Languages and Compilation

Based on the Jean-Christophe Filliâtre's Courses
given at École Polytechnique & École Normale Supérieure

Lecture 7 - Assembly

Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

lecture:

- small reminder about computer architecture
- overview of MIPS architecture (with demos)
- **functions calls & the stack** (slides 36-45)

  https://homes.cs.aau.dk/~lego/compil25/lab_sessions/7/demo/

lab session:

- writing manually some small MIPS programs
- implementing a compiler for a mini-language of arithmetic expressions generating automatically MIPS code

  https://homes.cs.aau.dk/~lego/compil25/lab_sessions/7/index.html

lecture:

- small reminder about computer architecture
- overview of MIPS architecture (with demos)
- **functions calls & the stack** (slides 36-45)

https://homes.cs.aau.dk/~lego/compil25/lab_sessions/7/demo/

lab session:

- writing manually some small MIPS programs
- implementing a compiler for a mini-language of arithmetic expressions generating automatically MIPS code

https://homes.cs.aau.dk/~lego/compil25/lab_sessions/7/index.html

an integer is represented using n bits,
written from right (least significant) to left (most significant)

b_{n-1}	b_{n-2}	$\dots$	b_1	b_0

typically, n is 8, 16, 32, or 64

$$\begin{aligned} \text{bits} &= b_{n-1} b_{n-2} \dots b_1 b_0 \\ \text{value} &= \sum_{i=0}^{n-1} b_i 2^i \end{aligned}$$

bits	value
000...000	0
000...001	1
000...010	2
⋮	⋮
111...110	$2^n - 2$
111...111	$2^n - 1$

example: $00101010_2 = 42$

the most significant bit b_{n-1} is the **sign bit**

$$\begin{aligned} \text{bits} &= b_{n-1} b_{n-2} \dots b_1 b_0 \\ \text{value} &= -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \end{aligned}$$

bits	value
100...000	-2^{n-1}
100...001	$-2^{n-1}+1$
⋮	⋮
111...110	-2
111...111	-1
000...000	0
000...001	1
000...010	2
⋮	⋮
011...110	$2^{n-1}-2$
011...111	$2^{n-1}-1$

example:

$$\begin{aligned} 11010110_2 &= -128 + 86 \\ &= -42 \end{aligned}$$

according to the context, the same bits are interpreted either as a signed or unsigned integer

example:

- $11010110_2 = -42$ (signed 8-bit integer)
- $11010110_2 = 214$ (unsigned 8-bit integer)

the machine provide operations such as

- logical (aka bitwise) operations: and, or, xor, not
- shift operations
- arithmetic operations: addition, subtraction, multiplication, etc.

operation		example
negation	`x`	`00101001`
	`not x`	`11010110`
and	`x`	`00101001`
	`y`	`01101100`
	`x and y`	`00101000`
or	`x`	`00101001`
	`y`	`01101100`
	`x or y`	`01101101`
xor	`x`	`00101001`
	`y`	`01101100`
	`x xor y`	`01000101`

- logical shift left (inserts least significant zeros)

$\leftarrow$	b_{n-3}	$\ldots$	b_1	b_0	**0**	**0**	$\leftarrow$

(<< in Java, `lsl` in OCaml)

- logical shift right (inserts most significant zeros)

$\rightarrow$	**0**	**0**	b_{n-1}	$\ldots$	b_3	b_2	$\rightarrow$

(>>> in Java, `lsr` in OCaml)

- arithmetic shift right (duplicates the sign bit)

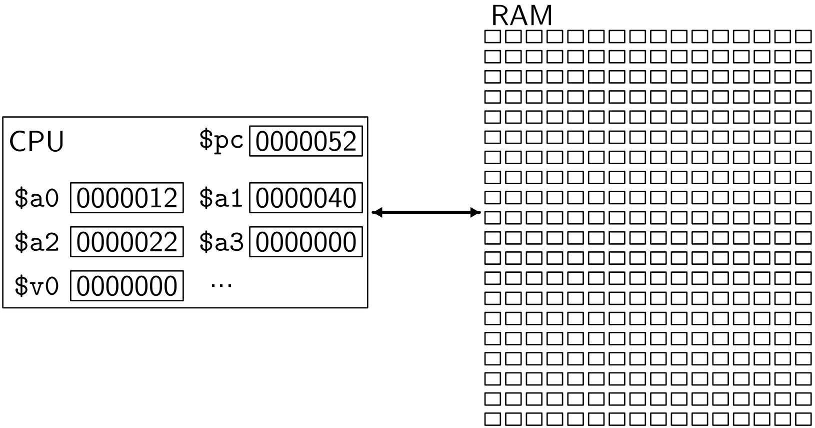
$\rightarrow$	b_{n-1}	b_{n-1}	b_{n-1}	$\ldots$	b_3	b_2	$\rightarrow$

