Assembly code

C. Paulin (courtesy of T. Balabonski)

M1 MPRI 2025–26

1. A target language : assembly code
   - MIPS architecture and binary code
   - Assemby language
   - Functions and stack-frames
   - Call convention
   - Optimizing tail-recursive functions

Compilers

- Multiple steps :
  - program analysis :
    find the structure and meaning of the program, reject bogus entries
  - program synthesis :
    transform the abstract syntax tree of the source program into another program, called the target program, “equivalent”
- Possible targets :
  - another high-level language (C, javascript,. . .)
  - bytecode : low-level code for a virtual machine (java, caml, python) that will be interpreted
  - assembly language : directly executable by the machine after assembling and linking

MIPS architecture

- MIPS stands for (Microprocessor without Interlocked Pipeline Stages)
- Processor RISC (Reduced Instruction Set Computer), 32 bits, precursor of modern RISC-V architectures.
- used in embedded systems, printers, smartphones…
- a processor, with a few registers and computing units
- a large amount of memory to store data and intructions

Binary code for MIPS

- Memory unit : a byte (8 bits) represented in hexa by 2 characters
- 32 bits architecture : a word corresponds to 4 bytes (8 hexa characters)
- Each instruction is encoded on a 32 bits word
  - opcode : 6 first bits encode the instruction, the rest are the arguments
- 32 general-purpose registers (need 5 bits to identify them)
  - computations are done between registers (a few accept an argument which is an immediate value - constant)
- General memory contains 2^{32} bytes, each one has an address (from $0x00000000$ to $0xFFFFFFFF$),
  Mainly use addresses which are a multiple of 4

Binary code for MIPS

- 3 forms of instructions depending on the opcode
  - R-instruction : involves 3 registers, 2 operands and a destination (11 bits left)
  - I-instruction : involves 2 registers and an immediate value on 16 bits
  - J-instruction : jump to a target represented on 26-bits (the 2 last bits are ommited because they should be 00)
- A uniform execution model
  - fetch the instruction
  - decode the instruction
  - execute the instruction
  - do the memory transfers
  - update the registers (including the code pointer)

  (the different steps are pipelined)

Binary code vs assembly language

- Binary code combines two difficulties
  - low level computation model
  - binary (often arbitrary) encoding of operations and data
  - limitation because of the size of the instruction : (an immediate value in an addition can only contain 16 bits)
- Assembly language (an high-level machine language)
  - names instead of binary code
    - for instructions, registers, addresses. . .
  - pseudo instructions (a small sequence of instructions) for current operations
  - focus on the low-level model of execution, not the encoding

Assembly code MIPS

1. A target language : assembly code
   - MIPS architecture and binary code
   - Assemby language
   - Functions and stack-frames
   - Call convention
   - Optimizing tail-recursive functions

Assembly code MIPS : advantage

- explicit names for operations (binary instructions but also pseudo-instructions)
- labels for identifying code addresses
- an easy way to position static data

Registers

number	name	purpose
\$0	`$zero`	value is always 0
\$1	`$at`	Assembler Temporary (reserved)
\$2, \$3	`$v0,$v1`	returned value
\$4 – \$7	`$a0-$a3`	arguments of a functions
\$8 – \$15, \$24, \$25	`$t0-$t9`	temporary values
\$16 – \$23	`$s0-$s7`	saved temporary values
\$26, \$27	`$k0,$k1`	Kernel (reserved)
\$28	`$gp`	Global pointer (do not touch)
\$29	`$sp`	Stack pointer, on top, last value
\$30	`$fp`	Frame pointer, , for function call
\$31	`$ra`	Return address, for function call

Other registers, cannot be addressed explicitely, used via specific instructions

- `$pc` for program counter
- `$hi,$lo` for multiplication (keep 64 bits) between integers and division (bo both division and reminder)

