



Computer Hardware Engineering (IS1200)

Computer Organization and Components (IS1500)

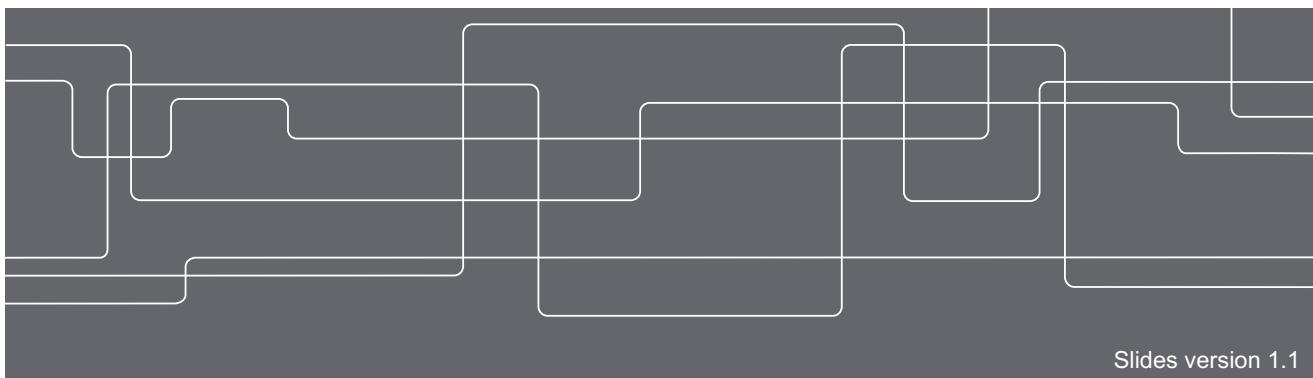
Spring 2019

Lecture 2: Assembly Languages

Elias Castegren

Postdoc, KTH Royal Institute of Technology

