



Computer Hardware Engineering (IS1200)

Computer Organization and Components (IS1500)

Spring 2021

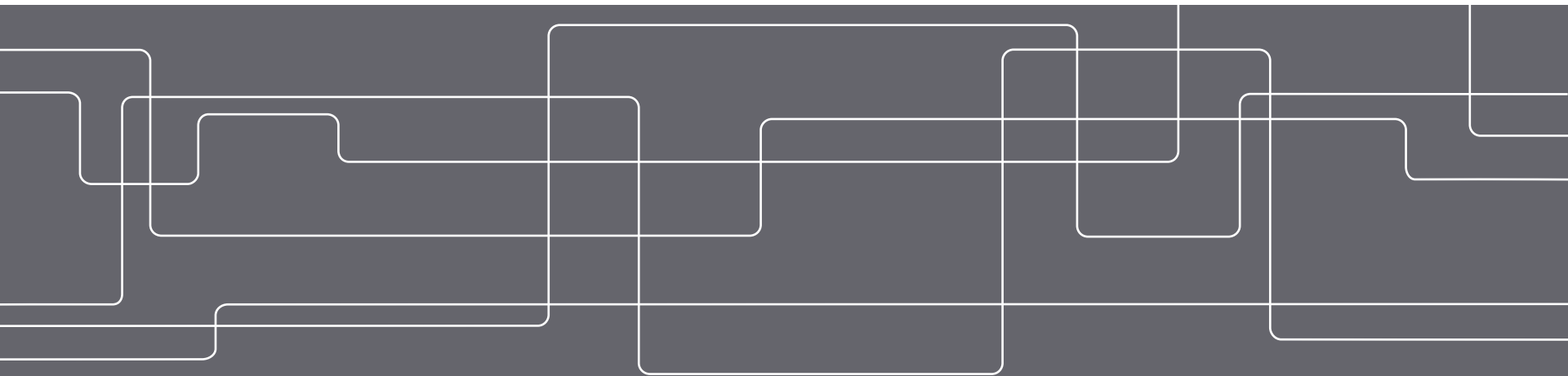
Lecture 3: Machine Languages

(Note: We start 10:15)

Artur Podobas

Researcher, KTH Royal Institute of Technology

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

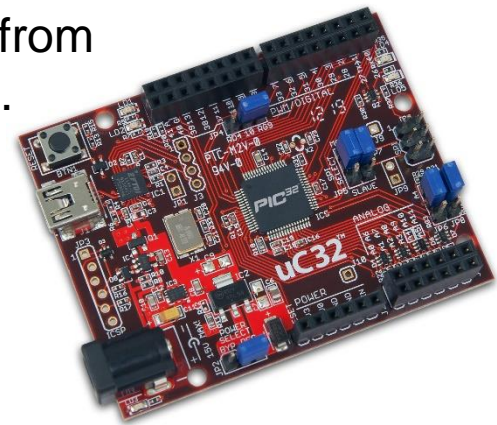




Announcement

Announcement #1: There are many questions about ChipKits. We will send out a Canvas announcement when they will be available for pickup. Note that a majority of the first lab does not use ChipKits, so start preparing already now for it.

Announcement #2: As mentioned previously, if we have any volunteers for being student representatives, please send a mail to **dlunde@kth.se**. Student representatives will have ~2 meetings with the course responsible and give feedback from the student perspective as well as discuss improvements.



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Course Structure



Module 1: C and Assembly Programming



Module 2: I/O Systems



Module 3: Logic Design (IS1500 only)