Instructions between registers

- adding 2 registers `$rs` and `$rt`, the result goes in `$rd`

  ```
  add $rd,$rs,$rt
  ```

  fails if the sum overflows
  same for `mul`, `sub`, `div`, `and`, `or` (bitwise).
- adding 1 register `$rs` and an immediate number *n* (16 bits), the result goes in `$rd`

  ```
  addi $rd,$rs,n
  ```

  same for `andi`, `ori`
- négation of `$rs` goes in `$rd`

  ```
  neg $rd,$rs
  ```

  same for `abs`, `non`

Shift and rotate operations

- shift the word in `$rs` to the left, inserting *n* zeros

  `sll $rd,$rs,`*n*

  Same for `srl` (right logical) and `sra` (right arithmetical : insert the sign bit), `rol` (left rotation), `ror` (right rotation),

Comparison

- compare 2 registers `$rs` and `$rt`, the result is 1 if `$rs<$rt` and 0 otherwise and it goes in `$rd`.

  ```
  slt $rd,$rs,$rt
  ```

  same for `sle`, `sgt`, `sge`, `seq`, `sne`, `sltu`, `sleu`, `sgtu`, `sgeu` for unsigned versions
- compare 1 register `$rs` and an immediate value *n*, the result is 1 if `$rs<n` and 0 otherwise and goes in `$rd`.

  ```
  slti $rd,$rs,n
  ```

  same for `slei`, `sgti`, `sgei`, `sltiu`, `sleiu`, `sgtiu`, `sgeiu` for unsigned versions

Branch, jump

- Branch instructions : relative jump from current address, a number represented on 16 bits (local move for loops, conditional...)
  if `$rs<$rt` then goes to instruction *etiq*

  ```
  blt $rs,$rt,etiq
  ```

  same for `ble`, `bgt`, `bge`, `beq`, `bne`, `bltu`, `bleu`, `bgtu`, `bgeu` for unsigned versions
  if `$rs<0` then goes to instruction *etiq*

  ```
  bltz $rs,etiq
  ```

  same for `blez`, `bgtz`, `bgez`, `beqz`, `bnez`
- Jump instruction : absolute jump to an address given explicitely (26 bits) or in a register
  - jump to the address *etiq*

    ```
    j etiq
    ```

  - save the next code address in `$ra` and jump to the address *etiq*

    ```
    jal etiq
    ```

  - jump to the address stores in register `$rs`

    ```
    jr $rs
    ```

    `jral` to save the next code address and jump

Compiling conditionals

```
#Code : condition evaluated in $t0
beqz $t0, lab_else
#Code : branch then
b lab_end
lab_else:
#Code : branch else
lab_end
```

- generate 2 new labels for each conditional
- don't forget to skip the code of the else branch after executing the then branch

Compiling loops

```
b lab_test
lab_loop:
#Code : body
lab_test
#Code : condition evaluated in $t0
bnez $t0, lab_loop
```

- two labels, one for the loop and the other for the test
- a test is always performed after each loop so putting the test code just after the body avoids an extra jump at each iteration

Memory access

- A memory address is composed of a base address (a value given by a label *etiq* or stored in a register `$r`) plus a shift number *d* (16 bits, signed) It is written : *d*`($r)` or *`etiq`*`+d`
- load word : place the value stored at address *d*`($rs)` in `$rd`

  `lw $rd` *d*`($rs)`

  Same for `lb`, `lh`, `ld`, to read bytes, half-words or double (using also next register)
- store word : place the value of `$rt` at address *d*`($rs)`

  `sw $rt` *d*`($rs)`

  Same for `sb`, `sh`, `sd`, to read bytes, half-words or double
- load address : place the value of the address itself *d*`($rs)` (not its contents) in register `$rd`

  `la $rd` *d*`($rs)`

System calls

- For input/output, memory allocation, the assembly calls the operating system
- In the MARS simulator : instruction `syscall`, with a parameter in `$v0` which gives the operations and possible argument in `$a0`

service	code	arg.	result
printing	1 (integer) 4 (string) 11 (ascii)	`$a0`	
read input	5 (integer)		`$v0`
stop	10		
memory extension	9	`$a0`	`$v0`

Static data

- Data with size known at compile time can be put in the `.data` segment
- Example : an array of integers, a string (1 byte by character) ...

```
.data
t: .word 1 2 3 4 5
s: .asciiz "Hello␣world!"
```

- When initial values are unknown, a default value may be used (or another assembly directive)
- the label gives the address of the first value, the remaining elements are accessed by pointer arithmetic (+4 for words, +1 for bytes values)

