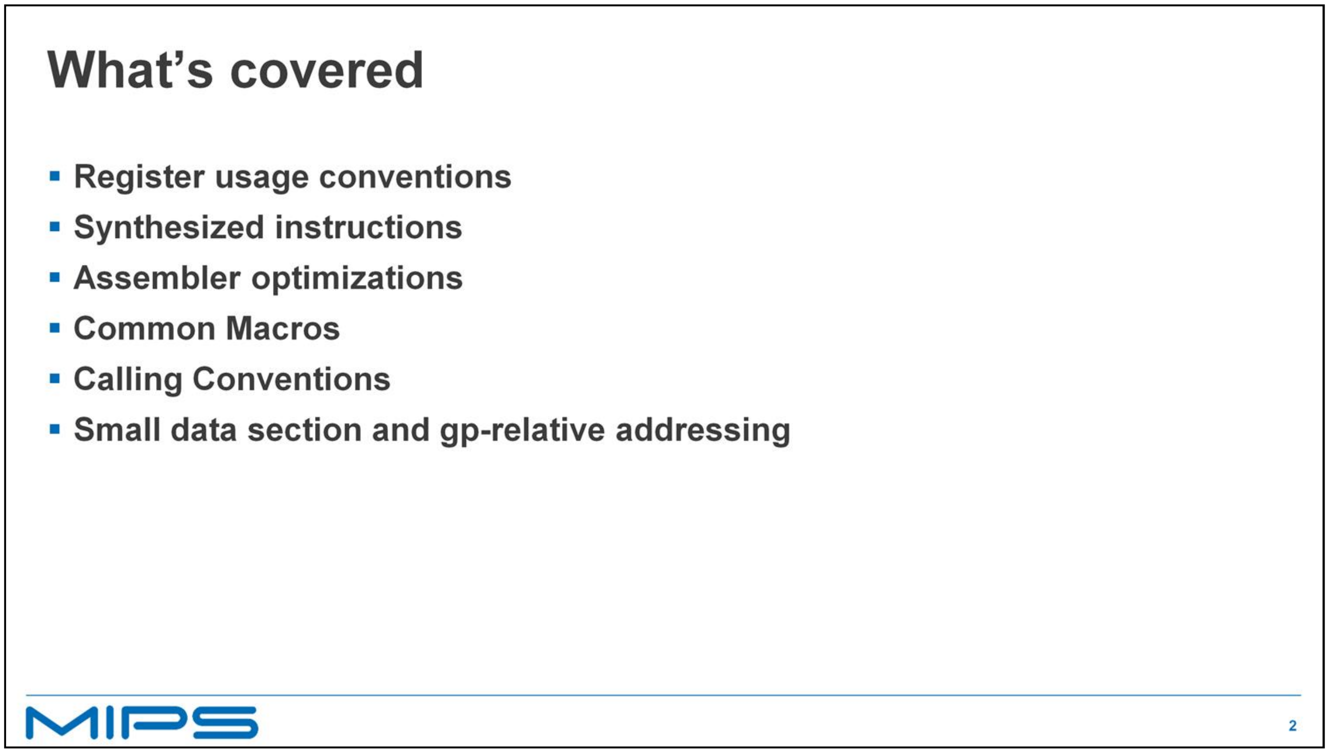
MIPS Assembly Language

www.mips.com

Welcome to the MIPS Assemble language section of the MIPS Software Training Course



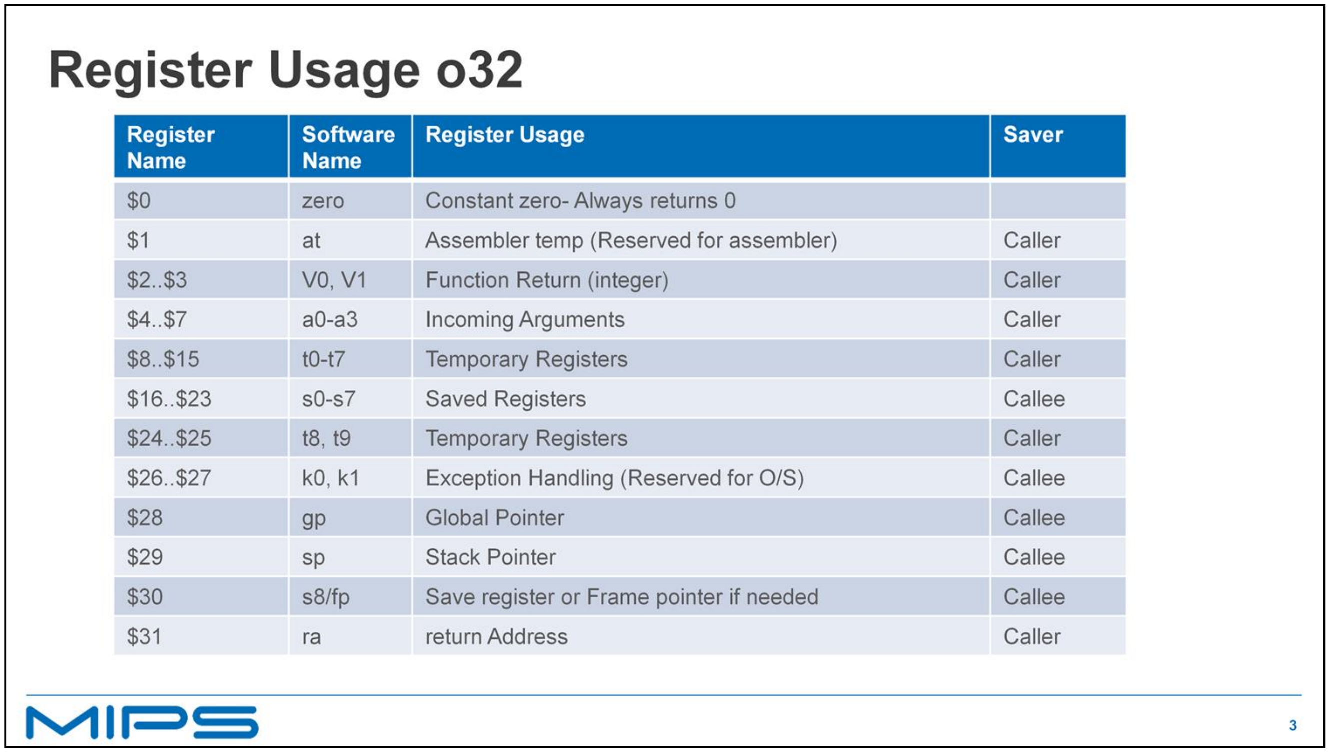
What's covered

- Register usage conventions
- Synthesized instructions
- Assembler optimizations
- Common Macros
- Calling Conventions
- Small data section and gp-relative addressing

2

In this section I will cover

+ the usage conventions for the 32 general purpose registers
+ synthesized instructions
+ assembler optimizations
+ Common macros you can use
+ Calling conventions
+ Use of the small data section and gp-relative addressing


Register Usage o32

Register Name	Software Name	Register Usage	Saver
$0	zero	Constant zero- Always returns 0	
$1	at	Assembler temp (Reserved for assembler)	Caller
$2..$3	V0, V1	Function Return (integer)	Caller
$4..$7	a0-a3	Incoming Arguments	Caller
$8..$15	t0-t7	Temporary Registers	Caller
$16..$23	s0-s7	Saved Registers	Callee
$24..$25	t8, t9	Temporary Registers	Caller
$26..$27	k0, k1	Exception Handling (Reserved for O/S)	Callee
$28	gp	Global Pointer	Callee
$29	sp	Stack Pointer	Callee
$30	s8/fp	Save register or Frame pointer if needed	Callee
$31	ra	return Address	Caller

MIPS 3

There are conventions for register usage and naming that will be used by compilers and debuggers. When you are coding in assemble it can be easier to understand your code if you use the common software name for registers. You can define these yourself or you can include the file mips/regdef.h in the beginning of your code.

Here are the register conventions;

+ Zero is a constant zero which can be read or written but will always be read as a zero. This is not really a convention because you really cannot use it for any thing else.

+ The at register is reserved for synthetic or macro instructions generated by the assembler.

There is an assembler directive to stop it from being used call .no at but this could make some macros take longer.

+ Registers v0 and v1 are used for function call return values

+ registers A0 through A3 are used for passing arguments to a function more on this later in this section

+ t0 – t9 are temporary registers are not guaranteed to survive function call so if the caller is using them it must save them to the stack before calling a function.

+ s0-s8 are the saved registers. These are guaranteed to survive a function call because the called function must save them if it needs to use any of them.

