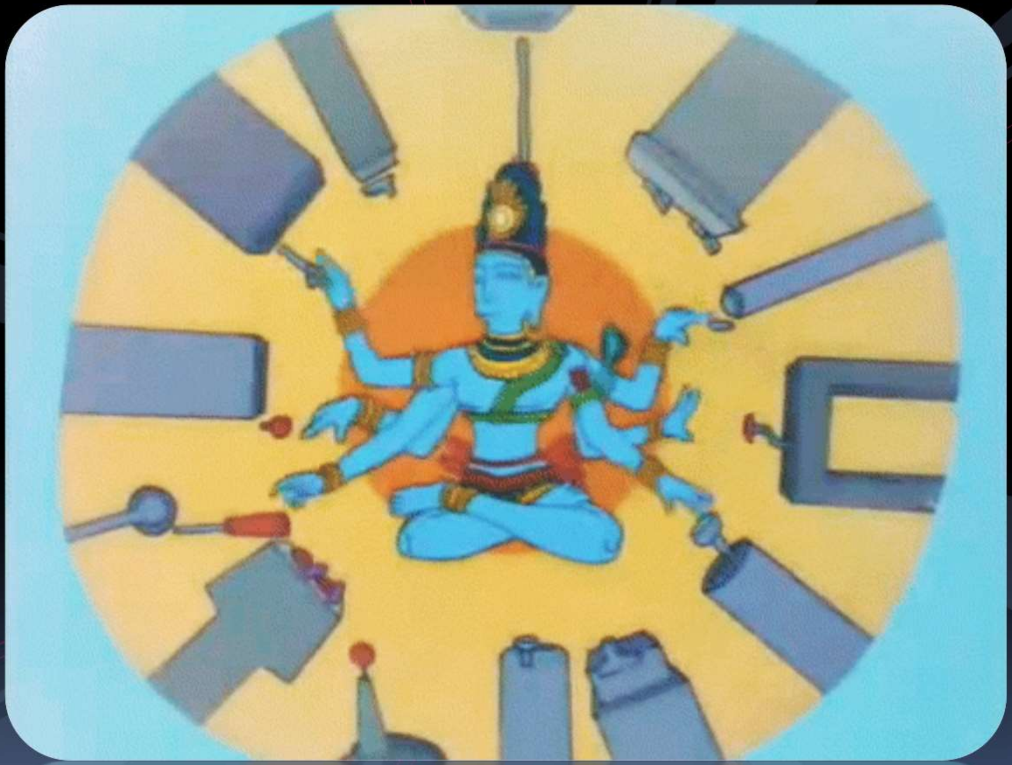
Assembly Language

* Created with contributions by Myrto Papadopoulou and Frank Plavec.

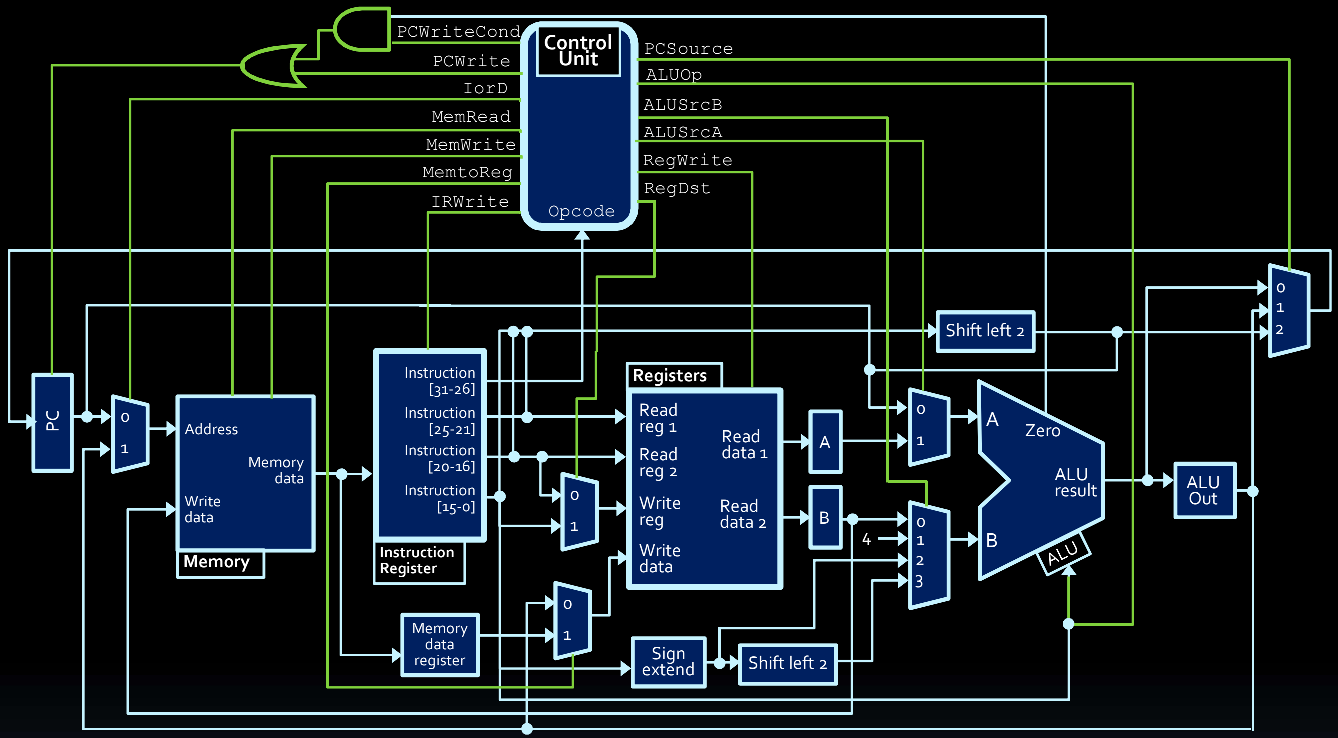
Programming the processor

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

Controlling the Datapath



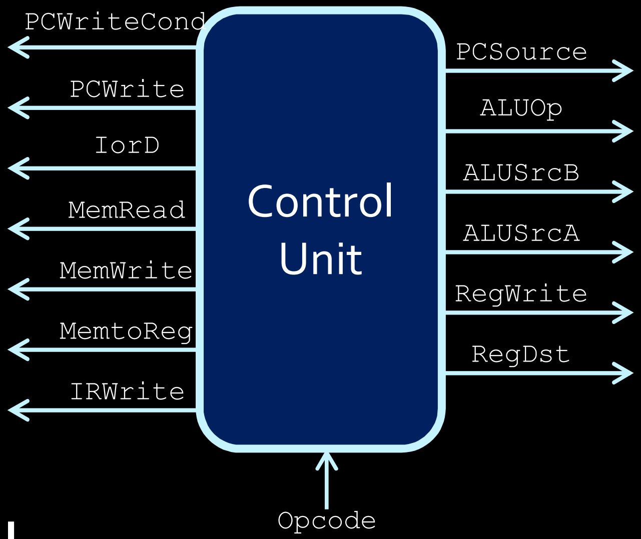
MIPS Datapath



- So, how do we do the following?
  - Increment the PC to the next instruction position.
  - Store $t1 + 12 into the PC.
  - Assuming that register $t3 is storing a valid memory address, fetch the data from that location in memory and store it in $t5.

