



EECS 388: Embedded Systems Spring 2023

Lecture 5 Instructions: Language of the Computer

Instructor: Tamzidul Hoque, Assistant Professor, Dept.
of EECS, University of Kansas (hoque@ku.edu), office:
Eaton 2038

Credits: Some slides are adopted from Henk
Corporaal's Computer Architecture and Organization
course

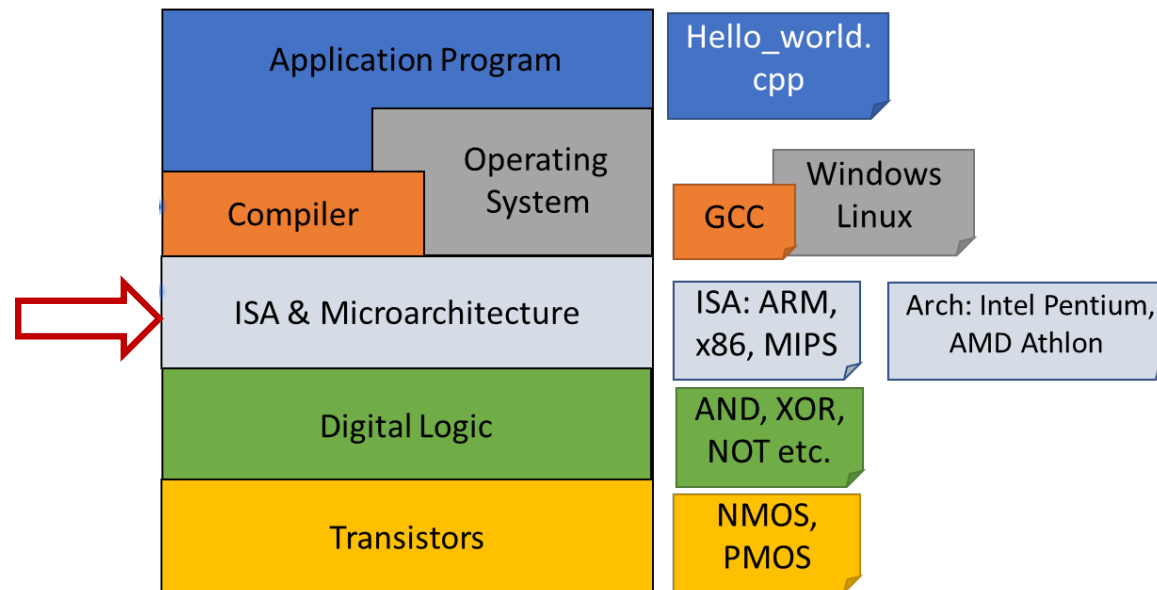




Motivation

Why we need to know
about instruction's

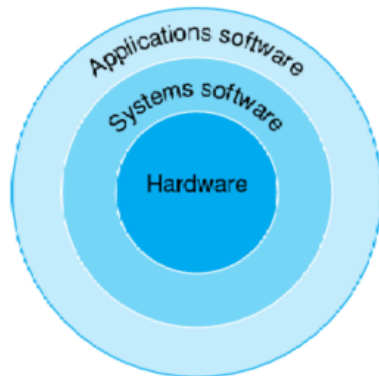
Abstractions: From Software to Silicon



ISA: Instruction Set Architecture

The Software Stack

- **Application software:** Word processor, Internet browser
- **System Software:** Operating system, compiler
 - **OS:** A supervising software that manages hardware resources to run the application software.
 - **Compiler:** Translates programs to machine executable binary
- **Hardware:** Processor, memory, IO



High-level
language
program
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

Assembly
language
program
(for MIPS)

```
swap:
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
```

Assembler

Binary machine
language
program
(for MIPS)

```
000000001010000100000000000011000
000000000000110000001100000100001
100011000110001000000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
0000000111110000000000000000001000
```

Motivation to Learn ISA

- Instructions are words used to command computer hardware
- The collection of this words, or the vocabulary is called “instruction set”

Why we need to understand this vocabulary?

