

Unit 3. Processor I: Instruction set and machine code

Escuela Politécnica Superior - UAM

Outline

- **Computer architecture**
- Assembler language
- ISA MIPS. Instruction set
- Programming

Computer architecture

APPLICATION SOFTWARE	PROGRAMS
OPERATING SYSTEMS	DRIVERS
ARCHITECTURE	INSTRUCTIONS REGISTERS
MICRO-ARCHITECTURE	DATAPATH CONTROLLERS
LOGIC	ADDERS MEMORIES
DIGITAL CIRCUITS	LOGIC GATES
ANALOG CIRCUITS	AMPLIFIERS FILTERS
DEVICES	TRANSISTORS DIODES
PHYSICS	ELECTRONS

- **Architecture:**

- The programmer's view of the computer.
- Defined by instructions (operations) and the operand locations

- **Microarchitecture:**

- The hardware implementation of the computer / microprocessor (U4 and U5).

Computer architecture

- To command a computer, you must understand its language.
 - ✓ **Instructions:** words in a computer's language
 - ✓ **Instruction set:** the vocabulary of a computer's language, its "dictionary"
- Each computer / microprocessor has its own instruction set
- Instructions indicate the operation to perform and the operands to use.
 - ✓ **Machine language:** computer-readable format (1's and 0's)
 - ✓ **Assembly language:** human-readable format of instructions.

"Both languages have a biunivocal correspondence"

Computer architecture

- MIPS architecture (*Microprocessor without Interlocking Pipeline Stages*)
 - ✓ MIPS is a real architecture used in many commercial systems like Silicon Graphics, Nintendo, and Cisco.
 - ✓ MIPS is a RISC (***Reduced Instruction Set Computer***) architecture.
 - ✓ The RISC concept was developed by Hennessy and Patterson in the 80's.
 - ✓ RISC is based on three design principles:
 1. Simplicity favors regularity
 2. Make the common case fast
 3. Smaller is faster
- Once you've learned one architecture, it's easy to learn others.

Outline

- Computer architecture
- **Assembler language**
- ISA MIPS. Instruction set
- Programming

Assembly Language

Design Principle 1: Simplicity favor regularity

Consistent instruction format

Same number of operands (two sources and one destination).

Easier to encode and handle in hardware

High-level code (C)

```
a = b + c;
```

```
a = b - c;
```

// MIPS assembly code

```
add a, b, c
```

```
sub a, b, c
```

add, sub: mnemonics indicate what operation to perform (addition, subtraction)

b, c: source operands on which the operation is performed

a: destination operand to which the result is written

Assembly Language

Design principle 2: Make the common case fast

MIPS is a RISC architecture. Other architectures, such as Intel's IA-32 found in most PC's, are CISC (*Complex Instruction Set Computer*).

- ✓ In a RISC architecture, such as MIPS, only simple, commonly used instructions are included.
- ✓ Hardware to decode and execute the instruction can be simple, small, and fast.
- ✓ More complex instructions (that are less common) can be performed using multiple simple instructions.

High-level code (C)

```
a = b + c - d;
```

// MIPS assembly code

```
add t, b, c      # t = b + c
sub a, t, d      # a = t - d
# comment to the end of line
```


Assembly Language

Design Principle 3: Smaller is faster

- ✓ MIPS includes only a small number of registers
- ✓ Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.

Operands

- ✓ An operand is a binary word which represents a variable or an *immediate* (constant)
- ✓ A computer needs a physical location from which to retrieve binary operands
- ✓ A computer retrieves operands from:
 - Memory: high capacity, but slow access
 - Registers: lower capacity, but fast access
 - MIPS has 32 registers, each one of 32 bits.
 - **MIPS is a 32 bits architecture because its ALU manages 32 bits operands.**
 - Constants (also called *immediates*)

Operands: Registers

- Operands of an instruction are usually registers or immediates (constants).

High-level code (C)

```
a = b + c;
```

// MIPS assembly code

```
#$s0=a, $s1=b, $s2=c
```

```
add $s0, $s1, $s2
```

b y c (source operands) **are read** from internal registers \$s1 and \$s2

a (destination operand) **is written over** register \$s0

The MIPS Register Set

Nombre	Nº Registro	Función
\$0	0	The constant value 0
\$at	1	Assembler temporary
\$v0-\$v1	2-3	Procedure return values
\$a0-\$a3	4-7	Procedure arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Saved variables
\$t8-\$t9	24-25	More temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Procedure return address (<i>link</i>)

Operands: Registers

- Registers in MIPS:
 - ✓ Written with a dollar sign (\$) before their name
 - ✓ For example, register 0 is written “\$0”, pronounced “register zero” or “dollar zero”.
- Certain registers used for specific purposes:
 - ✓ \$0 always holds the constant value 0 (read-only)
 - ✓ the saved registers, \$s0-\$s7, are used to hold variables
 - ✓ the temporary registers, \$t0 - \$t9, are used to hold intermediate values during a larger computation
 - ✓ Three pointers (\$gp, \$sp y \$fp) and the link register, \$ra, are always memory addresses.
- For the moment, we will only use the temporary registers (\$t0 - \$t9) and the saved registers (\$s0 - \$s7). The rest of registers will be used later.

Operands: Memory

- ✓ Too much data to fit in only 32 registers
- ✓ Store more data in memory
 - Memory is large, so it can hold a lot of data
 - But it's also slow (slower than registers)
- ✓ Commonly used variables kept in registers
- ✓ Using a combination of registers and memory, a program can access a large amount of data fairly quickly
- ✓ The architecture details for accessing hierarchic memory systems will be studied in future courses (computer architecture)

Byte-Addressable Memory

- MIPS, as most computers, uses byte-addressable memory (each byte of information in the memory has its own address)
- With a single address and instruction, MIPS can read/write (*load/store*) a 4 bytes *word* (lw/sw) or a single byte (lb/sb).
- Each 32-bit word has an address distant 4 from adjacent words (because each word has 4 bytes).

Word Address	Data								
⋮	⋮								
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

width = 4 bytes

Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4.

(ALIGNED ADDRESS)

Example: Read (*load*) word 1 (in address 4) and save it into \$s3

MIPS assembly code: `lw $s3, 4($0)`

- \$s3 holds the value 0xF2F1AC07 after the instruction completes

Word Address	Data	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

← width = 4 bytes →

Writing Byte-Addressable Memory

Example: Write (*store*) the value held in \$t7 into memory address 0x2C (44_{10})

MIPS assembly code: `sw $t7, 44($0)`

- MEM[44] holds value of \$t7 = 0xA2E5F0C3 after the instruction

Word Address

Data

0000002C

A2	E5	F0	C3
----	----	----	----

Word 11

...

...

00000008

01	EE	28	42
----	----	----	----

Word 2

00000004

F2	F1	AC	07
----	----	----	----

Word 1

00000000

AB	CD	EF	78
----	----	----	----

Word 0

Big-Endian and Little-Endian Memory

How to number bytes within a word?

- ✓ Word address is the same for big- or little-endian
- ✓ **Little-endian:** byte numbers start at the little (least significant) end
- ✓ **Big-endian:** byte numbers start at the big (most significant) end

Big-Endian

Byte Address			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

Little-Endian

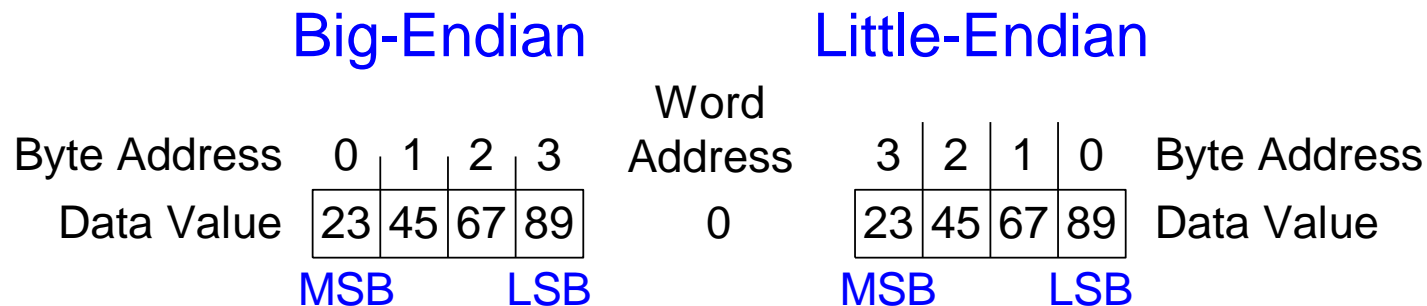
Byte Address			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	

Big-Endian and Little-Endian Memory

Example: Suppose `$t0` initially contains `0x23456789`. Which is the content of `$s0` after the following program is run:

a) In a big-endian system **b)** In a little-endian system

```
sw $t0, 0($0)  # Store $t0 in MEM[0]
lb $s0, 1($0)  # Load MEM[1] byte into $s0
```



a) Big-endian: `$s0 = 0x00000045`

b) Little-endian: `$s0 = 0x00000067`

Operands: Constants/Immediates

- Called *immediates* because they are *immediately* available from the instruction
- Immediates don't require a register or memory access
- The add immediate (`addi`) instruction adds an immediate to a variable (held in a register)
- An immediate is a 16-bit two's complement number

Is subtract immediate (`subi`) necessary?