Controlling the signals

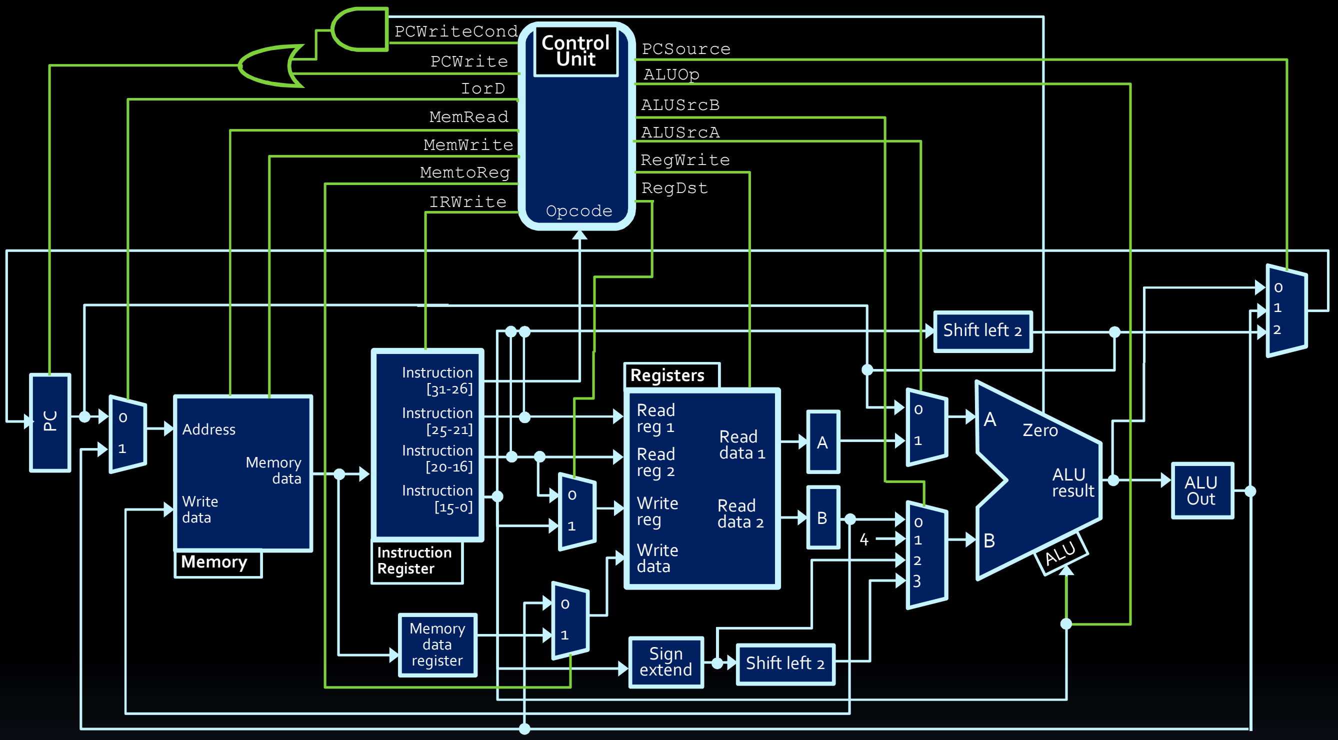
- Need to understand the role of each signal, and what value they need to have in order to perform the given operation.
- So, what's the best approach to make this happen?



Basic approach to datapath

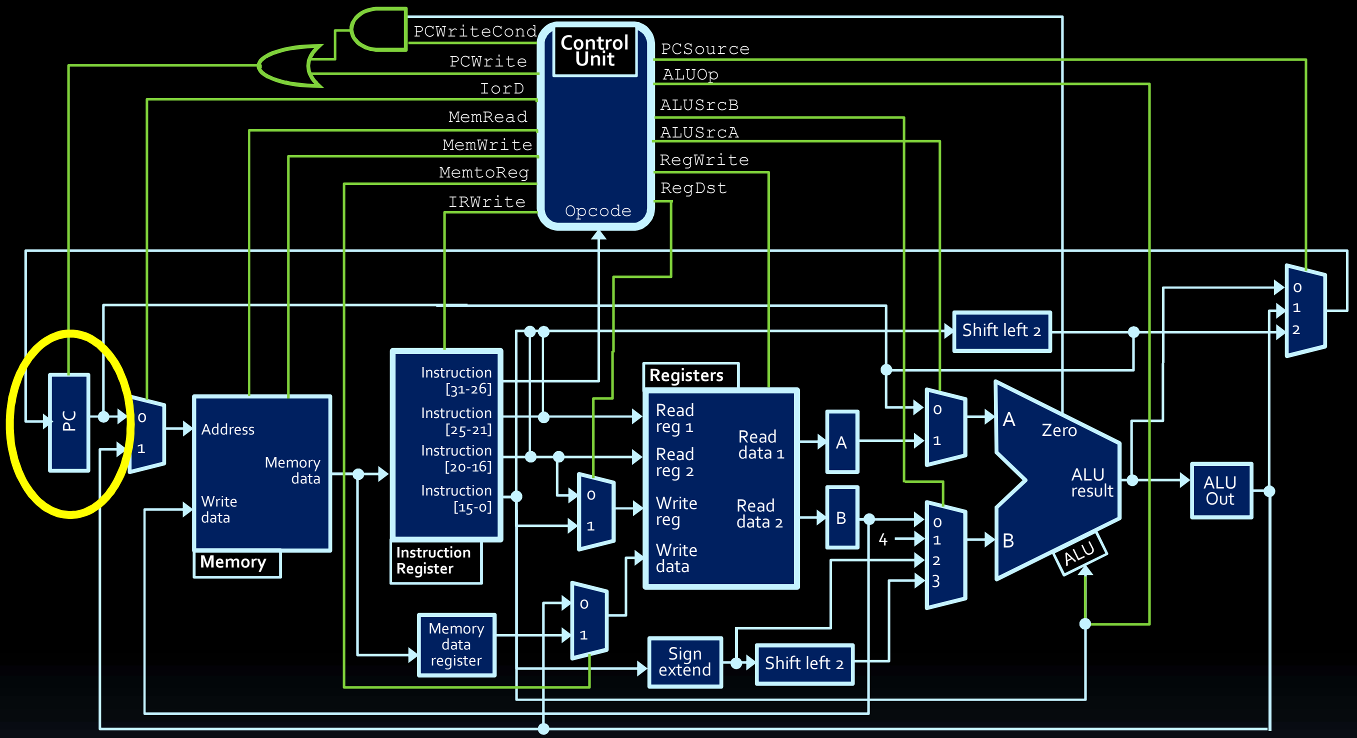
1. Figure out the data source(s) and destination.
2. Determine the path of the data.
3. Deduce the signal values that cause this path:
   - a) Start with `Read` & `Write` signals (at most one can be high at a time).
   - b) Then, mux signals along the data path.
   - c) Non-essential signals get an `X` value.

Example #1: Incrementing PC



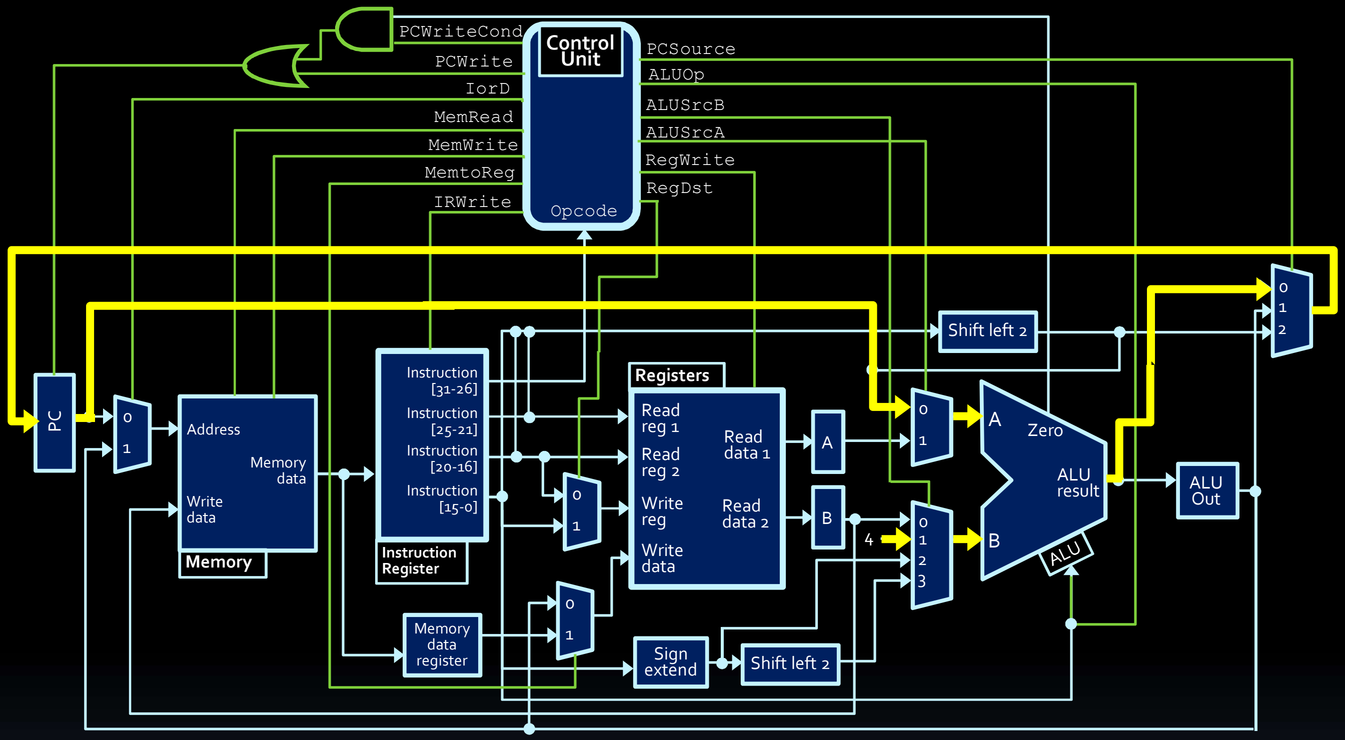
- Given the datapath above, what signals would the control unit turn on and off to increment the program counter by 4?