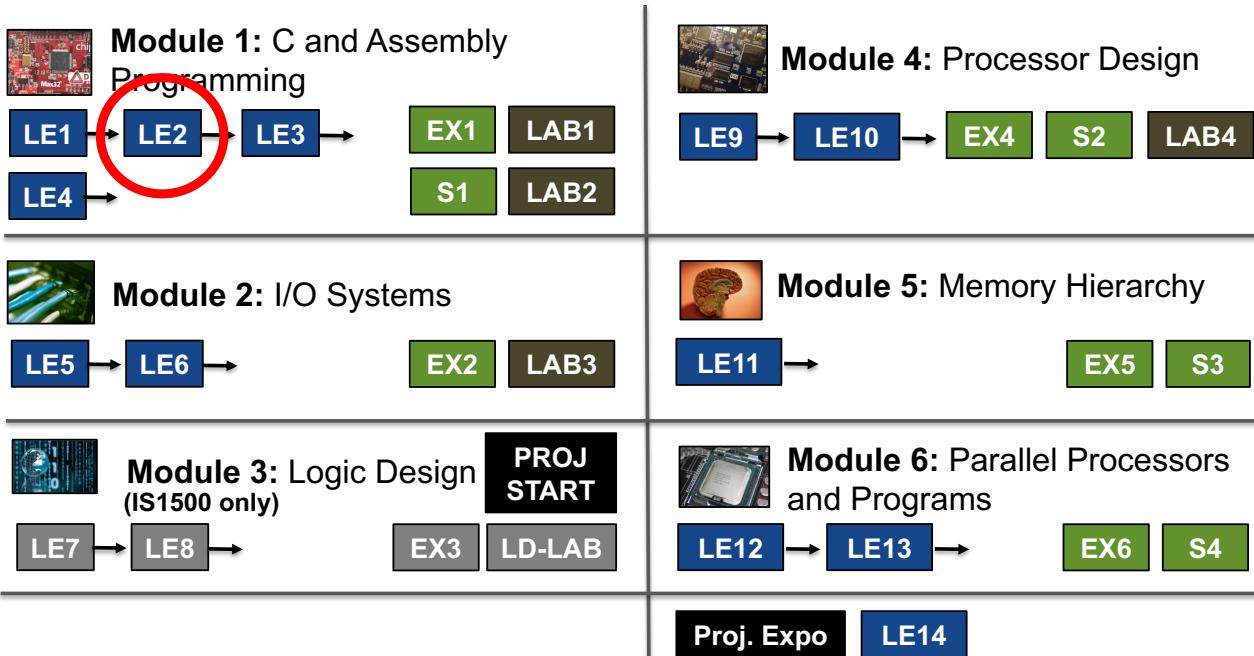
Slides by David Broman



2

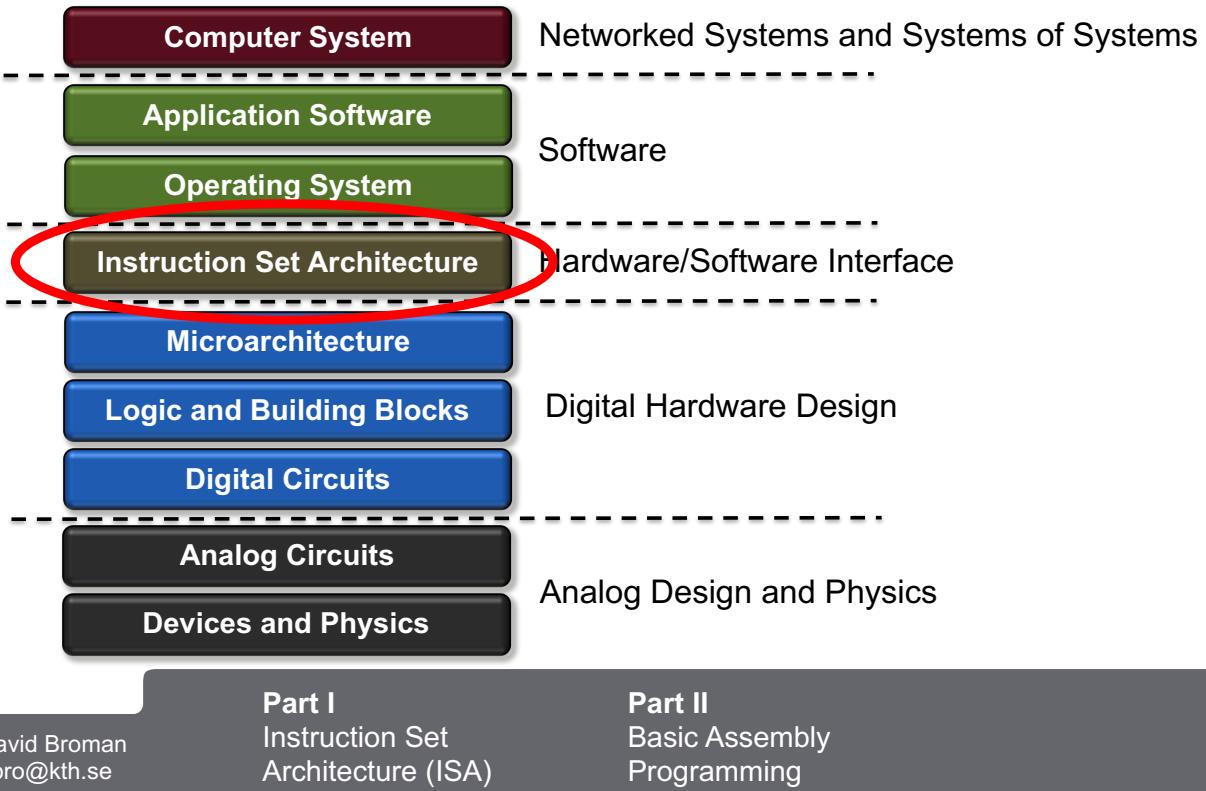


Course Structure





Abstractions in Computer Systems



Agenda

Part I
Instruction Set Architecture (ISA)



