

Assembly Language

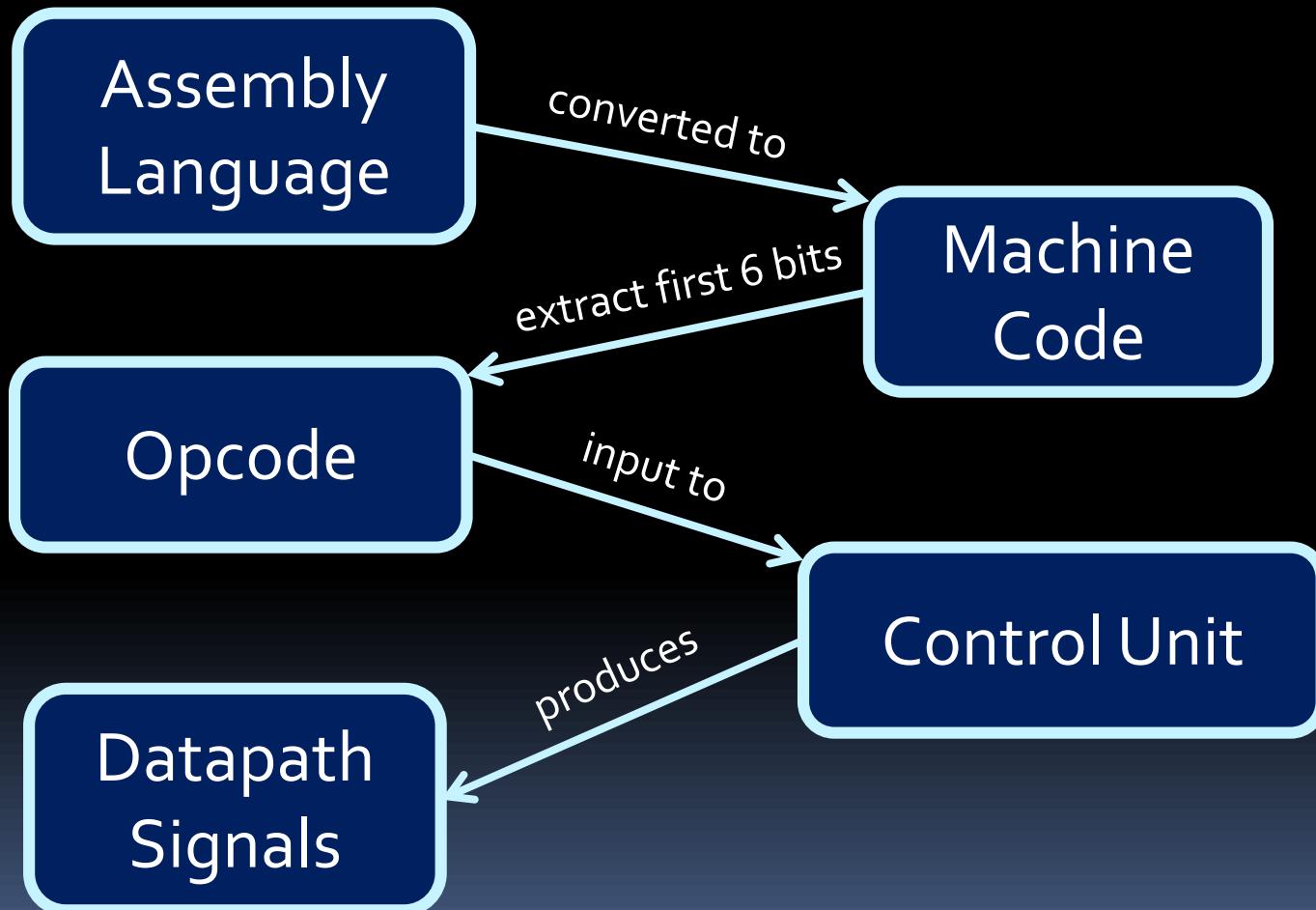
* Created with contributions by Myrto Papadopoulou and Frank Plavec.

Programming the processor

- Things you'll need to know:
 - Control unit signals to the datapath
 - Machine code instructions
 - Assembly language instructions
 - Programming in assembly language



How things fit together



Machine Code Instructions

01 00 FF FF 00 00 00 00 00 00 00 00 00 40 00 CC 80 @
00000010 0C 00 00 00 00 00 26 01 8F 00 00 00 00 00 53 00 . . . & . . . S
00000020 65 00 6C 00 65 00 63 00 74 00 20 00 52 00 75 00 e.l.e.c.t. R.u.
00000030 6C 00 65 00 00 00 08 00 00 00 00 01 4D 00 53 00 l.e. . . M.S
00000040 20 00 53 00 68 00 65 00 6C 00 6C 00 20 00 44 00 . . . S.h.e.l.l. D
00000050 6C 00 67 00 00 00 00 00 00 00 00 00 02 00 00 1.g. . .
00000060 03 01 A1 50 53 00 3A 00 C3 00 36 00 32 25 00 00 . . . P.S. . . 6.2%
00000070 FF FF 83 00 00 00 00 00 00 00 00 00 00 00 00 00 . . .
00000080 03 00 01 50 0E 00 00 56 00 41 00 0A 00 4A 26 00 00 . . . P. V.A. J.k.
00000090 FF FF 80 00 26 00 41 00 70 00 70 00 6C 00 79 00 . . . & A.p.p.l.y
000000a0 20 00 74 00 6F 00 20 00 61 00 6C 00 6C 00 00 00 t.o. a.l.l
000000b0 00 00 00 00 00 00 00 00 00 00 00 01 00 01 50 . . . F
000000c0 7E 00 7D 00 32 00 0E 00 01 00 00 00 FF FF 80 00 ~.}.2 . . .
000000d0 4F 00 4B 00 00 00 00 00 00 00 00 00 00 00 00 O.K.
000000e0 00 00 01 50 B4 00 7D 00 32 00 0E 00 02 00 00 00 . . . P. }.2 . . .
000000f0 FF FF 80 00 43 00 61 00 6E 00 63 00 65 00 6C 00 . . . C.a.n.c.e.l
00000100 00 00 00 00 00 00 00 00 00 00 00 00 01 50 . . . F
00000110 EA 00 7D 00 32 00 0E 00 09 00 00 00 FF FF 80 00 . . . }.2 . . .
00000120 26 00 48 00 65 00 6C 00 70 00 00 00 00 00 00 00 & H.e.l.p . . . P
00000130 00 00 00 00 00 00 00 00 80 08 81 50 0E 00 3A 00 P
00000140 3B 00 0E 00 2F 25 00 00 FF FF 81 00 00 00 00 00 . . . %. . . P
00000150 00 00 00 00 00 00 00 00 00 00 02 50 0E 00 30 00 . . . P. O
00000160 1E 00 08 00 EE 25 00 00 FF FF 82 00 46 00 69 00 . . . %. . . F.i
00000170 6C 00 65 00 20 00 54 00 79 00 70 00 65 00 00 00 l.e. T.y.p.e . . .
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 02 50 . . . F
00000190 54 00 30 00 2C 00 08 00 EE 25 00 00 FF FF 82 00 T.O. . . %. . .
000001a0 50 00 61 00 72 00 73 00 69 00 6E 00 67 00 20 00 P.a.r.s.i.n.g . . .
000001b0 52 00 75 00 6C 00 65 00 73 00 00 00 00 00 00 00 R.u.l.e.s . . .
000001c0 00 00 00 00 00 00 00 00 07 00 00 50 06 00 07 00 . . . P
000001d0 1A 01 71 00 ED 25 00 00 FF FF 80 00 00 00 00 00 . . . q. %. . . P
000001e0 00 00 00 00 00 00 00 00 00 02 50 0E 00 11 00 P
000001f0 3E 00 08 00 EC 25 00 00 FF FF 82 00 53 00 65 00 >. %. . . S.e
00000200 6C 00 65 00 63 00 74 00 20 00 52 00 75 00 6C 00 l.e.c.t. R.u.l.l
00000210 65 00 20 00 46 00 6F 00 72 00 20 00 46 00 69 00 e.F.o.r. F.i
00000220 6C 00 65 00 00 00 00 00 00 00 00 00 00 00 00 l.e. . .
00000230 80 08 81 50 0E 00 1B 00 08 01 0E 00 EB 25 00 00 . . . P. . . %
00000240 FF FF 81 00 00 00 00 00 00 00 00 00 00 00 00 00 . . .
00000250 00 00 02 50 19 00 61 00 37 00 08 00 6B 26 00 00 . . . P.a.7..k&
00000260 FF FE 82 00 00 00 00 00 00 00 00 00 00 00 00 00 . . .

Intro to Machine Code

- Now that we have a processor, operations are performed by:
 - The **instruction register**:
 - Sending instruction components to the processor.
 - The **control unit**:
 - Based on the **opcode** value (sent from the instruction register), sending a sequence of signals to the rest of the processor.
- Only questions remaining:
 - **Where do these instructions come from?**
 - How are they provided to the instruction memory?

Assembly language

- Each processor type has its own language for representing, say 32-bit, instructions as user-level code words.
- Example: $C = A + B$
 - Assume A is stored in \$t1, B in \$t2, C in \$t3.
 - **Assembly language instruction:**

```
add $t3, $t1, $t2
```

- **Machine code instruction:**

```
000000 01001 01010 01011 XXXXX 100000
```

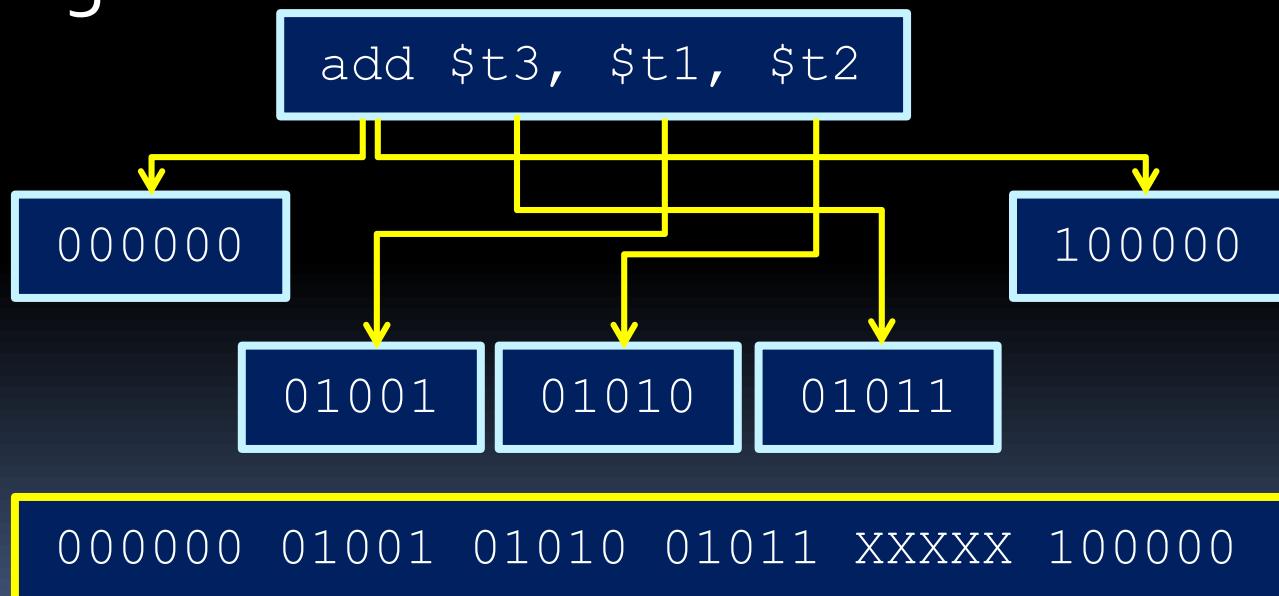
Note: There is a 1-to-1 mapping for all assembly code and machine code instructions!

Encoding the instruction

- Machine code instructions contain all the details about a processor operation, such as:
 - What operation is being performed (**opcode**),
 - What registers are being used in this operation,
 - What other information might be needed to make this operation happen (**immediate or shift values**)
- When we write (or interpret) a machine code instruction, we **need to know how to encode** (or decode) these details into these 32 bits.

R-type instructions

- For instance, how do we encode the earlier instruction that adds registers \$t1 and \$t2 and stores the result into register \$t3?.
- e.g.



Operating on Registers

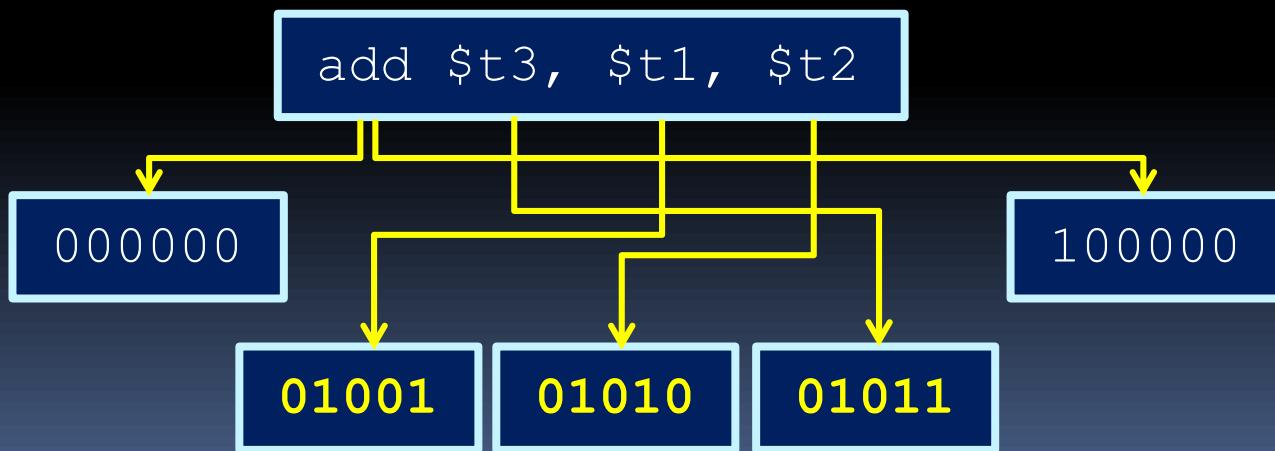
- The **add** instruction is one of many operations that processes two register values and stores the result in a third.
 - This is called an **R-type** instruction.
 - Any operations whose **inputs and outputs are all registers** are called R-type, even if they involve less than three registers.
 - e.g. the **j_r** instruction, which we get to later.
- In order to encode R-type instructions, we **need to know** the **5-bit codes** used to refer to our input and output **registers**.

Machine code + registers

- MIPS is **register-to-register**.
 - Almost all operations rely on register data.
- MIPS provides 32 registers (with numerical and label names).
 - Some have special values:
 - Register \$0 (\$zero): value 0 -- always.
 - Register \$1 (\$at): reserved for the assembler.
 - Registers \$28–\$31 (\$gp, \$sp, \$fp, \$ra): memory and function support
 - Registers \$26–\$27: reserved for OS kernel
 - Some are used by programs as functions parameters:
 - Registers \$2–\$3 (\$v0, \$v1): return values
 - Registers \$4–\$7 (\$a0-\$a3): function arguments
 - Some are used by programs to store values:
 - Registers \$8–\$15, \$24–\$25 (\$t0–\$t9): temporaries
 - Registers \$16–\$23 (\$s0–\$s7): saved temporaries
 - Also three special registers (PC, HI, LO) that are not directly accessible.
 - HI and LO are used in multiplication and division, and have special instructions for accessing them.

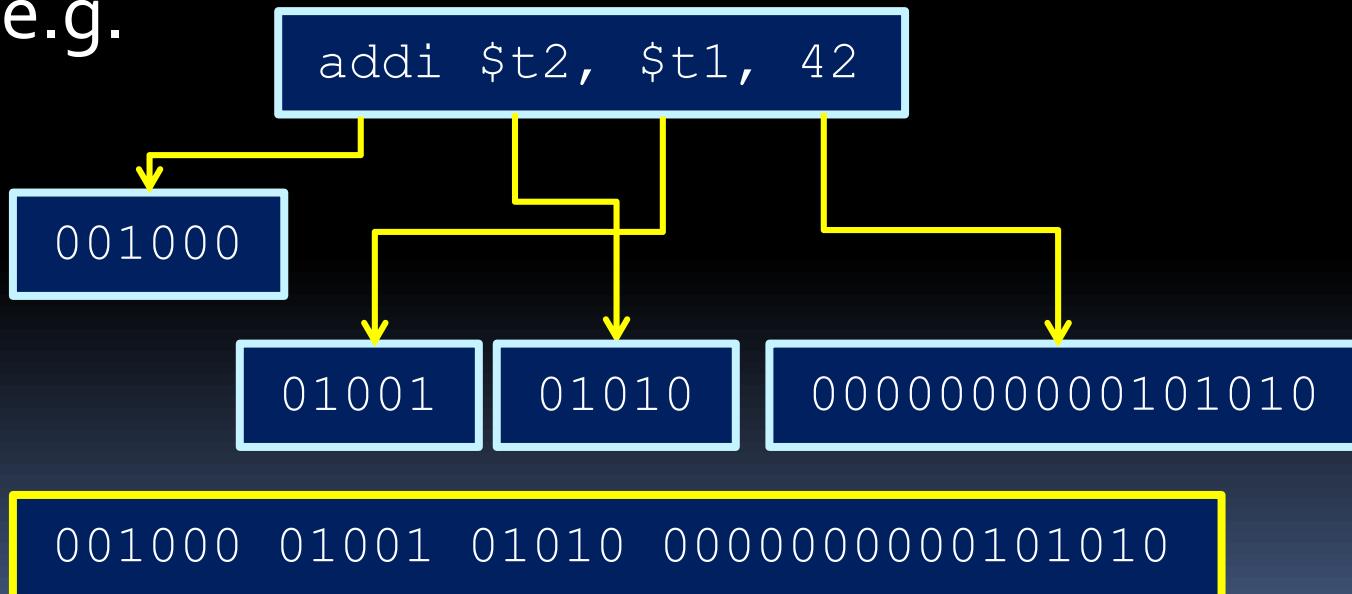
Filling in the blanks

- In our previous example, registers \$t1 and \$t2 are registers 9 and 10 respectively, and register \$t3 is the 11th register.
- The registers for this instruction are encoded as 01001 (\$t1), 01010 (\$t2) and 01011 (\$t3).



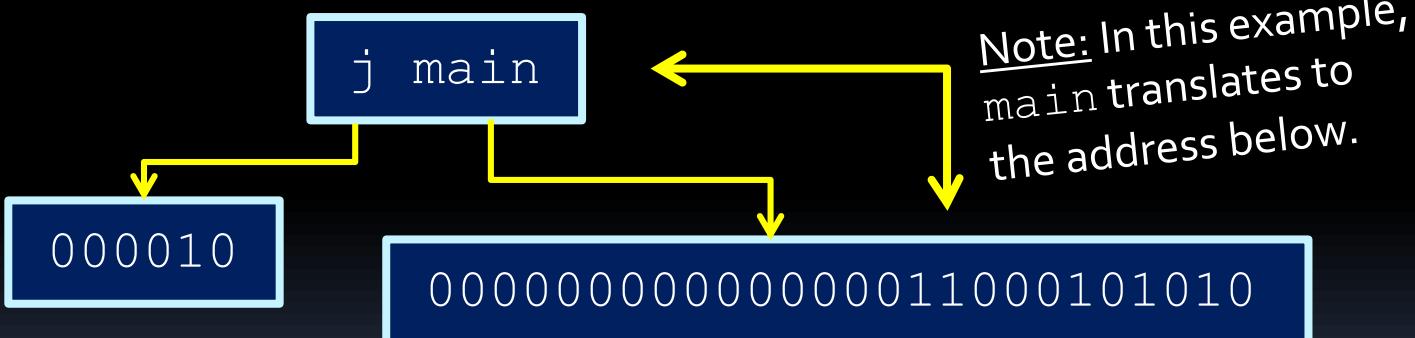
I-type instructions

- I-type instructions **also** operate on **registers**, but **involve** a **constant** value as well.
 - This constant is encoded in the last 16 bits of the instruction.
- e.g.



J-type instructions

- J-type instructions **jump to a location** in memory encoded by the last 26 bits of the instruction (everything but the opcode).
 - This location is stored as a **label**, which is **resolved when** the assembly program is **compiled**.
 - More later on how these 26 bits store jump addresses.
- e.g.



000010 000000000000000011000101010

Review: MIPS instruction types

- R-type:



- I-type:



- J-type:



Machine code details

- Things to note about machine code:
 - R-type instructions have an **opcode** of 000000, with a 6-bit function listed at the end.
 - Although we specify “**don’t care**” bits as X values, the assembly language interpreter always assigns them to **some value** (like 0).
- It’s possible to program your processor with machine code, but makes **more sense** to use an **equivalent language** that is **more natural** (for humans, that is).