Dynamic data : motivations

- Evaluating an expression requires storing intermediate values
- The number of intermediate values depends on the shape of the expression
  - Assume an expression Bop(o,e1,e2), we need enough registers to compute e1, then one register `$r` to store the value of e1 and then enough registers (different from `$r`) to compute e2.
    Unless e2 is an immediate value, we need at least two registers, and now if the expression is Bop(o,e1,Bop(o,e2,e3)) we shall need at least 3 different registers, and so on. . .
- We need to store intermediate values, we shall dedicate part of the memory to implement a stack

Implementing a stack

`.text`	`.data`		top	bottom
code	static data	... $\leftarrow$	$\rightarrow$	stack

- The bottom of the stack is at the highest address
- The register `$sp` gives the address of the last(top) element of the stack
- Each time an element is added on the stack, the pointer should be decreased by 4

Standard operations with the stack

- Read the value on top of the stack (peek),

```
lw    $t0, 0($sp)
```

  Accessing deeper values by adding $4 \times n$, with n the index. The 4th element has index 3 and is accessed with a shift of 12 bytes.

```
lw    $t0, 12($sp)
```

- Adding an element on the stack requires changing $sp and writing the value

```
addi  $sp, $sp, -4
sw    $t0, 0($sp)
```

- Removing an element just requires just changing $sp

```
addi  $sp, $sp, 4
```

  The pop instruction transfers the value on top of the heap to a register

```
lw    $t0, 0($sp)
addi  $sp, $sp, 4
```

Dynamic data : the heap

- Some time a data is created during execution
- Storage in the stack is only for temporary computations
  - if an array is created during a function call, it will disappear after the function returns (even if its address is returned by the function, it will point on a memory portion reused for other purposes)
- Need of a space with persistent data : the heap

Implementing the heap

.text	.data		brk		$sp
code	static data	heap	→	... ←	stack

- the `brk` pointer (memory break) points on the first free space on the heap
- the operating system is responsible for allocating memory
  - syscall with parameter `$v0=9`, uses `$a0` for the amount of bytes needed, returns the old value of `brk` (address of the first byte of the newly allocated space) in `$v0`

Example

instruction	`$a0`	`$v0`	`$t0`	`brk`
				0x10040000
`li    $a0, 24`	24			0x10040000
`li    $v0, 9`	24	9		0x10040000
`syscall`	24	0x10040000		0x10040018
`li    $t0, 1`	24	0x10040000	1	0x10040018
`sw    $t0, 0($v0)`	24	0x10040000	1	0x10040018
`li    $t0, 32`	24	0x10040000	32	0x10040018
`sw    $t0, 20($v0)`	24	0x10040000	32	0x10040018

After theses commands, the heap has the following shape
The address $@_1$ is the initial position of break brk (0x10040000) and $@_2$ is the new position after calling sbrk (0x10040018).

	$@_1$						$@_2$
...	1					32	...

Assembly code : MIPS

1. A target language : assembly code
   - MIPS architecture and binary code
   - Assemby language
   - Functions and stack-frames
   - Call convention
   - Optimizing tail-recursive functions

Function calls

```
let f x = x * 7 in
let g x y = let z = f (x + y + 6) in z * z
in g 0 0
```

- We have two function definitions f and g
- The body of g calls function f
  - g is the caller and f the callee
  - x is the formal parameter of f and x * 7 is the returned value
  - x+y+6 is the effective parameter (or argument) of f

Function calls in assembly

- In the assembly language we have code for `f` and code for g
- The code for `f` needs to know where to find the value of its argument (for instance in `$a0`) and where to return the result (for instance in `$v0`)
- The code for g put the value of the effective parameter in `$a0` and then tranfers the code pointer to the address of `f`, and expects the result in `$v0`
- But how can `f` know where to return after finishing ? (we usually have several different calls to the same function)
- g should save the return point before calling `f`, in a designated place, this is the purpose of the `$ra` register.
- the caller save the return point in `$ra` before jumping to the `f` code address

```
jal f
```

- at the end of `f`, a jump is done back to the address stored in `$ra`

```
jr $ra
```

