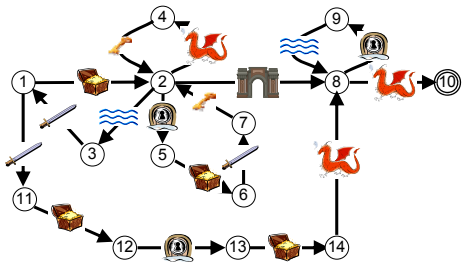
Sbírání pokladů bez omezení na klíče a meče
-> konečný automat

Animace řešení na úrovni push-down automatu



Potřeba meče k zabíjení draků a klíče na 1 použití
bezkontextová úloha *-> push-down automat -*

Ještě složitější



Ještě složitější úloha, tentokrát kontextová

- po zabití draka je meč příliš lepkavý od jeho krve, nelze s ním bojovat, dokud není očištěný ve vodě
- možná další omezení
  + s velkým množství zlata nelze plavat přes vodu
  + zlato nedokážeme zahodit, pokud jsme ho sebrali

Overview Machines versus Languages

Chomsky hierarchy of languages	Machines	Petri nets
regular - *regulární* (type-3)	**finite automata *cz:konečné automaty***	condition/event-systems *cz:podmínka/událost*
context-free - *bezkontextové* (type-2)	push-down automata	Place/transition-nets *cz:distribuce zdrojů*
context-sensitive - *kontextové* (type-1)	linear-bounded Turing machines *omezená páska*	
(type-0)	Turing machines *neomezená páska*	High-level Petri nets *cz:distribuce strukturovaných dat*

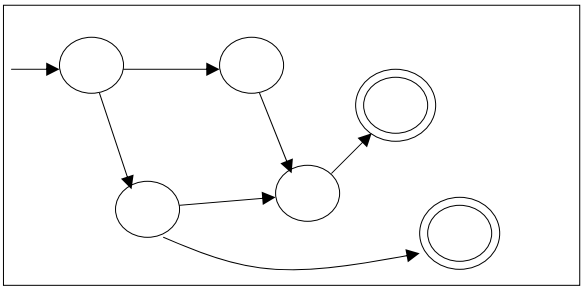
[More: A4BJAG http://math.feld.cvut.cz/demlova/teaching/jag/predn_jag.html**]**

Turing Machine versus Processor

Tape

Read-Write head

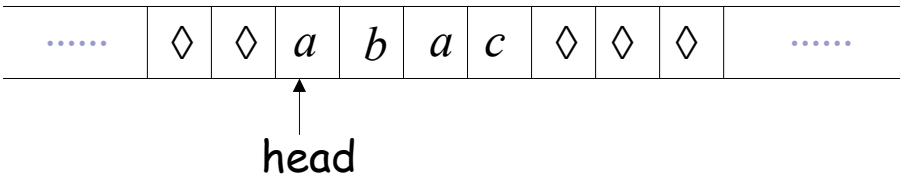
Control Unit



A Turing Machine Head

Input string

Blank symbol



head

The head at each time step:

1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Usage of Turing Machine

- Anything a real computer can compute, a Turing machine can also compute...
- *...but programming of Turing machines is very clumsy and their programs run very slow...*
- *...so Turing machines are not physical objects but mathematical ones suitable only for* ***proving of computability****.*

For more infromation, see AD4M01TAL Theory of Algorithms

Foundation Stones of Assembler

(cz: Základní (úhelné) kameny asembleru)

General Processor Instruction Formats

- Three-Address Instructions
  - ADD o1, o2, o3 — o1 ← o2 + o3
- Two-Address Instructions
  - MOV o1, o2 — o1 ← o2
- One-Address Instructions
  - NEXTPC o1 — o1 ← PC + 4
- Zero-Address Instructions
  - NOP — r1 ← r1 add r0 (NIOS)

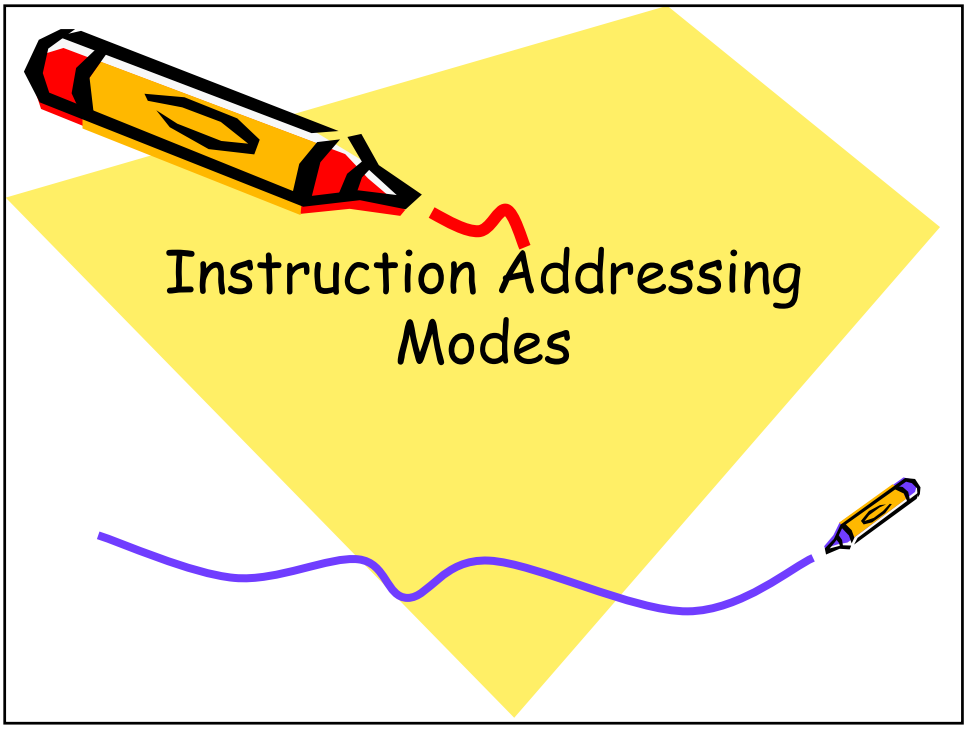
- *R-Type - operands o_i are registers only*
- *I-Type - instruction contains an immediate value*
- *C-Type - control instruction*

NIOS II Opcodes

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x00	call	0x10	cmplti	0x20	cmpeqi	0x30	cmpltui
0x01		0x11		0x21		0x31	
0x02		0x12		0x22		0x32	custom
0x03	ldbu	0x13		0x23	ldbuio	0x33	initd
0x04	addi	0x14	ori	0x24	muli	0x34	orhi
0x05	stb	0x15	stw	0x25	stbio	0x35	stwio
0x06	br	0x16	blt	0x26	beq	0x36	bltu
0x07	ldb	0x17	ldw	0x27	ldbio	0x37	ldwio
0x08	cmpgei	0x18	cmpnei	0x28	cmpgeui	0x38	
0x09		0x19		0x29		0x39	
0x0A		0x1A		0x2A		0x3A	R-Type
0x0B	ldhu	0x1B	flushda	0x2B	ldhuio	0x3B	flushd
0x0C	andi	0x1C	xori	0x2C	andhi	0x3C	xorhi
0x0D	sth	0x1D		0x2D	sthio	0x3D	
0x0E	bge	0x1E	bne	0x2E	bgeu	0x3E	
0x0F	ldh	0x1F		0x2F	ldhio	0x3F	

