Computer Hardware Engineering (IS1200)
Computer Organization and Components (IS1500)

Spring 2021

Lecture 2: Assembly Languages

Artur Podobas

Researcher, KTH Royal Institute of Technology

Slides by David Broman (extensions by Artur Podobas), KTH

Course Structure

Module 1: C and Assembly Programming

LE1 → LE2 → LE3 →

LE4 →

EX1 LAB1

S1 LAB2

Module 4: Processor Design

LE9 → LE10 → EX4 S2 LAB4

Module 2: I/O Systems

LE5 → LE6 →

EX2 LAB3

Module 5: Memory Hierarchy

LE11 →

EX5 S3

Module 3: Logic Design
(IS1500 only)

PROJ START

LE7 → LE8 →

EX3 LD-LAB

Module 6: Parallel Processors and Programs

LE12 → LE13 →

EX6 S4

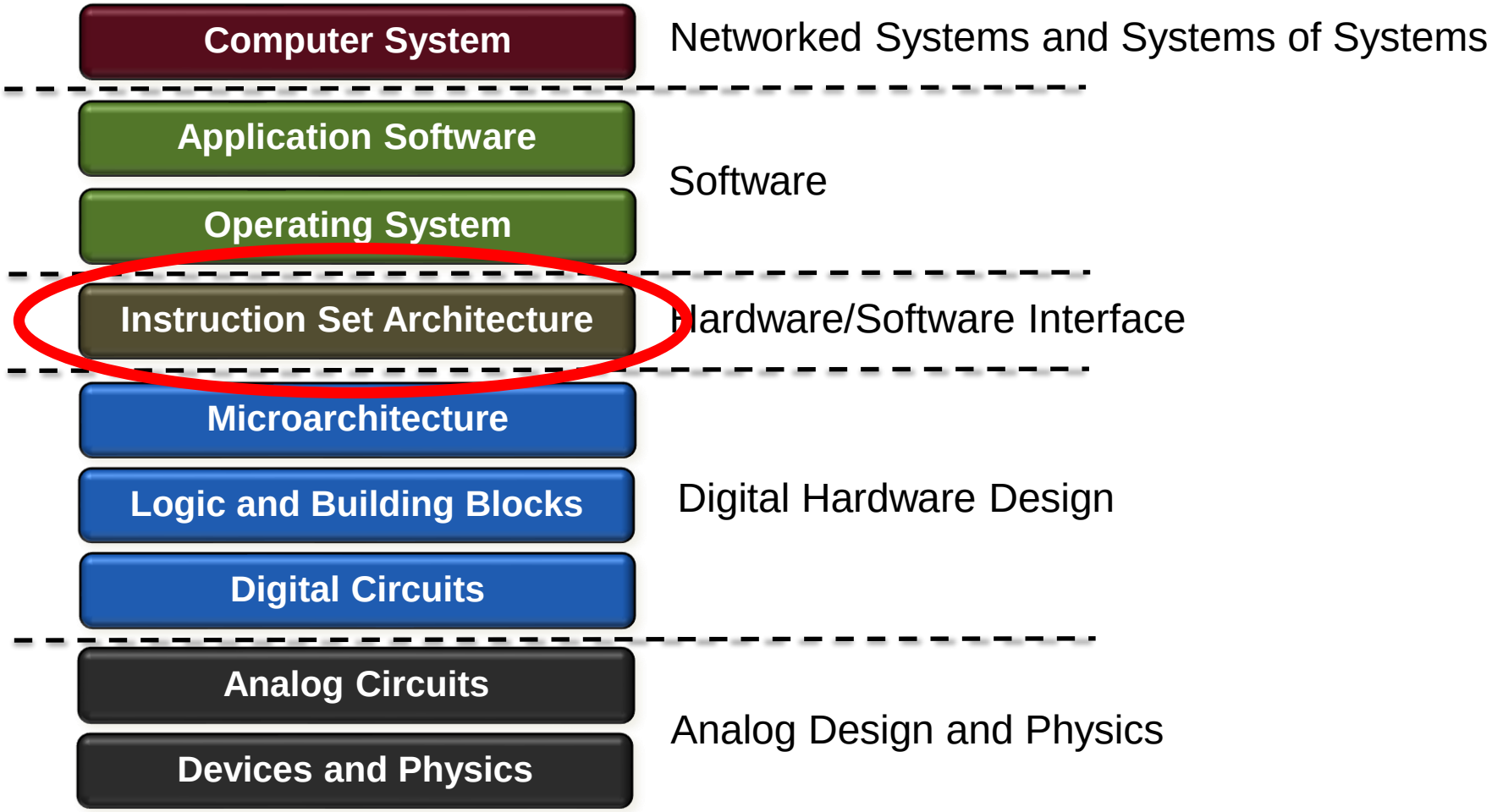
Proj. Expo LE14

Artur Podobas
podobas@kth.se

Part I
Instruction Set Architecture (ISA)

Part II
Basic Assembly Programming

Abstractions in Computer Systems



Agenda

Part I

Instruction Set Architecture (ISA)

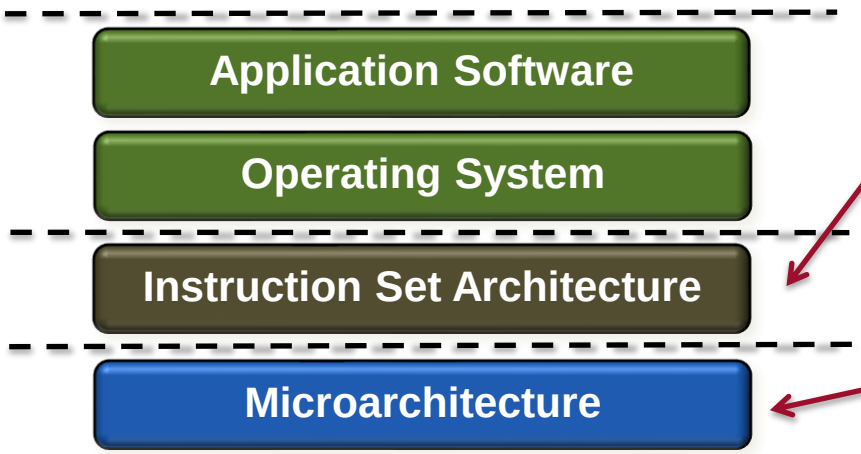
Part II

Basic Assembly Programming

Part I

Instruction Set Architecture

The Instruction Set Architecture (ISA) and its Surrounding



The ISA is the **interface** between hardware and software.

- **Instructions**:
  Encoding and semantics
- **Registers**
- **Memory**

The microarchitecture is the **implementation**.
For instance, both Intel and AMD implement the x86 ISA, but they have different implementations.

Microarchitecture design will be discussed in the course module 4: *Processor design*.

Different ISAs

MIPS is the focus in this course because
i) it is relatively easy to understand
ii) most text books focus on MIPS.

We will only briefly compare with ARM and x86, but they are complex…

RISC-V

ARMv8/SVE

Intel x86

Many other ISAs…
- **POWER/PPC**
- **SuperH (SH)**
- **+ more**

Embedded Real-Time Systems

Photo by Kyro

Personal Computers and Personal Mobile Devices

Picutre Source: Inside HPC

Photo by Robert Harker

Warehouse/Super Scale Computers

Instructions (1/2)
CISC vs. RISC

Each ISA has a set of instructions. Two main categories:

Complex Instruction Set Computers (CISC)

- Many special purpose instructions.
- Example: **x86**. Now almost 900 instructions.
- Typically various encoding lengths (x86, 1-15 bytes)
- Different number of clock cycles, depending on instruction.

