Chapter 2

Instruction Set and Machine Language

Topics:

- Types of instructions
- Addressing modes
- Assembler programming
- Classifying instruction set architectures

Trade offs

The instruction set defines the basic operations a processor is able to execute. All computations have to be mapped onto a sequence of basic instructions. It should satisfy the needs of the programmer and the compiler. (external view)

$\Rightarrow$ less instructions lead to larger machine language programs
$\Rightarrow$ more instructions require more sophisticated compilers

The architecture and the hardware implementation of a processor depend on the instruction set (internal view)

$\Rightarrow$ more instructions will lead to more complex hardware
$\Rightarrow$ less instructions lead to more simple and more regular hardware structures

Application

Desktop computing:

Important: Fast integer ***and*** floating point operations

Less important: Power consumption, memory requirement (code size)

Server: (data bases, web server etc.)

Important: Fast operations for processing integer numbers and character strings

Less important: Floating point arithmetic performance, memory requirement (code size)

Application

Embedded processor cores:

Important: power consumption (battery powered devices)

code size (affects number of memory modules being required)

cost (high volume consumer products)

real time applications: worst case performance more important than average performance

Anyway: most existing instruction sets are similar

RISC: reduced instruction set computer

(special purpose applications may require more complex instructions)

Arithmetic instructions

Integer operations for bytes, half words and words

Instruction set example: MIPS R2000/3000 (1982-89)

- typical for RISC processors

Example: Addition and subtraction in MIPS assembler language

```
add a, b, c
```
„add b and c and store the result in a“

```
sub a, b, c
```
„subtract c from b and store the result in a“

Operands are stored in registers: MIPS has 32 32-bit register.

Notation

Convention for naming registers in assembler commands:

$-sign followed by name of register, e.g. $s0, $s1, $s2..., $t0, $t1, $t2...

```
add $s1, $s2, $s3 # add s2 and s3 and store result in s1
```

comment

Further convention:

Registers corresponding to variables of a high level language program are denoted with s0, s1... . Registers allocated by the compiler for auxiliary variables are denoted with t0, t1... (temporary registers)

MIPS assembler maps:

register name	register number
$t0 - $t7	8 - 15
$t8 - $t9	24 - 25
$s0 - $s7	16 - 23

Arithmetic instructions

MIPS instruction format: register-register format (R - type)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op:	opcode, denotes the instruction to be executed
rs:	source register, first operand
rt:	source register, second operand
rd:	destination register where the result will be stored
shamt:	shift amount, used for shifting operations
funct:	function code, may specify a certain subtype of the operation defined by opcode (e.g., *add* and *sub* have the same opcode but different function codes)

Example

Consider the following arithmetic expression in C:

```
y = (a + b) – (c + d);
```

What assembler code will be generated by a compiler?

The variables y, a, b, c, d are stored in registers s0, s1, s2, s3, s4:

```
add $t0, $s1, $s2 #  register t0 now contains a + b
add $t1, $s3, $s4 #  register t1 now contains c + d
sub $s0, $t0, $t1 #  register s0 now contains result y
```

Arithmetic instructions

MIPS instruction format: immediate format (I – type)

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

In a typical program many operations use a constant value being known at compile time as second source operand.

Example: incrementing a counter variable inside a loop

MIPS instruction: „*add immediate*“, constant value is part of the instruction

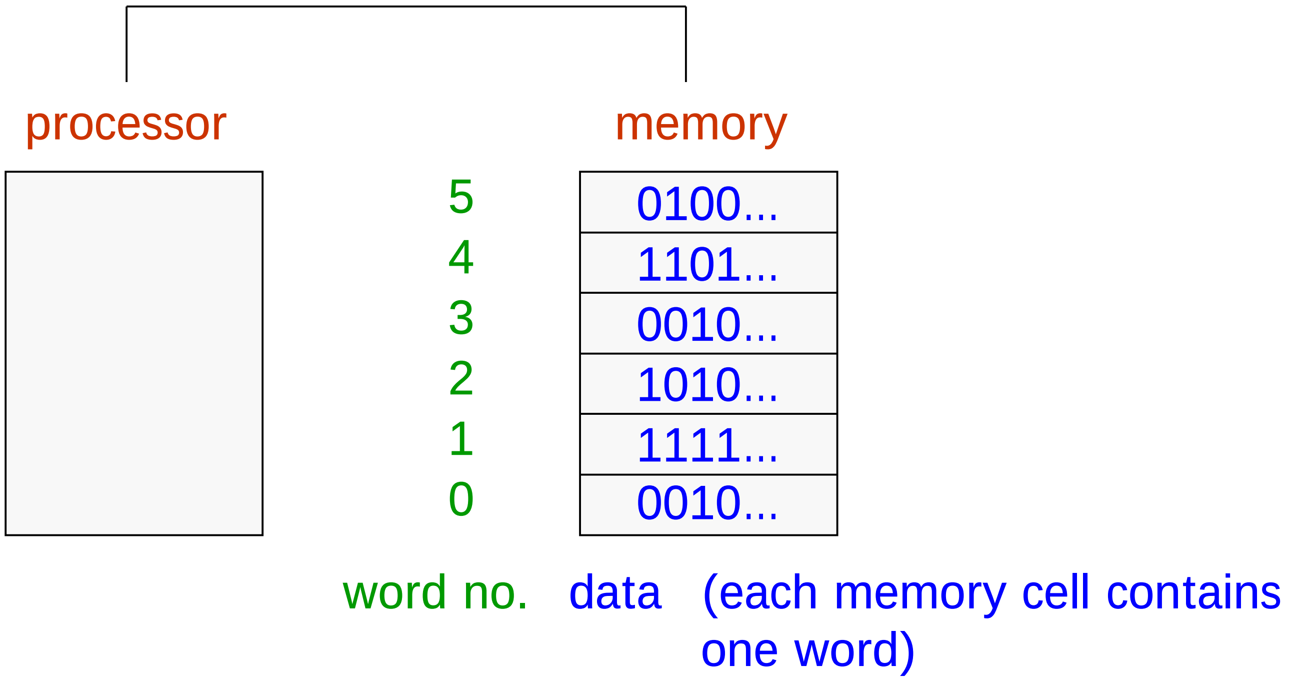
Target register defined by rt

```
addi $s1, $s1, 4    # $s1 = $s1 + 4
```

