UNT
UNIVERSITY OF NORTH TEXAS

CSCE 5610
Computer System Architecture

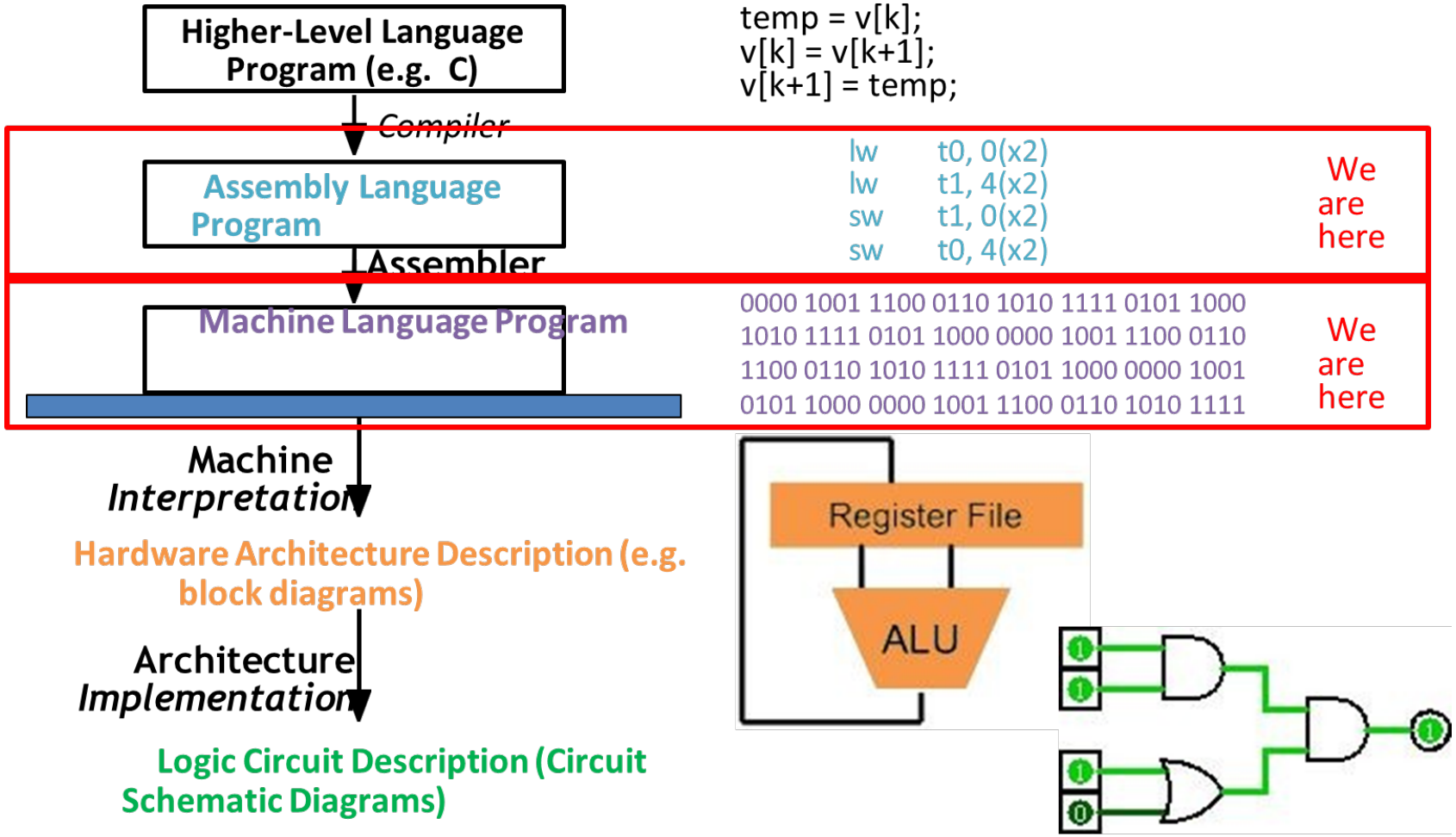
Machine Organization and Assembly Language

Content

- Assembly Language
- Machine Language

Credits of some slides go to University of Washington.

Great Idea #1: Abstraction



MIPS

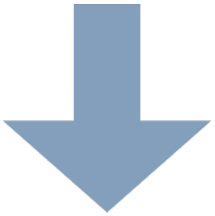
- In this class, we'll use the MIPS instruction set architecture (ISA) to illustrate concepts in assembly language and machine organization
  - Of course, the concepts are not MIPS-specific
  - MIPS is just convenient because it is real, yet simple (unlike x86)

- The MIPS ISA is still used in many places today. Primarily in embedded systems, like:
  - Various routers from Cisco
  - Game machines like the Nintendo 64 and Sony Playstation 2

From C to Machine Language

High-level language (C)

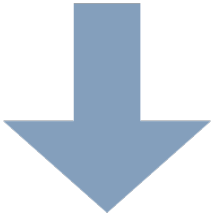
```
a = b + c;
```



Compiler

Assembly Language (MIPS)

```
add $16, $17, $18
```



Assembler

Binary Machine Language (MIPS)

```
01010111010101101...
```

What you will need to learn soon

- You must become "fluent" in MIPS assembly:
  — Translate from C to MIPS and MIPS to C

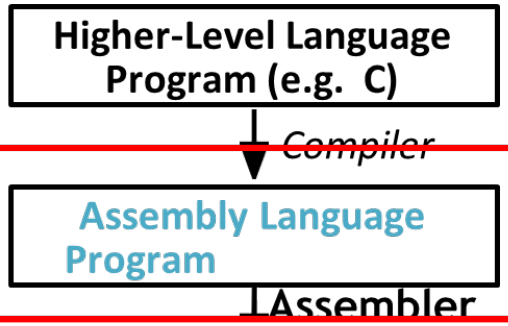
- Example problem: Write a recursive function

  Here is a function pow that takes two arguments (n and m, both 32-bit numbers) and returns n^m (i.e., n raised to the m^{th} power).

```
int pow(int n, int m) {
        if (m == 1)
                return n;
        return n * pow(n, m-1);
}
```

  Translate this into a MIPS assembly language function.