**PROJ
START**



Module 4: Processor Design



Module 5: Memory Hierarchy



Module 6: Parallel Processors and Programs

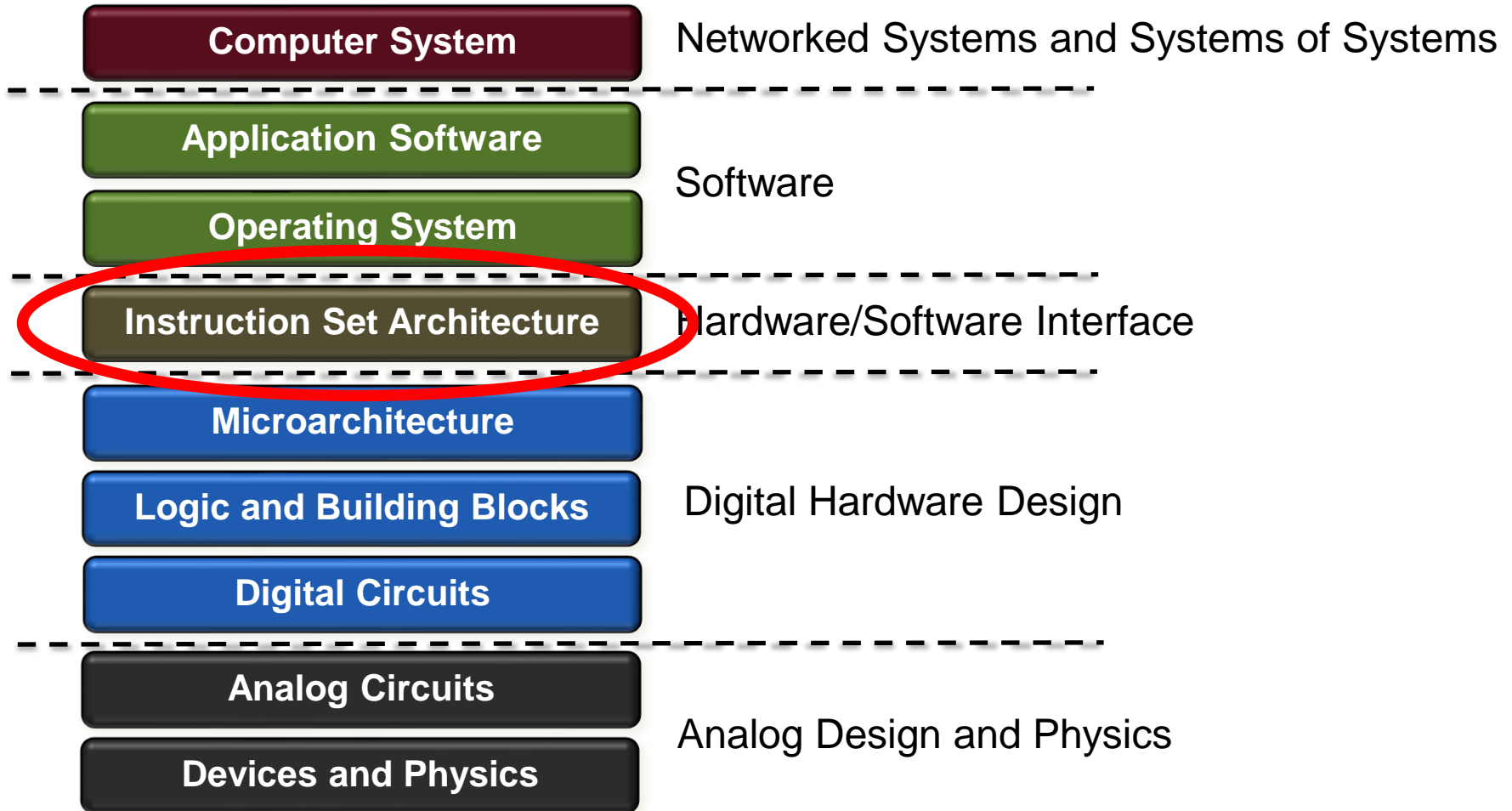


Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Abstractions in Computer Systems



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Agenda

Part I

Representing Numbers



Part II

Machine Instruction Encoding



Part III

Functions in Assembly Languages



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Part I

Representing Numbers



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Bytes, Nibbles, Words, MSB, LSB, ...