Imbricated calls

- What happens if the function f itself calls another function h ?
  - Need to give `$ra` to h for a safe return in f, but the return address to g will be lost
- Other registers also might be used both by f and g with a risk of collision
- The number of imbricated calls is unknow at compile-time (think of a recursive function)
- The solution is to save the values needed for a function call in a “frame”.
- A callee always finishes before the caller can finish, and then its frame can be destroyed. So the frame can safely be put on the stack, it is called a stack-frame.
- The structure of the stack-frame is known at compile-time
- On the stack : all frames of the current active calls

Format of a stack frame

- `$ra` is one of the saved registers
- the register `$fp` (frame-pointer) contains the address of the last saved register
- other elements of the frame are accessed relatively to the `$fp` address

Examples

- stack frame for f (one parameter, no local variables)

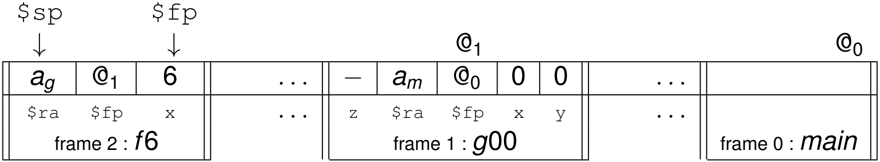
(-4)	$fp ↓	(+4)
$ra		x
saved		arg.

- stack frame for g (two parameters, one local variable)

(-8)	(-4)	$fp ↓	(+4)	(+8)
z	$ra		x	y
loc. var.	saved		arg.	

Compiling a call

- At the `$fp` address in the callee frame, we shall store the previous value of `$fp` (to be restored before the callee returns)
- In our example, on the stac, we shall have
  - the frame of the `main` function
  - the frame of the `g` function
  - the frame of the `f` function



1. A target language : assembly code
- MIPS architecture and binary code
- Assemby language
- Functions and stack-frames
- Call convention
- Optimizing tail-recursive functions

Protocol

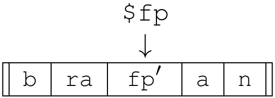
- Agrement between the caller and the callee on who is doing what and where
- Simple version
  1. **Caller, before the call :** evaluate the arguments, put them on the stack (last one first) call `jal` (or `jalr`), the callee is now in charge.
  2. **Callee, beginning of the call :** saves values of $fp and $ra on the stack, reserve space for local values, set $fp to the current address of the frame.
  3. **Callee :** execute the body of the function.
  4. **Callee, end of the call :** put the result in the dedicated register ($t0,$v0), restore $fp and $ra, free stack space reserved at step 2, et give control back to the caller `jr $ra`.
  5. **Caller, after the call :** remove the arguments on the stack.
- The stack frame is created at steps 1 and 2 and destroyed at step 4 and 5
- The stack is back to its initial form with computed value in $t0 (or $v0).

Example

We use FUN syntax, but without considering functions as first class values

```
let rec power a n =
if n = 0 then 1
else let b = power (a*a) (n>>1)
in if n && 1 = 0 then b else a * b
in power 2 9
```

Stack frame



On accède aux arguments

- access to a and n respectively at the adresses \$fp+4 and \$fp+8, and to the local variable b at the adresse \$fp–8.

Protocol for calling power

Call power 2 9 : put the arguments on the stack, call power

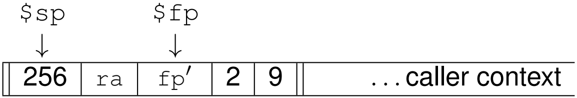
```
li      $t0, 9           # load 9
addi    $sp, $sp, -4     # store
sw      $t0, 0($sp)
li      $t0, 2           # load 2
addi    $sp, $sp, -4     # store
sw      $t0, 0($sp)
jal     power
```

The power function build the frame

```
addi    $sp, $sp, -12    # alloc (2 reg. + loc. var.)
sw      $fp, 8($sp)      # save fp
sw      $ra, 4($sp)      # save ra
addi    $fp, $sp, 8      # set fp
```