Data transfer instructions

Many programs use more variables than registers are available.

Variables are stored in main memory and copied into registers when their values are needed.

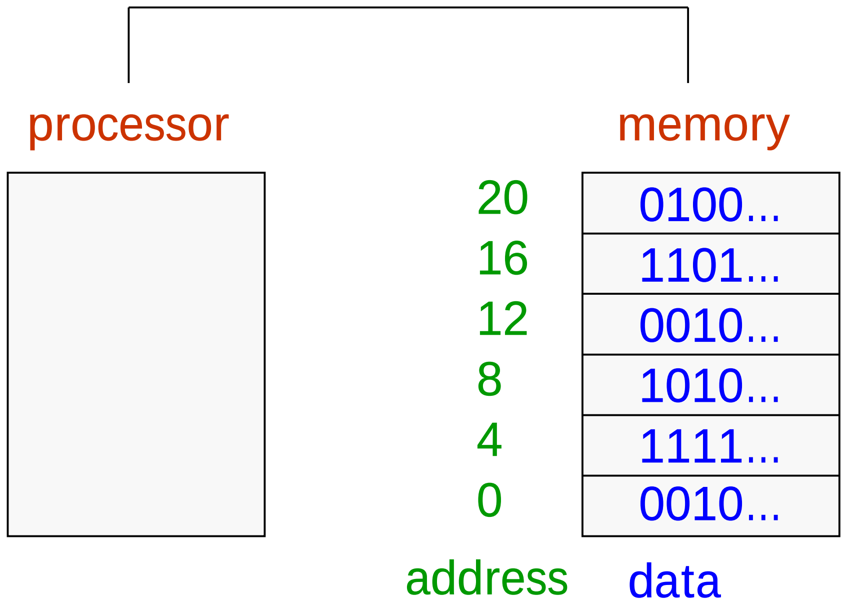


The process of moving the content of registers being less frequently used to memory is called *register spilling*

Data transfer instructions

Many programs use single bytes for storing information (e.g., characters). Using 32 bit words would be wasteful in those cases.

For most computers the smallest addressable memory block is 1 byte.



32 bit words always start at addresses being a multiple of 4:

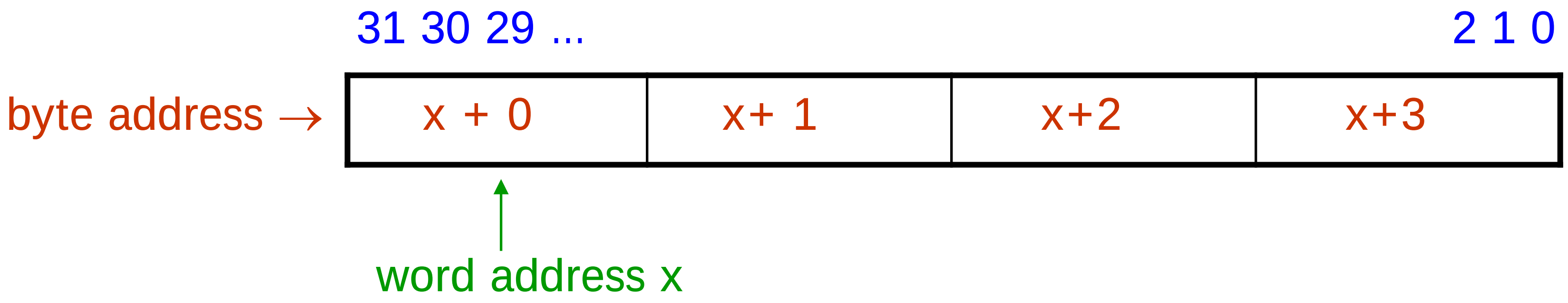
„alignment restriction“

Using 32 address bits we can address 4 GByte.