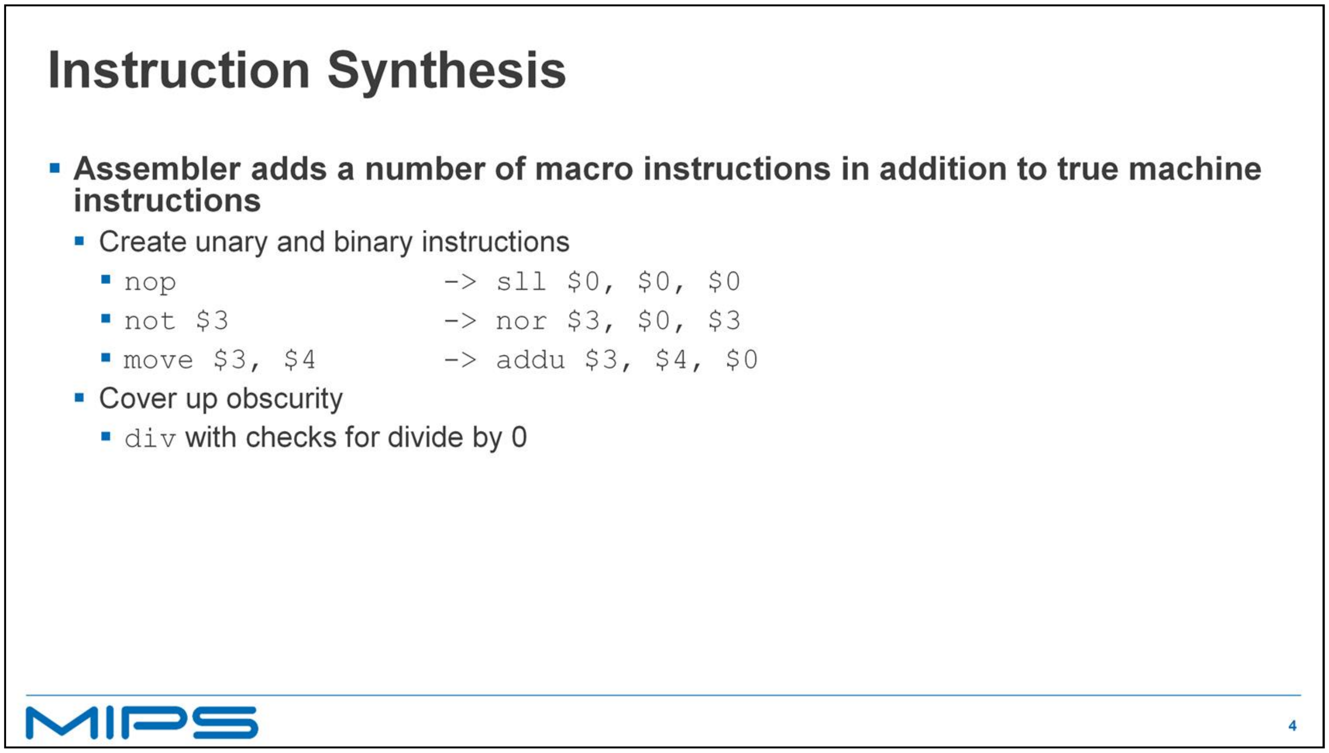
+ k0-k1 are kernel registers Used by the kernel mostly for the first parts of exception/interrupt processing if using a OS a user program should never use these registers.

+ $gp is used as a global pointer. It will point to a location that is determined when your application is initialized. This location in the midst of your static data section. Loads and stores to data lying within 32K bytes of either side of the global pointer value can be performed in a single instruction using gp as base the register. This is used differently for the Linux OS. GP Should be saved by calling function. Example use of GP: lw v0, addr translates to lw v0 (addr - _gp)(gp). Note _gp is the constant assigned in the linker file.

+ sp is the stack pointer must be preserved by the function being called.

+ fp is the frame pointer which is used by some languages for functions that need to create dynamic variables. In this case the frame pointer is used to save the incoming value of the stack pointer so the function is free to change the stack pointer. Note the frame pointer is also know as s8 which is one of the mandatory save registers that a called function must preserve.

+ ra is the return address register which as you have seen in the MIPS instructions section holds the return address stored there using the jump and link instruction. This register must be save on the stack before another function is called. You cannot use this register for anything else.



Instruction Synthesis

- **Assembler adds a number of macro instructions in addition to true machine instructions**
  - Create unary and binary instructions
    - `nop               -> sll $0, $0, $0`
    - `not $3            -> nor $3, $0, $3`
    - `move $3, $4       -> addu $3, $4, $0`
  - Cover up obscurity
    - `div` with checks for divide by 0

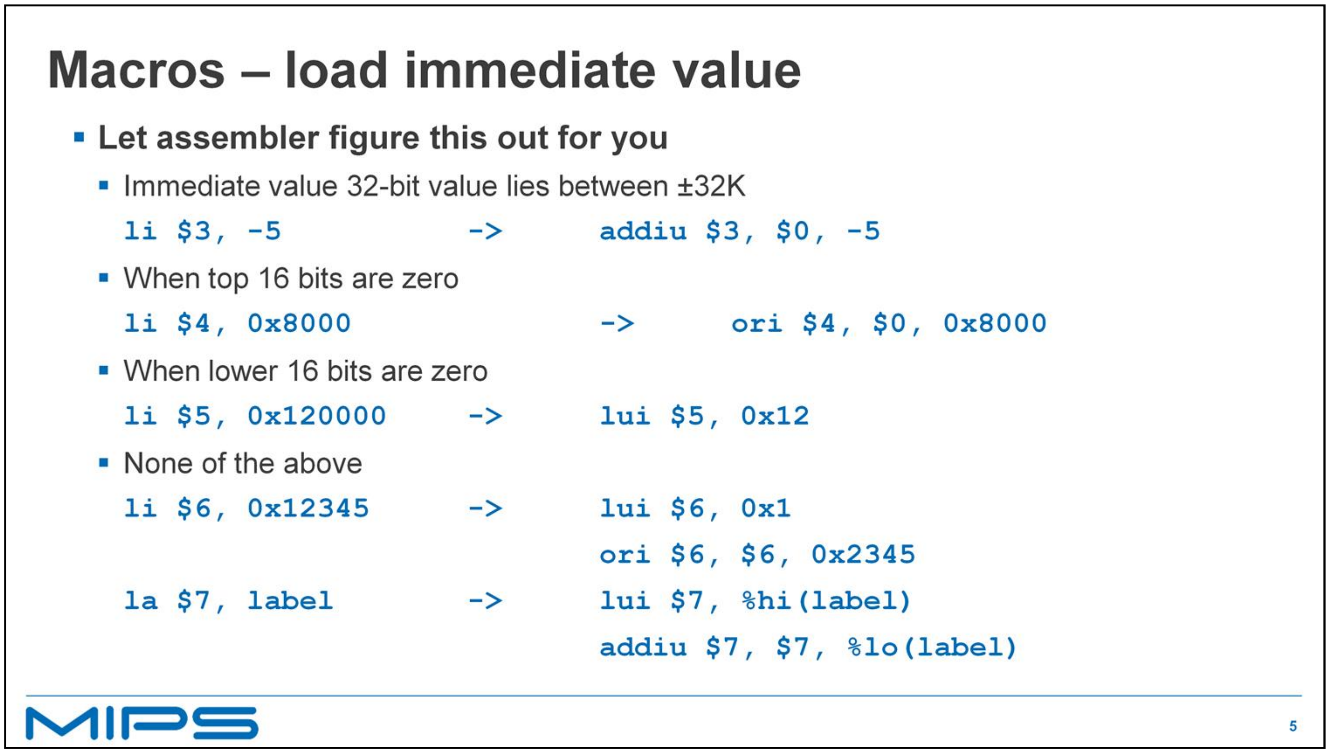
MIPS

4

As you have seen in the MIPS instruction section the Assembler adds macro instructions to round out the complete instruction set available to you.

