CS214 : MIPS Assembly Programming
Assignment 1: Sequential Construct-I Tutorial
Integer Arithmetic and Logical Instructions

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

August 23, 2023

Sequential Construct

The programming requires a dividing a task, into small unit of work. These unit of work are represented with programming construct that represents part of task. In the sequential construct, the designated task is broken into smaller task one follow by another.

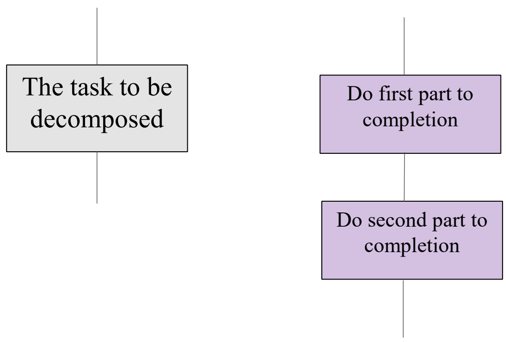

Figure 1: Representational view of Sequential Construct

Arithmetic Instructions

Following are the set of arithmetic instruction that is to be used in this assignment.
In all the list of instruction, \$1, \$2 and \$3 represent the registers for the understanding purposes. In the assignment, you have to use the register name not the corresponding register number. Note that the details and list of the register is already provided in the instruction manual.

Instruction	Example	Meaning	Comments
add	add $1, $2, $3	$1=$2+$3	
subtract	sub $1, $2, $3	$1=$2-$3	
add immediate	addi $1,$2,100	$1=$2+100	"immediate" means a constant number
add unsigned	addu $1,$2,$3	$1=$2+$3	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	subu $1,$2,$3	$1=$2-$3	Values are treated as unsigned integers, not two's complement integers
add immediate unsigned	addiu $1,$2,100	$1=$2+100	Values are treated as unsigned integers, not two's complement integers
Multiply (without overflow)	mul $1,$2,$3	$1=$2*$3	Result is only 32 bit
Multiply	mult $2,$3	$hi,$low=2∗3	Upper 32 bits stored in special register hi Lower 32 bits stored in special register lo
Divide	div $2,$3	$hi,$low=$2/$3	Remainder stored in special register hi Quotient stored in special register lo
Unsigned Divide	divu $2,$3	$hi,$low=$2/$3	$2 and $3 store unsigned values. Remainder stored in special register hi Quotient stored in special register lo

Table 1: Arithmetic Instruction with their details and explanations

Logical Instructions

Following are the set of logical instructions that is to be used in this assignment.

Instruction	Example	Meaning	Comments
and	and $1, $2, $3	$1=$2&$3	Bitwise AND
or	or $1, $2, $3	$1=$2—$3	Bitwise OR
and immediate	andi $1,$2,100	$1=$2&100	Bitwise AND with immediate value
or immediate	ori $1,$2,100	$1=$2—100	Bitwise OR with immediate value
nor	nor $1,$2,$3	$1=$2↓$3	Bitwise NOR
shift left logical	sll $1,$2,10	$1=$2<<10	Shift left by constant number of bits
shift right logical	srl $1,$2,10	$1=$2>>100	Shift right by constant number of bits

Table 2: Logical Instructions with their details and explanations

Data Movement Instructions

Following are the set of data movement instructions that is to be used in this assignment.

Instruction	Example	Meaning	Comments
load word	lw \$1, 100(\$2)	\$1=Memory[\$2+100]	Copy from memory to register
store word	sw \$1, 100(\$2)	Memory[\$2+100]=\$1	Copy from register to memory
load upper immediate	lui \$1, 100	\$1=100x2^16	Load constant into upper 16bits. Lower 16bits are set to zero.
load address	la \$1, label	\$1=Address of label	Pseudo-instruction (provided by the assembler, not processor!) Load computed address of label (not its contents) into register
load immediate	li \$1, 100	\$1=100	Loads immediate value into register
move from hi	mfhi \$2	\$2=hi	Copy from special purpose register hi to general register
move from lo	mflo \$2	\$2=lo	Copy from special purpose register lo to general register
move	move \$1,\$2	\$1=\$2	Copy from register to register

Table 3: List of Data Movement Instruction with their details and explanations

Note: Variation on load and store also exist for smaller data sizes.

