CS 201

Digital Logic

More Assembly

To do

- Go back over some of the lab material, to make sure everyone is on the same page
- Go into some detail on an example
- Go back and enumerate more assembly language options

Sample Assembly Program: recall from the lab

```
    .text
       .globl __start
__start:                # execution starts here

       la $a0,str       # put string address into a0
       li $v0,4         # system call to print
       syscall          # out a string

       li $v0,10 # Exit
       syscall          # Bye!

       .data
str:   .asciiz "hello world\n"
```

```
        .text
          .globl __start
__start:

          la $a0,str
          li $v0,4
          syscall

          li $v0,10
          syscall

          .data
str:      .asciiz "hello world\n"
```

Text segment of code: mainly for instructions

Symbol called "__start" can be accessed from other programs (specifically linker)

Label indicates the corresponding address of start

```
        .text
          .globl __start
__start:

          la $a0,str
          li $v0,4
          syscall

          li $v0,10
          syscall

          .data
str:      .asciiz "hello world\n"
```

address of “str” into $a0

load the value “4” into $v0

Tell the operating system to do something

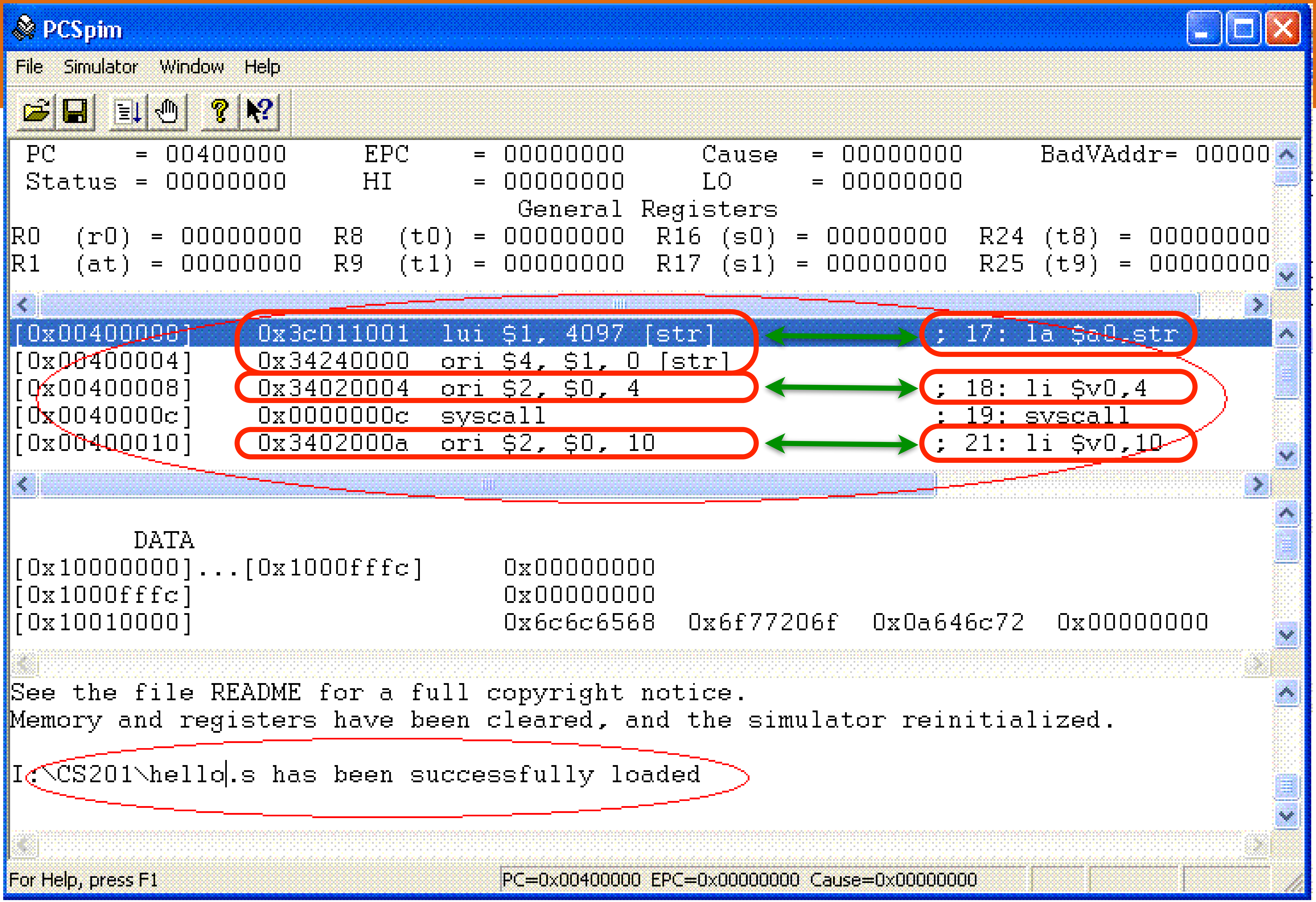
syscall

- Pass control to the OS
- the OS will check register $v0 to see what is expected
  - if $v0=4, OS will print a message on the screen, starting from the address in $a0

  - Only register $v0 and $a0 are consulted during sys call
  - called *implicit* or *implied* operands.