Part II
Basic Assembly Programming



David Broman dbro@kth.se	Part I Instruction Set Architecture (ISA)	Part II Basic Assembly Programming
-----------------------------	--	---

Part I

Instruction Set Architecture



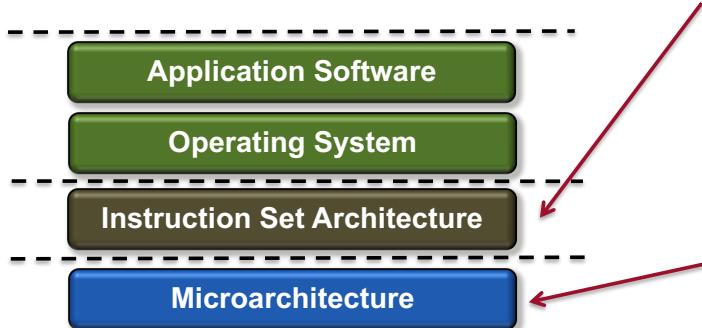
David Broman
dbro@kth.se



Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming

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*.

David Broman
dbro@kth.se



Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming

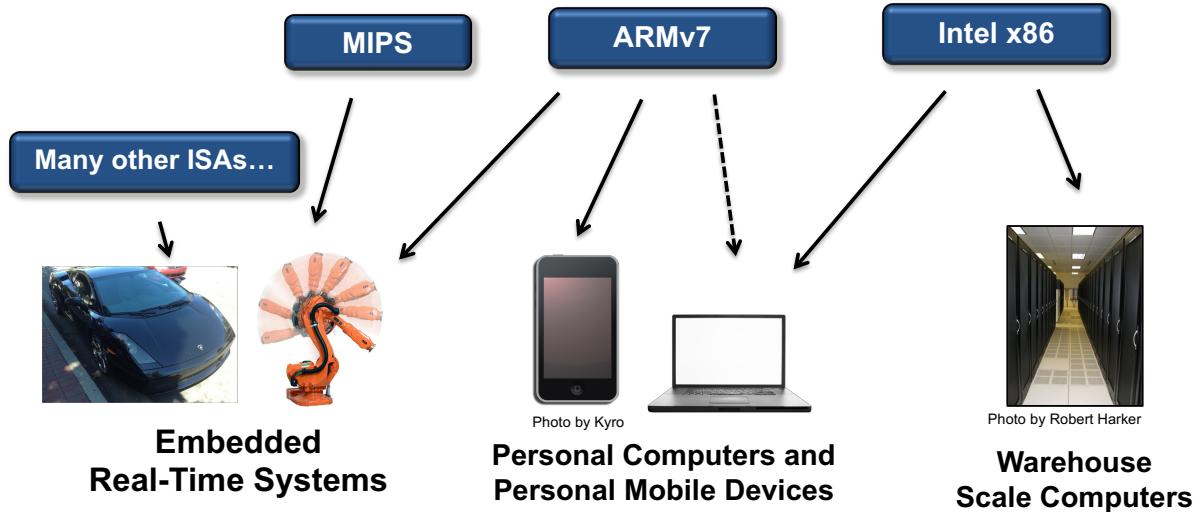


Different ISAs

MIPS is the focus in this course because

- it is relatively easy to understand
- most text books focus on MIPS.

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



David Broman
dbro@kth.se



Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



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)

David Broman
dbro@kth.se



Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



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.

David Broman
dbro@kth.se



Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



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

David Broman
dbro@kth.se



Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



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.