R-Format Instructions

OPX	Instruction	OPX	Instruction	OPX	Instruction	OPX	Instruction
0x00		0x10	cmplt	0x20	cmpeq	0x30	cmpltu
0x01	eret	0x11		0x21		0x31	add
0x02	roli	0x12	slli	0x22		0x32	
0x03	rol	0x13	sll	0x23		0x33	
0x04	flushp	0x14		0x24	divu	0x34	break
0x05	ret	0x15		0x25	div	0x35	
0x06	nor	0x16	or	0x26	rdctl	0x36	sync
0x07	mulxuu	0x17	mulxsu	0x27	mul	0x37	
0x08	cmpge	0x18	cmpne	0x28	cmpgeu	0x38	
0x09	bret	0x19		0x29	initi	0x39	sub
0x0A		0x1A	srli	0x2A		0x3A	srai
0x0B	ror	0x1B	srl	0x2B		0x3B	sra
0x0C	flushi	0x1C	nextpc	0x2C		0x3C	
0x0D	jmp	0x1D	callr	0x2D	trap	0x3D	
0x0E	and	0x1E	xor	0x2E	wrctl	0x3E	
0x0F		0x1F	mulxss	0x2F		0x3F	



Some of Possible Addressing Modes

Name	Assembler syntax	EA - Effective Address
Immediate	Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (direct)	address	EA = address
Indirect	(R*i*)	EA = [R*i*]
	(LOC)	EA = [LOC]
indirect stack	Opcode	EA = [stack pointer]
Displacement (offset)	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and displacement	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

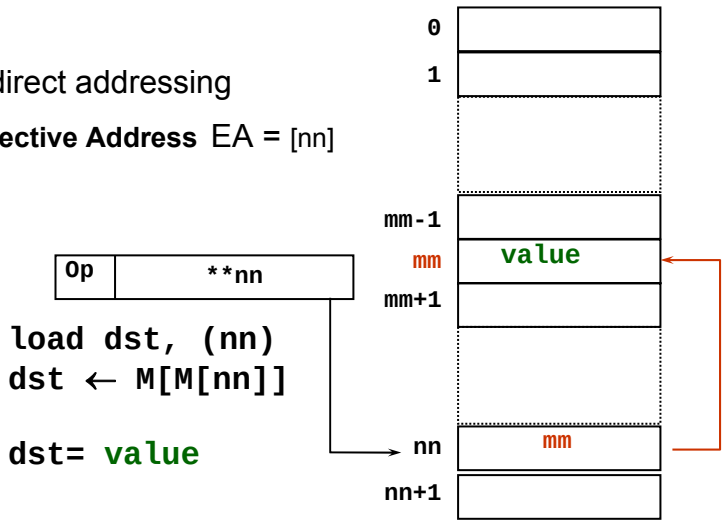
The term ***indexed*** is used when **the constant is the base of an array**, whose member is indexed by the register. The term ***displacement*** or ***offset*** is used when the base is held in a register and added to a **constant offset**.

[nn] - value stored at memory address nn

Principle of Indirect Addressing

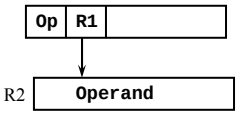
Indirect addressing

Effective Address EA = [nn]



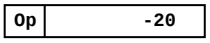
NIOS Addressing Modes

(a) Register direct addressing
Register contains the operand




```
mov R2,R1
R2  ← R1
```

(b) Immediate addressing
Instruction contains the operand

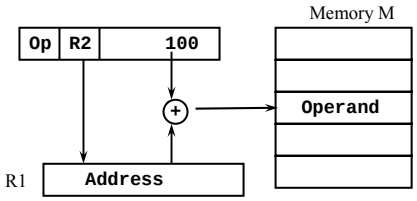



```
movi r2, -20
```

```
stw R2, byte_offset(R1)
```

stw R2, 100(R1) : **R2** → M[**R1**+100]

(c) Displacement (or offset) addressing
Address of operand = register + constant



Load/Store

ldw (load 32-bit word from memory)

ldw rB, im-const(rA) : **rB** ← MEM[**rA + im-const**]

stw (store 32-bit word into memory)

stw rB, im-const (rA) : **rB** → MEM[**rA + im-const**]

Example - demo1.s

```
/*.include "nios_macros.s"*/
.equ LEDR_BASE, 0x10000000
.equ SW_BASE, 0x10000040

.text/* executable code follows */
.global_start
_start:
      /* initialize base addresses of parallel ports */
      movia    r10, SW_BASE/* SW slider switch base address */
      movia    r12, LEDR_BASE/* red LED base address */

LOOP:
    ldwio   r8, 0(r10)
    stwio   r8, 0(r12)
    br      LOOP

.end
```

Comparison with Pentium Addressing Modes

Mode	*Algorithm*
Immediate	Operand = A
Register	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA=linear address ~ EA

[Source: Intel co.]

ALU Instructions

operation	R-format	I-format	
add	`add`	`addi`	
subtract	`sub`	`subi`	
multiply *High 32bits of 64bits*	`mul` `mulxuu, mulxsu mulxss`	`muli`	
AND	`and`	`andi`	`andhi`
OR	`or`	`ori`	`orhi`
XOR	`xor`	`xori`	`xorhi`
NOR	`nor`		

Lower 32bits of 64bit result

High 32bits of 64bit result

pseudo-instruction

`addi rB ← rA + se(number)`

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

NIOS Shift Instructions

shift	dir	Registe	Immediate	note
logical	left	`sll`	`slli`	C << ; *2
logical	right	`srl`	`srli`	C >> ; unsigned/2
arithmetic	right	`sra`	`srai`	signed / 2
rotate	left	`rol`	`roli`	barrel shifter
rotate	right	`ror`	`---`	barrell shifter

`sll rC, rA, rB` `rC ← rA << (rB[4..0])`

Immediate

`slli rC, rA, sh` `rC ← rA << (sh)`

Bitwise Logic Instructions

		Examples:
■ AND	&	n = n & 0xF0;
■ OR	|	n = n | 0xFF00;
■ XOR	^	n = n ^ 0x80;
■ left shift	<<	n = 0xFF << 4;
■ right shift	>>	n = n >> 4
■ NOT	~	n = ~0xFF;

In C, operator << usually behaves as logical for unsigned operand and as arithmetic shift for signed operands.