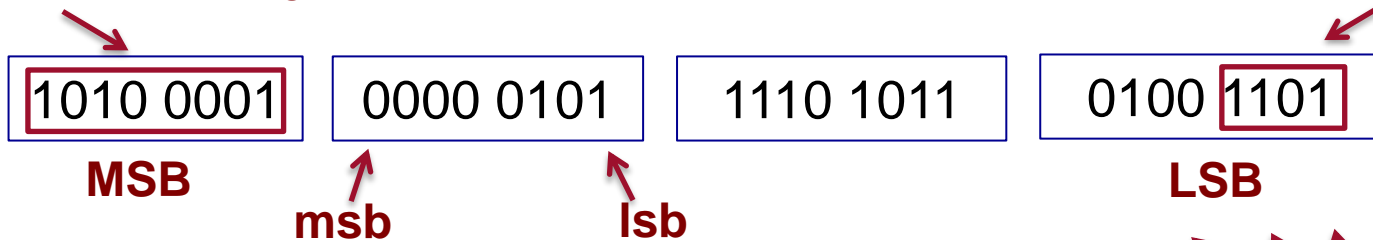
It's quite tedious to work with only binary numbers...



```
10100001000001011110101101001101
```

Therefore, bits are grouped into **bytes**, each consisting of 8 bits.

4 bits (half a byte) is called a **nibble**.



In a group of bits, the **least significant bit (lsb)** is to the right. The **most significant bit (msb)** is to the left.

Within a word, the terms are the **most significant byte (MSB)** and the **least significant byte (LSB)**.

Microprocessors use data in the size of **words**. A 32-bit processor has 32-bit words, a 64-bit processor has 64-bit words.



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Two's Complement Numbers (1/3)

How can we represent negative numbers?

Solution: **Two's complement numbers**

Which patterns can we find in the following 3-bit two's complement numbers.

Only one number represent 0



$000_2 = 0_{10}$

$001_2 = 1_{10}$

$010_2 = 2_{10}$

$011_2 = 3_{10}$

All negative numbers have msb set. This is called the *sign bit*.



$100_2 = -4_{10}$

$101_2 = -3_{10}$

$110_2 = -2_{10}$

$111_2 = -1_{10}$

Positive numbers are represented as usual.



Can represent one more negative number than positive numbers.



Negative numbers are listed in reverse order.



Two's Complement Numbers (2/3)

Why is it called two's complement?

Because the negation of an N-bit value X is $2^N - X$.

Example: $N = 3$ and $X = 3$.
Then $2^3 - 3 = 8 - 3 = 5 = 101_2$

$000_2 = 0_{10}$
$001_2 = 1_{10}$
$010_2 = 2_{10}$
$011_2 = 3_{10}$
$100_2 = -4_{10}$
$101_2 = -3_{10}$
$110_2 = -2_{10}$
$111_2 = -1_{10}$

Is binary addition working with two's complement numbers?

5 bits two's complement numbers
(check yourself)

$$\begin{aligned} 6_{10} &= 00110_2 & 7_{10} &= 00111_2 \\ -6_{10} &= 11010_2 & -7_{10} &= 11001_2 \end{aligned}$$

Should be $\rightarrow -6_{10} + 7_{10} = 1_{10} = 00001_2$

$$11010_2 + 00111_2 = 00001_2$$

Yes, it works (general proof left as an exercise). Note that the carry out is 1, but ignored here.



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages



Two's Complement Numbers (3/3)

But, in this case we need a subtract operator to get negative numbers...



Another way to take the two's complement of a number X with N bits:

- Invert all bit of number X and add 1.

Example: $N = 3$ and $X = 3$.

$X = 3 = 011_2$ $\text{inv}(X) = 100_2$,

$100_2 + 001_2 = 101_2$

(general proof omitted)

$000_2 = 0_{10}$

$001_2 = 1_{10}$

$010_2 = 2_{10}$

$011_2 = 3_{10}$

$100_2 = -4_{10}$

$101_2 = -3_{10}$

$110_2 = -2_{10}$

$111_2 = -1_{10}$



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Sign Extension

Sign extension

Assume you have a binary number **A** with **n** bits. We can sign extend **A** to have the length **m+n** bits, by copying **A**'s most significant bit into the **m** most significant bits.

Example

Assume we have value -3 encoded as a two's complement 4-bit value. Sign extend the 4-bit value into a 8-bit value.

Exercise

Assume we have value -10 encoded as a two's complement 8-bit value. Sign extend the 8-bit value into a 12-bit value.

