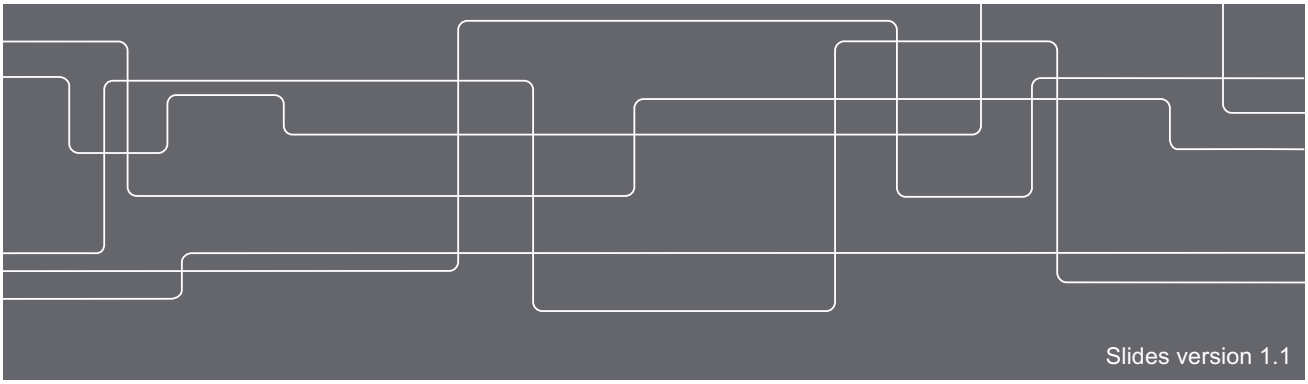
Computer Hardware Engineering (IS1200)
Computer Organization and Components (IS1500)

Spring 2018

Lecture 2: Assembly Languages

David Broman

Associate Professor, KTH Royal Institute of Technology



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

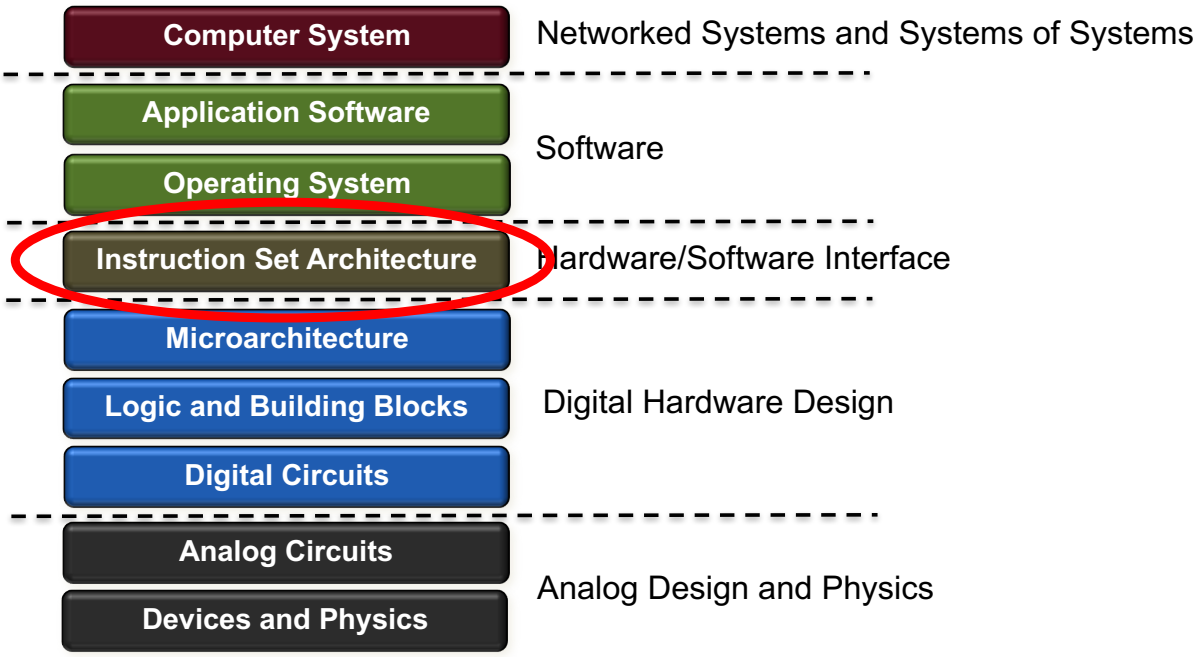
LE7 → LE8 → EX3 LD-LAB

Module 6: Parallel Processors and Programs

LE12 → LE13 → EX6 S4

Proj. Expo LE14

Abstractions in Computer Systems



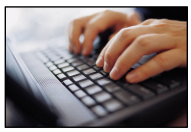
Agenda

Part I

Instruction Set Architecture (ISA)

