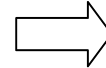
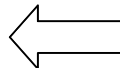
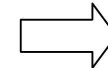
The Computer at Different Levels

High Level Languages 

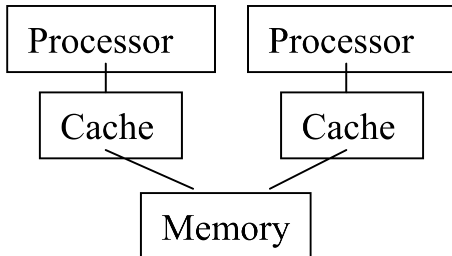
Java, C, Fortran,
Visual Basic

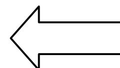
```
# Assembly code
  subu  $sp, $sp, 40
  sw    $ra, 32($sp)
  sw    $s0, 16($sp)
```

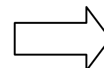
 Assembly Language

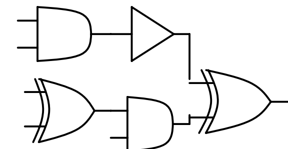
Instruction Set Architecture 

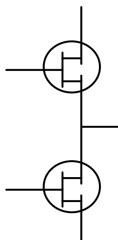
01110011
00111001
11001010
11101110

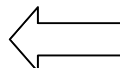


 Micro Architecture

Digital Logic 





 Transistors

Negative Numbers

Decimal	**Two's complement**
3	0000 0011
2	0000 0010
1	0000 0001
0	0000 0000
-1	1111 1111
-2	1111 1110
-3	1111 1101

Negate a Number

1. Complement all bits
2. Add one

   Done!

Example: Negate 23

23 = 0001 0111

Complement bits: => 1110 1000

Add one: => 1110 1001 = -23

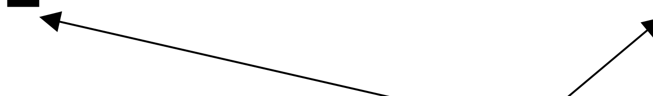
And back again:

Complement bits: => 0001 0110

Add one: => 0001 0111 = 23

Sign Bit & Sign Extension

1110 1001 = -23　　　　　　　0001 0111 = +23



Sign bit
0 => positive (or zero)
1 => negative

Sign Extension

8-bit: 0010 0101 (= 37)
32-bit: 0000 0000 0000 0000 0000 0000 0010 0101

8-bit: 1110 1001 (=-23)

32-bit: 1111 1111 1111 1111 1111 1111 1110 1001

Hexadecimal Numbers

Decimal numbers use base 10:
$3125 = 3*10^4 + 10^2 + 2*10^1 + 5$

Binary numbers use base 2:
Digits: 0, 1
$0101\ 1100 = 0*2^7+1*2^6+0*2^5+$
$+1*2^4+1*2^3+1*2^2+0*^2+0*1 = 92$ decimalt

Hexadecimal numbers use base 16:
Digits: 0, 1, 2, ..., 9, a, b, c, d, e, f
0x a3f8 $= 10*16^3 + 3*16^2 + 15*16 + 8 = 41976$ dec

hex - dec
a = 10
b = 11
c = 12
d = 13
e = 14
f = 15

Mark hex number

Characters and Strings - ASCII

- ASCII – American Standard Code for Information Interchange

- Modern Extension with åäö, Latin 1 or ISO 8859-15

Note:
lowercase = uppcase + 32

Character	Code (hex)	Code (decimal)
0	0x30	48
9	0x39	57
A	0x41	65
B	0x42	66
E	0x43	67
Z	0x5a	90
a	0x61	97
z	0x7a	122
newline	0x0a	10

