Quick exercise

- How to allow a function to return more information that cannot be completely stored in $v0 and $v1?
  - A. Use other registers such as $a0, $s0, etc. to carry more information.
  - B. Store extra information in the stack when returning back to the function caller.
  - C. Store the return results in the heap and pass the memory address back to the function caller.
  - D. None of the above

Global pointer

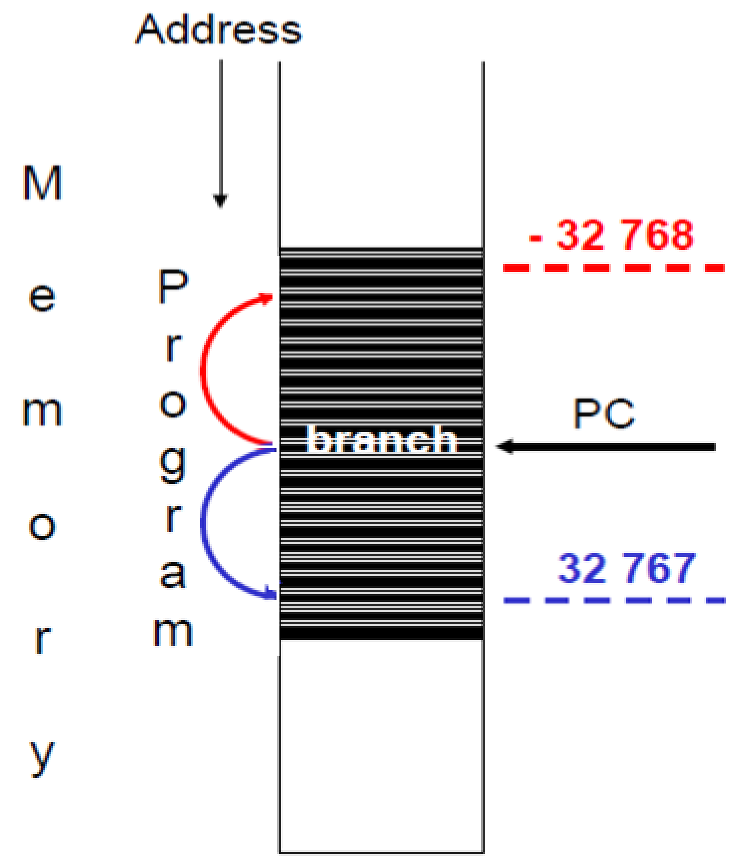
- Many languages like c support two storage classes
  - Automatic
    - Local to a procedure
    - Discarded when the procedure exits
  - Static
    - Exist across exits from and entries to procedures
    - Declared outside a procedure
    - Declared inside a procedure using the **static** keyword

- Global pointer $gp is used to access static variables
  - Static data in memory starts at $1000\ 0000_{hex}$.
  - To ease access to data, $gp is initialized to $1000\ 8000_{hex}$.
  - Using 16-bit offset from $gp, memory addresses from $1000\ 0000_{hex}$ to $1000\ FFFF_{hex}$ can be accessed.

Relative addressing

`beq register1, register2, L1`

- The immediate field contains 16 bits
- Branch offsets can be either positive or negative
- So, longest branch is either:
- 32 768 lines **up** or
- 32 767 lines **down**