NIOS and C

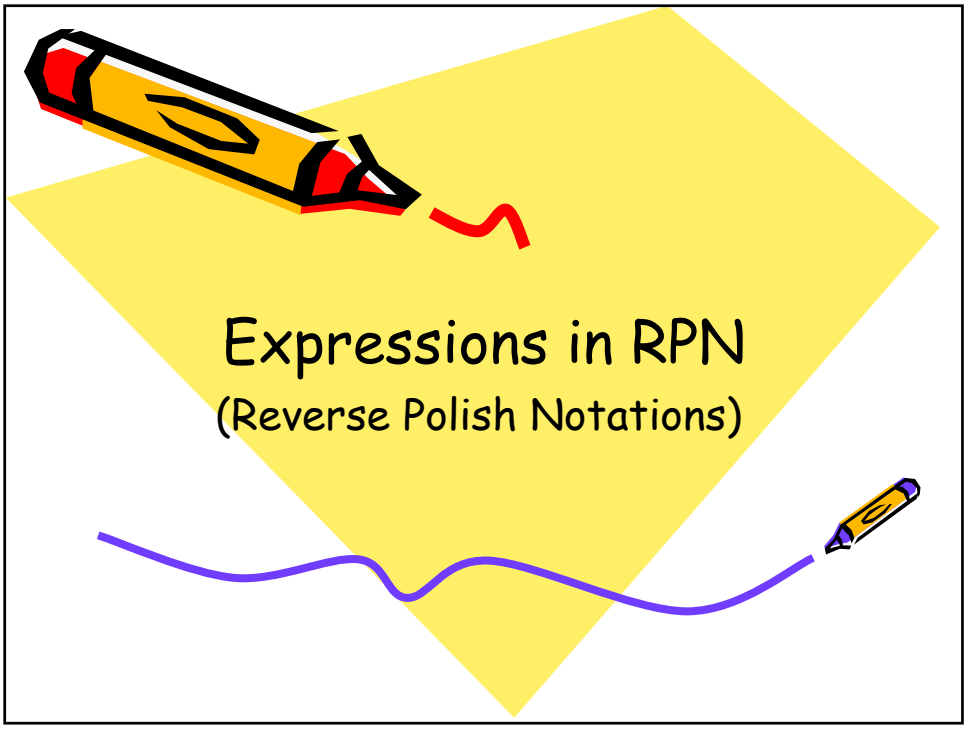
1. ORI r4, r2, 0x30
2. ANDI r4, r2, 0x0F
3. XORI r4, r2, 0b10000000
4. NOR r4, r2, r0 */* r4←not r2 */*
5. SLLI r4, r2, 2
6. SRLI r4, r2, 3
7. SRAI r4, r2, 4
8. *Rotations ROR and ROL has no direct C equivalents.*

```
int x4, x2=-10;
```

1. x4 = x2 | 0x30 ;
2. x4 = x2 & 0x0F ;
3. x4 = x2 ^ 0b10000000;
4. x4 = ~x2
5. x4 = x2<<2;
6. x4 = (unsigned int) x2>>3
7. x4 = (signed int) x2>>3;

Example - demo2.s

```
/*.include "nios_macros.s"*/
.equ LEDR_BASE, 0x10000000
.equ SW_BASE, 0x10000040

.text/* executable code follows */
.global_start
_start:
        /* initialize base addresses of parallel ports */
        movia     r10, SW_BASE/* SW slider switch base address */
        movia     r12, LEDR_BASE/* red LED base address */

LOOP:
     ldwio       r8, 0(r10)
     slli        r8, r8, 1
     stwio       r8, 0(r12)
     br          LOOP

.end
```



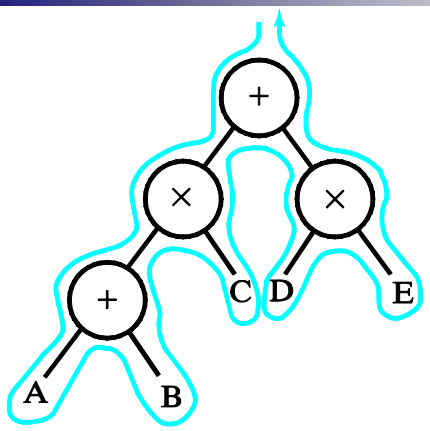
Arithmetic Expressions

Compile: infix->machine language this gets messy because of parentheses. Better way: infix->postfix->machine language

Prefix Notation	Infix Notation	Postfix Notation (RPN)
+A * B C	A + B * C	A B C * +
* + A B C	(A+B) * C	A B + C *
+ – A B C	A – B + C	A B – C +
– A + B C	A – (B+C)	A B C + –

RPN = Reverse Polish Notation

Infix to RPN Conversion



Graphical Conversion

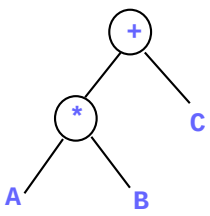
When the path shown traversing the graph passes a variable, that variable is entered into the RPN expression. When the path passes an operator for the final time (up direction), the operation is entered into the RPN expression.

$$(A+B)\times C+(D\times E) \quad \Longrightarrow \quad AB+C\times DE\times +$$

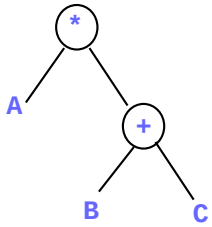
Converting Infix to RPN

By hand: Represent infix expression as an *expression tree*:

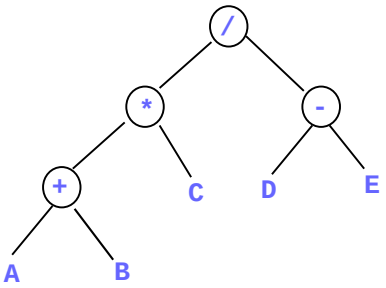
A * B + C

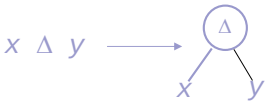


A * (B + C)



((A + B) * C) / (D - E)





Nyhoff, ADTs, Data Structures and Problem Solving with C++, Second Edition, Pearson Education, Inc.

Traverse the tree in *Left-Right-Parent* order (*postorder*) to get RPN:

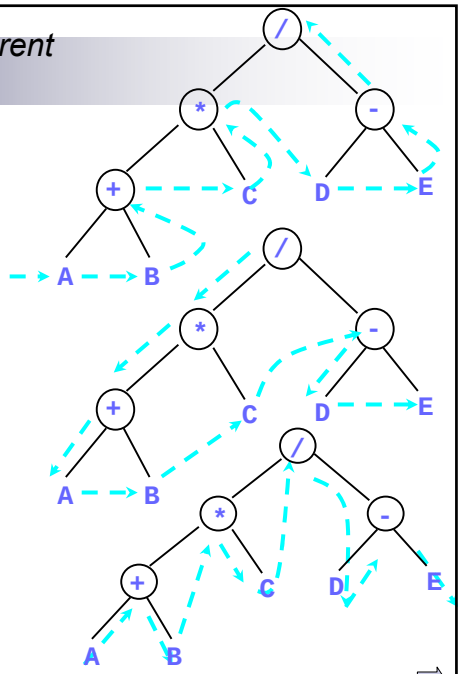
A B + C * D E - /

Traverse tree in *Parent-Left-Right* order (*preorder*) to get **prefix:**

/ * + A B C - D E

Traverse tree in *Left-Parent-Right* order (*inorder*) to get **infix:**
— must insert ()'s

(((A + B) * C) / (D - E))