Protocol for calling `power`

After some computation, including recursive calls, we return a∗b in `$t0`
The local variable b contains the value 256 (power(4, 4)).




```
lw      $t0, 4($fp)        # t0 <- a
lw      $t1, -8($fp)       # t1 <- b
mul     $t0, $t0, $t1      # t0 <- a*b
lw      $ra, 4($sp)        # restore ra
lw      $fp, 8($sp)        # restore fp
addi    $sp, $sp, 12       # clean the stack
jr      $ra                # return to g
```

Protocol for calling `power`

Back in main after the call

```
addi     $sp, $sp, 8      # clean the stack
move     $a0, $t0         # print the result
li       $v0, 1
syscall                   # affichage
```

Code for power

```
    power:
    # build the stack frame (step 2)
052     addi    $sp, $sp, -12
056     sw      $fp, 8($sp)
060     sw      $ra, 4($sp)
064     addi    $fp, $sp, 8
    # body of the function (step 3)
068     lw      $t0, 8($fp)      # test (n = 0)
072     bnez    $t0, power_rec
076     li      $t0, 1           # t0 <- 1
080     b       power_end
```

Code for power

```
power_rec: # recursive call
    # step 1
084     lw      $t0, 8($fp)     # push argument (n>>1)
088     sra     $t0, $t0, 1
092     addi    $sp, $sp, -4
096     sw      $t0, 0($sp)     # push argument (a*a)
100     lw      $t0, 4($fp)
104     mul     $t0, $t0, $t0
108     addi    $sp, $sp, -4
112     sw      $t0, 0($sp)
116     jal     power           # call
    # after the recursive call (STEP 5)
120     addi    $sp, $sp, 8     # clean arguments
```

```
    # back to the body of power
124     sw      $t0, -8($fp)    # b <- power (a*a) (n>>1)
128     lw      $t0, 8($fp)     # test (n&&1 = 0)
132     andi    $t0, $t0, 0x1
136     bnez    $t0, power_odd
140     lw      $t0, -8($fp)    # t0 <- b
144     b       power_end
    power_odd:
148     lw      $t0, 4($fp)
152     lw      $t1, -8($fp)
156     mul     $t0, $t0, $t1   # t0 <- a*b
    power_end:
    # end of call (STEP 4)
160     lw      $ra, 4($sp)    # restore ra
164     lw      $fp, 8($sp)    # restore fp
168     addi    $sp, $sp, 12   # clean the stack
172     jr      $ra            # back to caller
```

Full code for the main function

```
    # call power 2 9
000     li      $t0, 9              # prepare arguments
004     addi    $sp, $sp, -4
008     sw      $t0, 0($sp)
012     li      $t0, 2
016     addi    $sp, $sp, -4
020     sw      $t0, 0($sp)
024     jal     power               # call
028     addi    $sp, $sp, 8         # clean arguments
    # print result
032     move    $a0, $t0
036     li      $v0, 1              # code 1 : print integer
040     syscall
    # end
044     li      $v0, 10             # code 10 : stop
048     syscall
```

Stack frames

$sp ↓		$fp ↓					$@_2$					$@_1$				
-	120	$@_2$	256	1	-	120	$@_1$	16	2	-	120	$@_0$	4	4	-	028
power(256, 1)					power(16, 2)					power(4, 4)					power(	

Protocol MIPS with registers

- we want to use registers for variables or intermediate values to improve efficiency
- the caller/callee protocol should agree on which registers can be used freely and which one should be saved
- 2 sorts of registers :
  - callee saved : if the callee used them, it should save the old values and restore them afterwards
    registers `$s*` (saved) and `$fp`
  - caller saved : the callee is free to use them, if the caller want to preserve them, it should save them
    registers `$a*` (arguments) `$t*` (temporary) and `$ra`

Example for power

- use `$a0` and `$a1` for arguments and `$t2` for the variable b

Main program

```
    # Call power 2 9
000'     li      $a1, 9           # préparation des arguments
012'     li      $a0, 2
024      jal     power            # appel
    # Print
032'     move    $a0, $v0
036      li      $v0, 1
040      syscall
    # End
044      li      $v0, 10
048      syscall
```