Addressing in jumps

- J-type instruction format

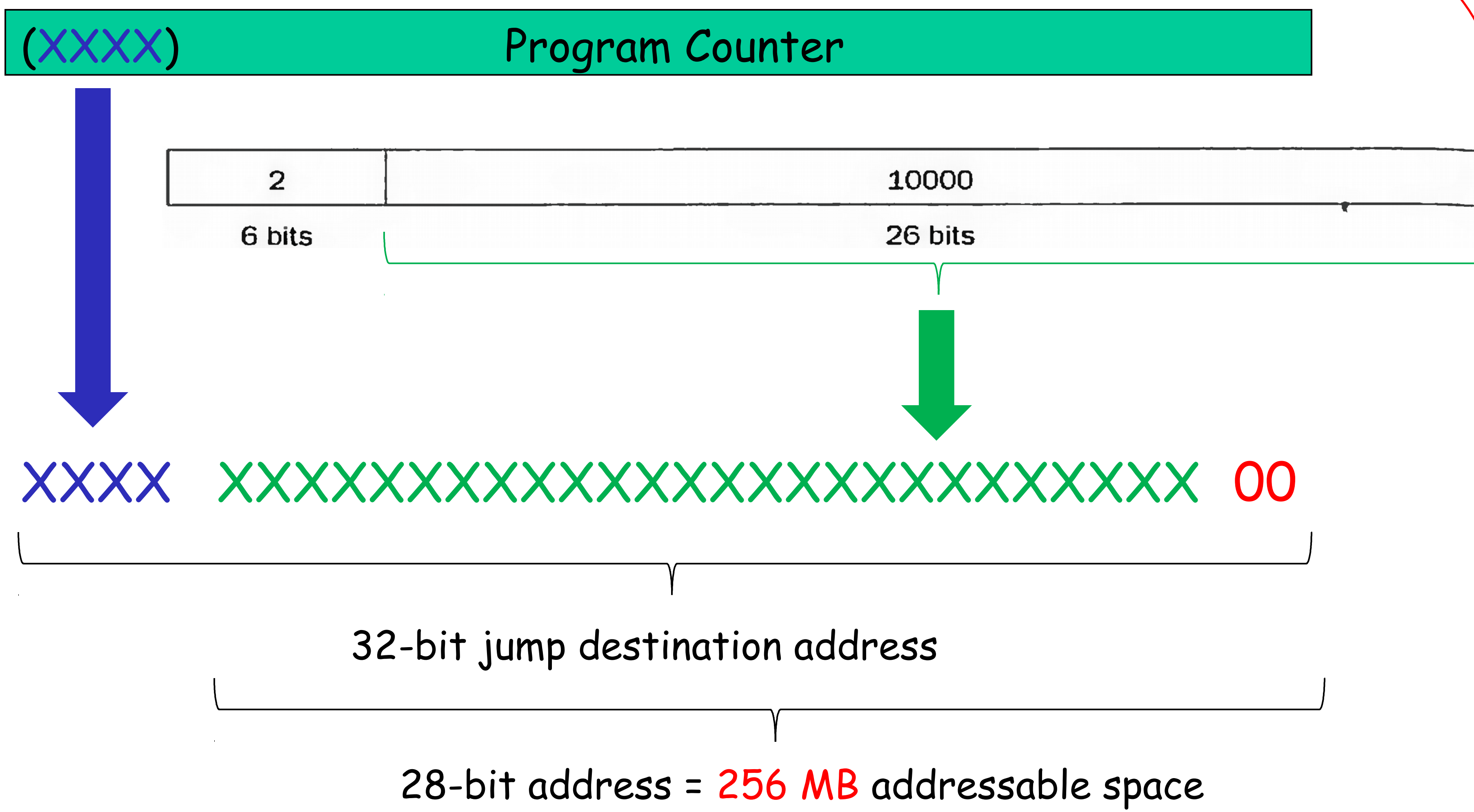
```
j   L1        # go to location 10000
```

2	10000
6 bits	26 bits

opcode=2 for j

  - Address in the J-type instruction is measured in words (32-bit unit)
  - 26 bits stands for a 28 bits address in memory (last two bits are 0s)

  - The 4 leftmost bits of a 32-bit address is directly inherited from the program counter