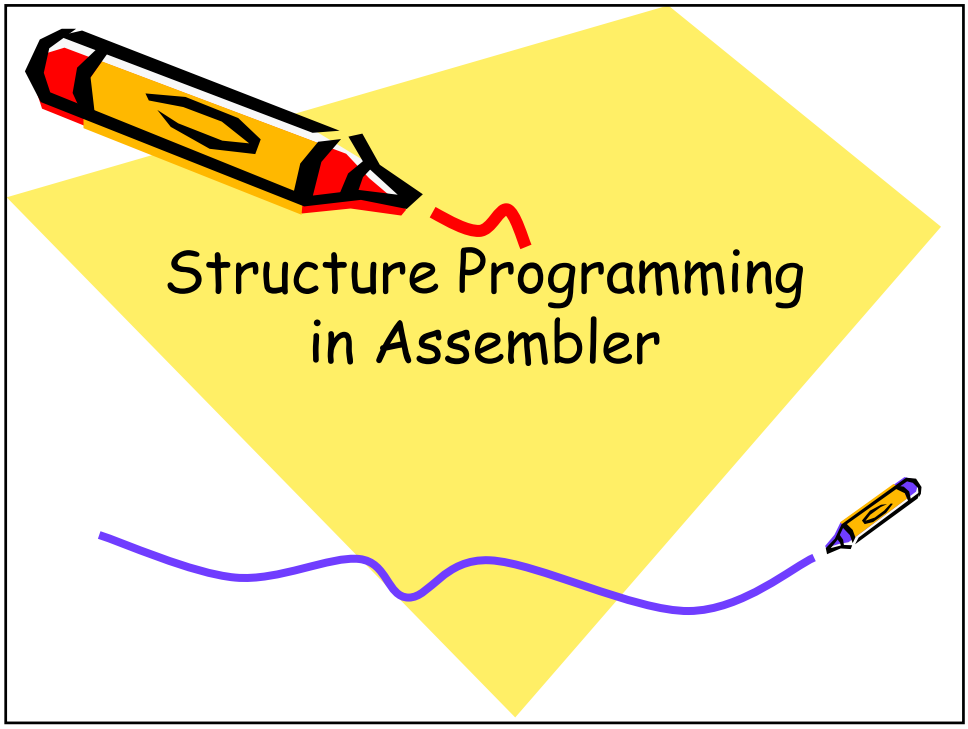


Converting Infix to Postfix

- Analysis:
  - *Operands* are in same order in infix and postfix
  - *Operators* occur later in postfix
- Strategy:
  - Send operands straight to output
  - Send higher precedence operators first
  - Hold pending operators on a stack
  - If same precedence, send in left to right order

Example

Circle for VGA Flag



Comparison Instructions

R-Format		I-Format		code	description
signed	unsigned	signed	unsigned		
`cmpeq`		`cmpeqi`		==	*a* equals *b*
`cmpne`		`cmpnei`		!=	*a* not equal to b
`cmplt`	`cmpltu`	`cmplti`	`cmpltui`	<	*a* less than *b*
`cmpgt`	`cmpgtu`	`cmpgti`	`cmpgtui`	>	*a* greater than *b*
`cmple`	`cmpleu`	`cmplei`	`cmpleui`	<=	*a* not greater than *b*
`cmpge`	`cmpgeu`	`cmpgei`	`cmpgeui`	>=	*a* not less than *b*

pseudo-instruction

`cmplt rC, rA, rB`

if (signed) **`rA`** < (signed) **`rB`**
then **`rC`** ← 1
else **`rC`** ← 0

Branch Instructions

I-Format		code	description
signed	**unsigned**		
br			unconditional
beq		==	*a* equals *b*
bne		!=	*a* not equal to *b*
blt	**bltu**	<	*a* less than *b*
bgt	**bgtu**	>	*a* greater than *b*
ble	**bleu**	<=	*a* not greater than *b*
bge	**bgeu**	>=	*a* not less than *b*

`blt rA, rB, label` pseudo-instruction

if (signed) **rA** < (signed) **rB**
then **PC** ← **PC** + 4 + *se*(**IMMED16**)

Comparison Examples

`cmplt rC, rA, rB`	if (signed) **rA** < (signed) **rB** then **rC** ← 1 else **rC** ← 0
`cmplti rB, rA, IMM16`	if (signed) **rA** < (signed) se(**IMM16**) then **rB** ← 1 else **rB** ← 0
`cmpltu rC, rA, rB`	if (unsigned) **rA** < (unsigned) **rB** then **rC** ← 1 else **rC** ← 0
`cmpltui rB, rA, IMM16`	if (unsigned) **rA** < (unsigned) (0x0000:**IMM16**) then **rB** ← 1 else **rB** ← 0

Branch Details

`blt rA, rB, label`

If (signed) **`rA`** < (signed) **`rB`**, then **`blt`** transfers program control to the instruction at **`label`**.

if (signed) **`rA`** < (signed) **`rB`**
then **`PC`** ← **`PC`** + 4 + *se*(**`IMMED16`**)

In the instruction encoding, the offset given by **`IMM16`** is treated as a signed number of bytes relative to the instruction immediately following the branch instruction. The two least significant bits of **`IMM16`** are always zero, because instruction addresses must be word-aligned.

Our processor is word-addressed, so we do on branching

`PC` ← **`PC`** + 4 + se(**`immed16)`**

Unconditional Branch

`br label`

The unconditional branch instruction has **`rA`** and **`rB`** set equal to zero. This means that the instruction can be implemented as: **`beq zero, zero, label`**

In our processor, the unconditional branch could have been a pseudo-instruction; it has a separate op-code in the NIOS II processor.

Example 2

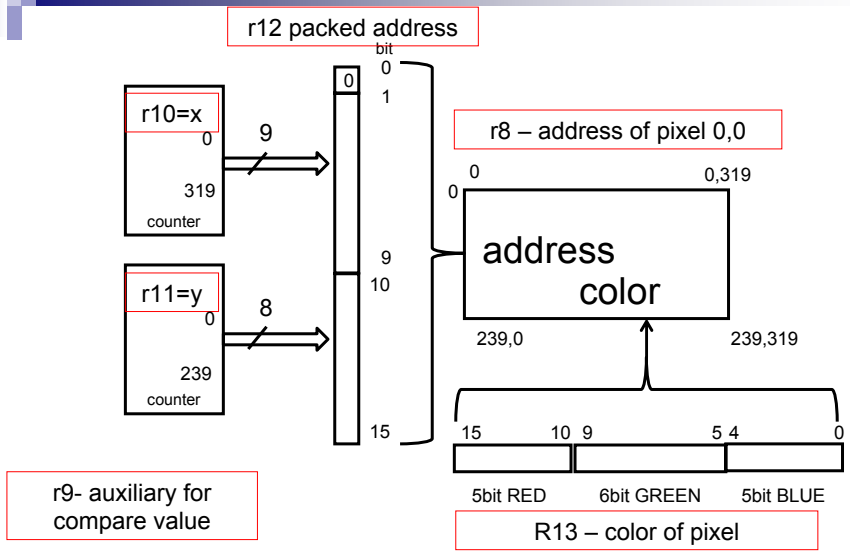
If-then-else; do-while; for; while; switch constructions in assembler

Example3 - demo3.s

```
.equ LEDR_BASE, 0x10000000
.equ SW_BASE, 0x10000040

.text/* executable code follows */
.global_start
_start:
  movia r10, SW_BASE
  movia  r12, LEDR_BASE
```