Assembly Language Overview

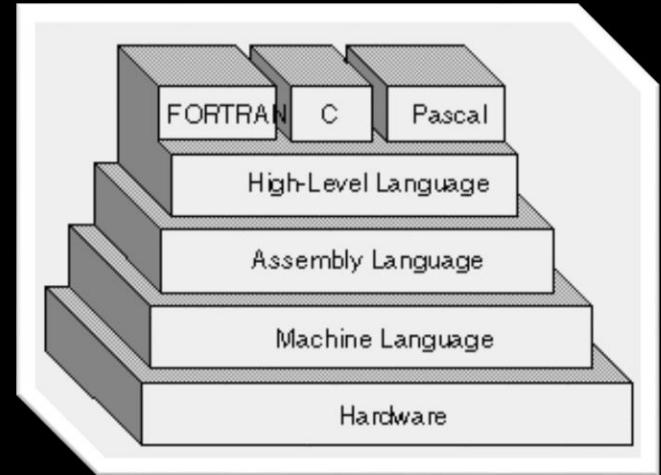
```
loop: lw    $t3, 0($t0)
      lw    $t4, 4($t0)
      add  $t2, $t3, $t4
      sw    $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```

Assembler

```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20ffff9
```

Assembly language

- Assembly language is the **lowest-level** language that you'll ever program in.
- Many **compilers** translate their high-level program commands into **assembly commands**, which are then **converted** into machine code and used by the processor.
- Note: There are **multiple types** of assembly language, especially for different architectures!

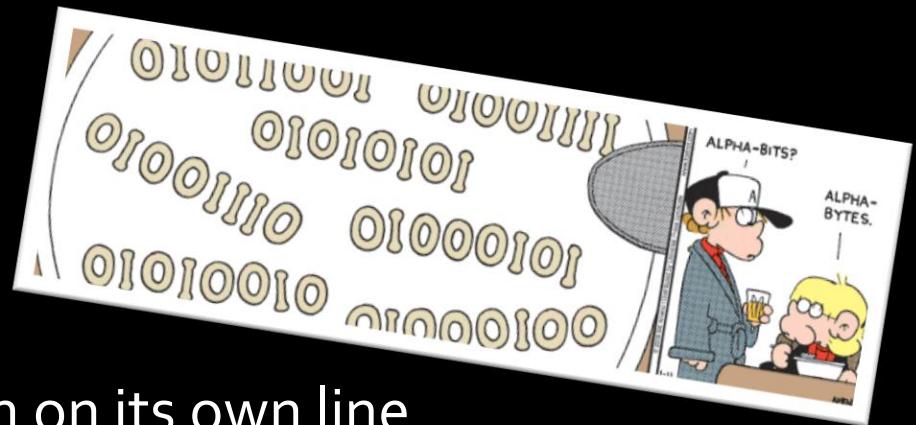


A little about MIPS

- MIPS
 - Short for Microprocessor without Interlocked Pipeline Stages
 - A type of RISC (Reduced Instruction Set Computer) architecture.
 - Provides a set of simple and fast instructions
 - Compiler translates instructions into 32-bit instructions for instruction memory.
 - Complex instructions (e.g. multiplication) are built out of simple ones by the compiler and assembler.

MIPS Instructions

- Things to note about MIPS instructions:
 - Instruction are written `<instr> <parameters>`
 - Each instruction is written on its own line
 - All instructions are 32 bits (4 bytes) long
 - Instruction addresses are measured in bytes, starting from the instruction at address 0.
 - Therefore, all instruction addresses are divisible by 4.
- The following tables show the most common MIPS instructions, the syntax for their parameters, and what operation they perform.



Frequency of instructions

Instruction Type	Examples	Usage	Integer Frequency	Floating point Frequency
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	operations in assignment statements	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	If statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, returns, and case/switch statements	2%	0%

Original source: *Computer Organization And Design: The Hardware/Software Interface*, 5th Edition, Patterson & Hennessy, 2014, p163

Assembly Language Instructions

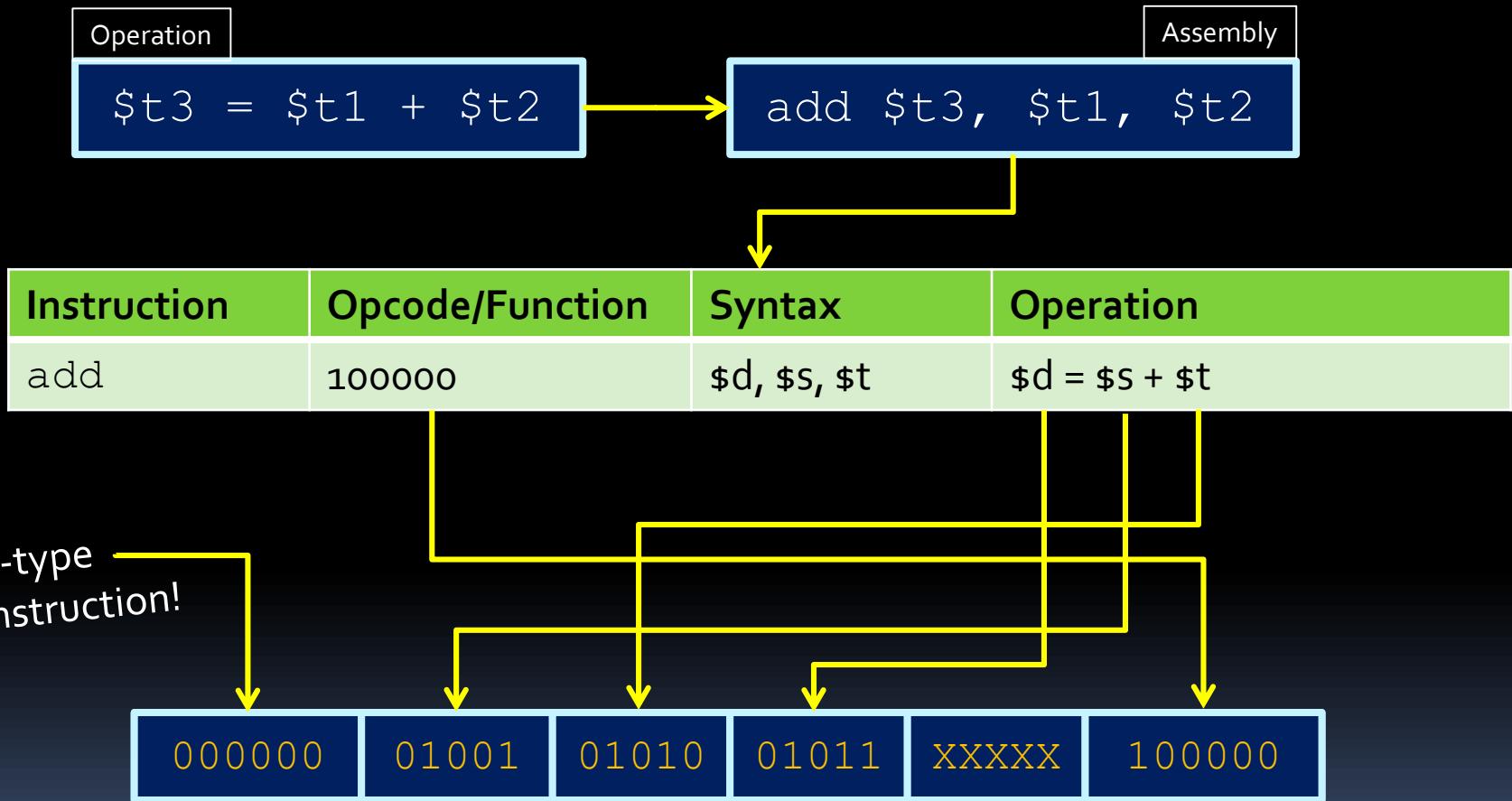
```
00000000 0000 0001 0001 1010 0010 0001 0004 012b  
0000010 0000 0016 0000 0028 0000 0010 0000 0020  
0000020 0000 0001 0004 0000 0000 0000 0000 0000  
0000030 0000 0000 0000 0010 0000 0000 0000 0204  
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9  
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc  
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857  
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888  
0000080 8888 8888 8888 8888 288e be88 8888 8888  
0000090 3b83 5788 8888 8888 7667 778e 8828 8888  
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188  
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988  
00000c0 8a18 880c e841 c988 b328 6871 688e 958b  
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec  
00000e0 3d86 dcbb 5cbb 8888 8888 8888 8888 8888  
00000f0 8888 8888 8888 8888 8888 8888 8888 0000  
0000100 0000 0000 0000 0000 0000 0000 0000 0000  
*  
0000130 0000 0000 0000 0000 0000 0000 0000 0000  
000013e  
0000J36  
0000J30 0000 0000 0000 0000 0000 0000 0000  
*  
0000J00 0000 0000 0000 0000 0000 0000 0000 0000
```

Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
add	100000	\$d, \$s, \$t	\$d = \$s + \$t
addu	100001	\$d, \$s, \$t	\$d = \$s + \$t
addi	001000	\$t, \$s, i	\$t = \$s + SE(i)
addiu	001001	\$t, \$s, i	\$t = \$s + SE(i)
div	011010	\$s, \$t	lo = \$s / \$t; hi = \$s % \$t
divu	011011	\$s, \$t	lo = \$s / \$t; hi = \$s % \$t
mult	011000	\$s, \$t	hi:lo = \$s * \$t
multu	011001	\$s, \$t	hi:lo = \$s * \$t
sub	100010	\$d, \$s, \$t	\$d = \$s - \$t
subu	100011	\$d, \$s, \$t	\$d = \$s - \$t

Note: “hi” and “lo” refer to the high and low bits referred to in the register slide.
“SE” = “sign extend”.

Assembly → Machine Code



R-type vs I-type arithmetic

R-Type

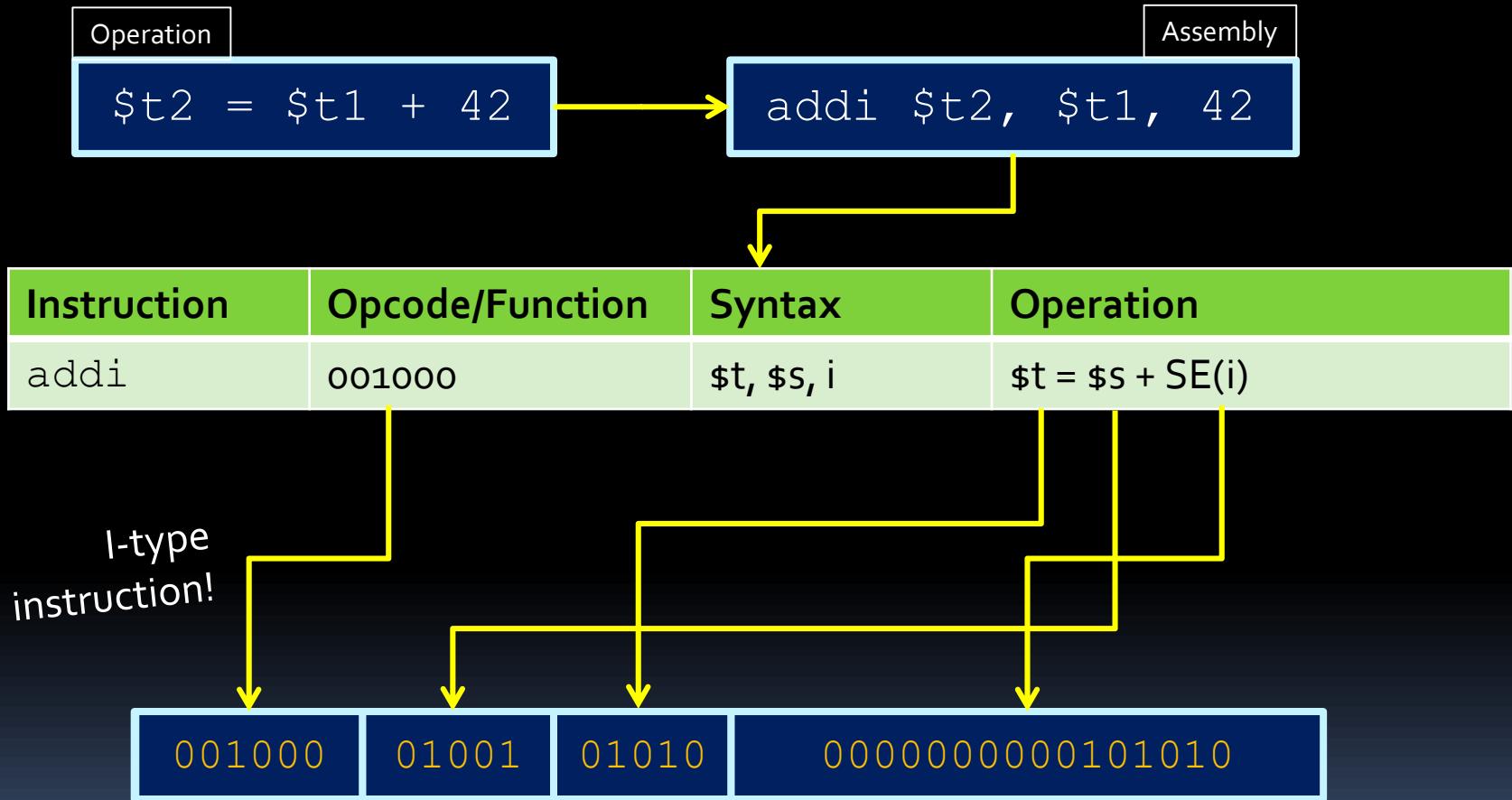
- add, addu
- div, divu
- mult, multu
- sub, subu

I-Type

- addi
- addiu

- In general, some instructions are R-type (meaning all operands are registers) and some are I-type (meaning they use an immediate/constant value in their operation).
- Can you recognize which of the following are R-type and I-type instructions?

Assembly → Machine Code II



Logical instructions

Instruction	Opcode/Function	Syntax	Operation
and	100100	\$d, \$s, \$t	\$d = \$s & \$t
andi	001100	\$t, \$s, i	\$t = \$s & ZE(i)
nor	100111	\$d, \$s, \$t	\$d = ~(\$s \$t)
or	100101	\$d, \$s, \$t	\$d = \$s \$t
ori	001101	\$t, \$s, i	\$t = \$s ZE(i)
xor	100110	\$d, \$s, \$t	\$d = \$s ^ \$t
xori	001110	\$t, \$s, i	\$t = \$s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

Shift instructions

Instruction	Opcode/Function	Syntax	Operation
sll	000000	\$d, \$t, a	$\$d = \$t \ll a$
sllv	000100	\$d, \$t, \$s	$\$d = \$t \ll \$s$
sra	000011	\$d, \$t, a	$\$d = \$t \gg a$
sraw	000111	\$d, \$t, \$s	$\$d = \$t \gg \$s$
srl	000010	\$d, \$t, a	$\$d = \$t \ggg a$
srlv	000110	\$d, \$t, \$s	$\$d = \$t \ggg \$s$

Note: `srl` = “shift right logical”, and `sra` = “shift right arithmetic”.
The “v” denotes a variable number of bits, specified by `$s`.
`a` is a **shift amount**, and is stored in `shamt` when encoding
the R-type machine code instructions.

Data movement instructions

Instruction	Opcode/Function	Syntax	Operation
mfhi	010000	\$d	\$d = hi
mflo	010010	\$d	\$d = lo
mthi	010001	\$s	hi = \$s
mtlo	010011	\$s	lo = \$s

- These are **R-type** instructions for operating on the HI and LO registers described earlier.

ALU instructions

- Note that for ALU instruction, most are R-type instructions.
 - The six-digit codes in the tables are therefore the function codes (opcodes are 000000).
 - Exceptions are the I-type instructions (addi, andi, ori, etc.)
- Not all R-type instructions have an I-type equivalent.
 - RISC architectures dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations.
 - Example: addi + div → divi

Example program

- Fibonacci sequence:
 - How would you convert this into assembly?
 - (ignoring function arguments, return call for now)

```
int fib(void) {  
    int n = 10;  
    int f1 = 1, f2 = -1;  
  
    while (n != 0) {  
        f1 = f1 + f2;  
        f2 = f1 - f2;  
        n = n - 1;  
    }  
    return f1;  
}
```

Assembly code

- Fibonacci sequence in assembly

```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
#
FIB:   addi $t3, $zero, 10          # initialize n=10
        addi $t4, $zero, 1           # initialize f1=1
        addi $t5, $zero, -1         # initialize f2=-1
LOOP:  beq $t3, $zero, END          # done loop if n==0
        add $t4, $t4, $t5          # f1 = f1 + f2
        sub $t5, $t4, $t5          # f2 = f1 - f2
        addi $t3, $t3, -1          # n = n - 1
        j LOOP                   # repeat until done
END:   sb $t4, 0($sp)              # store result
```

