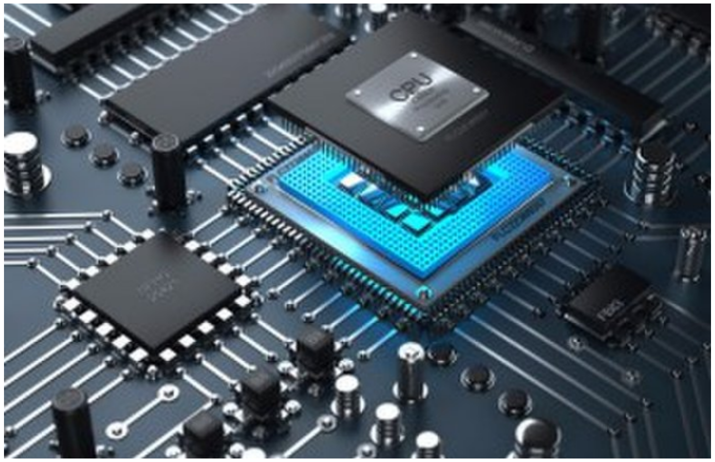


CPU Instructions and Procedure Calls

CS 154: Computer Architecture
Lecture #5
Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- Lab 02 – due today!
- Lab 03 – stay tuned…

Lecture Outline

- MIPS instruction formats
- Refresher on some other MIPS instructions and concepts

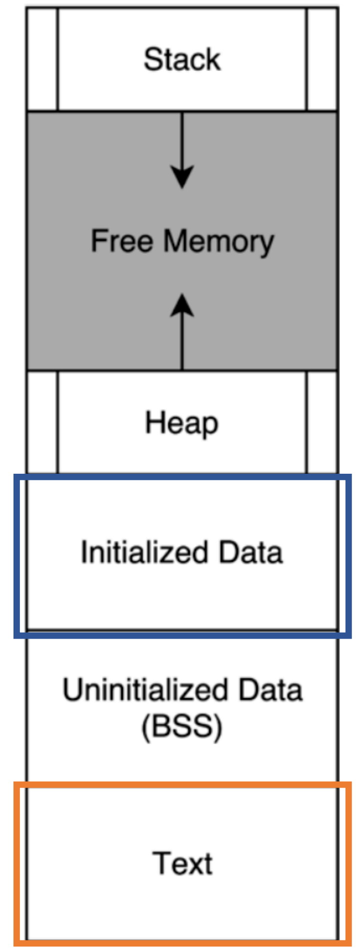
Reference material from CS64 – I'll be going over this a little fast…

Example
What does this do?

```
.data
name: .asciiz “Lisa speaks ”
rtn: .asciiz “ languages!\n”
age:  .word 7

.text
main:
      li $v0, 4
      la $a0, name   # la = load memory address
      syscall

      la $t2, age
      lw $a0, 0($t2)
      li $v0, 1
      syscall

      li $v0, 4
      la $a0, rtn
      syscall

      li $v0, 10
      syscall
```

What goes in here? →

What goes in here? →



.data Declaration Types
w/ Examples

```
var1:    .byte 9          # declare a single byte with value 9
var2:    .half 63         # declare a 16-bit half-word w/ val. 63
var3:    .word 9433       # declare a 32-bit word w/ val. 9433
num1:    .float 3.14      # declare 32-bit floating point number
num2:    .double 6.28     # declare 64-bit floating pointer number
str1:    .ascii "Text"    # declare a string of chars
str3:    .asciiz "Text"   # declare a null-terminated string
str2:    .space 5         # reserve 5 bytes of space (useful for arrays)
```

These are now reserved in memory and we can call them up by loading their memory address into the appropriate registers.

Integers in MIPS

Unsigned 32-bits

- Range is **0 to +2^{32} – 1** **(or +4,294,967,295)**
- Remember positional notation!
  - For when converting to decimal – remember LSB is position **<u>0</u>**
  - Example: What is 0x00881257 in decimal?
  - Answer: $7 + 2^4 + 2^6 + 2^9 + 2^{12} + 2^{19} + 2^{23} = 8{,}917{,}591$

Integers in MIPS

Signed (2s Complement) 32-bits

- Range is **-2^{31} to $+2^{31} - 1$**
- Remember the 2s complement formula!
  - Negate all bits and then add 1
  - Example: What is 0xFFFE775C in decimal?
  - Answer: negative 0x000188A4

$$= -(4 + 2^5 + 2^7 + 2^{11} + 2^{15} + 2^{16})$$

$$= -10,0516$$

Signed Integers in MIPS

- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

- Representing a number using more bits
  - You want to preserve the numeric value
  - Example: **+6** in 4-bits (0110) becomes 00000110 in 8-bits
  - Example: **-6** in 4-bits (1010) becomes 11111010 in 8-bits
  - When does this happen in MIPS?
    - Think of I-type instructions