`la` and `li`

- `la` = load address
  - moves the address of the operand into the indicated register, instead of the value of the operand
- `li` = load immediate
  - Loads an immediate value into a register
  - Registers are 32 bits, immediate values are only 16 bits.
- These are pseudo-instructions, which get changed by the assembler into real instructions

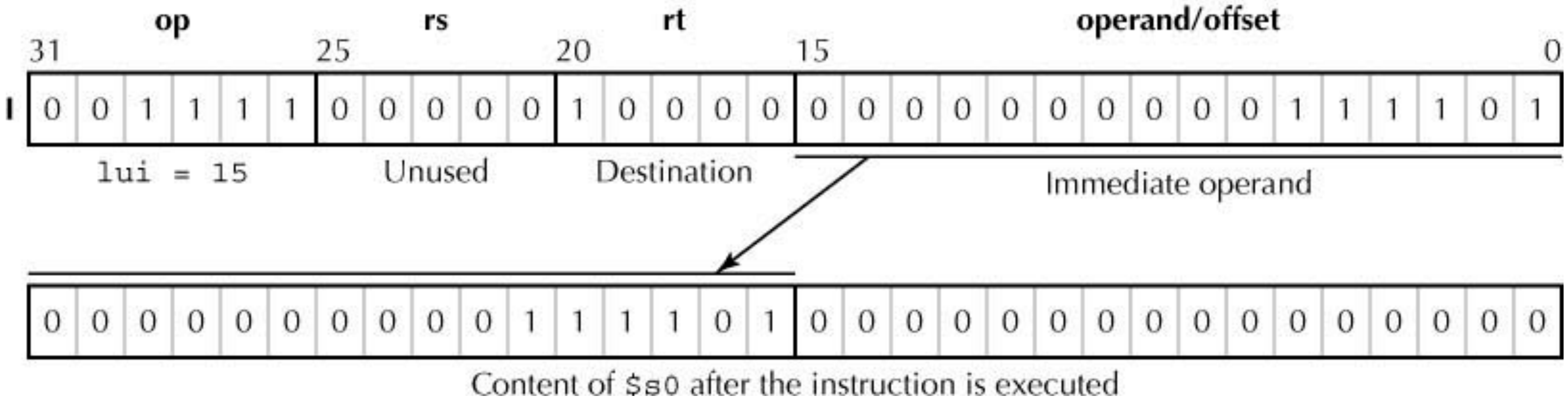

PCSpim
File Simulator Window Help
PC = 00400000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000
Status = 00000000 HI = 00000000 LO = 00000000
General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
[0x00400000] 0x3c011001 lui $1, 4097 [str] ; 17: la $a0,str
[0x00400004] 0x34240000 ori $4, $1, 0 [str]
[0x00400008] 0x34020004 ori $2, $0, 4 ; 18: li $v0,4
[0x0040000c] 0x0000000c syscall ; 19: syscall
[0x00400010] 0x3402000a ori $2, $0, 10 ; 21: li $v0,10
DATA
[0x10000000]...[0x1000fffc] 0x00000000
[0x1000fffc] 0x00000000
[0x10010000] 0x6c6c6568 0x6f77206f 0x0a646c72 0x00000000
See the file README for a full copyright notice.
Memory and registers have been cleared, and the simulator reinitialized.
I:\CS201\hello.s has been successfully loaded
For Help, press F1
PC=0x00400000 EPC=0x00000000 Cause=0x00000000

Pseudo-instructions

- On your sheet as "P" format
  - as opposed to R, I or J
- Assembler decodes these into "regular" instructions
  - eg. from your lab
  - `la $a0 str` was translated into
    - `lui $1 4097`
    - `ori $4, $1, 0`