```
int n = 10;
int f1 = 1, f2 = -1;
while (n != 0) {
    f1 = f1 + f2;
    f2 = f1 - f2;
    n = n - 1;
}
return f1;
```

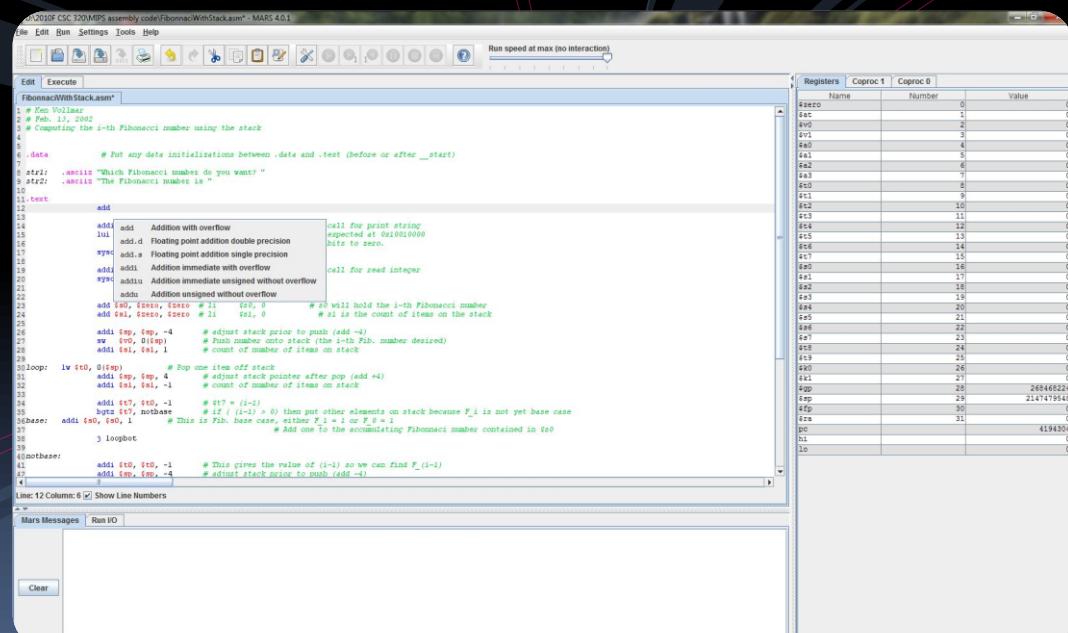
Making an assembly program

- Assembly language programs typically have structure similar to simple Python or C programs:
 - They set aside registers to store data.
 - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose!
 - More on this later ☺

Simulating MIPS

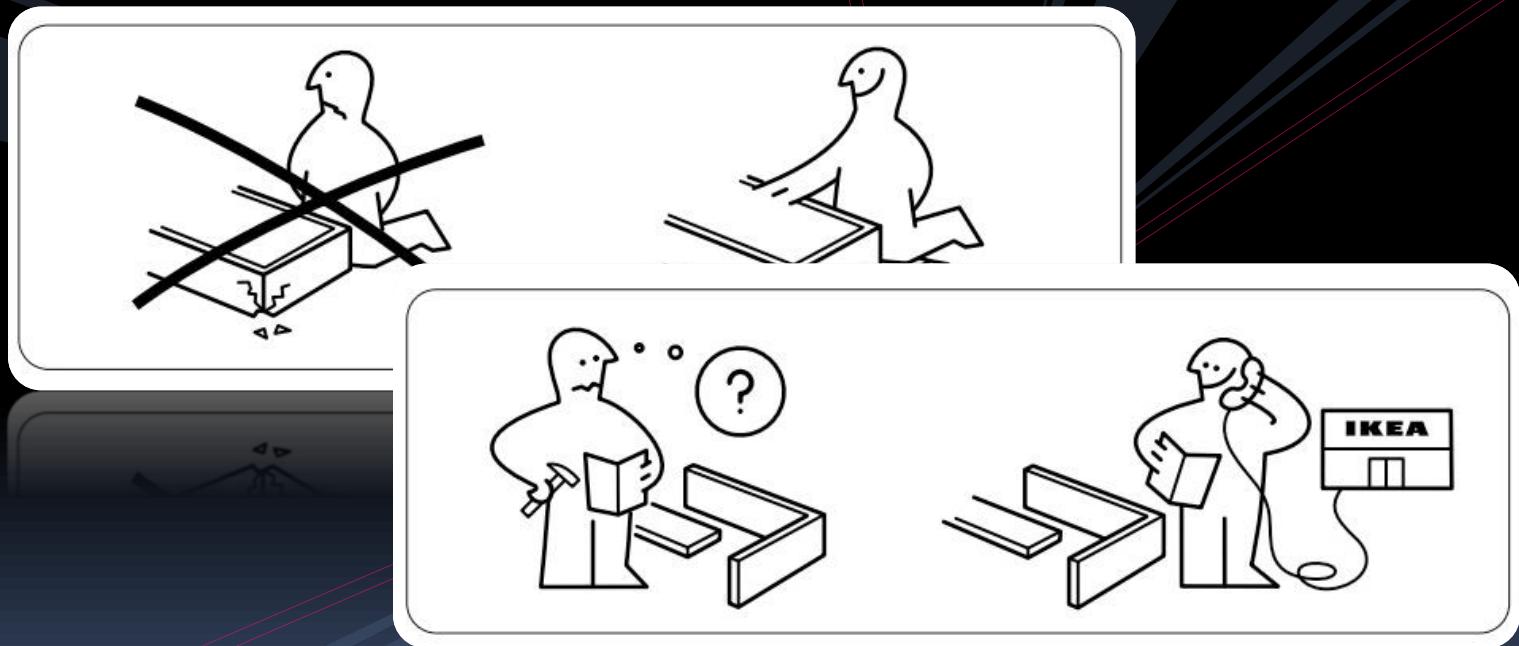
aka: MARS

- Link to download:
<http://courses.missouristate.edu/KenVollmar/mars/>
- Tutorial links available on Quercus!



The screenshot shows the MARS 4.0.1 assembly simulator interface. The main window displays an assembly code listing for "FibonacciWithStack.asm". The code implements a function to compute the i -th Fibonacci number using a stack. It includes data declarations for strings, a .text section with assembly instructions like add, add.d, and add.s, and a .loop section with conditional branches and loops. The Registers window on the right shows the state of the processor registers, including \$zero through \$t0, \$a0, \$v0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7, \$t8, \$t9, \$t10, \$t11, \$t12, \$t13, \$t14, \$t15, \$t16, \$t17, \$t18, \$t19, \$t20, \$t21, \$t22, \$t23, \$t24, \$t25, \$t26, \$t27, \$t28, \$t29, \$t30, \$t31, \$t32, \$t33, \$t34, \$t35, \$t36, \$t37, \$t38, \$t39, \$t40, \$t41, \$t42, \$t43, \$t44, \$t45, \$t46, \$t47, \$t48, \$t49, \$t50, \$t51, \$t52, \$t53, \$t54, \$t55, \$t56, \$t57, \$t58, \$t59, \$t60, \$t61, \$t62, \$t63, \$t64, \$t65, \$t66, \$t67, \$t68, \$t69, \$t70, \$t71, \$t72, \$t73, \$t74, \$t75, \$t76, \$t77, \$t78, \$t79, \$t80, \$t81, \$t82, \$t83, \$t84, \$t85, \$t86, \$t87, \$t88, \$t89, \$t90, \$t91, \$t92, \$t93, \$t94, \$t95, \$t96, \$t97, \$t98, \$t99, \$t100, \$t101, \$t102, \$t103, \$t104, \$t105, \$t106, \$t107, \$t108, \$t109, \$t110, \$t111, \$t112, \$t113, \$t114, \$t115, \$t116, \$t117, \$t118, \$t119, \$t120, \$t121, \$t122, \$t123, \$t124, \$t125, \$t126, \$t127, \$t128, \$t129, \$t130, \$t131, \$t132, \$t133, \$t134, \$t135, \$t136, \$t137, \$t138, \$t139, \$t140, \$t141, \$t142, \$t143, \$t144, \$t145, \$t146, \$t147, \$t148, \$t149, \$t150, \$t151, \$t152, \$t153, \$t154, \$t155, \$t156, \$t157, \$t158, \$t159, \$t160, \$t161, \$t162, \$t163, \$t164, \$t165, \$t166, \$t167, \$t168, \$t169, \$t170, \$t171, \$t172, \$t173, \$t174, \$t175, \$t176, \$t177, \$t178, \$t179, \$t180, \$t181, \$t182, \$t183, \$t184, \$t185, \$t186, \$t187, \$t188, \$t189, \$t190, \$t191, \$t192, \$t193, \$t194, \$t195, \$t196, \$t197, \$t198, \$t199, \$t200, \$t201, \$t202, \$t203, \$t204, \$t205, \$t206, \$t207, \$t208, \$t209, \$t210, \$t211, \$t212, \$t213, \$t214, \$t215, \$t216, \$t217, \$t218, \$t219, \$t220, \$t221, \$t222, \$t223, \$t224, \$t225, \$t226, \$t227, \$t228, \$t229, \$t230, \$t231, \$t232, \$t233, \$t234, \$t235, \$t236, \$t237, \$t238, \$t239, \$t240, \$t241, \$t242, \$t243, \$t244, \$t245, \$t246, \$t247, \$t248, \$t249, \$t250, \$t251, \$t252, \$t253, \$t254, \$t255, \$t256, \$t257, \$t258, \$t259, \$t260, \$t261, \$t262, \$t263, \$t264, \$t265, \$t266, \$t267, \$t268, \$t269, \$t270, \$t271, \$t272, \$t273, \$t274, \$t275, \$t276, \$t277, \$t278, \$t279, \$t280, \$t281, \$t282, \$t283, \$t284, \$t285, \$t286, \$t287, \$t288, \$t289, \$t290, \$t291, \$t292, \$t293, \$t294, \$t295, \$t296, \$t297, \$t298, \$t299, \$t299, \$t300, \$t301, \$t302, \$t303, \$t304, \$t305, \$t306, \$t307, \$t308, \$t309, \$t309, \$t310, \$t311, \$t312, \$t313, \$t314, \$t315, \$t316, \$t317, \$t318, \$t319, \$t319, \$t320, \$t321, \$t322, \$t323, \$t324, \$t325, \$t326, \$t327, \$t328, \$t329, \$t329, \$t330, \$t331, \$t332, \$t333, \$t334, \$t335, \$t336, \$t337, \$t338, \$t339, \$t339, \$t340, \$t341, \$t342, \$t343, \$t344, \$t345, \$t346, \$t347, \$t348, \$t349, \$t349, \$t350, \$t351, \$t352, \$t353, \$t354, \$t355, \$t356, \$t357, \$t358, \$t359, \$t359, \$t360, \$t361, \$t362, \$t363, \$t364, \$t365, \$t366, \$t367, \$t368, \$t369, \$t369, \$t370, \$t371, \$t372, \$t373, \$t374, \$t375, \$t376, \$t377, \$t378, \$t379, \$t379, \$t380, \$t381, \$t382, \$t383, \$t384, \$t385, \$t386, \$t387, \$t388, \$t389, \$t389, \$t390, \$t391, \$t392, \$t393, \$t394, \$t395, \$t396, \$t397, \$t398, \$t399, \$t399, \$t400, \$t401, \$t402, \$t403, \$t404, \$t405, \$t406, \$t407, \$t408, \$t409, \$t409, \$t410, \$t411, \$t412, \$t413, \$t414, \$t415, \$t416, \$t417, \$t418, \$t419, \$t419, \$t420, \$t421, \$t422, \$t423, \$t424, \$t425, \$t426, \$t427, \$t428, \$t429, \$t429, \$t430, \$t431, \$t432, \$t433, \$t434, \$t435, \$t436, \$t437, \$t438, \$t439, \$t439, \$t440, \$t441, \$t442, \$t443, \$t444, \$t445, \$t446, \$t447, \$t448, \$t449, \$t449, \$t450, \$t451, \$t452, \$t453, \$t454, \$t455, \$t456, \$t457, \$t458, \$t459, \$t459, \$t460, \$t461, \$t462, \$t463, \$t464, \$t465, \$t466, \$t467, \$t468, \$t469, \$t469, \$t470, \$t471, \$t472, \$t473, \$t474, \$t475, \$t476, \$t477, \$t478, \$t479, \$t479, \$t480, \$t481, \$t482, \$t483, \$t484, \$t485, \$t486, \$t487, \$t488, \$t489, \$t489, \$t490, \$t491, \$t492, \$t493, \$t494, \$t495, \$t496, \$t497, \$t498, \$t499, \$t499, \$t500, \$t501, \$t502, \$t503, \$t504, \$t505, \$t506, \$t507, \$t508, \$t509, \$t509, \$t510, \$t511, \$t512, \$t513, \$t514, \$t515, \$t516, \$t517, \$t518, \$t519, \$t519, \$t520, \$t521, \$t522, \$t523, \$t524, \$t525, \$t526, \$t527, \$t528, \$t529, \$t529, \$t530, \$t531, \$t532, \$t533, \$t534, \$t535, \$t536, \$t537, \$t538, \$t539, \$t539, \$t540, \$t541, \$t542, \$t543, \$t544, \$t545, \$t546, \$t547, \$t548, \$t549, \$t549, \$t550, \$t551, \$t552, \$t553, \$t554, \$t555, \$t556, \$t557, \$t558, \$t559, \$t559, \$t560, \$t561, \$t562, \$t563, \$t564, \$t565, \$t566, \$t567, \$t568, \$t569, \$t569, \$t570, \$t571, \$t572, \$t573, \$t574, \$t575, \$t576, \$t577, \$t578, \$t579, \$t579, \$t580, \$t581, \$t582, \$t583, \$t584, \$t585, \$t586, \$t587, \$t588, \$t589, \$t589, \$t590, \$t591, \$t592, \$t593, \$t594, \$t595, \$t596, \$t597, \$t598, \$t599, \$t599, \$t600, \$t601, \$t602, \$t603, \$t604, \$t605, \$t606, \$t607, \$t608, \$t609, \$t609, \$t610, \$t611, \$t612, \$t613, \$t614, \$t615, \$t616, \$t617, \$t618, \$t619, \$t619, \$t620, \$t621, \$t622, \$t623, \$t624, \$t625, \$t626, \$t627, \$t628, \$t629, \$t629, \$t630, \$t631, \$t632, \$t633, \$t634, \$t635, \$t636, \$t637, \$t638, \$t639, \$t639, \$t640, \$t641, \$t642, \$t643, \$t644, \$t645, \$t646, \$t647, \$t648, \$t649, \$t649, \$t650, \$t651, \$t652, \$t653, \$t654, \$t655, \$t656, \$t657, \$t658, \$t659, \$t659, \$t660, \$t661, \$t662, \$t663, \$t664, \$t665, \$t666, \$t667, \$t668, \$t669, \$t669, \$t670, \$t671, \$t672, \$t673, \$t674, \$t675, \$t676, \$t677, \$t678, \$t679, \$t679, \$t680, \$t681, \$t682, \$t683, \$t684, \$t685, \$t686, \$t687, \$t688, \$t689, \$t689, \$t690, \$t691, \$t692, \$t693, \$t694, \$t695, \$t696, \$t697, \$t698, \$t699, \$t699, \$t700, \$t701, \$t702, \$t703, \$t704, \$t705, \$t706, \$t707, \$t708, \$t709, \$t709, \$t710, \$t711, \$t712, \$t713, \$t714, \$t715, \$t716, \$t717, \$t718, \$t719, \$t719, \$t720, \$t721, \$t722, \$t723, \$t724, \$t725, \$t726, \$t727, \$t728, \$t729, \$t729, \$t730, \$t731, \$t732, \$t733, \$t734, \$t735, \$t736, \$t737, \$t738, \$t739, \$t739, \$t740, \$t741, \$t742, \$t743, \$t744, \$t745, \$t746, \$t747, \$t748, \$t749, \$t749, \$t750, \$t751, \$t752, \$t753, \$t754, \$t755, \$t756, \$t757, \$t758, \$t759, \$t759, \$t760, \$t761, \$t762, \$t763, \$t764, \$t765, \$t766, \$t767, \$t768, \$t769, \$t769, \$t770, \$t771, \$t772, \$t773, \$t774, \$t775, \$t776, \$t777, \$t778, \$t779, \$t779, \$t780, \$t781, \$t782, \$t783, \$t784, \$t785, \$t786, \$t787, \$t788, \$t789, \$t789, \$t790, \$t791, \$t792, \$t793, \$t794, \$t795, \$t796, \$t797, \$t798, \$t799, \$t799, \$t800, \$t801, \$t802, \$t803, \$t804, \$t805, \$t806, \$t807, \$t808, \$t809, \$t809, \$t810, \$t811, \$t812, \$t813, \$t814, \$t815, \$t816, \$t817, \$t818, \$t819, \$t819, \$t820, \$t821, \$t822, \$t823, \$t824, \$t825, \$t826, \$t827, \$t828, \$t829, \$t829, \$t830, \$t831, \$t832, \$t833, \$t834, \$t835, \$t836, \$t837, \$t838, \$t839, \$t839, \$t840, \$t841, \$t842, \$t843, \$t844, \$t845, \$t846, \$t847, \$t848, \$t849, \$t849, \$t850, \$t851, \$t852, \$t853, \$t854, \$t855, \$t856, \$t857, \$t858, \$t859, \$t859, \$t860, \$t861, \$t862, \$t863, \$t864, \$t865, \$t866, \$t867, \$t868, \$t869, \$t869, \$t870, \$t871, \$t872, \$t873, \$t874, \$t875, \$t876, \$t877, \$t878, \$t879, \$t879, \$t880, \$t881, \$t882, \$t883, \$t884, \$t885, \$t886, \$t887, \$t888, \$t889, \$t889, \$t890, \$t891, \$t892, \$t893, \$t894, \$t895, \$t896, \$t897, \$t898, \$t899, \$t899, \$t900, \$t901, \$t902, \$t903, \$t904, \$t905, \$t906, \$t907, \$t908, \$t909, \$t909, \$t910, \$t911, \$t912, \$t913, \$t914, \$t915, \$t916, \$t917, \$t918, \$t919, \$t919, \$t920, \$t921, \$t922, \$t923, \$t924, \$t925, \$t926, \$t927, \$t928, \$t929, \$t929, \$t930, \$t931, \$t932, \$t933, \$t934, \$t935, \$t936, \$t937, \$t938, \$t939, \$t939, \$t940, \$t941, \$t942, \$t943, \$t944, \$t945, \$t946, \$t947, \$t948, \$t949, \$t949, \$t950, \$t951, \$t952, \$t953, \$t954, \$t955, \$t956, \$t957, \$t958, \$t959, \$t959, \$t960, \$t961, \$t962, \$t963, \$t964, \$t965, \$t966, \$t967, \$t968, \$t969, \$t969, \$t970, \$t971, \$t972, \$t973, \$t974, \$t975, \$t976, \$t977, \$t978, \$t979, \$t979, \$t980, \$t981, \$t982, \$t983, \$t984, \$t985, \$t986, \$t987, \$t988, \$t989, \$t989, \$t990, \$t991, \$t992, \$t993, \$t994, \$t995, \$t996, \$t997, \$t998, \$t999, \$t999, \$t1000, \$t1001, \$t1002, \$t1003, \$t1004, \$t1005, \$t1006, \$t1007, \$t1008, \$t1009, \$t1009, \$t1010, \$t1011, \$t1012, \$t1013, \$t1014, \$t1015, \$t1016, \$t1017, \$t1018, \$t1019, \$t1019, \$t1020, \$t1021, \$t1022, \$t1023, \$t1024, \$t1025, \$t1026, \$t1027, \$t1028, \$t1029, \$t1029, \$t1030, \$t1031, \$t1032, \$t1033, \$t1034, \$t1035, \$t1036, \$t1037, \$t1038, \$t1039, \$t1039, \$t1040, \$t1041, \$t1042, \$t1043, \$t1044, \$t1045, \$t1046, \$t1047, \$t1048, \$t1049, \$t1049, \$t1050, \$t1051, \$t1052, \$t1053, \$t1054, \$t1055, \$t1056, \$t1057, \$t1058, \$t1059, \$t1059, \$t1060, \$t1061, \$t1062, \$t1063, \$t1064, \$t1065, \$t1066, \$t1067, \$t1068, \$t1069, \$t1069, \$t1070, \$t1071, \$t1072, \$t1073, \$t1074, \$t1075, \$t1076, \$t1077, \$t1078, \$t1079, \$t1079, \$t1080, \$t1081, \$t1082, \$t1083, \$t1084, \$t1085, \$t1086, \$t1087, \$t1088, \$t1089, \$t1089, \$t1090, \$t1091, \$t1092, \$t1093, \$t1094, \$t1095, \$t1096, \$t1097, \$t1098, \$t1099, \$t1099, \$t1100, \$t1101, \$t1102, \$t1103, \$t1104, \$t1105, \$t1106, \$t1107, \$t1108, \$t1109, \$t1109, \$t1110, \$t1111, \$t1112, \$t1113, \$t1114, \$t1115, \$t1116, \$t1117, \$t1118, \$t1119, \$t1119, \$t1120, \$t1121, \$t1122, \$t1123, \$t1124, \$t1125, \$t1126, \$t1127, \$t1128, \$t1129, \$t1129, \$t1130, \$t1131, \$t1132, \$t1133, \$t1134, \$t1135, \$t1136, \$t1137, \$t1138, \$t1139, \$t1139, \$t1140, \$t1141, \$t1142, \$t1143, \$t1144, \$t1145, \$t1146, \$t1147, \$t1148, \$t1149, \$t1149, \$t1150, \$t1151, \$t1152, \$t1153, \$t1154, \$t1155, \$t1156, \$t1157, \$t1158, \$t1159, \$t1159, \$t1160, \$t1161, \$t1162, \$t1163, \$t1164, \$t1165, \$t1166, \$t1167, \$t1168, \$t1169, \$t1169, \$t1170, \$t1171, \$t1172, \$t1173, \$t1174, \$t1175, \$t1176, \$t1177, \$t1178, \$t1179, \$t1179, \$t1180, \$t1181, \$t1182, \$t1183, \$t1184, \$t1185, \$t1186, \$t1187, \$t1188, \$t1189, \$t1189, \$t1190, \$t1191, \$t1192, \$t1193, \$t1194, \$t1195, \$t1196, \$t1197, \$t1198, \$t1199, \$t1199, \$t1200, \$t1201, \$t1202, \$t1203, \$t1204, \$t1205, \$t1206, \$t1207, \$t1208, \$t1209, \$t1209, \$t1210, \$t1211, \$t1212, \$t1213, \$t1214, \$t1215, \$t1216, \$t1217, \$t1218, \$t1219, \$t1219, \$t1220, \$t1221, \$t1222, \$t1223, \$t1224, \$t1225, \$t1226, \$t1227, \$t1228, \$t1229, \$t1229, \$t1230, \$t1231, \$t1232, \$t1233, \$t1234, \$t1235, \$t1236, \$t1237, \$t1238, \$t1239, \$t1239, \$t1240, \$t1241, \$t1242, \$t1243, \$t1244, \$t1245, \$t1246, \$t1247, \$t1248, \$t1249, \$t1249, \$t1250, \$t1251, \$t1252, \$t1253, \$t1254, \$t1255, \$t1256, \$t1257, \$t1258, \$t1259, \$t1259, \$t1260, \$t1261, \$t1262, \$t1263, \$t1264, \$t1265, \$t1266, \$t1267, \$t1268, \$t1269, \$t1269, \$t1270, \$t1271, \$t1272, \$t1273, \$t1274, \$t1275, \$t1276, \$t1277, \$t1278, \$t1279, \$t1279, \$t1280, \$t1281, \$t1282, \$t1283, \$t1284, \$t1285, \$t1286, \$t1287, \$t1288, \$t1289, \$t1289, \$t1290, \$t1291, \$t1292, \$t1293, \$t1294, \$t1295, \$t1296, \$t1297, \$t1298, \$t1299, \$t1299, \$t1300, \$t1301, \$t1302, \$t1303, \$t1304, \$t1305, \$t1306, \$t1307, \$t1308, \$t1309, \$t1309, \$t1310, \$t1311, \$t1312, \$t1313, \$t1314, \$t1315, \$t1316, \$t1317, \$t1318, \$t1319, \$t1319, \$t1320, \$t1321, \$t1322, \$t1323, \$t1324, \$t1325, \$t1326, \$t1327, \$t1328, \$t1329, \$t1329, \$t1330, \$t1331, \$t1332, \$t1333, \$t1334, \$t1335, \$t1336, \$t1337, \$t1338, \$t1339, \$t1339, \$t1340, \$t1341, \$t1342, \$t1343, \$t1344, \$t1345, \$t1346, \$t1347, \$t1348, \$t1349, \$t1349, \$t1350, \$t1351, \$t1352, \$t1353, \$t1354, \$t1355, \$t1356, \$t1357, \$t1358, \$t1359, \$t1359, \$t1360, \$t1361, \$t1362, \$t1363, \$t1364, \$t1365, \$t1366, \$t1367, \$t1368, \$t1369, \$t1369, \$t1370, \$t1371, \$t1372, \$t1373, \$t1374, \$t1375, \$t1376, \$t1377, \$t1378, \$t1379, \$t1379, \$t1380, \$t1381, \$t1382, \$t1383, \$t1384, \$t1385, \$t1386, \$t1387, \$t1388, \$t1389, \$t1389, \$t1390, \$t1391, \$t1392, \$t1393, \$t1394, \$t1395, \$t1396, \$t1397, \$t1398, \$t1399, \$t1399, \$t1400, \$t1401, \$t1402, \$t1403, \$t1404, \$t1405, \$t1406, \$t1407, \$t1408, \$t1409, \$t1409, \$t1410, \$t1411, \$t1412, \$t1413, \$t1414, \$t1415, \$t1416, \$t1417, \$t1418, \$t1419, \$t1419, \$t1420, \$t1421, \$t1422, \$t1423, \$t1424, \$t1425, \$t1426, \$t1427, \$t1428, \$t1429, \$t1429, \$t1430, \$t1431, \$t1432, \$t1433, \$t1434, \$t1435, \$t1436, \$t1437, \$t1438, \$t1439, \$t1439, \$t1440, \$t1441, \$t1442, \$t1443, \$t1444, \$t1445, \$t1446, \$t1447, \$t1448, \$t1449, \$t1449, \$t1450, \$t1451, \$t1452, \$t1453, \$t1454, \$t1455, \$t1456, \$t1457, \$t1458, \$t1459, \$t1459, \$t1460, \$t1461, \$t1462, \$t1463, \$t1464, \$t1465, \$t1466, \$t1467, \$t1468, \$t1469, \$t1469, \$t1470, \$t1471, \$t1472, \$t1473, \$t1474, \$t1475, \$t1476, \$t1477, \$t1478, \$t1479, \$t1479, \$t1480, \$t1481, \$t1482, \$t1483, \$t1484, \$t1485, \$t1486, \$t1487, \$t1488, \$t1489, \$t1489, \$t1490, \$t1491, \$t1492, \$t1493, \$t1494, \$t1495, \$t1496, \$t1497, \$t1498, \$t1499, \$t1499, \$t1500, \$t1501, \$t1502, \$t1503, \$t1504, \$t1505, \$t1506, \$t1507, \$t1508, \$t1509, \$t1509, \$t1510, \$t1511, \$t1512, \$t1513, \$t1514, \$t1515, \$t1516, \$t1517, \$t1518, \$t1519, \$t1519, \$t1520, \$t1521, \$t1522, \$t1523, \$t1524, \$t1525, \$t1526, \$t1527, \$t1528, \$t1529, \$t1529, \$t1530, \$t1531, \$t1532, \$t1533, \$t1534, \$t1535, \$t1536, \$t1537, \$t1538, \$t1539, \$t1539, \$t1540, \$t1541, \$t1542, \$t1543, \$t1544, \$t1545, \$t1546, \$t1547, \$t1548, \$t1549, \$t1549, \$t1550, \$t1551, \$t1552, \$t1553, \$t1554, \$t1555, \$t1556, \$t1557, \$t1558, \$t1559, \$t1559, \$t1560, \$t1561, \$t1562, \$t1563, \$t1564, \$t1565, \$t1566, \$t1567, \$t1568, \$t1569, \$t1569, \$t1570, \$t1571, \$t1572, \$t1573, \$t1574, \$t1575, \$t1576, \$t1577, \$t1578, \$t1579, \$t1579, \$t1580, \$t1581, \$t1582, \$t1583, \$t1584, \$t1585, \$t1586, \$t1587, \$t1588, \$t1589, \$t1589, \$t1590, \$t1591, \$t1592, \$t1593, \$t1594, \$t1595, \$t1596, \$t1597, \$t1598, \$t1599, \$t1599, \$t1600, \$t1601, \$t1602, \$t1603, \$t1604, \$t1605, \$t1606, \$t1607, \$t1608, \$t1609, \$t1609, \$t1610, \$t1611, \$t1612, \$t1613, \$t1614, \$t1615, \$t1616, \$t1617, \$t1618, \$t1619, \$t1619, \$t1620, \$t1621, \$t1622, \$t1623, \$t1624, \$t1625, \$t1626, \$t1627, \$t1628, \$t1629, \$t1629, \$t1630, \$t1631, \$t1632, \$t1633, \$t1634, \$t1635, \$t1636, \$t1637, \$t1638, \$t1639, \$t1639, \$t1640, \$t1641, \$t1642, \$t1643, \$t1644, \$t1645, \$t1646, \$t1647, \$t1648, \$t1649, \$t1649, \$t1650, \$t1651, \$t1652, \$t1653, \$t1654, \$t1655, \$t1656, \$t1657, \$t1658, \$t1659, \$t1659, \$t1660, \$t1661, \$t1662, \$t1663, \$t1664, \$t1665, \$t1666, \$t1667, \$t1668, \$t1669, \$t1669, \$t1670, \$t1671, \$t1672, \$t1673, \$t1674, \$t1675, \$t1676, \$t1677, \$t1678, \$t1679, \$t1679, \$t1680, \$t1681, \$t1682, \$t1683, \$t1684, \$t1685, \$t1686, \$t1687, \$t1688, \$t1689, \$t1689, \$t1690, \$t1691, \$t1692, \$t1693, \$t1694, \$t1695, \$t1696, \$t1697, \$t1698, \$t1699, \$t1699, \$t1700, \$t1701, \$t1702, \$t1703, \$t1704, \$t1705, \$t1706, \$t1707, \$t1708, \$t1709, \$t1709, \$t1710, \$t1711, \$t1712, \$t1713, \$t1714, \$t1715, \$t1716, \$t1717, \$t1718, \$t1719, \$t1719, \$t1720

More instructions!



Control flow in assembly

- Not all programs follow a linear set of instructions.
 - Some operations require the code to **branch** to one section of code or another (**if/else**).
 - Some require the code to **jump** back and repeat a section of code again (**for/while**).
- For this, we have **labels** on the left-hand side that indicate the points that the program flow might need to jump to.
 - References to these points in the assembly code are **resolved at compile time** to offset values for the program counter.

Jump instructions

Instruction	Opcode/Function	Syntax	Operation
j	000010	label	$\text{pc} = (\text{pc} \& 0xF0000000) (\text{i} \ll 2)$
jal	000011	label	$\$31 = \text{pc} + 4;$ $\text{pc} = (\text{pc} \& 0xF0000000) (\text{i} \ll 2)$
jalr	001001	\$s	$\$31 = \text{pc} + 4;$ $\text{pc} = \$s$
jr	001000	\$s	$\text{pc} = \$s$

- jal = “jump and link”.
 - Register \$31 (aka \$ra) stores the address that’s used when returning from a subroutine (i.e. the *next* instruction to run).
- Note: jr and jalr are jumps, but **not J-type** instructions.

jr and jalr (jump to register)

- For instructions such as the following:

jr \$ra

jalr \$t0

- The processor moves the address stored in \$ra and \$t0 into the program counter.
 - The next instruction to be fetched will be at this new address, and the program will continue from there.
- What happens in the other cases, when the destination address is stored in the instruction?

j and jal (jump to label)

- For j and jal instructions, the address is supplied by the instruction.
 - This is a potential **problem**.



- If the first 6 bits are occupied by the opcode, the remaining bits **aren't enough for a full 32-bit address!**
- How do we get around this?

Solution #1: Trailing zeros

- Since jump instructions load new addresses into the program counter, the **values** being loaded **must be divisible by 4**.
 - Therefore...the binary values of these addresses will always **end in “00”**.
 - Therefore...there's **no point in storing the last two bits** of the address in the instruction.
- **Solution:** Use the **26 bits** in the J-type instructions to store the new PC address, minus the last two zeros at the end.

Solution #2: Leading bits

- This still leaves us with a 28-bit address (26 bits from the instruction + “00” at the end).
- What should we use for **the first 4 bits**?
 - Several solutions exist, but the one that MIPS uses is to keep the **first 4 bits of the previous PC value**.
 - This is where the formula in the table comes from:

```
pc = (pc & 0xF0000000) | (i<<2)
```

Take the first four bits from the previous PC

Join the two parts together

Add “00” to the end of the instruction bits

Branch instructions

Instruction	Opcode/Function	Syntax	Operation
beq	000100	\$s, \$t, label	if (\$s == \$t) pc += i << 2
bgtz	000111	\$s, label	if (\$s > 0) pc += i << 2
blez	000110	\$s, label	if (\$s <= 0) pc += i << 2
bne	000101	\$s, \$t, label	if (\$s != \$t) pc += i << 2

- Branch operations are key when implementing `if` statements and `while` loops.
- The labels are (addresses of) memory locations, assigned to each label at compile time.

Branch instructions

- How does a branch instruction work?

```
.text

main:    beq $t0, $t1, end      # check if $t0 == $t1
          ...
          ...
          ...

end:     ...                   # if $t0 == $t1, then
          ...                   # execute these lines
```

Branch instructions

- Alternate implementation using bne:

```
.text

main:    bne $t0, $t1, end      # check if $t0 == $t1
          ...
          ...
          ...

end:     ...                   # if $t0 != $t1, then
          ...                   # execute these lines
```

- Used to produce if statement behaviour.

Branch's immediate (*i*) value



- Branch statements are I-type instructions.
- The **immediate value (*i*)** is a **16-bit offset** (i.e. a **relative address**) to add to the current instruction if the branch condition is satisfied (**not the absolute address**, like with jumps).
 - Calculated as the difference between the current PC value and the address of the instruction you're branching to.
 - Stored here as **# of instructions** (and not **# of bytes**)
 - Again, not storing the trailing "00" if it's not necessary.
 - The *i* value can be positive (if you're jumping *i* instructions **forward**) or negative (if you're jumping *i* instructions **backward**).

Calculating the i value

- The offset is computed differently, depending on the implementation (i.e. if the PC is incremented by 4 before or after the branch offset calculation).

- If relative to current-PC :

$$i = (\text{label location} - (\text{current PC})) \gg 2$$

- If relative to incremented PC:

$$i = (\text{label location} - (\text{current PC} + 4)) \gg 2$$

- For this course, we assume i is computed as:
 - $i = (\text{label} - (\text{current PC})) \gg 2$
 - Corresponds to the simulator we use for this course (MARS)
→ more on that later.

i in simulation

- Use a simple program in MARS to confirm this.

```
.text

main:    addi $t0, $zero, 1
         beq $t0, $zero, END
         addi $t1, $zero, 1
END:     addi $t3, $zero, 1
```

- What will i be for beq?
- In MARS, the 16 least significant bits of the machine code instruction are 0000000000000010.
 - END is 2 instructions down from the branch instruction.

Conditional Branch Terms

- When the branch condition is met, we say the branch is taken.
- When the branch condition is not met, we say the branch is not taken.
 - What is the next PC in this case?
 - It's the usual $\text{PC}+4$
- How far can a processor branch? Are there any constraints?

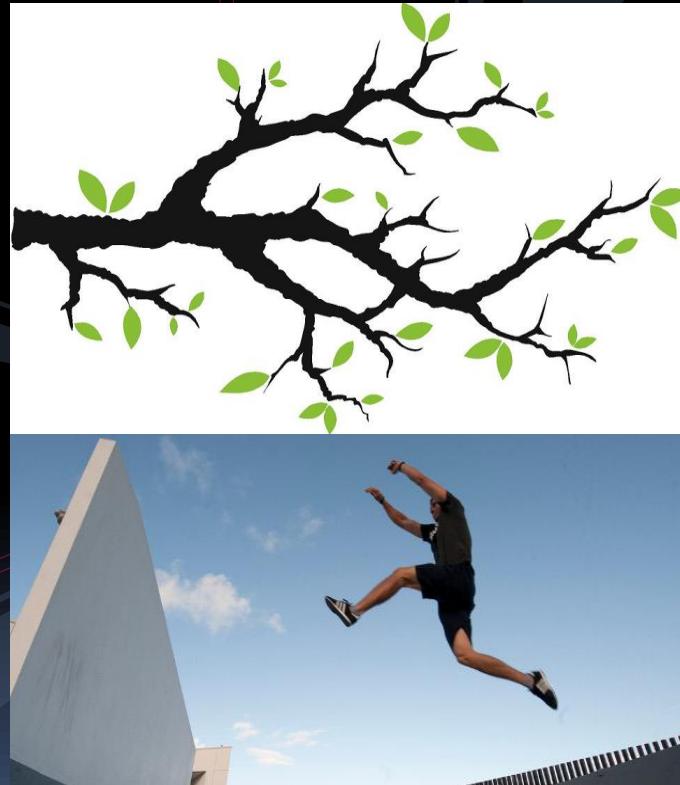
Comparison instructions

Instruction	Opcode/Function	Syntax	Operation
slt	101010	\$d, \$s, \$t	\$d = (\$s < \$t)
sltu	101001	\$d, \$s, \$t	\$d = (\$s < \$t)
slti	001010	\$t, \$s, i	\$t = (\$s < SE(i))
sltiu	001011	\$t, \$s, i	\$t = (\$s < ZE(i))

Note: Comparison operations store a 1 in the destination register if the less-than comparison is true, and stores a zero in that location otherwise. Not used too often, but useful in combination with branch instructions that only depend on one register (e.g., `bgtz`)

Using branches and jumps

if, else, while & for



If statements

- if statements test a condition and then execute lines of code if the condition is true.
 - For instance:

```
if ( i == j ) {  
    i++;  
}  
j = j + i;
```

- Testing conditions is done using either a beq instruction or a bne instruction.

Translated if statement

```
if ( i == j ) {  
    i++;  
}  
j = j + i;
```

- Use the bne instruction to skip the `i++` step and proceed straight to the `j=j+i` step:

```
# $t1 = i, $t2 = j  
main:   bne  $t1, $t2, END      # branch if (i != j)  
        addi $t1, $t1, 1        # i++  
END:    add  $t2, $t2, $t1      # j = j + i
```

if/else statements

```
if ( i == j )  
    i++;  
else  
    i--;  
j += i;
```

- Possible approach to if/else statements:
 - Test condition, and jump to if logic block whenever condition is true.
 - Otherwise, perform else logic block, and jump to first line after if logic block.

Translated if/else statements

```
# $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF      # branch if ( i == j )
          addi $t1, $t1, -1     # i--
          j END                 # jump over IF
IF:       addi $t1, $t1, 1      # i++
END:      add $t2, $t2, $t1     # j += i
```

```
if (i==j)
  i++;
else
  i--;
j += i;
```

- Or branch on the else condition first:

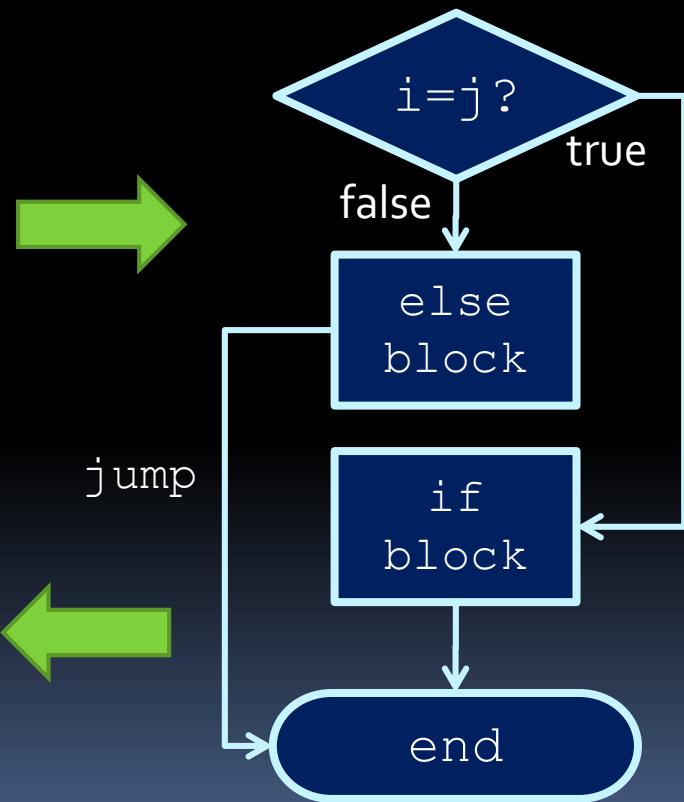
```
# $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE    # branch if ! ( i == j )
          addi $t1, $t1, 1      # i++
          j END                 # jump over ELSE
ELSE:     addi $t1, $t1, -1     # i--
END:      add $t2, $t2, $t1     # j += i
```

A trick with if statements

- Use flow charts to help you sort out the control flow of the code:

```
if ( i == j )
    i++;
else
    i--;
j += i;
```

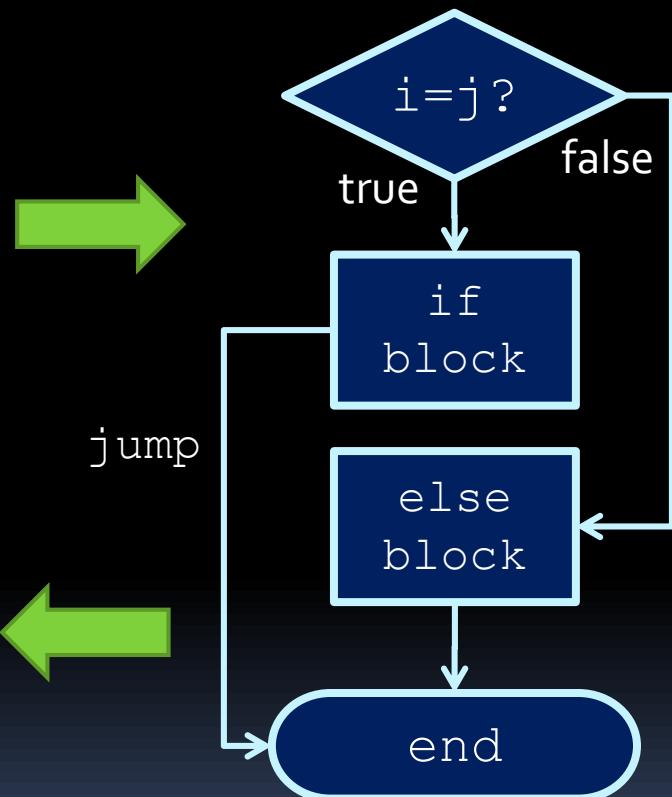
```
# $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF
          addi $t1, $t1, -1
          j END
IF:      addi $t1, $t1, 1
END:     add $t2, $t2, $t1
```



if/else statement flowcharts

```
if ( i == j )  
    i++;  
else  
    i--;  
j += i;
```

```
# $t1 = i, $t2 = j  
main:    bne  $t1, $t2, ELSE  
          addi $t1, $t1, 1  
          j END  
ELSE:  
        addi $t1, $t1, -1  
END:    add $t2, $t2, $t1
```



Multiple if conditions

```
if ( i == j || i == k )
    i++ ; // if-body
else
    i-- ; // else-body
j = i + k ;
```

- Branch statement for each condition:

```
# $t1 = i, $t2 = j, $t3 = k
main: beq $t1, $t2, IF      # cond1: branch if ( i == j )
      bne $t1, $t3, ELSE    # cond2: branch if ( i != k )
IF:   addi $t1, $t1, 1      # if (i==j||i==k) → i++
      j END                 # jump over else
ELSE: addi $t1, $t1, -1     # else-body: j--
END:  add $t2, $t1, $t3     # j = i + k
```

Multiple if conditions

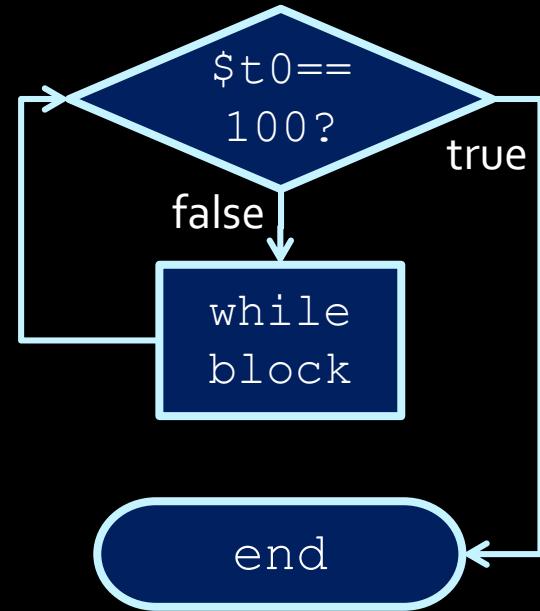
- How would this look if the condition changed?