(>> in Java, `asr` in OCaml)

roughly speaking, a computer is composed

- of a CPU, containing
  - few integer and floating-point registers
  - some computation power

- memory (RAM)
  - composed of a large number of bytes (8 bits)
    for instance, 1 GiB $= 2^{30}$ bytes $= 2^{33}$ bits, that is $2^{2^{33}}$ possible states
  - contains data and instructions



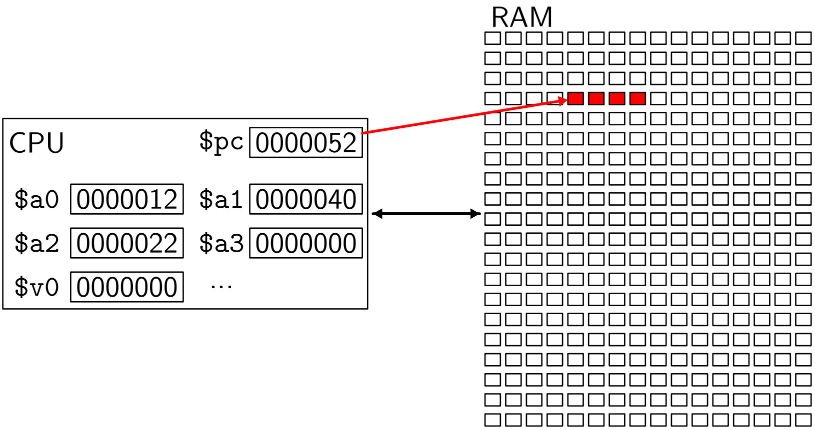
accessing memory is **costly** (at one billion instructions per second, light only traverses 30 centimeters!)

reality is more complex:

- several (co)processors, some dedicated to floating-point
- one or several memory caches
- virtual memory (MMU)
- etc.

execution proceeds according to the following:

- a register (%pc) contains the address of the next instruction to execute
- we read one or several bytes at this address (*fetch*)
- we interpret these bytes as an instruction (*decode*)
- we execute the instruction (*execute*)
- we modify the register %pc to move to the next instruction (typically the one immediately after, unless we jump)



instruction :	001000	00110	00101	0000000000001010
decoding :	addi	%a2	%a1	10

i.e. add 10 to register %a2 and store the result in the register %a1 again, reality is more complex:

- pipelines
  - several instructions are executed in parallel
- branch prediction
  - to optimize the pipeline, we attempt at predicting conditional branches

which architecture for this course?

two main families of microprocessors

- CISC (*Complex Instruction Set*)
  - many instructions
  - many addressing modes
  - many instructions read / write memory
  - few registers
  - examples: VAX, PDP-11, Motorola 68xxx, Intel x86
- RISC (*Reduced Instruction Set*)
  - few instructions
  - few instructions read / write memory
  - many registers
  - examples: Alpha, Sparc, MIPS, ARM

Which architecture to choose for this course?


NEC JAPAN
D30310R -33
VR3000A-33
9215E9 CPU


EyeQ™ 4 Mid

MIPS architecture

- 32 registers, \$0 to \$31
  - \$0 always stores 0
  - used under different names, respecting the following conventions: (`zero`, `at`, `v0`–`v1`, `a0`–`a3`, `t0`–`t9`, `s0`–`s7`, `k0`–`k1`, `gp`, `sp`, `fp`, `ra`)

- conceptually, three kinds of instructions
  - instructions for the **transfer** between registers and memory
  - instructions for **computations** (logical, arithmetic, comparison)
  - instructions for **jumping**

(see the documentation for MIPS)

we do not code in machine language, but using the **assembly language**

the assembly language provides several facilities:

- symbolic names
- allocation of global data

assembly language is turned into machine code by a program called an **assembler** (a compiler) the assembly directive