Without instruction set, the design and functionality of architecture cannot be understood

Provides ability to examine code at lower level of abstractions and solve bugs/ security issues

Understanding of instruction format is required for compiler designers

Useful when working with bare metal embedded systems

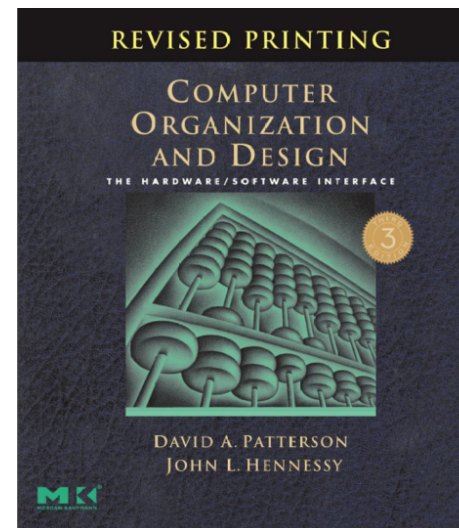
Topics

- Computer instructions & MIPS instruction set
- Arithmetic and data transfer instructions
- For details see the book (ch 2):

Computer Organization and Design,
Revised Printing, Third Edition : The
Hardware/Software Interface

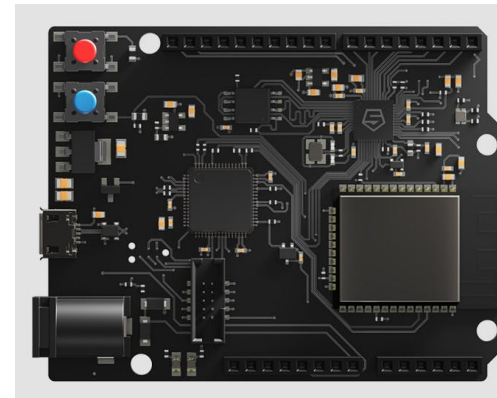
David A. Patterson, John L. Hennessy,
and Peter J Ashenden

Available online via KU Library: [Link](#)

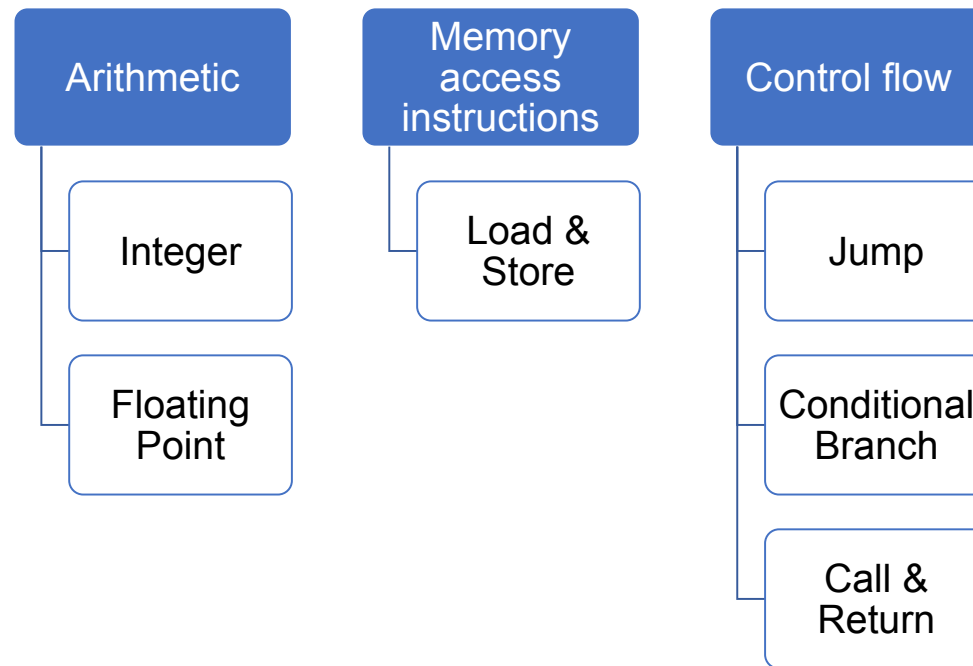


MIPS ISA

- MIPS (Microprocessor without Interlocked Pipelined Stages) MIPS Technologies, based in the United States.
- Different versions of MIPS: MIPS I, II, III, IV, and V with 32 and 64 version
- Widely adopted by the embedded market
 - HiFive1 Rev B in the lab has processor with RISC-V, which is very similar to MIPS
 - RISC-reduced instruction set computer
 - MIPS is derives from RISC-1 ISA
 - Example processors with MIPS ISA
 - PIC32, ATI/AMD Xilleon, Broadcom Sentry5, R4700 Orion



Classes of Instructions in MIPS



Opcode and Operand

- **Opcode:** operation that is executed by the CPU (ex: add, sub)
- **Operand:** data or memory location used to execute that operation.