Solution: The 12-bit value is 1111 1111 0110

Solution

The 8-bit value is
1111 1101

Exercise Sign extend the 4-bit value 2 into 8 bits. Vote for either
0000 0010 or for 1111 0010.

Solution: The 8-bit
value is 0000 0010



Part II

Machine Instruction Encoding



Part I
Representing
Numbers



Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages

Stored Programs with Instruction Encoding Formats

Stored program concept

Code is data. Code is stored in memory as any other data, enabling *general purpose computing*.

Word address

.	.	.
.	.	.
.	.	.
0040 000C	0f a0 b0 12	Word 3
0040 0008	44 93 4e aa	Word 2
0040 0004	33 fa 01 23	Word 1
0040 0000	21 a0 1b 33	Word 0

MIPS programs are typically stored from address 40 0000.

MIPS code must be word-aligned (start at addresses 0,4, 8, C etc.). X86 does not require word alignment.

For MIPS, there is 3 instruction formats:

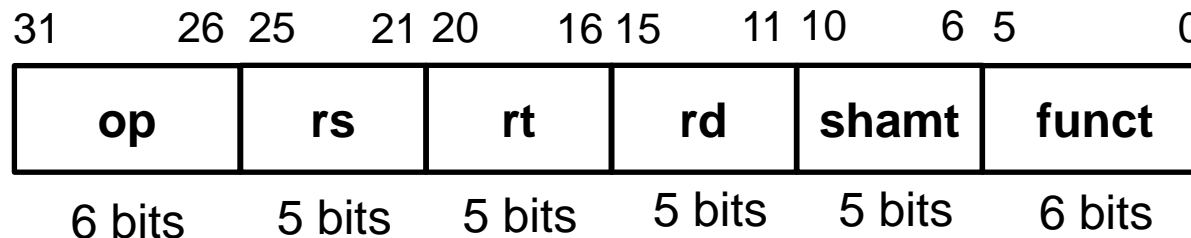
- R-Type (register-type)
- I-Type (immediate-type)
- J-Type (jump-type)

In MIPS, each instruction requires exactly one word (32 bits) of space.

R-Type Instructions

R-Type (register-type) instructions have three register operands: two sources and one destination.

“shamt” stands for “shift amount”. It is only used for shift instructions. It has value 0 for other instructions.



For most R-Type instructions, op is 0.