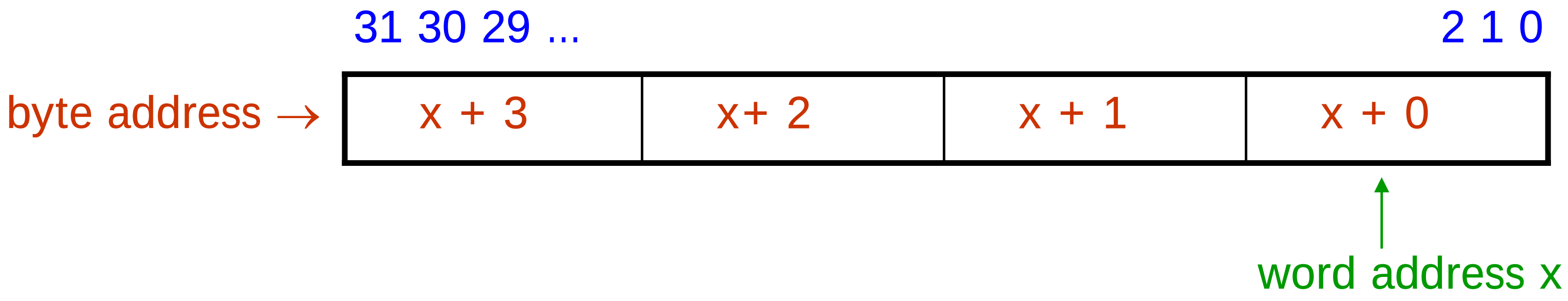
Data transfer instructions

2 possibilities for *alignment restriction* :

- The address of a word is given by the address of the most significant byte of the word: **big endian** architecture, e.g. MIPS



- The address of a word is given by the address of the least significant byte of the word: **little endian** architecture



Data transfer instructions

Copy some data from source to destination

- Source/target may be located in main memory as well as in register memory
- also used for moving data between IO – interfaces and register memory

Example: MIPS - instructions lw (load word) and sw (store word)

```
lw $s0, ($s1)  # take the content of s1 as an memory address
               # and load the data being stored there into s0
sw $s0, ($s1)  # copy the content of register s0 into the
               # (memory) cell with the address in s1
```

Data transfer instructions

Translate the following C-expression:

```
      A[8] = A[8] + h;
```

Assumption: register s2 contains variable h and register s3 contains start address of an array A of integers (= address of the first item).

```
addi $t0, $s3, 32    # calculate address of A[8]
lw   $t1, ($t0)      # load A[8] into t1
add  $t1, $t1, $s2   # t1 now contains h + A[8]
sw   $t1, ($t0)      # store result in A[8]
```

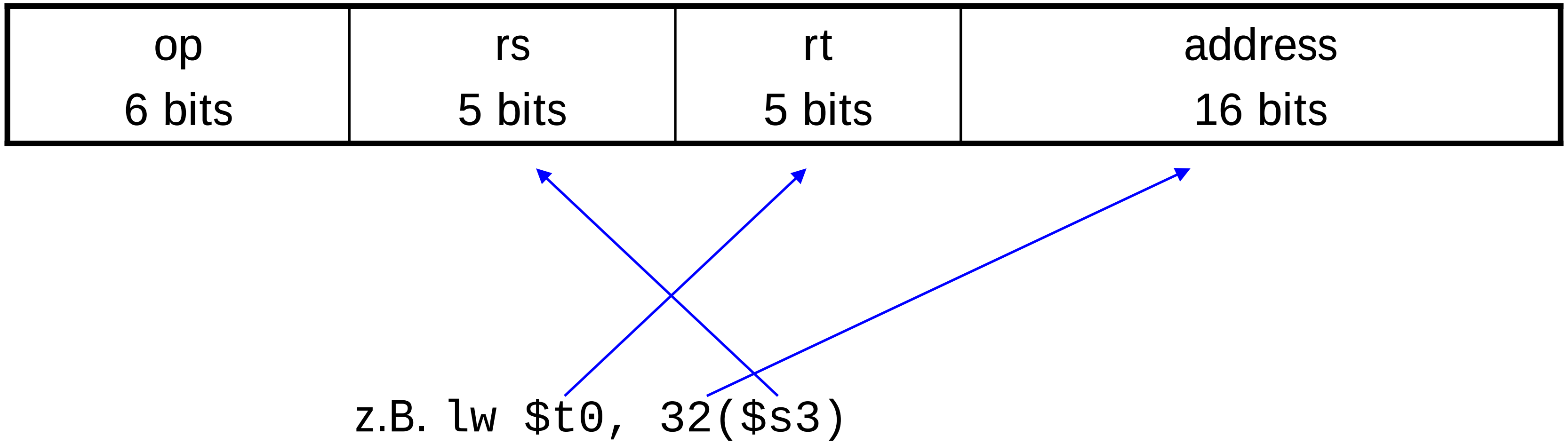
Addresses being calculated from a base and an index occur quite frequently!

That’s why load/store instructions allow to specify <u>an additional offset</u>:

```
lw   $t0, 32($s3)    # t0 contains A[8]
add  $t0, $s2, $t0   # t0 now contains A[8] + h
sw   $t0, 32($s3)    # store result in A[8]
```

Data transfer instructions

MIPS instructions format: load/store



op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

z.B. `lw $t0, 32($s3)`

Note: load/store uses immediate instruction format

⇒ simplifies hardware

Branching instructions

For writing a typical program we need further instructions for:

- performing conditional branches and unconditional jumps
- calling a subprogram
- returning from a subprogram

Branches:	evaluate a certain condition and jump to a certain target only if the condition is fulfilled;
Jumps:	jumps always to a certain target

Remember: *program counter* (PC) contains the address of the instruction to be executed next

Branching instructions

Examples for branching MIPS - instructions:

```
beq $s1, $s2, L1
```

branch if equal: go to the instruction labeled L1 if the value in s1 equals the value in s2 (conditional branch).

```
bne $s1, $s2, L1
```

branch if not equal: (conditional branch)

```
j L1
```

jump: go to the instruction labeled L1 (unconditional jump)

```
jr $s1
```

jump register: go to the instruction with the address in s1 (unconditional jump)

Labels are used to mark instructions being a target of a branch/jump. When the assembler translates a program into machine language it replaces all labels by the corresponding address.