Part II

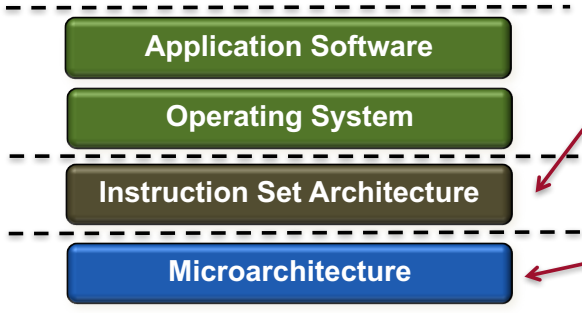
Basic Assembly Programming



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**

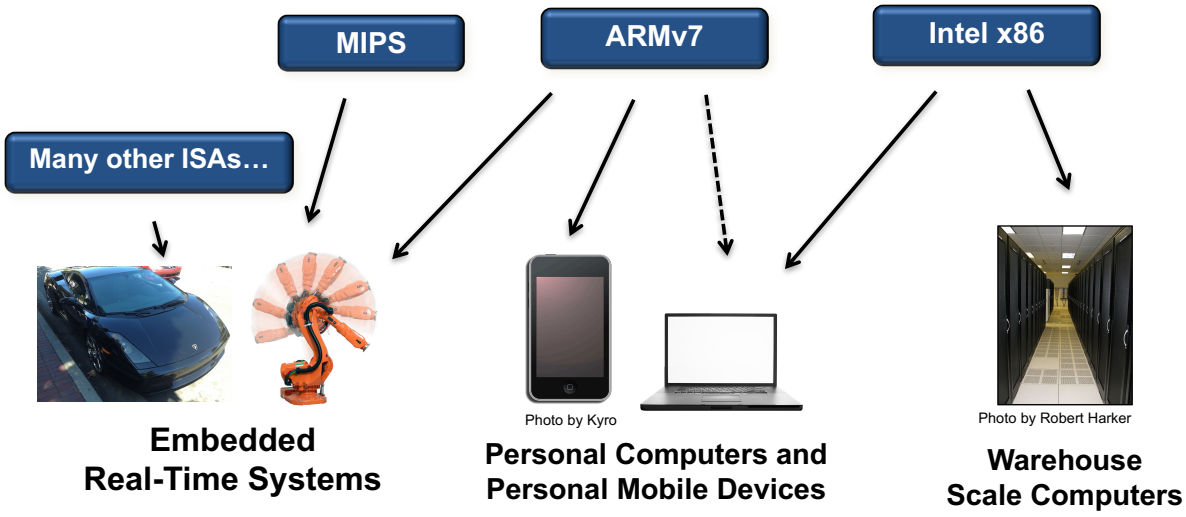
The microarchitecture is the **implementation**.
For instance, both Intel and AMD implement the x86 ISA, but they have different implementations.

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…



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;
```

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)

MIPS Assembly Code

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

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

MIPS Machine Code