Example #1: Incrementing PC



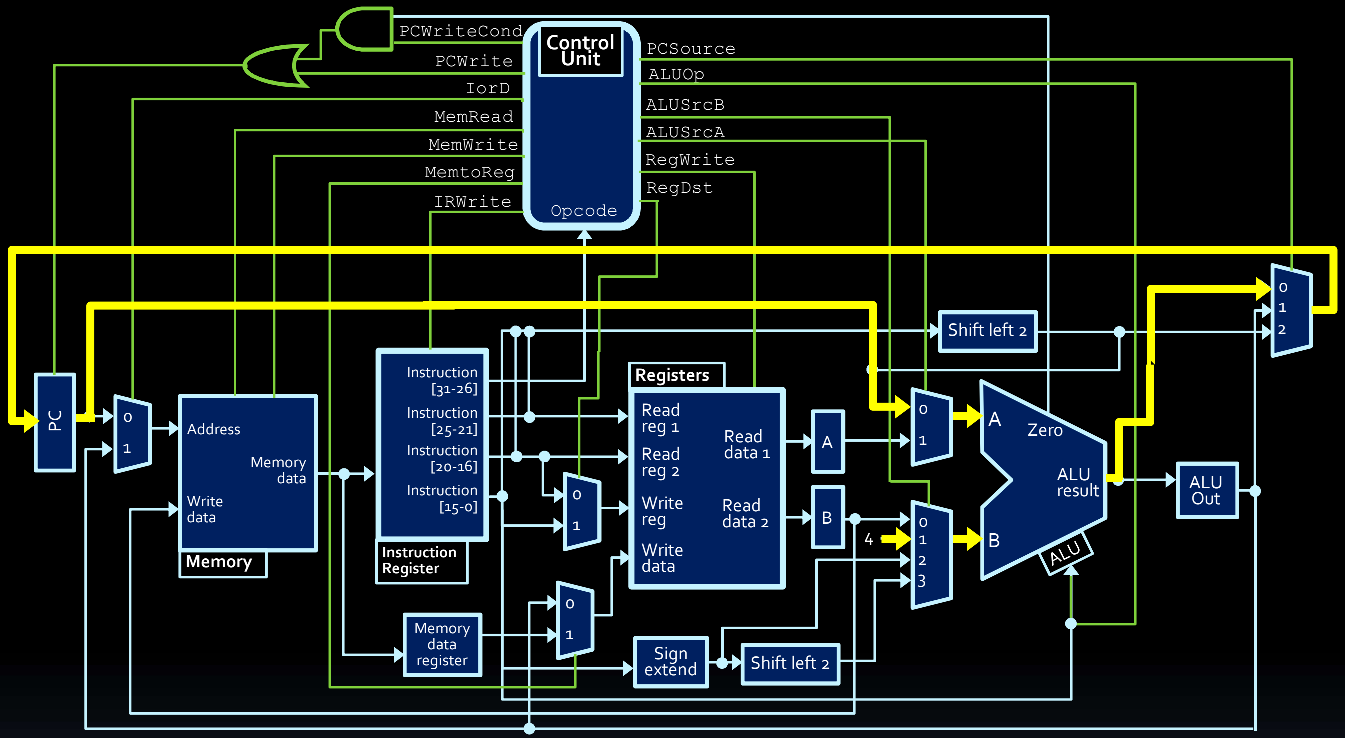
- Step #1: Determine data source and destination.
  - Program counter provides source,
  - Program counter is also destination.

Example #1: Incrementing PC



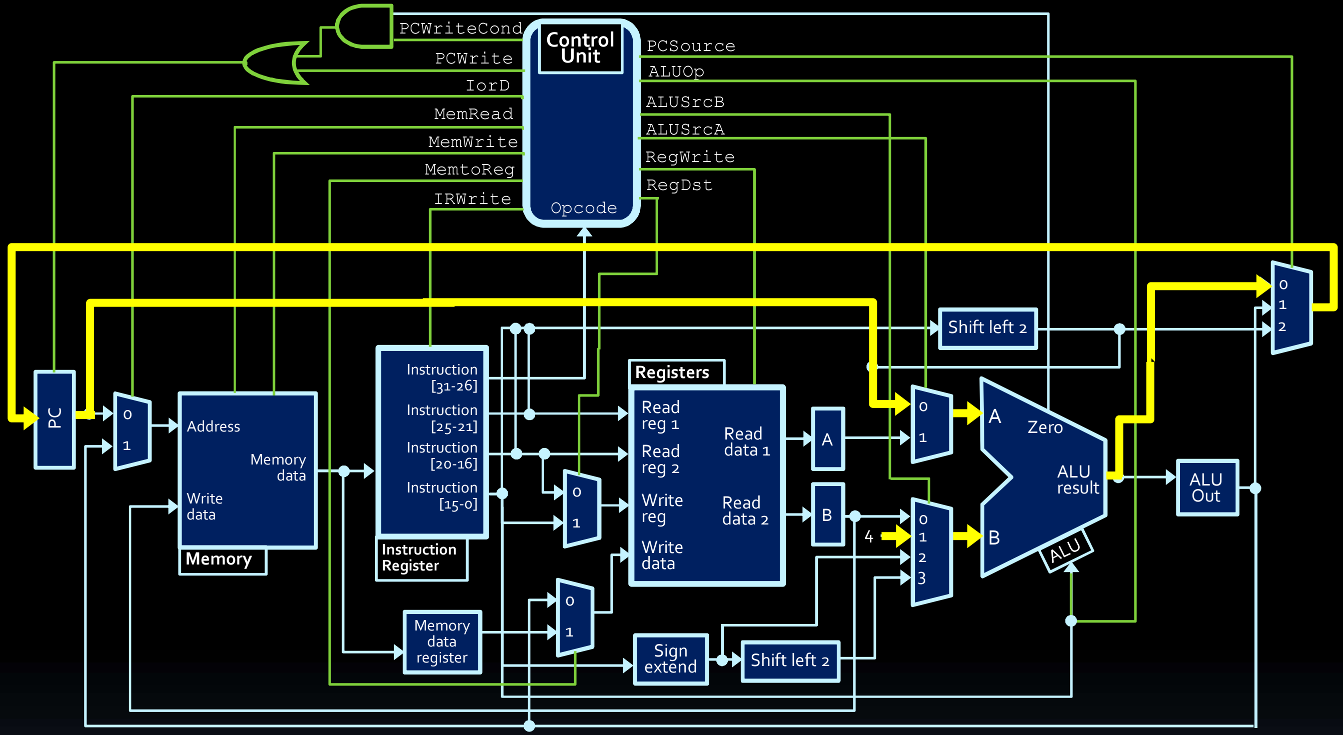
- Step #2: Determine path for data
  - Operand A for ALU: Program counter
  - Operand B for ALU: Literal value 4
  - Destination path: Through mux, back to PC