High-level code (C)

```
a = a + 4;  
b = a - 12;
```

// MIPS assembly code

```
# $s0 = a, $s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```

MIPS Directives

Directives are instructions, but not for the computer. Directives are instructions for the assembler.

- **.text** <address>: Sets the first address of the *text* segment (user code).
- **.data** <address>: Sets the first address of the *data* segment (user data).

Only for the data section:

- **.space** *n*: Allocates *n* bytes of space in the current segment

Example Program: Symbol Table

- ✓ MIPS directives `.data` and `.text` set the addresses for data and instructions.
- ✓ During assembly, a Symbol Table is generated with the address of each literal or symbol of the assembly code.

```
.text 0x0000
main:  lw $t3, A($0)
      add $s1, $t3, $t3
write: sw $s1, B($0)
```

```
.data 0x2000
A: 5
.space 8
B: 0
```

Symbol Table	
Symbol	Address
main	0x00000000
write	0x00000008
A	0x00002000
B	0x0000200C

Outline

- Computer architecture
- Assembler language
- **ISA MIPS. Instruction set**
- Programming

Machine Language

- Computers only understand 1's and 0's
- Machine language: binary representation of instructions
- 32-bit instructions
 - ✓ Again, simplicity favors regularity: 32-bit data and instructions
- Three instruction formats:
 - ✓ R-Type: register operands
 - ✓ I-Type: immediate operand (apart from registers)
 - ✓ J-Type: for jumping

MIPS instruction set (1)

op	Name	Description	Operation
000000	R-Type	R-Type instructions	See next table
000010	j	Jump (unconditional)	PC = JTA
000011	jal	Jump and link	\$ra = PC+4; PC = JTA
000100	beq	Branch if equal (Z = 1)	Si ([rs] == [rt]); PC =BTA
000101	bne	Branch if not equal (Z = 0)	Si ([rs] != [rt]); PC =BTA
001000	addi	Add immediate	[rt] = [rs] + SigImm
001100	andi	AND immediate	[rt] = [rs] & ZeroImm
001101	ori	OR immediate	[rt] = [rs] ZeroImm
001110	xori	XOR immediate	[rt] = [rs] \oplus ZeroImm
001111	lui	Load upper immediate	[rt] _{31:16} = [imm], [rt] _{15:0} = [0...0]
100011	lw	Load word	MEM ([rs]+SigImm) => [rt]
101011	sw	Store word	[rt] => MEM ([rs]+SigImm)

MIPS instruction set (2)

R-Type (sorted by funct):

Funct	Name	Description	Operation
000000	sll	Shift left logical	$[rd] = [rt] \ll \text{shamt}$
000010	srl	Shift right logical	$[rd] = [rt] \gg \text{shamt}$
000011	sra	Shift right arithmetic	$[rd] = [rt] \ggg \text{shamt}$
000100	sllv	Shift left logical variable	$[rd] = [rt] \ll [rs]_{4:0}$
000110	srlv	Shift right logical variable	$[rd] = [rt] \gg [rs]_{4:0}$
000111	srav	Shift right arithmetic variable	$[rd] = [rt] \ggg [rs]_{4:0}$
001000	jr	Jump register	$PC = [rs]$
100000	add	Add	$[rd] = [rs] + [rt]$
100010	sub	Subtract	$[rd] = [rs] - [rt]$
100100	and	AND	$[rd] = [rs] \& [rt]$
100101	or	OR	$[rd] = [rs] \mid [rt]$
100110	xor	XOR	$[rd] = [rs] \oplus [rt]$
100111	nor	NOR	$[rd] = \sim ([rs] \mid [rt])$
101010	slt	Set on less than	$[rs] < [rt] ? [rd]=1 : [rd]=0$

R-Type

Register-type

- Three register operands:
 - **rs, rt**: source registers (5 bits address)
 - **rd**: destination register (5 bits address)
- Other fields:
 - **op**: the *operation code* or *opcode* (6 bits) 0 for R-type instructions
 - **funct**: *function* (6 bits). Together, the *opcode* and *function* tell the computer what operation to perform
 - **shamt**: the *shift amount* (5 bits), shifting movement is shift operations, otherwise it's 0

R-Type



R-Type Examples

	Assembly code	Field values (decimal)					
		op	rs	rt	rd	shamt	funct
1	add \$s0, \$s1, \$s2	0	17	18	16	0	32
2	sub \$t0, \$t3, \$t5	0	11	13	8	0	34
	Machine code (HEX)	Machine code (binary)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x02328020	000000	10001	10010	10000	00000	100000
2	0x016D4022	000000	01011	01101	01000	00000	100010

Note the order of registers in the assembly code (add rd, rs, rt) and in the machine code are different

I-Type

Immediate-type

- 3 operands, 2 in registers and 1 as immediate:
 - **rs**: source operand (5 bits).
 - **rt**: destination register (5 bits). **Note** **rt** is destination, not source.
 - **imm**: immediate (16 bits).
- Other fields:
 - **op**: *opcode* (6 bits). Operation is completely determined by the opcode (no *funct* code).

I-Type



I-Type Examples

	Assembly code	Field values (decimal)			
		op	rs	rt	imm
1	addi \$s0, \$s1, 5	8	17	16	5
2	addi \$t0, \$s3, -12	8	19	8	-12
3	lw \$t2, 32(\$0)	35	0	10	32
4	sw \$s1, 4(\$t1)	43	9	17	4
	Machine code (HEX)	Machine code (binary)			
		op (6)	rs (5)	rt (5)	imm (16)
1	0x22300005	001000	10001	10000	0000000000000101
2	0x2268FFF4	001000	10011	01000	1111111111110100
3	0x8C0A0020	100011	00000	01010	0000000000100000
4	0xAD310004	101011	01001	10001	0000000000000100

Note the differing order of registers in the assembly and machine codes:

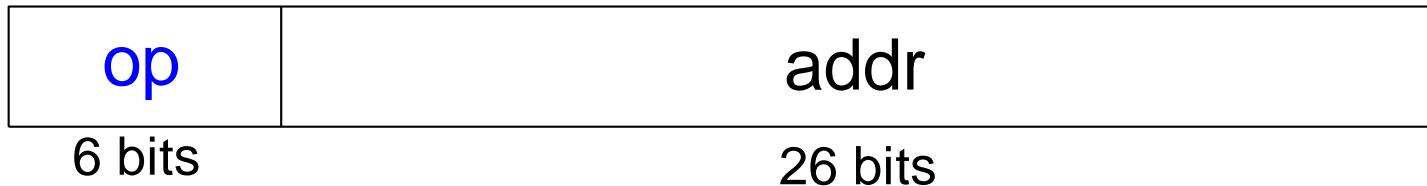
```
addi rt,rs,imm; lw rt,imm(rs); sw rt,imm(rs)
```

J-Type

Jump-type

- Only 2 fields:
 - **op:** *opcode* (6 bits). Identifies the instruction
 - **addr:** address operand (26 bits)

J-Type



Examples to be seen later.

Review: Instruction Formats

R-Type

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

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

Other (new) architecture: RISC-V

- ✓ RISC-V architecture was designed starting in 2010 in the UC Berkeley⁽¹⁾ as *open source*⁽²⁾ for both commercial and academic purposes.

Format of instructions for ISA RV32I

31		25	24		20	19		15	14		12	11		7	6		0	
Funct7				rs2			rs1			funct3		rd			opcode			R-type
imm[11:0]							rs1			funct3		rd			opcode			I-type
imm [11:5]				rs2			rs1			funct3		imm[4:0]			opcode			S-type
imm[12]	imm[10:5]			rs2			rs1			funct3		imm[4:1]	imm[11]	opcode			SB-type	
imm[31:12]												rd			opcode			U-type
imm[20]	imm[10:1]				imm[11]	imm[19:12]					rd			opcode			UJ-type	

⁽¹⁾A.S. Waterman. PhD Thesis. "Design of the RISC-V Instruction Set Architecture". UC Berkeley. 2016

⁽²⁾ <https://riscv.org/>

The Power of the Stored Program

- 32-bit instructions and data stored in memory
- Sequence of instructions: only difference between two applications (for example, a text editor and a video game)
- To run a new program:
 - No rewiring required
 - Simply store new program in memory
- The processor hardware executes the program:
 - *Fetches* (reads) the instructions from memory in sequence
 - Performs the specified operation
- The program counter (PC) keeps track of the current instruction
- In MIPS, programs typically start at memory address 0x00400000
 - **MARS mode “Compact, Text at Address 0” starts in 0x00000000**

The Stored Program

Assembly Code

lw \$t2, 32(\$0)

add \$s0, \$s1, \$s2

addi \$t0, \$s3, -12

sub \$t0, \$t3, \$t5

Machine Code

0x8C0A0020

0x02328020

0x2268FFF4

0x016D4022

Stored Program

Address

...

0040000C

00400008

00400004

00400000

...

Instructions

01	6D	40	22
22	68	FF	F4
02	32	80	20
8C	0A	00	20

← PC

← PC

← PC

← PC

Interpreting Machine Language Code

- Start with opcode (*opcode*, *op*)
- Opcode tells how to parse the remaining bits
 - If *op* = “000000”
 - R-type instruction
 - Function (*funct*) bits tell what instruction it is
 - If *op* ≠ “000000”