Reduced Instruction Set Computers (RISC)

- Few, regular instructions. Minimize hardware complexity.
- **MIPS** is a good example (ARM mostly RISC)
- Typically fixed instruction lengths (e.g., 4 bytes for MIPS)
- Typically one clock cycle per instruction (excluding memory accesses and cache misses)

Instructions (2/2)
C code, Assembly Code, and Machine Code

C Code

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

Compiler
(e.g., GCC)

MIPS Assembly Code

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

Compiler
(e.g., GAS)

MIPS Machine Code

```
0x02328020
```

The compiler maps (if possible) C variables to **registers** (small fast memory locations)

For instance, **a** to **$s0**, **b** to **$s1,** and **c** to **$s2**

(the register names using $ will be explained on the next slide)

The assembly code is in human readable form of the machine code

Each assembly instruction is mapped to one or more machine code instructions.
In MIPS, each instruction is 32 bits.

Registers

Name	Number	Use
$0	0	constant value of 0
$at	1	assembler temporary
$v0-$v1	2-3	function return value
$a0-$a3	4-7	function arguments
$t0-$t7	8-15	temporary (caller-saved)
$s0-$s7	16-23	saved variables (callee-saved)
$t8-$t9	24-25	temporary (caller-saved)
$k0-$k1	26-27	reserved for OS kernel
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Memory

Big problem if 32 registers set the limit of the number of variables in a program. Solution: memory.

Word address					
.					.
.					.
.					.
0000 000C	0f	a0	b0	12	Word 3
0000 0008	44	93	4e	aa	Word 2
0000 0004	33	fa	01	23	Word 1
0000 0000	21	a0	1b	33	Word 0
Byte address	0	1	2	3	

Memory

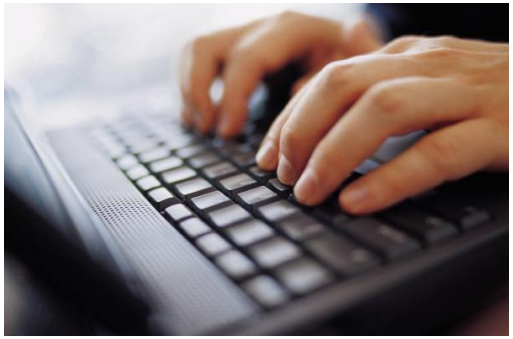
- Has many more data locations than registers.
- Accessing memory is slower than accessing registers.

- **Big-endian:** the most significant byte (MSB) of the word is stored at the lowest memory address.
- **Little-endian:** the least significant byte (LSB) is stored at the lowest memory address.

The choice of endianness is arbitrary, but creates problems when communicating between processors with different endianness.

