

CSE 31

Computer Organization

Lecture 11 – MIPS: Procedures (3)

Logical Operators



Announcement

- ▶ Lab #5 this week
 - Due next week
- ▶ HW #3 (from zyBooks)
 - Due Monday (10/8)
- ▶ Project #1
 - Due Monday (10/22)
 - Don't start late, you won't have time!
- ▶ Reading assignment
 - Chapter 3.1 – 3.7, 3.9 of zyBooks (Reading Assignment #3)
 - Make sure to do the Participation Activities
 - Due Wednesday (10/3)

Announcement

- ▶ Midterm exam Wednesday (10/3, postponed)
 - Lectures 1 – 7
 - HW #1 and #2
 - Closed book
 - 1 sheet of note (8.5" x 11")
 - Sample exam online

Instruction Support for Functions

- ▶ Syntax for `jr` (jump register):

`jr register`

- ▶ Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to.
- ▶ Very useful for function calls:
 - `jal` stores return address in register (`$ra`)
 - `jr $ra` jumps back to that address

Nested Procedures (1/2)

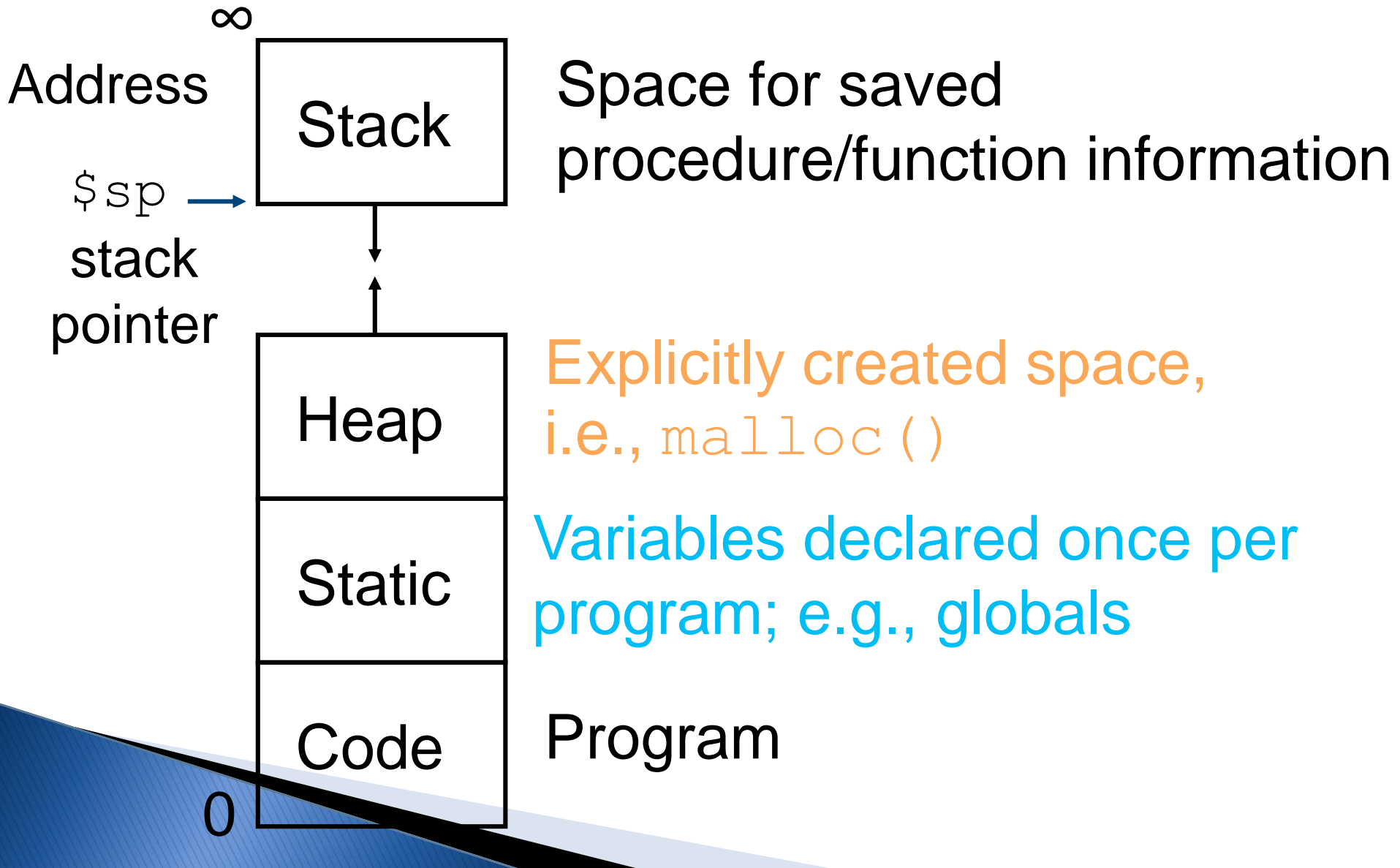
```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

- ▶ Something called **sumSquare**, now **sumSquare** is calling **mult**.
- ▶ So there's a value in `$ra` that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**.
- ▶ Need to save **sumSquare** return address before call to **mult**.
 - How to prevent the return address from being over-written?

Nested Procedures (2/2)

- ▶ In general, may need to save some other info in addition to `$ra`.
- ▶ When a C program is run, there are 3 important memory areas allocated:
 - **Static**: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
 - **Heap**: Variables declared dynamically via `malloc`
 - **Stack**: Space to be used by procedure during execution; this is where we can save register values

C memory Allocation review



Using the Stack (1/2)

- ▶ We have a register **\$sp** which always points to the last used space in the stack (top of stack).
- ▶ To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- ▶ So, how do we compile this?

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```


Using the Stack (2/2)

► Hand-compile

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y; }
```