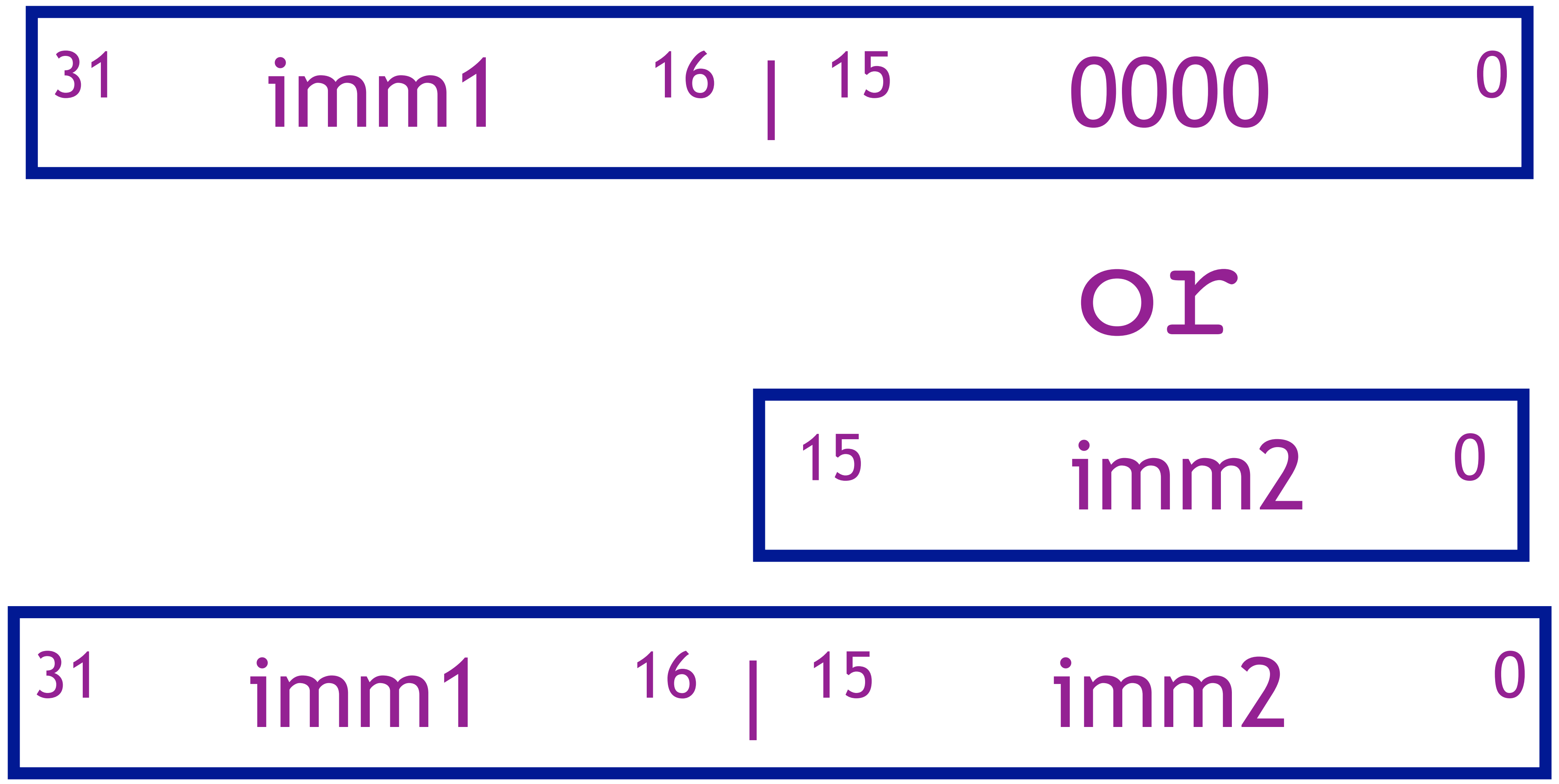
`lui $r, imm`   # Load Upper Immediate

- Immediate values are only 16 bits long
- LUI puts those 16 bits in the top of a register instead of the bottom
- Grabs the immediate value, shifts it left by 16, and writes it to the indicated register. Bottom half of the register is zeros.



`lui` and `ori` together

- To put a 32-bit number into a register, use `lui` and `ori`.
  - `lui` sets the top 16 bits to the desired value and clears the bottom 16 bits
  - `ori` changes the bottom 16 bits without altering the top 16 bits



the `la` pseudo instruction

- Loading an address is a common task
  - Figuring out how to split the address into pieces and using `lui` and `ori` is annoying
  - the pseudo-instruction does this for us
    - the assembler generates the immediate values for the address, and distributes them between the two real instructions.

```
        .text
            .globl __start
__start:

            la $a0,str
            li $v0,4
            syscall

            li $v0,10
            syscall

            .data
str:        .asciiz "hello world\n"
```

load the value “10” into $v0

Tell the operating system to do something

More on `syscall`

Service	System call	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	
exit	10		

sbrk = segment break = dynamical memory

```
        .text
          .globl __start
__start:

          la $a0,str
          li $v0,4
          syscall

          li $v0,10
          syscall

          .data
str:      .asciiz "hello world\n"
```

Data segment

Not code.

mainly for

data

Put the

correspondin

g string

into memory,

and label

the address

of the first

byte ~~add a~~

"null" at

the end

Segments

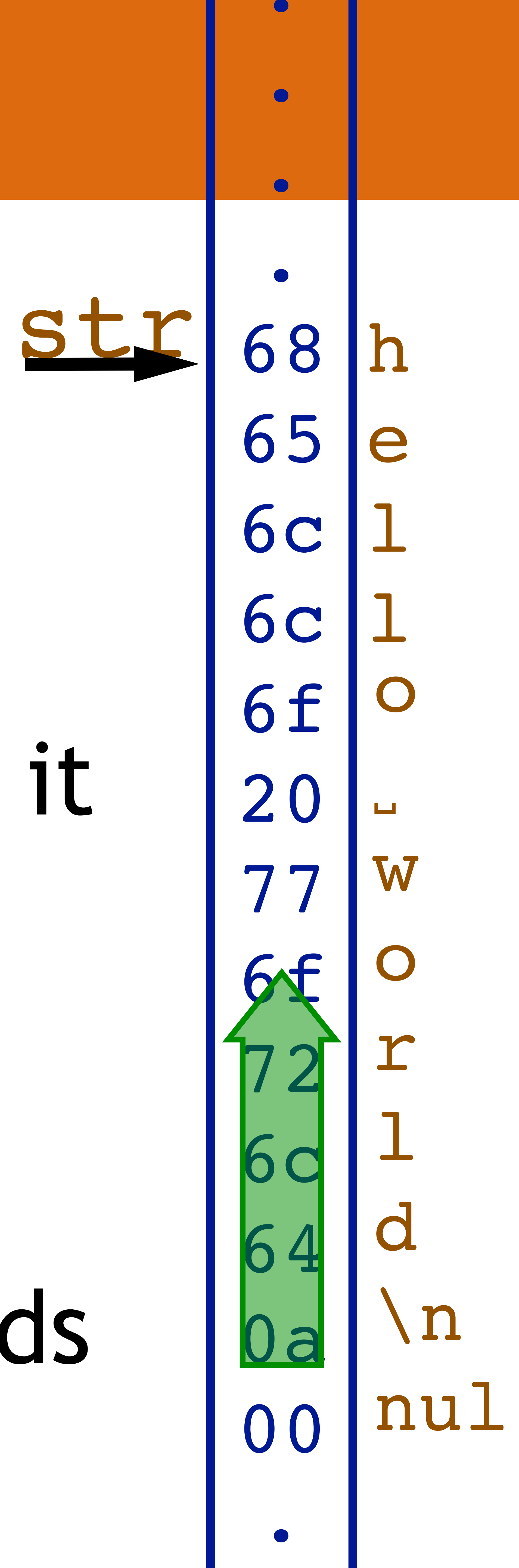
- .data and .text indicate that the assembler is to put the following information into the instruction memory or the data memory
  - or different parts of one big memory
- .data and .text can go in either order