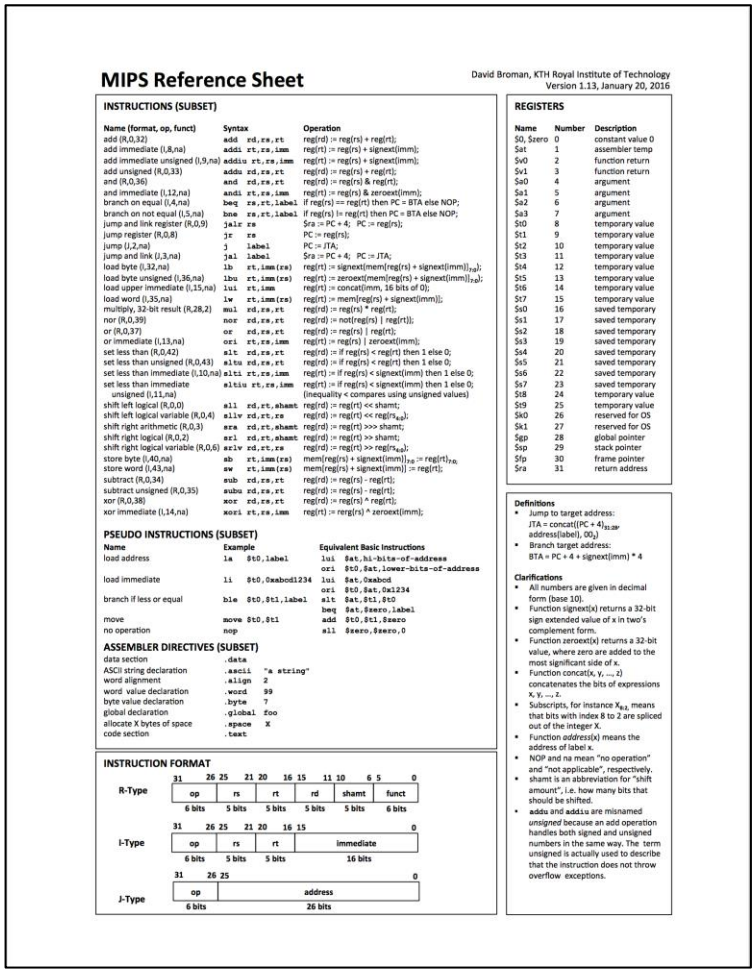
Part II

Assembly Programming



Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

MIPS Reference Sheet



MIPS Reference Sheet

David Broman, KTH Royal Institute of Technology
Version 1.13, January 20, 2016

INSTRUCTIONS (SUBSET)

Name (format, op, funct)	Syntax	Operation
add (R,0,32)	`add rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
add immediate (I,8,na)	`addi rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add immediate unsigned (I,9,na)	`addiu rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add unsigned (R,0,33)	`addu rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
and (R,0,36)	`and rd,rs,rt`	reg(rd) := reg(rs) & reg(rt);
and immediate (I,12,na)	`andi rt,rs,imm`	reg(rt) := reg(rs) & zeroext(imm);
branch on equal (I,4,na)	`beq rs,rt,label`	if reg(rs) == reg(rt) then PC = BTA else NOP;
branch on not equal (I,5,na)	`bne rs,rt,label`	if reg(rs) != reg(rt) then PC = BTA else NOP;
jump and link register (R,0,9)	`jalr rs`	$ra := PC + 4; PC := reg(rs);
jump register (R,0,8)	`jr rs`	PC := reg(rs);
jump (J,2,na)	`j label`	PC := JTA;
jump and link (J,3,na)	`jal label`	$ra := PC + 4; PC := JTA;
load byte (I,32,na)	`lb rt,imm(rs)`	reg(rt) := signext(mem(reg(rs) + signext(imm))$_{7:0}$);
load byte unsigned (I,36,na)	`lbu rt,imm(rs)`	reg(rt) := zeroext(mem(reg(rs) + signext(imm))$_{7:0}$);
load upper immediate (I,15,na)	`lui rt,imm`	reg(rt) := concat(imm, 16 bits of 0);
load word (I,35,na)	`lw rt,imm(rs)`	reg(rt) := mem[reg(rs) + signext(imm)];
multiply, 32-bit result (R,28,2)	`mul rd,rs,rt`	reg(rd) := reg(rs) * reg(rt);
nor (R,0,39)	`nor rd,rs,rt`	reg(rd) := not(reg(rs) | reg(rt));
or (R,0,37)	`or rd,rs,rt`	reg(rd) := reg(rs) | reg(rt);
or immediate (I,13,na)	`ori rt,rs,imm`	reg(rt) := reg(rs) | zeroext(imm);
set less than (R,0,42)	`slt rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than unsigned (R,0,43)	`sltu rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than immediate (I,10,na)	`slti rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0;
set less than immediate unsigned (I,11,na)	`sltiu rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0; (inequality < compares using unsigned values)
shift left logical (R,0,0)	`sll rd,rt,shamt`	reg(rd) := reg(rt) << shamt;
shift left logical variable (R,0,4)	`sllv rd,rt,rs`	reg(rd) := reg(rt) << reg(rs$_{4:0}$);
shift right arithmetic (R,0,3)	`sra rd,rt,shamt`	reg(rd) := reg(rt) >>> shamt;
shift right logical (R,0,2)	`srl rd,rt,shamt`	reg(rd) := reg(rt) >> shamt;
shift right logical variable (R,0,6)	`srlv rd,rt,rs`	reg(rd) := reg(rt) >> reg(rs$_{4:0}$);
store byte (I,40,na)	`sb rt,imm(rs)`	mem(reg(rs) + signext(imm))$_{7:0}$:= reg(rt)$_{7:0}$;
store word (I,43,na)	`sw rt,imm(rs)`	mem(reg(rs) + signext(imm)) := reg(rt);
subtract (R,0,34)	`sub rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
subtract unsigned (R,0,35)	`subu rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
xor (R,0,38)	`xor rd,rs,rt`	reg(rd) := reg(rs) ^ reg(rt);
xor immediate (I,14,na)	`xori rt,rs,imm`	reg(rt) := rerg(rs) ^ zeroext(imm);