⇒ relieves the programmer from calculating the target addresses

Branching instructions

Example:

Translate the following C-code into MIPS assembler language:

```
if (i == j)
  a = b + c;
else
  a = b – c;
```

Assume that a, b, c, i, j correspond to registers $s0, $s1, $s2, $s3, $s4:

```
      bne $s3, $s4, Else    # goto Else if i ≠ j
      add $s0, $s1, $s2     # a = b + c
      j Exit                # goto Exit
Else: sub $s0, $s1, $s2     # a = b – c
Exit:
```

Loops (do ... while, while, for) are translated following the same principle

Branching instructions

Instruction format:

of an unconditional branch: e.g. `j`

op	address
6 bits	26 bits

of a conditional branch: e.g. `beq, bne`

op	rs	rt	(branch) address
6 bits	5 bits	5 bits	16 bits

Can we address targets > 2^{16} ?

Most programs consist of more than 2^{16} words. However, typically the target of a conditional branch is located in the neighborhood of the branching instruction.

$\Rightarrow$ PC - relative addressing is used for the target: PC = PC ± branch address

Branching instructions

Procedures:

For calling a *procedure* (= *subprogram*) the following steps have to be executed:

1) Place arguments in a place where the procedure can access them.

2) Jump to the start address of the procedure

3) Allocate registers for the procedure

4) Perform the desired task

5) Place the results in a place where the calling procedure can access them

6) Return to the origin of the procedure call

Calling procedure = **Caller**   Called procedure = **Callee**

Branching instructions

Procedures (MIPS)

The MIPS programming convention reserves the following registers for procedure calls:

`$a0 - $a3`: four argument registers in which to pass parameters
`$v0 - $v1`: two value registers in which to return a result

To be able to return to the point of origin from a procedure the return address has to be saved:

`$ra`: one register for storing the point of origin of a procedure call

MIPS instruction for jumping to procedures: jump-and-link

```
jal procedure_address
```

stores address of the instruction to be executed immediately after returning from a procedure call (the instruction following the `jal`-instruction) in `$ra` („link“) and jumps to target address („jump“)

Branching instructions

Example:

Translate the following C-function into in MIPS assembler code:

```
int leaf_example(int g, int h, int i, int j)
{
  int f;

  f = (g + h) – (i + j)
  return f;
}
```

Assume that g, h, i, j correspond to the registers $a0, $a1, $a2, $a3 and f corresponds to register $s0.

⇒ The procedure uses $s0 and two temporary registers $t0 und $t1 (see 2-8)

But: Registers $t0, $t1, $s0 may already be occupied by the calling function!

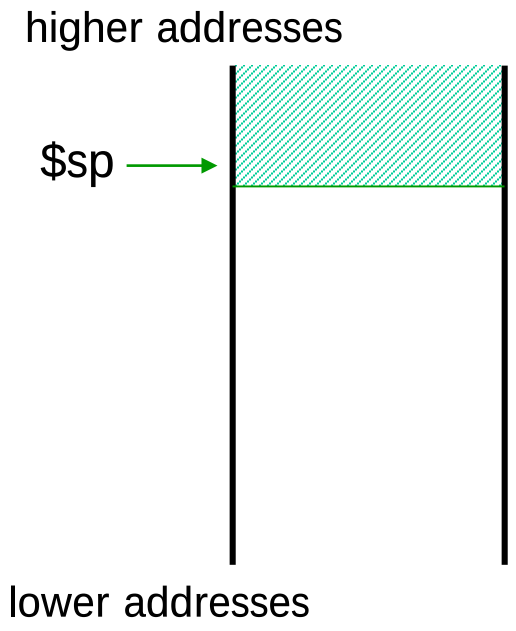
⇒ Save their content before overwriting them!

Branching instructions

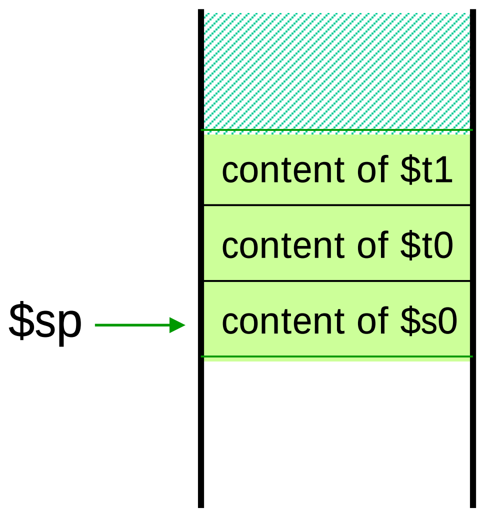
For saving register contents temporarily a ***stack*** is used. A *stack* is a memory following the last-in-first-out (LIFO) principle.

The address of the item inserted at last (= top of stack) is stored in a dedicated register `$sp` called ***stack pointer***

For historical reasons the *stack* grows to lower addresses ⇒ inserting data decrements the stack pointer.