 - *opcode* tells what instruction it is

Machine Code

(0x2237FFF1)

op	rs	rt	imm
001000	10001	10111	1111 1111 1111 0001
2	2	3	7 F F F 1

Field Values

op	rs	rt	imm
8	17	23	-15

Assembly Code

addi \$s7, \$s1, -15

(0x02F34022)

op	rs	rt	rd	shamt	funct
000000	10111	10011	01000	00000	100010
0	2	F	3	4	0 2 2

op	rs	rt	rd	shamt	funct
0	23	19	8	0	34

sub \$t0, \$s7, \$s3

Logical Instructions

`and, or, xor, nor, andi, ori, xori`

- `and, andi`: useful for *masking* bits
- `or, ori`: useful for combining bit fields
- `xor, xori`: useful for comparing bits
- `nor`: useful for inverting bits ($A \text{ nor } \$0 == \text{not } A$)
- **NOTE:** in logic instructions only, 16-bit immediate is zero-extended , **not sign-extended.**
- `nori` not needed

Logical Instruction Examples

	Assembly code	Field values (decimal)					
		op	rs	rt	rd	shamt	funct
1	and \$s3, \$s1, \$s2	0	17	18	19	0	36
2	or \$s4, \$s1, \$s2	0	17	18	20	0	37
3	xor \$s5, \$s1, \$s2	0	17	18	21	0	38
4	nor \$s6, \$s1, \$s2	0	17	18	22	0	39
	Machine code (HEX)	Machine code (binary)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x02329024	0000 00	10 001	1 0010	1001 1	000 00	10 0100
2	0x0232A025	0000 00	10 001	1 0010	1010 0	000 00	10 0101
3	0x0232A826	0000 00	10 001	1 0010	1010 1	000 00	10 0110
4	0x0232B027	0000 00	10 001	1 0010	1011 0	000 00	10 0111

Logical Instruction Examples

	Assembly code	Field values (decimal)			
		op	rs	rt	imm
1	andi \$s2, \$s1, 0xFA34	12	17	18	64052
2	ori \$s3, \$s1, 0xFA34	13	17	19	64052
3	xori \$s4, \$s1, 0xFA34	14	17	20	64052
	Machine code (HEX)	Machine code (binary)			
		op (6)	rs (5)	rt (5)	Imm (16)
1	0x3232FA34	0011 00	10 001	1 0010	1111 1010 0011 0100
2	0x3633FA34	0011 01	10 001	1 0011	1111 1010 0011 0100
3	0x3A34FA34	0011 10	10 001	1 0100	1111 1010 0011 0100

Logical Instruction Examples

Source operands

\$s1	1111	1111	1111	1111	0000	0011	1100	0011
\$s2	0100	0110	1010	0001	1111	0000	1011	0111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Instruction	Result									
and \$s3, \$s1, \$s2	\$s3:	0100	0110	1010	0001	0000	0000	1000	0011	
or \$s4, \$s1, \$s2	\$s4:	1111	1111	1111	1111	1111	0011	1111	0111	
xor \$s5, \$s1, \$s2	\$s5:	1011	1001	0101	1110	1111	0011	0111	0100	
nor \$s6, \$s1, \$s2	\$s6:	0000	0000	0000	0000	0000	1100	0100	1000	
andi \$s2, \$s1, 0xFA34	\$s2:	0000	0000	0000	0000	0000	0010	0000	0000	
ori \$s3, \$s1, 0xFA34	\$s3:	1111	1111	1111	1111	1111	1011	1111	0111	
xori \$s4, \$s1, 0xFA34	\$s4:	1111	1111	1111	1111	1111	1001	1111	0111	

Shift Instructions

sll, srl, sra, sllv, srlv, srav

- **sll**: *shift left logical*
 - Example: `sll $t0, $t1, 5` # `$t0 <= $t1 << 5`
- **srl**: *shift right logical*
 - Example: `srl $t0, $t1, 5` # `$t0 <= $t1 >> 5`
- **sra**: *shift right arithmetic*
 - Example: `sra $t0, $t1, 5` # `$t0 <= $t1 >>> 5`
- **sllv**: *shift left logical variable*
 - Example: `sllv $t0, $t1, $t2` # `$t0 <= $t1 << $t2`
- **srlv**: *shift right logical variable*
 - Example: `srlv $t0, $t1, $t2` # `$t0 <= $t1 >> $t2`
- **srav**: *shift right arithmetic variable*
 - Example: `srav $t0, $t1, $t2` # `$t0 <= $t1 >>> $t2`

Shift Instructions Examples

	Assembly code	Field values (decimal)					
		op	rs	rt	rd	shamt	funct
1	sll \$t0, \$t1, 5	0	0	9 (\$t1)	8 (\$t0)	5	0
2	srl \$s2, \$s1, 12	0	0	17 (\$s1)	18 (\$s2)	12	2
3	sra \$s3, \$s1, 4	0	0	17 (\$s1)	19 (\$s3)	4	3
4	sllv \$t0, \$t1, \$t3	0	11 (\$t3)	9 (\$t1)	8 (\$t0)	0	4
	Machine code (HEX)	Machine code (binary)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x00094140	00 0000	00 000	0 1001	0100 0	001 01	00 0000
2	0x00119302	00 0000	00 000	1 0001	1001 0	011 00	00 0010
3	0x00119903	00 0000	00 000	1 0001	1001 1	001 00	00 0011
4	0x01694004	00 0000	01 011	0 1001	0100 0	000 00	00 0100

Note the different order in assembly and machine code (sllv rd, rt, rs)

Shift Instructions Examples

Source operands								
\$s1	1111	0000	1111	1111	1111	0000	0000	0000
\$t3	0000	0000	0000	0000	0000	0000	0000	1000
\$t1	0000	0000	0000	0000	0000	0000	1111	1111

Instruction	Results									
sll \$t0, \$s1, 5	\$t0:	0001	1111	1111	1110	0000	0000	0000	0000	0000
srl \$s3, \$s1, 12	\$s3:	0000	0000	0000	1111	0000	1111	1111	1111	1111
sra \$s5, \$s1, 4	\$s5:	1111	1111	0000	1111	1111	1111	0000	0000	0000
sllv \$t0, \$t1, \$t3	\$t0:	0000	0000	0000	0000	1111	1111	0000	0000	0000

In variable shifts, we use the **5 lsb** of the second source operand

Generating Constants

- 16-bit constants using `addi`:

High-level code	//	MIPS assembly code
<code>//int is 32-bit signed word</code>		<code># \$s0 = a</code>
<code>int a = 0x4F3C;</code>		<code>addi \$s0,\$0,0x4F3C</code>

- 32-bit constants using load upper immediate (`lui`) and `ori`:
(`lui` loads the 16-bit immediate into the upper half of the register and sets the lower half to 0)

High-level code	//	MIPS assembly code
<code>//int is 32-bit signed word</code>		<code># \$s0 = a</code>
<code>int a = 0xFEDC8765;</code>		<code>lui \$s0, 0xFEDC</code>
		<code>ori \$s0, \$s0, 0x8765</code>

Branching

- Instructions that allow a program to execute instructions out of sequence

Types of branches:

✓ Conditional branches

Changes the address if a condition is met

- `beq rs, rt, target;` (*branch if equal*) if `[rs] = [rt]`
- `bne rs, rt, target;` (*branch if not equal*) if `[rs] ≠ [rt]`

“DECISION TAKEN IN EXECUTION TIME”

✓ Unconditional branches

Changes the address with no condition

- `j target,` (*jump*)
- `jal target,` (*jump and link*). Current address (+4) is saved in `$ra`.
- `jr,` (*jump register*)

Conditional Branching (beq)

# Address	# Assembly code	
0x00400000	addi \$s0, \$0, 4	# \$s0 = 0 + 4 = 4
0x00400004	addi \$s1, \$0, 1	# \$s1 = 0 + 1 = 1
0x00400008	sll \$s1, \$s1, 2	# \$s1 = 1 << 2 = 4
0x0040000C	beq \$s0, \$s1, target	# branch if \$s0 = \$s1
0x00400010	addi \$s1, \$s1, 1	# if branch, not executed
0x00400014	sub \$s1, \$s1, \$s0	# if branch, not executed
.	
0x004000FC	target:	# if branch, executed
	add \$s1, \$s1, \$s0	# \$s1 = 4 + 4 = 8

The **label** “target” indicates an instruction location in a program (**branch taken**).

For a **label**, you cannot use reserved words and must be followed by a colon (:).

The Branch Not Equal (bne)

# Address	# Assembly code	
0x00400000	addi \$s0, \$0, 4	# \$s0 = 0 + 4 = 4
0x00400004	addi \$s1, \$0, 1	# \$s1 = 0 + 1 = 1
0x00400008	sll \$s1, \$s1, 2	# \$s1 = 1 << 2 = 4
0x0040000C	bne \$s0, \$s1, target	# branch if \$s0 ≠ \$s1
0x00400010	addi \$s1, \$s1, 1	# \$s1 = 4 + 1 = 5
0x00400014	sub \$s1, \$s1, \$s0	# \$s1 = 5 - 4 = 1
.	
0x004000FC	target:	
	add \$s1, \$s1, \$s0	# not executed

If a branch is not taken, the program goes on in the normal sequential order

Conditional Branching Examples

Assembly code		Field values (decimal)			
		op	rs	rt	imm (BTA, Branch Target)
1	beq \$s0, \$s1, target	4	16 (\$s0)	17 (\$s1)	59
2	bne \$s0, \$s1, target	5	16 (\$s0)	17 (\$s1)	59
Machine code (HEX)		Machine code (binary)			
		op (6)	rs (5)	rt (5)	imm (16)
1	0x1211003B	000100	10000	10001	0000000000111011
2	0x1611003B	000101	10000	10001	0000000000111011

Conditional branches use the I-Type format

The immediate has the number of words (bytes/4) to *target*. It is signed (positive for forward branches, negative for backwards).