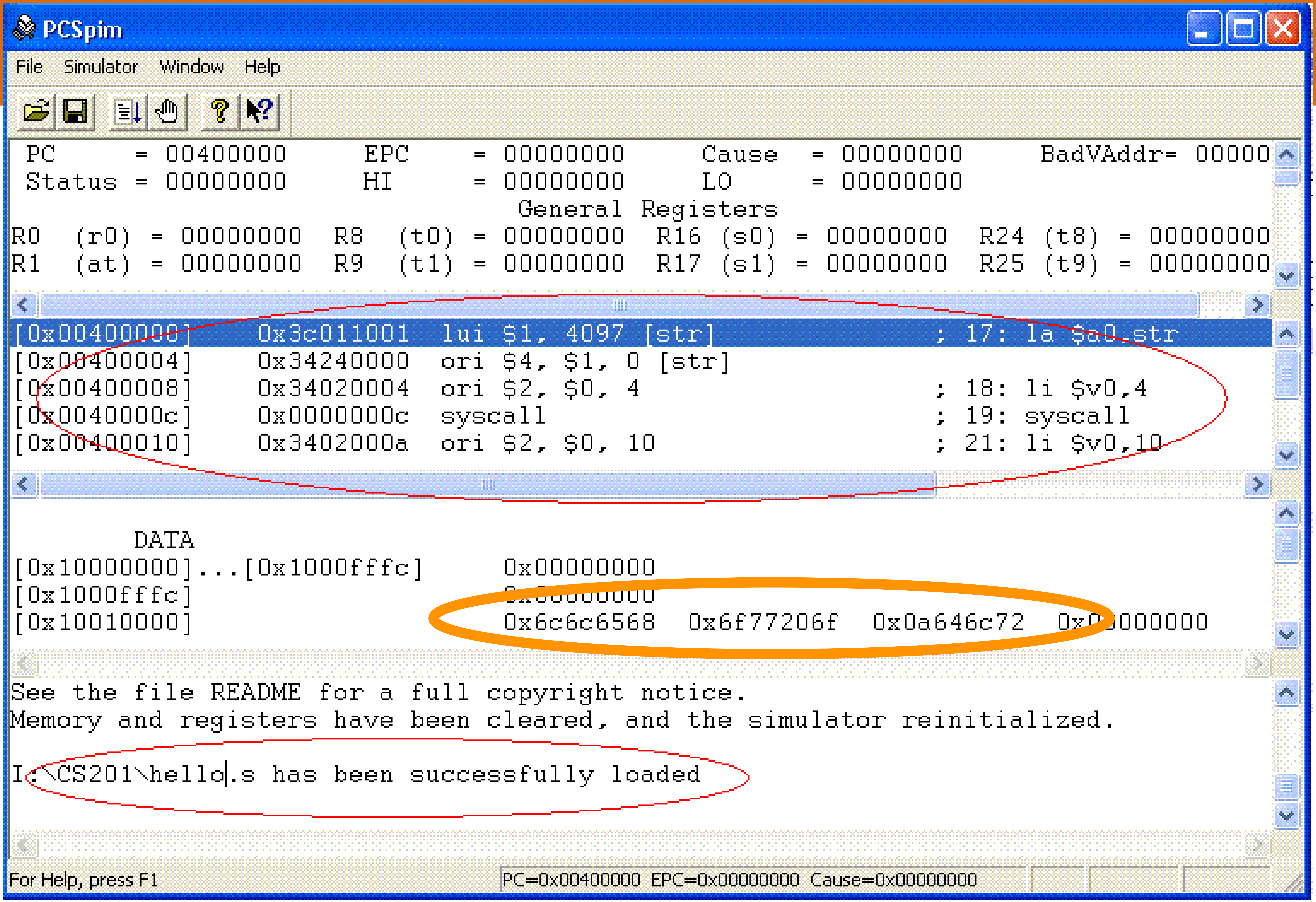
Placing Strings in Memory

- `str: .asciiz "hello world\n"`
- places data in memory
  - Null-terminated
    - `.ascii` doesn't null-terminate
  - `syscall` with `$v0=4` prints characters until it sees `nul` (`00`)
- in the assembler, something strange:

`0x6c6c6568 0x6f77206f 0x0a646c72`

- each 32-bit word (4 ascii chars) looks backwards
  - MIPS is little-endian.