PSEUDO INSTRUCTIONS (SUBSET)

Name	Example	Equivalent Basic Instructions
load address	`la $t0,label`	`lui $at,hi-bits-of-address` `ori $t0,$at,lower-bits-of-address`
load immediate	`li $t0,0xabcd1234`	`lui $at,0xabcd` `ori $t0,$at,0x1234`
branch if less or equal	`ble $t0,$t1,label`	`slt $at,$t1,$t0` `beq $at,$zero,label`
move	`move $t0,$t1`	`add $t0,$t1,$zero`
no operation	`nop`	`sll $zero,$zero,0`

ASSEMBLER DIRECTIVES (SUBSET)

data section	`.data`
ASCII string declaration	`.ascii "a string"`
word alignment	`.align 2`
word value declaration	`.word 99`
byte value declaration	`.byte 7`
global declaration	`.global foo`
allocate X bytes of space	`.space X`
code section	`.text`

INSTRUCTION FORMAT

	31–26	25–21	20–16	15–11	10–6	5–0
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	immediate		
	6 bits	5 bits	5 bits	16 bits		
J-Type	op	address				
	6 bits	26 bits				

REGISTERS

Name	Number	Description
$0, $zero	0	constant value 0
$at	1	assembler temp
$v0	2	function return
$v1	3	function return
$a0	4	argument
$a1	5	argument
$a2	6	argument
$a3	7	argument
$t0	8	temporary value
$t1	9	temporary value
$t2	10	temporary value
$t3	11	temporary value
$t4	12	temporary value
$t5	13	temporary value
$t6	14	temporary value
$t7	15	temporary value
$s0	16	saved temporary
$s1	17	saved temporary
$s2	18	saved temporary
$s3	19	saved temporary
$s4	20	saved temporary
$s5	21	saved temporary
$s6	22	saved temporary
$s7	23	saved temporary
$t8	24	temporary value
$t9	25	temporary value
$k0	26	reserved for OS
$k1	27	reserved for OS
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Definitions

- Jump to target address: JTA = concat((PC + 4)$_{31:28}$, address(label), 00_2)
- Branch target address: BTA = PC + 4 + signext(imm) * 4

Clarifications

- All numbers are given in decimal form (base 10).
- Function signext(x) returns a 32-bit sign extended value of x in two's complement form.
- Function zeroext(x) returns a 32-bit value, where zero are added to the most significant side of x.
- Function concat(x, y, ..., z) concatenates the bits of expressions x, y, ..., z.
- Subscripts, for instance $X_{8:2}$, means that bits with index 8 to 2 are spliced out of the integer X.
- Function *address*(x) means the address of label x.
- NOP and na mean "no operation" and "not applicable", respectively.
- shamt is an abbreviation for "shift amount", i.e. how many bits that should be shifted.
- `addu` and `addiu` are misnamed *unsigned* because an add operation handles both signed and unsigned numbers in the same way. The term unsigned is actually used to describe that the instruction does not throw overflow exceptions.

- Will be **available on the exam** (attached to the questions)
- Summarizes an important **subset of the MIPS instructions** and their coding.
- Available for **download** from the course page (under "Course Literature")

Arithmetic/Logical Instructions

MIPS Logical Instructions

```
and   $s0, $s1, $s2
or    $s0, $s1, $s2
xor   $s0, $s1, $s2
nor   $s0, $s1, $s2
```

Instructions AND, OR, XOR, and NOT OR.