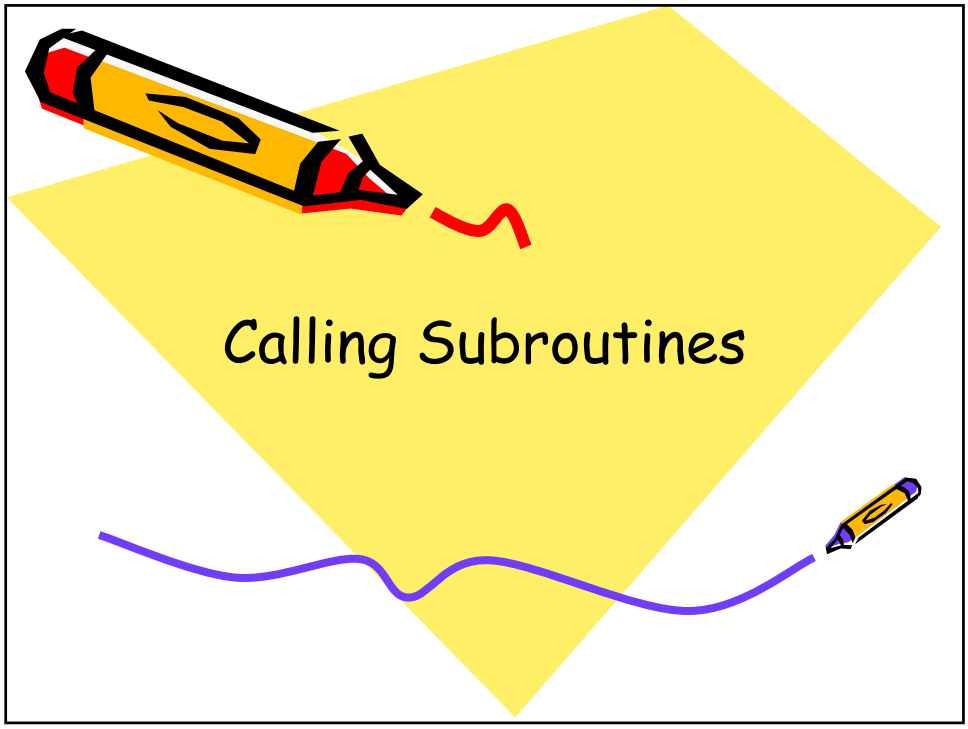
```
.
LOOP:
   ldwio       r8, 0(r10)
   movia       r16,17

    stwio      r8, 0(r12)
    movia      r16, 10000000
WAIT1:
    addi        r16, r16, -1
    bne         r16, r0, WAIT1

    roli        r8, r8, 1
    stwio       r8, 0(r12)
    movia      r16, 10000000
WAIT2:
    addi     r16, r16, -1
    bne     r16, r0, WAIT2
    br       LOOP
.end
```

Last lecture: Registers in VGA demo

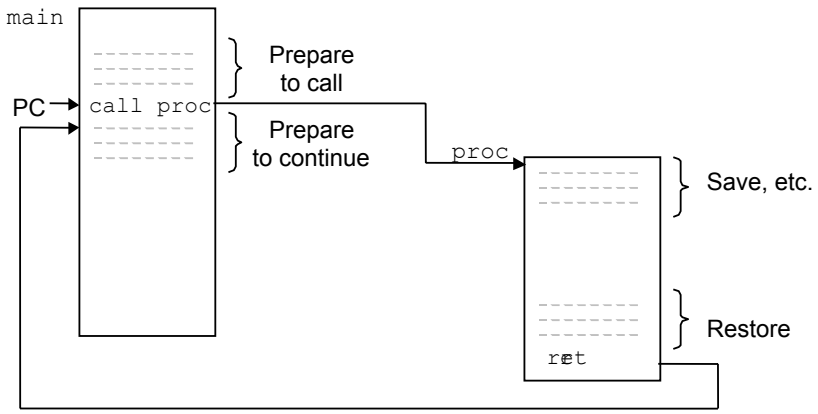


Last lecture: DEMO VGA program

```
.equ PIXELBUF,0x8000000  /* end = 0x803BE7E */
.equ PIXEL_X_END,319  /* width 0..319 */
.equ PIXEL_Y_END,239  /* height 0..239 */

.equ RED,  0xF800   /* 5 bit red */
.equ GREEN, 0x7E0   /* 6 bit green */
.equ BLUE, 0x1F     /* 5 bit blue */

.equ YSHIFT,9
.equ XYSHIFT,1
.text
        .global _start
_start:
       movia   r8, PIXELBUF   /* video memory */
       mov         r10, r0                        /* X */
       mov         r11, r0       /* Y */
       movia   r13, BLUE

LOOP:
  /* pack y-x into address in PIXELBUF */
       slli    r12, r11, YSHIFT /* shift left logical */
       or      r12, r12, r10
       slli    r12, r12, XYSHIFT
       add     r12, r12, r8   /* add PIXEL BUFFER */
       /* we create color frame for proving
          correct ranges of x-y loop */
       movia   r13, BLUE         /* 0 column blue  */
       beq     r10, r0, NOWHITE /* branch if r10=r0 */
       movia   r13, GREEN
       movia   r9, PIXEL_X_END    /* x-last column green */
        beq     r10, r9, NOWHITE

       movia   r13, RED      /* 0 line red  */
       beq     r11, r0, NOWHITE
       movia   r13, RED | GREEN
       movia   r9, PIXEL_Y_END            /* the last line yellow */
        beq     r11, r9, NOWHITE

       movia   r13, RED | GREEN | BLUE/* white color */

NOWHITE:
       sth     r13, 0(r12)       /* store pixel */
       addi    r10, r10, 1       /* increment x */
       movia   r9, PIXEL_X_END
        bleu     r10, r9, LOOP
       mov     r10, r0         /* next line, x=0 */
       addi    r11, r11, 1     /* increment line */
       movia   r9, PIXEL_Y_END
       bleu      r11, r9, LOOP
STOP:
       br STOP   /* You flag is finished. */
.end
```



Call procedure



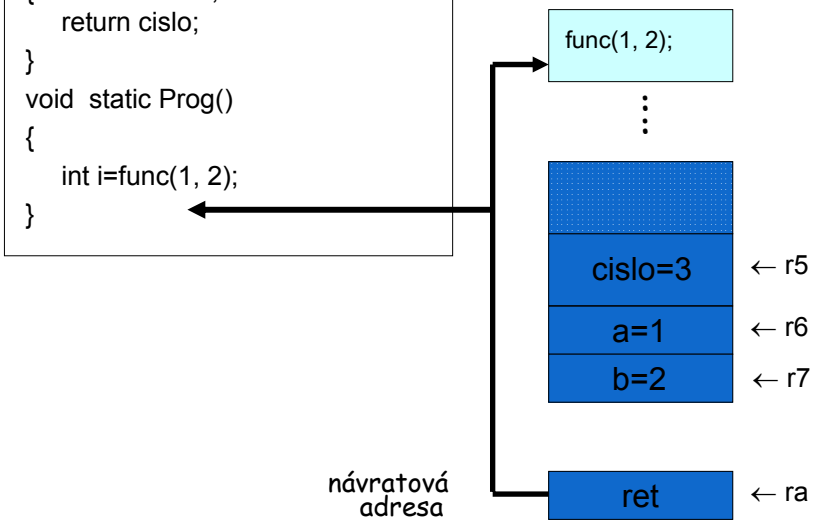
NIOS: Common Register Usage

Reg	Name	Normal usage
r0	**zero**	0x0000_0000
r1	**at**	Assembler Temporary
r2		Return Value (least-significant 32 bits)
r3		Return Value (most significant 32 bits)
r4		Register Arguments (First 32 bits)
r5		Register Arguments (Second 32 bits)
r6		Register Arguments (Third 32 bits)
r7		Register Arguments (Fourth 32 bits)
r8		**Caller**-Saved
r9		General-Purpose Registers
r10		
r11		
r12		
r14		
r14		
r15		