MIPS Assembly Language



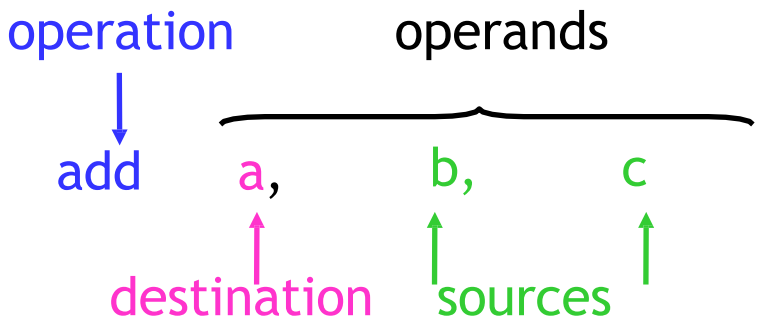

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw      t0, 0(x2)
lw      t1, 4(x2)
sw      t1, 0(x2)
sw      t0, 4(x2)
```

We are here

MIPS: Arithmetic Operations

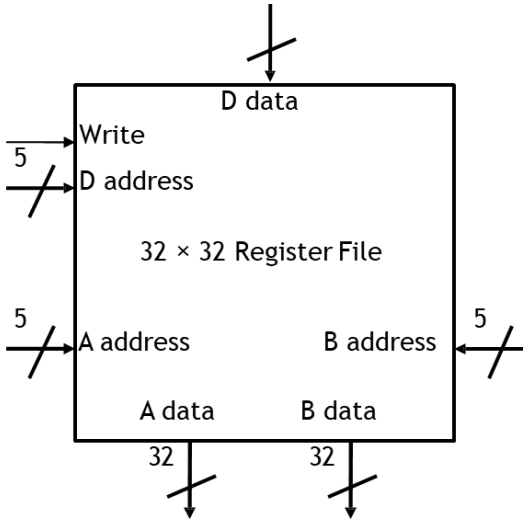
- High-level code:
  - a = b + c
- Compiled MIPS code (not actual code):



MIPS: Register Operands

- MIPS is a register-to-register, or load/store, architecture.
  - The destination and sources must all be registers (Arithmetic operations).
  - Special instructions, which we'll see soon, are needed to access main memory (memory related operations).

- MIPS processors have **32 registers**, each of which holds a **32-bit** value.
  - Register addresses are 5 bits long. Why?
  - The data inputs and outputs are 32-bits wide.



Register Correspondences

$zero	$0	Zero
$at	$1	Assembler temp
$v0-$v1	$2-3	Value (return from function)
$a0-$a3	$4-7	Argument (to function)
$t0-$t7	$8-15	Temporaries
$s0-$s7	$16-23	Saved Temporaries Saved
$t8-$t9	$24-25	Temporaries
$k0-$k1	$26-27	Kernel (OS) Registers
$gp	$28	Global Pointer Saved
$sp	$29	Stack Pointer Saved
$fp	$30	Frame Pointer Saved
$ra	$31	Return Address Saved

MIPS: Arithmetic Operations

- High-level code:
  - a = b + c
  - a, b and c in registers $t0, $t1 and $t2, respectively.
- Compiled MIPS code (not actual code):

  - add $t0, $t1, $t2

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

  add sub mul div

- And here are a few logical operations:

  and or xor

- Remember that these all require three register operands;
  - for example:

```
add   $t0, $t1, $t2          # $t0 = $t1 + $t2
mul   $s1, $s1, $a0          # $s1 = $s1 x $a0
```

Larger expressions

- More complex arithmetic expressions may require multiple operations at the instruction set level.

$$t0 = (t1 + t2) \times (t3 - t4)$$

Larger expressions

- More complex arithmetic expressions may require multiple operations at the instruction set level.

$$t0 = (t1 + t2) \times (t3 - t4)$$

```
add   $t0, $t1, $t2       # $t0 contains $t1 + $t2
sub   $s0, $t3, $t4       # Temporary value $s0 = $t3 - $t4
mul   $t0, $t0, $s0       # $t0 contains the final product
```

- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination.
  - In this example, we could re-use $t3 instead of introducing $s0.
  - But be careful not to modify registers that are needed again later.

Immediate operands

- The ALU instructions we've seen so far expect register operands. How do you get data into registers in the first place?

Immediate operands

- The ALU instructions we've seen so far expect register operands. How do you get data into registers in the first place?
  - Some MIPS instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register. For example, here is the immediate add instruction, addi:

```
addi  $t0, $t1, 4          # $t0 = $t1 + 4
```

  - Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi  $t0, $0, 4           # $t0 = 4
```

- MIPS is still considered a load/store architecture, because arithmetic operands cannot be from arbitrary memory locations. They must either be registers or constants that are embedded in the instruction.

A more complete assembly example

- How would you write code in MIPS assembly to compute:
  - 1 + 2 + 3 * 4

Registers vs. Memory

- Registers are **faster** to access than memory. But it is expensive and small in size.
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

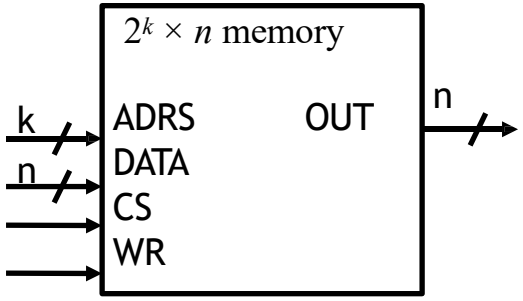
Memory Operands

- Main memory used for composite data.
  - Arrays, structures, dynamic data

- So, to compute with memory-based data, you must:
  - Load the data from memory to the register file.
  - Do the computation, leaving the result in a register.
  - Store that value back to memory if needed.

Memory review

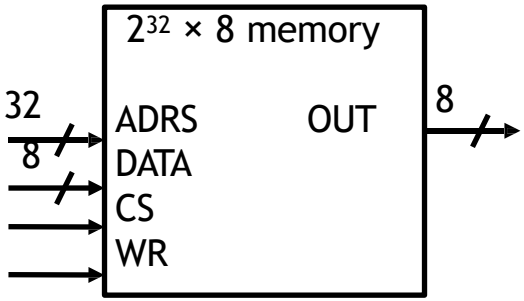
- Memory sizes are specified much like register files; here is a 2^k x n RAM.



CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

- A chip select input CS enables or "disables" the RAM.
- ADRS specifies the memory location to access.
- WR selects between reading from or writing to the memory.
  - To read from memory, WR should be set to 0. OUT will be the n-bit value stored at ADRS.
  - To write to memory, we set WR = 1. DATA is the n-bit value to store in memory.

MIPS memory



- MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity.
- The MIPS architecture can support up to 32 address lines.
  - This results in a 2^{32} x 8 RAM, which would be 4 GB of memory.
  - Not all actual MIPS machines will have this much!

Memory related instructions

- Load
  - Load data from memory to the registers
- Store
  - Store data back to memory

Loading and storing **bytes**

- The MIPS “load byte" instruction lb transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)          # $t0 = Memory[$a0 + 20]
```

- The “store byte" instruction sb transfers the lowest byte of data from a register into main memory.

```
sb $t0, 20($a0)          # Memory[$a0 + 20] = $t0
```

Example: Loading and storing **bytes**

- C code:
  - Char A = {‘1’,’2’,’3’,’4’}
  - A[1] = A[2] + A[3]
  - A’s address is in $a0

- Compiled MIPS code:
  - lb $t0, 2($a0) //$t0 = A[2]
  - lb $t1, 3($a0) //$t0 = A[3]
  - add $t0, $t0, $t1 //$t0 = A[2] + [3]
  - sb $t0, 1($a0) //A[1] = $t0

Loading and storing **words**

- You can also load or store 32-bit quantities—a complete word instead of just a byte—with the lw and sw instructions.