Example #1: Incrementing PC



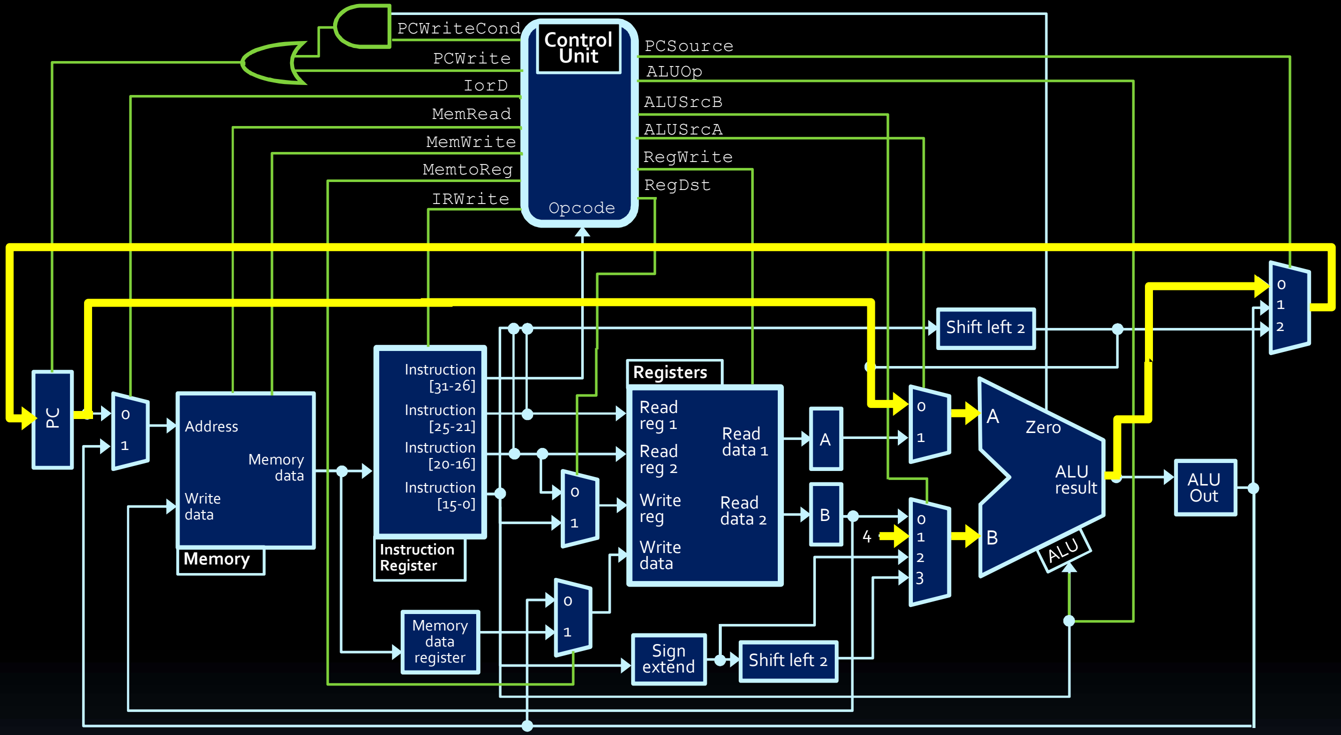
- Setting signals for this datapath:
  1. Read & Write signals:
     - `PCWrite` is high, all others are low.

Example #1: Incrementing PC



- Setting signals for this datapath:
  2. Mux signals:
     - PCSource is 0, AlUSrcA is 0, ALUSrcB is 1
     - all others are “don’t cares”.

Example #1: Incrementing PC



- Other signals for this datapath:
  - ALUOp is 001 (from chart on Slide 15 of Processor notes)
  - PCWriteCond is X when PCWrite is 1
    - Otherwise it is 0 except when branching.

Example #1 (final signals)

- PCWrite = 1
- PCWriteCond = X
- IorD = X
- MemRead = 0
- MemWrite = 0
- MemToReg = X
- IRWrite = 0

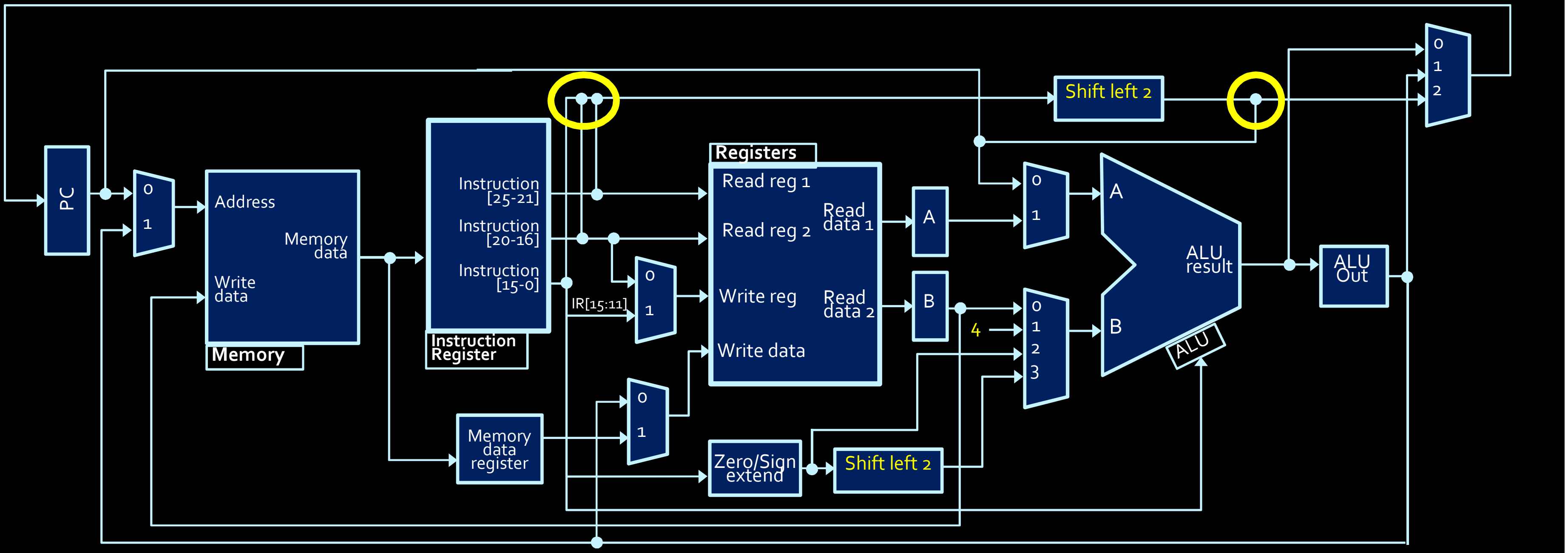
- PCSource = 0
- ALUOp = 001
- ALUSrcA = 0
- ALUSrcB = 01
- RegWrite = 0
- RegDst = X

When is PC incremented?

- This depends on the implementation.
    - Commonly done during decode stage, since the ALU is idle at that time.
        - Other implementations have a separate adder component just to update the PC.
- Key to remember:
    - Every instruction needs to update PC!
    - Otherwise the processor will always execute the same instruction over and over again…

MIPS datapath – Things to note

- In several spots, multiple inputs (each <32 bits) contribute to a single output. This happens when a 32-bit value is composed through concatenating the smaller components together.
  - Example: jump instructions.
  - 26 bits from instruction are shifted left, and filled out with the leftmost 4 bits from the program counter.