source
op 1

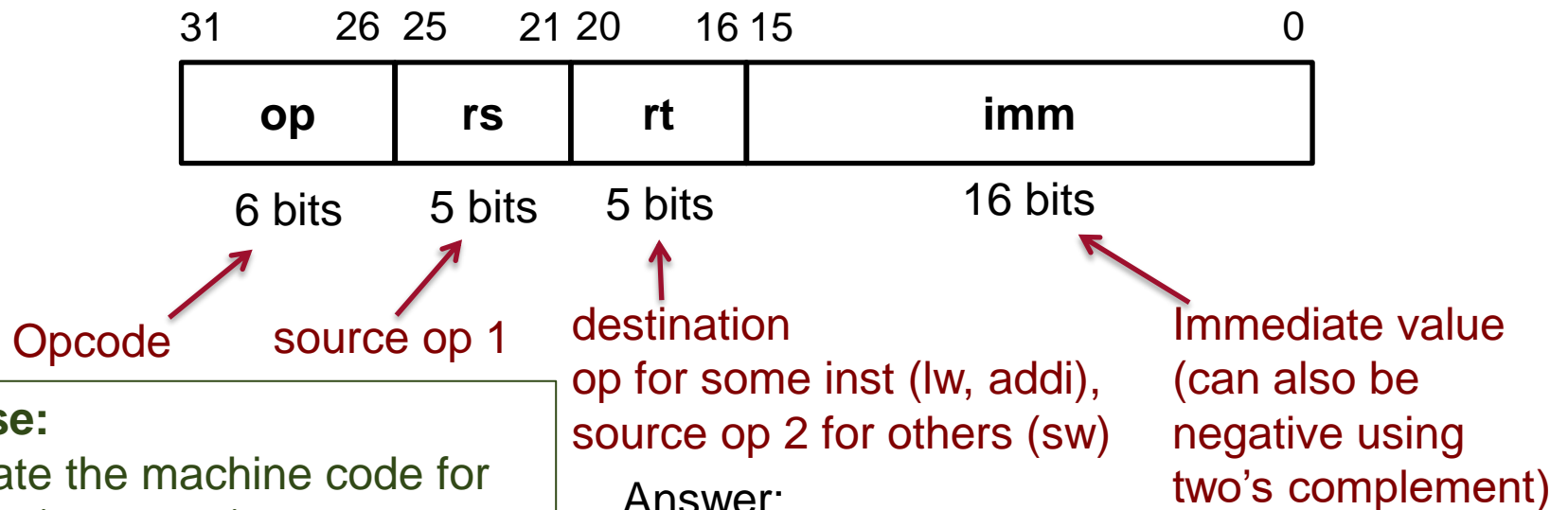
source
op 2

destination
op

funct determines
the specific R-type
instruction

I-Type Instructions

I-Type (immediate-type) instructions have two register operands and one immediate operand.



Exercise:

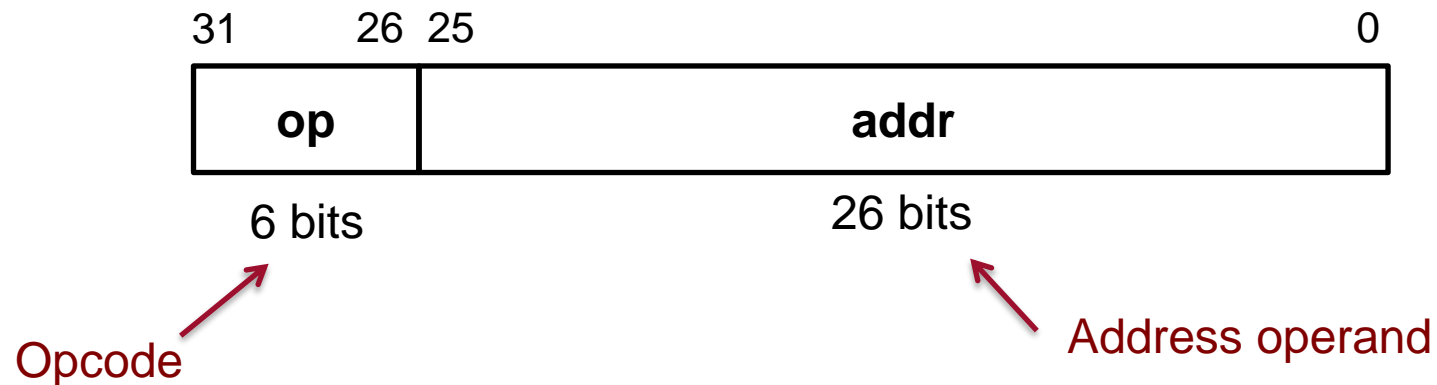
- Create the machine code for `lb $t0, -7($s1)`
Answer with a binary number.
- Same as above, but answer with a C code expression.

Answer:

- 1000 0010 0010 1000 1111 1111 1111 1001
- $(32 \ll 26) \mid (17 \ll 21) \mid (8 \ll 16) \mid (-7 \& 0xffff)$

J-Type Instructions

J-Type (jump-type) instructions has one 26-bit address operand.



Part III


Functions in Assembly Languages



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.

Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

 **Part III**
Functions in
Assembly Languages

Registers

MIPS has 32 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

Function Calls and Returns

```
int main() {
    dummy();
    ...
}

void dummy() {
}
```

The calling function is called the **caller**.
Function `main()` in this case.

The called function is called the **callee**.
Function `dummy()` in this case.

This simple example shows a function call without arguments or return values that returns immediately.

```
0x0040 0200  main:  jal dummy
0x0040 0204
...
0x0040 1040  dummy: jr $ra
```

The **jump and link** instruction (`jal`) performs two tasks:

1. It stores the address of the next instruction in the return address register **\$ra**.
2. It jumps to the target address (updates the **program counter, PC**).

What is the value of **\$ra** when entering the function `dummy`?

