What does this C code do?

str len

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

pointer arithmetic

Machine Language and Pointers

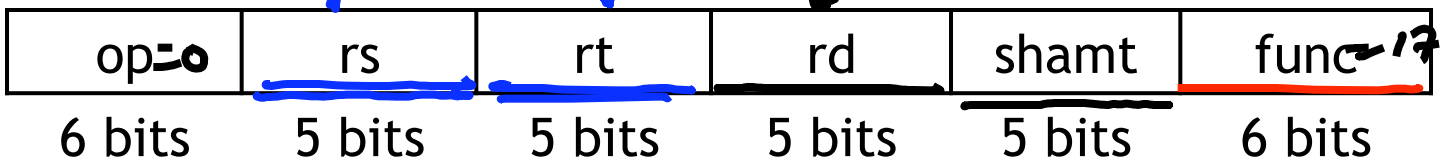
- Today we'll discuss machine language, the binary representation for instructions.
  - We'll see how it is **designed for the common case**
    - Fixed-sized (32-bit) instructions
    - Only 3 instruction formats
    - Limited-sized immediate fields
- Array Indexing vs. Pointers
  - Pointer arithmetic, in particular

Assembly vs. machine language

- So far we've been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU.
- MIPS machine language is designed to be easy to decode.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

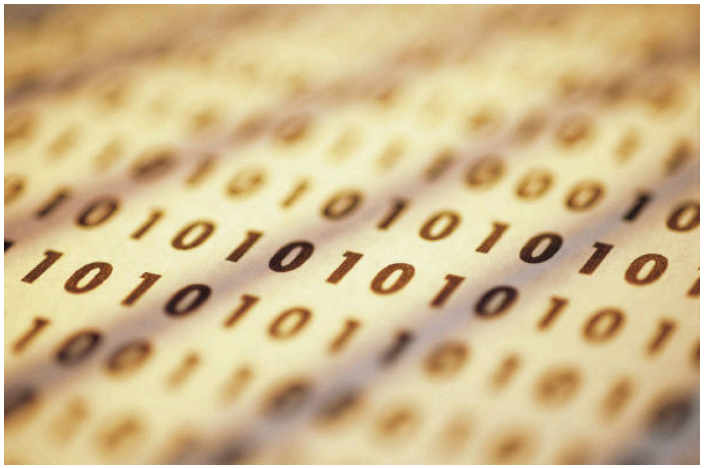
- Register-to-register arithmetic instructions use the R-type format.



- This format includes six different fields.
  - op is an operation code or opcode that selects a specific operation.
  - rs and rt are the first and second source registers.
  - rd is the destination register.
  - shamt is only used for shift instructions.
  - func is used together with op to select an arithmetic instruction.
- The inside cover of the textbook lists opcodes and function codes for all of the MIPS instructions.

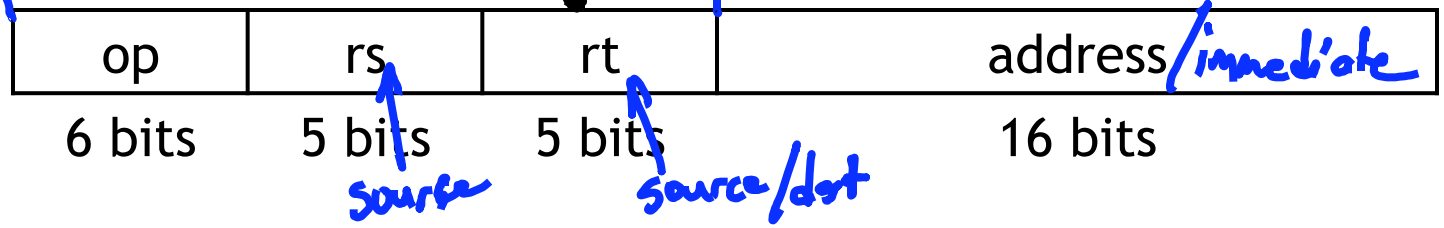
About the registers

- We have to encode register names as 5-bit numbers from 00000 to 11111.
  - For example, $t8 is register $24, which is represented as 11000.
  - The complete mapping is given on page A-23 in the book.
- The number of registers available affects the instruction length.
  - Each R-type instruction references 3 registers, which requires a total of 15 bits in the instruction word.
  - We can't add more registers without either making instructions longer than 32 bits, or shortening other fields like op and possibly reducing the number of available operations.