Initial situation



Stack after saving $t1, $t0, $s0

Branching instructions

Stack operations:

push: decrement stack pointer `$sp` and insert one word at the memory location `$sp` now points to (= new top of stack)

pop: remove word being stored at the top of stack and increment `$sp`

MIPS assembler code for procedure leaf_example:

```
leaf_example:
      addi $sp, $sp, -12  # allocate space for 3 words
      sw $t1, 8($sp)      # save $t1 on stack (push)
      sw $t0, 4($sp)      # save $t0 on stack (push)
      sw $s0, 0($sp)      # save $s0 on stack (push)
      add $t0, $a0, $a1   # register t0 gets g + h
      add $t1, $a2, $a3   # register t1 gets i + j
      sub $s0, $t0, $t1   # register s0 gets f
      addi $v0, $s0, 0    # copy f into result register v0
      lw $s0, 0($sp)      # restore $s0 for caller
      lw $t0, 4($sp)      # restore $t0 for caller
      lw $t1, 8($sp)      # restore $t1 for caller
      addi $sp, $sp, 12   # clean up stack (pop)
      jr $ra              # jump back to caller
```

Branching Instructions

Observation: the content of a temporary registers is only required for a very short time period and often not needed anymore after a procedure call ⇒ unnecessary to save them

MIPS convention:

\$t0 - \$t9:	are NOT saved by called procedure
\$s0 - \$s7:	caller expects them to be unchanged ⇒ have to be saved by called procedure if modified

⇒ reduces register spilling

simplified MIPS assembler code for procedure leaf_example():

```
leaf_example:
     addi $sp, $sp, -4    # allocate room for 1 word
     sw $s0, 0($sp)       # save $s0 on stack
     add $t0, $a0, $a1    # register t0 gets g + h
     add $t1, $a2, $a3    # register t1 gets i + j
     sub $s0, $t0, $t1    # register s0 gets f
     addi $v0, $s0, 0     # result register gets f
     lw $s0, 0($sp)       # restore $s0
     addi $sp, $sp, 4     # clean up stack
     jr $ra               # jump back to caller
```

Branching instructions

Nested procedures

Successive calls to nested procedures will overwrite arguments $a0 - $a3 as well as the return address $ra (still needed!)

Strategy:

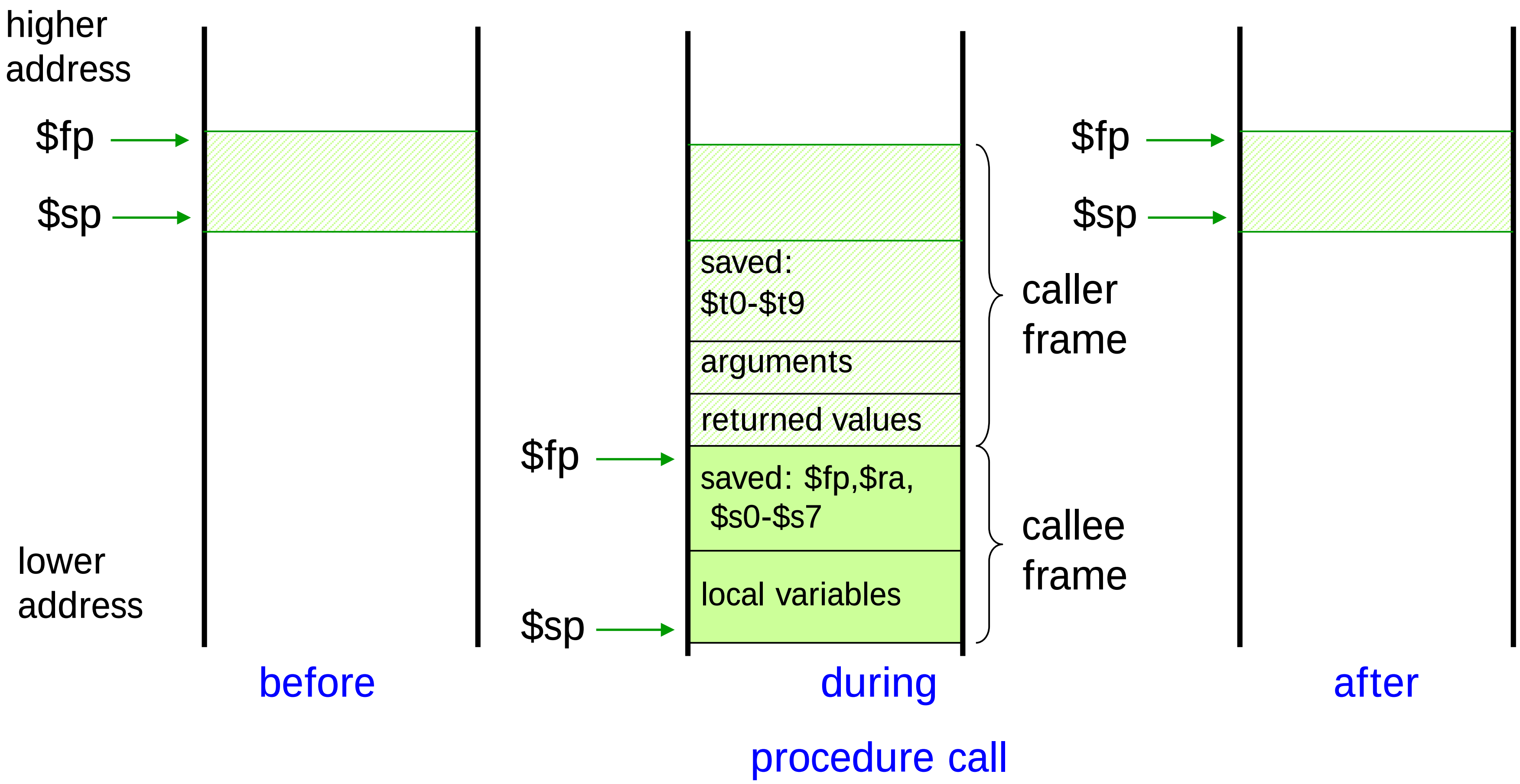
- Callee saves those registers on the stack which it will modify and which the caller expects to be unchanged, e.g. $ra.
- Caller saves all those registers that the callee is allowed to modify but whose values are still needed after the call
- The callee guarantees to restore the stack