```
lw $t0, 20($a0)        # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)        # Memory[$a0 + 20] = $t0
```

- Most programming languages support several 32-bit data types.
  - Integers
  - Single-precision floating-point numbers
  - Memory addresses, or pointers
- Unless otherwise stated, we'll assume words are the basic unit of data.

An array of words

- Remember to be careful with memory addresses when accessing words.

- For instance, assume an array of words begins at address 2000.
  - The first array element is at address 2000.
  - The second word is at address ***2004***, not 2001.

- Example, if $a0 contains 2000, then

  lw $t0, 0($a0)

  accesses the first word of the array, but

  lw $t0, 8($a0)

  would access the *third* word of the array, at address 2008.

Exercise

- Registers x Memory

lw $t0, 4($a0)

What does $a0 contain?

What will $t0 contain after the instruction is executed?

Example: Loading and storing **words**

- C code:
  - int A = {1,2,3,4}
  - A[1] = A[2] + A[3]
  - A's address is in $a0

- Compiled MIPS code:
  - lw $t0, 8($a0) //$t0 = A[2]
  - lw $t1, 12($a0) //$t0 = A[3]
  - add $t0, $t0, $t1 //$t0 = A[2] + [3]
  - sw $t0, 4($a0) //A[1] = $t0

MIPS: Shift Operators

- MIPS contains shift operators in addition to the bit-manipulation operations. Shift instructions move the bits to the right or left within a register.

- Left shift
  - The shift left logical (sll) instruction has the following format:
    - sll $t0, $t1, 2 //$t0 = $t1 * 2^2
    - sll by i bits multiplies by 2^i
- Right shift
  - The shift right logical (srl) instruction has the following format:
    - srl $t0, $t1, 2 //$t0 = $t1 / 2^2
    - srl by i bits divides by 2^i
- Sometimes, we can use shift operations to replace the multiplications/divisions
  - It is more efficient

More Example: Loading and storing **words**

- C code:
  - int A
  - A[1] = A[2] + A[i]
  - A's address is in $a0, i is saved in $s0

- Compiled MIPS code:
  - lw $t0, 8($a0) //$t0 = A[2]
  - lw $t1, i?($a0) //$t0 = A[i]
  - add $t0, $t0, $t1 //$t0 = A[2] + [3]
  - sw $t0, 4($a0) //A[1] = $t0

```
# Assume:
# $a0 = base address of array A
# $s0 = index i

   sll  $t0, $s0, 2      # $t0 = i * 4  (calculate byte offset for A[i])
   add  $t1, $a0, $t0     # $t1 = A + i*4  (address of A[i])

   lw   $t2, 8($a0)      # $t2 = A[2] (A + 2*4)
   lw   $t3, 0($t1)      # $t3 = A[i] (load A[i] from calculated address)

   add  $t4, $t2, $t3     # $t4 = A[2] + A[i] (sum of A[2] and A[i])

   sw   $t4, 4($a0)      # Store the result into A[1] (A + 1*4)
```

Computing with memory

- So, to compute with memory-based data, you must:
  1. Load the data from memory to the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.

```
lb   $t0, 0($a0)      # Load A[0] into $t0
lb   $t1, 1($a0)      # Load A[1] into $t1
add  $t2, $t0, $t1      # $t2 = A[0] + A[1]

lb   $t3, 2($a0)      # Load A[2] into $t3
add  $t2, $t2, $t3      # $t2 = A[0] + A[1] + A[2]

lb   $t4, 3($a0)      # Load A[3] into $t4
add  $t2, $t2, $t4      # $t2 = A[0] + A[1] + A[2] + A[3]
```

- For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language? (A's address is in $a0, result's address is in $a1)

```
char A[4] = {1, 2, 3, 4};
int result;

result = A[0] + A[1] + A[2] + A[3];
```

```
lb   $t0, 0($a0)      # Load A[0] into $t0
lb   $t1, 1($a0)      # Load A[1] into $t1
lb   $t2, 2($a0)      # Load A[2] into $t2
lb   $t3, 3($a0)      # Load A[3] into $t3

add  $t4, $t0, $t1    # $t4 = A[0] + A[1]
add  $t4, $t4, $t2    # $t4 = A[0] + A[1] + A[2]
add  $t4, $t4, $t3    # $t4 = A[0] + A[1] + A[2] + A[3]

move $v0, $t4         # Store the sum in result (return in $v0)
```

Computing with memory

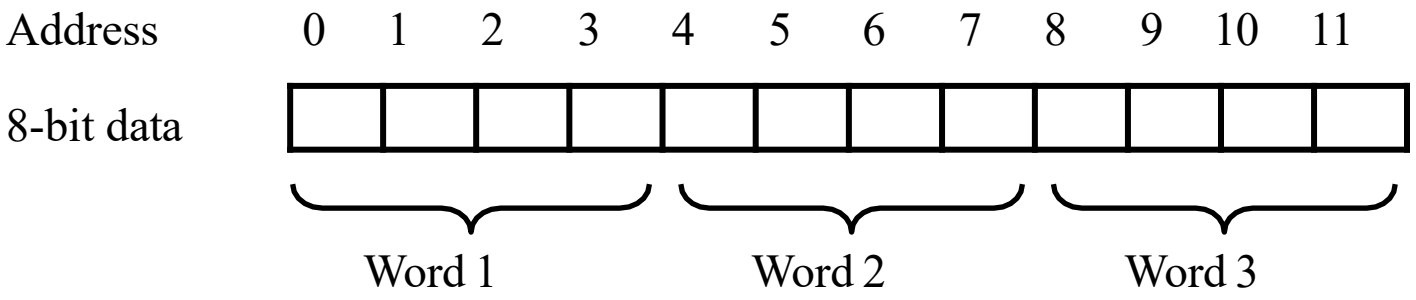
- So, to compute with memory-based data, you must:
  1. Load the data from memory to the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.

- For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language? (A's address is in $a0, result's address is in $a1)

```
int A[4] = {1, 2, 3, 4};
int result;

result = A[0] + A[1] + A[2] + A[3];
```

MIPS: Memory alignment

- Keep in mind that memory is **byte-addressable**, so a 32-bit word actually occupies **four contiguous locations** (bytes) of main memory.



- The MIPS architecture requires words to be aligned in memory; **32-bit words must start at an address that is divisible by 4.**
  - 0, 4, 8 and 12 are valid word addresses.
  - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
  - Unaligned memory accesses result in a bus error, which you may have unfortunately seen before.

- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

Pseudo-instructions

- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, "real" instructions.
- For example, you can use the li and move pseudo-instructions:

```
li $a0, 2000          # Load immediate 2000 into $a0
move  $a1, $t0        # Copy $t0 into $a1
```

- They are probably clearer than their corresponding MIPS instructions:

```
addi $a0, $0, 2000    # Initialize $a0 to 2000
add $a1, $t0, $0      # Copy $t0 into $a1
```

- We'll see lots more pseudo-instructions this semester.
  - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

Branches and Loops

- The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- Conditional statements execute only if some test expression is true.

```
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0;              // This might not be executed
v1 = v0 + v0;
```

- Loops cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];   // These statements will
    t0++;               // be executed five times
}
```

MIPS control instructions: Branches

- MIPS's control-flow instructions

j	// for unconditional jumps
bne and beq	// for conditional branches
slt and slti	// set if less than (w/o and w an immediate)

- Now we'll talk about
  - MIPS's pseudo branches
  - if/else
  - case/switch

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

```
blt   $t0, $t1, L1          // Branch if $t0 < $t1
ble   $t0, $t1, L2          // Branch if $t0 <= $t1

bgt   $t0, $t1, L3          // Branch if $t0 > $t1
bge   $t0, $t1, L4          // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

- Later we'll see how supporting just beq and bne simplifies the processor design.

Implementing pseudo-branches

- Most pseudo-branches are implemented using slt. For example, a branch- if-less-than instruction **blt $a0, $a1, Label** is translated into the following.