1. 16-bit halfword: lh and sh
2. 8-bit byte: lb and sb

System Calls

The SPIM provide a large number of system call. These are the call to Operating System and do not represent MIPS process instruction. These call are either implemented by the OS or standard library.

System calls are used for input, output and to exit the program. These calls are commences with the help of *syscall* function. To use the instruction, the appropriate arguments in registers \$v0, \$a0-\$a1, or \$f12 are supplied depending on the specific call required. Following are the list of system calls that are to be required in this assignment.

Service	Operation	Code (in \$v0)	Arguements
exit	stop program from running	10	none
exit2	stop program from running and return an integer	17	\$a0=result (integer number)

Table 4: List of System Calls with their usage and explanations

Instruction	Example	Meaning	Comments
Arithmetic Instructions			
add	add.s $f0, $f1, $f2	$f0=$f1+$f2	none
subtract	sub.s $f0, $f1, $f2	$f0=$f1-$f2	none
multiply	mul.s $f0, $f1, $f2	$f0=$f1*$f2	none
division	div.s $f0, $f1, $f2	$f0=$f1/$f2	none
absolute	abs.s $f0, $f1	$f0=—$f1—	Absolute Value of floating point number
negative	neg.s $f0, $f1	$f0=-$f1	Negate the floating point number
Data Movement and Conversion Instructions			
load float	l.s $f0, 100($t2)	$f0=Mem[$t2+100]	Copy from Memory to Floating Point Register
store float	s.s $f0, 100($t2)	Mem[$t2+100]=$f0	Copy from Floating Point Register to Memory
load float immediate	li.s $f0, 10.0	$f0=10.0	Load Floating point immediate value into Register
move	move.s $f0, $f1	$f0=$f1	Copy from register to register
convert to integer	cvt.w.s $f2, $f4	$f2=$f4	Convert from single precision FP (f4) to integer (f2)
convert to float	cvt.s.w $f2, $f4	$f2=$f4	Convert from integer (f2) to single precision (f4)

Table 1: Floating Point Instructions with their details and explanations

System Call Related to I/O

System calls are used for input, output and to exit the program. These calls are commences with the help of *syscall* function. To use the instruction, the appropriate arguments in registers $v0, $a0-$a1, or $f12 are supplied depending on the specific call required. The system call will return the result values into the register based on the datatype and operation conducted.

Following are the set of system call that is to be used in this assignment.

Service	Operation	Code (in $v0)	Arguement	Results
print_int	Print integer number (32 bit)	1	$a0=integer to be printed	none
print_float	Print floating-point number (32 bit)	2	$f12=float to be printed	none
print_double	Print floating-point number (64 bit)	3	$f12=float to be printed	none
print_string	Print null-terminated character string	4	$a0=address of string in memory	none
read_int	Read integer number from user	5	none	integer written in $v0
read_float	Read floating-point number from user	6	none	float written in $f0
read_double	Read double floating point number from user	7	none	double written in $f0
read_string	Work the same as standard C library fgets()	8	$a0=memory address of string input buffer $a1=length of string buffer (n)	none
sbrk	Returns the address to a block of memory containing n additional bytes (dynamic memory allocation)	9	$a0=amount	address in $v0
print_char	Print Character	11	$a0=character to be printed	none
read_char	Read Character from user	12	none	char written in $v0

Table 2: List of System Calls with their usage and explanations

Comparison Instructions

Following are the set of comparison instructions that is to be used in this assignment.