If a branch is taken: $PC_{new} = BTA = PC_{old} + 4 + (SignImm \ll 2)$

$\overbrace{\hspace{10em}}$
 Branch Target Address

Conditional Branching: Codification

# Address	# Assembly code	
0x00400000	addi \$s0, \$0, 4	# \$s0 = 0 + 4 = 4
0x00400004	addi \$s1, \$0, 1	# \$s1 = 0 + 1 = 1
0x00400008	sll \$s1, \$s1, 2	# \$s1 = 1 << 2 = 4
0x0040000C	bne \$s0, \$s1, target	# branches if \$s0 ≠ \$s1
0x00400010	addi \$s1, \$s1, 1	# \$s1 = 4 + 1 = 5
0x00400014	sub \$s1, \$s1, \$s0	# \$s1 = 5 - 4 = 1
.	
0x004000FC	target: add \$s1, \$s1, \$s0	# not executed

$$\text{BTA} = \text{PC} + 4 + (\text{SignImm} \ll 2)$$

$$\text{SignImm} = (\text{BTA} - \text{PC} - 4) \gg 2$$

$$\text{SignImm} = (0x004000FC - 0x0040000C - 4) \gg 2$$

$$\text{SignImm} = (0x000000F0 - 4) \gg 2$$

$$\text{SignImm} = (0x000000EC) \gg 2$$

$$\text{SignImm} = 0x0000003B = 59$$

	op	rs	rt	imm (BTA, Branch Target)
beq \$s0, \$s1, target	4	16 (\$s0)	17 (\$s1)	59

Conditional Branching: Codification

# Address	#Machine code	# Assembly code
0x00400000	0x20100004	addi \$s0, \$0, 4
0x00400004	0x20110001	addi \$s1, \$0, 1
0x00400008	0x16110001	bne \$s0, \$s1, target
0x0040000C	0x20110005	addi \$s1, \$0, 5
0x00400010	0x22310001	target: addi \$s1, \$s1, 1
0x00400014	0x02308822	sub \$s1, \$s1, \$s0

Where is *target*?

bne \$s0, \$s1, target => 0x16110001 =>
000101 10000 10001 000000000000000001
OPCODE RS RT IMM (=1)

BTA = PC + 4 + (SignImm << 2)

BTA = 0x00400008 + 4 + (0x0001 << 2)

BTA = 0x0040000C + (0x0004)

BTA = 0x00400010

Intuitive way:
The immediate has the
number of instructions
to jump from the next
one to the *branch*.

Conditional Branching: Codification

# Dirección	#Código máquina	# Ensamblador de MIPS
0x00400000	0x20100004 target:	addi \$s0, \$0, 4
0x00400004	0x20110001	addi \$s1, \$0, 1
0x00400008	0x1611FFFD	bne \$s0, \$s1, target
0x0040000C	0x20110005	addi \$s1, \$0, 5
0x00400010	0x22310001	addi \$s1, \$s1, 1
0x00400014	0x02308822	sub \$s1, \$s1, \$s0

Where is *target*?

bne \$s0, \$s1, target => 0x1611FFFD =>
000101 10000 10001 11111111111111101
OPCODE RS RT IMM (=-3)

BTA = PC + 4 + (SignImm << 2)

BTA = 0x00400008 + 4 + (0xFFFD << 2)

BTA = 0x0040000C + (0xFFF4)

BTA = 0x00400000

Intuitive way:
The immediate has the
number of instructions
to jump from the next
one to the *branch*.

Unconditional Jumping (j and jal)

# Address	# Assembly code	
0x00002000	addi \$s0, \$0, 4	# \$s0 = 4
0x00002004	addi \$s1, \$0, 1	# \$s1 = 1
0x00002008	j target	# jumps (always) to target
0x0000200C	sra \$s1, \$s1, 2	# not executed
0x00002010	addi \$s1, \$s1, 1	# not executed
.	
0x00004000	target:	
	add \$s1, \$s1, \$s0	# \$s1 = 1 + 4 = 5
0xF0002008	jal function	# jumps (always) to function
		# \$ra = 0xF000200C
0xF000200C	sra \$s1, \$s1, 2	# executed after return
.	
0xF0004000	function:	
	add \$s1, \$s1, \$s0	# \$s1 = 1 + 4 = 5
.	
0xF000401C	jr \$ra	# returns to caller

Unconditional Jumping: Codification

	Assembly code	Field values (decimal)	
		op	addr
1	j target	2	4096
2	jal funcion	3	4096
	Machine code (HEX)	Machine code (binary)	
		op (6)	addr (26)
1	0x08001000	0000 10	00 0000 0000 0001 0000 0000 0000
2	0x0C001000	0000 11	00 0000 0000 0001 0000 0000 0000

jump (j) and *jump and link* (jal) use the J-Type format

Apart from jumping, jal saves $\$ra = PC+4$ for the return address (*link*)

Jump address: $PC_{new} = JTA = (PC_{old}+4) [31:28] \& \text{addr} \& \text{"00"}$

Jump Target Address

Unconditional Jumping: Codification

# Address	# Assembly code
0x00002000	addi \$s0, \$0, 4 # \$s0 = 4
0x00002004	addi \$s1, \$0, 1 # \$s1 = 1
0x00002008	j target # jumps (always) to target
0x0000200C	sra \$s1, \$s1, 2 # not executed
0x00002010	addi \$s1, \$s1, 1 # not executed
...	...
0x00004000	target: add \$s1, \$s1, \$s0 # \$s1 = 1 + 4 = 5

JTA = { (PC+4)[31:28], addr, "00" }

0x00004000 = (0x00002008+4)[31:28], addr, "00"

0x00004000 = (0x0000200C)[31:28], addr, "00"

0x00004000 = "0000", addr, "00"

0000 00000000000000000000100000000000 00 = 0000 addr 00

addr = 00000000000000000000100000000000 = 2^{12} = 4096

Unconditional Jumping: Codification

# Address	# Machine code	# Assembly code
0x00400000	0x20100004	addi \$s0, \$0, 4
0x00400004	0x20110001	addi \$s1, \$0, 1
0x00400008	0x08100005	j target
0x0040000C	0x00118883	sra \$s1, \$s1, 2
0x00400010	0x22310001	addi \$s1, \$s1, 1
0x00400014	0x02308820	add \$s1, \$s1, \$s0

target:

Where is *target*?

j target => 0x08100005 =>
 000010 00000100000000000000000000101
 OPCODE addr

JTA = { (PC+4) [31:28], addr, "00" }

JTA = { (0x00400008+4) [31:28], 00000100000000000000000000101, 00 }

JTA = { (0x0040000C) [31:28], 00000100000000000000000000101, 00 }

JTA = { 0000, 00000100000000000000000000101, 00 }

JTA = 00000000010000000000000000000010100 = 0x00400014

Unconditional Jumping (jr)

	Assembly code	Field values (decimal)					
		op	rs	rt	rd	shamt	funct
1	jr \$s0	0	16	0	0	0	8
	Machine code (HEX)	Machine code (binary)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x02000004	0000 00	10 000	0 0000	0000 0	000 00	00 1000

jump register (jr) uses the R-Type format (but only uses one register)

rs content is directly the address to jump to (*target*):

PC = [rs]

┌
Content of register rs

Unconditional Jumping (jr)

# Address	# Assembly code
0x00002000	addi \$s0, \$0, 0x2010
0x00002004	jr \$s0
0x00002008	addi \$s1, \$0, 1
0x0000200C	sra \$s1, \$s1, 2
0x00002010	lw \$s3, 44(\$s1)

Always jumps (*branch taken*) to the address pointed by the register.

Which instruction is executed after jr?

lw \$s3, 44(\$s1)

Addressing modes

Addressing modes:

Five used in MIPS. The first three for getting an operand:

- ✓ **Register-Only Addressing.** Operand is the content of a register
 - Examples: `add $s0, $t2, $t3` // `sub $t8, $s1, $0`
- ✓ **Immediate Addressing.** An operand is an immediate.
 - Examples: `addi $s4, $t5, -73` // `ori $t3, $t7, 0xFF`
- ✓ **Base Addressing.** The effective address of the memory operand is found by adding the base address in a register to the sign-extended 16-bit offset of the immediate field.
 - Examples: `lw $s4, 72($0)` Address = $\$0 + 72$
`sw $t2, -25($t1)` Address = $\$t1 - 25$

Addressing modes

The last two modes are used for writing the PC

- ✓ **PC-relative Addressing**. The signed offset in the immediate field is added to (PC+4) to obtain the new PC.

➤ **Example:** conditional branches.

# Address	# Assembly code
0x10	beq \$t0, \$0, else # Jumps to “else”
0x14	addi \$v0, \$0, 1
0x18	addi \$sp, \$sp, i
0x1C	else: addi \$a0, \$a0, -1
0x20	jal factorial

Assembly code	Field values (decimal)			
	op	rs	rt	imm
beq \$t0, \$0, else	4	8	0	2

See conditional branches for details

Addressing modes

- ✓ **Pseudo-Direct Addressing.** The effective address is (almost) specified in the instruction.

➤ Example: `jal` and `j`.

# Address	# Assembly code
0x0040005C	<code>jal sum</code> # calls “sum”
...	. . .
0x004000A0	<code>sum: addi \$a0, \$a0, -1</code>

Assembly code	Field values (decimal, hexadecimal, binary)	
	op	addr
jal sum	3	1048616
0x0C100028	0000 11	00 0001 0000 0000 0000 0010 1000

See unconditional branches for details

Example Program: Assembly Code

High-level code (C)