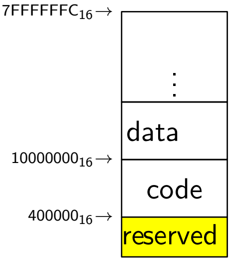
```
        .text
```

indicates that the instructions will follow and the directive

```
        .data
```

indicates that the data will follow

the code will be loaded starting from the address `0x400000`
and the data from the address `0x10000000`



```
        .text
main:   li      $v0, 4      # code of print_string
        la      $a0, hw     # address of the string
        syscall             # system call
        li      $v0, 10     # exit
        syscall
        .data
hw:     .asciiz "hello world\n"
```

(`.asciiz` is to avoid writing explicitly `.byte 104, 101, ... 0`)

running MIPS on our machines: we'll use **SPIM**, a MIPS simulator

demo: hw.s

- storing a constant in a register

```
        li    $a0, 42       # a0 <- 42
        li    $a0, -65536   # a0 <- -65536
```

- storing the address of a label in a register

```
        la    $a0, label
```

- copying the content of a register in a register

```
        move $a0, $a1    # copies a1 in a0!
```

demo: arith.s

- addition of two registers

```
        add  $a0, $a1, $a2  # a0 <- a1 + a2
        add  $a2, $a2, $t5  # a2 <- a2 + t5
```

similarly for `sub`, `mul`, `div`

- addition of a register and a constant

```
        addi $a0, $a1, 42   # a0 <- a1 + 42
```

(but no `subi`, `muli` or `divi`!)

- negation

```
        neg  $a0, $a1       # a0 <- -a1
```

- absolute value

```
        abs  $a0, $a1       # a0 <- |a1|
```

- logical NOT ($\mathtt{not}(100111_2) = 011000_2$)

```
        not  $a0, $a1        # a0 <- not(a1)
```

- logical AND ($\mathtt{and}(100111_2, 101001_2) = 100001_2$)

```
        and  $a0, $a1, $a2  # a0 <- and(a1, a2)
        andi $a0, $a1, 0x3f # a0 <- and(a1, 0...0111111)
```

- logical OR ($\mathtt{or}(100111_2, 101001_2) = 101111_2$)

```
        or   $a0, $a1, $a2  # a0 <- or(a1, a2)
        ori  $a0, $a1, 42   # a0 <- or(a1, 0...0101010)
```

- shift left (inserting zeros)

```
        sll  $a0, $a1, 2    # a0 <- a1 * 4
        sllv $a1, $a2, $a3  # a1 <- a2 * 2^a3
```

- arithmetic shift right (duplicating the sign bit)

```
        sra  $a0, $a1, 2    # a0 <- a1 / 4
```

- logical shift right (inserting zeros)

```
        srl  $a0, $a1, 2
```

- rotation

```
        rol  $a0, $a1, 2
        ror  $a0, $a1, 3
```

- comparison of two registers

```
        slt  $a0, $a1, $a2    # a0 <- 1 if a1 < a2
                              #       0 otherwise
```

or of a register and a constant

```
        slti $a0, $a1, 42
```

- variants : `sltu` (unsigned comparison), `sltiu`
- similarly : `sle`, `sleu` / `sgt`, `sgtu` / `sge`, `sgeu`
- equality tests : `seq`, `sne`

instruction set: transfer (reading)

- reading a word (32 bits) from memory

```
lw   $a0, 44($a1)   # a0 <- mem[a1 + 44]
```

  the address is given by a register and an offset over 16 signed bits

- variants for reading 8 or 16 bits, signed or not (`lb`, `lh`, `lbu`, `lhu`)

instruction set : transfer (writing)

- storing a word (32 bits) in the memory

```
sw   $a0, 44($a1)   # mem[a1 + 44] <- a0
                    # pay attention to the direction!
```

  the address is given by a register and an offeset over 16 signed bits

- variants for writing 8 or 16 bits (`sb`, `sh`) we distinguish

- **branching** : conditional jump, where the destination address is stored over 16 signed bits (from -32768 to 32767 instructions)
- **jump** : unconditional jump, where the destination address is stored over 26 bits

- conditional branching

```
beq  $a0, $a1, label # if a0 = a1 then jump to label
                     # otherwise do nothing
```

- variants: `bne`, `blt`, `ble`, `bgt`, `bge` (and unsigned comparisons)
- variants: `beqz`, `bnez`, `bgez`, `bgtz`, `bltz`, `blez`

unconditional jump

- to an address (*jump*)

```
        j    label
```

- saving the address of the instruction following the jump in %ra ("return address register")

```
        jal  label    # jump and link
```