Instruction	Example	Meaning	Comments
set on less than	slt \$t1, \$t2, \$t3	if(\$t2<\$t3)\$t1=1; else \$t1=0	Test if less than. If true, set \$t1 to 1. Otherwise set \$t1 to 0
set on less than immediate	slti \$t1, \$t2, 100	if(\$t2<100)\$t1=1; else \$t1=0	Test if less than. If true, set \$t1 to 1. Otherwise set \$t1 to 0
set equal	seq \$t1, \$t2, \$t3	if(\$t2=\$t3)\$t1=1; else \$t1=0	Test if equal. If true, set \$t1 to 1. Otherwise set \$t1 to 0
set greater than equal	sge \$t1, \$t2, \$t3	if(\$t2>=\$t3)\$t1=1; else \$t1=0	Test if greater than equal. If true, set \$t1 to 1. Otherwise set \$t1 to 0.
set greater than	sgt \$t1, \$t2, \$t3	if(\$t2>\$t3)\$t1=1; else \$t1=0	Test if greater than. If true, set \$t1 to 1. Otherwise set \$t1 to 0.
set less than equal	sle \$t1, \$t2, \$t3	if(\$t2<=\$t3)\$t1=1; else \$t1=0	Test if less than equal. If true, set \$t1 to 1. Otherwise set \$t1 to 0.
set not equal	sne \$t1, \$t2, \$t3	if(\$t2! =\$t3)\$t1=1; else \$t1=0	Test if not equal. If true, set \$t1 to 1. Otherwise set \$t1 to 0.

Table 3: List of Comparison Instructions with their details and explanations

Instruction	Example	Meaning	Comments
		Integer Conditional Instructions	
branch on equal	beq \$1, \$2, 1000	if(\$1 == \$2) go to PC+4+1000	Test if registers are equal
branch on not equal	bne \$1, \$2, 1000	if(\$1 != \$2) go to PC+4+1000	Test if registers are not equal
branch on greater than	bgt \$1, \$2, 1000	if(\$1 >\$2) go to PC+4+1000	Test if one register is greater than compared to other
branch on greater than or equal	bge \$1, \$2, 1000	if(\$1 >= \$2) go to PC+4+1000	Test if one register is greater than or equal to other
branch on less than	blt \$1, \$2, 1000	if(\$1 <\$2) go to PC+4+1000	Test if one register is less than compared to other
branch on less than or equal	ble \$1, \$2, 1000	if(\$1 <= \$2) go to PC+4+1000	Test if one register is less than or equal to other
		Floating Point Comparison and Conditional Instructions	
Equal Comparison	c.eq.s \$f2, \$f4	if(\$f2 == \$f4) set code = 1 else code = 0	Test if floating point registers are equal
Less than or Equal to Comparison	c.le.s \$f2, \$f4	if(\$f2 <= \$f4) set code = 1 else code = 0	Test if one floating point register is less than to equal to another one
Lesst than Comparison	c.lt.s \$f2, \$f4	if(\$f2 <\$f4) set code = 1 else code = 0	Test if one floating point register is less than to another one
Greater than or Equal to Comparison	c.ge.s \$f2, \$f4	if(\$f2 >= \$f4) set code = 1 else code = 0	Test if one floating point register is greater than or equal to another one
Greater than Comparison	c.gt.s \$f2, \$f4	if(\$f2 >\$f4) set code = 1 else code = 0	Test if one floating point register is greater than another one
branch on set code	bclt label	if code == 1 then jump to label	Jump to the label if code is set
branch on reset code	bclf label	if code == 0 then jump to label	Jump to label if code is reset

Table 1: List of Conditional Branch Instructions with their details and explanations

Problem 1: Write a MIPS assembly program that takes two number (can be anything floating point or integer) as an input and print maximum between two of them as follows:

```
32.6 is greater than 25.0
```

Problem 2: Write an assembly program that takes year as an input from the user and check whether the input year is leap year or not. If it is leap year prompt the message

```
Input year is a leap year
```

Otherwise, prompt the message

```
Input year is not a leap year
```

Problem 3: Write an assembly program that determines whether the student is allowed to sit the examination provided his/her attendance is 75%. For the given problem statement, the MIPS assembly program takes the following input: The name of student, Total number of class held and Total class attended by the student. The output format is as follows:

```
Ajay is allowed to sit in the exam.
or
Ajay is not allowed to sit in the exam.
```