```
int f, g, y; // global
int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);
    return y;
}

int sum(int a, int b) {
    return (a + b);
}
```

Symbol table	
Symbol	Address
f	0x00002000
g	0x00002004
y	0x00002008
main	0x00000000
sum	0x0000002C

// MIPS assembly code

```
.data
f: 0x00
g: 0x00
y: 0x00
.text
main:
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)    # store $ra
    addi $a0, $0, 2      # $a0 = 2
    sw   $a0, f          # f = 2
    addi $a1, $0, 3      # $a1 = 3
    sw   $a1, g          # g = 3
    jal  sum             # call sum
    sw   $v0, y          # y = sum()
    lw   $ra, 0($sp)     # restore $ra
    addi $sp, $sp, 4     # restore $sp
    jr   $ra             # return to OS

sum:
    add  $v0, $a0, $a1   # $v0 = a + b
    jr   $ra             # return
```

Example Program: Executable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	Address	Instruction
	0x00000000	0x23BDFFFC
	0x00000004	0xAFBF0000
	0x00000008	0x20040002
	0x0000000C	0xAC042000
	0x00000010	0x20050003
	0x00000014	0xAC052004
	0x00000018	0x0C00000B
	0x0000001C	0xAC022008
	0x00000020	0x8FBF0000
	0x00000024	0x23BD0004
	0x00000028	0x03E00008
	0x0000002C	0x00851020
	0x00000030	0x03E00008
Data segment	Address	Data
	0x00002000	0x00000000
	0x00002004	0x00000000
	0x00002008	0x00000000

```

addi $sp, $sp, 0xFFFC
sw   $ra, 0x0000($sp)
addi $a0, $0, 0x0002
sw   $a0, 0x2000($0)
addi $a1, $0, 0x0003
sw   $a1, 0x2004($0)
jal  0x0000002C
sw   $v0, 0x2008($0)
lw   $ra, 0x0000($sp)
addi $sp, $sp, 0x0004
jr   $ra
add  $v0, $a0, $a1
jr   $ra

```

The MIPS Memory Map

Memory size?

✓ The address bus is 32 bits, so the maximum memory is $2^{32} = 4$ gigabytes (4 GB)

✓ Memory map range:
0x00000000 to 0xFFFFFFFF

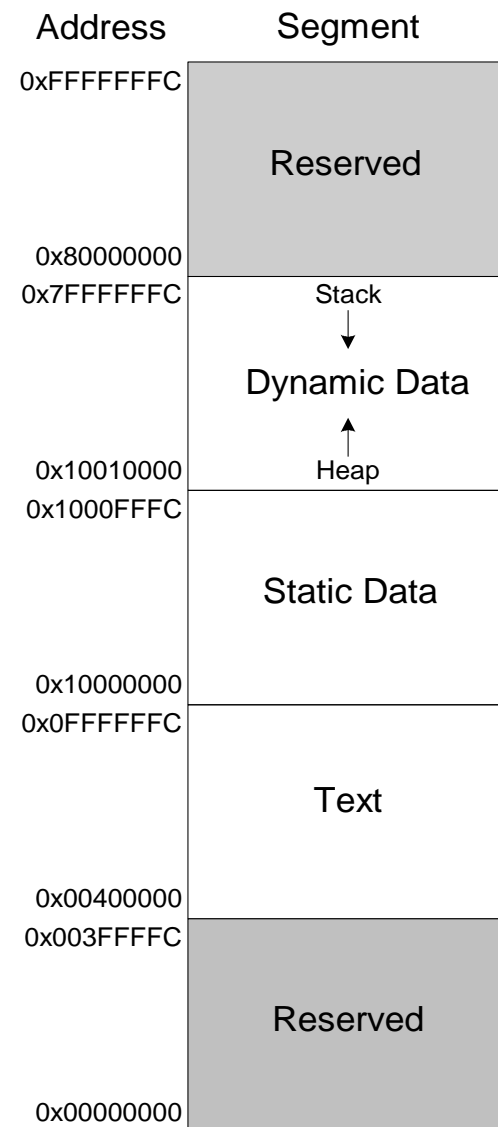
In the simulator:

where is the code and the data?

- ✓ Set by directives **.text** y **.data** but in the range defined by:
- By default: text (0x00400000) and data (0x10010000)
 - **Text Mode 0: text (0x00000000) and data (0x00002000)**
 - Data Mode 0: text (0x00003000) and data (0x00000000)

Types of data:

- ✓ Global/Static. Allocated before program begins
- ✓ Dynamic. Allocated during execution



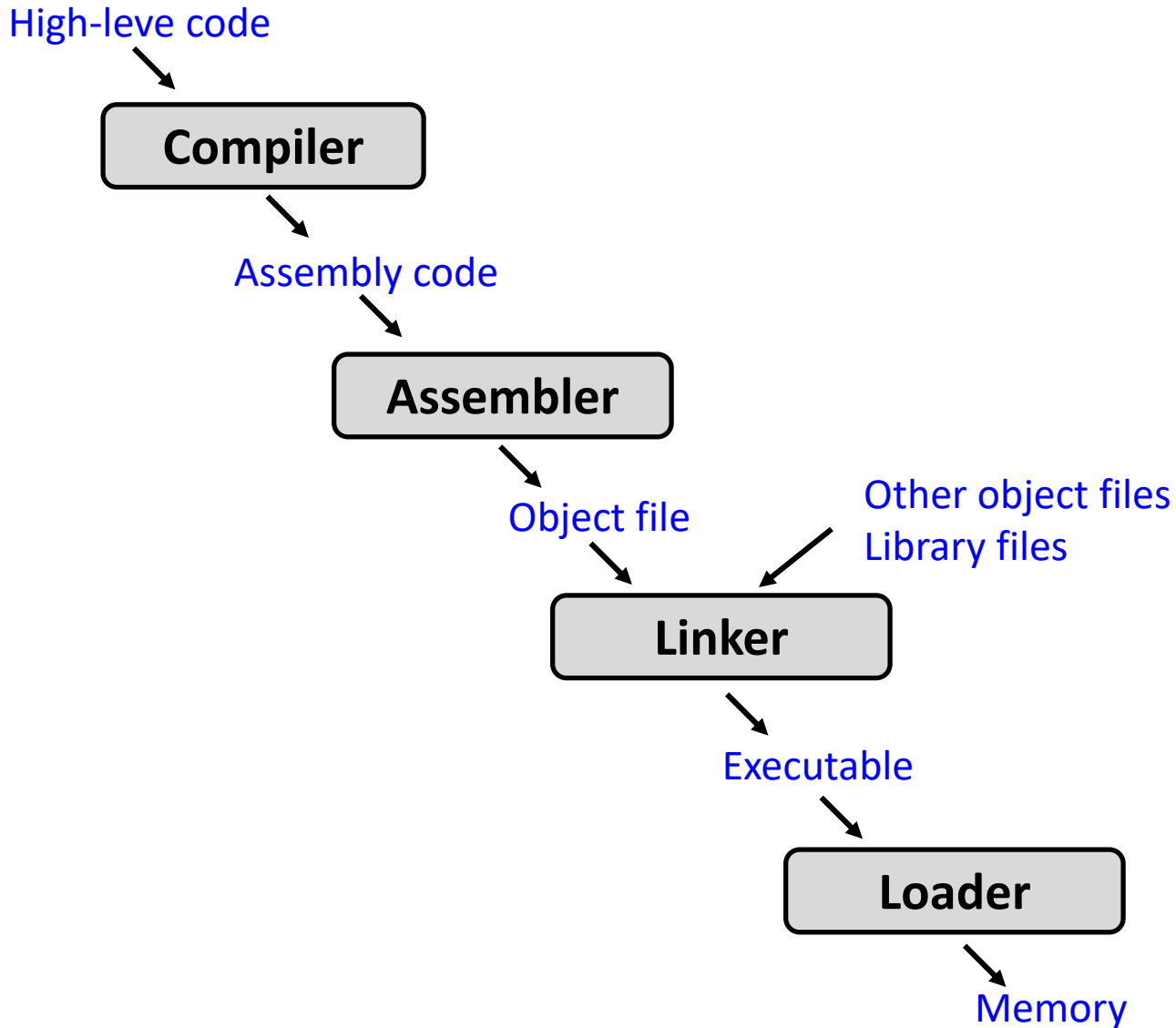
Example Program: In Memory

MARS Simulator. Mode Compact Text 0

- Code Memory (text): (0x00000000:0x00001FFC)
- Data Memory (data): (0x00002000:0x00003FFF)
 - Dynamic data: (0x00003FFC:0x00002000)

Dirección	Memoria	
	Reservado	
0x00003FFC	Pila (stack) ↓ . . .	← \$sp = 0x00003FFC
	.	
	0x00000000	
	0x00000000	
0x00002000	0x00000000	
	.	
	.	
	.	
	0x03E00008	
	0x00851020	
	0x03E00008	
	0x23BD0004	
	0x8FBF0000	
	0xAC022008	
	0x0C00000B	
	0xAC052004	
	0x20050003	
	0xAC042000	
	0x20040002	
	0xAFBF0000	
0x00000000	0x23BDFFFC	← PC = 0x00000000

How do we compile & run an application?



Outline

- Computer architecture
- Assembler language
- ISA MIPS. Instruction set
- **Programming**

Programming in MIPS

- Using the three type of instructions (arithmetic-logic, branches and memory) we can do any code for any functionality.
- High level languages (C, Java, Python, etc) are translated (compiled) into assembly code.
- In those high level languages, higher level of abstraction is used.
- We will go into:
 - procedure calls
 - if/else statements
 - for loops
 - while loops
 - working with arrays

Procedure Calls

Operations to do in a procedure call:

- Calling procedure (*caller*):
 - ✓ **Passes arguments** to secondary procedure (*callee*). Using specific registers ($\$a0-\$a3$) or using the *stack*.
 - ✓ **Jumps** to the callee. Using the `jal` (*jump and link*) instruction for keeping the return address (*link*) in $\$ra$.
- Called procedure (*callee*):
 - ✓ **Performs the procedure.**
 - ✓ **Returns the result** to caller. Using specific registers ($\$v0-\$v1$) or using the *stack*.
 - ✓ **Returns** to the point of call. Using the instruction `jr $ra`, (*jump register*).
 - ✓ **NOTE:** the callee must not overwrite registers or memory needed by the caller.

Procedure Calls

High-level code

//

MIPS assembly code