PCSpim
File Simulator Window Help
PC = 00400000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000
Status = 00000000 HI = 00000000 LO = 00000000
General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
[0x00400000] 0x3c011001 lui $1, 4097 [str] ; 17: la $a0,str
[0x00400004] 0x34240000 ori $4, $1, 0 [str]
[0x00400008] 0x34020004 ori $2, $0, 4 ; 18: li $v0,4
[0x0040000c] 0x0000000c syscall ; 19: syscall
[0x00400010] 0x3402000a ori $2, $0, 10 ; 21: li $v0,10
DATA
[0x10000000]...[0x1000fffc] 0x00000000
[0x1000fffc] 0x00000000
[0x10010000] 0x6c6c6568 0x6f77206f 0x0a646c72 0x00000000
See the file README for a full copyright notice.
Memory and registers have been cleared, and the simulator reinitialized.
I:\CS201\hello.s has been successfully loaded
For Help, press F1
PC=0x00400000 EPC=0x00000000 Cause=0x00000000

Encodings

- Converting from assembly language to machine language and back
- to verify assembled code
- eg. `lui $1 4097 = 0x3c011001`
  - We saw how 4097 = 0x1001 = 0b 0001 0000 0000 0001
  - 0x3c01 = 0b 0011 1100 0000 0001
  - fields are 6 for opcode, then 5 and 5 for registers
  - 001111 = 0F = lui (from sheet)
  - 00000 = $0 = zero; 00001 = $1 = assembler temp.

Encodings: the sheet

Move	move	P	move rd, rs	R[rd]=R[rs]		
Load Byte	lb	I		R[rt]={24'b0, M[R[rs]+ZeroExtImm](7:0)}	(3)	20
Load Byte Unsigned	lbu	I	lw $s1,100($s2) $s1 = Mem[100+$s2]	R[rt]={24'b0, M[R[rs]+SignExtImm](7:0)}	(2)	24
Load Halfword	lh	I		R[rt]={16'b0, M[R[rs]+ZeroExtImm](15:0)}	(3)	25
Load Halfword Unsigned	lhu	I	lui $s1,100 $s1 = 100*2^16	R[rt]={16'b0, M[R[rs]+SignExtImm](15:0)}	(2)	2[illegible]
Load Upper Imm.	lui	I		R[rt]={imm,16'b0}		f
Load Word	lw	I		R[rt]=M[R[rs]+SignExtImm]	(2)	23
Load Immediate	li	P		R[rd]=immediate		
Load Address	la	P		R[rd]=immediate		
Store Byte	sb	I		M[R[rs]+SignExtImm] (7:0)=R[rt](7:0)	(2)	28
Store Halfword	sh	I	sw $s1,100($s2) Mem[100+$s2]=$s1	M[R[rs]+SignExtImm] (15:0)=R[rt](15:0)	(2)	29
Store Word	sw	I		M[R[rs]+SignExtImm]=R[rt]	(2)	2b

REGISTERS

NAME	NMBR	USE	STORE?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes
$f0-$f31	0-31	Floating Point Registers	Yes

(1) May cause overflow exception
(2) SignExtImm = {16{immediate[15]},immediate }
(3) ZeroExtImm = {16{1b'0},immediate }
(4) BranchAddr = {14{immediate[15]},immediate,2'b0 }
(4) JumpAddr = {PC[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2 s comp.)

BASIC INSTRUCTION FORMATS, FLOATING POINT INSTRUCTION FORMATS

R	opcode (31–26)	rs (25–21)	rt (20–16)	rd (15–11)	shamt (10–6)	funct (5–0)
I	opcode (31–26)	rs (25–21)	rt (20–16)	immediate (15–0)		
J	opcode (31–26)	immediate (25–0)				
FR	opcode (31–26)	fmt (25–21)	ft (20–16)	fs (15–11)	fd (10–6)	funct (5–0)
FI	opcode (31–26)	fmt (25–21)	rt (20–16)	immediate (15–0)		

FORMAT=P: PSEUDO-INSTRUCTION

The MIPS reference sheet

NOTE: not everything is on this sheet. You should study it and get used to writing code with it.

Encodings: do these Examples:

`add $t1,$v0,$a0  addi $t8,$t9,-1`

NAME	MNEMONIC	FORMAT	OPERATION (in Verilog)		OPCODE/ FUNCT (Hex)
Add	add	R	R[rd]=R[rs]+R[rt]	(1)	0/20
Add Immediate	addi	I	R[rt]=R[rs]+SignExtImm	(1)(2)	8
Add Imm. Unsigned	addiu	I	R[rt]=R[rs]+SignExtImm	(2)	9
Add Unsigned	addu	R	R[rd]=R[rs]+R[rt]	(2)	0/21
Subtract	sub	R	R[rd]=R[rs]-R[rt]	(1)	0/22
Subtract Unsigned	subu	R	R[rd]=R[rs]-R[rt]		0/23

`add $t1,$s1,$s2`
$t1=$s1+$s2

`addi $t1,$s1,10`
$t1=$s1+10

REGISTERS

NAME	NMBR	USE	STORE?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes
$f0-$f31	0-31	Floating Point Registers	Yes

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(4) JumpAddr = { PC[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)