+ the assembler creates from existing machine instructions

+ with the div instruction it includes a check for divide by zero



Macros – load immediate value

- **Let assembler figure this out for you**
  - Immediate value 32-bit value lies between ±32K

```
li $3, -5            ->      addiu $3, $0, -5
```

  - When top 16 bits are zero

```
li $4, 0x8000                ->      ori $4, $0, 0x8000
```

  - When lower 16 bits are zero

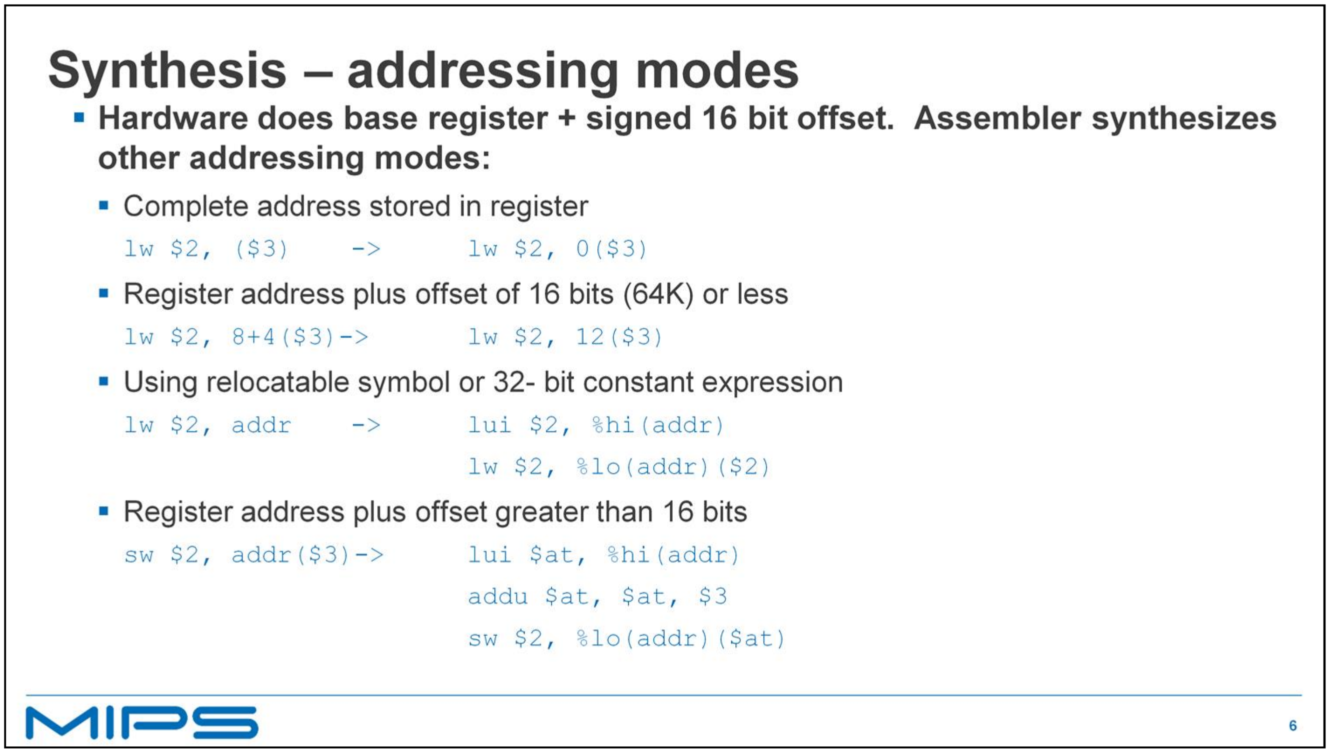
```
li $5, 0x120000      ->      lui $5, 0x12
```

  - None of the above

```
li $6, 0x12345       ->      lui $6, 0x1
                             ori $6, $6, 0x2345
la $7, label         ->      lui $7, %hi(label)
                             addiu $7, $7, %lo(label)
```

5

As you can see here it helps simplify the task for loading immediate values.



Synthesis – addressing modes

- **Hardware does base register + signed 16 bit offset. Assembler synthesizes other addressing modes:**
  - Complete address stored in register

```
lw $2, ($3)    ->      lw $2, 0($3)
```

  - Register address plus offset of 16 bits (64K) or less

```
lw $2, 8+4($3)->       lw $2, 12($3)
```

  - Using relocatable symbol or 32- bit constant expression

```
lw $2, addr    ->      lui $2, %hi(addr)
                       lw $2, %lo(addr)($2)
```

  - Register address plus offset greater than 16 bits

```
sw $2, addr($3)->      lui $at, %hi(addr)
                       addu $at, $at, $3
                       sw $2, %lo(addr)($at)
```

6

The assembler allows you to write code that is more simplified I'll show you some simplified instructions you can write and what the assembler translates them to.\

+ If the offset is zero then you can simplify your code and omit the offset

+ for the register offset you can let the assembler do arithmetic for you

+ as you can load a immediate data value you can also load a immediate address value. The assembler will break the address down into hi and lo order 16 bit pieces using multiple instructions and do the load.

+ and you can use register offset the are greater than 16 bits. In this example the assembler will break the address down into hi and lo order 16 bit pieces using multiple instructions and store the value.

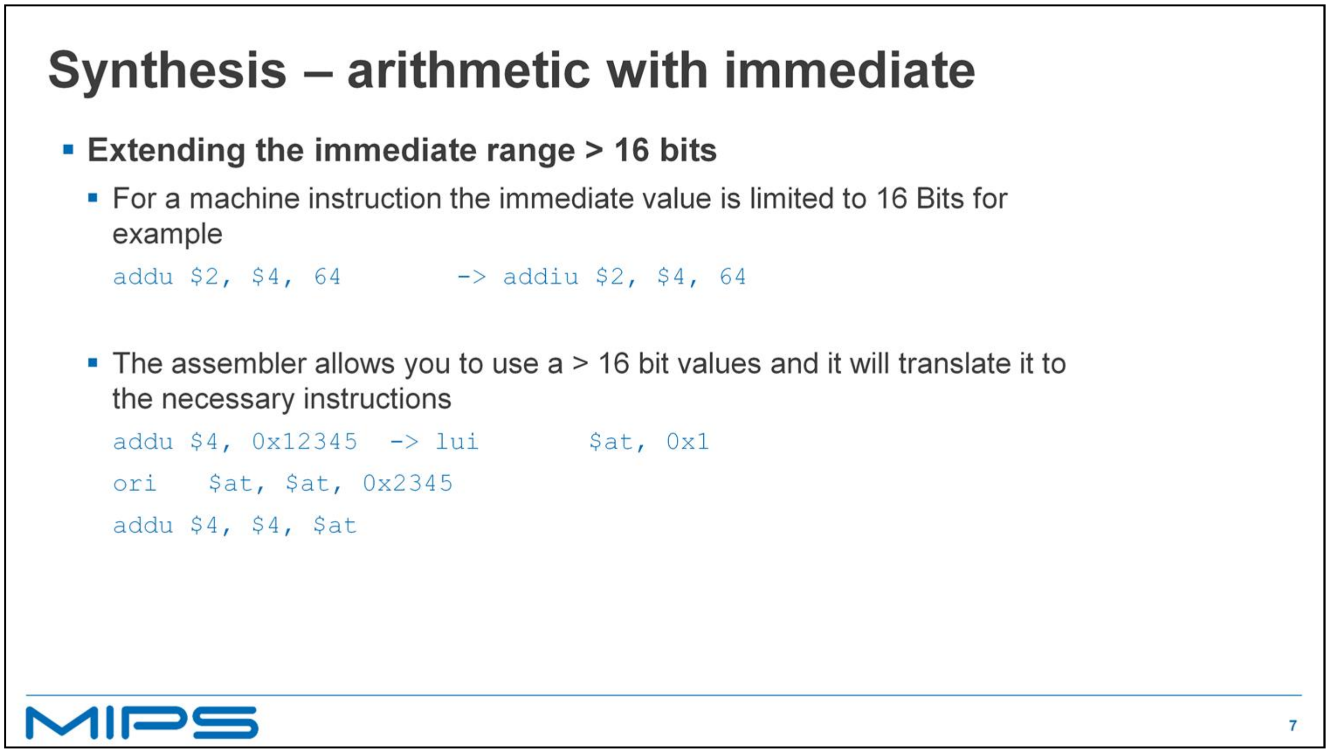
Note: When the assembler sees the constructs **%hi()** and **%lo()** it will convert the high or low 16 bits of

the 32 bit immediate value and correctly adjusting the resulting halfwords with the proper sign extension.