```
int main() {  
    simple();  
    a = b + c;  
}
```

Dirección

0x00400200
0x00400204
...

```
main:  jal  simple  
       add  $s0, $s1, $s2  
...
```

```
void simple() {  
    return;  
}
```

...

0x00401020 simple: jr \$ra

void means that simple doesn't return a value

`jal simple:` jumps to `simple` and stores PC+4 (0x00400204) in the link register (`$ra = 0x00400204`)

`jr $ra:` jumps back to the link address stored in `$ra`

Input Arguments and Return Values

High-level code

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5);    // 4 input arguments
                                   // 2=f, 3=g, 4=h, 5=i)
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;                // return value(y)
}
```

Arguments and returned values

MIPS assembly code

```
# $s0 = y
main:
    ...
    addi $a0, $0, 2      # argument $a0 = 2
    addi $a1, $0, 3      # argument $a1 = 3
    addi $a2, $0, 4      # argument $a2 = 4
    addi $a3, $0, 5      # argument $a3 = 5
    jal  diffofsums      # call procedure. $ra stores the return address
    add  $s0, $v0, $0     # y = returned value
    ...
# $s0 = result
diffofsums:
    add $t0, $a0, $a1     # $t0 = f + g
    add $t1, $a2, $a3     # $t1 = h + i
    sub $s0, $t0, $t1     # result = (f + g) - (h + i)
                          # the callee modifies: $t0, $t1 and $s0
    add $v0, $s0, $0      # put return value in $v0
    jr   $ra              # return to caller
```

The Stack

- Memory used to temporarily save variables.
- Like a stack of dishes, last-in-first-out (LIFO) queue.
- *Expands*: uses more memory when more space is needed.
- *Contracts*: uses less memory when the space is no longer needed.
- The stack is used for multiple purposes, like argument passing between procedures, store copies of registers, store local variables, etc.

The Stack


(MARS Compact text mode Address 0)

					← 0x00...4000	← \$sp	
Data segment (.data)	...1111	...1110	...1101	...1100	← 0x00...3FFC		STACK ↓
	...0011	...0010	...0001	...0000	← 0x00...3000		
	...0011	...0010	...0001	...0000	← 0x00...2000		
Code segment (.text)	...1111	...1110	...1101	...1100	← 0x00...0FFC		↑ MEMORY ADDRESSES
	...0011	...0010	...0001	...0000	← 0x00...0000		

The Stack

- Grows down (from higher to lower memory addresses).
- Stack pointer: $\$sp$, points to top of the stack.
- At the beginning of the user code, $\$sp$ points to the last stored value of the previous code (operating system if any, or the beginning of the stack).

Address	Data
00003FFC	
00003FF8	
00003FF4	
00003FF0	
⋮	⋮



The Stack

- Grows down (from higher to lower memory addresses).
- Stack pointer: `$sp`, points to top of the stack.
- At the beginning of the user code, `$sp` points to the last stored value of the previous code (operating system if any, or the beginning of the stack).

Address	Data
00003FFC	12345678 ← <code>\$sp</code>
00003FF8	
00003FF4	
00003FF0	
⋮	⋮

Address	Data
00003FFC	12345678
00003FF8	AABBCCDD
00003FF4	11223344 ← <code>\$sp</code>
00003FF0	
⋮	⋮

The Stack

After each stack operation (store and recover data), the stack pointer `$sp` must return to the original address.


Address	Data	
00003FFC	12345678	← <code>\$sp</code>
00003FF8	AABBCCDD	
00003FF4	11223344	
00003FF0		
⋮	⋮	

Registers and stack

A procedure should not modify any register.

There is an agreement of which registers can be freely modified.

Preserved <i>Callee-Saved</i>	Non Preserved <i>Caller-Saved</i>
<code>\$s0 - \$s7</code>	<code>\$t0 - \$t9</code>
<code>\$ra</code>	<code>\$a0 - \$a3</code>
<code>\$sp</code>	<code>\$v0 - \$v1</code>
Stack above <code>\$sp</code>	Stack below <code>\$sp</code>



The *callee* must restore them if used.



The *caller* must save them if needed.

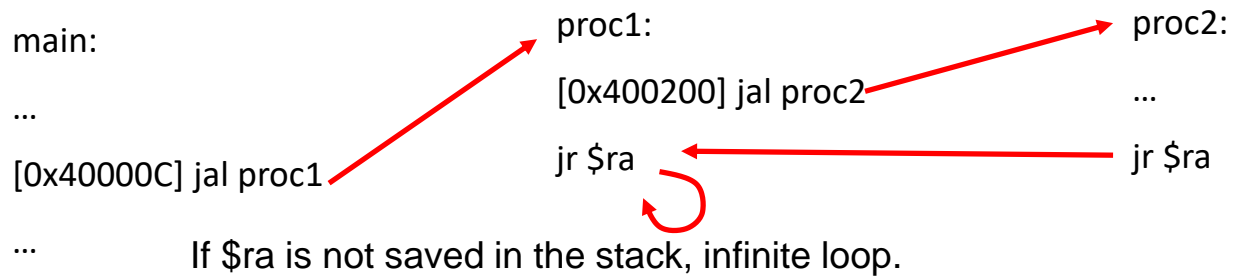
Storing Register Values on the Stack

The stack can be used to save and restore registers

```
# $s0 = result
diffofsums:
    addi $sp, $sp, -4    # make space on stack to store a register
                        # $s0 must be saved ($t0 and $t1 are not preserved)

    sw    $s0, 0($sp)    # save $s0 on stack
    add   $t0, $a0, $a1   # $t0 = f + g
    add   $t1, $a2, $a3   # $t1 = h + i
    sub   $s0, $t0, $t1   # result = (f + g) - (h + i)
    add   $v0, $s0, $0    # put return value in $v0
    lw    $s0, 0($sp)    # restore $s0 from stack
    addi  $sp, $sp, 4     # deallocate stack space
    jr    $ra            # return to caller
```