BASIC INSTRUCTION FORMATS, FLOATING POINT INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	immediate				
FR	opcode	fmt	ft	fs	fd	funct
FI	opcode	fmt	rt	immediate		

FORMAT=P: PSEUDO-INSTRUCTION

Boilerplate for assignments and exams

```
        .text
        .globl __start
__start:                        # execution starts here

    ###### Your code goes here ######

        li $v0,10  # Exit
        syscall   # Bye!

        .data

    ###### Your data goes here ######
```

More instructions: Instruction Types

- Arithmetic
  - Adding, subtracting, multiplying, dividing
- Logic
  - AND, OR, NOT, XOR…
- Data Transfer
  - Load from / store to data memory
- Program Flow
  - Branches and jumps

Instruction Types

- We’ve seen the basic operation of these already
  - R, I and J format discussion
- Here we’ll list some of the other instructions available
  - How they work, what’s strange, and how to use them

Arithmetic Instructions

`add $rd, $rs, $rt`

- add rs and rt, put result into rd
- R-format

`addi $rt, $rs, imm`

- add rs and imm, put result into rd
- I-format

`addu $rd, $rs, $rt`

- add unsigned
- add rs and rt as *unsigned numbers*, result into rd
- R-format

Arithmetic Instructions

`sub $rd, $rs, $rt`

- similar to `add`. rs - rt.

`subu $rd, $rs, $rt`

- similar to add unsigned

- No subtract immediate
  - immediate values can be positive or negative. don't need to have a subi.

Logic instructions

`and $rd, $rs, $rt`

- bitwise and of \$rs and \$rt, result into \$rd
- R-format

`andi $rt, $rs, imm`

- bitwise and of \$rs and imm, result into \$rt
- I-format

`or $rd, $rs, $rt`

- like and

`ori $rt, $rs, imm`

- like andi

Masking

- Instructions consist of several fields of information packed together
  - Other data is that way as well
  - eg a 32-bit number consists of 8 hex digits
  - if we want to look at a part of a word, we *mask* it.
    - using AND, we can allow some bits to pass and set some bits to 0
    - set a mask in one register with 1s where we want data, and 0s where we want to ignore

Masking example: select bits

- $a0 = 0x000103FF (data of interest)
- $a1 = 0xFFFF0000 (mask of upper bits)

```
and $a2, $a0, $a1
```

```
0000 0000 0000 0001 0000 0011 1111 1111
1111 1111 1111 1111 0000 0000 0000 0000
```

result:

```
0000 0000 0000 0001 0000 0000 0000 0000
```

Masking example: isolate a field

- `00444820` (instruction)
- `03E00000` (mask of register rs)
- result = 00010 = 2 = $v0

```
0000 0000 0100 0100 0100 1000 0010 0000
0000 0011 1110 0000 0000 0000 0000 0000
```

result:

```
0000 0000 0100 0000 0000 0000 0000 0000
```

Bit Setting

- Just as AND can be used to clear any bit, OR can be used to set any bit
  - e.g. difference in ascii between lower-case and upper-case is 1 bit. Setting this bit will change upper case to lower case

```
0x574f524c = WORL
0101 0111 0100 1111 0101 0010
0100 1100
0010 0000 0010 0000 0010 0000
0010 0000
result:
0111 0111 0110 1111 0111 0010
0110 1100
```

More Logic

`nor` (R-format)

`xor` (R-format)

`xori` (I-format)

- Notable things missing
  - there’s no NOR-immediate
  - there’s no NOT
  - there’s no NAND
  - These things can be done in a couple of instructions
  - There are pseudo-instructions to do some of these

Program flow: Branches and Jumps

- We saw jumps already
  - move to a new instruction
- Branches are different in two ways
  - *conditional*: may or may not move to a different instruction
  - *relative*: the new instruction is specified by an offset from the current instruction, instead of a new address

beq/bne: branch if equal/not equal

`beq $s1, $s2, L1`

- if `$s1` = `$s2`, then jump to the instruction labeled L1, otherwise just go to the next instruction
- the offset is signed (L1 can be before or after the current instruction)
- the offset is encoded relative to the current PC

`bne $s1, $s2, L2`

- if `$s1` ≠ `$s2`, then jump to the instruction labeled L1, otherwise just go to the next instruction

branch relative to the PC of the *next* instruction

- Recall: each instruction takes 4 memory elements
  - branch 10 instructions forward = 40 bytes forward
- After fetch, the computer has ***already incremented the PC by 4*** in preparation for the next instruction