Problem 4: Write an MIPS assembly program that takes the marks of the student as an input (in the range of 1-100) and assign the grade. The grading policy are as follows:

```
Grade: A if marks >= 80
Grade: B if 80 < marks >= 60
Grade: C if 60 < marks >= 40
Grade: F otherwise
```

CS214 : MIPS Assembly Programming Tutorial: Iterative Constructs

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE, IIT Mandi

September 9, 2023

Iterative Construct

The programming requires a dividing a task, into small unit of work. These unit of work are represented with programming construct that represents part of task. The iterative construct is used if the designated task consists of doing a sub-task a number of times, but only as long as some condition is true.

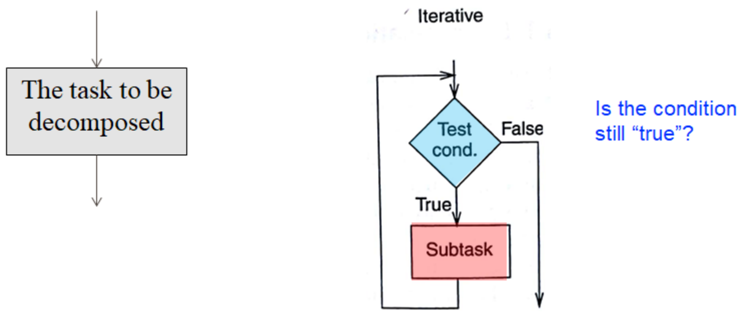

Figure 1: Representational view of Iterative Construct

Iterative Jump Instructions

MIPS provides four iterative jump instructions. All iterative instructions unconditionally jump into a specific address determined within the instruction. It is much easier to use a label for the jump instructions instead of an absolute number. For example, `j loop`, the label `loop` here should be defined somewhere else in the code.

Instruction	Example	Meaning	Comments
jump	`j 2000`	Go to address 2000	Jump to target address
jump register	`jr $2`	Go to address stored in $2	For switch, procedure return
jump and link	`jal 2000`	$ra=PC+4; go to address 2000	Use when making procedure call. This save the return address in $ra.
jump and link register	`jalr $2`	$ra=PC+4; go to address stored in $2	Use when making procedure call and return