Pseudo-direct addressing

Calculating a branch offset

- Let n be the number of MIPS program instructions between a branch instruction and the branch target at label L
  - The conditional branch instruction is counted.
  - n is negative if we are jumping backward.
  - n is positive if we are jumping forward.
- The branch offset is subsequently determined as

$$\text{Branch offset} = n - 1$$

Example

- Consider the MIPS assembly code:

```
Loop:sll    $t1,$s3,2     # Temp reg $t1 = 4 * i
     add $t1,$t1,$s6      # $t1 = address of save[i]
     lw  $t0,0($t1)       # Temp reg $t0 = save[i]
     bne $t0,$s5, Exit    # go to Exit if save[i] ≠ k
     addi $s3,$s3,1       # i = i + 1
     j   Loop             # go to Loop
Exit:
```

- What is the offset represented by Exit in the bne instruction?
  - A. -2
  - B. 2
  - C. -3
  - D. 3

- What is the corresponding value for the "loop" in "j loop"?
  - j uses pseudo-addressing
  - The loop starts at location 80,000 in memory

```
Loop:sll    $t1,$s3,2     # Temp reg $t1 = 4 * i
     add $t1,$t1,$s6      # $t1 = address of save[i]
     lw  $t0,0($t1)       # Temp reg $t0 = save[i]
     bne $t0,$s5, Exit    # go to Exit if save[i] ≠ k
     addi $s3,$s3,1       # i = i + 1
     j    Loop            # go to Loop
Exit:
```

  - A. -80,006
  - B. 80,006
  - C. -80,000
  - D. 80,000

Branch far away

- Invert the condition so that the branch decides whether to skip a long-distance jump.
- Insert an unconditional jump to the branch target

- Example:
  - Given the branch below:

```
beq    $s0, $s1, L1
```

  - Replace it by a pair of instructions that offers a much greater branching distance.

- Answer:

```
     bne    $s0, $s1, L2
     j      L1
L2:
```

MIPS addressing modes

- Immediate addressing
  - The operand is a constant within the instruction itself
- Register addressing
  - The operand is a register (e.g. $ra)
- Base or displacement addressing
  - The operand is at the memory location whose address is the sum of a register and a constant in the instruction (e.g. 4($s0))
- Relative addressing
  - The branch address is the sum of the PC and a constant in the instruction
- Pseudo-direct addressing
  - Jump address is the 26bits of the instruction concatenated with the upper 4 bits of the PC

A single operation can use more than one addressing mode.

Quick exercise

- Which addressing modes does the following instruction support?

  lw $t0, 0($t1)

  - A. Immediate addressing
  - B. Register addressing
  - C. PC-relative addressing
  - D. Displacement addressing

32-bit immediate operands

- Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger.

- Load upper immediate (lui)
  - Set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of a constant

The machine language version of `lui $t0, 255   # $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

Example

- What is the MIPS assembly code to load a 32-bit constant below into register $s0?

- First step

```
lui $s0, 61    # 61 decimal = 0000 0000 0011 1101 binary
```

$$\underbrace{\text{0000 0000 0011 1101}}_{\text{lui}}\ \underbrace{\text{0000 0000 0000 1000}}_{\text{??}}$$

- Second step
  - A. ori $s0, $s0, 4
  - B. ori $s0, $s0, 8
  - C. andi $s0, $s0, 4
  - D. andi $s0, $s0, 8

Communicating with people

- American standard code for information interchange (ASCII)
- A 7-bit character set containing 128 characters

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
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

Instructions for handling bytes

- Store byte (sb): takes a byte from the rightmost 8 bits of a register and writes it to memory.

```
lb $t0,0($sp)          # Read byte from source
sb $t0,0($gp)          # Write byte to destination
```

- Load byte (lb): loads a byte from memory, placing it in the rightmost 8 bits of a register and sign extends it to fill the 32 bit register.

- Unsigned load byte (lbu): treats the byte as an unsigned number and fills the remainder of the 32 bit register with 0s.
  - C programs almost always use bytes to represent characters rather than short signed integers.
  - lbu is used practically exclusively for byte loads.

Strings

- Characters are often combined into strings
- Ways to represent a string?
  - The first position of the string is reserved to give the length of a string.
    - Problem?
  - Define a structure with an accompanying variable that contains the length of a string (used in Java).
  - The last position of a string is indicated by a character used to mark the end of a string (used in c).
- It is very important for a computer system to have efficient string manipulation routines.