Reg	Name	Normal usage
r16		**Callee**-Saved
r17		General-Purpose Registers
r18		
r19		
r20		
r21		
r22		
r23		
r24	**et**	Exception Temporary
r25	**bt**	Break Temporary
r26	**gp**	Global Pointer
r27	**sp**	Stack Pointer
r28	**fp**	Frame Pointer
r29	**ea**	Exception Return Address
r30	**ba**	Break Return Address
r31	**ra**	Return Address

Animace volání funkce NIOS bez SP

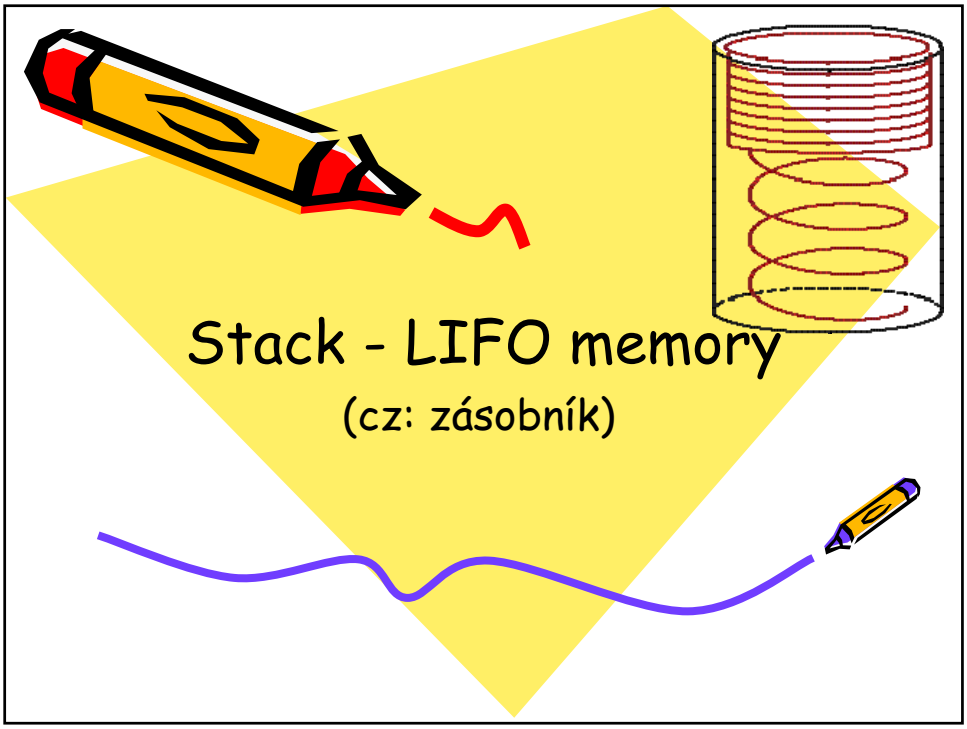
```
int static funkce(int a, int b)
{   int cislo=a+b;
    return cislo;
}
void  static Prog()
{
    int i=func(1, 2);
}
```



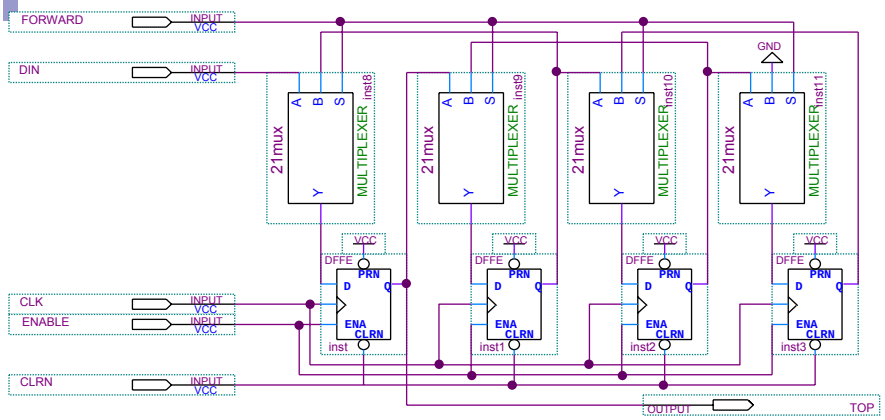
Example - demo4.s

```
.equ LEDR_BASE, 0x10000000
.equ SW_BASE, 0x10000040

.text/* executable code follows */
.global_start
_start:
  movia r10, SW_BASE
  movia  r12, LEDR_BASE
```

```
.
LOOP: ldwio  r8, 0(r10)
        stwio  r8, 0(r12)
         call    WAITING
         roli     r8, r8, 1
         stwio   r8, 0(r12)
         call     WAITING
         br       LOOP

 WAITING:
        movia     r16, 10000000
WAIT1:
        addi      r16, r16, -1
        bne       r16, r0, WAIT1
        ret
.end
```



4bit LIFO - Stack



LIFO = bidirectional shift register

	DIN=1	DIN=1	DIN=1	DIN=1	DIN=1	DIN=1	DIN=1	DIN=1	DIN=1
TOP	0	1	1	1	1	1	1	1	0
	0	0	1	1	1	1	1	0	0
	0	0	0	1	1	1	0	0	0
	0	0	0	0	1	0	0	0	0

LIFO

```
library ieee;use ieee.std_logic_1164.all;
entity LIFO is generic ( NUM_STAGES : natural := 4);
    port  ( din, clk, enable, forward, clrn  : in std_logic;  top: out std_logic );
end entity;
architecture rtl of LIFO is
  type sr_length is array ((NUM_STAGES-1) downto 0) of std_logic;
  signal sr: sr_length;
  begin process (clk, clrn)
    begin  if (clrn = '0') then sr <= (others=>'0');
         elsif (rising_edge(clk)) then
           if (enable = '1') then
              if (forward = '1') then -- shift forward, bottom data is lost
                sr((NUM_STAGES-1) downto 1) <= sr((NUM_STAGES-2) downto 0);
                 sr(0) <= din;
              else -- shift data backward, top data is lost
                sr((NUM_STAGES-2) downto 0) <= sr((NUM_STAGES-1) downto 1);
                 sr(NUM_STAGES-1) <= '0';
             end if;
           end if;
    end if;
   end process;
 top <= sr(0);end rtl;
```

NIOS: Common Register Usage

Reg	Name	Normal usage
r0	**zero**	0x0000_0000
r1	**at**	Assembler Temporary
r2		Return Value (least-significant 32 bits)
r3		Return Value (most significant 32 bits)
r4		Register Arguments (First 32 bits)
r5		Register Arguments (Second 32 bits)
r6		Register Arguments (Third 32 bits)
r7		Register Arguments (Fourth 32 bits)
r8		**Caller**-Saved
r9		General-Purpose Registers
r10		
r11		
r12		
r14		
r14		
r15		

Reg	Name	Normal usage
r16		**Callee**-Saved
r17		General-Purpose Registers
r18		
r19		
r20		
r21		
r22		
r23		
r24	**et**	Exception Temporary
r25	**bt**	Break Temporary
r26	**gp**	Global Pointer
r27	**sp**	Stack Pointer
r28	**fp**	Frame Pointer
r29	**ea**	Exception Return Address
r30	**ba**	Break Return Address
r31	**ra**	Return Address