Extra material not to be included in video dialog in case you have questions on the use of %hi or %lo:

This is not quite the straightforward division into low and high
halfwords that it looks, because the 16-bit offset field of an **lw** is interpreted
as signed. So if the ***addr*** value is such that bit 15 is a 1, then the
%lo(*addr*) value will act as negative and we need to increment **%hi(*addr*)**
to compensate:

addr	**%hi(*addr*)**	**%lo(*addr*)**
0x1234.5678	0x1234	0x5678
0x1000.8000	0x1001	0x8000 (1001 0000 – 8000 = 1000 8000



Synthesis – arithmetic with immediate

- **Extending the immediate range > 16 bits**
  - For a machine instruction the immediate value is limited to 16 Bits for example

```
addu $2, $4, 64        -> addiu $2, $4, 64
```

  - The assembler allows you to use a > 16 bit values and it will translate it to the necessary instructions

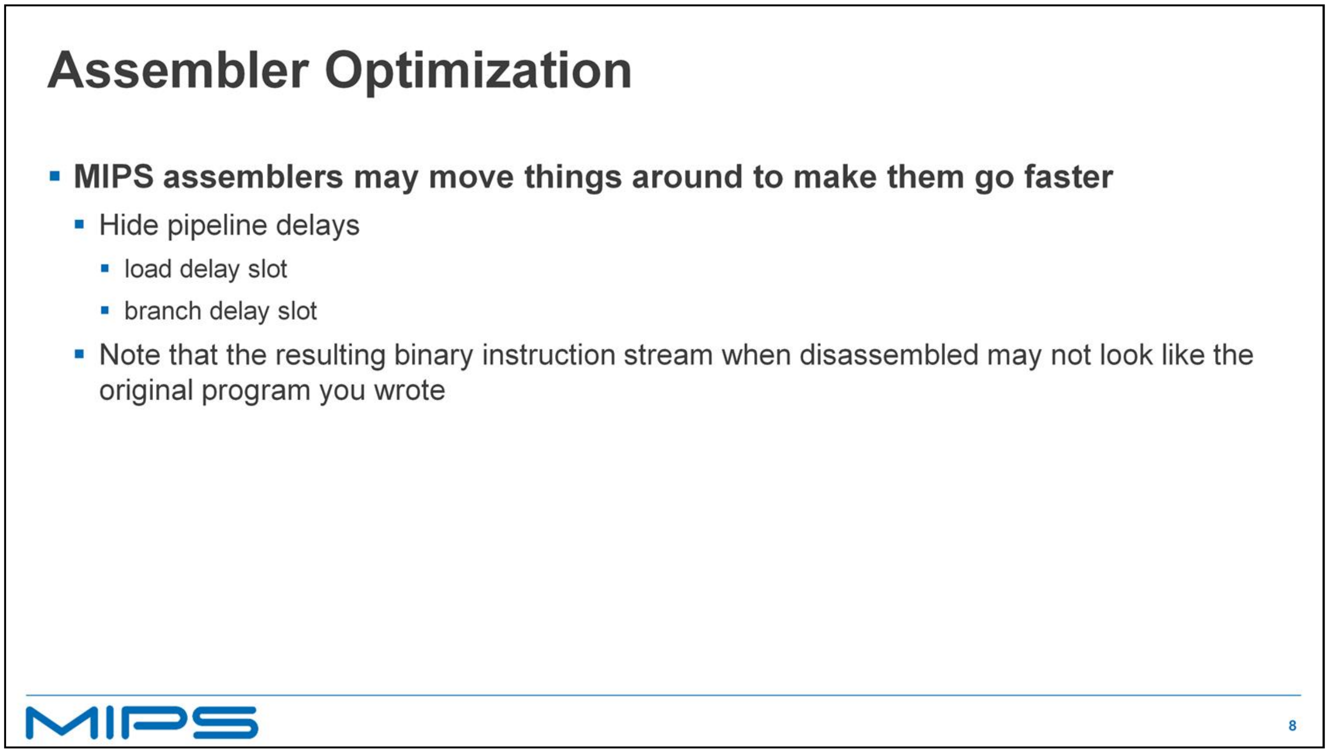
```
addu $4, 0x12345  -> lui       $at, 0x1
ori   $at, $at, 0x2345
addu $4, $4, $at
```

MIPS 7

It will also simplify the writing of immediate values that are larger than 16 bits. For example

+ in using the add unsigned instruction it is straight forward when using a 16 bit value

+ the assembler make it just as easy for to use immediate values larger than 16 bits and converts the one instruction into the three to do the job.



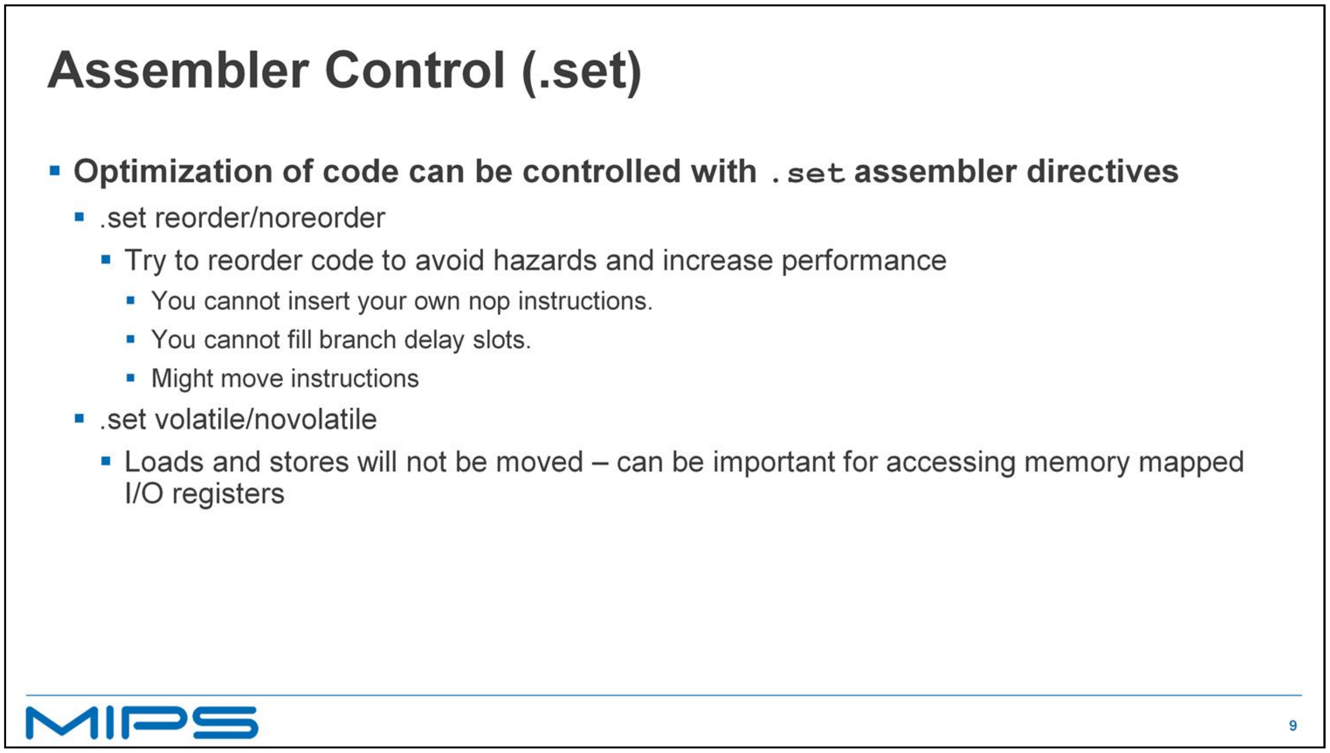
Assembler Optimization

- **MIPS assemblers may move things around to make them go faster**
  - Hide pipeline delays
    - load delay slot
    - branch delay slot
  - Note that the resulting binary instruction stream when disassembled may not look like the original program you wrote

MIPS 8

+ The assembler will optimize your assemble code by filling in the load, branch and jump delay slots with an appropriate instruction when it can find one

+ this will cause the dissemble to look different than the code you have written



Assembler Control (.set)

- Optimization of code can be controlled with `.set` assembler directives
  - .set reorder/noreorder
    - Try to reorder code to avoid hazards and increase performance
      - You cannot insert your own nop instructions.
      - You cannot fill branch delay slots.
      - Might move instructions
  - .set volatile/novolatile
    - Loads and stores will not be moved – can be important for accessing memory mapped I/O registers