Working with Bits

Bit-Wise Logical Operations

0110 0111 **and** 0100 1110 = 0100 0110

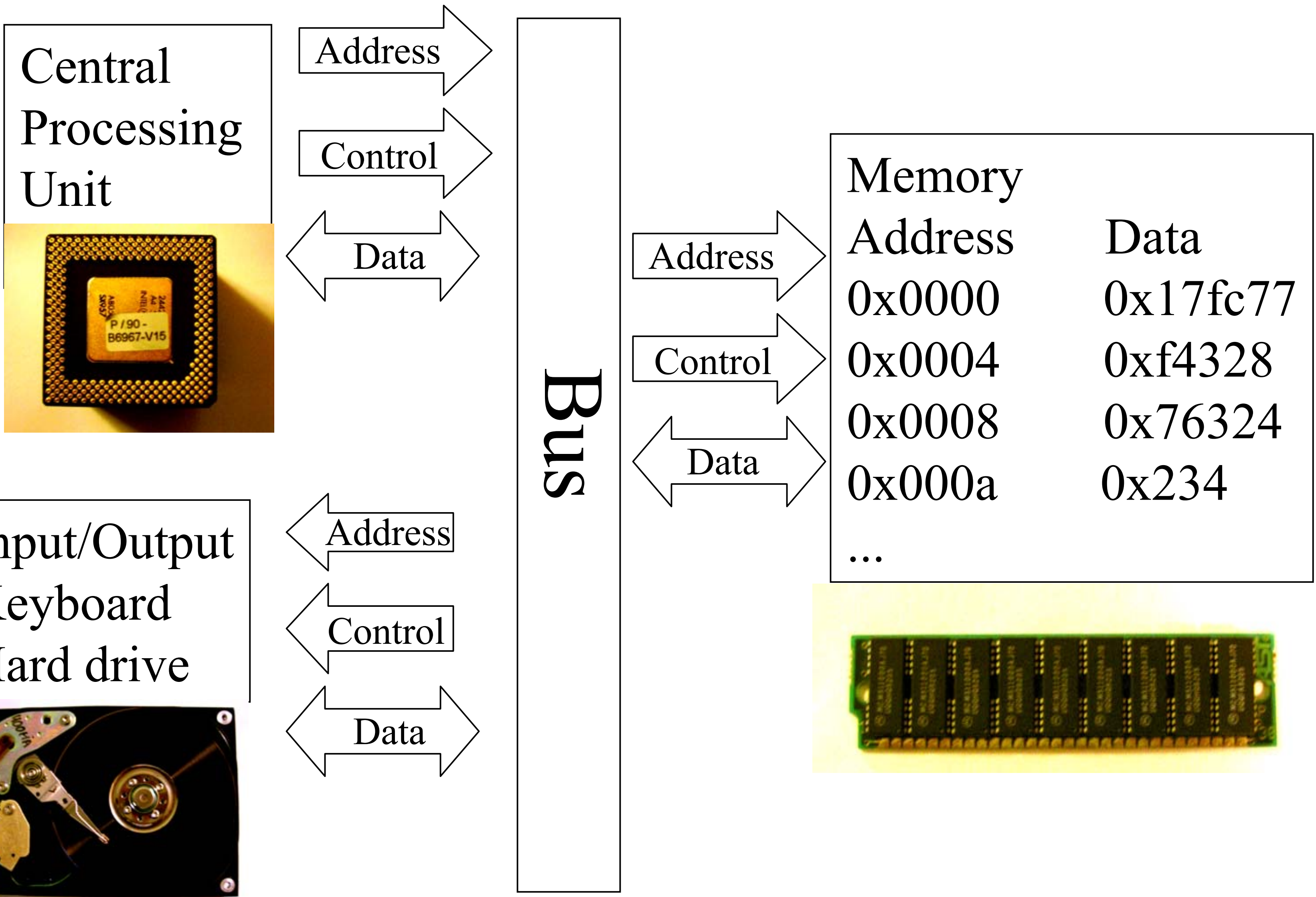
0110 0111 **or** 0100 1110 = 0110 1111

0110 0111 **xor** 0100 1110 = 0010 1001

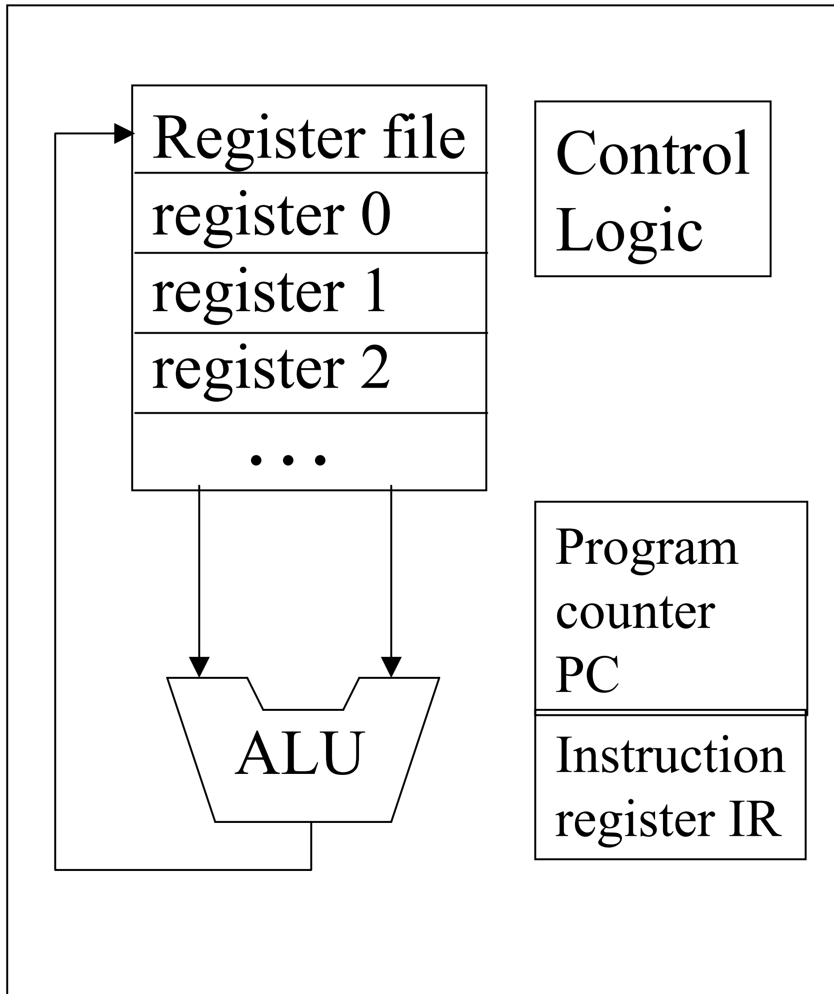
complement 0110 0111 = 1001 1000

Notation in C	
and	&
or	|
xor	^
compl-ement (not)	~

The von Neumann Architecture



The Processor

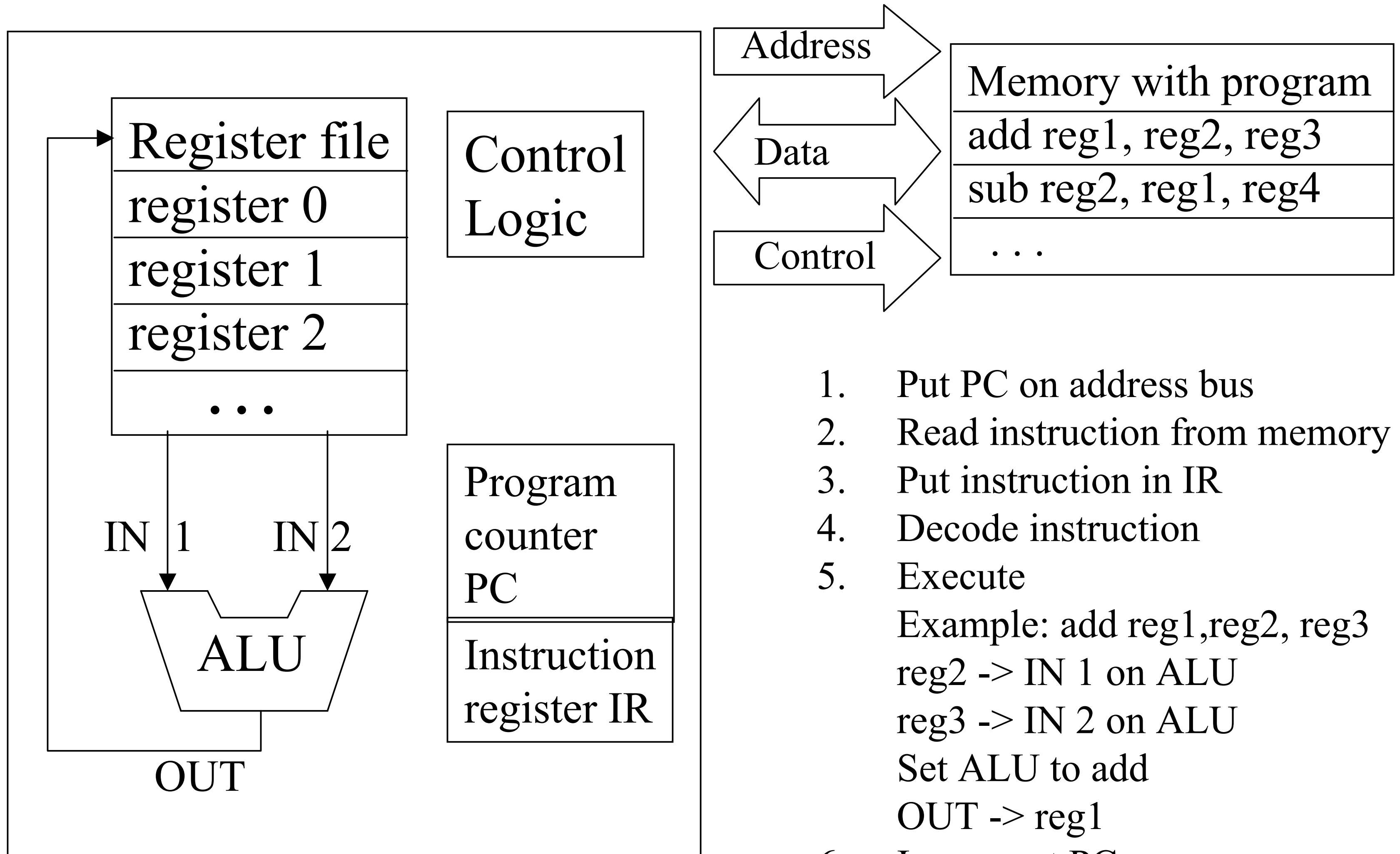