Table 1: List of Iterative Jump Instructions with their details and explanations

CS214 : MIPS Assembly Programming
Tutorial 7: Functions

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

October 8, 2023

Functions

Functions are required to utilize the frequently accessed code, make a program more modular and readable and easier for debugging. Execution of a function change the control flow of the program two times: one at the time of calling the function and other at the time of returning from the function.

```
void main()
{
  int y, z;
  y = sum(42, 7);
  z = sum(10, -8);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Figure 1: Function Code Example in C

In the above example, the `main` function invokes the function `sum` twice and the function `sum` return two times, but at the different control point in the `main`. Note that each time the `sum` invoke, the control flow has to remember the appropriate return address.

MIPS uses the jump and link instruction `jal` (format details and example are already given in Assignment-5) to invoke the function

- The `jal` store the return address (which is the address of the next instruction in the control flow of `main`) into the dedicated register `$ra`, before changing the control flow to called function.
- It is the only instruction that can access the value in the program counter. Hence it can easily store the return address `PC + 4` of the caller function in `$ra`.

To transfer control back to caller function, the callee function has to jump to address provided by the `$ra` using the following instruction: `jr $ra`.

Function accept some number arguments and operate upon them and produce return values. For example, in the above code snippet, the values 42 and 7 in the function `sum` are the actual argument and the variables a and b are the formal argument. The function return the sum of a and b as a return value.

MIPS uses the following rules for the function arguments and the return values:

- With MIPS, upto four arguments can be passed by using only argument register $a0 - a3$ before invoking the function with `jal` command.
- A callee function can return upto two values using the register $v0 - v1$, before returning via `jr`.

Note that these above conventions are not enforced by the assembler or hardware, but it will be easy for different programmers to interface with the written code.

A Note about data types of the arguments passed in the function

- MIPS assembly language is untyped, means there is no distinction between integer, float, characters or pointers passed through argument.
- It is the assembly programmer job to type check different variable argument passed in the function. In other words, programmer need to make sure that argument value(s) and return value(s) are consistent to the program.

CS214: MIPS Assembly Programming
Tutorial 8: Stacks

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

October 28, 2023

Stacks

The stack is a memory area used to save local variables. A certain chunk of main memory is reserved for the stack, called stack space/area in the MIPS machine. The following points need to be considered while working with stack in MIPS:

- The stack expands downwards in terms of memory addresses.
- The stack's top element's memory address is stored in the special purpose register, called Stack Pointer (`sp`).
- MIPS does not provide any `push` and `pop` statement. Instead, it will be explicitly handled by the MIPS assembly programmer.

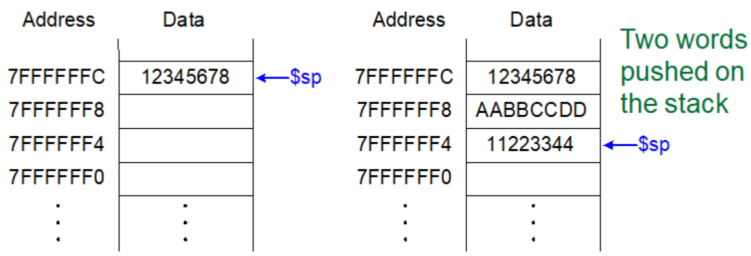

Figure 1: Representational view of Stack

Pushing an Element in the stack In order to place the data or address element in the stack, the following are the two steps that are necessary to follow:

- Progress the stack pointer, `sp` to the down to make space for the newly added element.
- Store the element into the stack.

Following are the two sample example code to push the data elements from the register `$t7` and `$t9` into the stack.

One Way:

```
sub $sp, $sp, 8
sw $t9, 4($sp)
sw $t7, 8($sp)
```

Alternate Way:

```
sw $t9, -4($sp)
sw $t7, -8($sp)
sub $sp, $sp, 8
```

Accessing and Popping Elements With stack pointer (`$sp`), you can access an element in the stack at any position (not just the top one) if and only if you know the position relative to top element `$sp`.

For example, to retrieve the value of `$t7`:

```
lw $a0, 8($sp)
```

With the above command, you can also `pop` or "wipe" the element simply by making the stack pointer upward. For example, to pop the value of `$t9` that was previously added, yielding the stack shown at the bottom:

```
addi $sp, $sp, 8
```

Note that the popped data is still present in memory, but the data past the stack pointer is considered invalid.

CS214 : MIPS Assembly Programming
Tutorial 9: Recursion and Nested Function Calls

Instructor: Dr. Sukarn Agarwal,
Assistant Professor,
SCEE,
IIT Mandi

November 9, 2023

Recursion

As same as the recursive relation exist, for example calculating the factorial of a given number,

```
n! = n * (n-1) * (n-2) ... *2 * 1 = n* (n-1)!
```

The recursive function exist in the programming languages. The recursive functions is a functions that calls itself repeatedly. The recursive function will keep on calling itself, with different parameters, until a terminating condition is met. It's like writing a loop. You can also write a loop to do something over and over again, until some exiting condition is met.

In the MIPS assembly programming, in the case of writing a recursive function, it is the responsibility of the MIPS programmer to save on the stack the contents of all registers relevant to the current invocation of the function before a recursive call is executed. Upon returning from a recursive function call the values saved on the stack must be restored to the relevant registers.

Nested Function Calls As same as recursive function, a similar case happens when you call a function that can call another function.

```
A: ....
   # Put B's args in $a0-$a3
   jal B # $ra = A2
A2: ....


B: ...
   # Put C's args in $a0-$a3
   # erasing B's args !
   jal C
B2: ...
   jr $ra # where does this go?