Example for power

Stack frame needs room to save `$ra`, `$fp`, and (if needed) also `$a0`, `$a1`,

```
    power: # stack frame
052     addi    $sp, $sp, -16    # 4 words reserved
056     sw      $fp, 12($sp)
060     sw      $ra, 8($sp)
064     addi    $fp, $sp, 12
# Execute the body of the function
072'    bnez    $a1, power_rec   # test (n = 0)
076     li      $v0, 1           # v0 <- 1
080     b       power_end
    power_rec: # recursive call
084'    sw      $a0, -8($fp)     # save a0 and a1
084''   sw      $a1, -12($fp)
088'    sra     $a1, $a1, 1      # argument (n>>1)
104'    mul     $a0, $a0, $a0    # argument (a*a)
116     jal     power            # call
120'    lw      $a0, -8($fp)     # restore a0 and a1
120''   lw      $a1, -12($fp)
```

Example for power

```
    # back to the body
124      move    $t2, $v0          # b <- power(a*a) (n>>1)
132'     andi    $t0, $a1, 0x1     # test (n&&1 == 0)
136      bnez    $t0, power_odd
140      move    $v0, $t2          # v0 <- b
144      b       power_end
    power_odd:
156'     mul     $v0, $t2, $a0     # v0 <- a * b
    power_end:
    # clean stack frame
160      lw      $ra, 8($sp)
164      lw      $fp, 12($sp)
168      addi    $sp, $sp, 16
172      jr      $ra
```

Simplification

- Use directly `$v0` instead of `$t2` for local variable
- Simplify branching at the end

```
132'      andi      $t0, $a1, 0x1     # test (n&&1 = 0)
136'      beqz      $t0, power_end
156'      mul       $v0, $v0, $a0     # v0 <- a*b
     power_end:
```

Assembly code : MIPS

1. A target language : assembly code
   - MIPS architecture and binary code
   - Assemby language
   - Functions and stack-frames
   - Call convention
   - Optimizing tail-recursive functions

Optimizing tail-recursive functions

- If an argument is not used after a call then it does not need to be saved and restored
- For instance *n* in the following function

```
let power a n =
if n = 0 then 1
else if n && 1 = 0 then power (a*a) (n>>1)
else a * (power (a*a) (n>>1))
```

- Change code such that when a recursive call is made, this is the last instruction of the caller
- The arguments will not need to be saved
- No frame stack is needed

Tail-recursive functions/terminal call

```
let power a n = power_aux a n 1

let rec power_aux a n acc =
      if n = 0 then acc
      else if n && 1 = 0 then power_aux (a*a) (n>>1) acc
      else power (a*a) (n>>1) a*acc

in power 2 9
```

Terminal call **let** power a n = power_aux a n 1
the value computed by power_aux a n 1 is the result of power a n and the next instruction after power_aux is the return address of the call to power

- no need to store `$ra` before calling power_aux, we do a simple jump
- no use of the stack frame of power after the call

Power code

Main part as before

```
    power:
32      li      $a2, 1              # initialise the accumulato
36      j       power_aux           # call power_aux a n 1
    power_aux:
40      beqz    $a1, power_end
44      andi    $t0, $a1, 0x1       # test (n = 0)
48      beqz    $t0, power_even
52      mul     $a2, $a0, $a2       # acc' <- a*acc
    power_even:
56      mul     $a0, $a0, $a0       # a' <- a*a
60      sra     $a1, $a1, 1         # n' <- n>>1
64      j       power_aux
    power_end:
68      move    $v0, $a2            # return acc
72      jr      $ra
```

Terminal call

- The code works only with registers
- Code equivalent to the one of a loop

```
let  power a n =
  let acc = 1 in
  while (n <> 0) do
    if (n&&1 <> 0) then acc:= a*acc
    a := a*a;
    n := n>>1
  done;
  acc
}
```

- Optimisation found in a language like ocaml which intensively uses recursive functions
- Also possible with high-level optimisations in gcc
- Not done in languages like Python, Java...

Summary

- A low-level programming languages (binary or assembly)
  - sequence of instructions (no control structures, just jump)
  - computation between registers
  - explicit organisation of a linear memory (no data-structures, no variables)
  - stack and heap
- Schemes to translate high-level features
  - loops, conditionals, global variables, computation of expressions
  - function call (communication protocol between caller/callee)
- Optimize the generated code
  - using registers instead of the stack