```
slt   $at, $a0, $a1      // $at = 1 if $a0 < $a1
bne   $at, $0, Label     // Branch if $at != 0
```

- This supports immediate branches, which are also pseudo-instructions. For example, **blti $a0, 5, Label** is translated into two instructions.

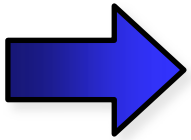
```
slti $at, $a0, 5          // $at = 1 if  $a0 < 5
bne $at, $0, Label        // Branch if $a0 < 5
```

- All of the pseudo-branches need a register to save the result of slt, even though it's not needed afterwards.
  - MIPS assemblers use register $1, or $at, for temporary storage.
  - You should be careful in using $at in your own programs, as it may be overwritten by assembler-generated code.

Translating an if-then statement

- We can use branch instructions to translate if-then statements into MIPS assembly code.

```
v0 = a0;
if (v0 < 0)
        v0 = -v0;
v1 = v0 + v0;
```



```
move $v0 $a0
bge $v0, $0          Label
sub $v0, 0, $v0
Label: add $v1, $v0, $v0
```

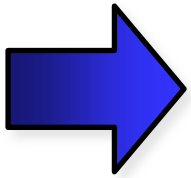
- Sometimes it's easier to *invert* the original condition.
  - In this case, we changed "continue if v0 < 0" to "skip if v0 >= 0".

Translating an if-then-else statements

- If there is an else clause, it is the target of the conditional branch
  — And the then clause needs a jump over the else clause

// increase the magnitude of v0 by one

```
    if (v0 < 0)
        v0 --;

    else
        v0 ++;
    v1 = v0;
```



```
        bge  $v0, $0, E
        sub  $v0, $v0, 1
        j    L

E:      add  $v0, $v0, 1
L:      move $v1, $v0
```

Exercise

Convert the below C code snippet to MIPS assembly code. Assume variables a, b are stored in registers t0, t1 respectively and are 32-bits non-zero positive integer. Base address of c is stored in register t8. Do not use multiply and divide instruction, use sll and srl instead.

```
if(a > b)
        c[a] = c[b];
else
        c[b] = c[a];
```

```
# Assume:
# $t0 = a (input: a)
# $t1 = b (input: b)
# $t8 = base address of array c

# Compare a and b
bgt  $t0, $t1, greater_a    # If a > b, jump to 'greater_a'

# Else: a <= b
# c[b] = c[a]
sll  $t2, $t1, 2        # $t2 = b * 4 (calculate byte offset for c[b])
add  $t3, $t8, $t2      # $t3 = address of c[b]
lw   $t4, 0($t3)        # Load c[b] into $t4
sll  $t2, $t0, 2        # $t2 = a * 4 (calculate byte offset for c[a])
add  $t3, $t8, $t2      # $t3 = address of c[a]
sw   $t4, 0($t3)        # Store c[b] into c[a]
j    end_if             # Jump to the end of the if-else statement

greater_a:
   # c[a] = c[b]
   sll  $t2, $t0, 2        # $t2 = a * 4 (calculate byte offset for c[a])
   add  $t3, $t8, $t2      # $t3 = address of c[a]
   lw   $t4, 0($t3)        # Load c[a] into $t4
   sll  $t2, $t1, 2        # $t2 = b * 4 (calculate byte offset for c[b])
   add  $t3, $t8, $t2      # $t3 = address of c[b]
   sw   $t4, 0($t3)        # Store c[a] into c[b]

end_if:
   # End of if-else statement
```

Case/Switch Statement

- Many high-level languages support multi-way branches, e.g.

```
switch (two_bits) {
    case 0:    break;
    case 1:    /* fall through */
    case 2:    count ++;     break;
    case 3:    count += 2;   break;
}
```

- We could just translate the code to if, thens, and elses:

```
if ((two_bits == 1) || (two_bits == 2)) {
    count ++;
} else if (two_bits == 3) {
    count += 2;
}
```

- This isn't very efficient if there are many, many cases.

Case/Switch Statement

```
switch (two_bits) {
    case 0:    break;
    case 1:    /* fall through */
    case 2:    count ++;     break;
    case 3:    count += 2;   break;
}
```

- Alternatively, we can:
  1. Create an array of jump targets
  2. Load the entry indexed by the variable two_bits
  3. Jump to that address using the jump register, or jr, instruction

Loops

```
Loop:    j    Loop              # goto Loop
```

```
        for (i = 0; i < 4; i++) {
             // stuff
        }
```

```
        add    $t0, $zero, $zero    # i is initialized to 0, $t0 = 0
Loop:   // stuff
        addi   $t0, $t0, 1          # i ++
        slti   $t1, $t0, 4          # $t1 = 1 if i < 4
        bne    $t1, $zero, Loop     # go to Loop if i < 4
```

Exercise

Convert the below C code snippet to MIPS assembly code. Assume variables a, b are stored in registers t0, t1, respectively and are 32-bits non-zero positive integer. Base address of c is stored in register t8. Do not use multiply and divide instruction, use sll and srl instead.

```
for(i=0; i<a; i++)
        c[2*i] = c[b/4] – i;
```

```
    addi $t2, $zero, 0      # i = 0

Loop:
    bge $t2, $t0, end_loop    # if (i >= a) break

    sll $t3, $t2, 1          # t3 = 2 * i
    sll $t4, $t3, 2          # t4 = 8 * i (c[2*i] offset)
    add $t6, $t8, $t4        # address of c[2*i] = base + 8*i

    srl $t5, $t1, 2          # t5 = b / 4
    sll $t5, $t5, 2          # t5 = b (index for c) * 4
    add $t7, $t8, $t5        # address of c[b/4] = base + b

    lw $t9, 0($t7)           # load c[b/4] into t9
    sub $t9, $t9, $t2        # t9 = c[b/4] - i
    sw $t9, 0($t6)           # store the result in c[2*i]

    addi $t2, $t2, 1         # i++
    j Loop                   # jump back to Loop

end_loop:
    # Continue with the rest of the program
```

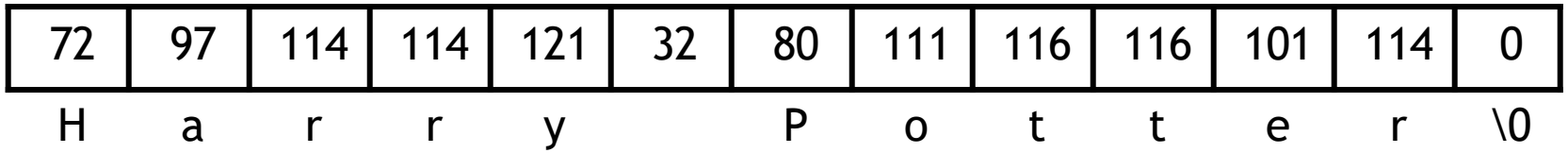
Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.



- Since strings can vary in length, we put a 0, or null, at the end of the string.
  - This is called a null-terminated string

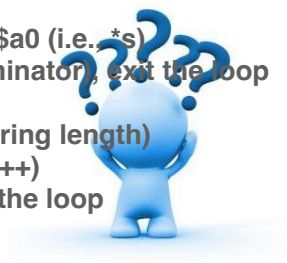
- Computing string length
  - We’ll look at two ways.

What does this C code do?