```
if ( i == j && i == k )
    i++ ; // if-body
else
    j-- ; // else-body
j = i + k ;
```

```
# $t1 = i, $t2 = j, $t3 = k
main: bne $t1, $t2, ELSE      # cond1: branch if ( i != j )
      bne $t1, $t3, ELSE      # cond2: branch if ( i != k )
IF:   addi $t1, $t1, 1        # if (i==j||i==k) → i++
      j END                  # jump over else
ELSE: addi $t2, $t2, -1       # else-body: j--
END:  add $t2, $t1, $t3       # j = i + k
```

while loops

- Loops look similar to if statements.
 - Test if the loop condition fails.
 - If it does, branch to the end.
 - Otherwise, execute the while loop contents.
 - Make sure to update the loop condition values.
 - Jump back to the beginning.



while loops

- Example of a simple loop, in assembly:

```
main:    add $t0, $zero, $zero      # set $t0 to 0
          addi $t1, $zero, 100      # set $t1 to 100
START:   beq $t0, $t1, END        # while $t0 < $t1
          addi $t0, $t0, 1         #     $t0 = $t0 + 1
          j START                 #     jump back
END:
```

- ...which is the same as saying (in C):

```
int i = 0;
while (i < 100) {
    i++;
}
```

for loops

```
for ( <init> ; <cond> ; <update> ) {  
    <for body>  
}
```

- For loops (such as above) are usually implemented with the following structure:

```
main:      <init>  
START:     if (!<cond>) branch to END  
          <for-body>  
UPDATE:    <update>  
          jump to START  
END:
```

for loop example

```
for ( i=0, j=0 ; i<100 ; i++ ) {  
    j = j + i;  
}
```

- This translates to:

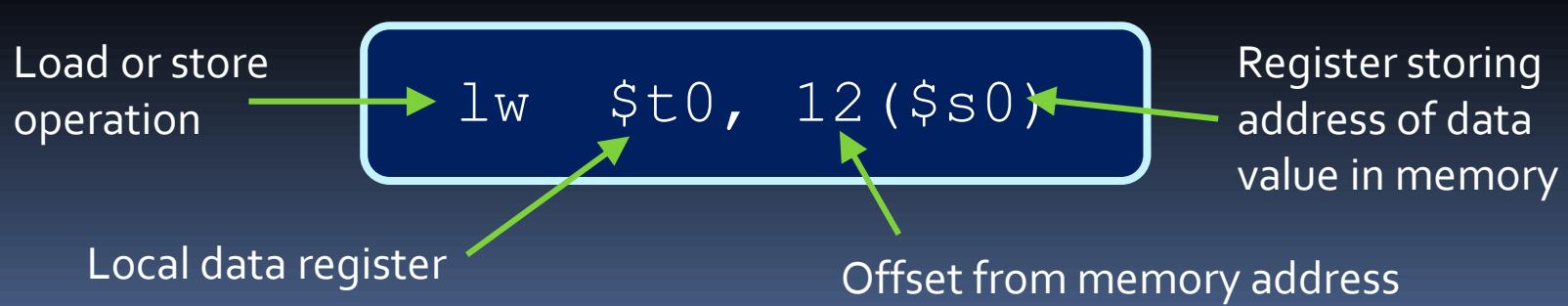
```
# $t0 = i, $t1 = j  
main:    add $t0, $zero, $zero          # set $t0 to 0  
           add $t1, $zero, $zero          # set $t1 to 0  
           addi $t9, $zero, 100          # set $t9 to 100  
START:   beq $t0, $t9, EXIT            # branch if i==100  
           add $t1, $t1, $t0            # j = j + i  
UPDATE:  addi $t0, $t0, 1              # i++  
           j START  
  
EXIT:
```

- Without the initialization and update sections, this is the same as a while loop.

Only a few more
instructions left!

Interacting with memory

- All of the previous instructions perform operations on registers and immediate values.
 - What about memory?
- All programs must **fetch values from** memory into registers, operate on them, and then **store the values back** into memory.
- Memory operations are I-type, with the form:

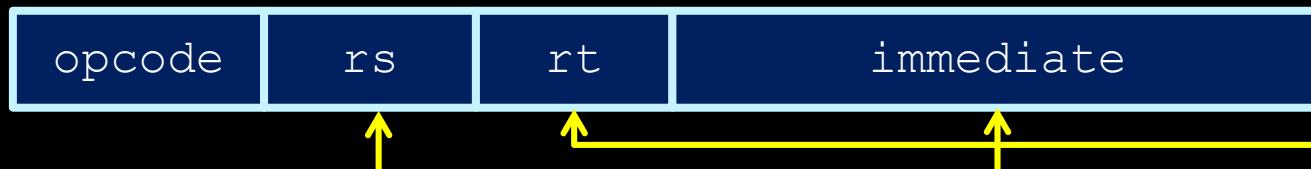


Loads vs. Stores

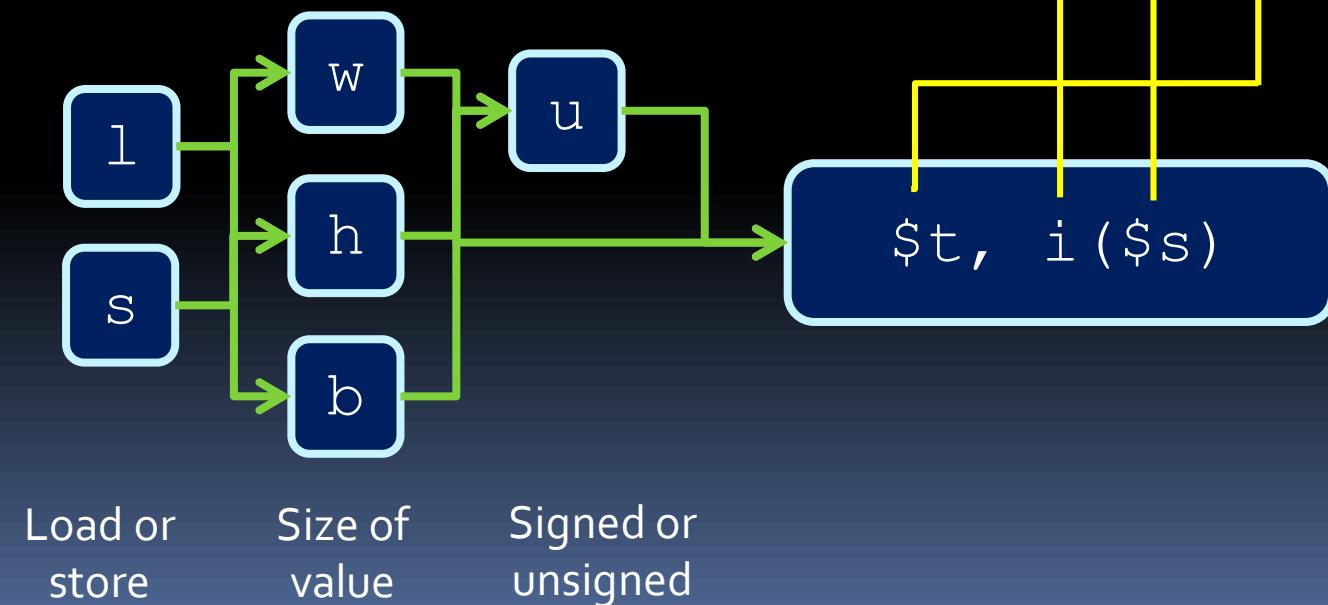
- The terms “load” and “store” are seen from the perspective of the processor, looking at memory.
- Loads are **read** operations.
 - We load (i.e., read) from memory.
 - We load a value **from** a memory address into a register.
- Stores are **write** operations.
 - We **store** (i.e., write) a data value from a register **to** a memory address.
 - Store instructions do not have a destination register, and therefore do not write to the register file.

Memory Instructions in MIPS assembly

- Load & store instructions are I-type operations:



- ...which are written in this format:



Load & store instructions

Instruction	Opcode/Function	Syntax	Operation
lb	100000	\$t, i (\$s)	\$t = SE (MEM [\$s + i]:1)
lbu	100100	\$t, i (\$s)	\$t = ZE (MEM [\$s + i]:1)
lh	100001	\$t, i (\$s)	\$t = SE (MEM [\$s + i]:2)
lhu	100101	\$t, i (\$s)	\$t = ZE (MEM [\$s + i]:2)
lw	100011	\$t, i (\$s)	\$t = MEM [\$s + i]:4
sb	101000	\$t, i (\$s)	MEM [\$s + i]:1 = LB (\$t)
sh	101001	\$t, i (\$s)	MEM [\$s + i]:2 = LH (\$t)
sw	101011	\$t, i (\$s)	MEM [\$s + i]:4 = \$t

- “b”, “h” and “w” correspond to “byte”, “half word” and “word”, indicating the length of the data.
- “SE” stands for “sign extend”, “ZE” stands for “zero extend”.

Memory Instructions in MIPS assembly

Only applicable when loading a byte or a half-word. Choose between **u** for unsigned or leave it blank as for all other cases.

Specifies the location to access as `MEM[$s + SE(i)]`



I for load or
S for store

b for byte,
h for half-word,
w for word

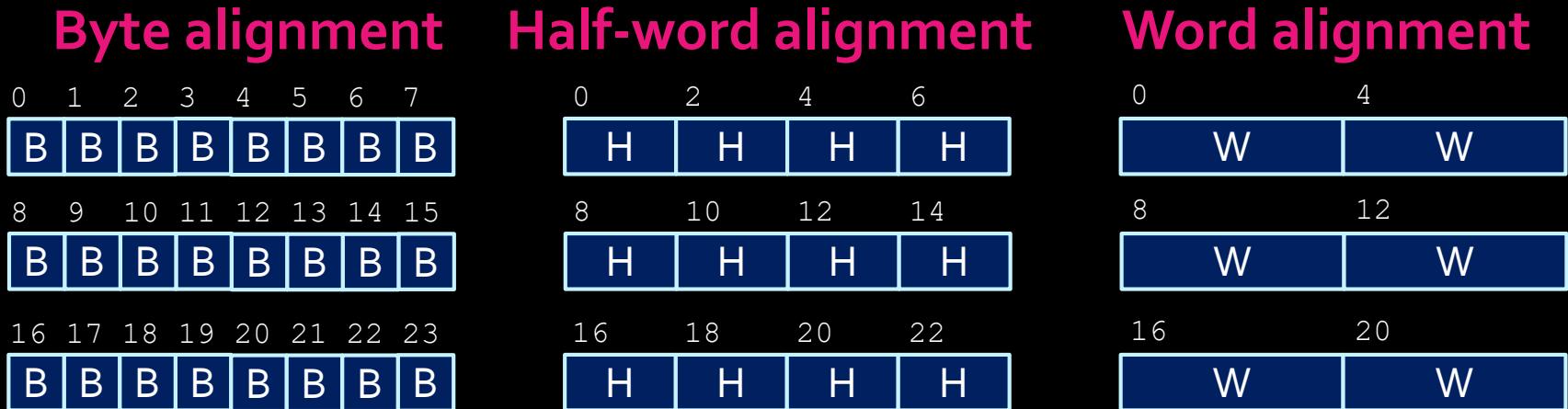
\$t, i(\$s)

Destination register for loads, source register for stores.

Alignment Requirements

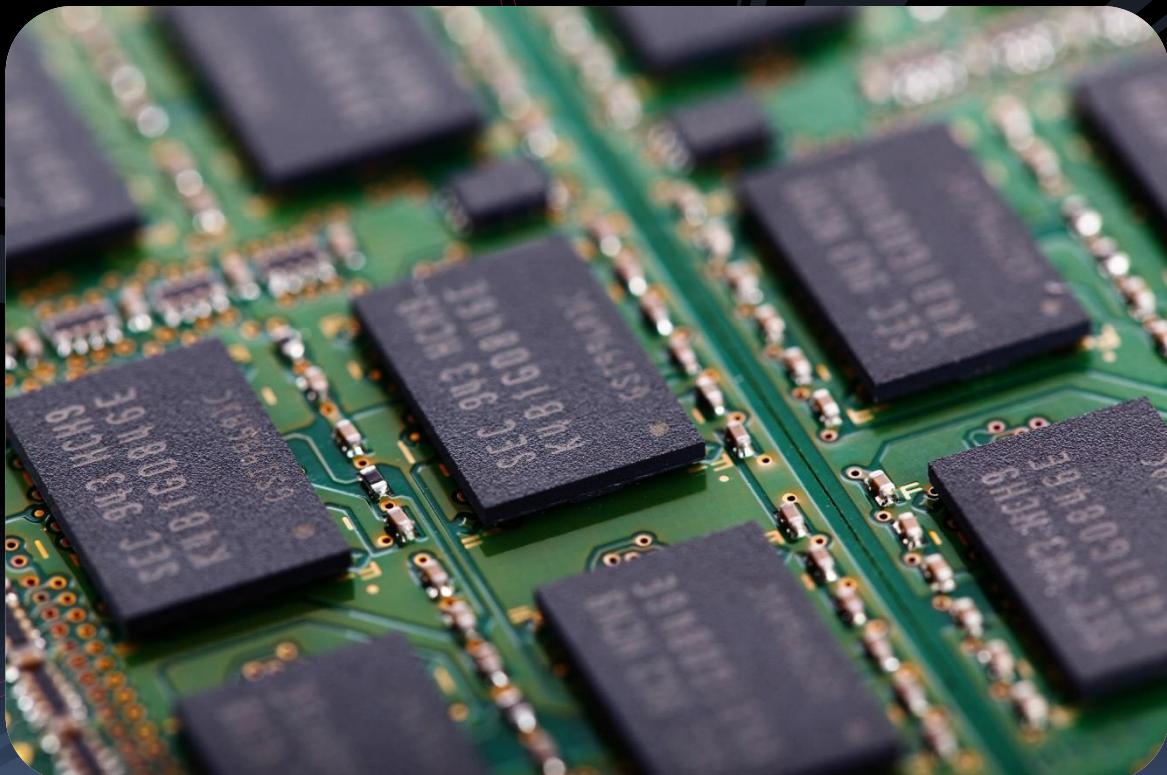
- Misaligned memory accesses result in errors.
 - Word accesses (i.e., addresses specified in a `lw` or `sw` instruction) should be **word-aligned** (divisible by 4).
 - **Half-word** accesses should only involve half-word aligned addresses (i.e., **even addresses**).
 - No constraints for byte accesses.

Legal memory alignment



- These are the same sections of memory, seen from the viewpoint of different memory accesses.
- Fetching words and half-words from invalid addresses will cause the processor to raise an **address error exception**.
 - This is also why addresses stored in the PC need to be divisible by 4,
 - Instruction fetches are word accesses and need to be word-aligned.
- Next question: How are the bytes within a word or half-word stored?

Some notes about memory



Little Endian vs. Big Endian

- Let's say we want to read a word (4 bytes) starting from address X.
- How do we assemble these multiple bytes into a larger data-type?
 - What would you do?

Address	Byte
X	Byte A
X + 1	Byte B
X + 2	Byte C
X + 3	Byte D

BigEndian:

Byte A	Byte B	Byte C	Byte D
--------	--------	--------	--------

LittleEndian:

Byte D	Byte C	Byte B	Byte A
--------	--------	--------	--------

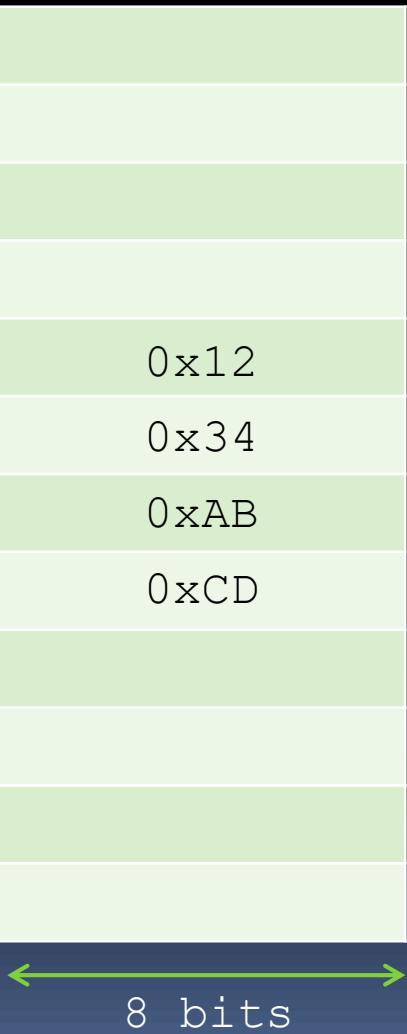
Least significant byte

Big Endian vs. Little Endian

- Big Endian
 - The **most significant byte** of the word is stored first (i.e., at address X). The 2^{nd} most significant byte at address $X+1$ and so on.
- Little Endian
 - The **least significant byte** of the word is stored first (i.e., at address X). The 2^{nd} least significant byte at address $X+1$ and so on.

Big Endian Example

0x00000000	
0x00000001	
0x00000002	
0x00000003	
0x00000004	0x12
0x00000005	0x34
0x00000006	0xAB
0x00000007	0xCD
...	
0xFFFFFFFF	



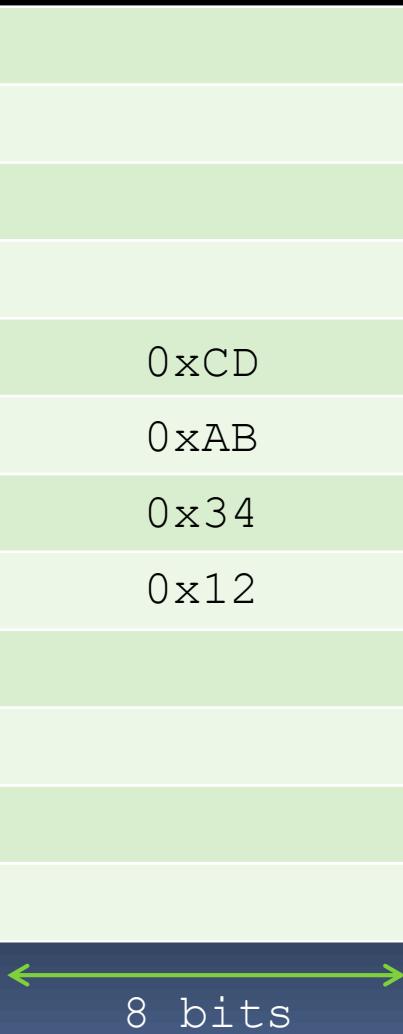
```
#assume $t0 contains  
#0x00000004  
sw $t1, 0($t0)
```

0x1234ABCD

32 bits

LittleEndian Example

0x00000000	
0x00000001	
0x00000002	
0x00000003	
0x00000004	0xCD
0x00000005	0xAB
0x00000006	0x34
0x00000007	0x12
...	
0xFFFFFFFF	



```
#assume $t0 contains  
#0x00000004  
sw $t1, 0($t0)
```

0x1234ABCD

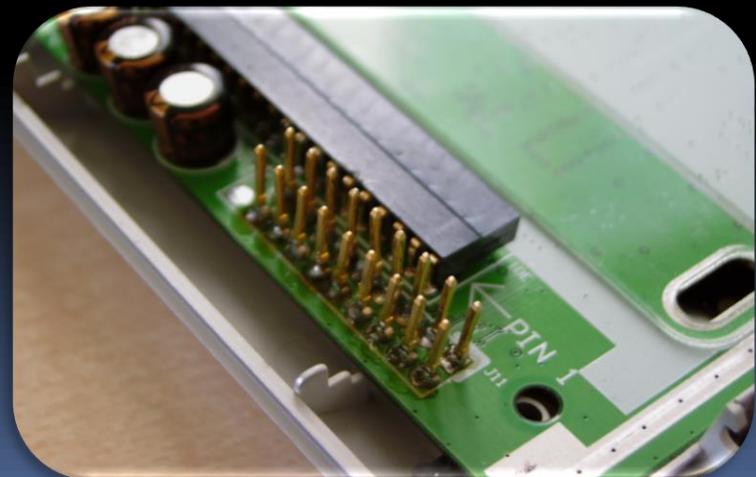
32 bits

MIPS Endianness

- MIPS processors are **bi-endian**, i.e., they can operate with either big-endian or little-endian byte order
- **MARS simulator** uses the same endianness as the machine it is running on.
 - X86 CPUs (like the one in my laptop) are little-endian, for instance.

Reading from devices

- The **offset value** is useful for objects or stack parameters, when multiple addresses are needed relative to a given **base memory location**.
sw \$t0, 10(\$t1)
- Memory is also used to communicate with outside devices, such as keyboards and monitors.
 - Known as **memory-mapped IO**.
 - Invoked with a **trap** or function.



It's a trap!

Instruction	Function	Syntax
trap	011010	i

- Trap instructions send system calls to the operating system
 - e.g. interacting with the user, and exiting the program.
- Similar but not quite the same as the syscall command.

Service	Trap Code	Input/Output
print_int	1	\$4 is int to print
print_float	2	\$f12 is float to print
print_double	3	\$f12 (with \$f13) is double to print
print_string	4	\$4 is address of ASCII string to print
read_int	5	\$2 is int read
read_float	6	\$f12 is float read
read_double	7	\$f12 (with \$f13) is double read
read_string	8	\$4 is address of buffer, \$5 is buffer size in bytes
sbrk	9	\$4 is number of bytes required, \$2 is address of allocated memory
exit	10	
print_byte	101	\$4 contains byte to print
read_byte	102	\$2 contains byte read
set_print_inst_on	103	
set_print_inst_off	104	
get_print_inst	105	\$2 contains current status of printing instructions

Memory segments & syntax

- Programs are divided into two main sections in memory:
 - `.data`
 - Indicates the start of the data values section (typically at beginning of program).
 - `.text`
 - Indicates the start of the program instruction section.
- Within the instruction section are program labels and branch addresses.
 - `main:`
 - The initial line to run when executing the program.
 - Other labels are determined by the function names used in one's program.

`.data`

`.text`

`main:`

Labeling data values

- Data storage:
 - At beginning of program, create labels for memory locations that are used to store values.
 - Always in form: label .type value(s)