MIPS Instructions: Syntax

<op> <rd>, <rs>, <rt>

op : operation

rd : register destination

rs : register source

rt : register target

<op> <rt>, <rs>, immed

op : operation

rs : register source

rt : register target

MIPS Instruction Formats

Recall:

- There are **three** different ***instruction formats***: **R**, **I**, **J**
- ALL core instructions are 32 bits long

	6 b	*5 b*	*5 b*	*5 b*	*5 b*	*6 b*
R-Type	op	rs	rt	rd	shamt	funct
I-Type	op	rs	rt	address		
	6 b	*5 b*	*5 b*	*16 b*		

Instruction Representation in R-Type

op	rs	rt	rd	shamt	funct
6 b	5 b	5 b	5 b	5 b	6 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 11*	*10 – 6*	*5 – 0*

- The combination of the **opcode** and the **funct** code tell the processor what it is supposed to be doing
- Example:

`add $t0, $s1, $s2` 0x02324020

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

op = 0, funct = 32 (0x20) means "add"

rs = 17 means "$s1"

rt = 18 means "$s2"

rd = 8 means "$t0"

shamt = 0 means this field is unused in this instruction

Instruction Representation in I-Type

op	rs	rt	address
6 b	5 b	5 b	16 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 0*

- Example:

addi $t0, $s0, 124 0x2208007C

op	rs	rt	address/const
8	16	8	124

op = 8	mean "addi"
rs = 16	means "$s0"
rt = 8	means "$t0"
address/const = 124 (0x007C)	is the 16b immediate value

Worth checking out: https://www.eg.bucknell.edu/~csci320/mips_web/

Pseudoinstructions

- Instructions that are NOT core to the CPU
- They're "macros" of other actual instructions
- Often they are slower (higher CPI) than core instructions

- Examples:

```
    li $t0, C
```

Is a macro for:

```
    lui $t0, C_hi
    ori $t0, $t, C_lo
```

```
    move $t0, $t1
```

Is a macro for:

```
    addu $t0, $zero, $t1
```

https://github.com/MIPT-ILab/mipt-mips/wiki/MIPS-pseudo-instructions *has more examples*

Bitwise Operations

Operation	C/C++	MIPS
Shift left	<<	sll
Shift right	>>	srl, sra
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori
Bitwise NOT	~	nor*
Bitwise XOR	^	xor

* Specifically, **`nor $t0, $t0, 0`** is equivalent to **`not(t0)`**

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- **`beq rs, rt, L1`** often used with **`slt, slti`**
  - if (rs == rt) branch to instruction labeled L1;

- **`bne rs, rt, L1`** often used with **`slt, slti`**
  - if (rs != rt) branch to instruction labeled L1;

- MIPS also has the pseudoinstructions: **`ble`**, **`blt`**, **`bge`**, **`bgt`**
  - But pseudoinstructions run slower…

- **`j L1`**
  - Unconditional jump to instruction labeled L1

Example

- C/C++ code: `while (save[i] == k) i += 1;`
- Given: var `i` in `$s3`, k in `$s5`, address of save in `$s6`

- In MIPS:

```
Loop:
      sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: …
```

Procedure Calls (aka Calling Functions)

- Procedure call: jump and link

```
jal FunctionLabel
```

  - Address of following instruction put in $ra
  - Jumps to target address

- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Calling Nested or Recursive Functions

- What happens when you have a saved return address in $ra….
  … and then you call ANOTHER function?
- We have to use a standardized way of calling functions
  - The MIPS Calling Convention
- Especially important when different dev. teams are making different functions in a project
  - Also simplifies program testing
- Some registers will be presumed to be “preserved” across a call ; Others will not

The MIPS Calling Convention In Its Essence

- Remember: **Preserved** vs **Unpreserved** Regs
- **Preserved**: **\$s0 - \$s7**, and **\$ra**, and **\$sp (by default)**
- **Unpreserved**: **\$t0 - \$t9**, **\$a0 - \$a3**, and **\$v0 - \$v1**

- Values held in **Preserved Regs** immediately before a function call MUST be the same immediately after the function returns.
  - Use the **stack memory** to save these

- Values held in **Unpreserved Regs** must always be assumed to change after a function call is performed.
  - \$a0 - \$a3 are for passing arguments into a function
  - \$v0 - \$v1 are for passing values from a function

YOUR TO-DOs for the Week

- Readings!
  - Chapters 2.10 – 2.13

- Stay Tuned for Lab Assignment!
  - Will be announced on Piazza

</LECTURE>