```
int foo(char *s) {
        int L = 0;
        while (*s++) {
                ++L;
        }
        return L;
}
```

Explanation:
char *s is a pointer to a string (array of characters).
***S++**
means dereference the character pointed to by s, then increment s to point to the next character.
++L increments the counter L each time a non-zero character is found.
• The loop stops when *s
is zero (null-terminator \0 in the string).
MIPS Assembly
We'll assume:
• The pointer s is passed in register $a0 (as per the MIPS calling convention).
• The return value will be in register $vo (also standard for MIPS).
• Local variable L will be stored in $to (used to count the length of the string).

```
foo:
   # Prologue: save the return address (if needed)
   # No need to save registers if no nested function calls.
   # But you may push $ra if you want to be cautious.

   addi $t0, $zero, 0      # Initialize L = 0 (counter for string length)

Loop:
   lb   $t1, 0($a0)        # Load byte from address in $a0 (i.e., *s)
   beq  $t1, $zero, End    # If the byte is 0 (null-terminator), exit the loop

   addi $t0, $t0, 1        # Increment L (counter for string length)
   addi $a0, $a0, 1        # Increment the pointer s (s++)
   j Loop                  # Jump to the next iteration of the loop

End:
   move $v0, $t0           # Return L in $v0
   jr   $ra                # Return from function
```



Array Indexing Implementation of strlen

```
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

```
strlen:
   addi $t0, $zero, 0      # Initialize len = 0

Loop:
   lb   $t1, 0($a0)        # Load the byte string[len] into $t1
   beq  $t1, $zero, End    # If string[len] == 0 (null terminator), exit the loop

   addi $t0, $t0, 1        # Increment len by 1
   addi $a0, $a0, 1        # Move to the next character in the string (string++)
   j Loop                  # Jump back to the start of the loop

End:
   move $v0, $t0           # Return len in $v0
   jr   $ra                # Return to the caller
```

Pointers & Pointer Arithmetic

- Many programmers have a vague understanding of pointers
  - Looking at assembly code is useful for their comprehension.

```
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len ++;
    }
    return len;
}
```

```
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string ++;
        len ++;
    }
    return len;
}
```

What is a Pointer?

- A pointer is an address.
- Two pointers that point to the same thing hold the same address
- Dereferencing a pointer means loading from the pointer's address
- A pointer has a type; the type tells us what kind of load to do
  - Use load byte (lb) for char *
  - Use load half (lh) for short *
  - Use load word (lw) for int *
  - Use load single precision floating point (l.s) for float *

- Pointer arithmetic is often used with pointers to arrays
  - Incrementing a pointer (i.e., ++) makes it point to the next element
  - The amount added to the pointer depends on the type of pointer
    - pointer = pointer + sizeof(*pointer's type)*
      - 1 for char *, 4 for int *, 4 for float *, 8 for double *

What is really going on here…

```
int strlen(char *string) {
    int len = 0;

    while (*string != 0) {
        string ++;
        len ++;
    }

    return len;
}
```

```
strlen:
   addi $t0, $zero, 0      # Initialize len = 0 (register $t0 holds the length of the string)

Loop:
   lb   $t1, 0($a0)        # Load the byte from string (pointed to by $a0) into $t1
   beq  $t1, $zero, End     # If *string == 0 (null terminator), exit the loop

   addi $t0, $t0, 1         # Increment len by 1
   addi $a0, $a0, 1         # Increment the string pointer (string++)
   j Loop                   # Jump back to the start of the loop

End:
   move $v0, $t0            # Return len in $v0 (the return register)
   jr   $ra                 # Return to the caller
```

Pointers Summary

- Pointers are just addresses!!
  - "Pointees" are locations in memory

- Pointer arithmetic updates the address held by the pointer
  - "string ++" points to the next element in an array
  - Pointers are typed so address is incremented by sizeof(pointee)

An Example Function: Factorial

```
int fact(int n) {

   int i, f = 1;
   for (i = n; i > 0; i--)
       f = f * i;
   return f;

}
```

```
fact:
            li $t0, 1
            move $t1,$a0            # set i to n

loop:
            ble  $t1, $0, exit      # exit if done
            mul $t0,$t0,$t1         # build factorial
            addi $t1, $t1,-1        # i--
            j loop


exit:
            move $v0,$t0
```

Functions in MIPS

- We'll talk about the 3 steps in handling function calls:
  1. The program's flow of control must be changed.
  2. Arguments and return values are passed back and forth.
  3. Local variables can be allocated and destroyed.
- And how they are handled in MIPS:
  - New instructions for calling functions.
  - Conventions for sharing registers between functions.
  - Use of a stack.

Control flow in C

- Invoking a function changes the control flow of a program twice.
  1. Calling the function
  2. Returning from the function
- In this example the main function calls fact twice, and fact returns twice—but to *different* locations in main.
- Each time fact is called, the CPU has to remember the appropriate return address.
- Notice that main itself is also a function! It is called by the operating system when you run the program.

```
int main()
{
   ...
   t1 = fact(8);
   t2 = fact(3);
   t3 = t1 + t2;
   ...
}


int fact(int n)
{
   int i, f = 1;
   for (i = n; i > 1; i--)
      f = f * i;
   return f;
}
```

Control flow in MIPS

- MIPS uses the jump-and-link instruction jal to call functions.
  - The jal saves the return address (the address of the *next* instruction) in the dedicated register $ra, before jumping to the function.
  - jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in $ra.

jal Fact

- To transfer control back to the caller, the function just has to jump to the address that was stored in $ra.

jr $ra

- Let's now add the jal and jr instructions that are necessary for our factorial example.

Data flow in C

- Functions accept arguments and produce return values.
- The blue parts of the program show the actual and formal arguments of the fact function.
- The purple parts of the code deal with returning and using a result.

```
int main()
{
   ...
   t1 = fact(8);
   t2 = fact(3);
   t3 = t1 + t2;
   ...
}

int fact(int n)
{
   int i, f = 1;
   for (i = n; i > 1; i--)
      f = f * i;
   return f;
}
```

Data flow in MIPS

- MIPS uses the following conventions for function arguments and results.
  - Up to four function arguments can be "passed" by placing them in argument registers $a0-$a3 before calling the function with jal.
  - A function can "return" up to two values by placing them in registers $v0-$v1, before returning via jr.
- These conventions are not enforced by the hardware or assembler, but programmers agree to them so functions written by different people can interface with each other.
- Later we'll talk about handling additional arguments or return values.