I-type format

- Load, store, branch and immediate instructions all use the I-type format.



- For uniformity, op, rs and rt are in the same positions as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions.
  - rt is a source register for branches and stores, but a destination register for the other I-type instructions.
- The address is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767.
  - But that's not always enough!

Larger constants

- Larger constants can be loaded into a register 16 bits at a time.
  - The load upper immediate instruction lui loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  - An immediate logical OR, ori, then sets the lower 16 bits.
- To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui $s0, 0x003D              # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900         # $s0 = 003D 0900
```

- This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.
- Pseudo-instructions may contain large constants. Assemblers including SPIM will translate such instructions correctly.
  - Yay, SPIM!!

Loads and stores

- The limited 16-bit constant can present problems for accesses to global data.
  - As we saw in our memory example, the assembler put our result variable at address 0x10010004.
  - 0x10010004 is bigger than 32,767
- In these situations, the assembler breaks the immediate into two pieces.

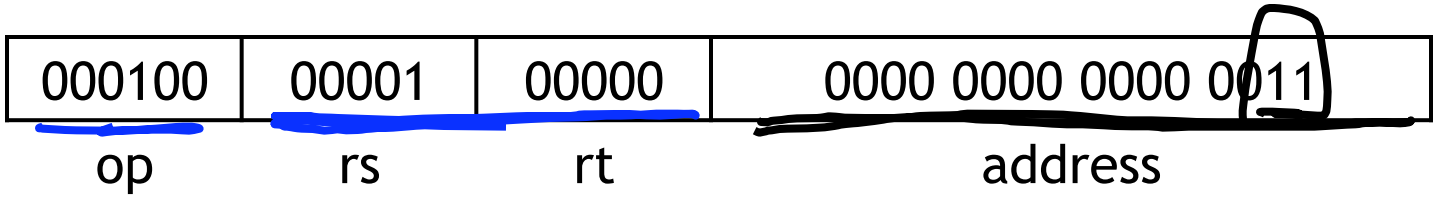
```
lui   $at, 0x1001                  # 0x1001 0000
lw    $t1, 0x0004($at)             # Read from Mem[0x1001 0004]
```

Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

```
        beq   $at, $0, L
        add   $v1, $v0, $0
        add   $v1, $v1, $v1
        j     Somewhere
L:      add   $v1, $v0, $v0
```

- Since the branch target L is three *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:



- SPIM's encoding of branches offsets is off by one, so the code it produces would contain an address of 4. (But it has a compensating error when it executes branches.)

Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if "Far" is very far away, then the effect of:

```
beq    $s0, $s1, Far
...
```

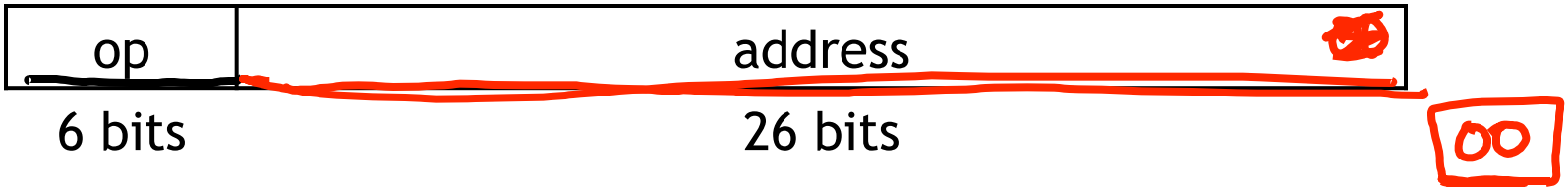
can be simulated with the following actual code.