sumSquare:

“push”

```
    addi $sp,$sp,-8    # space on stack  
    sw $ra, 4($sp)     # save ret addr  
    sw $a1, 0($sp)     # save y  
    add $a1,$a0,$zero  # mult(x,x)  
    jal mult           # call mult  
    lw $a1, 0($sp)     # restore y  
    add $v0,$v0,$a1    # mult()+y  
    lw $ra, 4($sp)     # get ret addr  
    addi $sp,$sp,8     # restore stack  
    jr $ra
```

“pop”

mult: ...

Steps for Making a Procedure Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. **jal** call
4. Restore values from stack.

Basic Structure of a Function

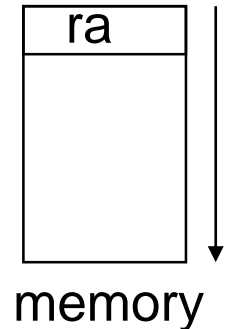
Prologue

```
entry_label:  
addi $sp,$sp, -framesize  
sw $ra, framesize-4($sp)  # save $ra  
save other regs if need be
```

Body ... (call other functions...)

Epilogue

```
restore other regs if need be  
lw $ra, framesize-4($sp)  # restore $ra  
addi $sp,$sp, framesize  
jr $ra
```



Rules for Procedures

- ▶ Called with a **jal** instruction
 - returns with a **jr \$ra**
- ▶ Accepts up to 4 arguments in **\$a0**, **\$a1**, **\$a2** and **\$a3**
- ▶ Return value is always in **\$v0** (and if necessary in **\$v1**)
- ▶ Must follow **register conventions**
 - What are they?