David Broman
dbro@kth.se



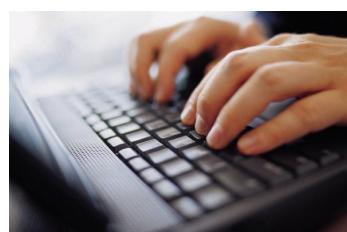
Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



Part II

Assembly Programming



David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



MIPS Reference Sheet

MIPS Reference Sheet																																																																																															
<small>David Broman, KTH Royal Institute of Technology Version 1.13, January 20, 2013</small>																																																																																															
INSTRUCTIONS (SUBSET) Name (Format, op, func) Syntax Operation																																																																																															
add (R-Type) add \$d,\$s,\$t register = register + register; add immediate (I-Type) addi \$d,\$s,\$i register = register + signextended(\$i); add unsigned (R-Type) addu \$d,\$s,\$t register = register + register; addu immediate (I-Type) addiu \$d,\$s,\$i register = register + signextended(\$i); and (R-Type) and \$d,\$s,\$t register = register & register; and immediate (I-Type) andi \$d,\$s,\$i register = register & signextended(\$i); branch on not equal (L-Type) beq \$s,\$t,\$label if register = register then PC = PC else NOP; branch on equal (L-Type) bne \$s,\$t,\$label if register != register then PC = PC else NOP; jump register (R-Type) jr \$s PC = register; jump (I-Type) j \$label PC = \$label; jump and link (J-Type) jal \$label Sra = PC; PC = \$label; load byte (I-Type) lb \$t,\$s register = memory[\$s]; load halfword (I-Type) lh \$t,\$s register = memory[\$s]; load word (I-Type) lw \$t,\$s register = memory[\$s]; load word immediate (I-Type) lwz \$t,\$s,\$i register = signextended(\$i); multiply, 32-bit result (U-Type) mul \$d,\$s,\$t register = register * register; or (R-Type) or \$d,\$s,\$t register = register register; or immediate (I-Type) ori \$d,\$s,\$i register = register signextended(\$i); set less than (A-Type) slt \$t,\$s,\$t register = 1 if register < register then 1 else 0; set less than or equal (A-Type) sltu \$t,\$s,\$t register = 1 if register <= register then 1 else 0; set less than inverted (A-Type) sltu \$t,\$s,\$t register = 0 if register < register then 1 else 0; set less than or equal inverted (A-Type) sltui \$t,\$s,\$t register = 0 if register <= register then 1 else 0; shift left logical (A-Type) sll \$d,\$s,\$t register = register << register; shift right logical (A-Type) srl \$d,\$s,\$t register = register >> register; shift right arithmetic (A-Type) sra \$d,\$s,\$t register = register >> register; store byte (A-Type) sb \$t,\$s memory[\$s] = signextended(\$t); store halfword (A-Type) sh \$t,\$s memory[\$s] = signextended(\$t); store word (A-Type) sw \$t,\$s memory[\$s] = signextended(\$t); subtract (R-Type) sub \$d,\$s,\$t register = register - register; subtract immediate (I-Type) subi \$d,\$s,\$i register = register - signextended(\$i); xor (R-Type) xor \$d,\$s,\$t register = register ^ register; xor immediate (I-Type) xori \$d,\$s,\$i register = register ^ signextended(\$i);																																																																																															
PSEUDO INSTRUCTIONS (SUBSET) Name Example Equivalent Basic Instructions																																																																																															
load address la \$t,\$label equivalent basic instruction: add \$t,\$0,\$label branch if less or equal blt \$t,\$label branch target address: \$t < \$label branch if not equal bne \$t,\$label branch target address: \$t != \$label move move \$t,\$s add \$t,\$0,\$s no operation nop add \$t,\$t,\$t																																																																																															
ASSEMBLER DIRECTIVES (SUBSET) directives example																																																																																															
data .data ASCII string declaration .ascii "a string" word alignment .align 2 word .word byte value declaration .byte 99 global declaration .global _foo global space .space X code space .text																																																																																															
INSTRUCTION FORMAT <table border="1"> <tr> <td>R-Type</td> <td>op</td> <td>r1</td> <td>r2</td> <td>16</td> <td>15</td> <td>11</td> <td>20</td> <td>21</td> <td>25</td> <td>26</td> <td>27</td> <td>28</td> <td>29</td> <td>30</td> <td>shamt</td> <td>funct</td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> <td>6 bits</td> <td>5 bits</td> <td>3 bits</td> <td>5 bits</td> <td>3 bits</td> <td>5 bits</td> <td>6 bits</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </table> <table border="1"> <tr> <td>I-Type</td> <td>op</td> <td>r1</td> <td>r2</td> <td>16</td> <td>15</td> <td>11</td> <td>20</td> <td>21</td> <td>25</td> <td>26</td> <td>27</td> <td>28</td> <td>29</td> <td>30</td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> <td>6 bits</td> <td>5 bits</td> <td>3 bits</td> <td>5 bits</td> <td>3 bits</td> <td>5 bits</td> <td>6 bits</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </table> <table border="1"> <tr> <td>J-Type</td> <td>op</td> <td>address</td> <td>0</td> </tr> <tr> <td></td> <td></td> <td>26 bits</td> <td></td> </tr> </table>			R-Type	op	r1	r2	16	15	11	20	21	25	26	27	28	29	30	shamt	funct	0					6 bits	5 bits	3 bits	5 bits	3 bits	5 bits	6 bits								I-Type	op	r1	r2	16	15	11	20	21	25	26	27	28	29	30	0					6 bits	5 bits	3 bits	5 bits	3 bits	5 bits	6 bits							J-Type	op	address	0			26 bits																	