ALU = Arithmetic Logic Unit

Execute instructions:

add	**add** two numbers
sub	**sub**tract
ld	fetch data from memory,**l**oad **w**ord
st	store data in memory **s**tore **w**ord

The Fetch-Execute Cycle



1. Put PC on address bus
2. Read instruction from memory
3. Put instruction in IR
4. Decode instruction
5. Execute
   Example: add reg1,reg2, reg3
   reg2 -> IN 1 on ALU
   reg3 -> IN 2 on ALU
   Set ALU to add
   OUT -> reg1
6. Increment PC
7. Go to 1

CISC

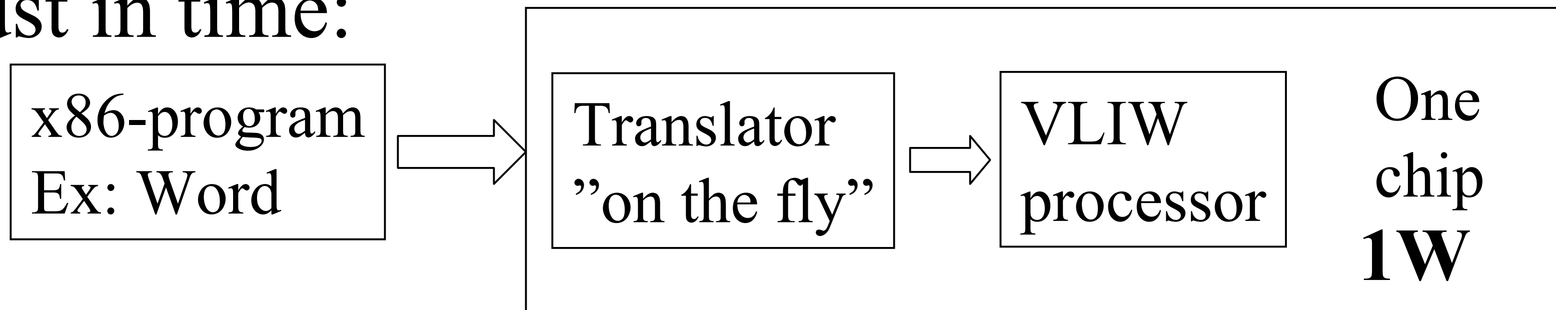
- Complex Instructions
- Variable-length instructions
- Long time to execute each instruction
- Short programs
- Examples:
  - x86 (AMD, Intel)
  - VAX (DEC)
  - 68000 (Motorola)
  - ??

RISC

- Simple instruction
- Single-size instructions
- Fast exection
- Longer programs
- Examples:
  - PowerPC (Motorola, IBM)
  - SPARC (Sun, Fujitsu)
  - Alpha (DEC)
  - …

Itanium and Transmeta ?