```
# create a single integer variable with initial value 3  
var1:      .word    3
```

```
# create a 2-element character array with elements  
# initialized to a and b  
array1:      .byte    'a', 'b'
```

```
# allocate 40 consecutive bytes, with uninitialized  
# storage. Could be used as a 40-element character  
# array, or a 10-element integer array.  
array2:      .space    40
```

Pseudo-Instructions



Pseudo-Instructions

- Pseudo-instructions are there for the convenience of the programmer.
- The assembler translates them into 1 or more real MIPS assembly instructions.
 - “Real” MIPS instructions have opcodes. Pseudo-instructions do not!
 - The assembler often uses the special \$at register (also written as \$1) when mapping pseudo-instructions to MIPS instructions.

* When using MARS, make sure the use of Pseudo-instructions is enabled.

Example: The la pseudo-instruction

- **la** (load address) is a **pseudo-instruction** written in the format:
 - **la \$d, label**
 - loads a register $\$d$ with the memory address that **label** corresponds to.
- Usually translated by the assembler into the following two MIPS instructions:
 - **lui \$at, immediate1 # load upper immediate**
 - The “**immediate1**” represents the **upper 16 bits** of the memory address **label** corresponds to. These bits are loaded in the upper 16 bits of the dest. register. Lowest 16 bits are set to 0.
 - Register **\$at (\$1)** is the register used by the assembler.

Instruction	Opcode/Function	Syntax	Operation
lui	001111	$\$t, i$	$\$t = i \ll 16$

- **ori \$d, \$at, immediate2**
 - “**immediate2**” represents the **lower 16 bits** of the memory address **label** corresponds to.

Another pseudo-instruction example

- Some branch instructions are pseudo-instructions.
 - `bge $s, $t, label`
 - Branch to label iff $\$s \geq \t
 - (comparing register contents).
 - Implemented by using one of comparison instructions followed by `beq` or `bne`.
 - `slt $at, $s, $t # set $at to 1 if $s < $t`
 - `beq $at, $zero, label # branch if $at == 0`

Recall that the `$at` register is reserved for the assembler.

load_store_example.asm

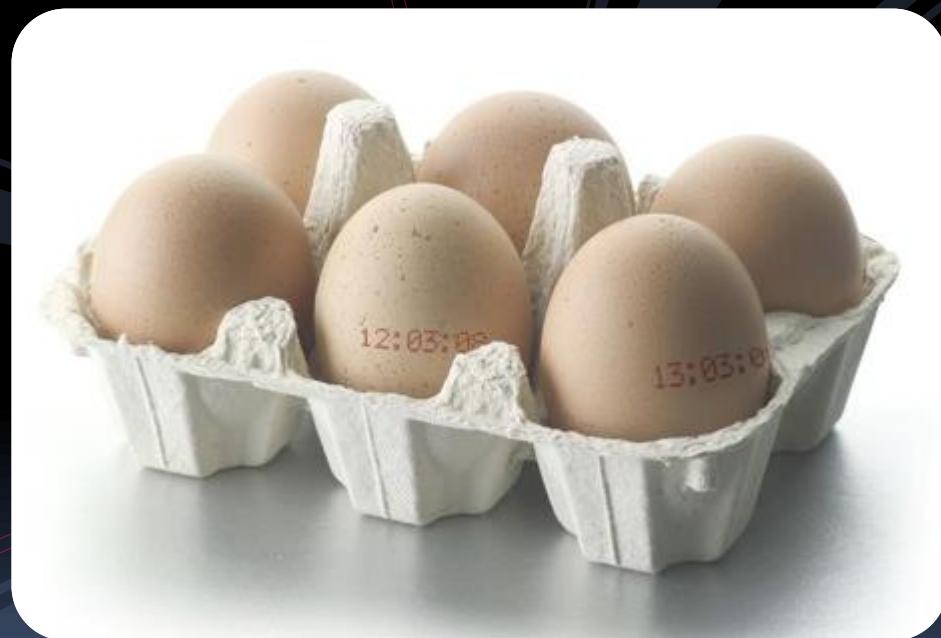
- Practice with loads and stores!
- Note: la is sometimes translated into one instruction instead of two.
 - la \$t1, RESULT1
 - RESULT1 corresponds to address 0x10010000

```
lui $9, 4097
```

- la \$t5, RESULT2
 - RESULT2 corresponds to address 0x10010008

```
lui $1, 4097  
ori $13, $1, 8
```

Arrays and Structs



Arrays!

```
int A[100], B[100];  
for (i=0; i<100; i++) {  
    A[i] = B[i] + 1;  
}
```

- Arrays in assembly language:
 - Arrays are stored in **consecutive locations** in memory.
 - The **address** of the array is the address of the **array's first element**.
 - To access **element *i*** of an array, use *i* to calculate an offset distance. Add that offset to the address of the first element to get the address of the *i*th element.
 - **offset = *i* * the size of a single element**
 - To operate on array elements, **load** the array values into registers. Operate on them, then **store** them back into memory.

Translating arrays

```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

```
.data
A:   .space 400          # array of 100 integers
B:   .word 42:100        # array of 100 integers, all
                           # initialized to value of 42

.text
main: la $t8, A           # $t8 holds address of array A
     la $t9, B           # $t9 holds address of array B
     add $t0, $zero, $zero # $t0 holds i = 0
     addi $t1, $zero, 100  # $t1 holds 100

LOOP: bge $t0, $t1, END    # exit loop when i>=100
      sll $t2, $t0, 2      # $t2 = $t0 * 4 = i * 4 = offset
      add $t3, $t8, $t2      # $t3 = addr(A) + i*4 = addr(A[i])
      add $t4, $t9, $t2      # $t4 = addr(B) + i*4 = addr(B[i])
      lw $t5, 0($t4)         # $t5 = B[i]
      addi $t5, $t5, 1        # $t5 = $t5 + 1 = B[i] + 1
      sw $t5, 0($t3)         # A[i] = $t5

UPDATE: addi $t0, $t0, 1      # i++
        j LOOP               # jump to loop condition check

END:   ...                  # continue remainder of program.
```

Another translation

```
int A[100], B[100];  
for (i=0; i<100; i++) {  
    A[i] = B[i] + 1;  
}
```

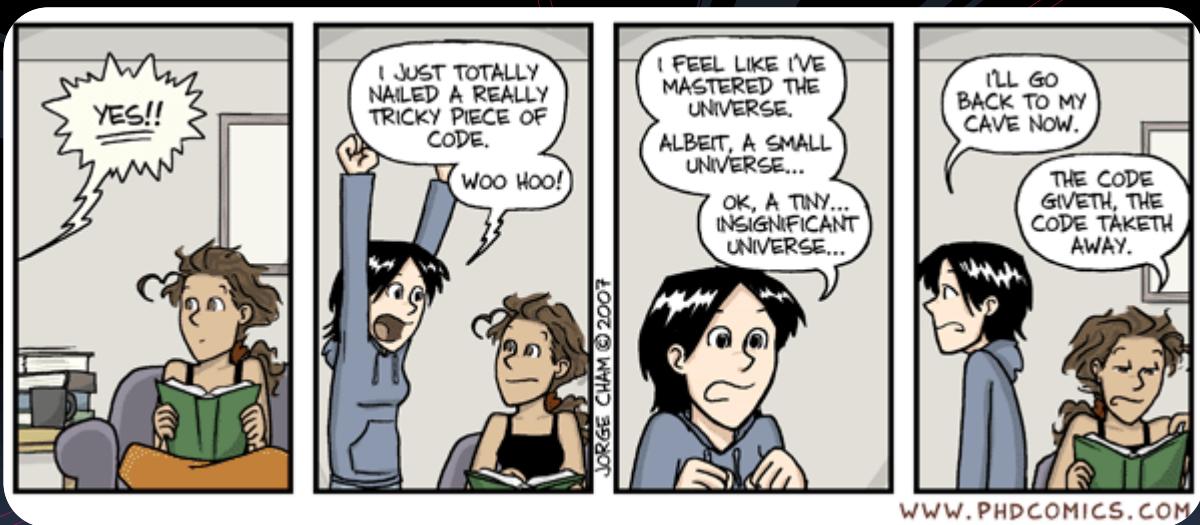
```
.data  
A:      .space    400          # array of 100 integers  
B:      .word     21:100        # array of 100 integers,  
                           # all initialized to 21 decimal.  
  
.text  
main:   la $t8, A            # $t8 holds address of A  
        la $t9, B            # $t9 holds address of B  
        add $t0, $zero, $zero # $t0 holds 4*i; initially 0  
        addi $t1, $zero, 400  # $t1 holds 100*sizeof(int)  
  
LOOP:   bge $t0, $t1, END    # branch if $t0 >= 400  
        add $t3, $t8, $t0    # $t3 holds addr(A[i])  
        add $t4, $t9, $t0    # $t4 holds addr (B[i])  
        lw $t5, 0($t4)       # $t5 = B[i]  
        addi $t5, $t5, 1     # $t5 = B[i] + 1  
        sw $t5, 0($t3)       # A[i] = $t5  
        addi $t0, $t0, 4      # update offset in $t0  
        j LOOP  
  
END:
```

Example: A struct program

- How can we figure out the main purpose of this code?
- The sw lines indicate that values in \$t1 are being stored at \$t0, \$t0+4 and \$t0+8.
 - Each previous line sets the value of \$t1 to store.
- Therefore, this code stores the values 5, 13 and -7 into the struct at location a1.

```
.data  
a1: .space 12  
  
.text  
main:  
    addi    $t0, $zero, a1  
    addi    $t1, $zero, 5  
    sw     $t1, 0($t0)  
    addi    $t1, $zero, 13  
    sw     $t1, 4($t0)  
    addi    $t1, $zero, -7  
    sw     $t1, 8($t0)
```

Designing Assembly Code



Making sense of assembly code

- Assembly language **looks** intimidating because the programs involve a lot of code.
 - No worse than your CSC108 assignments would look to the untrained eye!
- The **key** to reading and designing assembly code is **recognizing** portions of code that represent **higher-level operations** that you're familiar with.

Array code example

- How did we create our solutions?
- First stage: Initialization
 - Store locations of A[0] and B[0] (in \$t8 and \$t9, for example).
 - Create a value for i (\$t0), and set it to zero.
 - Create a value to store the max value for i, as a stopping condition (in \$t1, in this case).
- Note: Best to initialize all the registers that you'll need at once, even ones that don't have variable names in the original code.

```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

Array code example

- Second stage: Main processing operation
 - Fetch source ($B[i]$).
 - Get the address of $B[i]$ by adding i to the address of $B[0]$ (stored here in $\$t3$).
 - Load the value of $B[i]$ from that memory address (in $\$s4$).
 - Ready destination ($A[i]$).
 - Same steps as for $B[i]$, but address is stored in $\$t4$.
 - Add 1 to $B[i]$ (storing the result in $\$t6$).
 - Store this new value into $A[i]$.
 - Same as fetching a value from memory, but in reverse.
 - Increment i to the next offset value.
 - Loop to the beginning if i hasn't reached its max value.

```
int A[100], B[100];  
for (i=0; i<100; i++) {  
    A[i] = B[i] + 1;  
}
```

Loop exercise

```
int j=0;  
for (int i=0; i<50; i++) {  
    j += i;  
}
```

```
.data  
i: .space 4      # $t0 for addr. $t2 as temp  
j: .space 4      # $t1 for addr. $t3 as temp  
.text  
main:    la      $t0, i          # load addr of i  
         la      $t1, j          # load addr of j  
         sw      $zero, 0($t0)    # set mem i to 0  
         sw      $zero, 0($t1)    # set mem j to 0  
         add    $t2, $zero, $zero  # set reg i to 0  
         add    $t3, $zero, $zero  # set reg j to 0  
         addi   $t9, $zero, 50     # end: i==50  
loop:    beq    $t2, $t9, end    # i==50?  
         add    $t3, $t3, $t2     # j = j+i  
         addi   $t2, $t2, 1       # i++  
         sw      $t2, 0($t0)      # store i in mem  
         sw      $t3, 0($t1)      # store j in mem  
         j      loop  
end:    # do the next thing
```

Shorter version

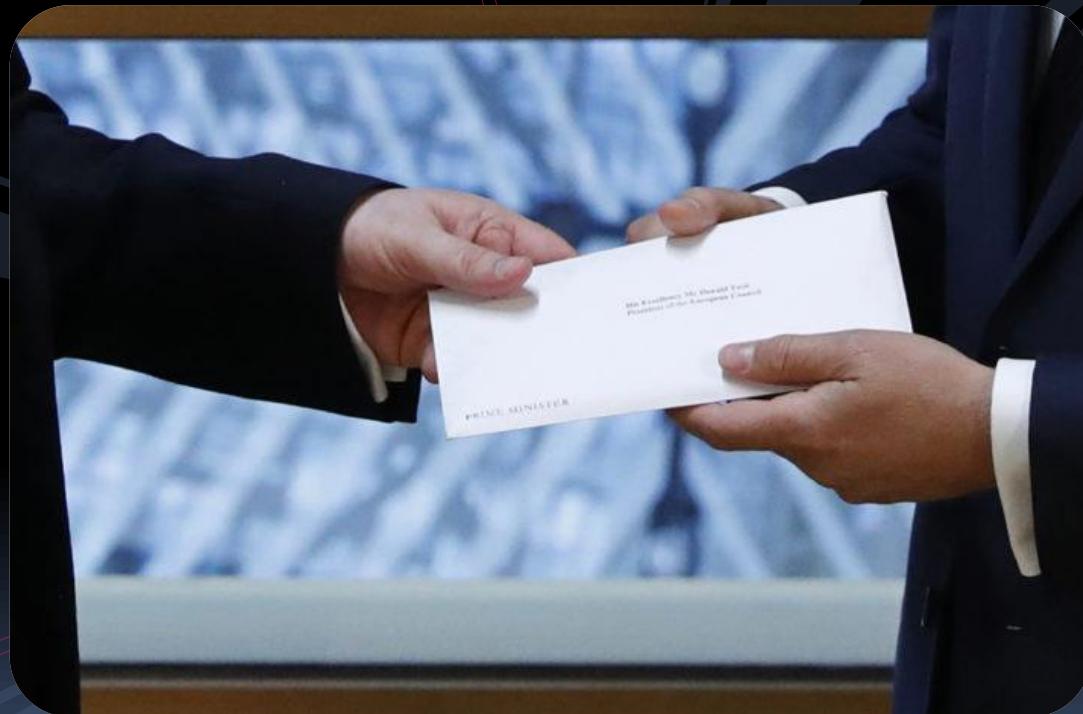
```
.data
i:      .space    4
j:      .space    4
.text
main:   la      $t0, i          # load addr of i
        la      $t1, j          # load addr of j
        add   $t2, $zero, $zero # set reg i to 0
        add   $t3, $zero, $zero # set reg j to 0
        addi  $t9, $zero, 50   # end when i==50
loop:   sw      $t2, 0($t0)    # store i in mem
        sw      $t3, 0($t1)    # store j in mem
        beq   $t2, $t9, end    # i==50?
        add   $t3, $t3, $t2   # j = j+i
        addi  $t2, $t2, 1     # i++
        j     loop
end:   # do the next thing
```

```
int j=0;
for (int i=0; i<50; i++) {
    j += i;
}
```

Can you spot what was changed, and why?

Functions in Assembly

or: *why we need the stack*



Functions vs Code

- Up to this point, we've been looking at how to create pieces of code in isolation.
- A **function** creates an interface to this piece of code by defining an entry and exit point to it. the input and output parameters.
- This requires two major considerations:
 1. How to jump into and out of a function.
 2. How to pass values to and from a function.

Function Calls & Returns



How Functions Work

- To support functions we need to be able to:
 - Define the start of a function
 - Label the first line to provide a target address to jump to.
 - Take in function arguments and return values
 - Could use registers...but might need another solution.
 - Store variables local to the function and also ensure functions don't clobber useful data on registers
 - Registers for storing data, but are any off-limits?
 - Return to the calling site
 - after the last line in the function, return to the instruction after the one that did the function call.

How do we call a function?

- `jal FUNCTION_LABEL`
 - This jumps to the first line of the function, which has the specified label (i.e. `function_X` here)
- `jal` is a J-Type instruction.
 - It updates register `$31 ($ra, return address register)` and also the program counter.
 - After it's executed, `$ra` contains the address of the instruction *after* the line that called `jal`.

```
...  
sum = 3;  
function_X(sum);  
sum = 5;
```

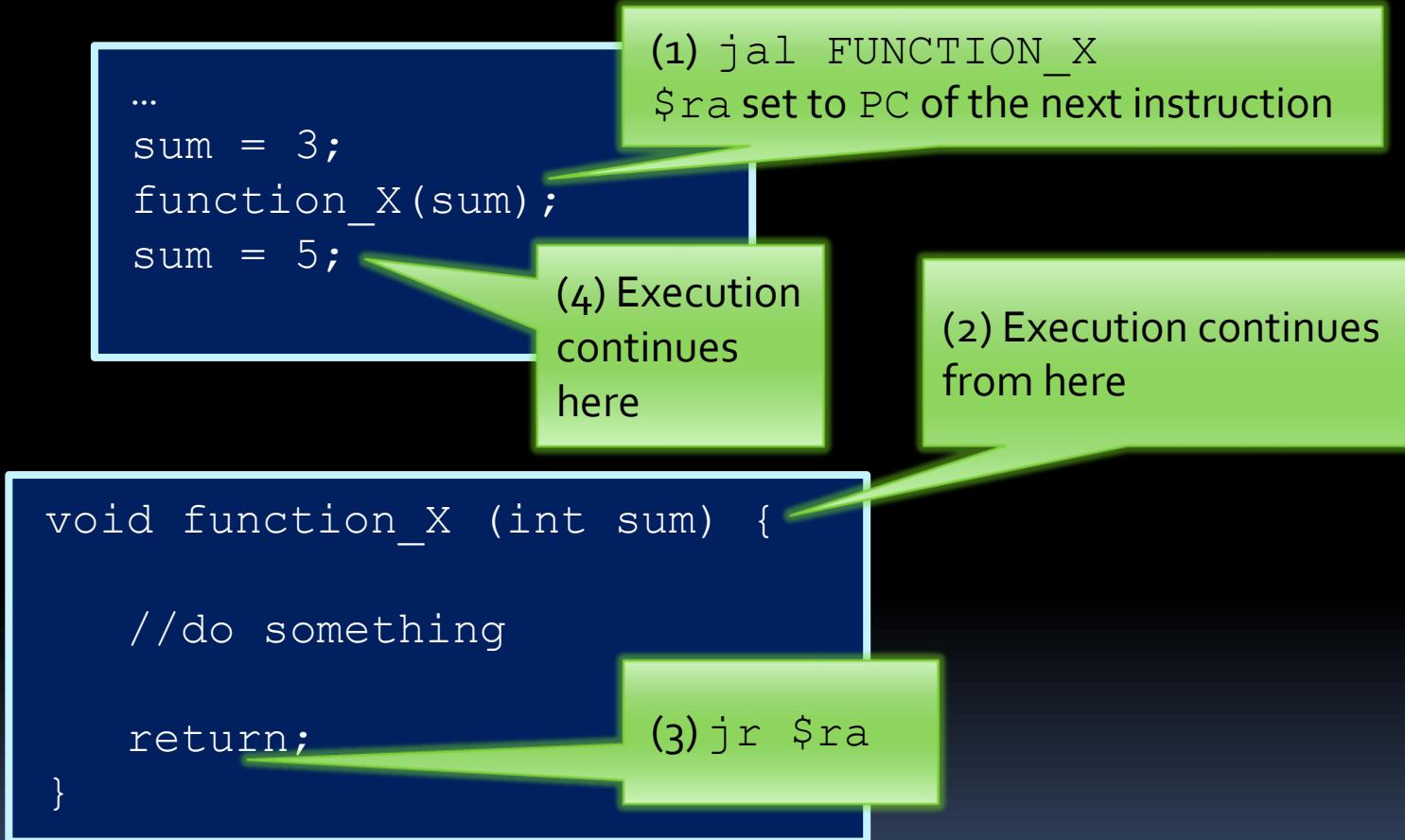
How do we return from a function?

- `jr $ra`
 - The PC is set to the address in `$ra`.
- But how do we know what's in `$ra`?
 - `$ra` was set by the most recent `jal` instruction (function call)!

