Outline of Lecture

- Procedure calls
- Saving and restoring registers
- Summary of MIPS instructions

Procedure Calls

- A procedure of a subroutine is like an “agent” which needs certain information to perform a certain job.
- In MIPS a *callller* calls a procedure (*callee*).
- When the callee is executed, the following steps are taken:

1. Place parameters in a place where the callee can access them;
2. Transfer control to the callee.
3. Acquire the storage resources needed for the callee.
4. Perform the task.
5. Place result value in a place where the caller can access it.
6. Return the control to the caller.

Procedure Calls

- The following registers are used during the procedure call:

  $a0 — $a3 (four argument registers)

  $v0 — $v1 (two value registers)

  $ra (return address register)

- The instruction that places the address of callee in the PC and stores the return address in $ra is

```
jal    CalleeAddress
```

- The callee must have a call to return to the return address

```
jr     $ra
```

- The nested calls must save the intermediate value of $ra register in stack.

Example

- SInce the registers are used by various procedures, their old values can be written by new values, so they must be saved.
- Look at the following code:

```
int leaf_example(int g, int h, int i, int j)

{

      int f;

      f = (g+h) - (i+j)

      return f;

}
```

Suppose

$a0 <— g

$a1 <— h

$a2 <— i

$a3 <— j

Then the MIPS code for the instruction

```
f = (g+h) - (i+j)
```

would look like this

```
add $t0, $a0, $a1 #register $t0 contains g+h

add $t1, $a2, $a3 #register $t1 contains i+j

sub $s0, $t0, $t1 #register f = $t0 - $t1
```

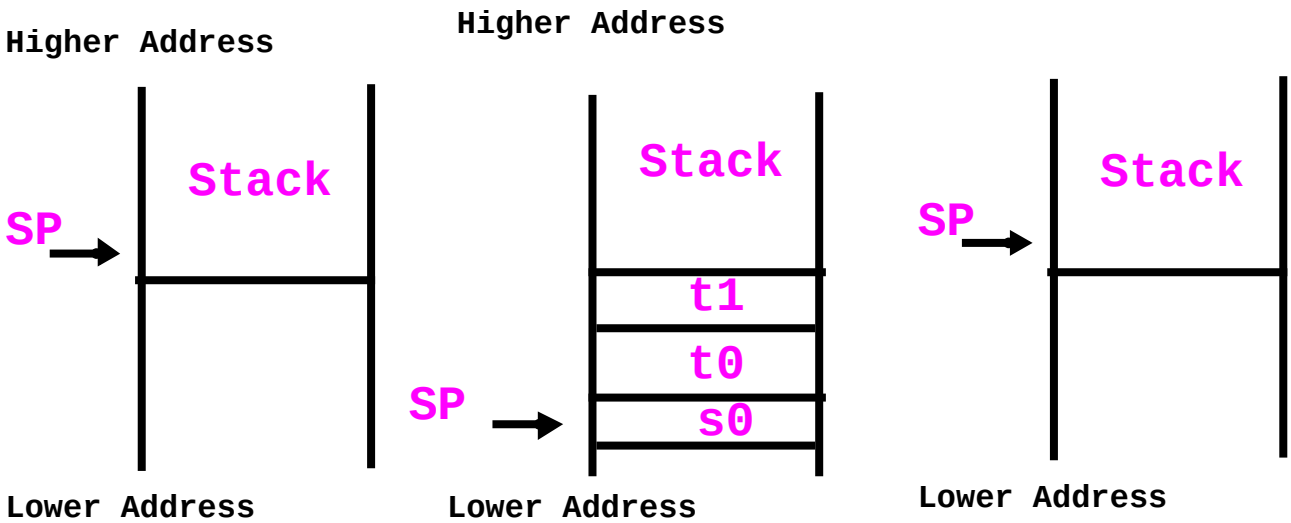
to return the value of f, we copy it into a return value register

```
add $v0, $s0, $zero # return f ($v0 = $s0 +0)
```

Saving Register

- In this code, we use temporary registers. Suppose their old values must be saved and then restored.
- To do so, we use stack.
- Before executing the above code, we execute the following code

```
sub $sp, $sp, 12    #adjust stack for 3 items

sw $t1, 8($sp)      #save register $t1

sw $t0, 4($sp)      #save register $t1

sw $s0, 0($sp)      #save register $t1
```