9

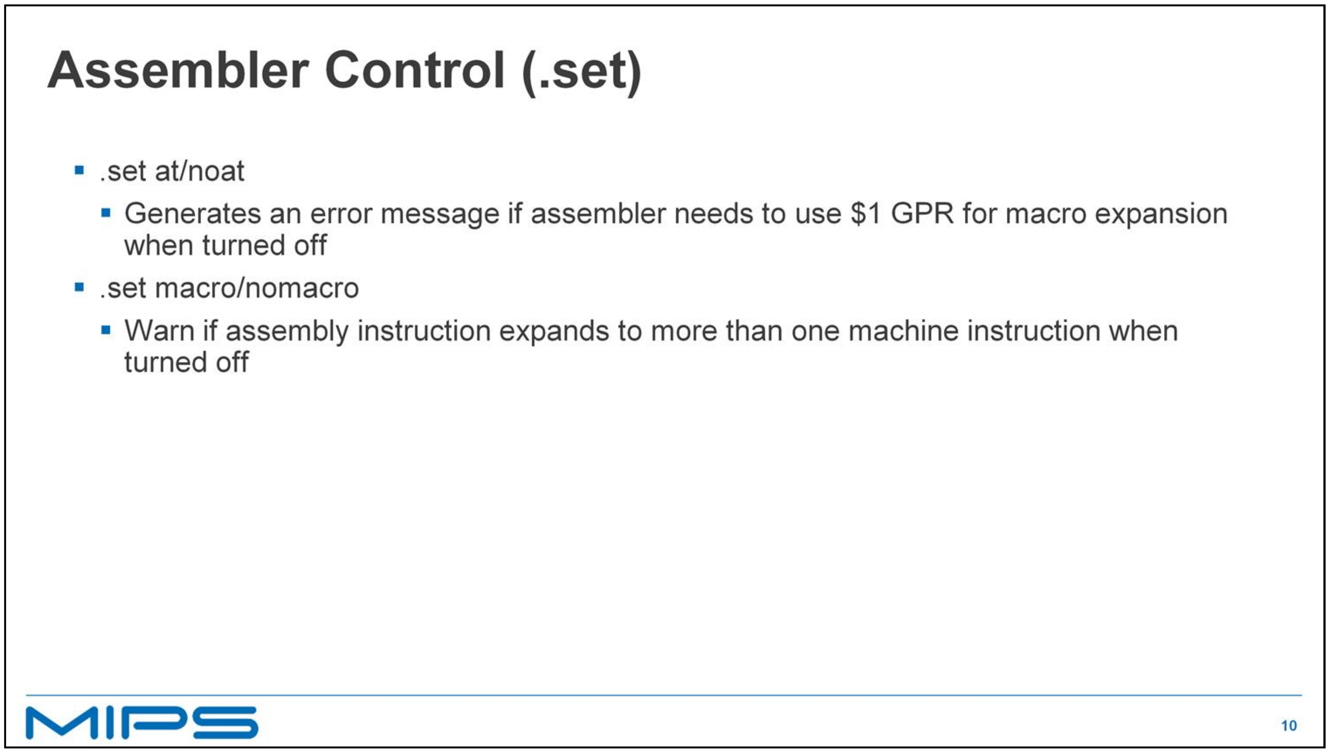
You can control some of the optimization s through the use of the .set directive

+ The assembler defaults to reorder which means it will control the instructions branch and jump delay slots. It will also move instructions around to avoid load delay stalls. If you want to hand tune some of your code you can you can tell the assembler to leave it alone by placing a .set noreorder directive at the start of the code you don't want touched and then resume optimizations by using the .set reorder directive when you want it to start optimizing again.

+ Sometimes you may have a section of code that you don't want loads or stores reordered or removed all together because you are reading or writing device registers or other volatile memory locations. You can tell the assembler to not optimize loads and stores by starting that section of code with the .set volatile directive and then use .set novolatile when you want load store optimization to continue.

+ If you find you need to use the assemble temporary register in a section of code you can have the assembler give you an error message when it needs the AT register for a macro by the .set noat directive. You can resume the use of AT for macros by using .set at.

+ when you are tuning code you might want the assembler to use all optimizations but warn you when it had to generate more code due to macro expansion. You can do this by using the .set no macro directive.



Assembler Control (.set)

- .set at/noat
  - Generates an error message if assembler needs to use $1 GPR for macro expansion when turned off
- .set macro/nomacro
  - Warn if assembly instruction expands to more than one machine instruction when turned off

MIPS 10

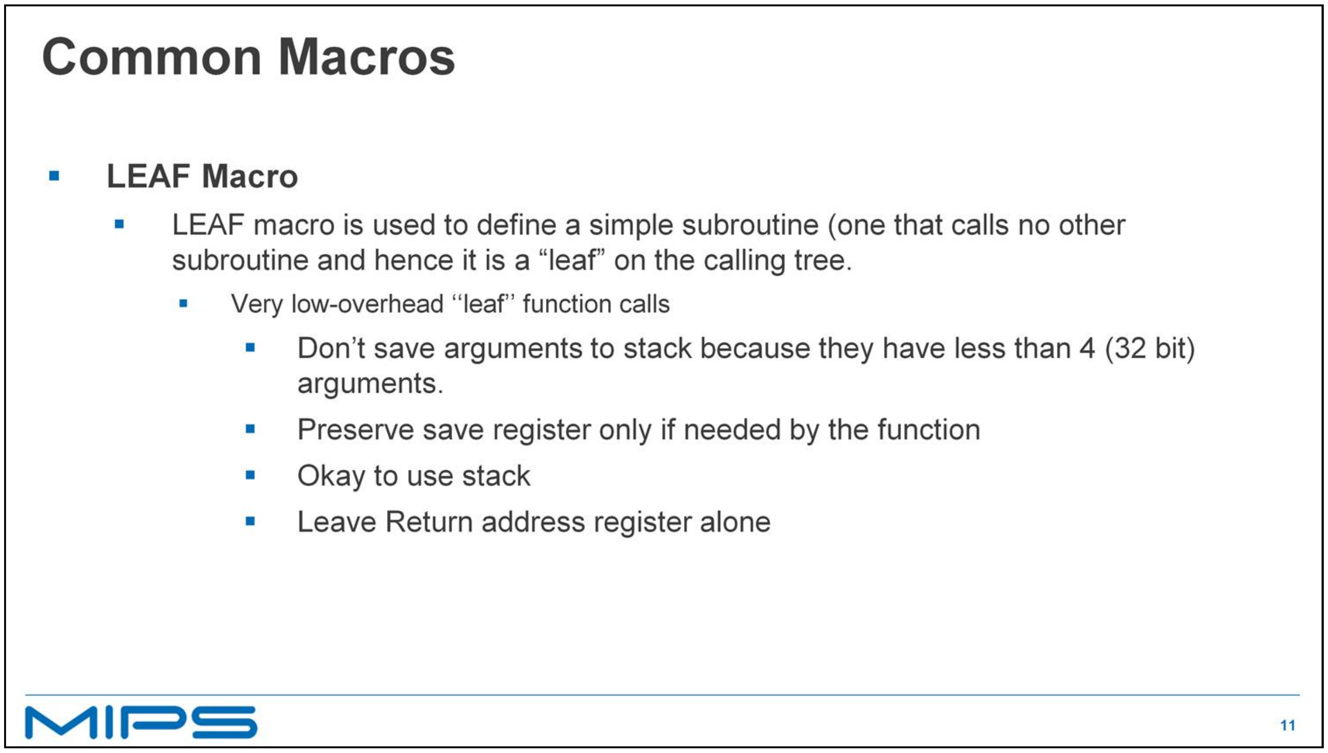
You can control some of the optimization s through the use of the .set directive

+ The assembler defaults to reorder which means it will control the instructions branch and jump delay slots. It will also move instructions around to avoid load delay stalls. If you want to hand tune some of your code you can you can tell the assembler to leave it alone by placing a .set noreorder directive at the start of the code you don't want touched and then resume optimizations by using the .set reorder directive when you want it to start optimizing again.

+ Sometimes you may have a section of code that you don't want loads or stores reordered or removed all together because you are reading or writing device registers or other volatile memory locations. You can tell the assembler to not optimize loads and stores by starting that section of code with the .set volatile directive and then use .set novolatile when you want load store optimization to continue.

+ If you find you need to use the assemble temporary register in a section of code you can have the assembler give you an error message when it needs the AT register for a macro by the .set noat directive. You can resume the use of AT for macros by using .set at.

+ when you are tuning code you might want the assembler to use all optimizations but warn you when it had to generate more code due to macro expansion. You can do this by using the .set no macro directive.


Common Macros

- **LEAF Macro**
  - LEAF macro is used to define a simple subroutine (one that calls no other subroutine and hence it is a "leaf" on the calling tree.
    - Very low-overhead ''leaf'' function calls
      - Don't save arguments to stack because they have less than 4 (32 bit) arguments.
      - Preserve save register only if needed by the function
      - Okay to use stack
      - Leave Return address register alone

11

A function that does not call another function is called a Leaf function. You could also think of it as a dead end function because it goes no where else. It can also be used if there is no stack manipulation. For example boot code that starts a the boot exception vector which will not return.