R-Type	op	r1	r2	16	15	11	20	21	25	26	27	28	29	30	shamt	funct	0																																																																														
				6 bits	5 bits	3 bits	5 bits	3 bits	5 bits	6 bits																																																																																					
I-Type	op	r1	r2	16	15	11	20	21	25	26	27	28	29	30	0																																																																																
				6 bits	5 bits	3 bits	5 bits	3 bits	5 bits	6 bits																																																																																					
J-Type	op	address	0																																																																																												
		26 bits																																																																																													
REGISTERS <table border="1"> <tr> <td>Name</td> <td>Number</td> <td>Description</td> </tr> <tr> <td>\$t</td> <td>1</td> <td>assembler temp</td> </tr> <tr> <td>\$t1</td> <td>2</td> <td>function return</td> </tr> <tr> <td>\$t2</td> <td>3</td> <td>argument</td> </tr> <tr> <td>\$t3</td> <td>4</td> <td>argument</td> </tr> <tr> <td>\$t4</td> <td>5</td> <td>argument</td> </tr> <tr> <td>\$t5</td> <td>6</td> <td>argument</td> </tr> <tr> <td>\$t6</td> <td>7</td> <td>argument</td> </tr> <tr> <td>\$t7</td> <td>8</td> <td>temporary value</td> </tr> <tr> <td>\$t8</td> <td>9</td> <td>temporary value</td> </tr> <tr> <td>\$t9</td> <td>10</td> <td>temporary value</td> </tr> <tr> <td>\$t10</td> <td>11</td> <td>temporary value</td> </tr> <tr> <td>\$t11</td> <td>12</td> <td>temporary value</td> </tr> <tr> <td>\$t12</td> <td>13</td> <td>temporary value</td> </tr> <tr> <td>\$t13</td> <td>14</td> <td>temporary value</td> </tr> <tr> <td>\$t14</td> <td>15</td> <td>temporary value</td> </tr> <tr> <td>\$t15</td> <td>16</td> <td>used temporary</td> </tr> <tr> <td>\$t16</td> <td>17</td> <td>used temporary</td> </tr> <tr> <td>\$t17</td> <td>18</td> <td>used temporary</td> </tr> <tr> <td>\$t18</td> <td>19</td> <td>used temporary</td> </tr> <tr> <td>\$t19</td> <td>20</td> <td>used temporary</td> </tr> <tr> <td>\$t20</td> <td>21</td> <td>used temporary</td> </tr> <tr> <td>\$t21</td> <td>22</td> <td>used temporary</td> </tr> <tr> <td>\$t22</td> <td>23</td> <td>used temporary</td> </tr> <tr> <td>\$t23</td> <td>24</td> <td>temporary value</td> </tr> <tr> <td>\$t24</td> <td>25</td> <td>reserved for OS</td> </tr> <tr> <td>\$t25</td> <td>26</td> <td>reserved for OS</td> </tr> <tr> <td>\$t26</td> <td>27</td> <td>global pointer</td> </tr> <tr> <td>\$t27</td> <td>28</td> <td>frame pointer</td> </tr> <tr> <td>\$t28</td> <td>29</td> <td>frame pointer</td> </tr> <tr> <td>\$t29</td> <td>30</td> <td>return address</td> </tr> </table>			Name	Number	Description	\$t	1	assembler temp	\$t1	2	function return	\$t2	3	argument	\$t3	4	argument	\$t4	5	argument	\$t5	6	argument	\$t6	7	argument	\$t7	8	temporary value	\$t8	9	temporary value	\$t9	10	temporary value	\$t10	11	temporary value	\$t11	12	temporary value	\$t12	13	temporary value	\$t13	14	temporary value	\$t14	15	temporary value	\$t15	16	used temporary	\$t16	17	used temporary	\$t17	18	used temporary	\$t18	19	used temporary	\$t19	20	used temporary	\$t20	21	used temporary	\$t21	22	used temporary	\$t22	23	used temporary	\$t23	24	temporary value	\$t24	25	reserved for OS	\$t25	26	reserved for OS	\$t26	27	global pointer	\$t27	28	frame pointer	\$t28	29	frame pointer	\$t29	30	return address
Name	Number	Description																																																																																													
\$t	1	assembler temp																																																																																													
\$t1	2	function return																																																																																													
\$t2	3	argument																																																																																													
\$t3	4	argument																																																																																													
\$t4	5	argument																																																																																													
\$t5	6	argument																																																																																													
\$t6	7	argument																																																																																													
\$t7	8	temporary value																																																																																													