C: ...
```

```
jr $ra
```

Consider the above example, that have function `A` that calls `B`, which calls `C`.

- As observe in the above code, the arguments for the call to `C` would be placed in `$a0-$a3`, therefore overwriting the original arguments for `B`.
- Similarly, `jal C` overwrites the return address that was saved in `$ra` by the earlier `jal B`.

Register Spilling These cases incurs due to MIPS machine has a limited number of registers for use by all functions, and it's possible that several functions will require the same register for the different purposes. To handle this, we can save important register from being overwritten by a function and restore them after the function completes.

There are two possible ways to save important registers across function calls:

- The caller saves the important register as it knows which registers are important to it.
- The callee knows exactly which register it will use and accordingly it overwrites it.

The caller save the register This can be done by the caller to save the registers that it needs before the function calls. The saved register restore at the later point of time, when the control return back from the callee. However, the problem with this method is sometimes the caller does not know which registers are important for their execution. As a result, it may end up with saving large number of registers.

```
A: li $s0, 2
   li $s1, 3
   li $t0, 1
   li $t1, 2

   # Code pertaining to save the register $s0, $s1, $t0, $t1

   jal B

   # Code pertaining to restore the register $s0, $s1, $t0, $t1

   add $v0, $t0, $t1
   add $v1, $s0, $s1
   jr $ra
```

In the above example, function `A` saves the register `$s0, $s1, $t0 and $t1` before jump to the procedure `B`. However, it may be possible that the procedure `B` may not use these registers.

The callee save the register Another alternative ways is if callee save the value of register before the callee statement starts or before the register is being overwritten. The saved register is restored at the later point of time, when the callee function finishes their execution.

```
B: # Save registers
# $t0 $t1 $s0 $s2

li $t0, 2
li $t1, 7
li $s0, 1
li $s2, 8
...

# Restore registers
# $t0 $t1 $s0 $s2

jr$ra
```

For example, in the above code, function `B` uses register `$t0, $t1, $s0, $s2`. Hence, before using them, the callee procedure save the original values first. Thereafter, restore them before returning. However, as same as the case with the caller, the callee does not know what registers are important to the caller. As a result, it may again end up with saving more number of register.

The caller and callee work together To overcome the scenario that leads to saving more number of registers. MIPS machines uses conventions again to split the registers spilling chores.

- The caller is responsible for handling it **caller saved registers** by saving and restoring them. In other words, the callee may now freely modifying the following set of register, under the assumption that caller already saved them before jumps to callee.

  ```
  $t0-$t9 $a0-$a3 $v0-$v1
  ```

- The callee is responsible for handling it **callee saved register** by saving and restoring them. In particular, the caller now assume that these following set of registers are not altered by the called. Note that the register `$ra` is handled here carefully; as it is saved by a callee who may be caller at some point of time.

  ```
  $s0-$s7 $ra
  ```

Hence, with register spilling, be careful when working with nested functions, which can act as both caller and callee.

Problem 1: Write a MIPS assembly recursive function to find the determinant of a 3 x 3 matrix (it can be integer or floating point array). The address of the array M and the size N are passed to the function on the stack, and the result R is returned on the stack:

Problem 2: Write an efficient MIPS assembly language function that will scan through a string of characters with the objective of locating where all the lower case vowels appear in the string, as well as counting how many total lower case vowels appeared in a string. Vowels are the letters a, e, i, o, u. The address of the string is passed to the function on the stack, and the number of vowels found is returned on the stack. A null character terminates the string. Within this function, you need to call other function that Justifies the relative position within the string where each vowel was found. Notice this will be a nested function call (calling of a function inside the function). Here is an example string:

```
The quick brown fox.

For the above example string the output of your program would be

A Vowel was Found at Relative Position :                3
A Vowel was Found at Relative Position :                6
A Vowel was Found at Relative Position :                7
A Vowel was Found at Relative Position :               13
A Vowel was Found at Relative Position :               18
```