Caller expects to be unchanged	**Callee is allowed to modify (without restoring)**
register $s0 - $s7	register $t0 - t9
stack pointer $sp	argument registers $a0 - $a3
return address $ra	result registers $v0-$v1
stack (range above stack pointer)	range below stack pointer

Branching instructions

Frame pointer: points to the first stack entry belonging to the procedure being actually executed

Frame pointer does not change during the execution of a procedure ⇒ well suited for being used as base address for address calculations



Further categories of instructions

Shift and rotation instructions:

- logic and arithmetic shifts,
- rotate (insert the bits that are shifted out on one one side at the other side)

String instructions: operations working on bytes and byte arrays (copy, compare, etc.)

Instructions for controlling the processor:

- reset (initialize components)
- stop (stop program)
- trap (interrupts)
- NOP (no operation)

Properties of MIPS instructions

R-type arithmetic instructions:

op 6 bits	rs 5 bits	rt 5 bits	rd 5 bits	shamt 5 bits	funct 6 bits

I-type arithmetic instructions, conditional branches, load/store

op 6 bits	rs 5 bits	rt 5 bits	immediate 16 bits

Jump instructions

op 6 bits	address 26 bits

Properties of instructions:

- All: have the same length (32 bit)
- I-type and R-type: rs denotes first source register
- R-type: rt denotes second source register
- I-type: Immediate field contains second operand

⇒ Simplifies hardware implementation

Alignment Restriction

Observation:

- All instructions are 32 bit wide
- Many operands (addresses, standard integers, single precision floating point numbers) are 32 bit wide, too

⇒ Make CPU-memory interface at least 32 bit wide

Simplify the design of the memory interface by introducing an

Alignment Restriction:

- Use byte addressing in order to enable efficient storage of bytes, but
- 32-bit words must start at byte addresses being a multiple of 4

Addressing modes

Address calculation

For calculating the actual address of an object (*= effective address*) various methods are used by various instruction set architectures.

Used for calculating an effective address:
• constant value
• register content
• content of a memory

In the following: some examples of addressing modes being often used.

Addressing modes

Register addressing (0 stage addressing)

Operand is the content of a register: instruction contains number of register: „explicit“ register addressing

Further possibility: instruction always accesses one particular register implicitly: „implicit“ register addressing, e.g. `jal label`

No memory accesses necessary ⇒ *0 stage addressing*

Immediate addressing (0 stage)

Instruction contains operand value

MIPS instruction: „*set on less than immediate*“,

```
slti $t0, $s2, 10 # $t0 = 1 if $s2 < 10 else $t0 = 0
```

MIPS instruction: „*add immediate*“

```
addi $sp, $sp, 4  # $sp = $sp + 4
```

Addressing modes

One-stage addressing: One single memory access necessary for accessing data

1. possibility: instruction contains address: “direct“ addressing (e.g. `j`)

2. possibility: instruction contains no. of the register which contains the address: ”indirect“ addressing (e.g. `jr`)

Index addressing:

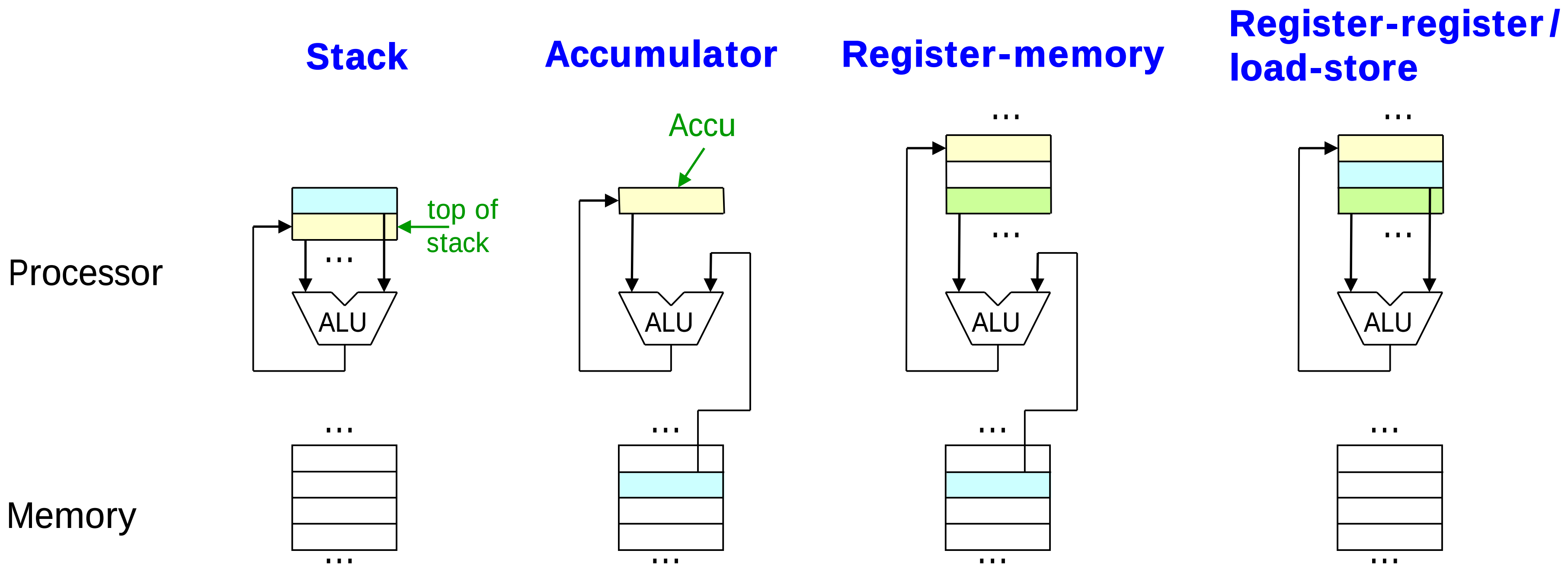
The effective address is calculated from a base (being stored in a register or in memory) and an offset being contained in the instruction or in a register (e.g. `lw`, `sw`)