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add a,b,c	$a = b + c$	Always three operands
	subtract	sub a,b,c	$a = b - c$	Always three operands

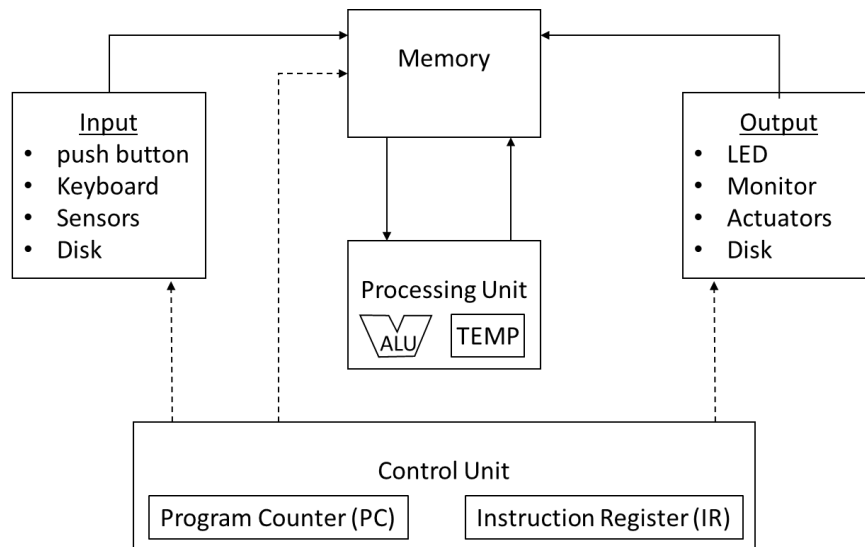
Arithmetic Instructions

- Most instructions have 3 operands
- Operand order is fixed (destination first)

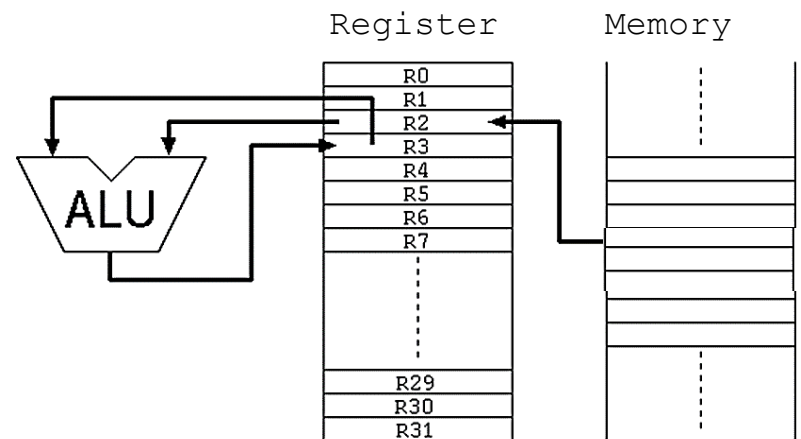
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add a,b,c	$a = b + c$	Always three operands
	subtract	sub a,b,c	$a = b - c$	Always three operands

- All arithmetic instructions in MIPS must use registers
 - CPU uses registers to store variable a, b, c

Simplified Model of Computer



The Von Neumann computer model



Registers

Registers:

- Limited number of memory location connected to ALU
- MIPS32 has 32 registers (there is a MIPS64 also)
 - Each of the 32 register has capacity to hold 32 bits (also known as word)
 - When we say MIPS in this class we assume the MIPS32