NIOS Macro

```
.equ SPINIT, 0x800000
.macro push reg
    subi sp, sp, 4
    stw \reg, 0(sp)
.endm
.macro pop reg
    ldw \reg, 0(sp)
    addi sp, sp, 4
.endm
```

```
.text
  .global _start
  _start:
    movia  sp, SPINIT
```

GNU assembler .macro

- Deklaration:

```
MACRO   < nameOfMacro> <formal parameters>
        <body>
ENDM
```

- Removing:

```
PURGE   < nameOfMacro1> {,< nameOfMacro2>}
```

- Parameters:
  - comma separated list of parameters
  - paramaters inside body must be proceeded by \
- Usage:

```
<nameOfMacro> <param> {,<param>}
```

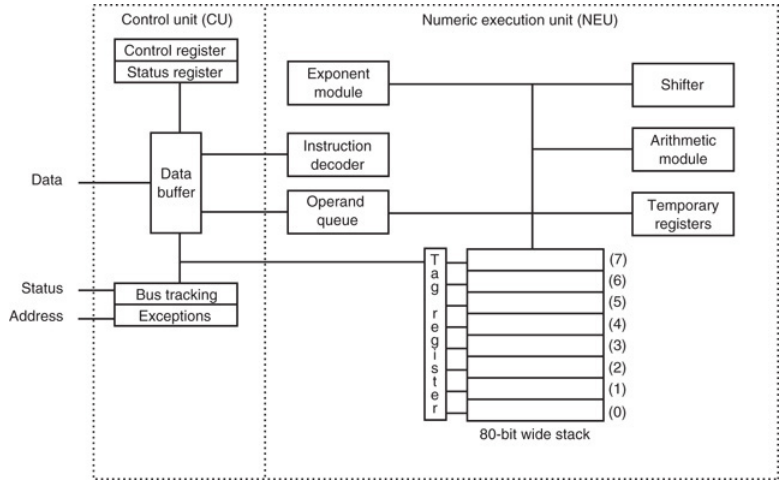
demo5.s

```
LOOP:   ldwio   r8, 0(r10)
        movia   r16, LED_COUNT
NEXT:   stwio   r8, 0(r12)
        call    WAITING
        roli    r8, r8, 1
        addi    r16, r16,-1
        bne     r16, r0, NEXT
        br      LOOP

WAITING: push   r16     -- protect register
        movia   r16, 10000000
WAIT1:  addi    r16, r16, -1
        bne     r16, r0, WAIT1
        pop     r16
        ret
```

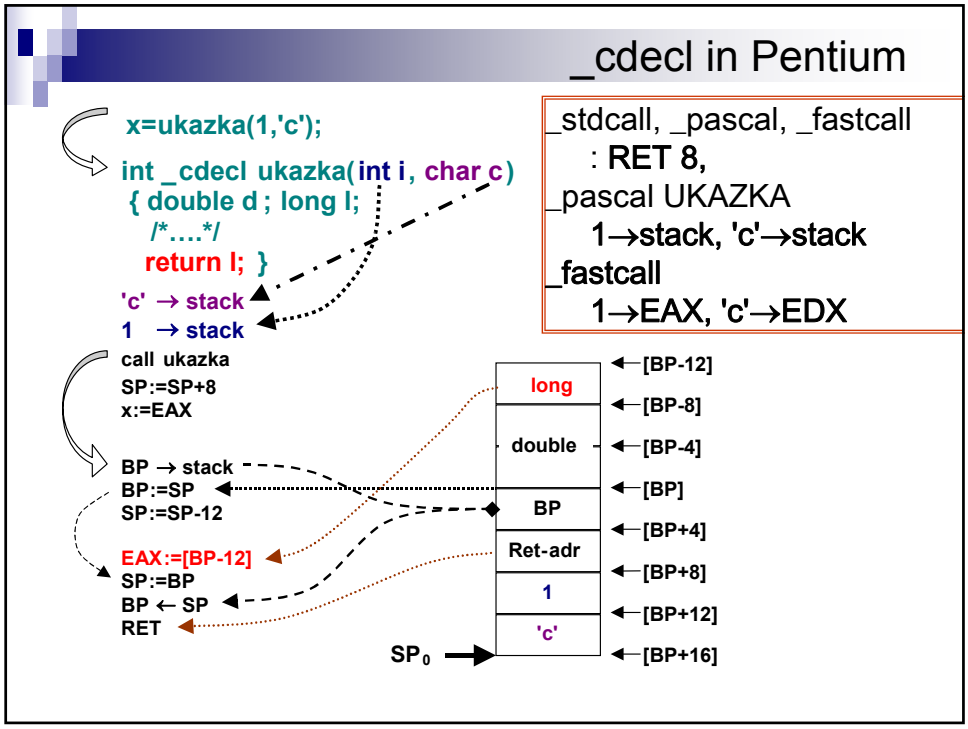
Frequent Stacks in Programs

- Software data stacks
  - general stack for program and data - SP
  - stack for handling exceptions
  - data stack for heap
- Hardware data stacks
  - numeric cooprocessor