MIPS Registers

The constant 0

Reserved for Assembler

Return Values

Arguments

Temporary

Saved

More Temporary

Used by Kernel

Global Pointer

Stack Pointer

Frame Pointer

Return Address

\$0

\$1

\$2-\$3

\$4-\$7

\$8-\$15

\$16-\$23

\$24-\$25

\$26-27

\$28

\$29

\$30

\$31

\$zero

\$at

\$v0-\$v1

\$a0-\$a3

\$t0-\$t7

\$s0-\$s7

\$t8-\$t9

\$k0-\$k1

\$gp

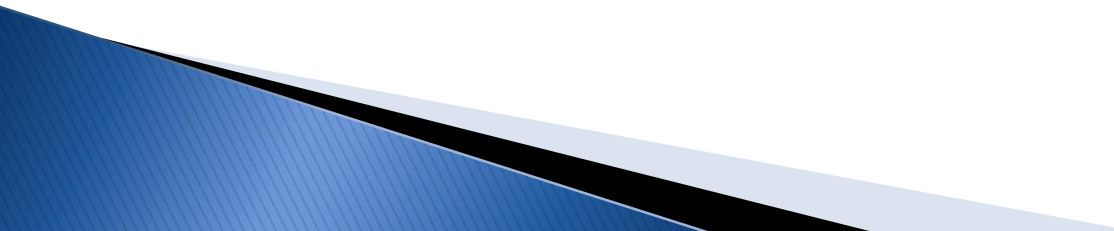
\$sp

\$fp

\$ra

Use names for registers -- code is clearer!

Other Registers

- ▶ **\$at**: may be used by the assembler at any time; unsafe to use
 - ▶ **\$k0–\$k1**: may be used by the OS at any time; unsafe to use
 - ▶ **\$gp**, **\$fp**: don't worry about them
- 

Register Conventions (1/4)

- ▶ CalleR: the calling function (where you call a function)
- ▶ CalleE: the function being called
- ▶ When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- ▶ **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**jal**) and which may be changed.

Register Conventions (2/4) – saved

- ▶ **\$0: No Change.** Always 0.
- ▶ **\$s0-\$s7: Restore if you change.** Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values **before returning**. i.e. callee's job to restore.
- ▶ **\$sp: Restore if you change.** The stack pointer must point to the same place before and after the `jal` call, or else the caller won't be able to restore values from the stack.
- ▶ HINT -- All saved registers start with **S**!

Register Conventions (3/4) – volatile

- ▶ **\$ra: Can Change.**
 - The jal call itself will change this register. Caller needs to save on stack before next call (nested call).
- ▶ **\$v0-\$v1: Can Change.**
 - These will contain the new returned values.
- ▶ **\$a0-\$a3: Can change.**
 - These are volatile argument registers. Caller needs to save if they are needed after the call.
- ▶ **\$t0-\$t9: Can change.**
 - That's why they're called temporary: any procedure may change them at any time. Caller needs to save if they'll need them afterwards.

Register Conventions (4/4)

- ▶ What do these conventions mean?
 - If function **R** calls function **E**, then function **R** must save any **temporary** registers that it may be using onto the stack **before making** a **jal** call.
 - Function **E** must save any **S (saved) registers** it intends to use before garbling up their values. It must restore any modified S registers **before returning** back to **R**
- ▶ Remember: **caller/callee** need to save only **temporary/saved** registers **they are using**, not all registers.

Example: Fibonacci Numbers 1/7

- ▶ The **Fibonacci** numbers are defined as follows: $F(n) = F(n - 1) + F(n - 2)$,
 $F(0)$ and $F(1)$ are defined to be 1
- ▶ In C we have:

```
int fib(int n) {  
    if (n == 0)  
        return 1;  
    if (n == 1)  
        return 1;  
    return (fib(n - 1) + fib(n - 2));  
}
```

Example: Fibonacci Numbers 2/7

- ▶ Now, let's translate this to MIPS!
- ▶ You will need space for three words on the stack
- ▶ The function will use one `$s` register, `$s0`
- ▶ Write the Prologue:

`fib:`

```
addi $sp, $sp, -12    # Space for 3 words
sw  $ra, 8($sp)        # Save return address
sw  $s0, 4($sp)        # Save s0 (value from caller)
```

Example: Fibonacci Numbers 3/7

- Now write the Epilogue:

fin:

lw \$s0, 4(\$sp)

Restore \$s0 for caller

lw \$ra, 8(\$sp)

Restore return address

addi \$sp, \$sp, 12

Pop the stack frame

jr \$ra

Return to caller

Example: Fibonacci Numbers 4/7

```
int fib(int n) {  
    if(n == 0)      /* Base case 0*/  
        return 1;  
    if(n == 1)      /* Base case 1 */  
        return 1;  
    return (fib(n - 1) + fib(n - 2));  
}
```

```
addi $v0, $zero, 1          # $v0 = 1  
beq  $a0, $zero, fin        # Base case 0  
addi $t0, $zero, 1          # $t0 = 1  
beq  $a0, $t0, fin          # Base case 1
```

Continued on next slide. . .