Restoring Register

- After calculating the value of f, we must restore the old values of temporary registers.
- For this purpose, we execute the following code

```
lw $s0, 0($sp)      #restore register $t1

lw $t0, 4($sp)      #restore register $t1

lw $t1, 8($sp)      #restore register $t1

add $sp, $sp, 12    #adjust stack for 3 items
```

Dont forget to add

```
jr $ra    # jump back to the caller
```

Register Spilling

- The technique of saving register is called register spilling.
- Register spilling can generate a lot of work.
- To avoid that, MIPS offers two classes of registers.
- $t0 — $t9: 10 temporary registers that are not preserved by the callee.
- $s0 — $s7: 8 registers that must be saved by the callee.
- This convention reduces register spilling.
- In the above example, $t0 and $t1 need not be saved.
- This reduces unnecessary work.

Constants

- 50% of the operands are constants.
- For example:

```
A = A +5

B = B - 5
```

- For such cases, we have learned that we can use immediate instructions

```
addi $29, $29, 4

slti $8, $18, 10

andi $29, $29, 6

ori $29, $29, 4
```

However, the constant field is 16 bits wide

Larger Constants

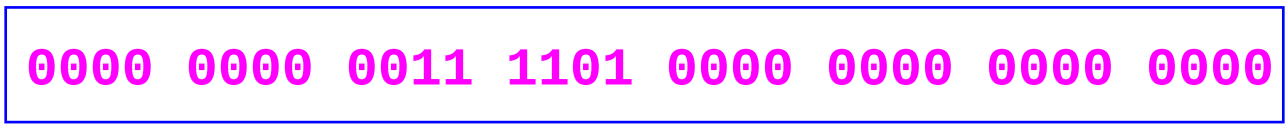
- 32 bit constants are added using two instruction
- Suppose, we want to load this 32 bit constant into register $s0

0000 0000 0011 1101	0000 1001 0000 0000
integer value = 61	integer value = 2304

```
lui $s0, 61

addi $s0, $s0, 2304
```

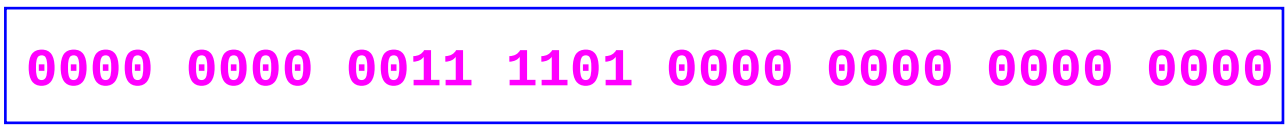
Another Way of Loading a Long Constant

```
lui $s0, 0000 0000 0011 1101
```

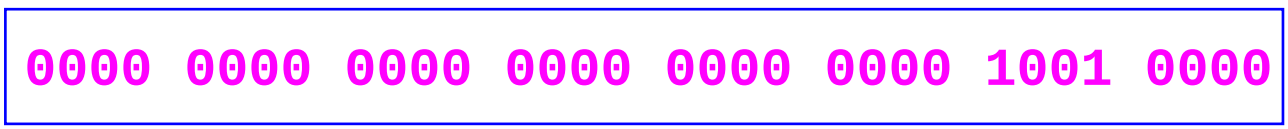


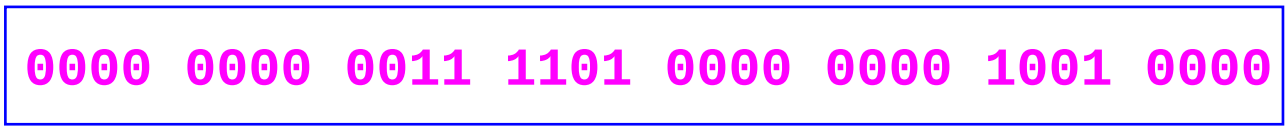
contents of $t0

```
ori, $t0, $t0, 0000 0000 1001 0000
```



ori operation





Handling a Single Byte

- A single byte can also be read or written.