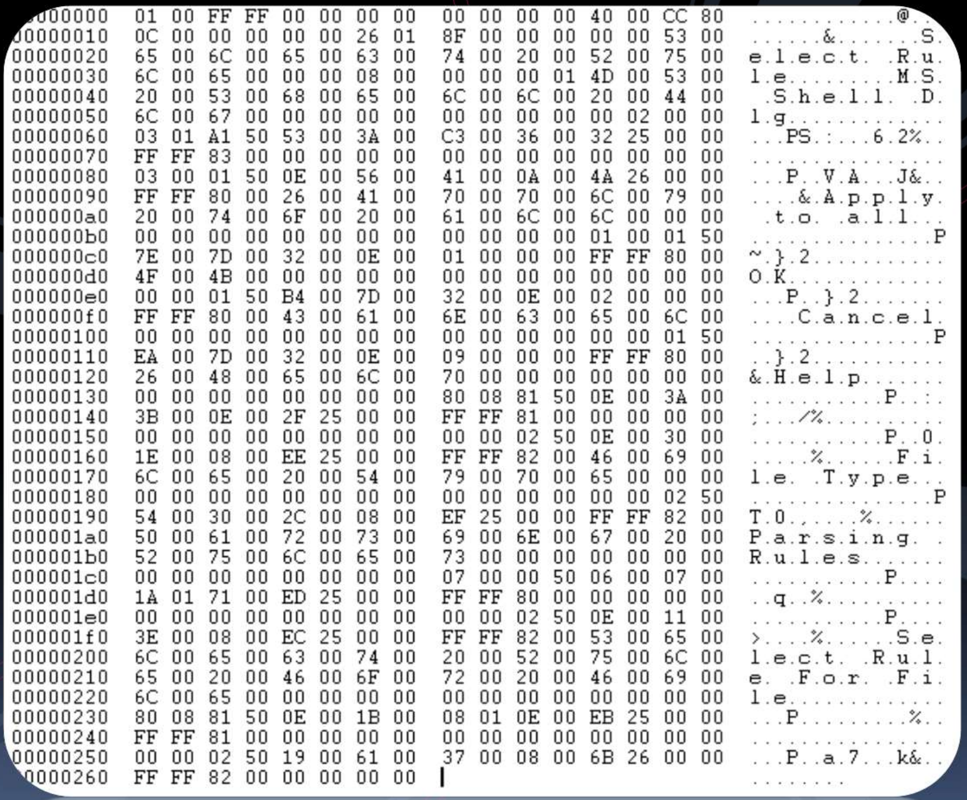


The remaining 26 bits

- The control unit sends these signals to the processor, based on the instruction's opcode.
  - So what is the rest of the instruction used for?

    → Providing values that the instruction needs.
- Examples:
  - Register operations → instruction provides addresses of source and destination registers.
  - Jump operations → instruction provides offset to be added to PC to execute jump.
- How are these are encoded in the instruction?

Machine Code Instructions



```
00000000  01 00 FF FF 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 00 02 00 00  l.g.............
00000060  03 01 A1 50 53 00 3A 00  C3 00 36 00 32 25 00 00  ...PS.:...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 56 00  41 00 0A 00 4A 26 00 00  ...P..V.A...J&..
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 00 01 00 01 50  ...............P
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 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 00 00 01 50  ...............P
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.......
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  ;.../%..........
00000150  00 00 00 00 00 00 00 00  00 00 02 50 0E 00 30 00  ...........P..0.
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 00 02 50  ...............P
00000190  54 00 30 00 2C 00 08 00  EF 25 00 00 FF FF 82 00  T.0.,....%......
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..%..........
000001e0  00 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.
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 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 FF 82 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 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
```

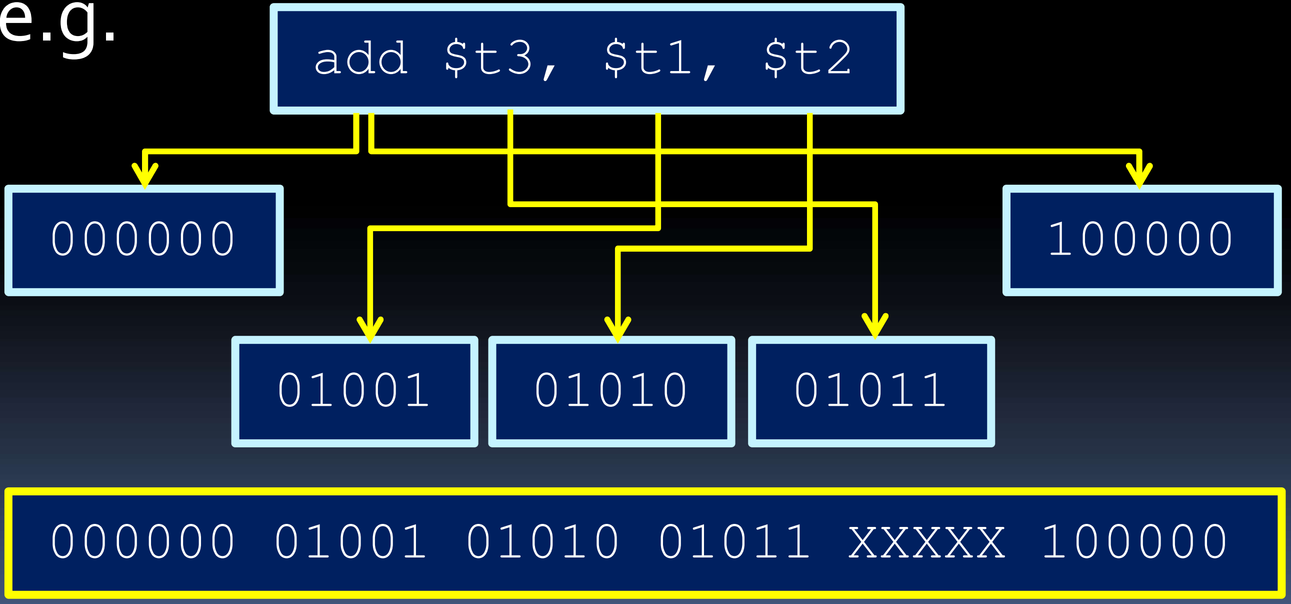
  - Machine code instruction:

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

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

Filling in the blanks

- When writing machine code instructions (or interpreting them), we need to know how to encode (or decode) the operation to perform and the register values to operate on.
- e.g.



Register operations

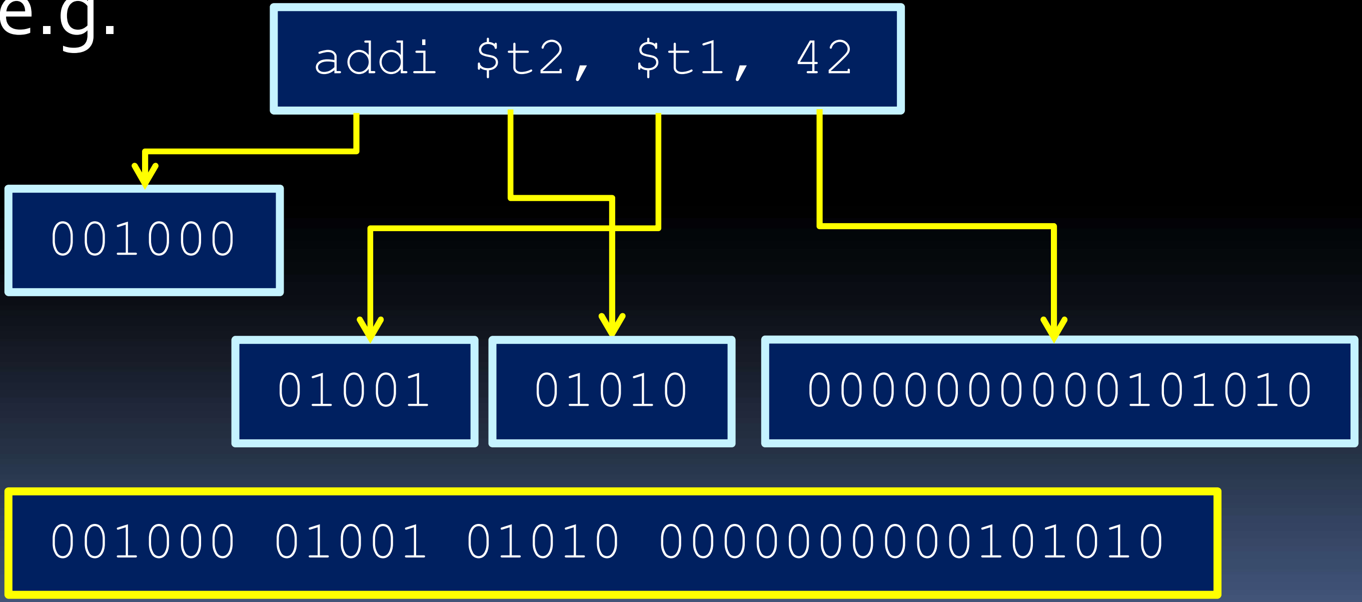
- The `add` instruction is one of many operations that processes two register values and stores the result in a third.
  - This is called an R-type instruction.
  - Any operations whose inputs and outputs are all registers are called R-type, even if they involve less than three registers.
    - e.g. the `jr` 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.
    - Every operation operates on register data in some way.
- MIPS provides 32 registers.
    - 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.