Multiple Procedure Calls

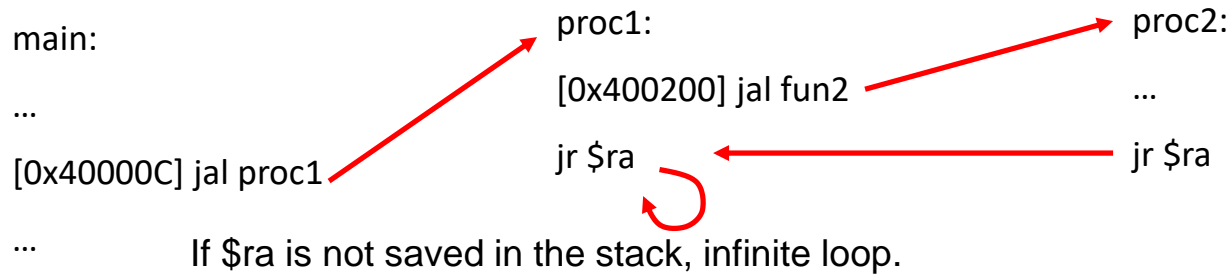


Multiple Procedure Calls

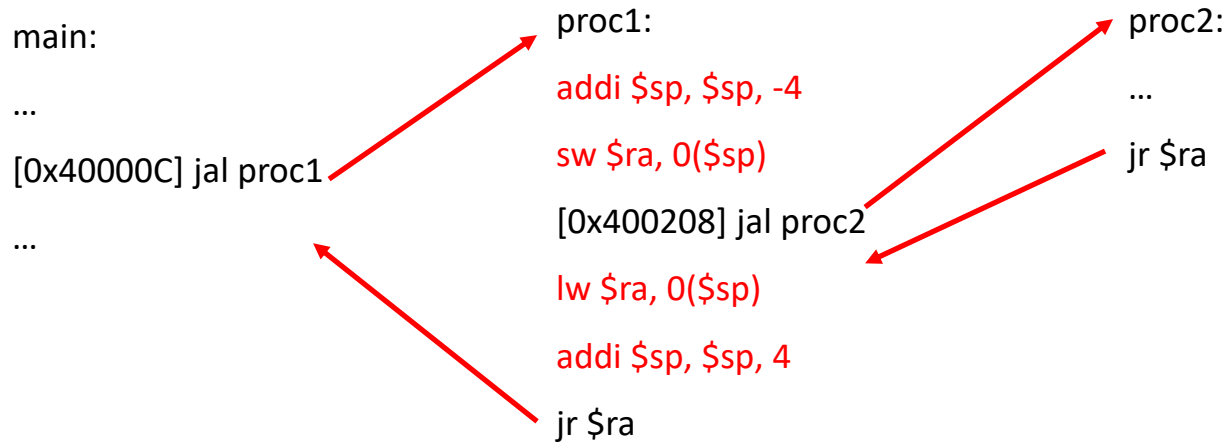
En llamadas a procedimientos múltiples (llamadas anidadas), cada subrutina debe preservar en la pila su dirección de retorno `$ra`, antes de saltar.

```
proc1:                                # start of proc1
    addi $sp, $sp, -4                # make space on stack
    sw   $ra, 0($sp)                 # save $ra on stack
    jal  proc2                       # jump to callee
    ...
    ...
    lw   $ra, 0($sp)                 # after return from proc2, restore $s0 from stack
    addi $sp, $sp, 4                 # deallocate stack space
    jr   $ra                         # return to caller
```


Multiple Procedure Calls



Solution:



Using the Stack


Already seen uses of the stack:

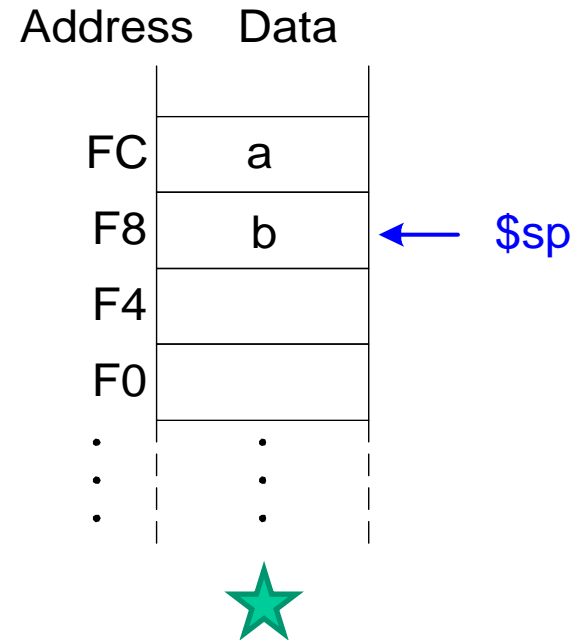
- Preserve registers when calling a procedure.
- Preserve the link address (`$ra`) in multiple procedure calls.

Also used for:

- Store local variables.
- Passing arguments.
- Returning arguments.

Stack: local variables

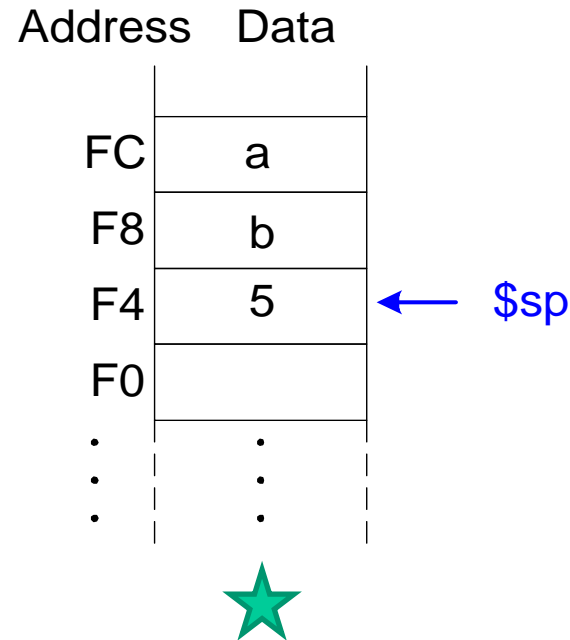
```
int main() {  
    int a, b;   
    ...  
}
```



a and *b* are stored in the stack (local variables). They can be read without changing the stack pointer, i.e. without making *pop*: *lw* <reg>, $\$sp(4)$.

Stack: passing arguments

```
int main() {  
    int a, b;  
    ...  
    funct(5);  
    ...  
}
```



The passed argument is stored without deallocating *a* and *b*, as they may be used later.

Stack: passing arguments

```
int main() {
```

```
  int a, b;
```

```
  ...
```

```
  funct(5);
```

```
  ...
```

```
}
```

Address	Data
FC	a
F8	b
F4	5
F0	
:	:
:	:
:	:

Address	Data
FC	a
F8	b
F4	5
F0	c
:	:
:	:
:	:

Address	Data
FC	a
F8	b
F4	10
F0	c
:	:
:	:
:	:

```
int funct(int x)
```

```
{
```

```
  int c;
```

```
  return x+5;
```

```
}
```



← \$sp

← \$sp

← \$sp

funct must return its output using the stack, and overwrites the received argument. It also deallocates its own local variables, *c*.

If Statement

High-level code

```
if (i == j)
    f = g + h;
f = f - i;
```

//

MIPS assembly code

```
# $s0=f, $s1=g, $s2=h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Note that the assembly tests for the opposite case ($i \neq j$) than the test in the high-level code ($i == j$).

If / Else Statement

High-level code

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

//

MIPS assembly code

```
# $s0=f, $s1=g, $s2=h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2
    j   done
L1:   sub $s0, $s0, $s3
done:
```

Note that the assembly tests for the opposite case ($i \neq j$) than the test in the high-level code ($i == j$).

While Loops

High-level code

```
// determina la potencia
// de x que cumple  $2^x = 128$ 
int pow = 1;
int x = 0;

while (pow != 128)
{
    pow = pow * 2;
    x = x + 1;
}
```

//

MIPS assembly code

```
# $s0 = pow, $s1 = X

addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
while: beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
done:
```

Note that the assembly tests for the opposite case ($\text{pow} = 128$) than the test in the high-level code ($\text{pow} \neq 128$).

For Loops

The general form of a *for* loop is:

```
for (initialization; condition; loop operation)
    loop body
```

- **Initialization:** executes before the loop begins
- **Condition:** is tested at the beginning of each iteration
- **Loop operation:** executes at the end of each iteration
- **Loop body:** executes each time the condition is met

In assembly, five steps:

1. Initialization.
2. Condition.
3. Loop body.
4. Loop operation.
5. Jump to 2.

For Loops

High-level code

```
// add the numbers from
// 0 to 9
int sum = 0 ;
int i ;

for (i=0; i!=10; i = i+1)
{
    sum = sum + i;
}
```

//

MIPS assembly code

```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
1       add  $s0, $0, $0
        addi $t0, $0, 10
2 for:   beq  $s0, $t0, done
        add  $s1, $s1, $s0
3       addi $s0, $s0, 1
4       j    for
5
done:
```

Note that the assembly tests for the opposite case ($i == 10$) than the test in the high-level code ($i != 10$).

Less Than Comparisons

High-level code

//

```
// add the power of 2
// from 1 to 100
int sum = 0 ;
int i ;

for (i=1; i<101; i = i*2)
{
    sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$\$t1 = 1$ if $i < 101$

Arrays

- ✓ Useful for accessing large amounts of similar data
 - ✓ Array element: accessed by *index*
 - ✓ Array *size*: number of elements in the array
- Example: 5-element arrays
 - *Base address*: 0x00002000, address of the first array element, array[0]
 - lw and sw access the memory address after adding an immediate and a register.
 - If the base address fits into 16 bits, it can be the immediate.
 - Then the register is the index: (0, 1, 2,...).