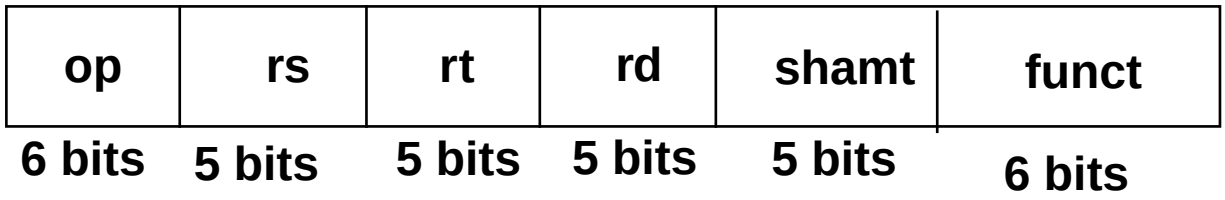
```
lb, $t0, 0($sp)  #load byte from address

sb, $t0, 0($sp)  #store byte to address
```

- The lb instruction reads one byte from memory and loads it into the rightmost 8 bits of a register
- The sb instruction reads the rightmost 8 bots of a register and stores them into the memory.

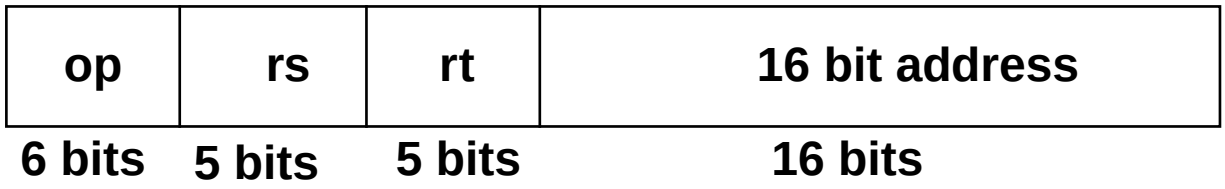
Summary of Instructions

- Three major kinds of instructions

R type Instruction



I type Instruction



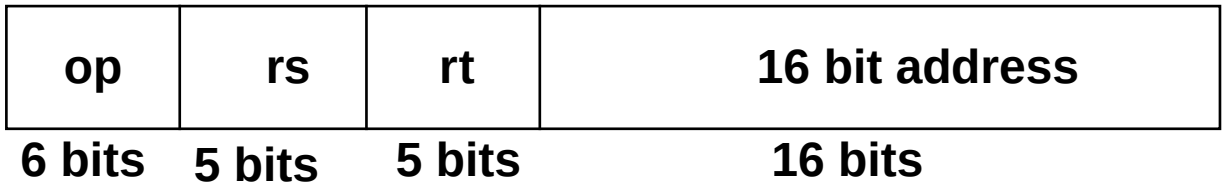
J type instruction

Branching

```
bne, $t4, $t5, Label
```

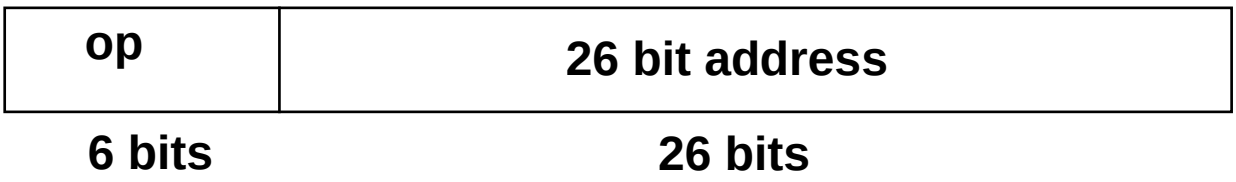
means that the next instruction at Label if $t4 is not equal to $t5

```
beq, $t4, $t5, Label
```



means that the next instruction at Label if $t4 is equal to $t5

```
j Label
```



Remember

The addresses in jump instructions are instruction addresses and not byte addresses.

Larger Jumps in Branches

- The addresses in *beq* and *beq* are 16 bits.
- How to jump to larger address.
- Use relative addressing, that is, jump relative to some register (such as PC)