Example: Fibonacci Numbers 5/7

```
int fib(int n) {  
    if(n == 0)      /* Base case 0*/  
        return 1;  
    if(n == 1)      /* Base case 1 */  
        return 1;  
    return (fib(n - 1) + fib(n - 2));  
}
```

Write fib(n-1):

addi \$a0, \$a0, -1	# \$a0 = n - 1
sw \$a0, 0(\$sp)	# Need \$a0 after jal
jal fib	# fib(n - 1)
lw \$t0, 0(\$sp)	# restore \$a0
addi \$a0, \$t0, -1	# \$a0 = n - 2

Example: Fibonacci Numbers 6/7

```
int fib(int n) {  
    if(n == 0)      /* Base case 0*/  
        return 1;  
    if(n == 1)      /* Base case 1 */  
        return 1;  
    return (fib(n - 1) + fib(n - 2));  
}
```

Write `fib(n-2)` and `+` :

```
add $s0, $v0, $zero
```

Place `fib(n - 1)` somewhere
it won't get clobbered

```
jal fib
```

`fib(n - 2)`

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

`$v0 = fib(n-1) + fib(n-2)`

To the epilogue and beyond. . .

Example: Fibonacci Numbers 7/7

- Here's the complete code for reference:

```
fib:  addi $sp, $sp, -12
      sw $ra, 8($sp)
      sw $s0, 4($sp)
      addi $v0, $zero, 1
      beq $a0, $zero, fin
      addi $t0, $zero, 1
      beq $a0, $t0, fin
      addi $a0, $a0, -1
      sw $a0, 0($sp)
      jal fib
```

```
      lw $a0, 0($sp)
      addi $a0, $a0, -1
      add $s0, $v0, $zero
      jal fib
      add $v0, $v0, $s0
fin:  lw $s0, 4($sp)
      lw $ra, 8($sp)
      addi $sp, $sp, 12
      jr $ra
```

Quiz

```
int fact(int n) {  
    if(n == 0) return 1; else return (n*fact(n-  
1));  
}
```

When translating this to MIPS...

- 1) We **COULD** copy `$a0` to `$a1` (and then not store `$a0` or `$a1` on the stack) to store `n` across recursive calls.
- 2) We **MUST** save `$a0` on the stack since it gets changed.
- 3) We **MUST** save `$ra` on the stack since we need to know where to return to...

	1	2	3
a)	F	F	F
b)	F	F	T
c)	F	T	F
d)	F	T	T
e)	T	F	F
f)	T	F	T
g)	T	T	F
h)	T	T	T

Quiz

```
int fact(int n) {  
    if(n == 0) return 1; else return (n*fact(n-  
1));  
}
```

When translating this to MIPS...

- 1) We COULD copy `$a0` to `$a1` (and then not store `$a0` or `$a1` on the stack) to store `n` across recursive calls.
- 2) We MUST save `$a0` on the stack since it gets changed.
- 3) We MUST save `$ra` on the stack since we need to know where to return to...

We can implement it using iterations

	123
a)	FFF
b)	FFT
c)	FTF
d)	FTT
e)	TFF
f)	TFT
g)	TTF
h)	TTT

Quiz

```
r:  ...      # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
    ...      ### PUSH REGISTER(S) TO STACK?
    jal e     # Call e
    ...      # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
    jr $ra    # Return to caller of r

e:  ...      # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
    jr $ra    # Return to r
```

What does `r` have to push on the stack before “`jal e`”?