C code:

```
A = B + C;  
A = B - C;
```

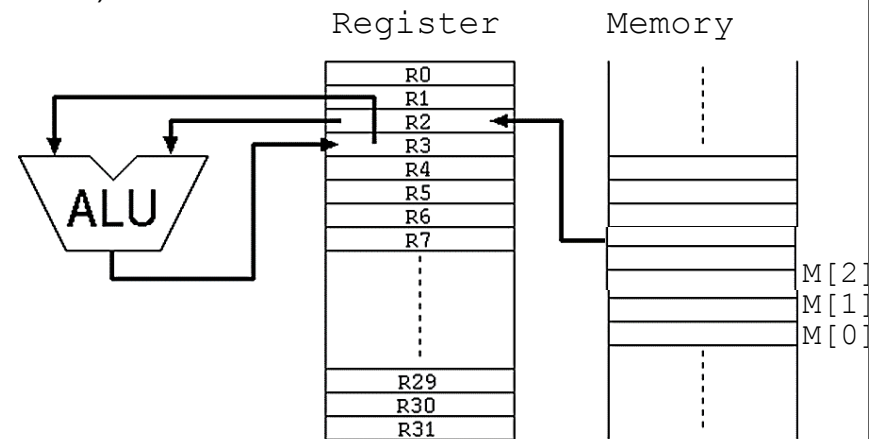
MIPS code:

```
add $r1, $r2, $r3  
sub $r1, $r2, $r3
```

Is 64 register better than 32?

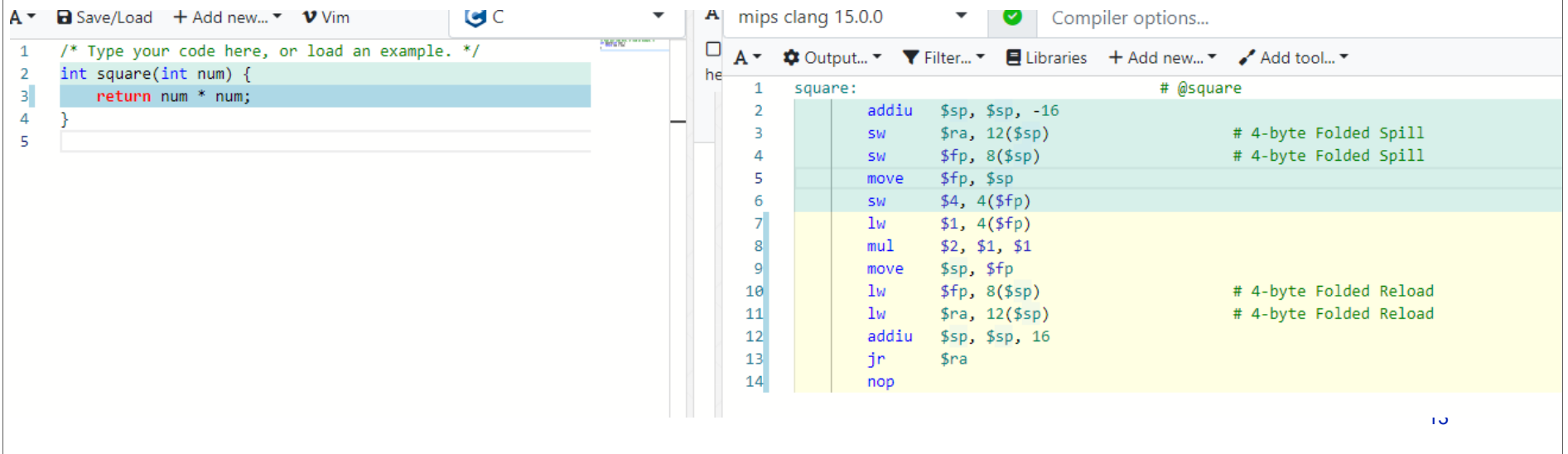
- Design Principle: *smaller is faster*.

What is the reason behind the design principle “smaller is faster”?



Register Naming

- Assembly codes use naming convention for the registers
- Helps to understand and debug assembly codes better
- <https://godbolt.org/>



The screenshot displays the Godbolt online compiler interface. On the left, the C source code is shown in a text editor with a light blue background. The code defines a function `square` that takes an integer `num` and returns its square. On the right, the generated MIPS assembly code is displayed, color-coded to match the C code. The assembly code includes comments for stack frame management and register usage. The assembly code is as follows:

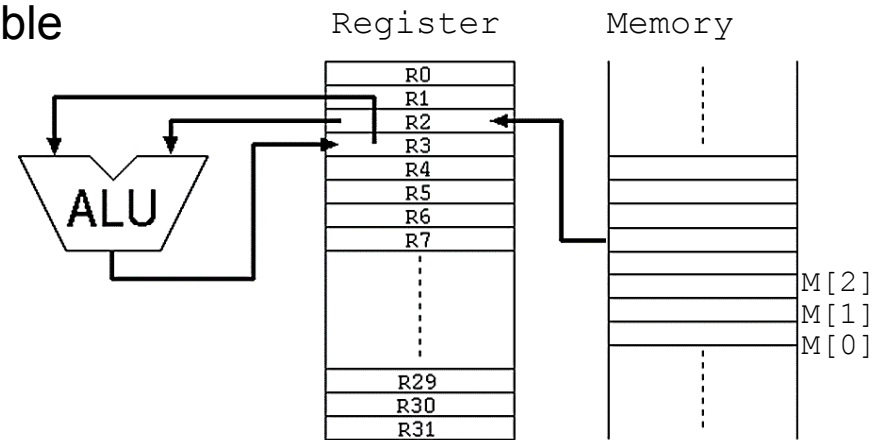
```
1 square:                                     # @square
2     addiu   $sp, $sp, -16
3     sw      $ra, 12($sp)                    # 4-byte Folded Spill
4     sw      $fp, 8($sp)                     # 4-byte Folded Spill
5     move    $fp, $sp
6     sw      $4, 4($fp)
7     lw      $1, 4($fp)
8     mul     $2, $1, $1
9     move    $sp, $fp
10    lw      $fp, 8($sp)                     # 4-byte Folded Reload
11    lw      $ra, 12($sp)                    # 4-byte Folded Reload
12    addiu   $sp, $sp, 16
13    jr      $ra
14    nop
```

Register Naming in MIPS

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved (by callee)
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Memory

- Your program can have large number of variable and complex data types
 - Ex: Arrays with 100 elements
- 32 or 64 registers are not sufficient
- We use large memory to store this data
- But we cannot perform arithmetic operations directly from memory
 - Use load (lw) to bring data to register from memory



C code:

```
A = B + C;
A = B - C;
```

MIPS code:

```
add M[2], M[1], M[0]
sub M[2], M[1], M[0]
```

C code:

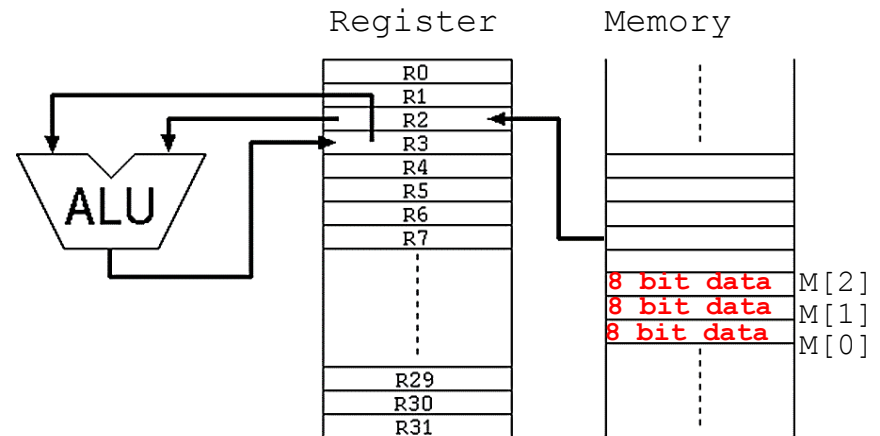
```
A = B + C;
```

MIPS code:

```
lw  $t0, M[1]      //B is in M[1]
lw  $t1, M[0]      //C is in M[0]
add $t1, $t1, $t0  // $t1=A
```

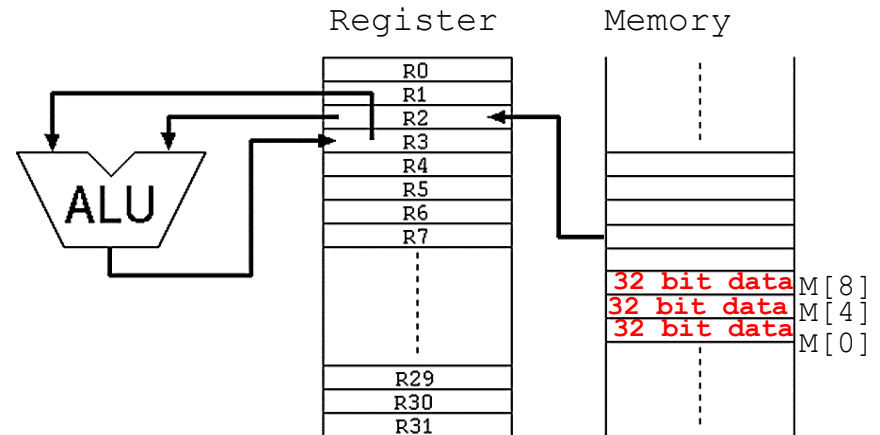
Complexities with Memory

- Your program can have large number of variable and complex data types
 - Ex: Arrays with 100 elements
- 32 or 64 registers are not sufficient
- We use large memory to store this data
- Memory is viewed as a large, single-dimension array, each with an address
 - A memory address is an index into the array (say M, where M[0], M[1], M[2] etc. exists)
 - Each address is holding 8 bits or 1 byte
 - But most variables we use is beyond 8 bits, the register also holds 32 bits