There is no **not** instruction. How can we do **not $s1** and store it in $t0?

```
andi   $s0, $s1, 0xAB41
ori    $s0, $s1, 0xFF01
xori   $s0, $s1, 0x78
```

Instructions AND immediate, OR immediate, and XOR immediate.

```
sll   $t0, $s0, 3
srl   $t0, $s0, 29
sra   $t0, $s0, 29
```

Shift left logical (same as C operator <<).

Shift right logical (same as C operator >>).

Shift right arithmetic. Shifts in the sign bit as the most significant bit. Dividing signed numbers.

Constants Values

How can we assign a register a constant value?

Max 16-bit

```
addi  $s0, $0, 2342
```

How can we give a register a 32-bit constant?

```
int a = 0x6af022e7;
```

Hint: There is an instruction load upper immediate, **lui $t0, 0xff12** that loads the 16 most significant bits to the immediate value, and sets the lower to 0.

```
lui $s0,0x6af0
ori $s0,$s0,0x22e7
```

Requires 2 instructions.

Conditional Branches (1/3) beq and bne

Branch if equal (beq) branches if two operands have equal values.

Branch if not equal (bne) branches if two operands do not have equal values.

```
  addi   $s0, $0, 4
  xori   $s1, $s0, 1
  sll    $t0, $s1, 1
  beq    $t0, $s0, foo
  add    $s1, $s1, $s0
foo:
  add    $s5, $s1, $0
```

Set $s0 to 4. XOR immediate results in $s1=5. Shift logic left results in that $t0 is 10. Hence, $t0 and $s0 are not equal, so the branch is not taken and add is executed. This results in that $s1 is 9.

There is no MOV instruction in MIPS, but **add** can be used for this (as it is done here).

Exercise: What is the value of $s5?

Note: There is a **pseudoinstruction** called **move** in the MIPS assembler. It is implemented using add.

Conditional Branches (2/3) if-statement, if/else-statement

```
if(i==j)
  f = i;
f = f - j;
```

How can the C code be translated to MIPS code? Assume mapping, i to $s0, j to $s1, and f to $t0.

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

Translate to MIPS code, using previous mapping

Conditional Branches (3/3) for-loops

```
int sum = 0;
for(int i=1; i < 101; i = i * 2)
  sum = sum + i;
```

Help: Instruction **set less than (slt)**.
`slt $t0,$s0,$s1` sets $t0 to 1 if $s0 is less than $s1, else $t0 is set to 0.

Translated to MIPS code using mapping:
i to $s0, sum to $s1

Example from Harris & Harris, 2013, page 320

Arrays and Memory Access

```
int ar[5];
ar[0] = ar[0] * 8;
ar[1] = ar[1] * 8;
```

Arrays are defined and accessed using [] in C.

Translate to MIPS code.
Let the Array address be 0x10007000

MARS Simulator Demo (1/2) Example

```
.data
.align 2
msg:      .space    8

.text
main:     la        $t1, msg
          addi      $t2,$zero,0x27
          sb        $t2,0($t1)
          addi      $t2,$zero,0x18
          sb        $t2,1($t1)
          li        $t2,0x4b544800
          sw        $t2,4($t1)

stop:     j         stop
```

Assembler directives:

.data the following is stored in the data section

.align 2 the following is word aligned

.space 8 the assembler reserves 8 bytes of space

.text the following is machine code

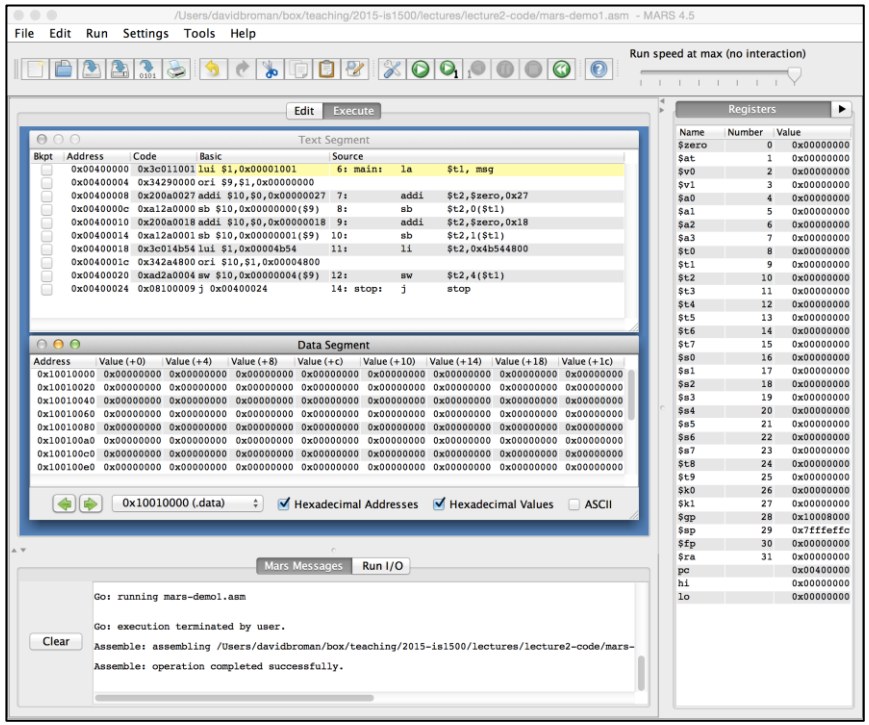
la = load address of a label

sb = store byte

Infinite loop. Makes the program "stop"

li = load immediate
Pseudo instruction (translated by the assembler into 1 or 2 basic instructions)

MARS Simulator Demo (2/2)
Understanding the Previous Example

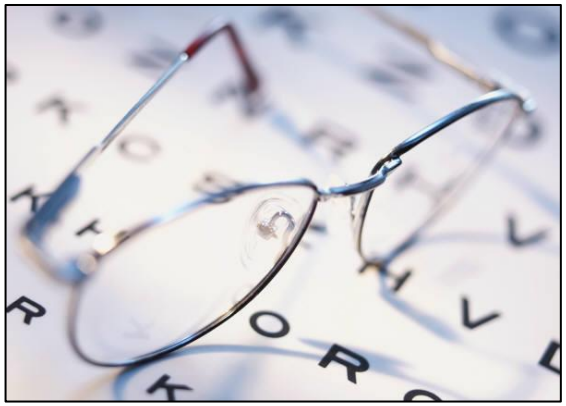


The demo shows the following:

- Where is the help?
- Registers
- Debugging a program
- Instruction encoding
- Run to breakpoint
- Pseudo instruction encoding
- Data segment, HEX and ASCII views

Exercise:
What is the program actually doing? What is stored at the **msg** label?
(Try the example yourself in the simulator)

Reading Guidelines – Module 1



Introduction
P&H5 Chapters 1.1-1.4, or P&H4 1.1-1.3

Number systems
H&H Chapter 1.4

C Programming
H&H Appendix C
Online links on the literature webpage

Assembly and Machine Languages
H&H Chapters 6.1-6.9, 5.3
The MIPS sheet (see the literature page)

Reading Guidelines
See the course webpage for more information.

You can focus on Chapters 6.1-6.4 for Lab 1

Just one more thing…

(please do not fumble with thebags)

Summary

Some key take away points:

- An **Instruction Set Architecture (ISA)** defines the software/hardware interface, whereas a **microarchitecture** implements an ISA.
- There are many different ISAs. Some of the major ones are **x86**, **ARM**, **RISC-V**, and **MIPS**.
- MIPS is a simple yet powerful ISA. It is a good idea to thoroughly understand the **MIPS Reference Sheet**.
- It is important to understand **the concept of assembly programming**, although very few programs are actually written in assembly today.

Thanks for listening!