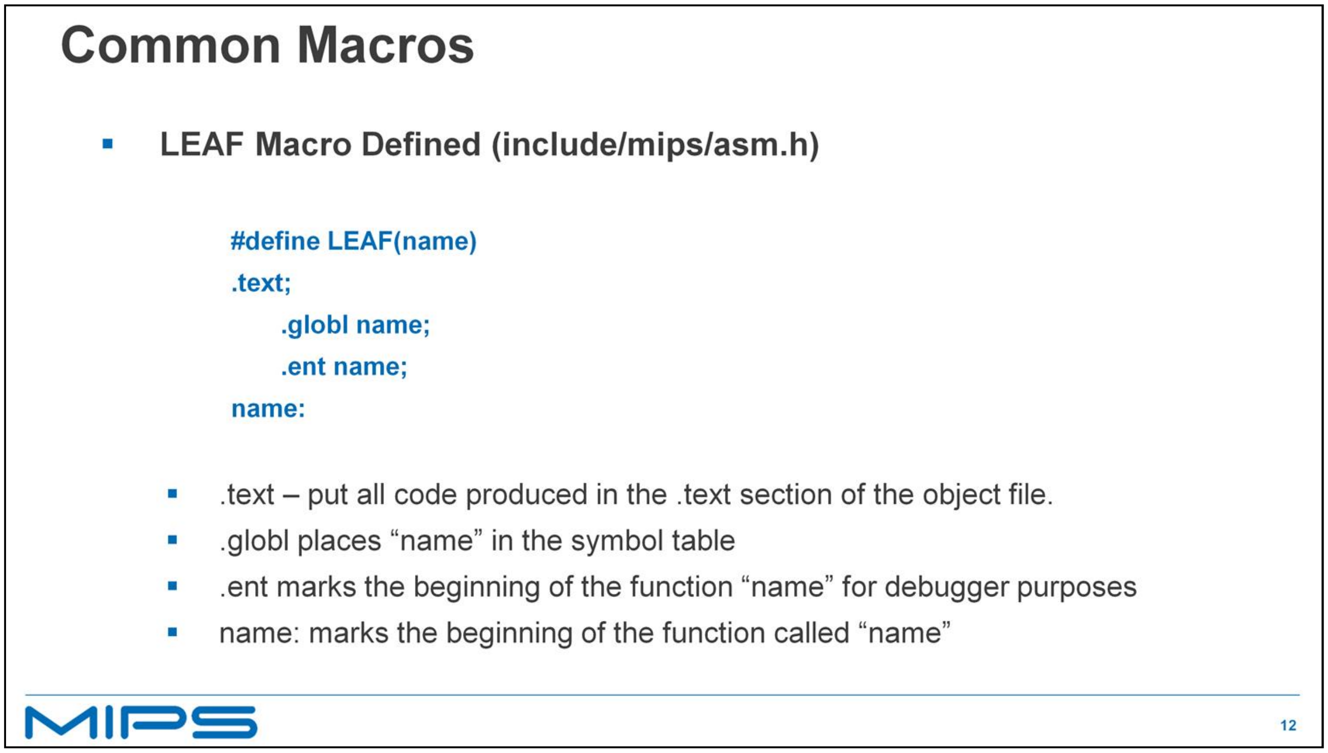
+ There is very little over head needed for a leaf function.

+ Since your not making any function calls you only need to worry about preserving registers that you actually use. For example in a Leaf function you don't have to worry about saving the arguments

+ or the save registers because you usually can get by with using the temporary registers

+ You can use the stack for storage as long as you preserve the stack pointer register.

+ Remember to leave the return address in register ra and return directly to it.



Common Macros

- LEAF Macro Defined (include/mips/asm.h)

```
#define LEAF(name)
.text;
    .globl name;
    .ent name;
name:
```

- .text – put all code produced in the .text section of the object file.
- .globl places “name” in the symbol table
- .ent marks the beginning of the function “name” for debugger purposes
- name: marks the beginning of the function called “name”

12

There are several code macros you can use in the asm.h include file.

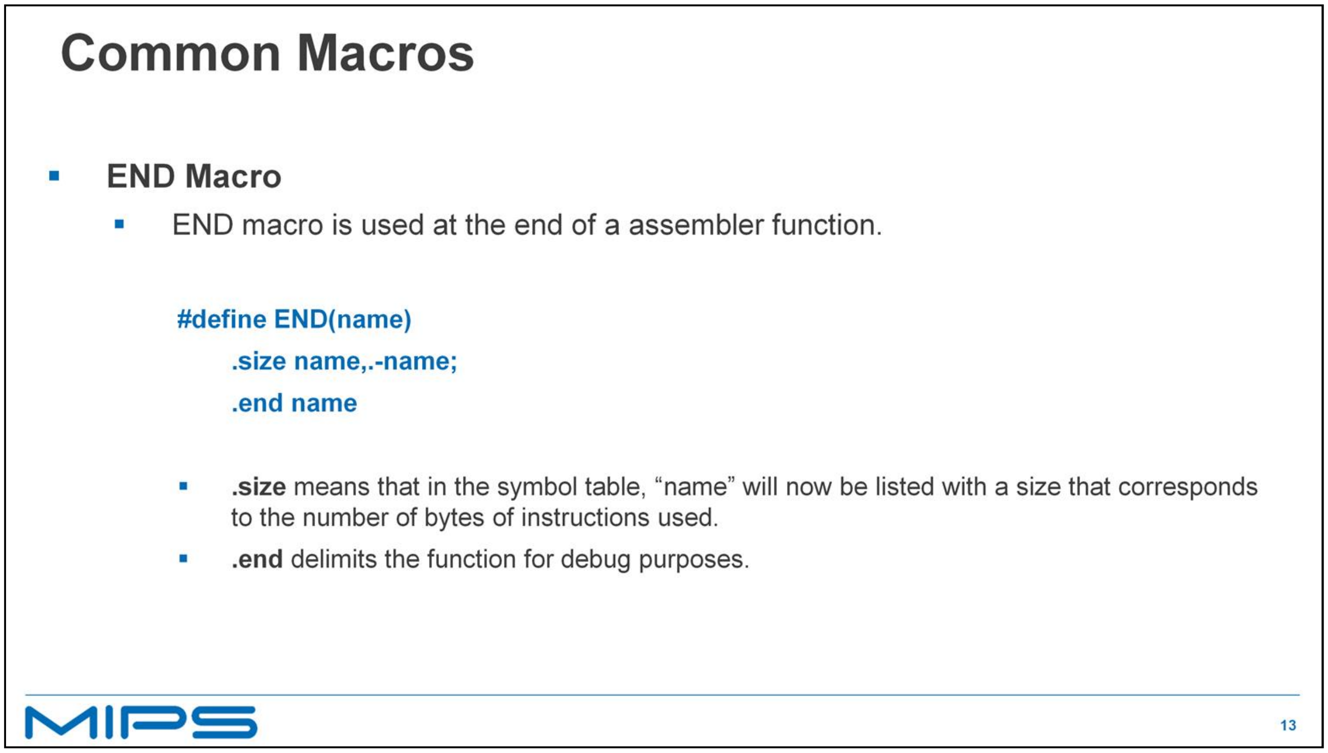
+ One of them is this covenant macro for Leaf functions. Let me explain what it does:

+ first the .text directive tells the assembler to put the code in the text section of the object file. The text section is normally where the assembler put the machine instructions.

+ . globl places the name of the function in the symbol table which makes the symbol “*name”* globally visible to the linker and available to any file that is linked with this assemble file.

+**.**ent has no effect on the code produced but tells the assembler to mark this point as the beginning of a function called “name” and to use that information in debug records.

+name colon labels this point in the assembler’s output with this functions name and identifies the address that other functions will use in calling this one.


Common Macros

- **END Macro**
  - END macro is used at the end of a assembler function.

```
#define END(name)
      .size name,.-name;
      .end name
```

  - **.size** means that in the symbol table, “name” will now be listed with a size that corresponds to the number of bytes of instructions used.
  - **.end** delimits the function for debug purposes.