I-type instructions

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



```
addi $t2, $t1, 42
```

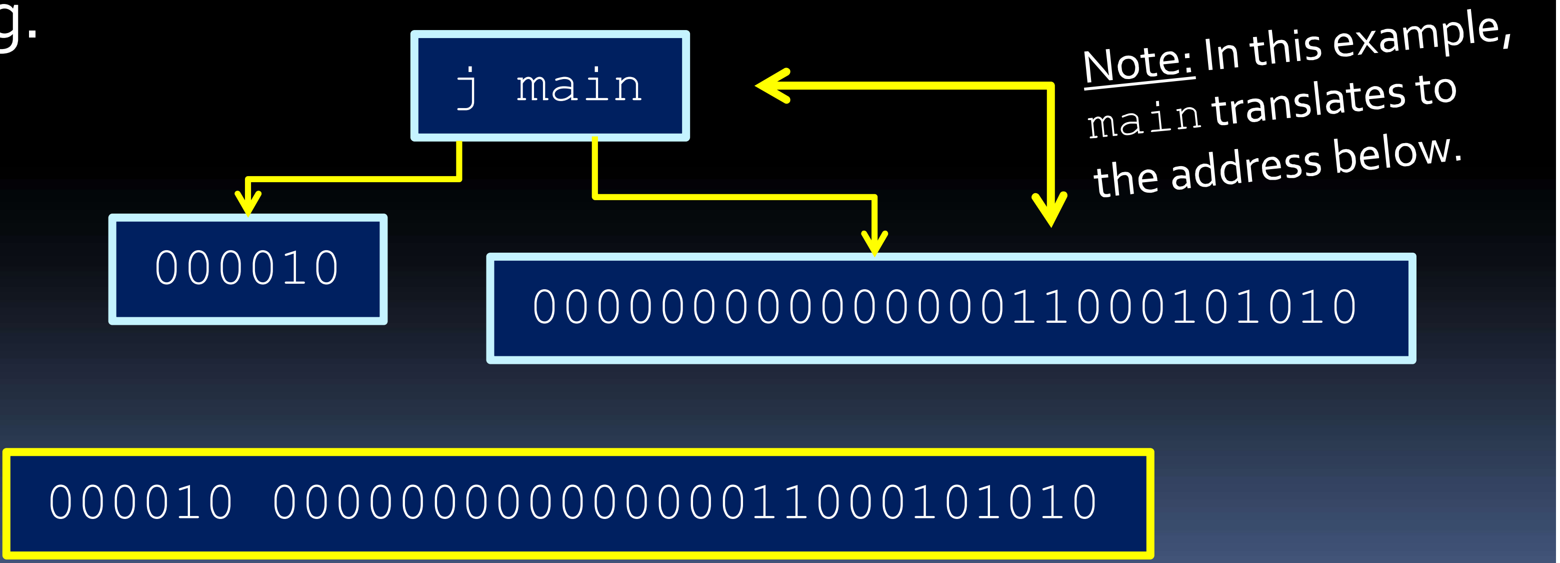
```
001000
```

```
01001    01010    0000000000101010
```

```
001000 01001 01010 0000000000101010
```

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.



Note: In this example, main translates to the address below.

Review: MIPS instruction types

- R-type:

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- I-type:

opcode	rs	rt	immediate
6	5	5	16

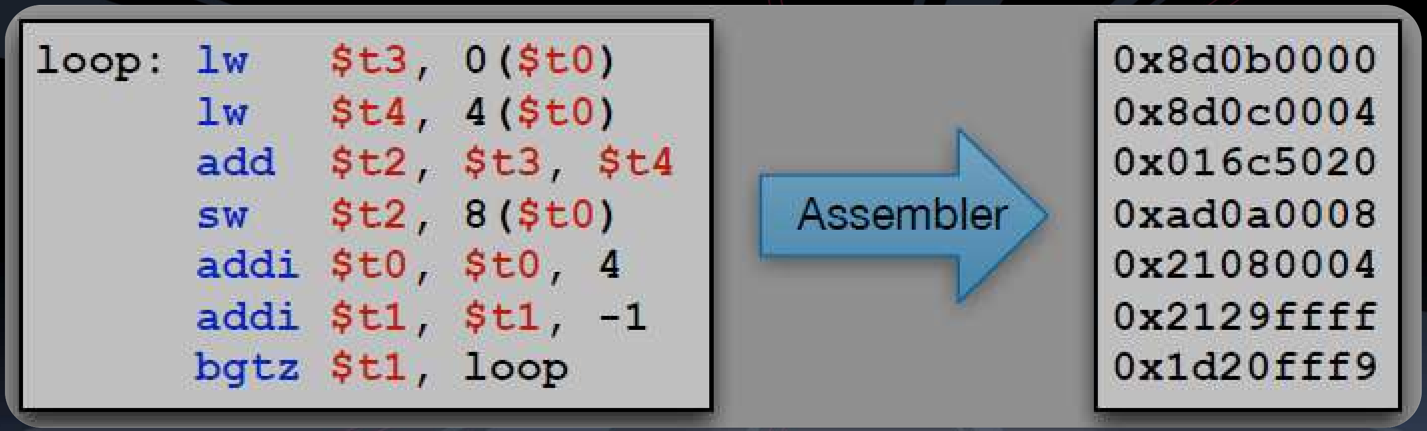
- J-type:

opcode	address
6	26

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).