```
      bne    $s0, $s1, Next
      j      Far
Next: ...
```

- Again, the MIPS designers have taken care of the common case first.

J-type format

- Finally, the jump instruction uses the J-type instruction format.



- The jump instruction contains a *word* address, not an offset
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - So instead of saying "jump to address 4000," it's enough to just say "jump to instruction 1000."
  - A 26-bit address field lets you jump to any address from 0 to 2^{28}.
    - your MP solutions had better be smaller than 256MB

- For even longer jumps, the jump register, or jr, instruction can be used.

```
jr      $ra       # Jump to 32-bit address in register $ra
```

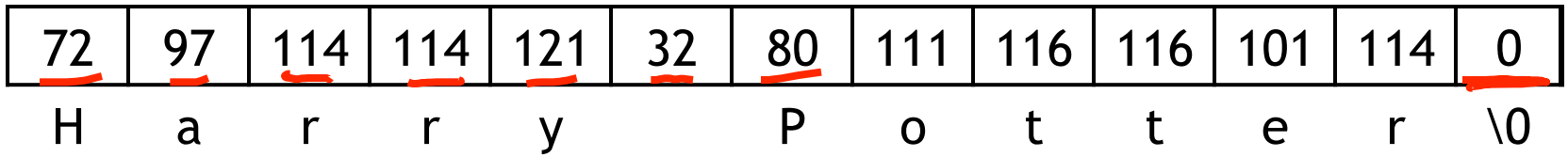
Representing strings

- A C-style string is represented by an array of bytes. ← 1B
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array. null-terminate

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.

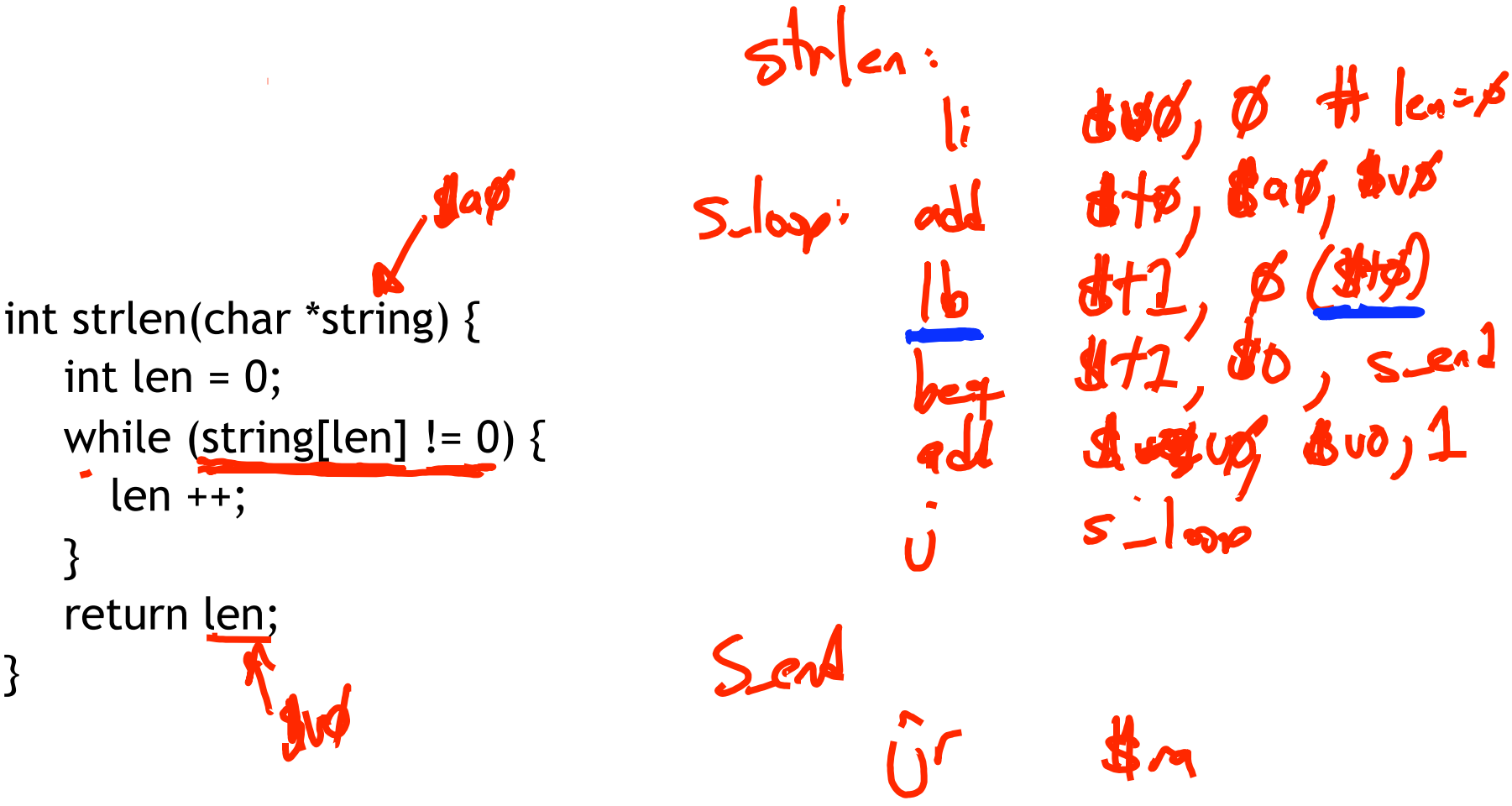