- VLIW (Very Long Instruction Word)
- Transmeta uses software to translate x86 just in time:



- Itanium execute large (43 bits long) instructions in parallel

The SPARC processor

- RISC processor (few, simple instructions)
- 32 bit word length
- 32 integer registers (visible at a time)
- 32 floating point registers
- Used by Sun & Fujitsu
- Similar to MIPS & PowerPC (Mac)

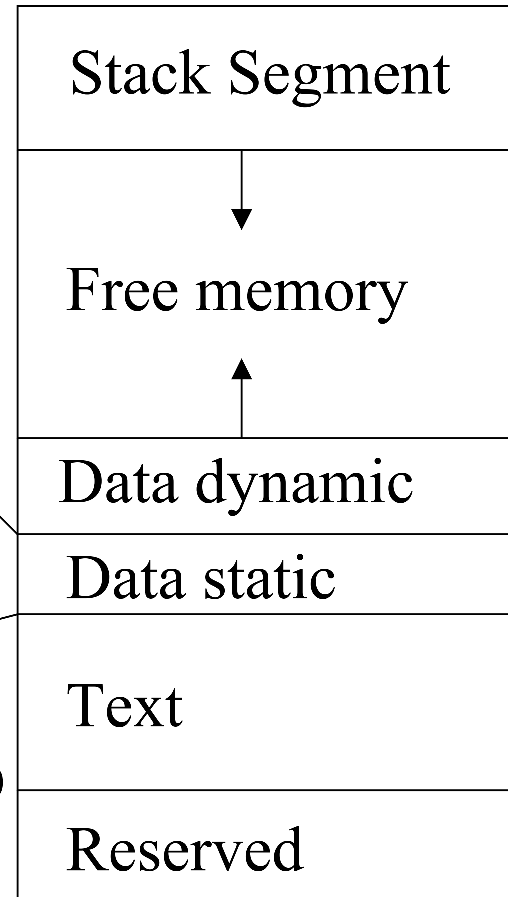
SPARC Registers

Register	Type	Usage
%g0-%g7 (%r0-%r7)	Global	Temporal variables. Global data and pointers.
%o0 - %o7 (%r8-%r15)	Output	6 first function parameters
%l0-%l7 (%r16-%r23)	Local	Procedure local variables
%i0-%i7 (%r24-%r31)	Input	6 first function parameters
%sp (%r14 = %o6)	Stack pointer	Points at the top of the stack
%fp (%r30 = %i6)	Frame pointer	Points at beginning of frame
PC	Program counter	Points at the currently executed instruction
nPC	next Program counter	Points at the next intruction to execute

The Memory

"A table of numbers"

Word	Byte 0	Byte 1	Byte 2	Byte 3
...				
0x0000 4010	...			
0x0000 400c	...			
0x0000 4008	0x47	0x00		
0x0000 4004	0x20	0x42	0x45	0x52
0x0000 4000	0x45	0x52	0x49	0x4b



Register Convention

$zero	always zero
$at	reserved for assembler
$v0, $v1	return values
$a0 - $a3	parameters
$t0-$t9	temporary (caller saved)
$s0-$s7	temporary (callee saved)
$sp, $fp	stack and frame pointer
$ra	return address

Syntax

- Labels (sv. etikett):

  {letter, ’.’} + {letter, number, ’_’, ’.’}* + :

  Example: **main: , n1: , M_17: , .L1345**
- Directives

  .word N ! Put value N in memory

  .skip M ! Reserve M bytes in memory

  .text ! Start text segment in program

  .data ! Start data segment in program

Syntax

- Instructions

```
add     %l1, 3, %l2   ! Add content of %l1 with
                      ! 3 and put result in %l2
set     45, %l3       ! Set register %l3 to 45
```