0x00002010	array [4]
0x0000200C	array [3]
0x00002008	array [2]
0x00002004	array [1]
0x00002000	array [0]

Arrays (base address in 16 bits)

High-level code

```
int array [5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;  
:
```

If the base address is
0x00002000 (fits into 16 bits).
Each position occupies 4 bytes

MIPS assembly code

```
# index = $s0, base address as immediate.  
addi $s0, $0, 0          # $s0 = index 0  
lw   $t1, array($s0)     # $t1 = array[0]  
sll  $t1, $t1, 1         # $t1 = $t1 * 2  
sw   $t1, array($s0)     # array[0] = $t1  
addi $s0, $0, 4          # $s0 = index 1 (byte +4)  
lw   $t1, array($s0)     # $t1 = array[1]  
sll  $t1, $t1, 1         # $t1 = $t1 * 2  
sw   $t1, array($s0)     # array[1] = $t1  
:
```

Arrays (base address more than 16 bits)

High-level code

```
int array [5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;  
:
```

If the base address is
0x12348000 (**does not fit into 16 bits**).
Each position occupies 4 bytes

MIPS assembly code

```
# base address = $s0, index as immediate  
lui    $s0, 0x1234          # 0x1234 in upper half of $s0  
ori    $s0, $s0, 0x8000     # 0x8000 in lower half of $s0  
lw     $t1, 0($s0)          # $t1 = array[0]  
sll    $t1, $t1, 1          # $t1 = $t1 * 2  
sw     $t1, 0($s0)          # array[0] = $t1  
lw     $t1, 4($s0)          # $t1 = array[1]  
sll    $t1, $t1, 1          # $t1 = $t1 * 2  
sw     $t1, 4($s0)          # array[1] = $t1  
:
```

Arrays Using For Loops

High-level code (using for loops)

```
int array [1000];  
int i ;  
for (i=0; i < 1000; i = i + 1)  
    array[i] = array[i] * 8;
```

1000 elements, so 1000 executions of the loop.

Each element occupies 4 bytes, so the index increments 4 each time, so the condition must be modified.

Arrays Using For Loops

High-level code (using for loops)

Base address 0x00002000

```
# $s1 = i
# Initialization code
    addi $s1, $0, 0          # i = 0;
    addi $t2, $0, 4000       # $t2 = 4000.
                                # (As i is incremented 4 each
                                # iteration, the condition
                                # 1000 must be multiplied by 4)

loop:
    slt  $t0, $s1, $t2       # i < 4000?
    beq  $t0, $0, done       # if not then done
    lw   $t1, array($s1)     # $t1 = array[i]
    sll  $t1, $t1, 3          # $t1 = array[i] * 8
    sw   $t1, array($s1)     # array[i] = array[i] * 8
    addi $s1, $s1, 4          # i = i + 4
    j    loop                # repeat
done:
```


Unit 3. Processor I: Instruction set and machine code

Escuela Politécnica Superior - UAM

Exercises U3



3.6. The following code has mixed assembly and machine code (hexadecimal).

- Complete the assembly code.
- Which is the functionality of the code? Also write the final value of the memory position labelled F.
- Fill in the Symbol Table.

Símbolo	Dirección
L1	0x00000010
Fin	0x00000020
Parar	0x00000024
Func	0x00000028
L2	0x00000030
L3	0x00000040
N	0x00002000
F	0x00002004

```
# Code
.text 0x0000
    lw $s1, N($0)
    add $s3, $s1, $0
    addi $s2, $s1, -1
    beq $s1, $0, fin
L1:   jal func
      addi $s2, $s2, -1
      beq $s2, $0, fin
      j L1
fin:  sw $s3, F($0)
parar: j parar
func: and $s6, $s6, $0
      add $s7, $s2, $0
L2:   add $s6, $s3, $s6
      addi $s7, $s7, -1
      beq $s7, $0, L3
      j L2
L3:   add $s3, $s6, $s0
      jr $ra
.data 0x2000
N:    0x0003
F:    0x0000
```

3.5. The following MIPS program calls procedure sub1, passing two arguments A and B. Procedure sub1 uses two local variables M and N, and returns a result Z. sub1 also calls to procedure func1 (multiple procedure calls) passing argument P and receiving back argument S. func1 uses one local variable. The arguments are exchanged by stack. The stack pointer is 0x0004000 before starting the program. Using the given templates, please, give the value of \$sp and the values in the stack for each of these points.

- a) Just before executing instruction labelled a.
- b) Just before executing instruction labelled d.
- c) Just before executing instruction labelled func1.
- d) Just before executing instruction labelled f.
- e) Just before executing instruction labelled e.
- f) Write the instructions needed in labels b and c.

```
# Program
a:  jal  sub1
b:  # New instruction 1
c:  # New instruction 2
    # -----
sub1: # Beginning of sub1
    # ----
d:  jal  func1
    # ----
e:  jr   $ra   # return from sub1
    # -----
func1: # Beginning of func1
    # ----
    # ----
f:  jr   $ra   # return from func1
```

3.17. The following is a C code to be compiled for the MIPS architecture. The argument exchange will be implemented using the stack, which is also used for local variables. Please, determine the state of the stack in some points marked in the code. The stack pointer (\$sp) starts at x4000 and the compiler is not making any code optimization regarding the stack. When possible, please also indicate all the values inside the stack. All procedures, main and called ones, store their local variables in the stack.

<pre>void main(){ int aux = 5; aux=fun1(3, 7); (a) fun2(aux); (d) } (f)</pre>	<pre>int fun1(int a, int b){ int k; (b) k = a+b; return k+1; (c) }</pre>	<pre>void fun2(a){ printf("El número es\n",a); return; (e) }</pre>
---	--	--

Exercises U3

3.11. The size of all the instructions of a given microprocessor is 24 bits. This processor has 64 registers in the register file. There are three types of instructions. The number of instructions of each type is: 14 instructions using three registers (two sources and one destination), 47 instructions using two registers (one source and one destination) and 4 instructions using a single register (destination).

Note: Consider that the opcode (OP) does not need to be the same size in each type of instructions.

Justify your answers:

- Necessary fields for each type of instructions. Include the number of bits of each field, such as indicating the type of instruction, the opcode, each register and any other fields.
- How many different addresses can be accessed when using an instruction with a single register? Put an example of a possible instruction, indicating the operation it perform.

3.9. Complete the MIPS program that calculates $P = M * N$, where N is given as $N = 2X - 2Y$. M, X and Y are read from memory. X and Y are positive numbers between 0 and 31.

Note: include small comments for the added instructions.

T

# Programa	Comment
.text 0x0000	
lw \$t1, M(\$0)	#Reads M into \$t1
....
....
sw \$s1, P(\$0)	#writes the result
fin: j fin	
.data 0x2000	
M:	
X:	
Y:	
P:	

Exercises U3

3.19. MIPS program calls a procedure using the specific registers for passing the arguments. We know the callee receives two arguments and returns one argument.

- a) Indicate the instruction/s in the main procedure to store the received solution in the memory position 0x2008, and its machine code in hexadecimal.
- b) Indicate the instruction/s to store in \$t4 the constant 0x12345678.
- c) Indicate the instruction/s to read from memory the value stored in the memory position labelled A and write its two's complement in the following memory position.

3.18. Two signed numbers are kept in the memory positions labelled A and B. The objective of the program to be written is to store in the same positions the addition and the average of both numbers. In order to do so, there must be a procedure called SumMed that receives both signed numbers as input parameters and that returns the addition and average also as parameters.

- a. Use the stack for exchanging parameters between procedures. Write both the code of the caller and callee.
- b. Use the specific registers for sending arguments to procedures (\$a0-\$a3) and for obtaining return values (\$v0-\$v1). Write both the code of the caller and callee.

Note: For the maximum qualification, the number of instructions should be kept to the minimum, and include the appropriate comments.

3.16. The following code calculates the arithmetic average (rounded down) of the numbers in the memory positions *A*, *B*, *C* and *D*. The result is written in *R*. The main procedure, *main*, calls to procedure *med4*, which calculates the arithmetic average. The arguments are exchanged using the specific registers of MIPS for such a purpose.

Fill in the gaps of *med4* with the three missing instructions (one instruction per gap). Also give the final value of *\$ra*, content of memory *R* and the labels (symbols) *A*, *med4* and *fin*.

<pre>.text 0 lw \$a0, A lw \$a1, B lw \$a2, C lw \$a3, D jal med4 sw \$v0, R fin: j fin</pre>	<pre>.text 0x0100 med4: add \$t0, \$a0, \$a1 add \$t0, \$t0, \$t1</pre>	<pre>.data 0x2000 A: 5 B: 6 C: 7 D: 8 R: 0</pre>
---	---	--

3.12. A 16 bits architecture (ALU word size and registers size) has 6 registers and a total memory space of 64 kBytes. The ALU can perform 20 different operations (add, subtract, and, or, etc...). Please, take into account that this is not a MIPS architecture, and that:

1. All the opcodes have the same length.
2. Not all instructions are of the same size.
3. It is possible to read/write a single byte from memory.

Design the following instruction formats (one format for each type). **Reason your answers.**

Type1: Instructions for operations between registers made in the ALU, at least 40 instructions.

Type2: Instructions for reading/writing in the memory using absolute addressing mode to any address (address included as immediate). At least 5 instructions.

Type3: Instructions for memory but using the addressing mode “relative to register”, with a maximum range of 128 bytes from the register address (the address is obtained adding an immediate to a register). At least 5 instructions.

Type4: Instructions for unconditional jump, with the possibility to jump to any other instruction in the memory. At least 4 instructions.