- a) 1 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- b) 2 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- c) 3 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- d) 4 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- e) 5 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)

Quiz

```
r:  ...      # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
    ...      ### PUSH REGISTER(S) TO STACK?
    jal e     # Call e
    ...      # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
    jr $ra    # Return to caller of r

e:  ...      # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
    jr $ra    # Return to r
```

What does `r` have to push on the stack before “`jal e`”?

Saved Volatile! -- need to push

- a) 1 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- b) 2 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- c) 3 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- ☺ d) 4 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- e) 5 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)

Summary

- ▶ Functions called with **jal**, return with **jr \$ra**.
- ▶ The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- ▶ Instructions we know so far...
 - Arithmetic: **add, addi, sub, addu, addiu, subu**
 - Memory: **lw, sw, lb, sb, lbu**
 - Decision: **beq, bne, slt, slti, sltu, sltiu**
 - Unconditional Branches (Jumps): **j, jal, jr**
- ▶ Registers we know so far
 - All of them!
 - There are CONVENTIONS when calling procedures!

Bitwise Operations

- ▶ So far, we've done arithmetic (**add**, **sub**, **addi**), mem access (**lw** and **sw**), & branches and jumps.
- ▶ All of these instructions view contents of register as a single quantity (e.g., signed or unsigned int)
- ▶ **New Perspective**: View register as 32 raw bits rather than as a single 32-bit number
 - Since registers are composed of 32 bits, wish to access individual bits (or groups of bits) rather than the whole.
- ▶ Introduce two new classes of instructions
 - **Logical & Shift Ops**

Logical Operators (1/3)

- ▶ Two basic logical operators:
 - AND: outputs 1 only if **all** inputs are 1
 - OR: outputs 1 if **at least one** input is 1
- ▶ Truth Table: standard table listing all possible combinations of inputs and resultant output

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical Operators (2/3)

- ▶ Logical Instruction Syntax:
 - 1 2,3,4
 - where
 - 1) operation name
 - 2) register that will receive value
 - 3) first operand (register)
 - 4) second operand (register) or immediate (numerical constant)
- ▶ In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
 - Again, rigid syntax, simpler hardware

Logical Operators (3/3)

▶ Instruction Names:

- **and, or**: Both of these expect the third argument to be a register
- **andi, ori**: Both of these expect the third argument to be an immediate

▶ MIPS Logical Operators are all **bitwise**, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.

- C: Bitwise AND is `&` (e.g., **`z = x & y;`**)
- C: Bitwise OR is `|` (e.g., **`z = x | y;`**)

Uses of Logical Operators (1/3)

- ▶ Note that **and**ing a bit with 0 produces a 0 at the output while **and**ing a bit with 1 produces the original bit.
- ▶ This can be used to create a **mask**.

- Example:

1011 0110 1010 0100 0011
0000 0000 0000 0000 0000

- The result of **and**ing these:

0000 0000 0000 0000 0000

mask:

1101 1001 1010
1111 1111 1111

1101 1001 1010

mask last 12 bits

Uses of Logical Operators (2/3)

- ▶ The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting to all 0s).
- ▶ Thus, the **and** operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
 - In particular, if the first bitstring in the above example were in `$t0`, then the following instruction would mask it:
`andi $t0, $t0, 0xFFF`

Uses of Logical Operators (3/3)

- ▶ Similarly, note that **O**ring a bit with **1** sets a **1** at the output while **O**ring a bit with **0** keeps the original bit.
- ▶ Often used to force certain bits to **1**s.
 - For example, if `$t0` contains `0x12345678`, then after this instruction:

```
ori    $t0, $t0, 0xFFFF
```


... `$t0` will contain `0x1234FFFF`
 - (i.e., the high-order 16 bits are untouched, while the low-order 16 bits are forced to **1**s).

Shift Instructions (review) (1/4)

- ▶ Move (shift) all the bits in a word to the left or right by a number of bits.