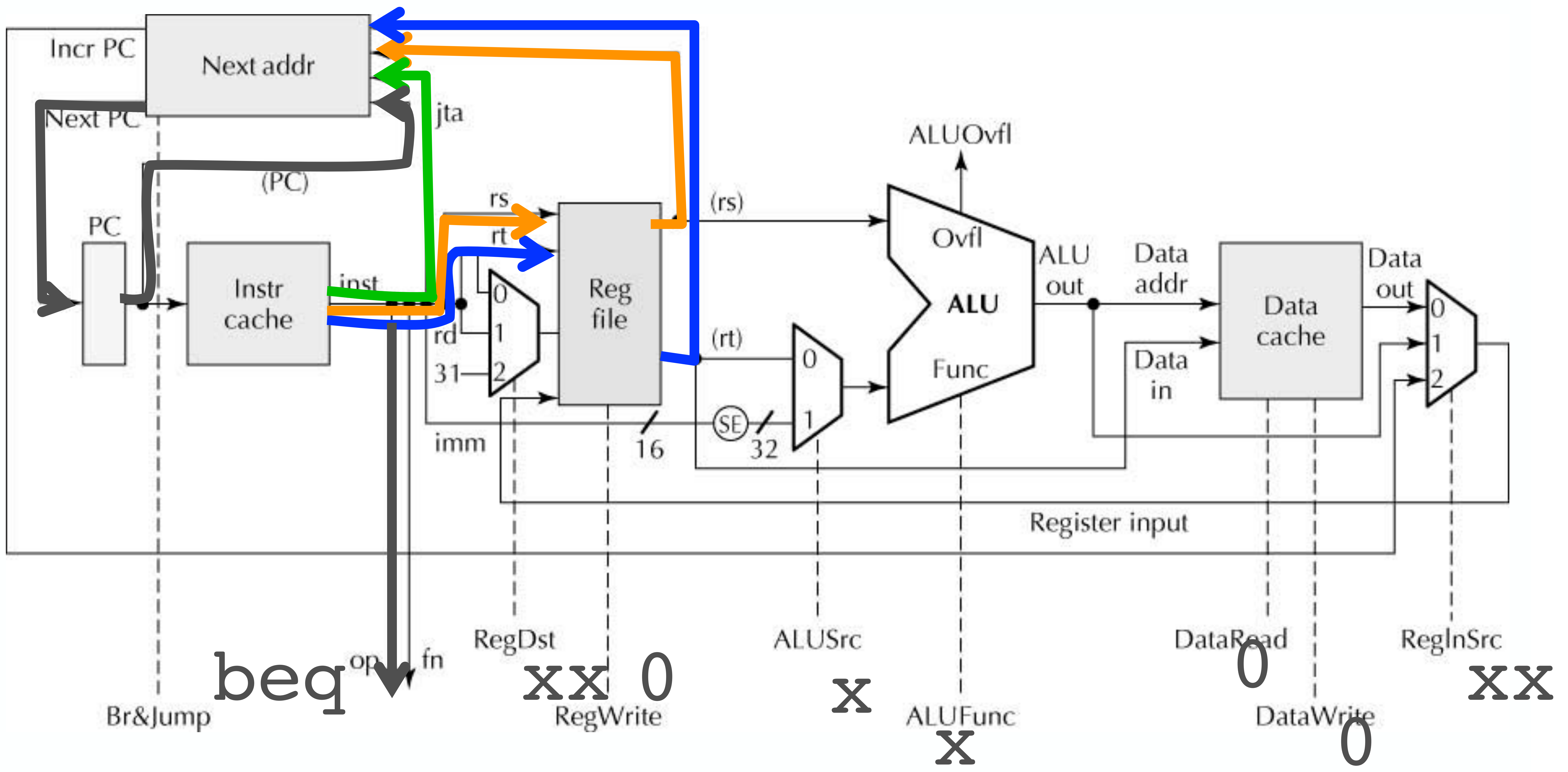
  - 10 instructions past the branch instruction is actually 10*4+4 = 44 bytes from the address of the branch instruction
  - Reminder on sheet:
    - if(R[rs]==R[rt]) PC=PC+4+BranchAddr

branch instructions

	op	rs	rt	operand/offset
	31	25	20	15 … 0
I	0 0 0 1 0 x	1 0 0 0 1	1 0 0 1 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
	beq = 4 bne = 5	Source 1	Source 2	Relative branch distance in words

- Branch is an immediate instruction format
  - branch offset is an immediate value
- Remember, this amount is multiplied by 4 for the byte offset from the next instruction
- $rt and $rs are compared using a dedicated bank of XORs in the nextaddr logic
- offset is shifted left by 2, then added to current PC

branch


Incr PC
Next addr
jta
Next PC
(PC)
PC
Instr cache
inst
rs
rt
rd
31
0
1
2
Reg file
(rs)
(rt)
imm
16
SE
32
0
1
ALUOvfl
Ovfl
ALU
Func
ALU out
Data addr
Data in
Data cache
Data out
0
1
2
Register input
op
fn
beq
RegDst
xx
0
RegWrite
ALUSrc
x
ALUFunc
x
DataRead
0
DataWrite
0
RegInSrc
xx
Br&Jump

branch

- assembler does the math for us
  - you the programmer indicates “branch to a symbol”
  - assembler computes the distance
- furthest branch is 0xFFFF=65,535 instructions away
- if a branch symbol is too far:
  - assembler automatically replaces with a branch and jump:

```
    beq $s0, $s1, L1     # would be replaced with
    bne $s0, $s1, L2
    j L1
L2:...
```

Different branches

- Branching on whether or not two things are equal is good, but we can do better
  - What if we want to check if one thing is bigger than another?
- Set one register based on a comparison
- then compare that register to zero
  - if it is zero, the comparison is false
  - if it is not zero, the comparison is true

set instructions

`slt $rd, $rs, $rt`

- if \$rs < \$rt, \$rd = 1 otherwise \$rd = 0

`slti $rt, $rs, imm`

- if \$rs < imm, \$rt = 1 otherwise \$rt = 0

- unsigned versions also available

  `sltu, sltiu`

- no “set greater-than instruction”
  - just re-arrange your problem: a<b = b>a
- no “set less than or equal to”
  - again, re-arrange your problem: a≤b = (b<a)false