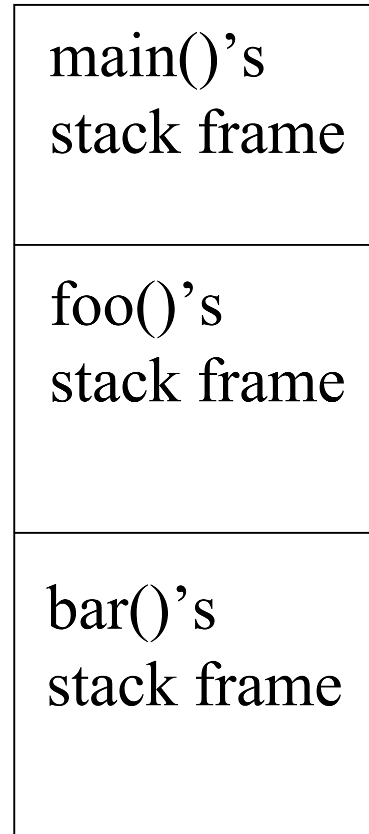
- Comments
  Any character after a '!' is treated as comment.
  (Like '//' in C++ or java)

```
! Add two numbers
        .data                   ! Put variables in data segment
        .align  4               ! Make shure data is word aligned
n1:     .word   17              ! First number
n2:     .word   23              ! Second number
result: .skip   4               ! Reserve space for result

        .text                   ! Put instructions in text segment
main:
        save    %sp, -96, %sp       ! Make space on stack
        set     n1, %l1  ! Pointer to first number
        set     n2, %l2  ! Pointer to second number
        ld      [%l1], %l3          ! Fetch value of n1
        ld      [%l2], %l4          ! Fetch value of n2
        add     %l3, %l4, %l3       ! Do the addition
        set     result, %l1         ! Get address of result
        st      %l3, [%l1]          ! Store result
        ret                         ! Return from function main
        restore
```

The Stack

- The stack is divided in **stack frames**
- Each function call creates a new stack frame
- **$sp** and **$fp** must allways be double-word aligned
- **Minimum** stack frame size is **24** bytes

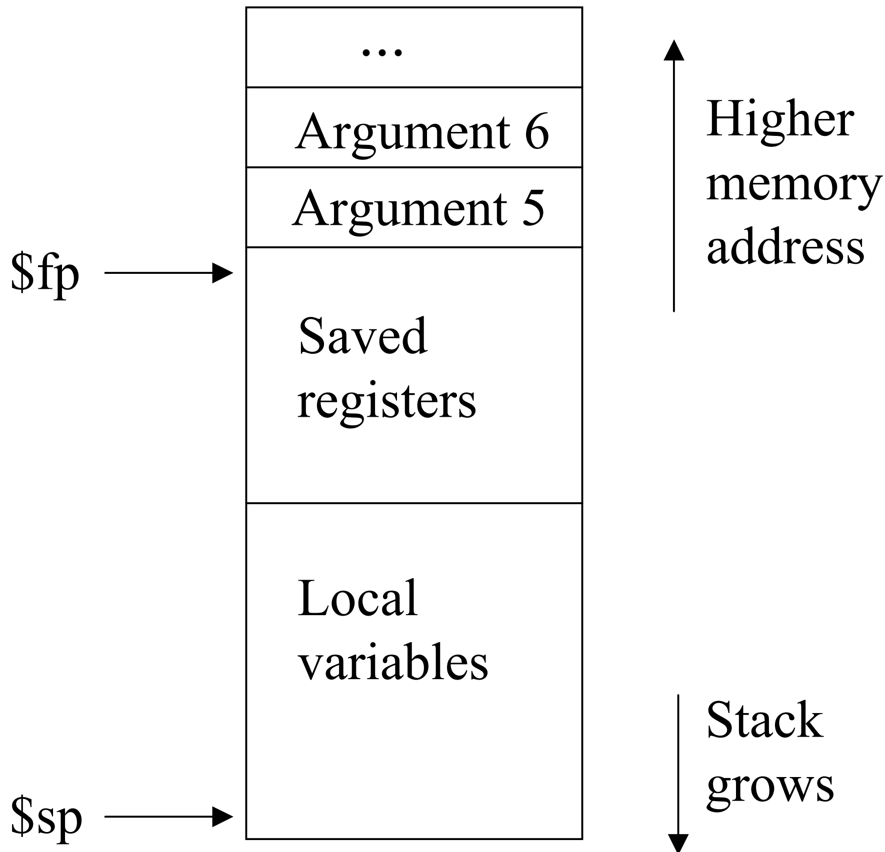



```
main () {
  foo();
}

foo() {
  bar();
}

bar() {
  puts(”Hello”);
}
```

The Stack Frame



Important!
$sp and $fp
must always be
double-word aligned.