- to an address stored in a register

```
        jr   $a0
```

- saving the address of the instruction following the jump in a register

```
        jalr $a0, $a1
```

some system calls are provided by the special instruction

```
        syscall
```

the code of the instruction must be stored in the register %v0, the arguments in the registers %a0–%a3 ;
and the return result (if any) will be stored in the register %v0

example : system call print_int to print an integer

```
        li      $v0, 1      # code for print_int
        li      $a0, 42     # value to print
        syscall
```

similarly for read_int, print_string, etc. (see the documentation)

many of those instructions are in fact **pseudo-instructions** :
they are translated by the assembler in a single or multiple machine instructions

example : when we write

```
  li    $a0, 42
```

assembler translates it into

```
  addiu $a0, $zero, 42
```

another example :

if the label hw corresponds to an address 0x10010020, then the instruction

```
  la    $a0, hw
```

is translated by the assembler into

```
  lui   $at, 0x1001       # load upper immediate
  ori   $a0, $at, 0x0020
```

$at, known as the "assembler temporary" register, is a special-purpose register used by the assembler for temporary storage

demo: fact_loop.s

is to translate a high-level program into this instruction set

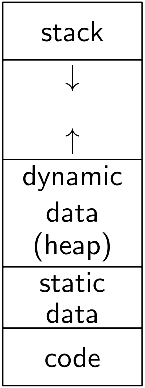
in particular, we have to

- translate control structures (tests, loops, exceptions, etc.)
- translate function calls
- translate complex data structures (arrays, structures, objects, closures, etc.)
- allocate dynamic memory

observation: function calls can be arbitrarily nested
$\Rightarrow$ registers cannot hold all the local variables
$\Rightarrow$ we need to allocate memory

yet function calls obey a *last-in first-out* mode, so we can use a **stack**

the stack



the **stack** is allocated at the top of the memory, and increases downwards; `%sp` points to the top of the stack

dynamic data (which needs to survive function calls) is allocated on the **heap** above static data, and increases upwards

this way, no collision between the stack and the heap (unless we run out of memory)

note: each program has the illusion of using the whole and contiguous memory; the OS creates this illusion, using the MMU (Memory Management Unit) when a *f* (*caller*) wants to call a function *g* (*callee*), it executes the instruction

```
        jal  g
```

and when the *callee* has finished the work, it gives the control back to the *caller* with the instruction

```
        jr   $ra
```

problem:

- if *g* itself calls yet another function, `%ra` will be overwritten
- similarly, any register used locally by *g* will be lost for *f*

there are many solutions, but we typically resort to **calling conventions**

- `%ra` stores the **return address**
- `%a0`–`%a3` used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and `%v0`–`%v1` to **return the result**
- `%sp` is **stack pointer** (pushing and popping values, moves downward)
  `%fp` is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- `%t0`–`%t9` are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- `%s0`–`%s7` are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- `%at`, `%k0` and `%k1` are reserved to assembler and OS
- `%gp` points to the middle of a 64K block of memory in the static data segment (10008000_{16})

- `%ra` stores the **return address**
- `%a0`–`%a3` used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and `%v0`–`%v1` to **return the result**
- `%sp` is **stack pointer** (pushing and popping values, moves downward) `%fp` is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- `%t0`–`%t9` are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- `%s0`–`%s7` are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- `%at`, `%k0` and `%k1` are reserved to assembler and OS
- `%gp` points to the middle of a 64K block of memory in the static data segment (10008000_{16})

- %ra stores the **return address**
- %a0–%a3 used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and %v0–%v1 to **return the result**
- %sp is **stack pointer** (pushing and popping values, moves downward)
  %fp is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- %t0–%t9 are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- %s0–%s7 are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- %at, %k0 and %k1 are reserved to assembler and OS
- %gp points to the middle of a 64K block of memory in the static data segment (10008000_{16})

- %ra stores the **return address**
- %a0–%a3 used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and %v0–%v1 to **return the result**
- %sp is **stack pointer** (pushing and popping values, moves downward)
  %fp is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- %t0–%t9 are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- %s0–%s7 are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- %at, %k0 and %k1 are reserved to assembler and OS
- %gp points to the middle of a 64K block of memory in the static data segment (10008000_{16})

- `%ra` stores the **return address**
- `%a0`–`%a3` used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and `%v0`–`%v1` to **return the result**
- `%sp` is **stack pointer** (pushing and popping values, moves downward)
  `%fp` is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- `%t0`–`%t9` are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- `%s0`–`%s7` are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- `%at`, `%k0` and `%k1` are reserved to assembler and OS
- `%gp` points to the middle of a 64K block of memory in the static data segment (10008000_{16})