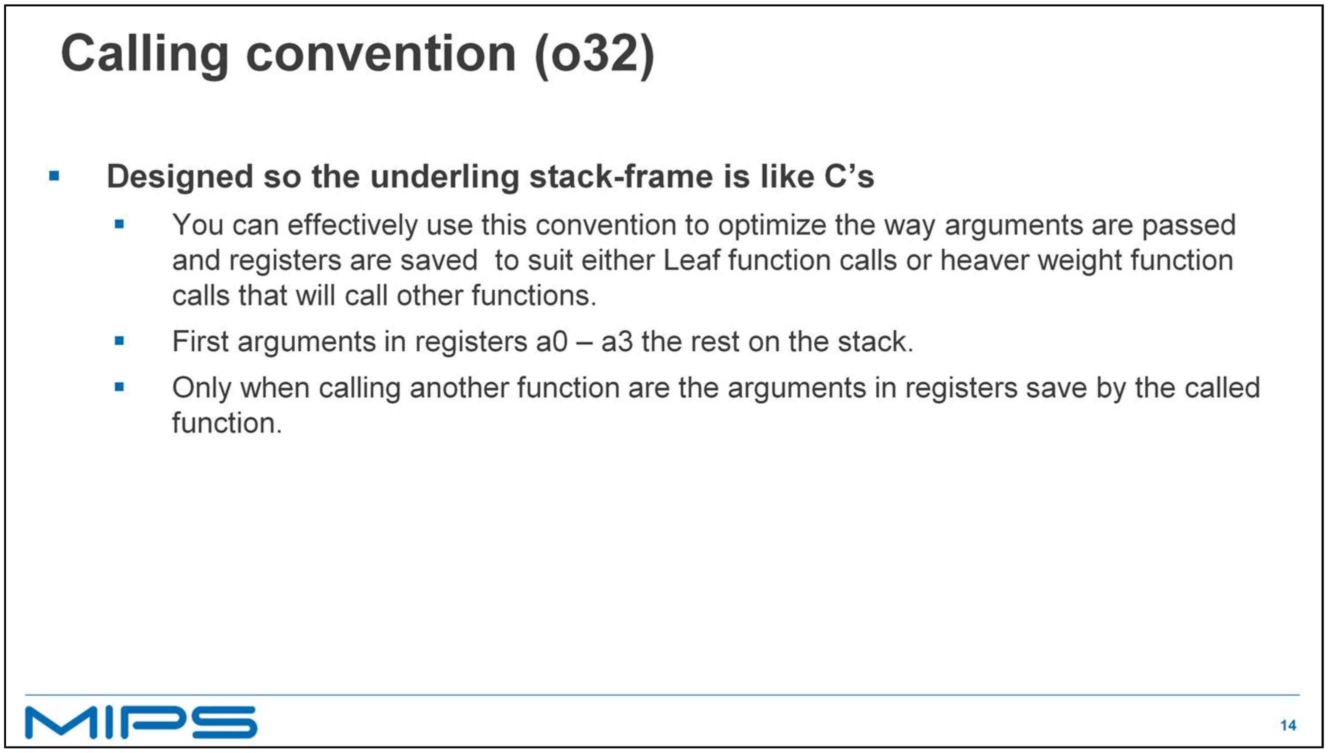
13

Another convent macro is the END macro which you can use at the end of an assemble function.

+ It looks like this

+ .size Make it easy for the assemble to compute the total number of bytes the function will use and place the size with the corresponding function name in the symbol table.

+ .end is used by debuggers to mark the end of the function



Calling convention (o32)

- **Designed so the underling stack-frame is like C's**
  - You can effectively use this convention to optimize the way arguments are passed and registers are saved to suit either Leaf function calls or heaver weight function calls that will call other functions.
  - First arguments in registers a0 – a3 the rest on the stack.
  - Only when calling another function are the arguments in registers save by the called function.

MIPS 14

Up to this point you have seen which registers need to be preserved from one function to another and where to find the current stack pointer and the return address.

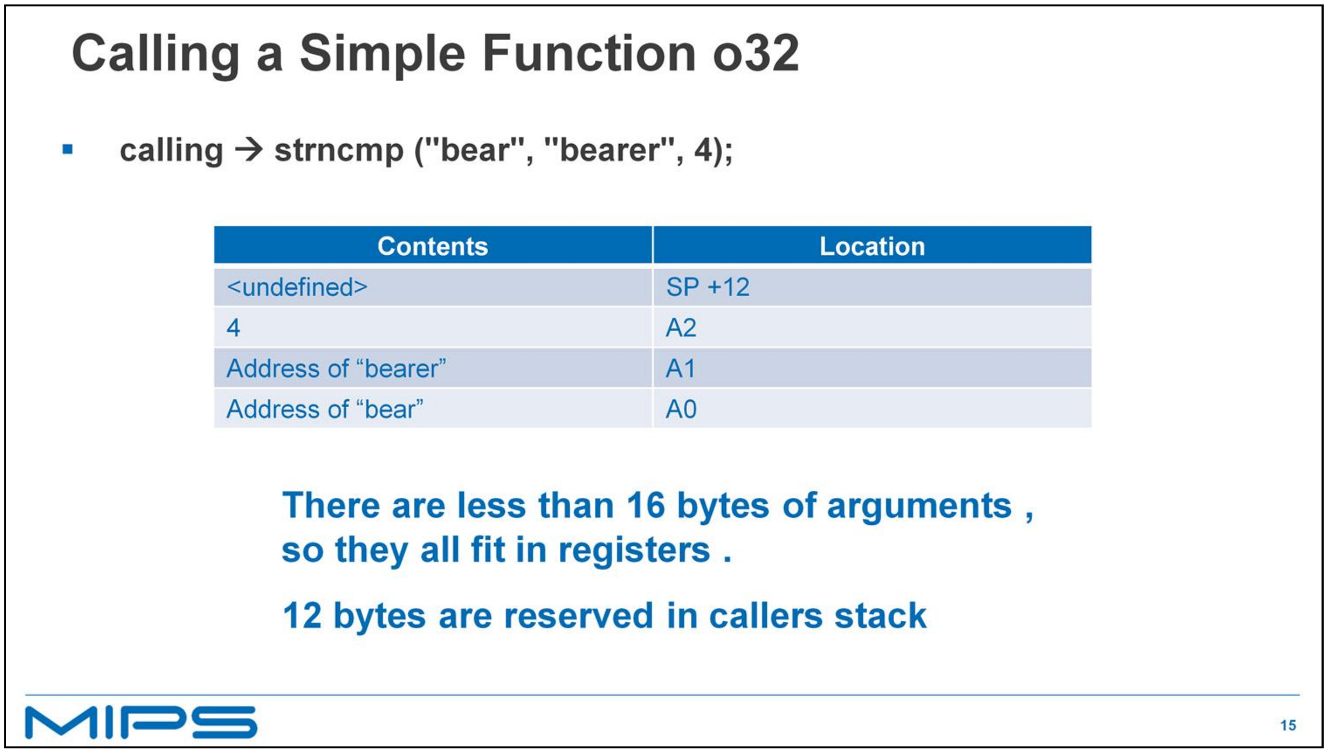
Now I will turn your attention to argument passing. In order for your assembler code to be called and to call other functions you need to follow an agreed upon convention. I have been using the o32 convention in this section to explain the register layout and I will continue using it to explain the calling convention.

+The calling convention is designed to be compatible with how C functions call each other.

+ To optimize argument passing the o32 convention uses four general purpose registers register a0 through a3, to pass arguments to any function. It still allocates the stack space for these arguments but it does not actually copy the arguments from the argument registers to those location. The space is allocated in case the called function will need to save its arguments to the stack.

+ When your assemble code gets called it can find its first arguments in the argument register and the rest in their expected location on the stack.

+ When your assemble code calls another function it needs to save the arguments it was called with on its stack to their pre allocated stack location locations. Your function then needs to place the arguments it is passing to the function it is calling in the argument registers and any overflow into a new stack frame.



Calling a Simple Function o32

- calling → strncmp ("bear", "bearer", 4);

Contents	Location
<undefined>	SP +12
4	A2
Address of "bearer"	A1
Address of "bear"	A0

There are less than 16 bytes of arguments , so they all fit in registers .