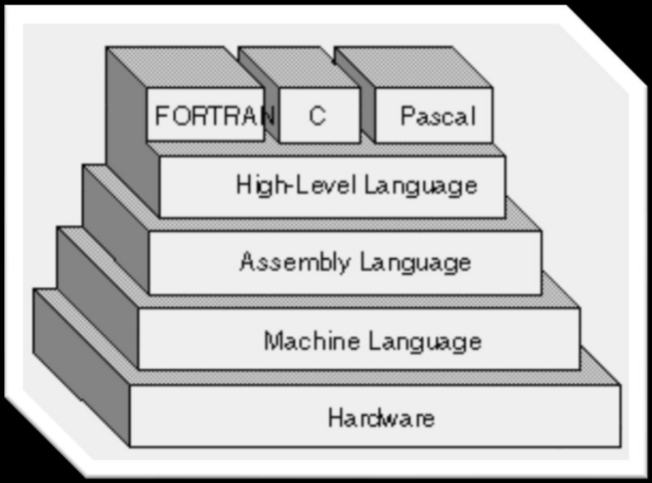
Assembly Language Overview



```
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
```

```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```

Assembly language

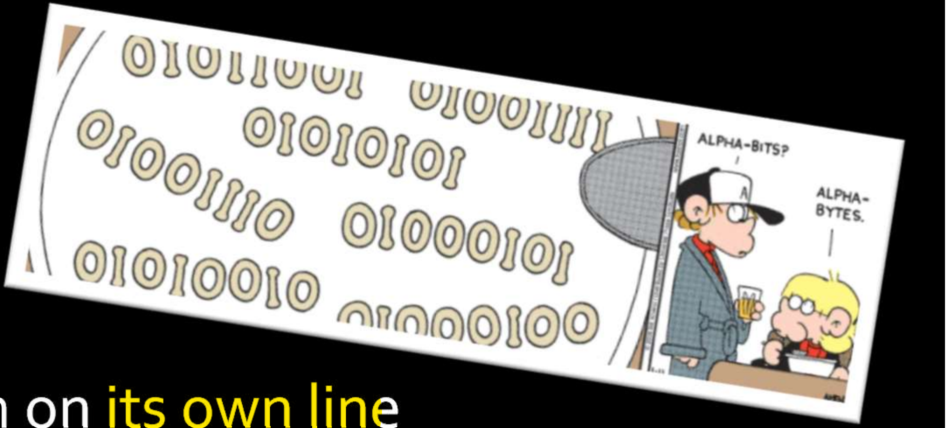


- Assembly language is the lowest-level language that you'll ever program in.
- Many compilers translate their high-level program commands into assembly commands, which are then converted into machine code and used by the processor.
- Note: There are multiple types of assembly language, especially for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



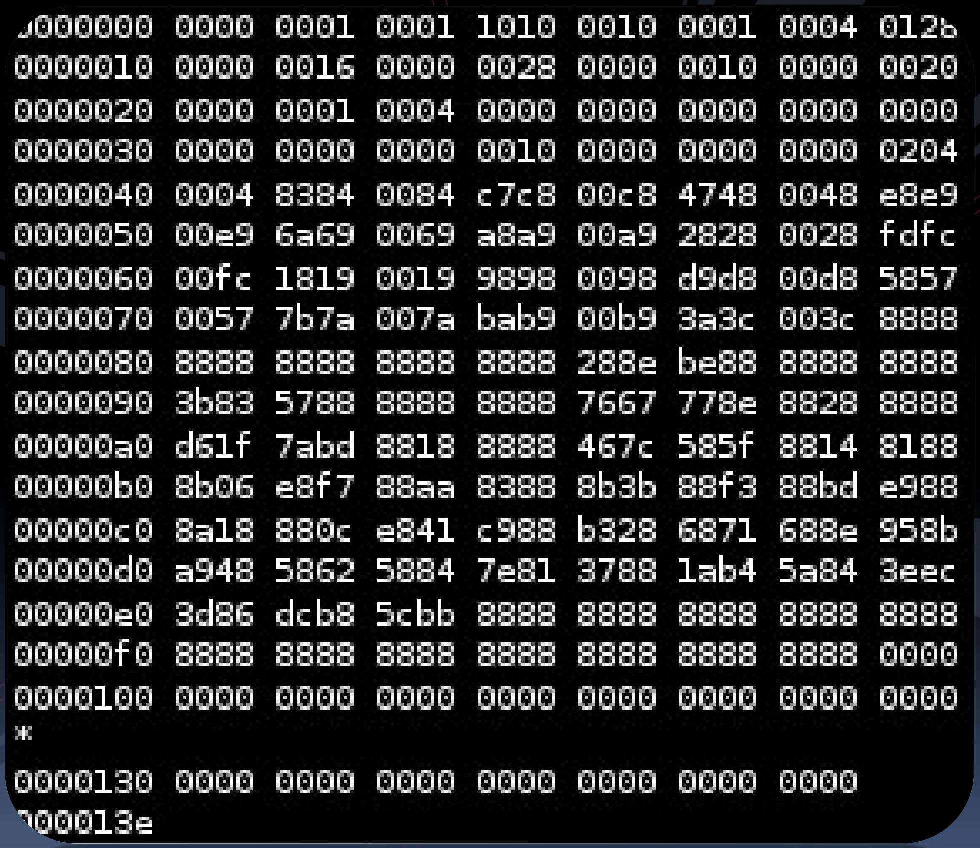
- Things to note about MIPS instructions:
  - Instruction are written `<instr> <parameters>`
  - Each instruction is written on its own line
  - All instructions are 32 bits (4 bytes) long
  - Instruction addresses are measured in bytes, starting from the instruction at address `0`.
    - Therefore, 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 statement s	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 statement s	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



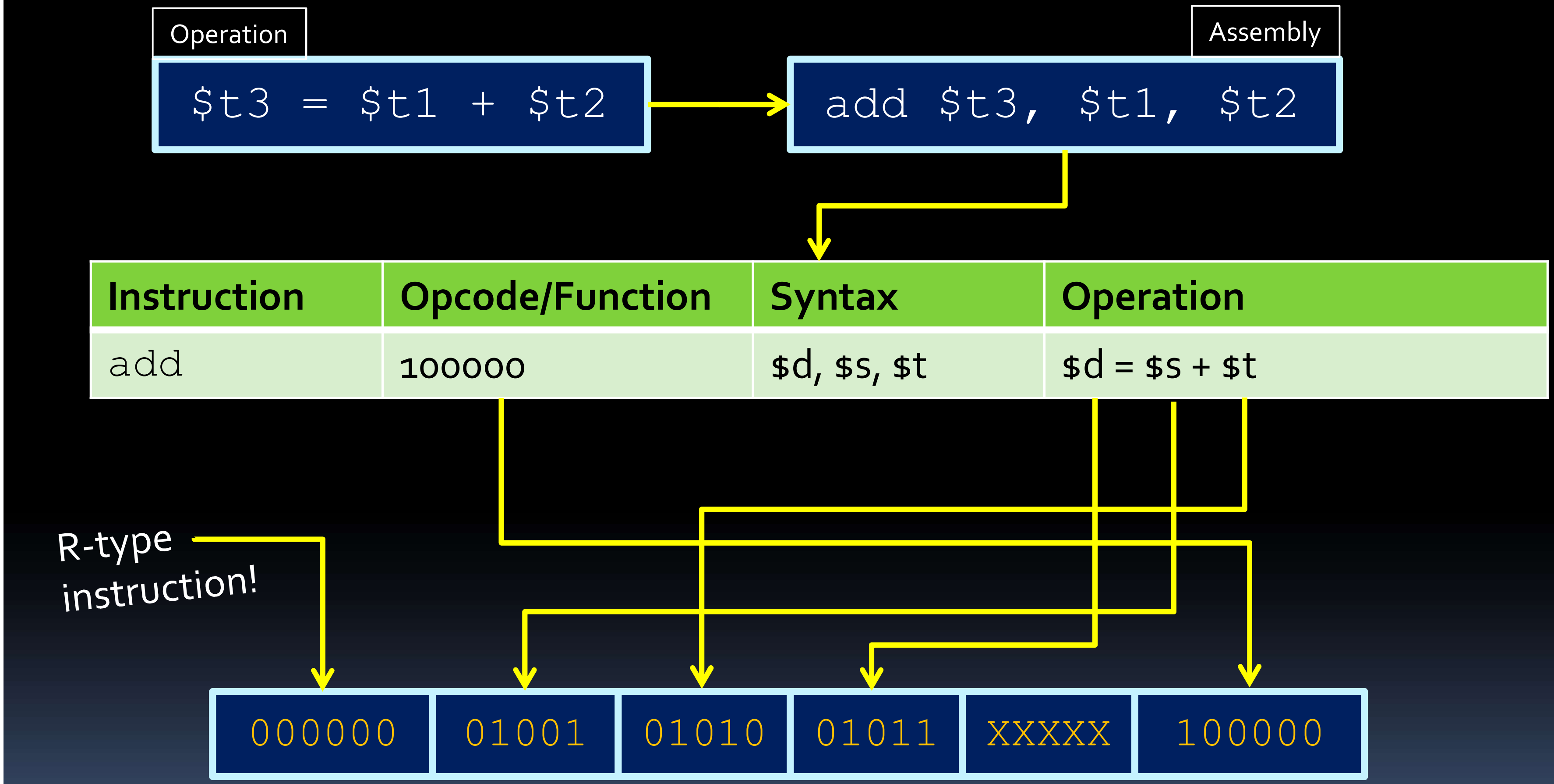
```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 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 dcb8 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
000013e
```

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



Operation

`$t3 = $t1 + $t2`

Assembly

`add $t3, $t1, $t2`

Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t

R-type instruction!

000000	01001	01010	01011	XXXXX	100000

R-type vs I-type arithmetic

R-Type

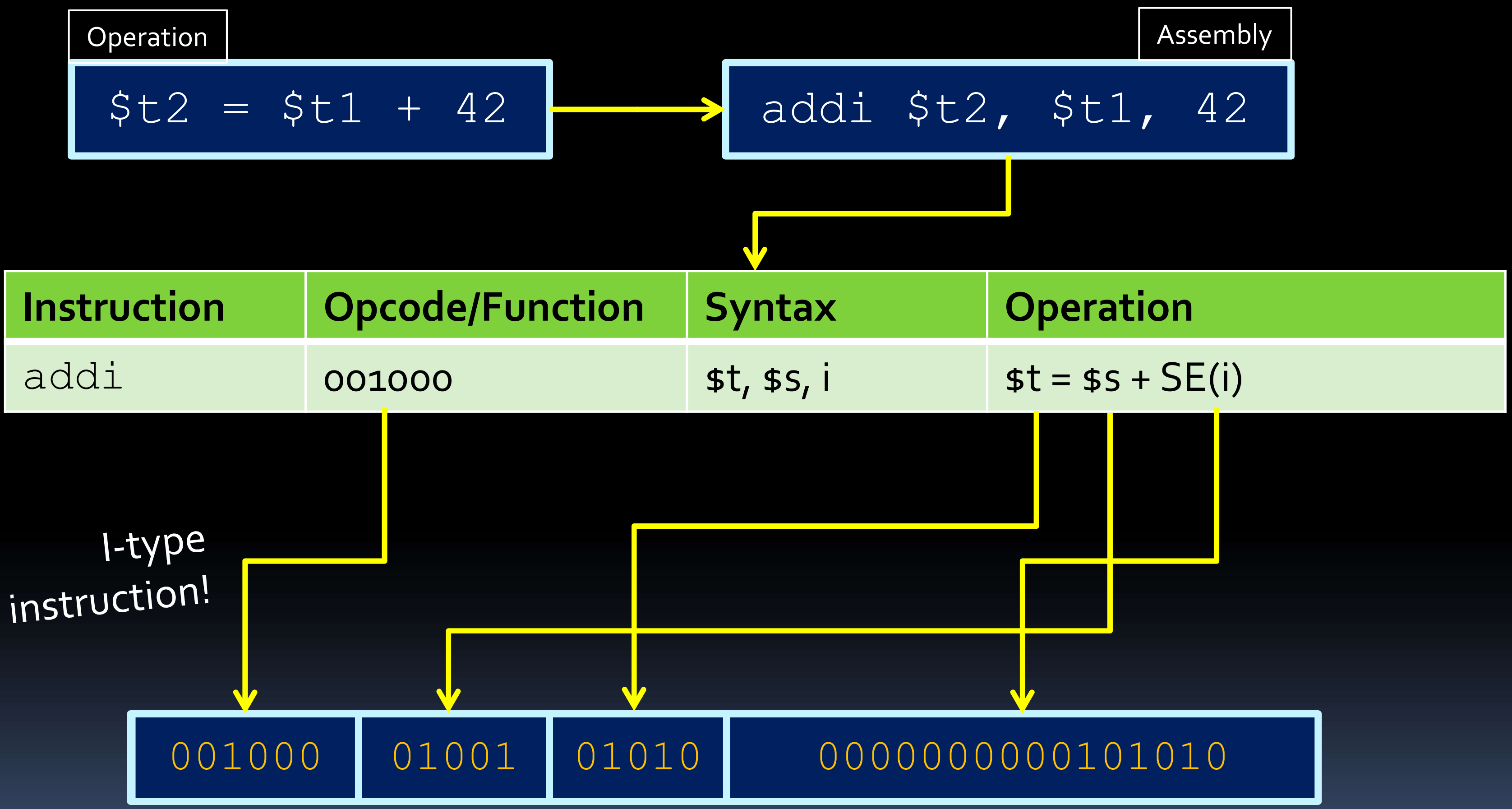
- add, addu
- div, divu
- mult, multu
- sub, subu

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



Operation