- `%ra` stores the **return address**
- `%a0`–`%a3` used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and `%v0`–`%v1` to **return the result**
- `%sp` is **stack pointer** (pushing and popping values, moves downward)
  `%fp` is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- `%t0`–`%t9` are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- `%s0`–`%s7` are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- `%at`, `%k0` and `%k1` are reserved to assembler and OS
- `%gp` points to the middle of a 64K block of memory in the static data segment (10008000_{16})

- `%ra` stores the **return address**
- `%a0`–`%a3` used to pass the **first 4 arguments** (the other arguments will be passed on the stack) and `%v0`–`%v1` to **return the result**
- `%sp` is **stack pointer** (pushing and popping values, moves downward) `%fp` is **stack frame** pointer (base of the current stack frame, useful for accessing function parameters & local variables at fixed offsets; remains constant during function's execution)
- `%t0`–`%t9` are **caller-saved** registers used to hold temporary quantities that need not be preserved across calls (i.e. the caller must save them if needed before the call)
- `%s0`–`%s7` are **callee-saved** registers that hold long-lived values that should be preserved across calls (i.e. the callee must save them)
- `%at`, `%k0` and `%k1` are reserved to assembler and OS
- `%gp` points to the middle of a 64K block of memory in the static data segment (10008000_{16})

there are four steps in a function call:

1. for the caller, **just before** the call
2. for the callee, **at the beginning** of the call
3. for the callee, **at the end** of the call
4. for the caller, **just after** the call

the stack segment where the effect of those steps takes place is called **stack frame** located right on the top of the stack between `%fp` and `%sp`

1. passes arguments in %a0–%a3, and others on the stack, if more than 4
2. saves caller-saved registers %t0–%t9, in its own stack frame, if they are needed after the call
3. executes

```
        jal   callee
```

caller-saved = not preserved across function calls.

- The caller must save them if needed.
- The callee (function) is free to modify them without saving.

demo: caller_saved.s

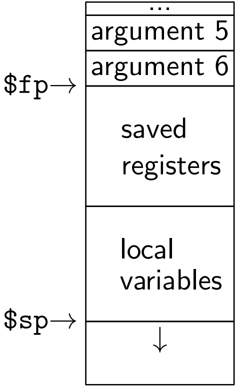
1. allocates its stack frame, e.g.

```
addi  $sp, $sp, -28
```

2. saves %fp on the stack frame and moves it, e.g.

```
sw    $fp, 24($sp)
addi  $fp, $sp, 24
```

3. saves %s0–%s7 and %ra if needed



%fp eases access to arguments and local variables, with a fixed offset (whatever the top of the stack)

demo: callee_saved.s

1. stores the result in `%v0` (or `%v1`)
2. restores the callee-saved registers, including `%fp` for example

```
        lw   $fp, 24($sp)
```

3. destroys its stack frame, e.g.

```
        addi $sp, $sp, 28
```

4. and executes

```
        jr   $ra
```

the caller, just after the call

1. pops arguments 5, 6, ... (if any)
2. restores the caller-saved registers `%t0`–`%t9`, if needed

exercise : let's implement the following function

$$
\begin{array}{l}
\text{isqrt}(n) \equiv \\
\quad c \leftarrow 0 \\
\quad s \leftarrow 1 \\
\quad \text{while } s \leq n \\
\quad\quad c \leftarrow c + 1 \\
\quad\quad s \leftarrow s + 2c + 1 \\
\quad \text{return } c
\end{array}
$$

the idea why it works is that the invariant of the loop is $s_i = (c_i + 1)^2$ so that when c is returned, we have $c^2 \leq n < (c + 1)^2$

i	0	1	2	...	i	$i+1$
c	0	1	2	...	i	$i+1$
s	1	3	9	...	$(i+1)^2$	$(i+1)^2 + 2(i+1) + 1 = ((i+1)+1)^2$

demo: isqrt.s

exercise : let's program factorial with a recursive function

demo: fact_recursive.s

- a machine provides
  - a limited instruction set
  - efficient registers, costly access to the memory
- the memory is split into
  - code / static data / dynamic data (heap) / stack
- function calls make use of
  - a notion of stack frame
  - calling conventions

- **lesson:** producing efficient assembly code is not easy,
  we have to automate all this in a compiler

- writing manually some small MIPS programs
- implementing a compiler for a mini-language of arithmetic expressions generating automatically MIPS code

  https://homes.cs.aau.dk/~lego/compil25/lab_sessions/7/index.html