- Example: shift right by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

0000 0000 1001 0010 0011 0100 0101 0110



- Example: shift left by 8 bits

0001 0010 1011 0100 0101 0110 0111 1000

1011 0100 0101 0110 0111 1000 0000 0000



Shift Instructions (2/4)

- ▶ Shift Instruction Syntax:

1 2,3,4

...where

1) operation name

2) register that will receive value

3) first operand (register)

4) shift amount (constant < 32)

- ▶ MIPS shift instructions:

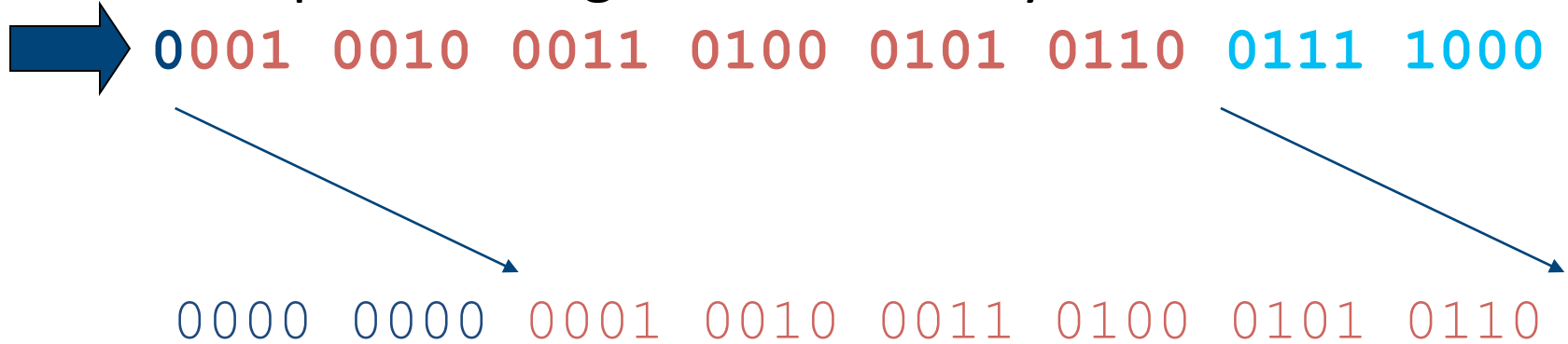
1. **sll** (shift left logical): shifts left and fills emptied bits with 0s

2. **srl** (shift right logical): shifts right and fills emptied bits with 0s

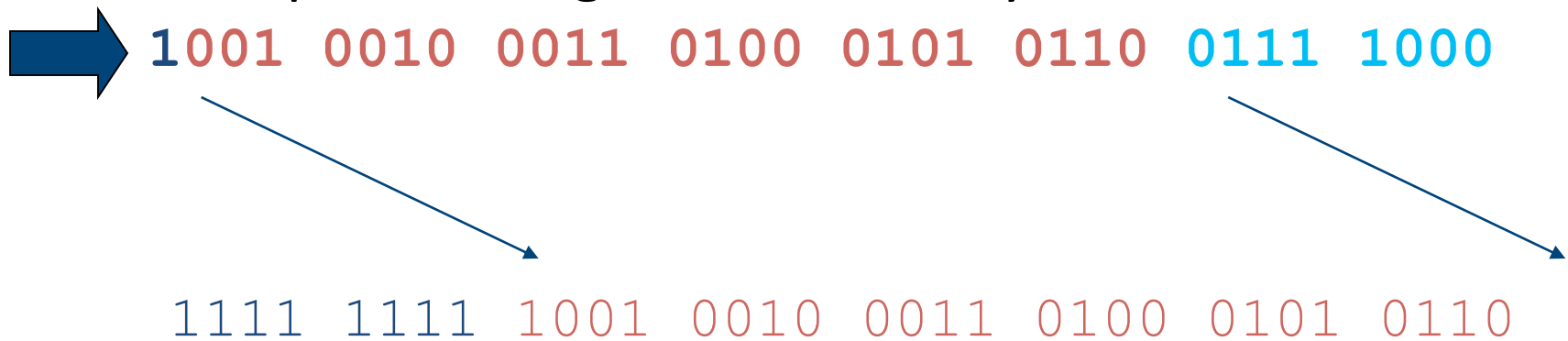
3. **sra** (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (3/4)