A string handling example

- Compiling a string copy procedure, showing how to use c strings.

```
void strcpy (char x[], char y[])
{
     int i;

     i = 0;
     while ((x[i] = y[i]) != '\0') /* copy & test byte */
      i += 1;
}
```

Assign $a0 to base of x array,
$a1 to base of y array,
$s0 is the index.

MIPS assembly code

```
strcpy:
     addi  $sp,$sp,-4     # adjust stack for 1 more item
     sw    $s0, 0($sp)    # save $s0
     add   $s0,$zero,$zero # i = 0 + 0
L1:  add   $t1,$s0,$a1    # address of y[i] in $t1
     lbu   $t2, 0($t1)    # $t2 = y[i]
     add   $t3,$s0,$a0    # address of x[i] in $t3
     sb    $t2, 0($t3)    # x[i] = y[i]
     beq   $t2,$zero,L2   # if y[i] == 0, go to L2
     addi  $s0, $s0,1     # i = i + 1
     j     L1             # go to L1
L2:  lw    $s0, 0($sp) # y[i] == 0: end of string. Re-
store old $s0
     addi  $sp,$sp,4      # pop 1 word off stack
     jr    $ra            # return
```

Unicode

- Unicode is an industry standard for consistent text encoding
  - representation and handling of text expressed in most of the world's writing systems

- Java uses Unicode for characters. It uses 16 bits (halfword) to represent a character.

- MIPS instructions for handling Unicode characters.
  - Load halfword: lh, lhu (unsigned)
  - Save halfword: sh

```
lhu $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

Quick exercise

- How many bytes does the string "hello world!" take up?
  - Use a word to keep the string length
  - Characters are encoded in ASCII

- A. 3
- B. 4
- C. 12
- D. 13

Quick exercise

- Which of the following statements about working with strings in MIPS is incorrect?
  - A. To load individual characters from memory or write them back into memory, we use the instructions lc and sc, respectively.
  - B. When loading a character from memory, the numerical value is sign-extended to fill all 32 bits of the register.
  - C. When saving a character into memory, the upper 24 bits in the register are ignored.
  - D. We can compute the length of a string by making the use of the fact that the null terminator for strings is $0.

Decode machine language

op(31:26)								
28–26 31–29	[redacted](000)	[redacted](001)	[redacted](010)	[redacted](011)	[redacted](100)	[redacted](101)	[redacted](110)	[redacted](111)
[redacted](000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
[redacted](001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
[redacted](010)	TLB	FlPt						
[redacted](011)								
[redacted](100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
[redacted](101)	store byte	store half	swl	store word			swr	
[redacted](110)	load linked word	lwc1						
[redacted](111)	store cond. word	swc1						

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	[redacted](000)	[redacted](001)	[redacted](010)	[redacted](011)	[redacted](100)	[redacted](101)	[redacted](110)	[redacted](111)
[redacted](000)	shift left logical		shift right logical	sra	sllv		srlv	srav
[redacted](001)	jump register	jalr			syscall	break		
[redacted](010)	mfhi	mthi	mflo	mtlo				
[redacted](011)	mult	multu	div	divu				
[redacted](100)	add	addu	subtract	subu	and	or	xor	not or (nor)
[redacted](101)			set l.t.	set l.t. unsigned				
[redacted](110)								
[redacted](111)								

Check yourself

- What is the MIPS assembly language instruction corresponding to the machine instruction with the value 0000 0000$_{hex}$?
  - A. add $t0, $t0, $t0
  - B. lw $t0, 0($t0)
  - C. sll $0, $0, 0
  - D. srl $0, $0, $0

Quick exercise

- Which of the following appears to be correct?
  - A. More powerful instructions mean higher performance.
  - B. Write in assembly language to obtain the highest performance.
  - C. The importance of commercial binary compatibility means successful instruction sets never change.
  - D. None of the above.