The big problem so far

- There is a big problem here!
  - The main code uses $t1 to store the result of fact(8).
  - But $t1 is also used within the fact function!
- The subsequent call to fact(3) will overwrite the value of fact(8) that was stored in $t1.

```
# main function
    .text
    .globl main

main:
    # Call fact(8)
    li   $a0, 8         # Load 8 into $a0 (argument for fact(8))
    jal  fact           # Jump and link to fact function
    move $t1, $v0       # Store the result of fact(8) in t1

    # Call fact(3)
    li   $a0, 3         # Load 3 into $a0 (argument for fact(3))
    jal  fact           # Jump and link to fact function
    move $t2, $v0       # Store the result of fact(3) in t2

    # Add t1 and t2, store result in t3
    add  $t3, $t1, $t2   # t3 = t1 + t2

    # Return from main (exit program)
    li   $v0, 10        # Load 10 into $v0 (exit syscall)
    syscall             # Make syscall to exit

# fact function
fact:
    # Function prologue
    addi $sp, $sp, -8    # Allocate space on stack for local variables
    sw   $ra, 4($sp)     # Save return address
    sw   $s0, 0($sp)     # Save $s0 (for preservation)

    # Initialize f = 1 and i = n
    li   $t0, 1          # f = 1 (store in $t0)
    move $t1, $a0        # i = n (store in $t1)

fact_loop:
    blez $t1, fact_end   # if i <= 1, exit loop
    mul  $t0, $t0, $t1   # f = f * i
    sub  $t1, $t1, 1     # i = i - 1
    j    fact_loop       # Jump to fact_loop

fact_end:
    move $v0, $t0        # Store result in $v0 (return value)

    # Function epilogue
    lw   $ra, 4($sp)     # Restore return address
    lw   $s0, 0($sp)     # Restore $s0
    addi $sp, $sp, 8     # Deallocate space on stack
    jr   $ra             # Return to the caller
```

Nested functions

- A similar situation happens when you call a function that then calls another function.
- Let's say A calls B, which calls C.
  - The arguments for the call to C would be placed in $a0-$a3, thus *overwriting* the original arguments for B.
  - Similarly, jal C overwrites the return address that was saved in $ra by the earlier jal B.

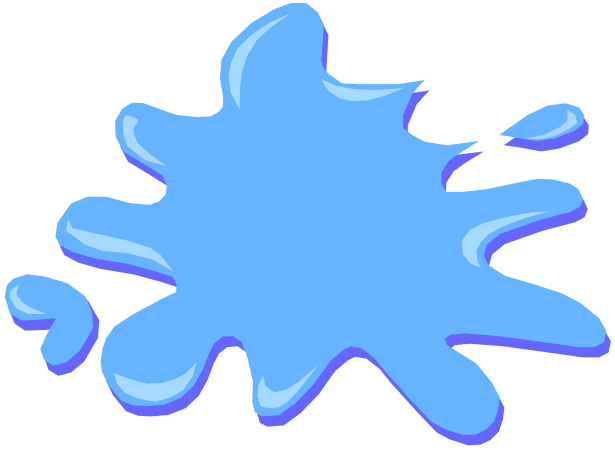
```
A:   ...
     # Put B's args in $a0-$a3
     jal  B       # $ra = A2
A2:  ...
```

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

```
C:   ...
     jr   $ra
```

Spilling registers

- The CPU has a limited number of registers for use by all functions, and it's possible that several functions will need the same registers.
- We can keep important registers from being overwritten by a function call, by saving them before the function executes, and restoring them after the function completes.
- But there are two important questions.
  - Who is responsible for saving registers—the caller or the callee?
  - Where exactly are the register contents saved?



Who saves the registers?

- Who is responsible for saving important registers across function calls?
  - The caller knows which registers are important to it and should be saved.
  - The callee knows exactly which registers it will use and potentially overwrite.
- However, in the typical "black box" programming approach, the caller and callee do not know anything about each other's implementation.
  - Different functions may be written by different people or companies.
  - A function should be able to interface with any client, and different implementations of the same function should be substitutable.
- So how can two functions cooperate and share registers when they don't know anything about each other?

The caller could save the registers…

- One possibility is for the *caller* to save any important registers that it needs before making a function call, and to restore them after.
- But the caller does not know what registers are actually written by the function, so it may save more registers than necessary.
- In the example on the right, frodo wants to preserve $a0, $a1, $s0 and $s1 from gollum, but gollum may not even use those registers.

```
frodo: li   $a0, 3
       li   $a1, 1
       li   $s0, 4
       li   $s1, 1

       # Save registers
       # $a0, $a1, $s0, $s1

       jal  gollum

       # Restore registers
       # $a0, $a1, $s0, $s1

       add  $v0, $a0, $a1
       add  $v1, $s0, $s1
       jr   $ra
```

…or the callee could save the registers…

- Another possibility is if the *callee* saves and restores any registers it might overwrite.
- For instance, a gollum function that uses registers $a0, $a2, $s0 and $s2 could save the original values first, and restore them before returning.
- But the callee does not know what registers are important to the caller, so again it may save more registers than necessary.

```
gollum:
        # Save registers
        # $a0 $a2 $s0 $s2

        li   $a0, 2
        li   $a2, 7
        li   $s0, 1
        li   $s2, 8
        ...

        # Restore registers
        # $a0 $a2 $s0 $s2

        jr   $ra
```

…or they could work together

- MIPS uses conventions again to split the register spilling chores.
- The *caller* is responsible for saving and restoring any of the following caller-saved registers that it cares about.

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

  In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.
- The *callee* is responsible for saving and restoring any of the following callee-saved registers that it uses. (Remember that $ra is "used" by jal.)

  $s0-$s7 $ra

  Thus the caller may assume these registers are not changed by the callee.
  - $ra is tricky; it is saved by a callee who is also a caller.
- Be especially careful when writing nested functions, which act as both a caller and a callee!

Register spilling example

- This convention ensures that the caller and callee together save all of the important registers—frodo only needs to save registers $a0 and $a1, while gollum only has to save registers $s0 and $s2.

```
frodo:  li    $a0, 3
        li    $a1, 1
        li    $s0, 4
        li    $s1, 1

        # Save registers
        # $a0 and $a1

        jal   gollum

        # Restore registers
        # $a0 and $a1

        add   $v0, $a0, $a1
        add   $v1, $s0, $s1
        jr    $ra
```

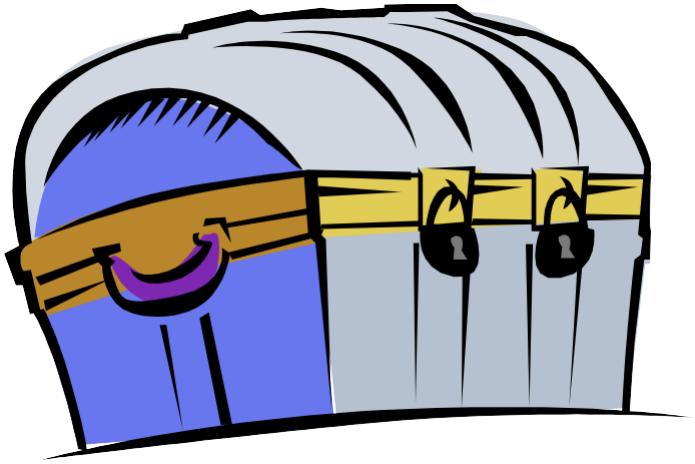
```
gollum:
        # Save registers
        # $s0 and $s2

        li    $a0, 2
        li    $a2, 7
        li    $s0, 1
        li    $s2, 8
        ...

        # Restore registers
        # $s0 and $s2

        jr    $ra
```

How to fix factorial

- In the factorial example, main (the caller) should save two registers.
  - $t1 must be saved before the second call to fact.
  - $ra will be implicitly overwritten by the jal instructions.
- But fact (the callee) does not need to save anything. It only writes to registers $t0, $t1 and $v0, which should have been saved by the caller.

Where are the registers saved?

- Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.
- It would be nice if each function call had its own private memory area.
  - This would prevent other function calls from overwriting our saved registers—otherwise using memory is no better than using registers.
  - We could use this private memory for other purposes too, like storing local variables.