```
...  
sum = 3;  
function_X(sum);  
sum = 5;
```

```
void function_X (int sum) {  
  
    //do something  
  
    return;  
}
```

Function Calls - Cont'd



Nested Function Call Issue

```
...  
sum = 3;  
function_X(sum);  
sum += 5;
```

(1) jal FUNCTION_X
\$ra set to PC of the next instruction.

(2) Execution continues from here

```
void function_X (int sum) {  
  
    //do something  
    function_Y(),  
  
    return;  
}
```

(3)
jal FUNCTION_Y
\$ra set to PC of next instruction

(4) Execution continues from here

```
void function_Y () {  
  
    //do something  
  
    return;
```

(5) jr \$ra

(6) Execut
jr \$ra

Which \$ra?
No way back! 😞

Solving the \$ra dilemma

- How do we **make sure** that we **don't clobber** the value of \$ra **every time** we call a nested function?
- Need to **put \$ra away somewhere** for safe keeping if we know we're about to overwrite it.
- Would be **ideal** if the structure used for this made it **easy to find the last item** that was stored away.

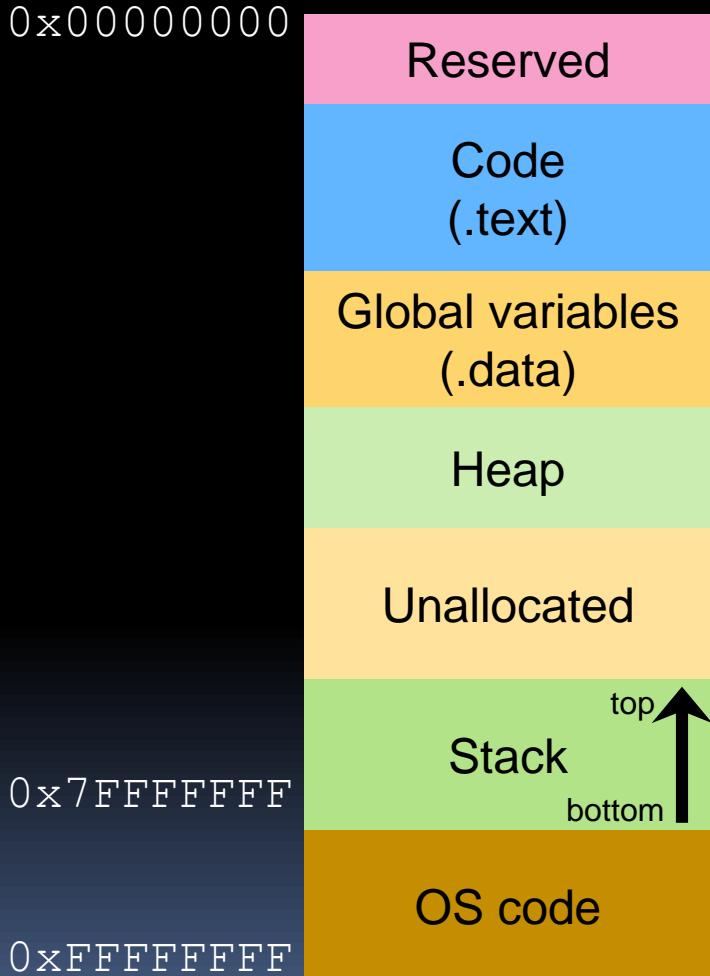
The Stack



The Stack and the Stack Pointer

- The **stack** is a spot in memory used to store values independent of the registers (which can get overwritten easily)
- A special register stores the **stack pointer**, which points to *the last element pushed onto the top of the stack*.
 - For MIPS the stack pointer is \$29 (\$sp). This **holds the address of the last element pushed to the top of the stack**
 - In other systems \$sp could point to the first empty location on top of the stack.
- We can **push** data (like \$ra) onto the stack (which makes it **grow**) and **pop** data from the stack (which makes it **shrink**).
- The stack is allocated a **maximum space** in memory. If it grows too large, there is the risk of it exceeding this predefined size and/or **overlapping with the heap**.

The programmer's view of memory

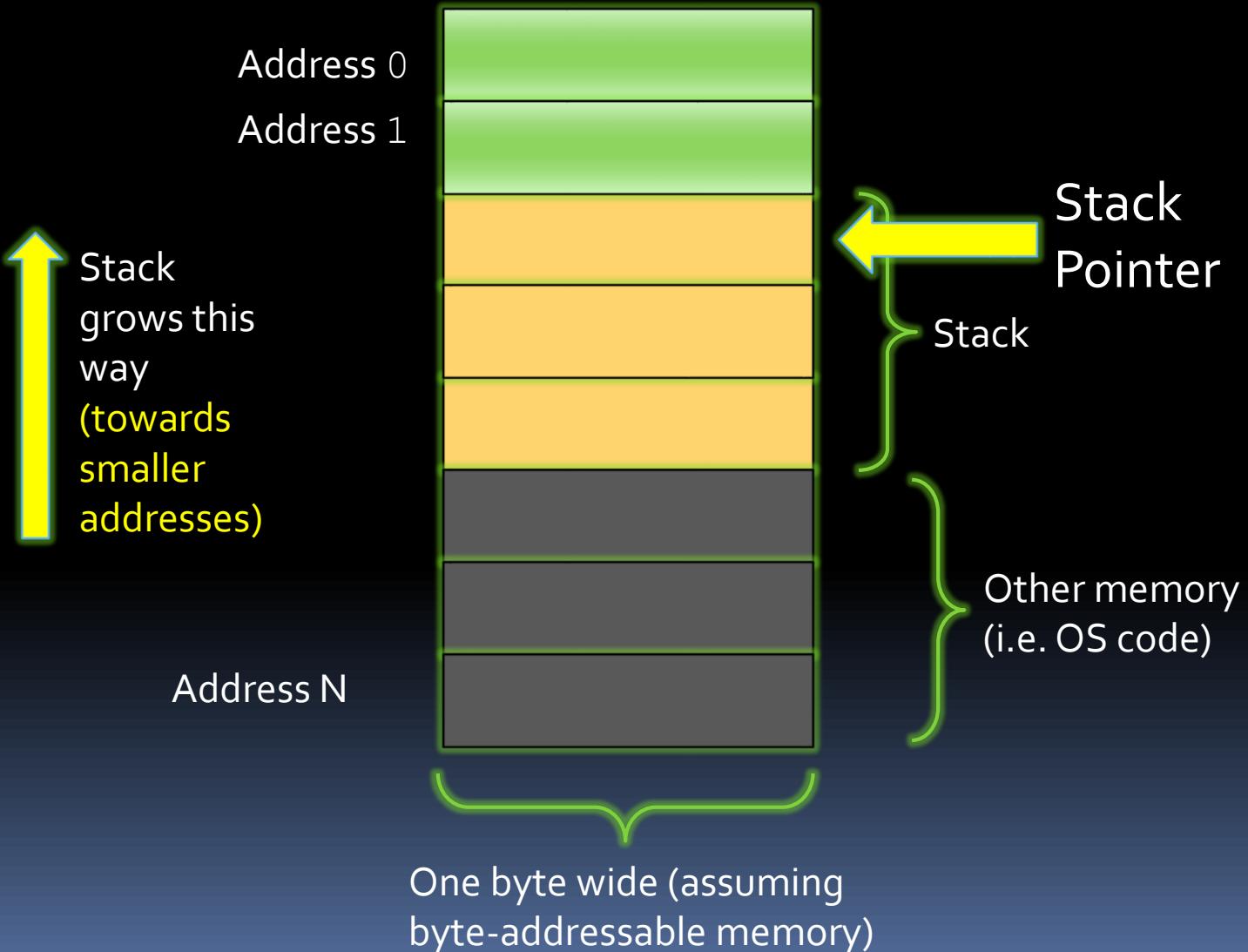


- The stack is a part of memory used for function calls etc.
- The **stack grows towards smaller (lower) addresses**
 - see arrow.
- The stack uses **LIFO (last-in first-out) order**.
 - Like a physical pile that you add and remove items from.

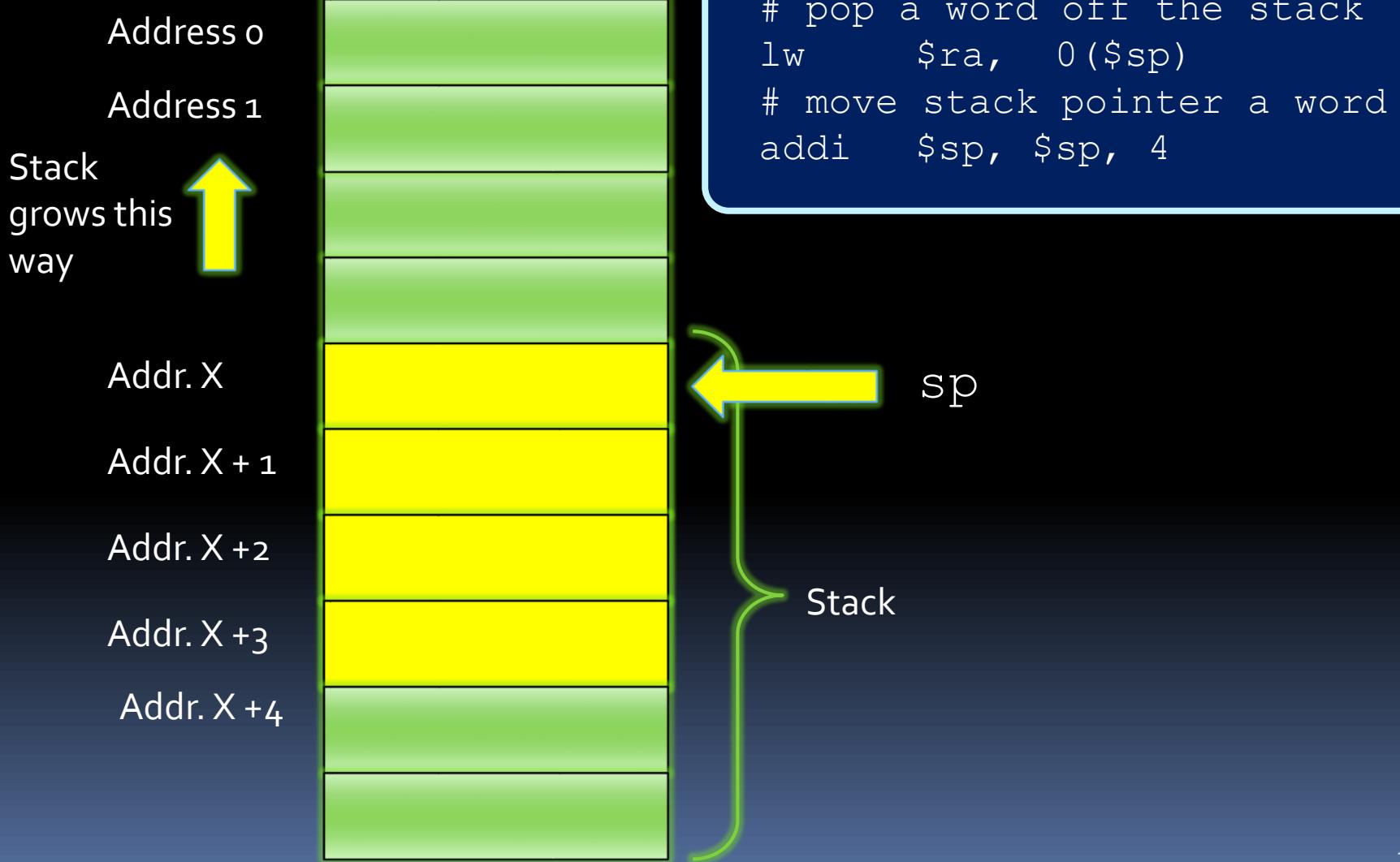
The stack to the rescue!

- When do you store values onto the stack?
 - Whenever you call a function and want to preserve values from getting overwritten (like \$ra).
- What happens when you have nested function calls, each of which stores \$ra on the stack?
 - Different \$ra values will exist in layers on the stack over time.
- We can also use the stack to store:
 - Function arguments
 - Function return values
 - (more on this later).

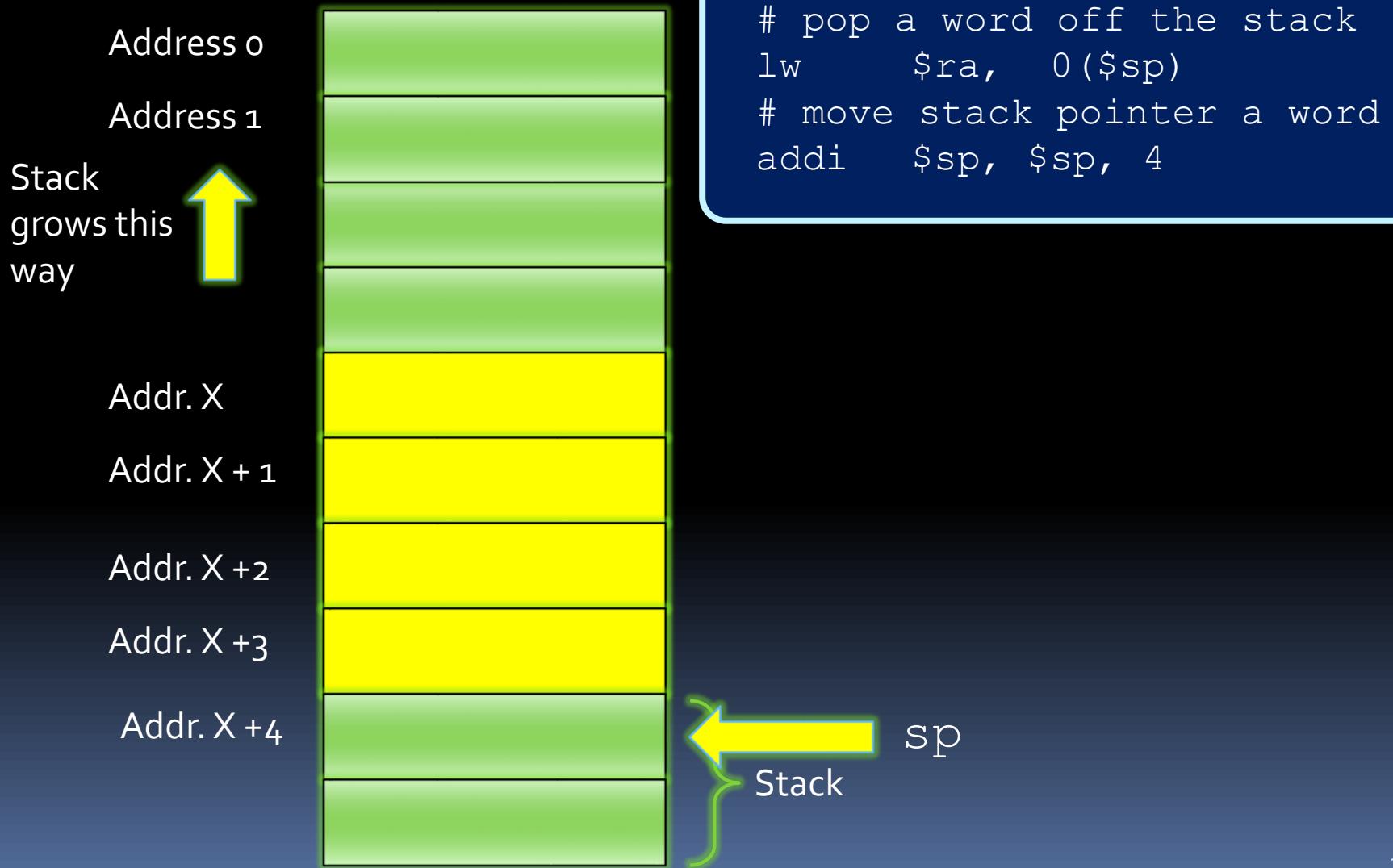
The Stack, illustrated



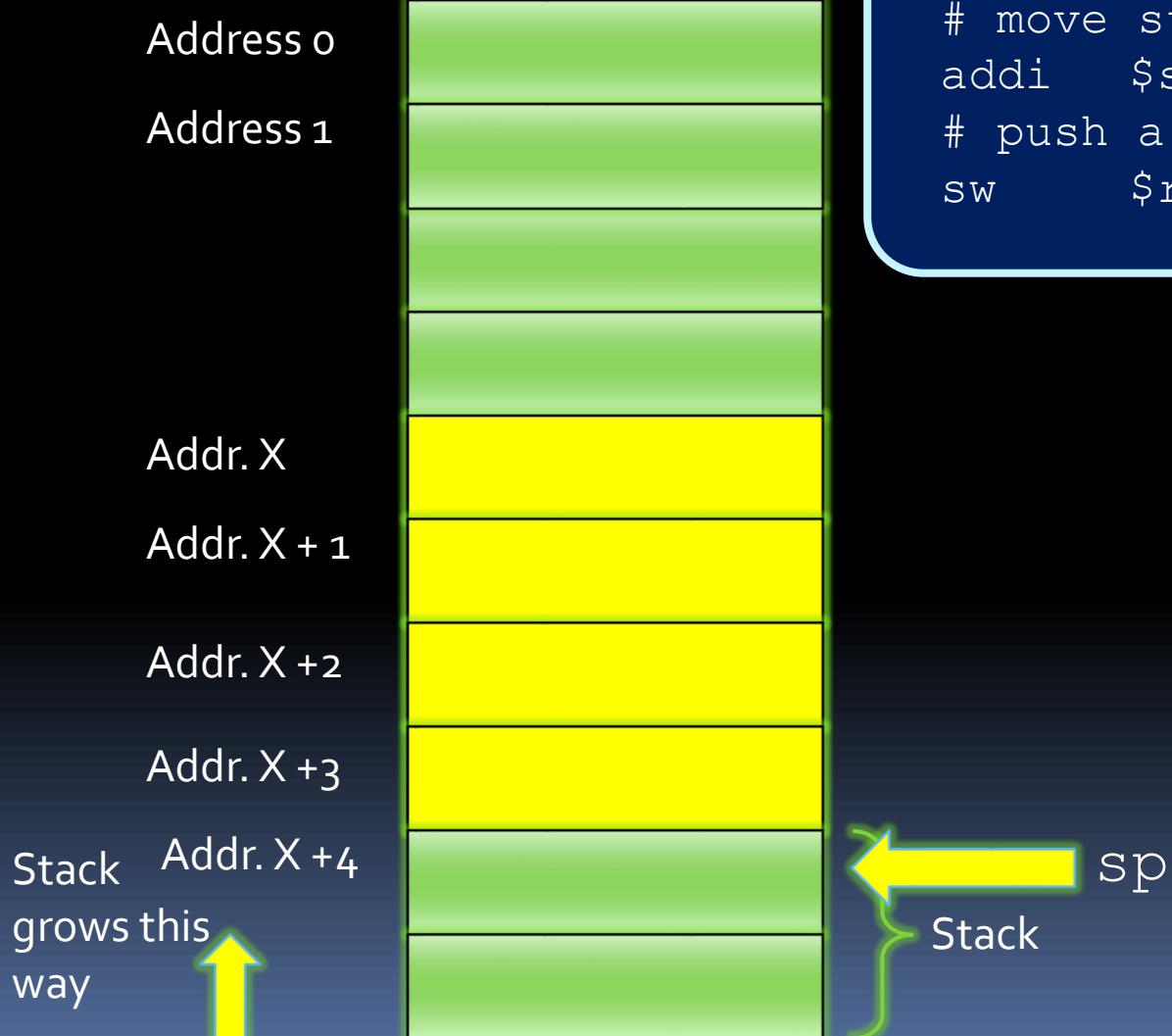
Popping Values off the stack - Before



Popping Values off the stack - After

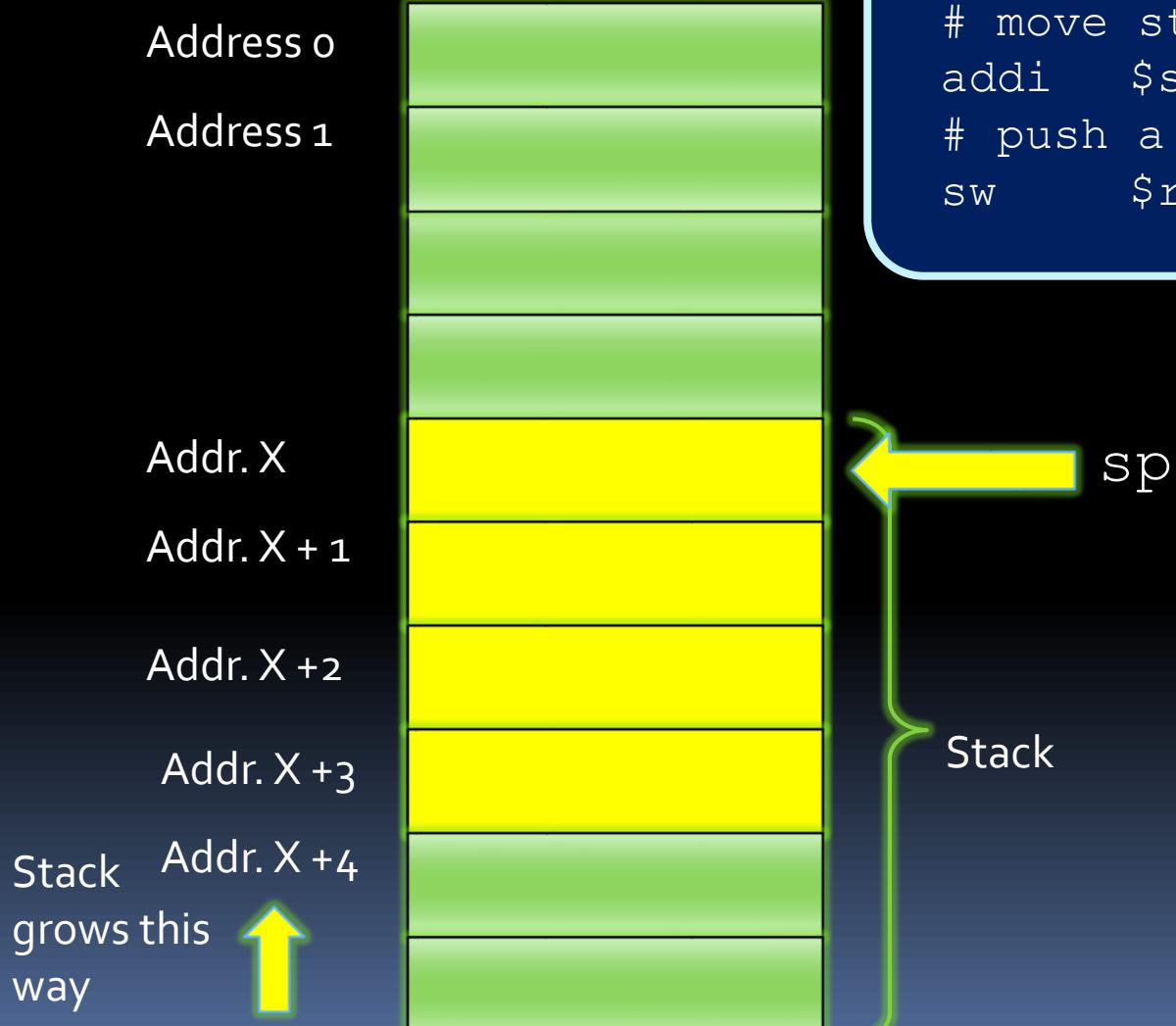


Pushing Values to the stack - Before