High-level language program flow

- if-then-else
  - test a condition, and do one of two things
- while loop
  - Do something while a condition is true
  - zero or more times
- for loop
  - usually a count-up or count-down
  - can be converted to while
- do loop
  - executed 1 or more times

writing if-then statements

- if condition then statements
  - branches let us skip some instructions based on a condition
  - to write an "if", we must check for the opposite condition, and skip if true
  - e.g. if A = B then C

```
    bne A,B,Lbl
    C
Lbl:...
```

    - Recall: unless a branch or jump is used, the next instruction in the memory will be executed.

if-then example

\$a0=1
if \$a0 ≥ 5 then
\$a0 = \$a0+1
\$a0=\$a0+1

what is the value of `$a0`?

`$a0` = 2

the first "`$a0+1`" is skipped

\$a0 ≥ 5 is opposite of \$a0 < 5
if \$a0 < 5≠0, skip the "then"
if \$a0 < 5=0, do the "then"

```
     li   $a0, 1
     slti $t0, $a0, 5
     bne  $t0, $0,  lbl
     addi $a0, $a0, 1
lbl: addi $a0, $a0, 1
```

Note: `sgei` - `beq` might make more sense but `sgei` doesn't exist

if-then-else statements

- if *condition* then *statements1* else *statements2*
- e.g. if A<B then C else D
  - if A < B = 1 , do C and skip over D
  - if A < B = 0, do D and skip over C

```
      slt Tmp, A, B
      beq Tmp, ZERO, Lb1
      C
      j Lb2
Lb1: D
Lb2: ...
```

if-then-else example

$a0=1
if** $a0 ≥ 5 **then
$a0 = $a0+1
else
$a0 = $a0-1
$a0=$a0+2

```
      li $a0,1
      slti $t0,$a0,5
      bne $t0,$0,lb1
      addi $a0,$a0,1      #then
      j lb2
lb1:  addi $a0,$a0,-1     #else
lb2:  addi $a0,$a0,2      #continue
```

Another way to do if-then-else

- Use the condition from the if statement as-is and reverse the order of the clauses in code
- if the condition is true, skip to the "then" clause
- otherwise, fall through to the "else" clause
- after the " else" clause, skip over the "then" clause

\$a0=1
if** \$a0 ≥ 5 **then
    \$a0 = \$a0+1
else
    \$a0 = \$a0 -1
\$a0=\$a0+2

```
        li $a0,1
        slti $t0,$a0,5
        beq $t0,$0,lb1
        addi $a0,$a0,-1 # else
        j lb2
lb1: addi $a0,$a0,1    # then
lb2: addi $a0,$a0,2    # continue
```

If statements and the opposite condition

- If you want to test for the opposite condition, 2 places to do that
  - set the opposite condition and branch true
    - true = 1, so use branch if not equal zero
    - bne $a1, $0, Label
  - set the real condition and branch if false
    - by using beq $a1, $0, Label
- This is why we have a "zero" register
  - an easy and consistent value to test against

branching pseudoinstructions

```
blt $rs, $rt, Label      # (<). equivalent to:
  slt $at, $rs, $rt
  bne $at, $0, Label
bgt $rs, $rt, Label      # (>). equivalent to:
  slt $at, $rt, $rs
  bne $at, $0, Label
ble $rs, $rt, Label      # (≤). equivalent to:
  slt $at, $rt, $rs
  beq $at, $0, Label
bge $rs, $rt, Label      # (≥). equivalent to:
  slt $at, $rs, $rt
  beq $at, $0, Label
```

while statements

while *(i < k)* ***do*** *i = i+1;*

\# assume $s1 is used for i and $s3 for k

```
loop:    slti $t0, $s1, $s3    # i<k
          beq $t0, $0, endloop # exit condition
          addi $s1,$s1,1
          j loop                 # loop always
endloop:...
```

- Test the loop condition

Program flow notes

- Be careful to test for what you want to test for
- Be careful with unconditional jumps over conditional clauses
- Be careful of fall-through
  - if there is no jump instruction after a conditional clause, execution "falls through" to the next instruction
  - make sure this is what you want to have happen
  - more examples in specific circumstances

Program flow things to try

- Do-while loop
- break and continue within a loop
- Switch /case with default, fall through etc
- nested if/then; nested loops

- What happens if you run out of registers to use as temporary comparison values, loop index variables etc?

Addressing Modes

an *Addressing mode* is a way to get an operand for an instruction

6 addressing modes for MIPS

1. implied: operand is implied in the instruction
   e.g. `jal` (jump and link); syscall
   stores PC in a special register $ra. we'll see
2. immediate: operand value is in instruction
3. register: operand value is in a register indicated by the instruction
4. base (plus offset): operand value is in a memory location indicated by a register base plus immediate offset (e.g. load/store)

Addressing Modes

5. PC-Relative: for branch instructions.
   - similar to base addressing mode, except offset (branch offset) is shifted left by 2, and base register is always program counter. branch offset is immediate
6. Pseudo-direct: for jump instructions
   - direct addressing is when an address is available in the instruction
     - like immediate, except address not data
   - pseudo direct converts the 24 bits remaining after using up 6 bits for opcode, into a 32 bit address
     - as discussed before, with shifting and 4 bits from PC

Addressing Modes