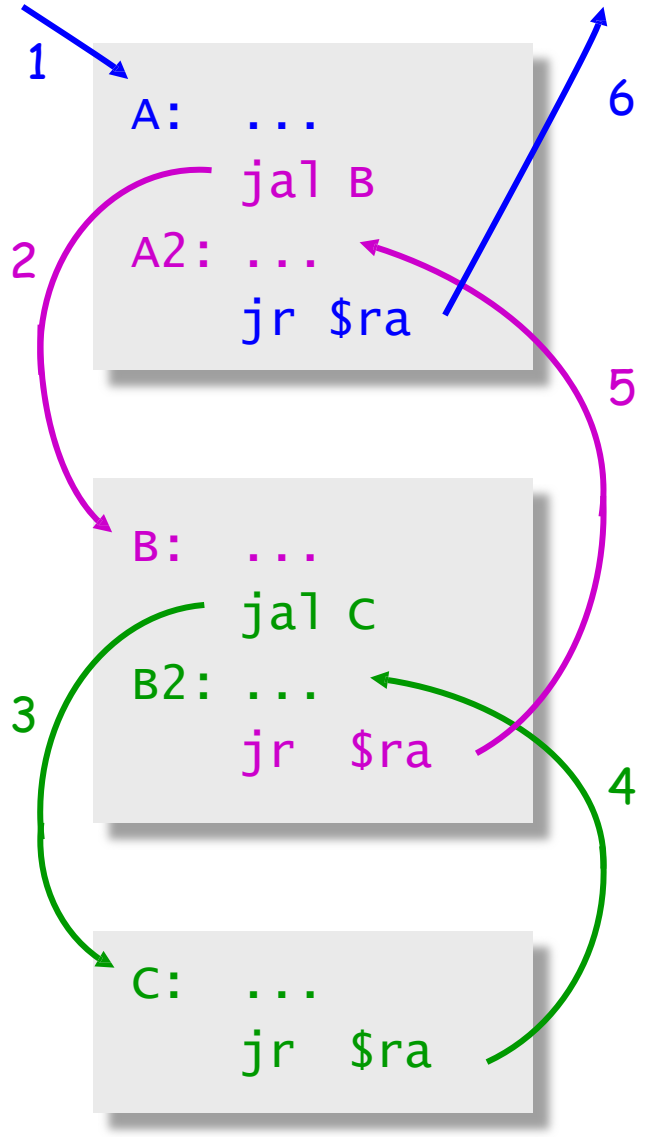


Function calls and stacks

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns

- Here, for example, C must return to B *before* B can return to A.

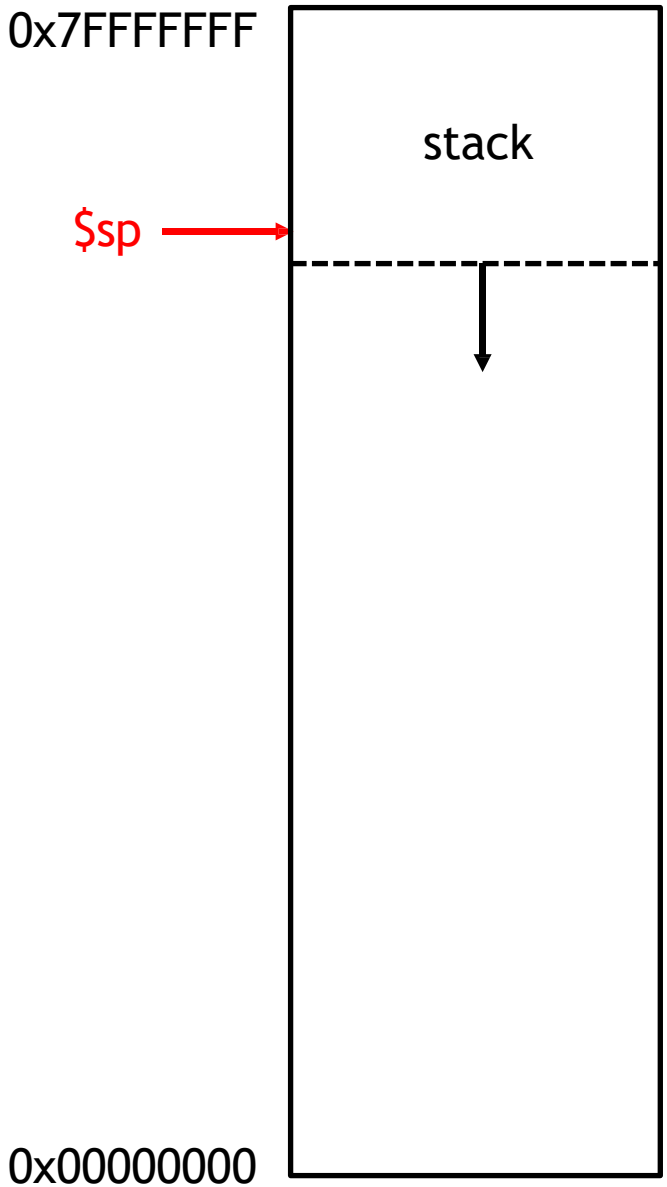


Stacks and function calls

- It's natural to use a stack for function call storage. A block of stack space, called a stack frame, can be allocated for each function call.
  - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
  - Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- The stack frame can be used for several purposes.
  - Caller- and callee-save registers can be put in the stack.
  - The stack frame can also hold local variables, or extra arguments and return values.

The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
  - The stack grows downward in terms of memory addresses.
  - The address of the top element of the stack is stored (by convention) in the "stack pointer" register, $sp.
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.



Pushing elements

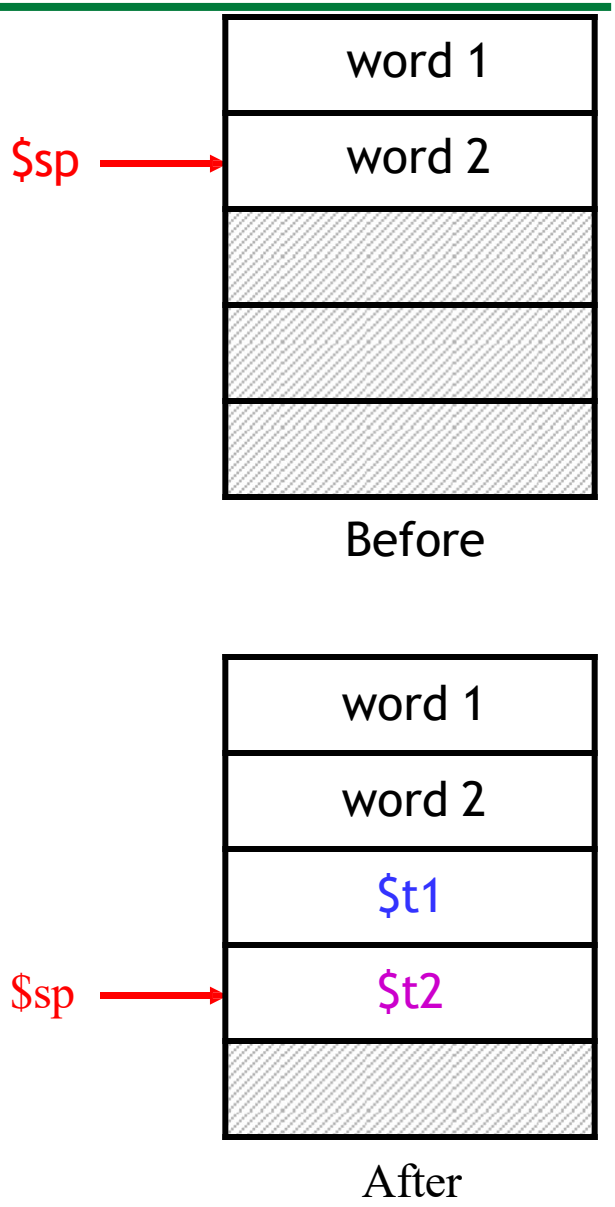
- To push elements onto the stack:
  - Move the stack pointer $sp down to make room for the new data.
  - Store the elements into the stack.
- For example, to push registers $t1 and $t2 onto the stack:

```
sub  $sp, $sp, 8
sw   $t1, 4($sp)
sw   $t2, 0($sp)
```