Answer: 0x0040 0204

The **jump registers** (`jr`) returns by jumping to the address stored in **\$ra**.

Branch Delay Slots

.global directive makes the symbol visible to other linked files.

A simple single cycle processor that executes instructions one by one

```
main:    jal    double
...
.global double
double:  add    $v0,$a0,$a0
        jr     $ra
```

A processor with **branch delay slots** (e.g., PIC32 with 5 stage pipeline) executes the next instruction before the branch is taken. Can be fixed using **nop** (pseudo instruction, “no operation”)

```
main:    jal    double
        nop
...
.global double
double:  add    $v0,$a0,$a0
        jr     $ra
        nop
```

Exercise: How can we make function double more efficient?

Trick: perform the computation in the branch delay slot.

```
main:    jal    double
        nop
...
.global double
double:  jr     $ra
        add    $v0,$a0,$a0
```

Arguments and Return Values

```
int main() {
    int y;
    y = sum(3, 8);
    ...
}
int sum(int x, int y) {
    return x + y;
}
```

```
main:
    addi $a0, $0, 3
    addi $a1, $0, 8
    jal  sum
    add  $s0, $v0, $0
    ...
sum:
    add  $v0, $a0, $a1
    jr   $ra
```

Save arguments in **\$a0** and **\$a1**.

Move the result to **\$s0** (mapped from **y**)

Save the return value in **\$v0**.

In the rest of the lecture slides, we will not use branch delay slots.

Problem: We are limited to four arguments **\$a0**, **\$a1**, **\$a2**, and **\$a3**.

The Stack (1/4)

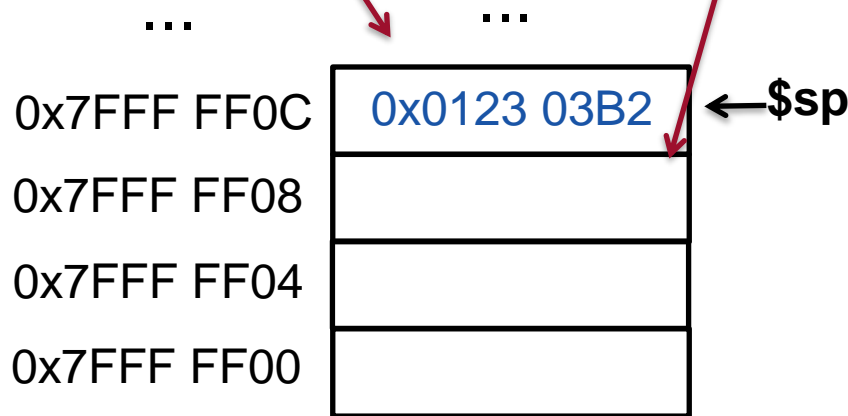
The Basic Idea

Solution: A **stack** is used to store local variables and additional arguments.

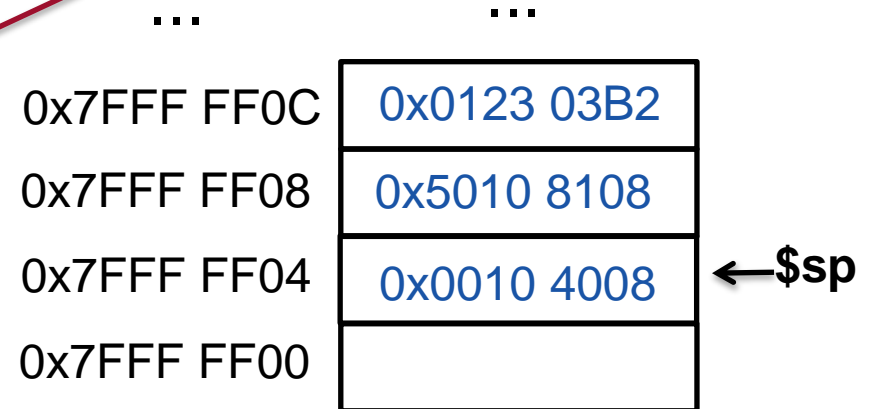
A stack is a last-in-first-out (LIFO) queue.

The stack grows traditionally down in memory.