PC-relative addressing:

Index addressing: instruction contains the offset and the program counter contains the base (e.g. `beq`)

Categories of instruction sets architectures

4 categories of instruction set architectures



Instruction set architectures (*ISA*) differ in where the operands are located immediately before an instruction is executed.

Example: HP Pocket Calculator

Categories of instruction sets

C-expression C = A + B translated for 4 ISAs:

Stack	**Accumulator**	**Register (Register-memory)**	**Register (load-store)**
push A	load A	load $r1, A	load $r1, A
push B	add B	add $r3, $r1, B	load $r2, B
add	store C	store $r3, C	add $r3,$r1, $r2
pop C			store $r3, C

Stack und accumulator machine:

+ simple hardware
- accessing operands is more complicated than for machines using general purpose registers
- many memory accesses

⇒ most modern computers use general purpose register

Categories of instruction sets

Register-memory:

+ Data can be accessed without using an additional *load* instruction
- Number of clock cycles used per instruction may vary depending on addressing mode
- Varying instruction sizes (depending on addressing modes)

Register-register:

+ Simple (only a few instruction formats)
+ Many instructions require similar number of clock cycles for execution
+ 3 register addresses (source, source, target) can be easily integrated into one 32 bit instruction

- More instructions (because of load/stores) $\Rightarrow$ larger programs

Categories of instruction sets

Memory-memory:

+ simple hardware, no registers necessary

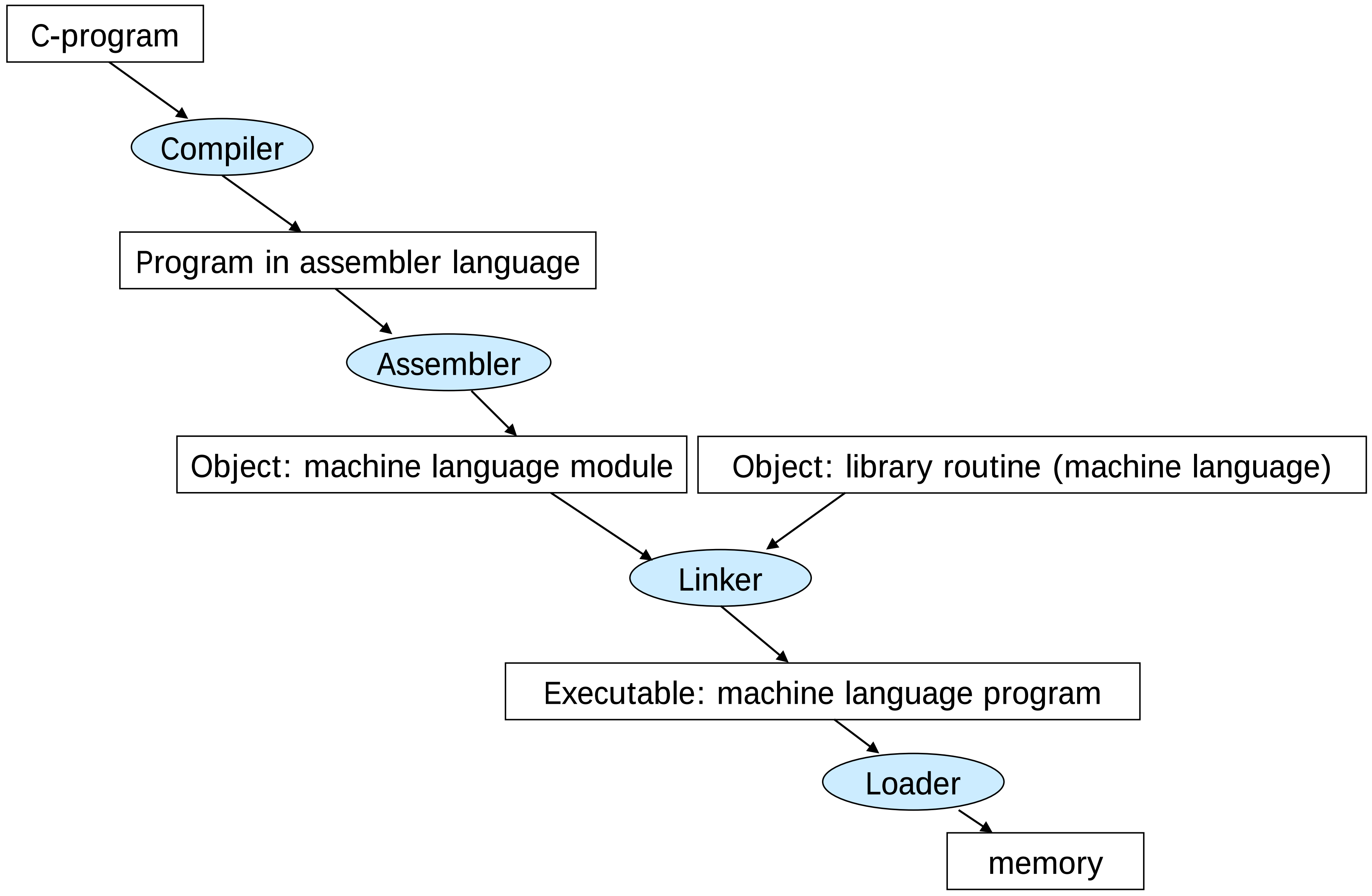
- memory accesses limit speed

$\Rightarrow$ not used today! (in earlier days used by VAX-computers)

Today, mainly register-memory and register-register architectures are used:

number of operands in memory	**max. number of operands**	**architecture**	**examples**
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000

Starting a program


C-program
Compiler
Program in assembler language
Assembler
Object: machine language module
Object: library routine (machine language)
Linker
Executable: machine language program
Loader
memory

Starting a program

Compiler

- translates high level language program into assembler language

Assembler

- translates assembler language program (symbolic) into *object files* written in machine language (binary)
- assigns addresses to labels; pairs of addresses and labels are stored in a *symbol table*
- accepts numbers given in binary, decimal or hexadecimal representation

Starting a program

Assembler

- translates **pseudo assembler instructions** (being provided by the assembler for convenience) into real assembler instructions, e.g.:

1. `move $t0,$t1 # copy content of register t1 into t0`

   is translated into:

   ```
   addi $t0,$t1,0
   ```

2. `blt $s1,$s2,L # branch on less than: if $s1<$s2 go to L`

   is translated into:

   ```
   slt $at,$s1,$s2
   bne $zero,$at,L
   ```

   Register `$at` is reserved for assembler

   Register `$zero` is a read-only register containing always 0.

Starting a program

Linker

Many parts of a program rarely change $\rightarrow$ are stored in libraries. A linker

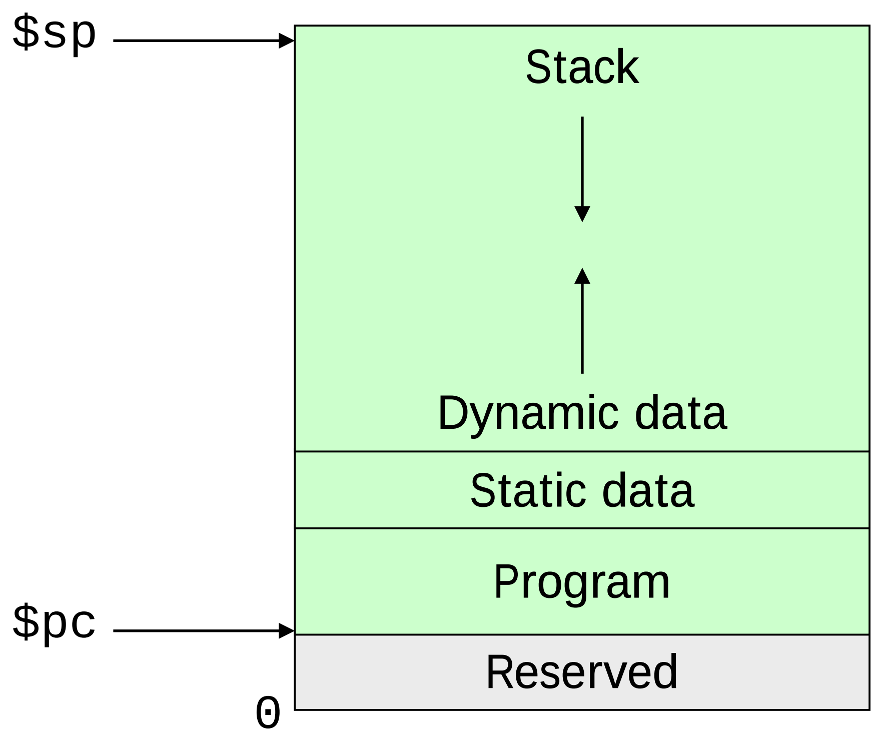
- binds together library files and machine language level program files (object files)
- Re-calculates addresses of objects and instruction labels
- Puts an executable program into an external memory (hard disk)

Loader

- allocates space in memory and copies executable into memory
- copies all input parameters onto stack
- initializes (clears) registers

Starting a program

MIPS memory allocation for program and data


$sp
Stack
Dynamic data
Static data
Program
$pc
Reserved
0