12 bytes are reserved in callers stack

Let see how this works in practice

+ here I have a simple function call with 3 arguments

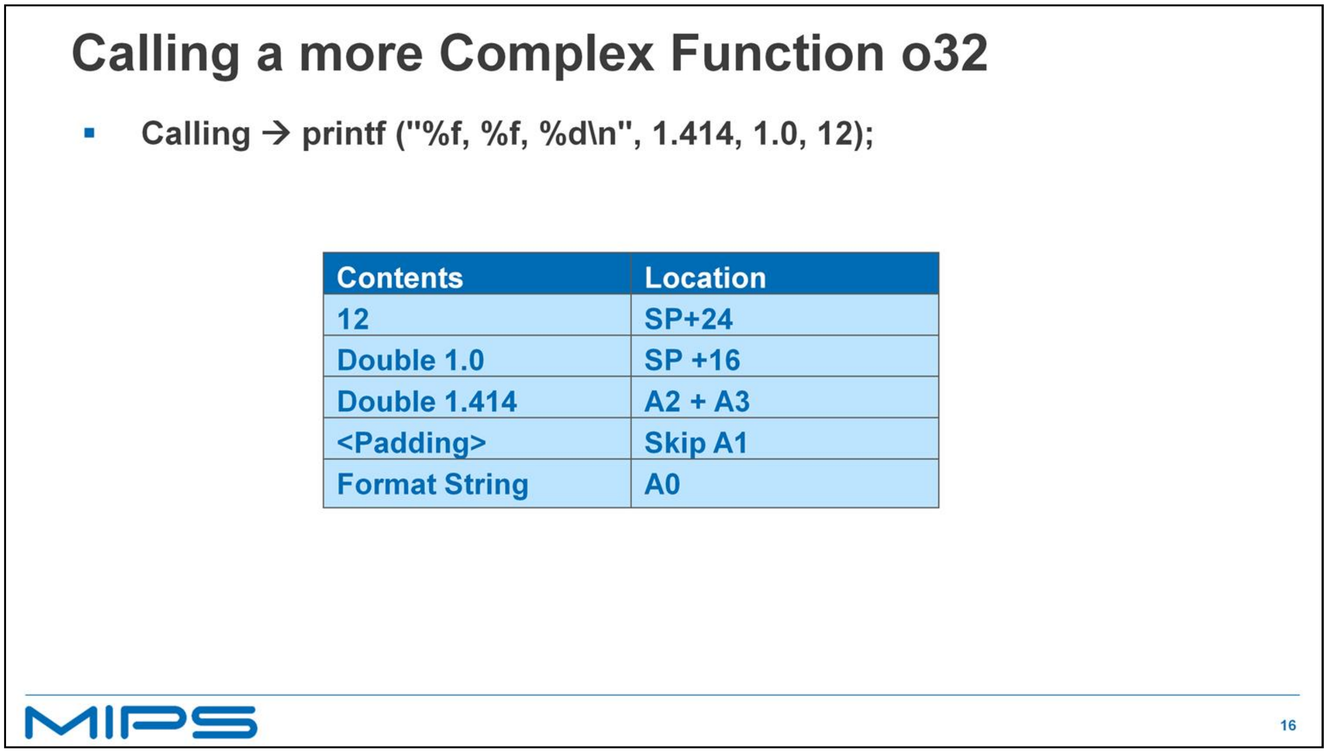
+ The first argument will be a pointer to a string and placed into argument register 0.

+ The second argument is also a pointer to a string and placed into argument register 1.

+ The third argument is a integer and it will go into argument A2

+ This was easy because we only had 3 arguments and they all aligned and fit into the first 3 argument registers.

+Note the stack pointer will point to the word reserved for the first argument, Stack pointer + 4 to the second and stack pointer +8 the third argument but no data will actually be stored in these locations.



Calling a more Complex Function o32

- Calling → printf ("%f, %f, %d\n", 1.414, 1.0, 12);

Contents	Location
12	SP+24
Double 1.0	SP +16
Double 1.414	A2 + A3
<Padding>	Skip A1
Format String	A0

16

Now lets see what a more complicated function call would look like

+ Say your assemble code was going to call printf . Printf is a complicated example because not only are the number of arguments variable but so is the size of the arguments. This all makes the example more illuminating, so listen closely.

In this example the printf call has four arguments.

+ The first argument is a pointer to the format string.

+ it goes into the a0 argument register. There is also room allocated on the stack for this argument in case the function being called needs to save it. Note the stack pointer should be aligned to a double word boundary.

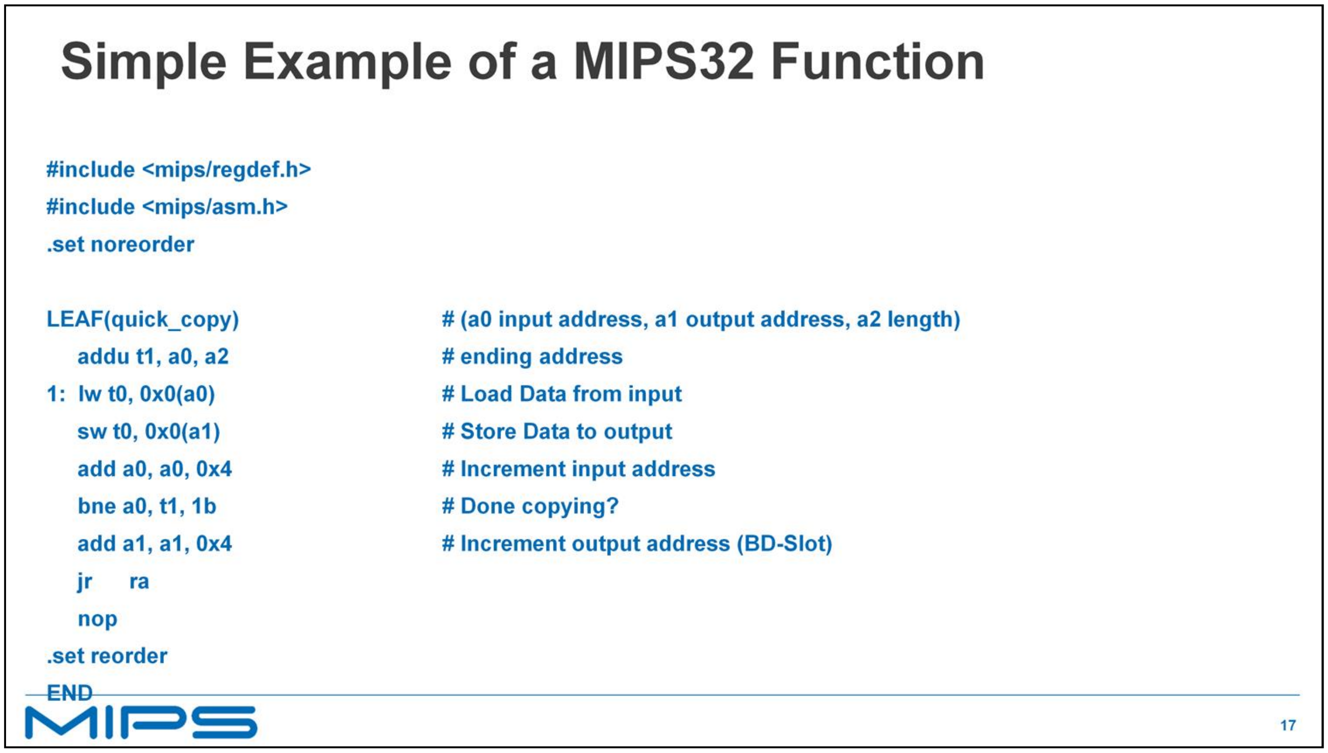
Next is a floating point number of type double which, for MIPS that means, 2 consecutive words aligned on a double word boundary. Since the stack pointer is aligned to a double word boundary then to align this arguments on the stack means

+ skipping the next word to get the proper alignment. To keep everything even you also need to skip the use of the a1 register.

+This all means that the second argument will be placed in argument registers a2 and a3.

+ now we are out of registers to store arguments so the next argument will be copied to the stack. It is also a double and since we need to allocate space for the first two arguments the double word will start a the new stack pointer plus 16.

+ The last argument is a integer and will go on the stack a position stack pointer +24.



Simple Example of a MIPS32 Function

```
#include <mips/regdef.h>
#include <mips/asm.h>
.set noreorder

LEAF(quick_copy)              # (a0 input address, a1 output address, a2 length)
    addu t1, a0, a2           # ending address
1:  lw t0, 0x0(a0)            # Load Data from input
    sw t0, 0x0(a1)            # Store Data to output
    add a0, a0, 0x4           # Increment input address
    bne a0, t1, 1b            # Done copying?
    add a1, a1, 0x4           # Increment output address (BD-Slot)
    jr    ra
    nop
.set reorder
END
```

17

Now we are ready to look at a simple assemble function. This function will copy an input buffer to an output buffer with the assumption that the input and output buffers are word aligned and the number of bytes to copy is equal to a even number of words.

+ First I have included two include files:

The regdef.h file allows use to use register names such as a0 instead of register numbers

And asm.h contains the LEAF and END macros

+ Next I use the .set noreorder directive because I want to control the order of the code and the use of the branch delay slot.

+ For the start of the function I use the LEAF macro with the name of the function to start thing off

+ I'll compute the ending address of the input buffer by adding the length argument to the address of the buffer argument.

+ next load input data into temporary 0 register

+ then store it to the output buffer

+ and increment the input address

+ test to see if we are at the end of the copy and branch back to the line that is labeled 1 if we are not

+ while the branch address is being calculated the branch delay slot will execute so I use the slot to increment the output address

+ when the above branch is false and the input address is equal to the output address the execution will fall through the branch and I will use the jump register instruction to return to the calling function.

+ I have nothing for the jump delay slot to do so I put a no opp there

+ then I turn reording back on so any code that follows will be optimized.

+ and last use the END macro to show the end of the function.