- ▶ Example: shift right arithmetic by 8 bits



- ▶ Example: shift right arithmetic by 8 bits



Shift Instructions (4/4)

- ▶ Since shifting is faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

`a *= 8;` (in C)

would compile to:

`sll $s0, $s0, 3` (in MIPS)

- ▶ Likewise, shift right to divide by powers of 2 (rounds towards $-\infty$)
 - remember to use `sra`

Summary

▶ Logical and Shift Instructions

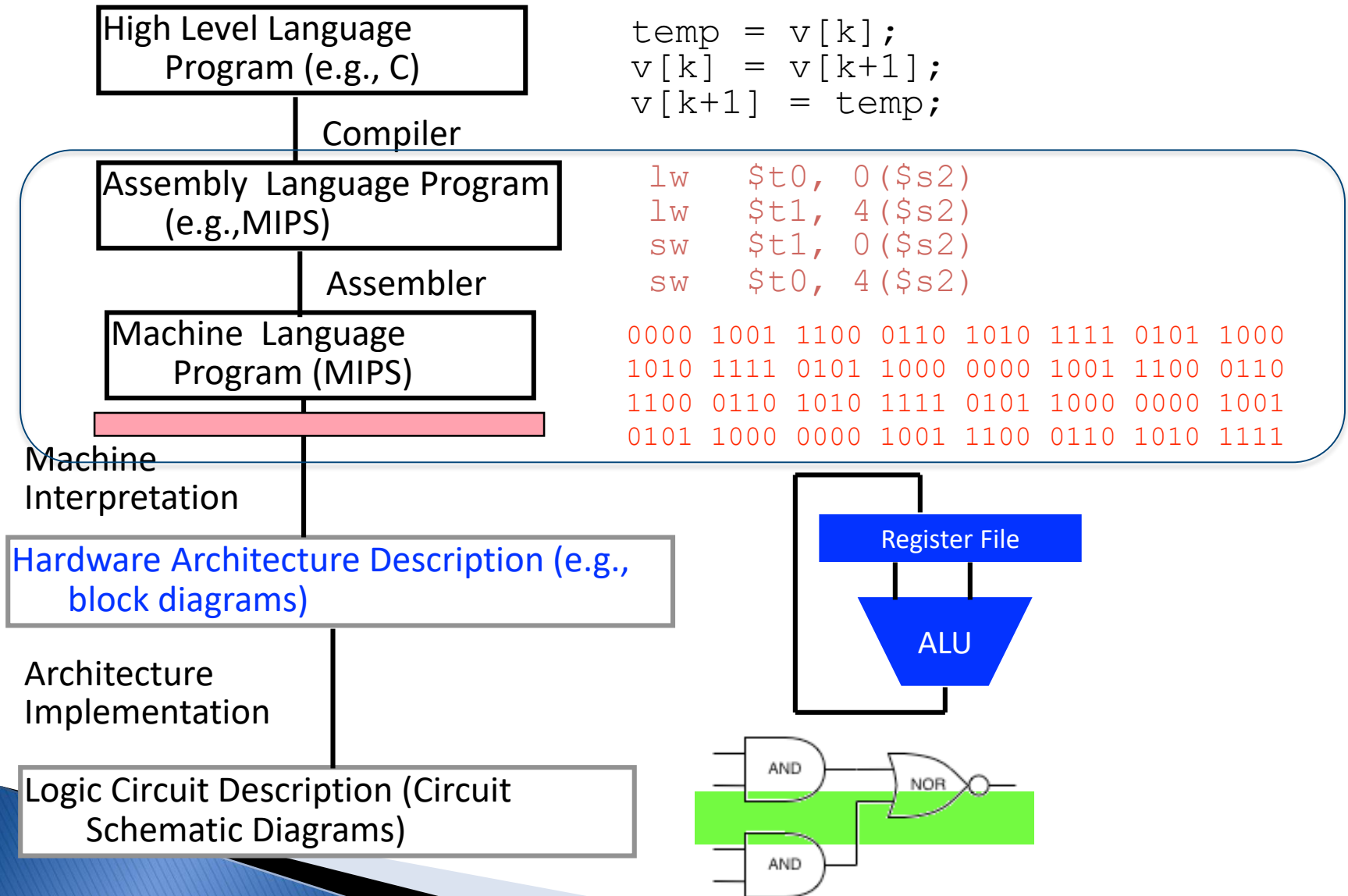
- Operate on bits individually, unlike arithmetic, which operate on entire word.
- Use to isolate fields, either by masking or by shifting back and forth.
- Use shift left logical, **sll**, for multiplication by powers of 2
- Use shift right logical, **srl**, for division by powers of 2 of unsigned numbers (**unsigned int**)
- Use shift right arithmetic, **sra**, for division by powers of 2 of signed numbers (**int**)