The stack pointer points to the top of the stack.



Before adding two words to the stack.



After adding two words to the stack.

The Stack (2/4)

Push and Pop Macros

One usage of the stack is to save registers that are modified in the callee function.

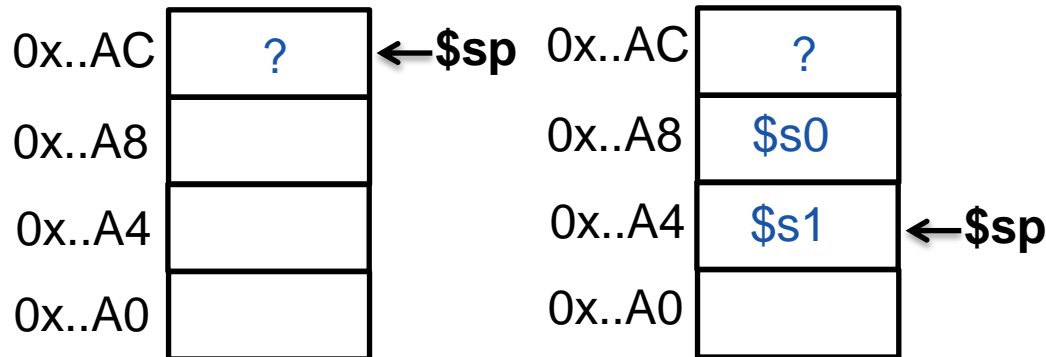
Exercise:

Assume that a callee function **foo** modifies registers `$s0` and `$s1`. Write the **foo** function in ASM, so that `$s0` and `$s1` are preserved when the function returns.

```
.macro    PUSH    (%reg)
    addi    $sp, $sp, -4
    sw      %reg, 0($sp)
.end_macro

.macro    POP    (%reg)
    lw      %reg, 0($sp)
    addi    $sp, $sp, 4
.end_macro
```

```
foo:
    PUSH    ($s0)
    PUSH    ($s1)
    ...
    modifies $s0 and $s1
    ...
    POP     ($s1)
    POP     ($s0)
    jr      $ra
```



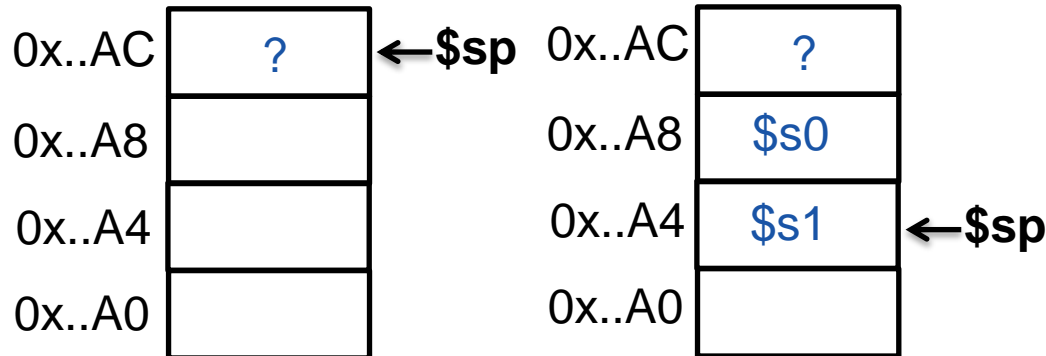
The Stack (3/4)

Preserving Registers more Efficiently

Can we make it more efficient?

Trick: we only decrement the stack pointer once, and use the index in the **sw** instruction.

```
foo:
    addi $sp, $sp, -8
    sw   $s0, 4($sp)
    sw   $s1, 0($sp)
    ...
    modifies $s0 and $s1
    ...
    lw   $s1, 0($sp)
    lw   $s0, 4($sp)
    addi $sp, $sp, 8
    jr   $ra
```





The Stack (4/4)

Preserving Registers, Continued

If the callee saves registers that are not used by the caller, this saving of registers is a waste.

The *caller* can use the save registers \$s0 to \$s7 before the function call, knowing that they are preserved when the function returns.