\$t8	9	temporary value																																																																																													
\$t9	10	temporary value																																																																																													
\$t10	11	temporary value																																																																																													
\$t11	12	temporary value																																																																																													
\$t12	13	temporary value																																																																																													
\$t13	14	temporary value																																																																																													
\$t14	15	temporary value																																																																																													
\$t15	16	used temporary																																																																																													
\$t16	17	used temporary																																																																																													
\$t17	18	used temporary																																																																																													
\$t18	19	used temporary																																																																																													
\$t19	20	used temporary																																																																																													
\$t20	21	used temporary																																																																																													
\$t21	22	used temporary																																																																																													
\$t22	23	used temporary																																																																																													
\$t23	24	temporary value																																																																																													
\$t24	25	reserved for OS																																																																																													
\$t25	26	reserved for OS																																																																																													
\$t26	27	global pointer																																																																																													
\$t27	28	frame pointer																																																																																													
\$t28	29	frame pointer																																																																																													
\$t29	30	return address																																																																																													

- 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”)

David Broman
dbro@kth.se

Part I Instruction Set Architecture (ISA)

Part II Basic Assembly Programming



Logical Instructions

14
E

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 Shift 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.

David Broman
dbro@kth.se

Part I Instruction Set Architecture (ISA)

Part II Basic Assembly Programming



Constant 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.



David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming

16
E



Conditional Branches (1/3) beq and bne

Branch if equal (**beq**) branches if two operands 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**.

David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming

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

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

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
```

```
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.

Translate 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:
```

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

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

Arrays are defined and accessed using [] in C.

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)

```
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)
```

Example from Harris & Harris, 2013, page 321

David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming

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
```

Infinite loop.
Makes the
program “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

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

David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



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)

David Broman
dbro@kth.se

Part I

Instruction Set Architecture (ISA)

Part II

Basic Assembly Programming



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

Reading Guidelines
See the course webpage
for more information.

You can focus on Chapters 6.1-6.4 for Lab 1

David Broman
dbro@kth.se

Part I

Instruction Set Architecture (ISA)

Part II

Basic Assembly Programming



Just one more thing...



David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming



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!

David Broman
dbro@kth.se

Part I
Instruction Set
Architecture (ISA)

Part II
Basic Assembly
Programming