```
0x02328020
```

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 kernal
$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

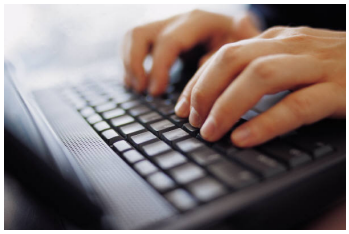
- Has many more data locations than registers.
- Accessing memory is slower than accessing registers.

- **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.

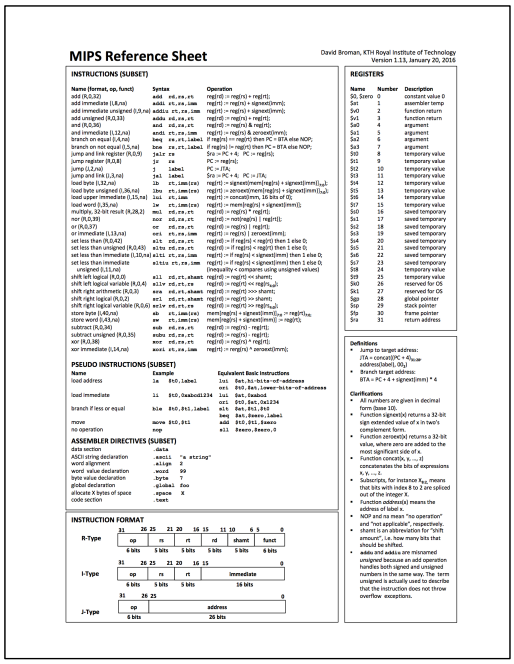
The choice of endianness is arbitrary, but creates problems when communicating between processors with different endianness.

Part II

Assembly Programming



MIPS Reference Sheet



- 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?

```
nor   $t0, $s1, $0
```

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

Instructions AND immediate, OR immediate, and XOR immediate.

MIPS Logical Instructions

```
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?

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

Max 16-bit

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).

What is the value of $s5?
Stand for 9, sleep for 10.

Answer: 9

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.

```
  bne $s0, $s1, L1
  add $t0, $s0, $0
L1:
  sub $t0, $t0, $s1
```

Note: Tests the opposite.

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

Translate to MIPS code, using previous mapping

```
  beq $s0, $s1, else
  add $t0, $s0, $0
  j   L1
else:
  add $t0, $s0, $s1
L1:
  sub $t0, $t0, $s1
```


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

```
  addi $s1, $0, 0    # sum = 0
  addi $s0, $0, 1    # i = 1
  addi $t0, $0, 101  # $t0 = 101
loop:
  slt $t1, $s0, $t0  # if(i<101)
                     # $t1=1 else $t1=0
  beq $t1, $0, done  # if $t1 == 0, branch
  add $s1, $s1, $s0  # sum = sum + i
  sll $s0, $s0, 1    # i = i * 2
  j   loop
done:
```

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.

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

```
lui $s0, 0x1000
ori $s0, $s0,0x7000

lw  $t1, 0($s0)
sll $t1, $t1, 3
sw  $t1, 0($s0)

lw  $t1, 4($s0)
sll $t1, $t1, 3
sw  $t1, 4($s0)
```

lw loads a word from the **effective address** $s0 + 0. The effective address is the sum of the base address ($s0) and offset (4 in the second case).

sw stores back a word using the computed effective address.

Note the byte address (each word is 32-bit)

Example from Harris & Harris, 2013, page 321

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

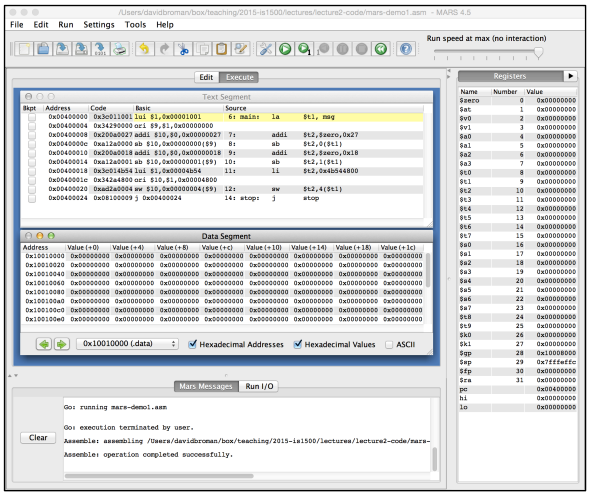
la = load address of a label

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

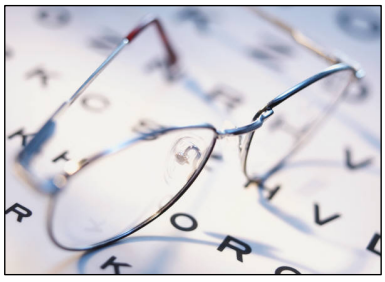


The demo shows the following:

- 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 the bags)

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**, 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!