If the *caller* use the \$t registers and they are live before and after a function call, it must save these registers on the stack.

Name	Number	Use
\$t0-\$t7	8-15	temporary (caller-saved)
\$s0-\$s7	16-23	saved variables (callee-saved)
\$t8-\$t9	24-25	temporary (caller-saved)

The *callee* only saves the registers \$s0 to \$s7.

The *callee* can freely use the temporary registers \$t0 to \$t9, without saving them.

PC-Relative Addressing

Conditional branches compute their addresses relative to the **program counter (PC)** because the instructions have not enough address bits.

0x40 lo: add \$t1, \$a0, \$s0 ← BTA for the **bne** instruction is 0x40
 0x44 lb \$t1, 0(\$t1)
 0x48 add \$t2, \$a1, \$s0
 0x4c sb \$t1, 0(\$t2)
 0x50 addi \$s0, \$s0, 1
 0x54 bne \$t1, \$0, lo
 0x58 lw \$s0, 0(\$sp)

Calculate branch target address (BTA):

$$\text{BTA} = \text{PC} + 4 + \text{signext}(\text{imm}) * 4$$

← The imm field needs to be sign extended.

op	rs	rt	imm
5	9	0	-6
6 bits	5 bits	5 bits	16 bits
000101	01001	00000	1111 1111 1111 1010

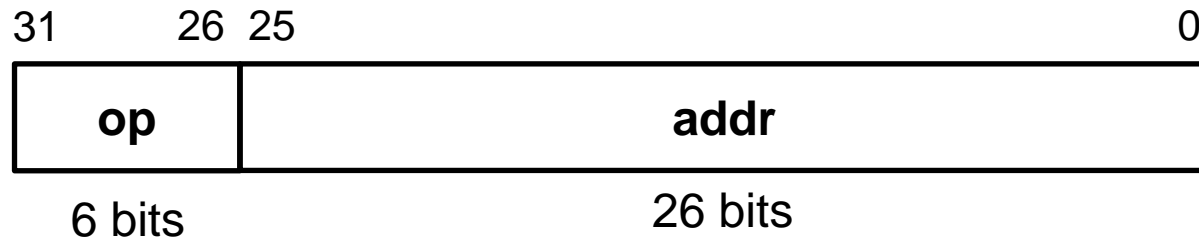
Exercise:

Compute the machine code for the **bne** instruction.



Pseudo-Direct Addressing

The **J** and **JAL** instructions are encoded using the J-type. But, the address is not 32 bits, only 26 bits.

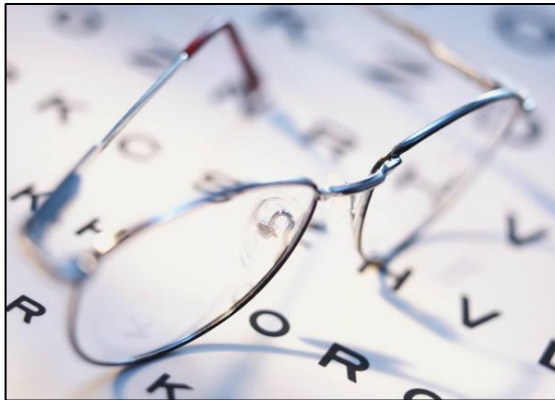


A 32-bit Pseudo-Direct Address is computed as follows:

- Bits 1 to 0 (least significant) are always zero because word alignment of code.
- Bits 27 to 2 is taken directly from the **addr** field of the machine code instruction.
- Bits 31 to 28 are obtained from the four most significant bits from $PC + 4$.

Note: **JR** instructions are not using Pseudo-Direct Addresses (it is in R-format)

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

Next lecture: C

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.

Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages



You are soon released from the lecture room...



Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages



Summary

Some key take away points:

- **Two's complement** is the standard way of representing negative and positive integer numbers.
- MIPS main instruction formats are **R-Type**, **I-Type**, and **J-Type**.
- Arguments and return values are passed in registers when calling functions. If there are too many arguments, the **stack** is used.



Thanks for listening!

Part I
Representing
Numbers

Part II
Machine
Instruction Encoding

Part III
Functions in
Assembly Languages