- An equivalent sequence is:

```
sw   $t1, -4($sp)
sw   $t2, -8($sp)
sub  $sp, $sp, 8
```

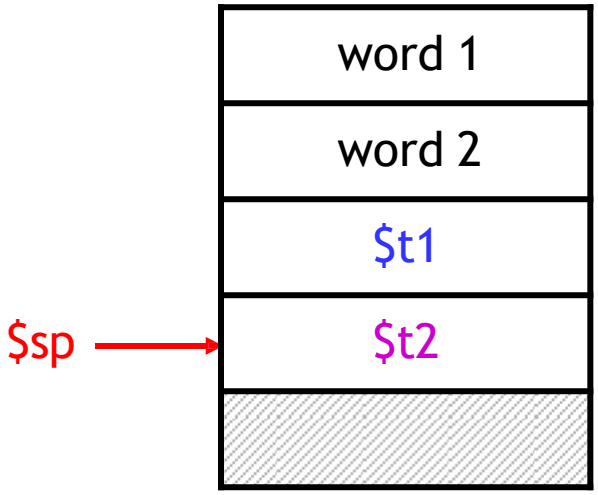
- Before and after diagrams of the stack are shown on the right.



Accessing and popping elements

- You can access any element in the stack (not just the top one) if you know where it is relative to $sp.
- For example, to retrieve the value of $t1:

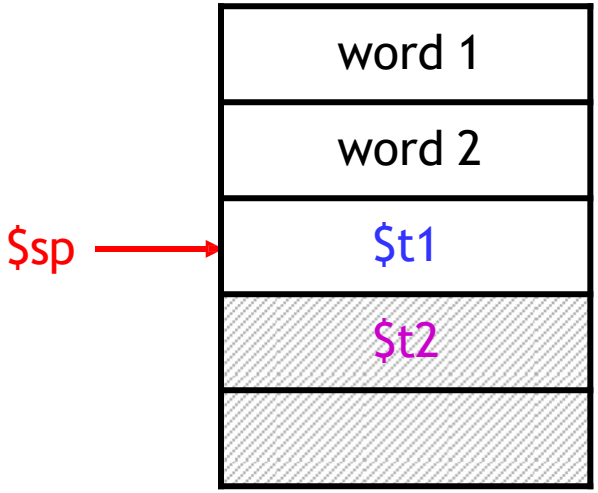
```
lw      $s0, 4($sp)
```



- You can pop, or "erase," elements simply by adjusting the stack pointer upwards.
- To pop the value of $t2, yielding the stack shown at the bottom:

```
addi $sp, $sp, 4
```

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



Exercise

```
int pow(int n, int m) {
        int res = 1;
        for(int i = 0; i < m; i++){
                res = res * n;
        }
        return res;
}
```

```
li $a0, 2; //n = 2
li $a1, 3; //m = 3
jal pow
```

pow:

```
main:
   # Set up for the pow function call
   li   $a0, 2       # Load n = 2 into $a0
   li   $a1, 3       # Load m = 3 into $a1
   jal  pow          # Call pow function

# After returning from pow, the result is in $v0
   move $t0, $v0     # Move the result into $t0 (just for example)

   # Exit program (using syscall)
   li   $v0, 10      # Load 10 into $v0 (exit syscall)
   syscall           # Exit the program

# pow function
pow:
   # Function prologue: Save return address and registers
   addi $sp, $sp, -8   # Make space on stack for saving registers
   sw   $ra, 4($sp)    # Save return address
   sw   $a2, 0($sp)    # Save $a2 (will be used for loop counter)

   # Initialize res = 1 (store in $t0)
   li   $t0, 1         # res = 1

   # Initialize i = 0 (store in $t2)
   li   $t2, 0         # i = 0

pow_loop:
   bge  $t2, $a1, pow_end  # if i >= m, exit loop
   mul  $t0, $t0, $a0      # res = res * n (res = res * n)
   addi $t2, $t2, 1        # i = i + 1
   j    pow_loop           # Jump to the start of the loop

pow_end:
   move $v0, $t0     # Store result in $v0 (return value)

   # Function epilogue: Restore saved registers
   lw   $ra, 4($sp)    # Restore return address
   lw   $a2, 0($sp)    # Restore $a2 (loop counter)
   addi $sp, $sp, 8    # Deallocate space on stack

   jr   $ra          # Return to the caller
```

Exercise

```
int pow(int n, int m) {
        if (m == 1)
                return n;
        return n * pow(n, m-1);
}
```

```
li $a0, 2; //n = 2
li $a1, 3; //m = 3
jal pow
```

pow:

```
.text
.globl main

# main function
main:
    # Setup for the pow function call
    li   $a0, 2       # Load n = 2 into $a0
    li   $a1, 3       # Load m = 3 into $a1
    jal  pow          # Call pow function

    # After returning from pow, the result is in $v0
    move $t0, $v0     # Move the result into $t0 (just for example)

    # Exit program (using syscall)
    li   $v0, 10      # Load 10 into $v0 (exit syscall)
    syscall           # Exit the program

# pow function
pow:
    # Base case: if m == 1, return n
    beq  $a1, 1, pow_base_case  # If m == 1, jump to the base case

    # Recursive case: return n * pow(n, m-1)
    addi $a1, $a1, -1       # Decrement m by 1 (m = m - 1)
    jal  pow                # Call pow(n, m-1)

    # Multiply n * result of pow(n, m-1)
    mul  $v0, $a0, $v0      # $v0 = n * pow(n, m-1)

    jr   $ra                # Return to the caller

# Base case: if m == 1, return n
pow_base_case:
    move $v0, $a0           # Return n (since m == 1)
    jr   $ra                # Return to the caller
```

Summary

- Today we focused on implementing function calls in MIPS.
  - We call functions using jal, passing arguments in registers $a0-$a3.
  - Functions place results in $v0-$v1 and return using jr $ra.
- Managing resources is an important part of function calls.
  - To keep important data from being overwritten, registers are saved according to conventions for caller-save and callee-save registers.
  - Each function call uses stack memory for saving registers, storing local variables and passing extra arguments and return values.
- Assembly programmers must follow many conventions. Nothing prevents a rogue program from overwriting registers or stack memory used by some other function.

Next Class

Machine Language