```
# move stack pointer a word  
addi    $sp, $sp, -4  
# push a word onto the stack  
sw      $ra, 0($sp)
```

Pushing Values to the stack - After



Stack Usage

- Whenever we refer to “pushing” and “popping” values...
- Pushing something onto the stack means:
 - Allocate space by decrementing the stack pointer by the appropriate number of bytes.
 - Do a store (or multiple stores as needed).
- Popping something from the stack means:
 - Do a load (or multiple loads as needed)
 - De-allocate space by incrementing the stack pointer by the appropriate number of bytes.

Advice on using the stack

- Any space you allocate on the stack, you should later de-allocate.
- If you **push** items in a **certain order**, you should **pop** the items in the **reverse order**.
 - It might help to draw out an image of how your stack will look like.
- When pushing **more than one item** onto the stack, you can :
 - Either allocate all the space in the beginning or allocate space as you go.
 - Same principle applies for popping.

Passing function parameters



Functions vs Code

- Since functions have entry and exit points, they also need **input** and **output parameters**.
 - In **other languages**, these parameters are assumed to be available at the start of the function.
 - In **assembly**, you have to fetch those values from memory first before you can operate on them.
- Where do you look for these parameters?

Common Calling Conventions

- While most programs pass parameters through the stack, it is also possible to use registers to pass values to and from programs:
 - Registers 2–3 (\$v0, \$v1): return values
 - Registers 4–7 (\$a0-\$a3): function arguments
- If your function has up to 4 arguments, you would use the \$a0 to \$a3 registers in that order. Any additional arguments would be pushed on the stack.
 - First argument in \$a0, second in \$a1, and so on.
 - More common convention is to just push all arguments to the stack. On a final exam, we'll tell you what to do.

String function program

```
void strcpy (char x[], char y[]) {  
    int i;  
    i=0;  
    while ((x[i] = y[i]) != '\0')  
        i += 1;  
    return 1;  
}
```

- Let's convert this to assembly code!
- Let's also take in parameters from the stack!
 - In this case, the **parameters** **x** and **y** are passed into the function, in that order.
 - The pointer to the stack is stored in register \$29 (aka \$sp), which is the address of *the top element of the stack*.

Converting strcpy()

- Initialization:
 - What values do we need to store?
 - The address of $x[0]$ and $y[0]$
 - The current offset value (i in this case)
 - Temporary values for the address of $x[i]$ and $y[i]$
 - The current value being copied from $y[i]$ to $x[i]$.

```
void strcpy (char x[], char y[]) {  
    int i;  
    i=0;  
    while ((x[i] = y[i]) != '\0')  
        i += 1;  
    return 1;  
}
```

Converting strcpy()

- Initialization (cont'd):
 - Consider that the locations of $x[0]$ and $y[0]$ are passed in on the stack, we need to fetch those first.
 - Basic code for popping values off the stack:

```
lw      $t0, 0($sp)    # pop that word off the stack  
addi   $sp, $sp, 4     # move stack pointer by a word
```

- Basic code for pushing values onto the stack:

```
addi   $sp, $sp, -4    # move stack pointer one word  
sw      $t0, 0($sp)    # push a word onto the stack
```

Converting strcpy()

- Main algorithm:

- What steps do we need to perform?
 - Get the location of $x[i]$ and $y[i]$.
 - Fetch a character from $y[i]$ and store it in $x[i]$.
 - Jump to the end if the character is the null character.
 - Otherwise, increment i and jump to the beginning.
 - At the end, push the value 1 onto the stack and return to the calling program.

```
int strcpy (char x[], char y[]) {  
    int i;  
    i=0;  
    while ((x[i] = y[i]) != '\0')  
        i += 1;  
    return 1;  
}
```

Translated S-

```
int strcpy (char x[], char y[]) {  
    int i=0;  
    while ((x[i] = y[i]) != '\0')  
        i += 1;  
    return 1;  
}
```

strcpy:	lw \$a0, 0(\$sp)	# pop y address
	addi \$sp, \$sp, 4	# off the stack
initialization {	lw \$a1, 0(\$sp)	# pop x address
	addi \$sp, \$sp, 4	# off the stack
L1:	add \$t0, \$zero, \$zero	# \$t0 = offset i
	add \$t1, \$t0, \$a0	# \$t1 = y + i
	lb \$t2, 0(\$t1)	# \$t2 = y[i]
main algorithm {	add \$t3, \$t0, \$a1	# \$t3 = x + i
	sb \$t2, 0(\$t3)	# x[i] = \$t2
	beq \$t2, \$zero, L2	# y[i] = '\0' ?
	addi \$t0, \$t0, 1	# i++
	j L1	# loop
L2:	addi \$sp, \$sp, -4	# push 1 onto
end {	addi \$t0, \$zero, 1	# the top of
	sw \$t0, 0(\$sp)	# the stack
	jr \$ra	# return

Function Considerations

We need to calculate the total price
The sales tax rate is 8.65 %
Your program needs to multiply the purchase price by the tax rate, and then add the results and the price and store them in the total price field.



I need to know:

- What is the op-code to load from memory?
- Where is the purchase price stored in memory?
- What is the op-code to multiply?
- What do I multiply by?
- What is the op-code to add two values?
- What is the op-code to store a value in memory?

```
Machine Language
```

0AEF:0220	00	02	8B	76	02	3B	1C	74	45
0AEF:0230	B3	3A	3B	5C	FE	32	D8	86	1C
0AEF:0240	D8	39	EB	3B	D6	F2	8C	E8	B2
0AEF:0250	E1	74	09	4C	3B	3B	F1	72	ED
0AEF:0260	59	5E	3A	5C	FF	1C	73	95	E8
0AEF:0270	9B	D8	E9	C9	D7	46	B1	0C	BB
0AEF:0300	00	3E	F0	97	01	B1	AC	E9	58
0AEF:0310	B1	89	3E	32	99	00	B9	3E	32
0AEF:0320	99	C6	06	34	99	1D	E8	8F	E3
0AEF:0330	75	18	5A	0B	12	3A	E8	29	01
0AEF:0340	58	89	3E	32	99	EB	74	06	E8
0AEF:0350	17	B1	4C	EB	78	EB	CE	E0	3C
0AEF:0360	2E	75	09	FB	06	3C	3F	75	83
0AEF:0370	88	CF	02	3C	2A	6E	99	00	75

Program Entered
And Executed As
Machine Language

Function Call Refresher

- How do I call a function?
 - `jal FUNCTION_LABEL`
 - Which register does `jal` set? To what value?
- How do I return from a function?
 - `jr $ra`
 - But what if I have nested calls? Won't `$ra` get overwritten?
 - Yes. You need to push it to the stack! And then restore it.

Maintaining register values

- We've already demonstrated why we'd need to push \$ra onto the stack when having nested function calls.
- What about the other registers?
 - How do we know that a function we called didn't overwrite registers that we were using?
 - Remember there is only one register file!

Solution: **caller vs. callee calling conventions.**

Calling Conventions

- **Caller vs. Callee**
 - Caller is the function calling another function.
 - Callee is the function being called.
- We separate registers into:
 - Caller-Saved registers (\$t0–\$t9)
 - Callee-Saved registers (\$s0–\$s7)

A function can be both a caller and a callee (i.e. recursion).

Register Saving Conventions

- **Caller-Saved registers**
 - Registers 8–15, 24–25 (\$t0–\$t9): temporaries
 - **Registers that the caller should save to the stack before calling a function.** If they don't save them, there is no guarantee the contents of these registers will not be clobbered.
- **Callee-Saved registers**
 - Registers 16–23 (\$s0–\$s7): saved temporaries
 - **It is the responsibility of the callee to save these registers and later restore them,** if it's going to modify them.
 - Push them to the stack first thing in your function body and restore them just before you return!

Push them to the stack just before you call another function and restore them immediately after.

Stack & function summary

- Before calling a subroutine:
 - Push registers onto the stack to preserve their values.
 - Push the input parameters onto the stack.
- At the start of a subroutine:
 - Pop the input parameters from the stack.
- At the end of a subroutine:
 - Push the return values onto the stack.
- Coming back from a subroutine call:
 - Pop the return values from the stack.
 - Pop the saved register values and restore them.

Recursion in Assembly



Example: factorial(int n)

- Basic pseudocode for recursive factorial:

- Base Case ($n == 0$)
 - return 1
- Get $\text{factorial}(n-1)$
 - Store result in “product”
- Multiply product by n
 - Store in “result”
- Return result

$n!$

Recursive programs

- How do we handle recursive programs?
 - Still needs base case and recursive step, as with other languages.
 - Main difference (w.r.t. other languages): Maintaining register values.
 - When a recursive function calls itself in assembly, it calls `jal` back to the beginning of the program.
 - What happens to the `previous value for $ra?`
 - What happens to the `previous register values`, when the program runs a second time?

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

Recursive programs

- Solution: the **stack!**
 - Before recursive call, **store** the register values that you use onto the stack, and **restore** them when you come back to that point.
 - **Don't forget to store \$ra** as one of those values, or else the program will loop forever!

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

Factorial solution

- Steps to perform:
 - Pop x off the stack.
 - Check if x is zero:
 - If $x==0$, push 1 onto the stack and return to the calling program.
 - If $x \neq 0$, push $x-1$ onto the stack and call factorial again (i.e. jump to the beginning of the code).
 - After recursive call, pop result off of stack and multiply that value by x .
 - Push result onto stack, and return to calling program.

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

Pseudocode

- Pop n off the stack
 - Store in $\$t0$
- If $\$t0 == 0$,
 - Push 1 onto stack
 - Return to calling program
- If $\$t0 != 0$,
 - Calculate $n-1$
 - Store (i.e. push) $\$t0$ and $\$ra$ onto stack
 - Push $n-1$ onto stack
 - Call factorial
 - *...time passes...*
 - Pop the result of factorial ($n-1$) from stack, store in $\$t2$
 - Restore (i.e. pop) $\$ra$ and $\$t0$ from stack
 - Multiply factorial ($n-1$) and n
 - Push result onto stack
 - Return to calling program

```
n → $t0
n-1 → $t1
fact(n-1) → $t2
```

Translated recursive factorial

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

main:	addi \$t0, \$zero, 10	# call fact(10)
	addi \$sp, \$sp, -4	# by putting 10
	sw \$t0, 0(\$sp)	# onto stack
	jal factorial	# result will be
	...	# on the stack
<hr/>		
factorial:	lw \$t0, 0(\$sp)	# get x from stack
	bne \$t0, \$zero, rec	# base case?
base:	addi \$t1, \$zero, 1	# put return value
	sw \$t1, 0(\$sp)	# onto stack
	jr \$ra	# return to caller
rec:	addi \$t1, \$t0, -1	# x--
	addi \$sp, \$sp, -4	# put \$ra value
	sw \$ra, 0(\$sp)	# onto stack
	addi \$sp, \$sp, -4	# put x-1 on stack
	sw \$t1, 0(\$sp)	# for rec call
	jal factorial	# recursive call

Translated recursive factorial

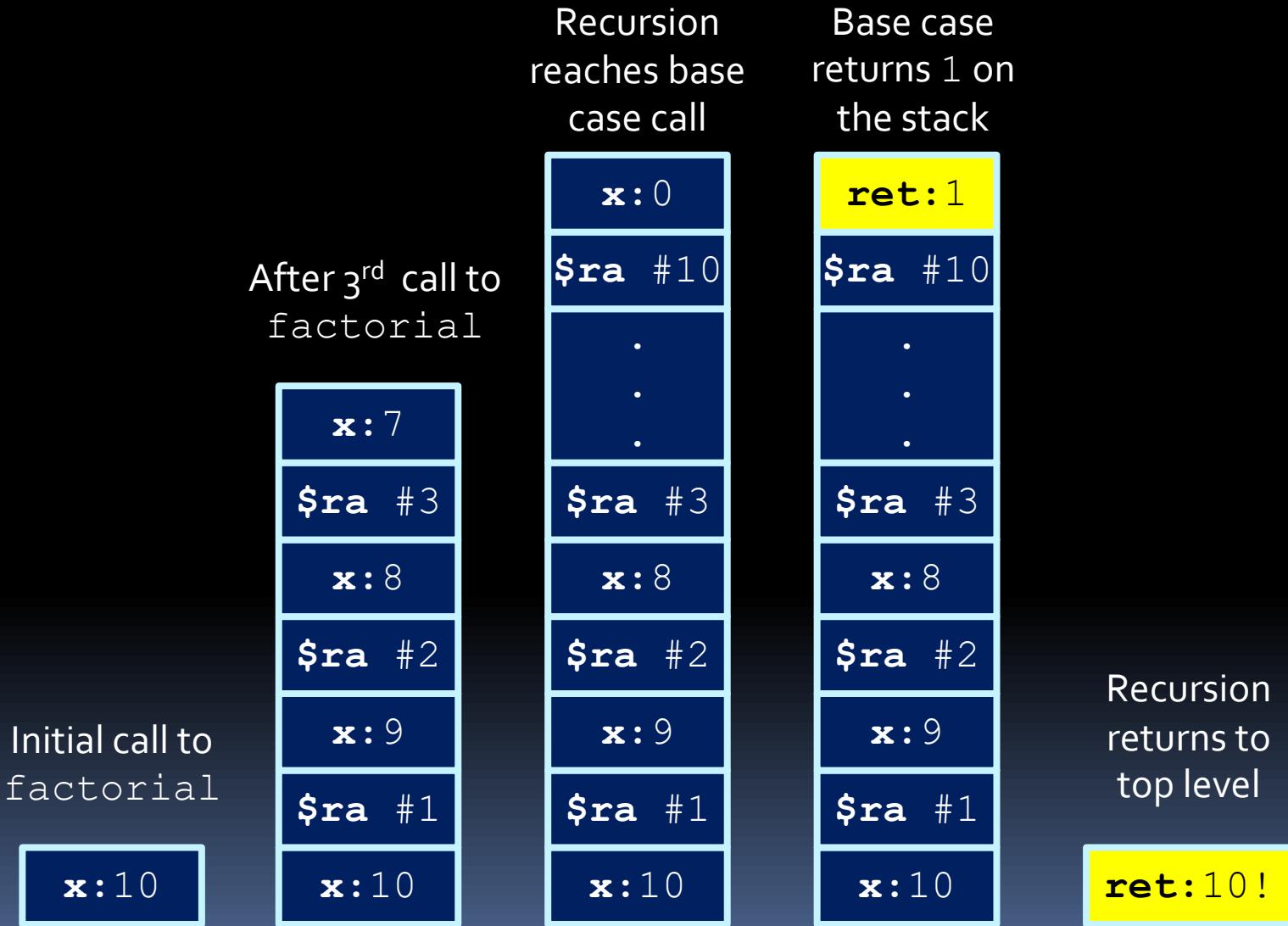
```
int fact (int x) {  
    if (x==0)  
        return 1;  
    else  
        return x*fact(x-1);  
}
```

(continued from part 1 - returning from recursive call)

```
lw      $t2, 0($sp)          # pop return value  
addi   $sp, $sp, 4           # from stack  
lw      $ra, 0($sp)          # restore return  
addi   $sp, $sp, 4           # address value  
lw      $t0, 0($sp)          # restore x value  
addi   $sp, $sp, 4           # for this call  
mult   $t0, $t2              # x*fact(x-1)  
mflo   $v0                  # fetch product  
addi   $sp, $sp, -4          # push n! result  
sw      $v0, 0($sp)          # onto stack  
jr      $ra                  # return to caller
```

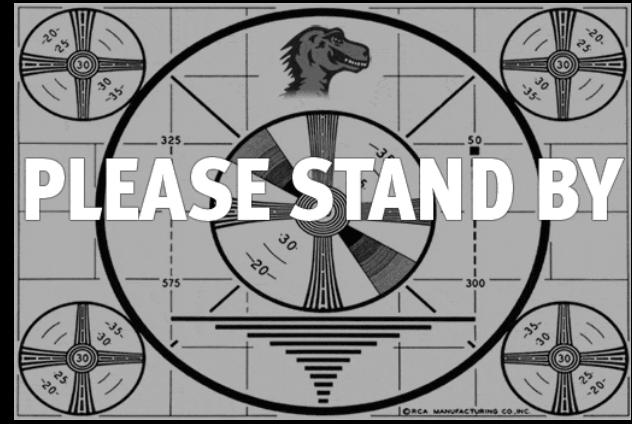
- Remember: `jal` always stores the next address location into `$ra`, and `jr` returns to that address.

Factorial stack view



A note on interrupts / exceptions

- **Interrupts** take place when an **external** event requires a change in execution.
 - **Exception** examples: arithmetic overflow, undefined instructions, division by zero. All **internal** to the processor.
 - **Interrupt** examples: key pressed, a printout finished, a network packet arrived.
 - **Interrupts** signaled by an **external** input wire, usually checked at the end of each instruction.



Handling interrupts/ exceptions

- ISR: Interrupt service routine
 - The piece of code in the OS that handles the interrupt and serves the interrupting device or the exception event.
- Similar to a function-call, but with a major difference: You never know when it happens!
 - Interrupts: are caused by external event
 - Exceptions: are thrown by instructions, but you (usually) don't know at the time of writing the program.
- Thus, extra things should be done by HW (processor)
 - Disabling the interrupts
 - Saving return address (and some other registers)
 - Jumping to the ISR
- You can view it like a function-call by hardware!

A note on interrupts

- Interrupts can be handled in two general ways:
 - **Single handler:** (Interrupt source identification done in SW) The processor branches to the address of interrupt handling code, which begins a sequence of instructions that check the cause of the exception. This branches to handler code sections, depending on the type of exception encountered.
→ This is what MIPS uses.
 - **Vectorized handling:** (Interrupt source identification done in HW) The processor can branch to a different address for each type of exception. Each exception address is separated by only one word. A jump instruction is placed at each of these addresses for the handler code for that exception.

Interrupt handling

- In the case of single-handler interrupt handling, the handler (SW) checks the value in the **cause register** (see table) and jumps to corresponding exception handler code.
 - If the original program can resume afterwards, this interrupt handler returns to program by calling `rfe` instruction.
 - Otherwise, the stack contents are dumped and execution will continue elsewhere. (i.e. back to OS)
- | | |
|-------------|---|
| 0 (INT) | external interrupt. |
| 4 (ADDRL) | address error exception (load or fetch) |
| 5 (ADDRS) | address error exception (store). |
| 6 (IBUS) | bus error on instruction fetch. |
| 7 (DBUS) | bus error on data fetch |
| 8 (Syscall) | Syscall exception |
| 9 (BKPT) | Breakpoint exception |
| 10 (RI) | Reserved Instruction exception |
| 12 (OVF) | Arithmetic overflow exception |