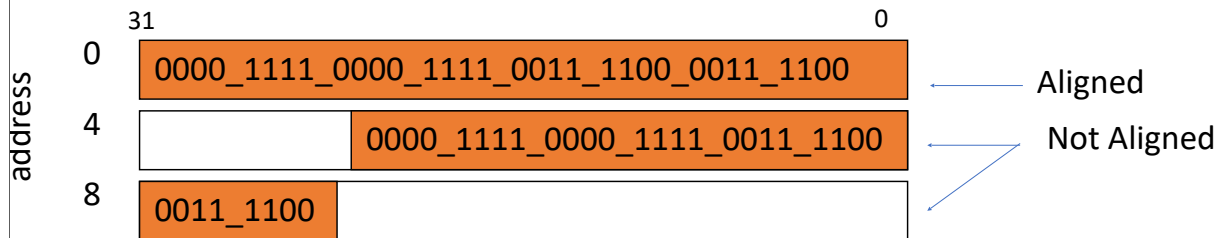
Memory Organization

- Most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.
- Address of a sequence of words differ by 4
 - $M[0]$, $M[4]$, $M[8]$



- Alignment Restriction in MIPS
 - Words must start at an address that are multiples of 4

$A = 0000_1111_0000_1111_0011_1100_0011_1100$



Instructions: load and store

Example:

- Let us assume that we want to add 8th element of array **A** with variable **h**. Each element of array is in a word address starting from the base address.
- Register **\$s1** has the base address of array **A**. Variable **h** is in **\$s2**.

• C code: `g= h + A[7]; //first element is A[0]`

• MIPS code: `lw $t0, 28($s1) //32 because 8 elements`
`add $t0, $s2, $t0`

MIPS operands

Name	Example	Comments
32 registers	<code>\$s0, \$s1, . . . ,</code> <code>\$t0, \$t1, . . .</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2 ³⁰ memory words	<code>Memory[0],</code> <code>Memory[4], . . . ,</code> <code>Memory[4294967292]</code>	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Instructions: load and store

Example:

- Store the result back to A[12]
 - Register **\$s1** has the base address of array **A**.
 - Store instruction: **sw \$source_reg \$dest_mem_address**

- MIPS code:

```
lw    $t0, 28($s1)    //32 because 8 elements
add   $t0, $s2, $t0
sw    $t0, ?($s1)
```

.

Practice

Example:

- C code: `A[10] = y + A[11];`
- Register `$s1` has the base address of array `A`. Variable `y` is in `$s2`.
- Write the MIPS code

So far we've learned:

- MIPS
 - loading words but addressing bytes
 - arithmetic on registers only

- Instruction

Meaning

add \$s1, \$s2, \$s3	// \$s1 = \$s2 + \$s3
sub \$s1, \$s2, \$s3	// \$s1 = \$s2 - \$s3
lw \$s1, 100(\$s2)	// \$s1 = Memory[\$s2+100]
sw \$s1, 100(\$s2)	// Memory[\$s2+100] = \$s1

Machine Language: R-type instr

- Instructions, like registers and words of data, are also 32 bits long

- Example: `add $t0, $s1, $s2`
- Registers have numbers: `$t0=9, $s1=17, $s2=18`

- Instruction Format:

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>funct</i>
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *op*: Basic operation of the instruction, traditionally called the **opcode**.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (Section 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
- *funct*: Function. This field selects the specific variant of the operation in the *op* field and is sometimes called the *function code*.

Machine Language: I-type instr

- Introduce a new type of instruction format
 - I-type for data transfer instructions
 - other format was R-type for register
- Example: lw \$t0, 32(\$s2)

35	18	9	32
----	----	---	----

op	rs	rt	16 bit number
----	----	----	---------------

Translating Instruction

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?