



University of
Nottingham

UK | CHINA | MALAYSIA

MIPS Programming 2

Dr. Heng Yu

AY2023-24, Spring Semester
COMP1047: Systems and Architecture
Week 3



- **MIPS Decision Making and Branching**
- **MIPS Arrays**
- **MIPS Procedure**





Learning Objectives

- Understand and write MIPS programs with branching instructions
- Understand and write MIPS programs involving arrays
- Understand and write MIPS procedures
 - Understand and implement caller- and callee-saved registers
 - Understand the concept and usage of stack memory
 - Implement with procedure calling conventions.



- **MIPS Decision Making and Branching**
- MIPS Arrays
- MIPS Procedure





- So far, All instructions learnt allow us to manipulate data.
- So we've built a calculator.
- In order to build a computer, we need the ability to make decisions...
- **Branching and Control flow**
 - Branch instruction affect the Program Counter (PC) and hence the control flow of the program
 - **Conditional** branch instructions perform a branch depending on a condition.
 - **Unconditional branch** instructions (e.g. **goto**) perform a branch unconditionally.



Control Flow in High Level Languages

- **goto** has (mostly) been eliminated from high level programming languages
 - It will lead to an unmaintainable mess
- Structured statements: **if**, **if/else**, **while** and **for** are used instead
- But in MIPS, both types of branching (goto and if-family) are provided. We will learn MIPS branching instructions that correspond to the above structs.



MIPS Branch Instructions

- **beq a, b, L** - Branch on equal
 - Go to instruction at label L **if** $a == b$, otherwise, continue with the next instruction
- **bne a, b, L** - Branch on not equal
 - Go to instruction at label L **if** $a \neq b$, otherwise, continue with the next instruction
- **j L** – Jump to
 - jump to the instruction at label L



Conditional Branching (beq)

MIPS assembly

```
addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq  $s0, $s1, target # branch is taken
addi $s1, $s1, 1      # not executed
sub  $s1, $s1, $s0     # not executed

target:               # label
add  $s1, $s1, $s0     # $s1 = 4 + 4 = 8
```

Labels indicate instruction locations in a program. They cannot use reserve words and must be followed by a colon (:).



Conditional Branching (bne)

MIPS assembly

```
addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
bne     $s0, $s1, target    # branch not taken
addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
sub     $s1, $s1, $s0       # $s1 = 5 - 4 = 1
```

target:

```
add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```



Unconditional Branching (j)

MIPS assembly

```
addi $s0, $0, 4          # $s0 = 4
    addi $s1, $0, 1       # $s1 = 1
    j     target          # jump to target
    sra   $s1, $s1, 2      # not executed
    addi  $s1, $s1, 1      # not executed
    sub   $s1, $s1, $s0    # not executed

target:
    add   $s1, $s1, $s0    # $s1 = 1 + 4 = 5
```

What is the operand for j instruction?



Unconditional Branching (jr)

MIPS assembly

0x00002000	<code>addi \$s0, \$0, 0x2010</code>
0x00002004	<code>jr \$s0</code>
0x00002008	<code>addi \$s1, \$0, 1</code>
0x0000200C	<code>sra \$s1, \$s1, 2</code>
0x00002010	<code>lw \$s3, 44(\$s1)</code>



Translating the 'if' statement

High-level code

```
if (i == j)
    f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```



Translating the 'if' statement

High-level code

```
if (i == j)
    f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

        bne $s3, $s4, L1
        add $s0, $s1, $s2

L1:     sub $s0, $s0, $s3
```

Notice that the assembly tests for the opposite case ($i \neq j$) than the test in the high-level code ($i == j$).

What if we use `beq`?



Translating the 'if-else' statement

High-level code

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

        bne $s3, $s4, L1
        add $s0, $s1, $s2
        j   done
L1:     sub $s0, $s0, $s3
done:
```



Translating the 'while loop'

High-level code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x    = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128

while:   beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while

done:
```

Notice that the assembly tests for the opposite case (`pow == 128`) than the test in the high-level code (`pow != 128`).

What if we use `bne` here?