Hardware Data Stack in Numeric Cooprocessor

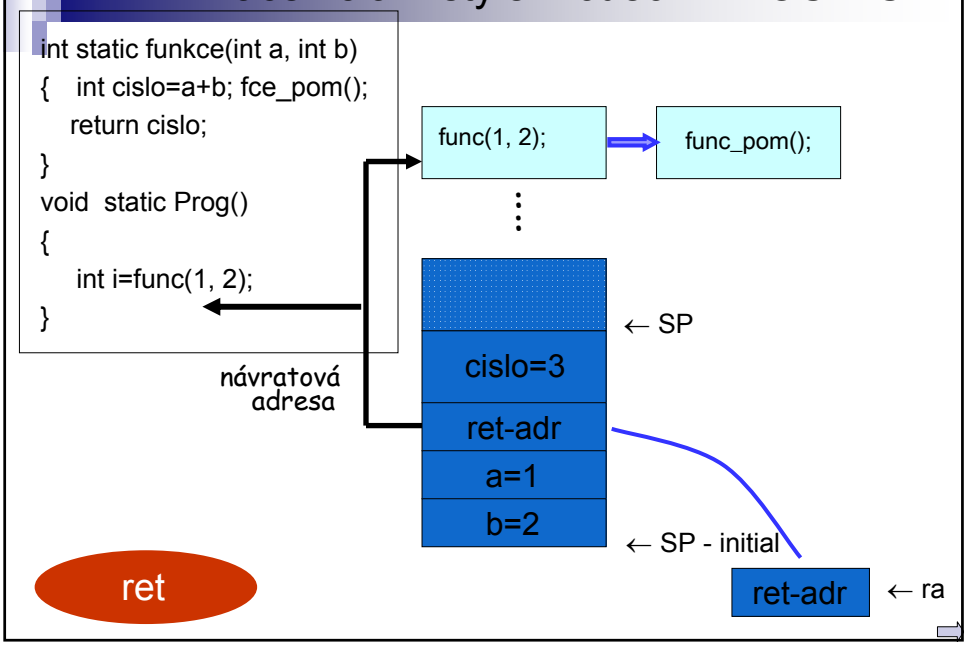


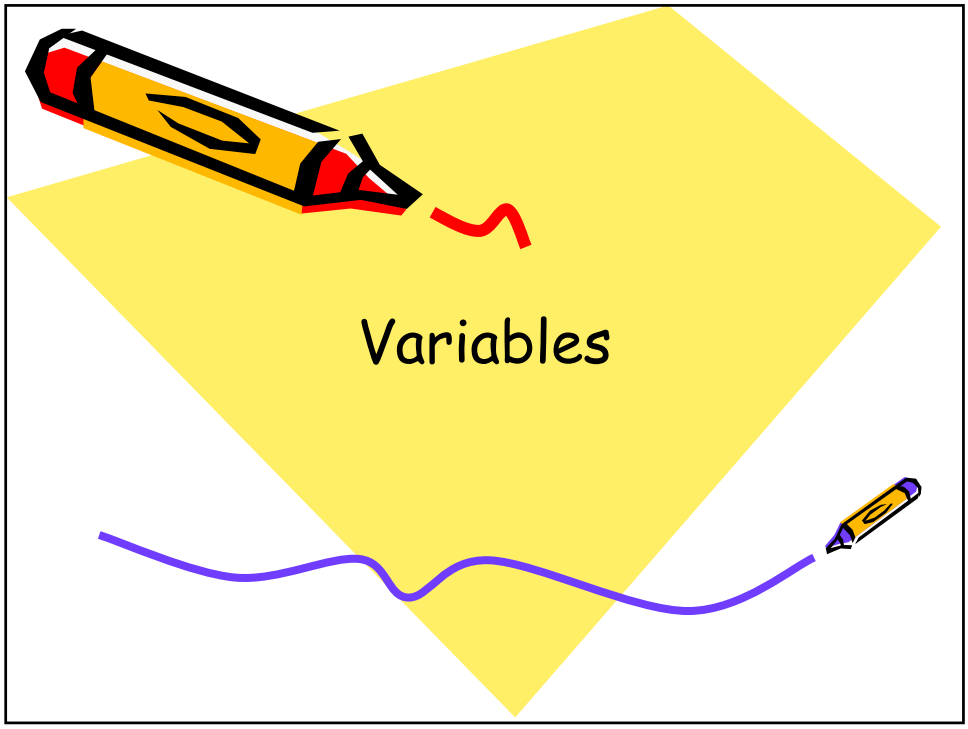
_cdecl in Pentium



_stdcall, _pascal, _fastcall
: RET 8,
_pascal UKAZKA
1→stack, 'c'→stack
_fastcall
1→EAX, 'c'→EDX

Animace volání stylem cdecl – NIOS + SP





Char [ ]

C-kod:

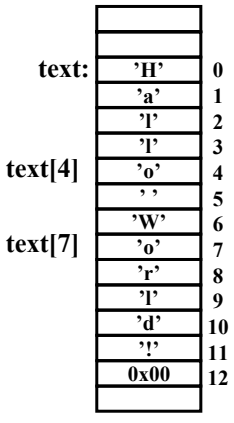
char text[] = ”Hello World!”;

/* text = &text[0] */

Assembler:

.data

text: .asciz ”Hello World!”



int [ ]

C-kod:

int vect[] = {1, 10, 0xA};

/* vect = &(vect[0]) */

Assembler:

.data

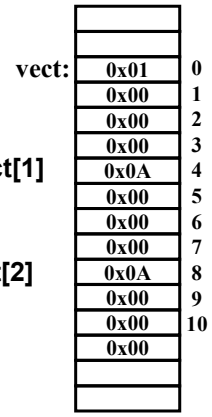
.align 2

vect: .word 1, 10, 0xA

#little endian byte ordering

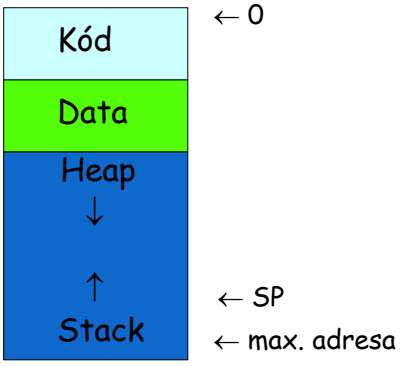
***(vect+1) ≡ vect[1]**

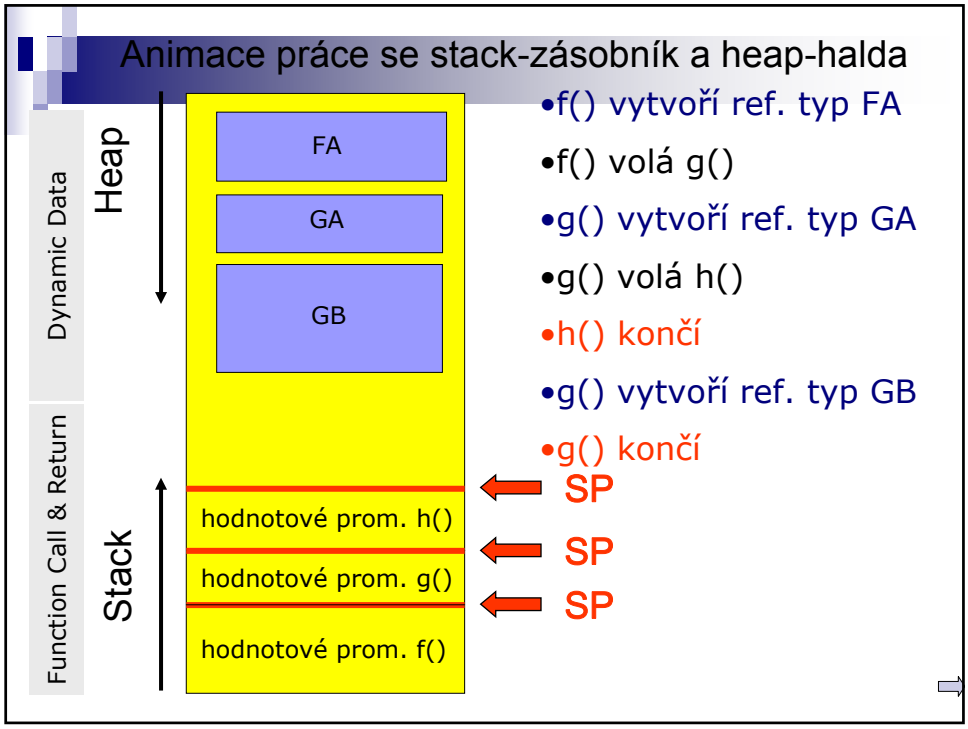
***(vect+2) ≡ vect[2]**



Organizace obecného programu

- **Kód** – kód programu
- **Data** – statické proměnné
- **Heap** – referenční typy
- **Stack**
  - Parametry funkcí
  - Návratové adresy
  - Hodnotové lokální proměnné




Animace práce se stack-zásobník a heap-halda
Dynamic Data
Heap
FA
GA
GB
Function Call & Return
Stack
hodnotové prom. h()
hodnotové prom. g()
hodnotové prom. f()
SP
SP
SP
•f() vytvoří ref. typ FA
•f() volá g()
•g() vytvoří ref. typ GA
•g() volá h()
•h() končí
•g() vytvoří ref. typ GB
•g() končí