- Since strings can vary in length, we put a 0, or null, at the end of the string.
  - This is called a null-terminated string

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

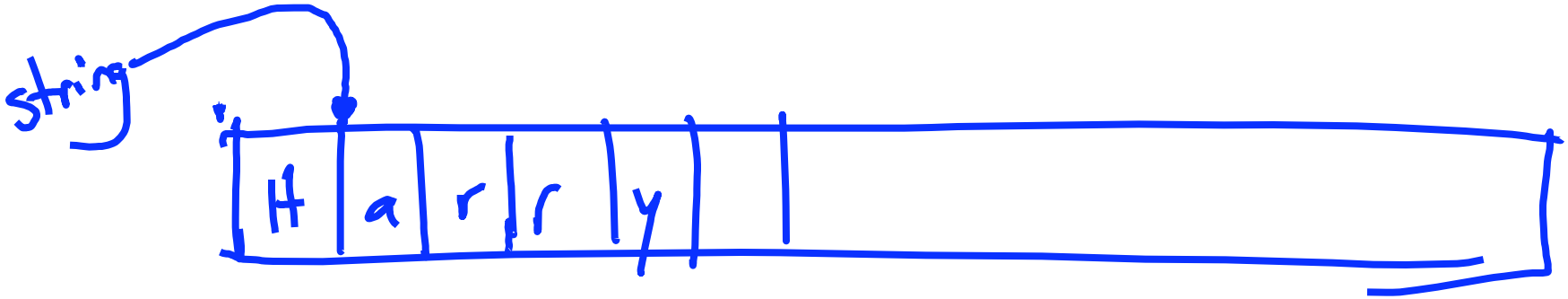
Array Indexing Implementation of strlen

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



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 point 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:
        li   $v0, 0
sloop:  lb   $t0, 0($a0)
        beq  $t0, $0, s_end
        add  $a0, $a0, 1
        add  $v0, $v0, 1
        j    s_loop
send:
        jr   $ra
```

Summary

- Machine language is the binary representation of instructions:
  - The format in which the machine actually executes them
- MIPS machine language is designed to simplify processor implementation
  - Fixed length instructions
  - 3 instruction encodings: R-type, I-type, and J-type
  - Common operations fit in 1 instruction
    - Uncommon (e.g., long immediates) require more than one
- 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)