Translating the 'for loop'

High-level code

```
// add the numbers from 0 to 9
int i;
int sum = 0;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

MIPS assembly code

```
addi $s0, $0, 0    # $s0 = i
add  $s1, $0, $0    # $s1 = sum

addi $t0, $0, 10

for: beq  $s0, $t0, done
    add  $s1, $s1, $s0
    addi $s0, $s0, 1
    j    for

done:
```



Inequality in MIPS

- Until now, we've only tested equalities (beq and bne), but general programs need to test '<' and '>'
- Set on Less Than:
 - `slt rd, rs, rt`
 - if (rs < rt) rd = 1; else rd = 0;
 - `slti rt, rs, constant`
 - if (rs < constant) rt = 1; else rt = 0;

Compile by hand: `if (g < h) goto Less;`

Let g: \$s0, h: \$s1

```
slt $t0, $s0, $s1    # $t0 = 1 if g < h
bne $t0, $0, Less     # goto Less if $t0 != 0
```



Branch Instruction Design

- MIPS has no “branch on less than”, i.e., `blt`, `bge`. Why?
- Hardware for $<$, \geq , ... are slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- Although `beq` and `bne` are less direct (need to combine with `slt`), this is a good design compromise between performance and code efficiency.



Signed vs. Unsigned 'slt'

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`

`slt $t0, $s0, $s1 # signed`

- $-1 < +1$, so `$t0 = 1`

`sltu $t0, $s0, $s1 # unsigned`

- $+4,294,967,295 > +1$, so `$t0 = 0`



Using 'slt' in the 'for loop'

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

MIPS assembly code

```
addi $s0, $0, 1    # $s0 = i
addi $s1, $0, 0    # $s1 = sum

addi $t0, $0, 101

loop:  slt  $t1, $s0, $t0
      beq  $t1, $0, done
      add  $s1, $s1, $s0
      sll  $s0, $s0, 1
      j    loop

done:
```

$\$t1 = 1$ if $i < 101$.



Exercise: Maximum of two numbers

```
.text
main:  li $t0, 0
       li $v0, 5
       syscall
       move $s0, $v0           # read and store input x in $s0
       li $v0, 5
       syscall                 # read and store input y in $v0
       ?                       # if $v0 < $s0, $t0 = 1
       ?                       # if $t0 != 0 (i.e., $t0 = 1, $v0 < $s0), goto out
       ?                       # otherwise (i.e., $v0 >= $s0), store large in $s0
out:   move $a0, $s0           # print maximum number stored in $a0
       li $v0, 1
       syscall                 # print integer
       li $v0, 10
       syscall                 # exit
```



- MIPS Decision Making and Branching
- **MIPS Arrays**
- MIPS Procedure





Arrays

- A data structure that is useful for accessing large amounts of similar data
- Array element: accessed by **index**
- Array **size**: number of elements in the array

```
int z[10];           // an array of 10 ints, z points to start  
z[0] = 2; z[1] = 3; // assigns 2 to the first, 3 to the next
```




Accessing Array Data in MIPS

- Since arrays can store lots of data, and we have only a small (~ 32) number of registers, it is infeasible to use the registers for long-term storage of the array data

- Hence, arrays are stored in the data segment of a MIPS program

- E.g. the declaration of an array with 8 elements is:

```
arr: .word 3, 10, 4, 1, 15, 9, 2, 6
```

- To access the data in the array requires that we know the address of the data and then use the load word (lw) or store word (sw) instructions



Accessing Array Data in MIPS

```
arr: .word 3, 10, 4, 15, 5, 9, 2, 6
```

- To find where the array is: `la $t0, arr`
 - `$t0` contains **the address of the first element** '3' in the array
 - The index address of the second element '10' is `$t0 + 4`
 - The address of the fifth element '5' is `$t0 + 16`
- The following code will place the value of `arr[6]` into the `$t4`:

```
la  $t3, arr      # put address of arr into $t3
li  $t2, 6        # put the index into $t2
sll $t2, $t2, 2    # 4x the index to find the byte location
add $t1, $t2, $t3  # obtain the address
lw  $t4, 0($t1)    # get the value from the array cell
```



Another way to load the array head

- Given the base address = 0x12348000 (address of the first array element, array[0])
- Use **lui + ori** to load 32-bit base address into a register
- **lui** (load upper immediate)
 - `lui $s0, 0x1234 # $s0 = 0x12340000`
- **ori** (or immediate)
 - `ori $s0, $s0, 0x8000 # $s0 = 0x12348000`

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

Notice the usage difference between `li` and `lui`, when loading an immediate.

- If the constant would fit 16 bits, use `li`
- If the constant needs (16, 32] bits, use `lui + ori`



Another way to load the array head

// High-level code

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

MIPS assembly code

array base address = \$s0