▶ New Instructions:

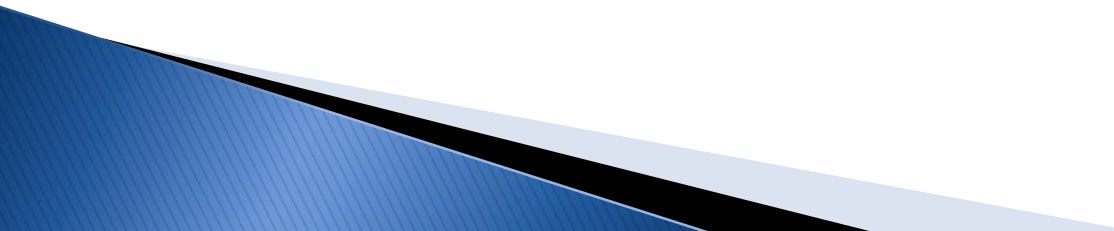
and, andi, or, ori, sll, srl, sra

▶ That's all you need to know about MIPS!

Levels of Representation (abstractions)



Overview – Instruction Representation

- ▶ Big idea: stored program
 - consequences of stored program
 - ▶ Instructions as numbers
 - ▶ Instruction encoding
 - ▶ MIPS instruction format for arithmetic instructions
 - ▶ MIPS instruction format for Immediate, Data transfer instructions
- 

Big Idea: Stored-Program Concept

- ▶ Where are programs stored when they are being run?
 - How are they stored in memory?
- ▶ Computers built on 2 key principles:
 - Instructions are represented as bit patterns - can think of these as numbers.
 - Therefore, entire programs can be stored in memory to be read or written just like data.
- ▶ Simplifies SW/HW of computer systems:
 - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- ▶ Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
 - both branches and jumps use these
- ▶ C pointers are just memory addresses: they can point to anything in memory
 - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- ▶ One register keeps address of instruction being executed: “Program Counter” (PC)
 - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Consequence #2: Binary Compatibility

- ▶ Programs are distributed in binary form
 - Programs bound to specific instruction set
 - Different version for **Macintoshes** and **PCs**
- ▶ New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
 - Leads to “backward compatible” instruction set evolving over time
 - Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

Instructions as Numbers (1/2)

- ▶ Currently all data we work with is in words (32-bit blocks):
 - Each register is a word.
 - **lw** and **sw** both access memory one word at a time.
- ▶ So how do we represent instructions?
 - Remember: Computer only understands 1s and 0s, so “**add \$t0, \$0, \$0**” is meaningless.
 - MIPS wants simplicity: since data is in words, make instructions into words too!

Instructions as Numbers (2/2)

- ▶ One word is 32 bits, so divide instruction word into “fields”.
- ▶ Each field tells processor something about the instruction.
- ▶ We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
 - I-format
 - J-format
 - R-format
 - What do these letters (I, J, R) stand for?