```
lui  $s0, 0x1234      # put 0x1234 in upper half of $s0  
ori  $s0, $s0, 0x8000 # put 0x8000 in lower half of $s0
```

```
lw    $t1, 0($s0)      # $t1 = array[0]  
sll   $t1, $t1, 1      # $t1 = $t1 * 2  
sw    $t1, 0($s0)      # array[0] = $t1
```

```
lw    $t1, 4($s0)      # $t1 = array[1]  
sll   $t1, $t1, 1      # $t1 = $t1 * 2  
sw    $t1, 4($s0)      # array[1] = $t1
```

- Assembly strings are arrays of ASCII characters
 - A string is finished with a NUL (0) character.
 - 1 ASCII character is 1 byte.
- Declare a string in assembly code
 - with the `.ascii` directive

```
li $v0, 0           # length = 0 ;
j strlen_cond       # assume $a0 points to the string head

strlen_loop:
    addi $v0, $v0, 1      # length++

strlen_cond:
    lbu $t0, 0($a0)      # load char at address $a0
    addi $a0, $a0, 1      # point $a0 to next char
    bne $t0, $zero, strlen_loop # while not NUL
end:                  # now $v0 contains the string length
```




- MIPS Decision Making and Branching
- MIPS Arrays
- **MIPS Procedure**





- Procedures are portion of code, within a larger program, which runs frequently
- Procedures help to
 - Reduce code duplication
 - Improve code re-usability
 - Decompose complex programs into manageable parts
- Other names
 - Methods – java and other OO languages
 - Functions – C, C++, Haskell
 - Routines, subroutines – (seems not popular now)



Definitions

- Caller: calling procedure (in this example, `main`)
- Callee: called procedure (in this example, `sum`)

High-level code

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

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



Procedure calling conventions:

- Caller:
 - passes **arguments** to callee.
- Callee:
 - **must not overwrite** registers or memory needed by the caller
 - **returns to the point of call**
 - **returns the result** to caller

MIPS conventions:

- Call procedure: jump and link (**jal**)
- Return from procedure: jump register (**jr**)
- Argument values: **\$a0 - \$a3**
- Return value: **\$v0, (\$v1 for 64-bit double)**

High-level code

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

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



High-level code

```
int main() {  
    simple();  
    a = b + c;  
}  
  
void simple() {  
    return;  
}
```

MIPS assembly code

```
0x00400200 main: jal    simple  
0x00400204          add    $s0, $s1, $s2  
...  
  
0x00401020 simple: jr    $ra
```

jal: jumps to `simple` and **saves PC+4 to the return address register (\$ra)**.

In this case, **\$ra = 0x00400204** after `jal` executes.

jr \$ra: **jumps to address in \$ra**, in this case 0x00400204.



Input Arguments and Return Values

High-level code

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

MIPS assembly code

```
main:
    ...
    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal  diffofsums    # call procedure
    add  $s0, $v0, $0   # y = returned value
    ...

# $s0 = result
diffofsums:
    add $s2, $a0, $a1   # $s2 = f + g
    add $s1, $a2, $a3   # $s1 = h + i
    sub $s0, $s2, $s1   # result = (f + g) - (h + i)
    add $v0, $s0, $0    # put return value in $v0
    jr  $ra             # return to caller
```



MIPS assembly code

diffofsums:

```
add $s2, $a0, $a1    # $s2 = f + g
add $s1, $a2, $a3    # $s1 = h + i
sub $s0, $s2, $s1    # result = (f + g) - (h + i)
add $v0, $s0, $0     # put return value in $v0
jr  $ra              # return to caller
```

diffofsums **overwrites** 3 registers: \$s2, \$s1, and \$s0

diffofsums can use **stack** to temporarily store registers

The Stack

- Memory used to temporarily save variables
- Like a stack of dishes, **last-in-first-out (LIFO) queue**
- *Expands*: uses more memory when more space is needed
- *Contracts*: uses less memory when the space is no longer needed
- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp`, points to top of the stack

Address	Data		Address	Data	
7FFFFFFC	12345678	← <code>\$sp</code>	7FFFFFFC	12345678	
7FFFFFF8			7FFFFFF8	AABBCCDD	
7FFFFFF4			7FFFFFF4	11223344	← <code>\$sp</code>
7FFFFFF0			7FFFFFF0		
⋮	⋮		⋮	⋮	





How Procedures Use the Stack

- The callee must make no unintended side effects on the caller.
- But `diffofsums` overwrites 3 registers: **`$s2`, `$s1`, `$s0`**

MIPS assembly

```
# $s0 = result
```

```
diffofsums:
```

```
    add $s2, $a0, $a1    # $s2 = f + g
```

```
    add $s1, $a2, $a3    # $s1 = h + i
```

```
    sub $s0, $s2, $s1    # result = (f + g) - (h + i)
```

```
    add $v0, $s0, $0      # put return value in $v0
```

```
    jr  $ra               # return to caller
```



Use Stack to Protect Caller Values

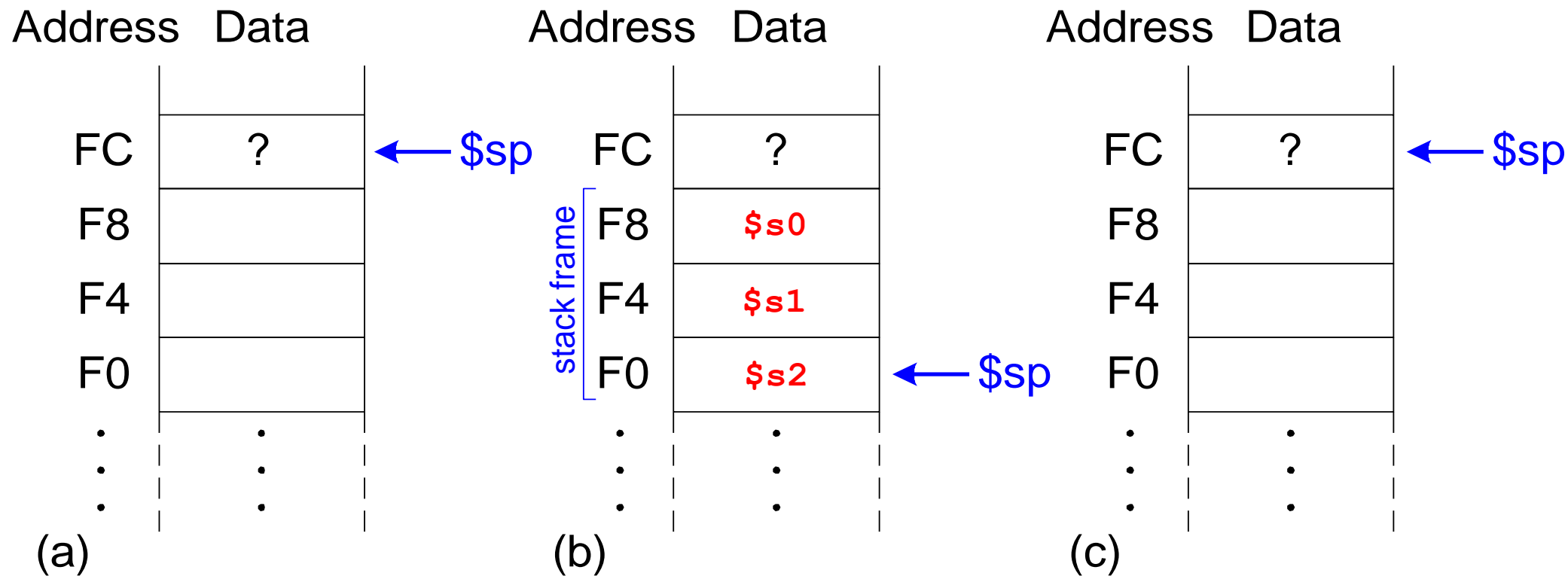
diffofsums:

```
    addi $sp, $sp, -12    # make space on stack
                           # to store 3 registers

    sw    $s0, 8($sp)     # save $s0 on stack
    sw    $s1, 4($sp)     # save $s1 on stack
    sw    $s2, 0($sp)     # save $s2 on stack
    add    $s2, $a0, $a1   # $s2 = f + g
    add    $s1, $a2, $a3   # $s1 = h + i
    sub    $s0, $s2, $s1   # result = (f + g) - (h + i)
    add    $v0, $s0, $0     # put return value in $v0
    lw     $s2, 0($sp)     # restore $s2 from stack
    lw     $s1, 4($sp)     # restore $s1 from stack
    lw     $s0, 8($sp)     # restore $s0 from stack
    addi   $sp, $sp, 12    # deallocate stack space
    jr     $ra             # return to caller
```



The Stack During diffofsums Call





Who should push/pop which Registers?

- MIPS registers are divided into two types: **caller-saved** and **callee-saved**.

<i>Callee-Saved</i> <i>(since caller may have used them)</i>	<i>Caller-Saved</i> <i>(since callee may use them)</i>
\$s0 – \$s7	\$t0 – \$t9
\$ra	\$a0 – \$a3
\$sp	\$v0 – \$v1



Use Stack to Protect Caller/Callee Values

Caller

```
main:
...
addi $a0, $0, 2      # argument 0
addi $a1, $0, 3      # argument 1
addi $a2, $0, 4      # argument 2
addi $a3, $0, 5      # argument 3

addi $sp, $sp, -8
sw    $t0, 4($sp)
sw    $t1, 0($sp)
jal   diffofsums     # call procedure
lw    $t1, 0($sp)
lw    $t0, 4($sp)
addi  $sp, $sp, 8
add   $s0, $v0, $0
...
add   $t0, $t1, $s1
...
```

Callee

```
diffofsums:
    addi $sp, $sp, -12
    sw    $s0, 8($sp)
    sw    $s1, 4($sp)
    sw    $s2, 0($sp)
    add   $s2, $a0, $a1
    add   $s1, $a2, $a3
    sub   $s0, $s2, $s1
    add   $t0, $0, $a1
    add   $t1, $0, $a2
    add   $v0, $s0, $0
    lw    $s2, 0($sp)
    lw    $s1, 4($sp)
    lw    $s0, 8($sp)
    addi  $sp, $sp, 12
    jr    $ra
```



Use Stack to Protect Caller/Callee Values

Caller

```
main:
...
addi $a0, $0, 2    # argument 0
addi $a1, $0, 3    # argument 1
addi $a2, $0, 4    # argument 2
addi $a3, $0, 5    # argument 3

addi $sp, $sp, -8
sw    $t0, 4($sp)
sw    $t1, 0($sp)
jal   diffofsums   # call procedure
lw    $t1, 0($sp)
lw    $t0, 4($sp)
addi  $sp, $sp, 8
add   $s0, $v0, $0
...
add   $t0, $t1, $s1
...
```

Callee

diffofsums:

```
addi $sp, $sp, -16
```

```
sw    $ra, 12(sp)
```

```
sw    $s0, 8($sp)
```

```
sw    $s1, 4($sp)
```

```
sw    $s2, 0($sp)
```

```
add   $s2, $a0, $a1
```

```
add   $s1, $a2, $a3
```

```
sub   $s0, $s2, $s1
```

```
add   $t0, $0, $a1
```

```
add   $t1, $0, $a2
```

```
add   $v0, $s0, $0
```

```
lw    $s2, 0($sp)
```

```
lw    $s1, 4($sp)
```

```
lw    $s0, 8($sp)
```

```
lw    $ra, 12(sp)
```

```
addi  $sp, $sp, 16
```

```
jr    $ra
```

MIPS Calling Convention

- Caller
 - Push any of $\$a0-\$a3$, $\$v0-\$v1$ and $\$t0-\$t9$ if necessary
 - Place arguments in $\$a0$ to $\$a3$ if needed
 - Make the call using `jal callee`
 - Pop saved registers and/or extra arguments off stack
- Callee
 - Push any of $\$ra$, $\$s0-\$s7$ that may be overwritten
 - Perform desired task
 - Place result in $\$v0$ and $\$v1$
 - Pop above registers off the stack
 - Return to caller with `jr $ra`





- MIPS branching instructions and programming
- MIPS arrays
- MIPS procedures
 - caller- and callee-saved registers
 - stack memory
 - procedure calling conventions



University of
Nottingham

UK | CHINA | MALAYSIA

Stay Tuned.