```
$t2 = $t1 + 42
```

Assembly

```
addi $t2, $t1, 42
```

Instruction	Opcode/Function	Syntax	Operation
addi	001000	$t, $s, i	$t = $s + SE(i)

I-type instruction!

```
001000 | 01001 | 01010 | 0000000000101010
```

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 << a
`sllv`	000100	$d, $t, $s	$d = $t << $s
`sra`	000011	$d, $t, a	$d = $t >> a
`srav`	000111	$d, $t, $s	$d = $t >> $s
`srl`	000010	$d, $t, a	$d = $t >>> a
`srlv`	000110	$d, $t, $s	$d = $t >>> $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 example

- Fibonacci sequence in assembly code:

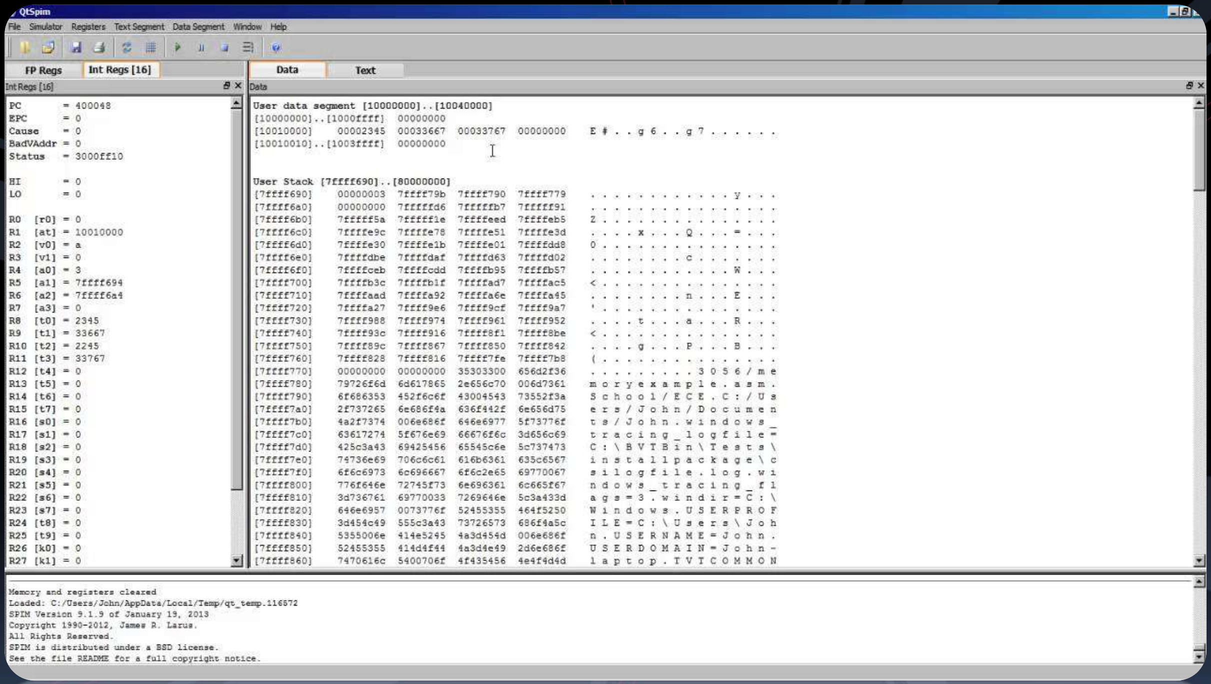
```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
#
FIB:  addi $t3, $zero, 10     # initialize n=10
      addi $t4, $zero, 1      # initialize f1=1
      addi $t5, $zero, -1     # initialize f2=-1
LOOP: beq $t3, $zero, END     # done loop if n==0
      add $t4, $t4, $t5       # f1 = f1 + f2
      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
```

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: QtSpim



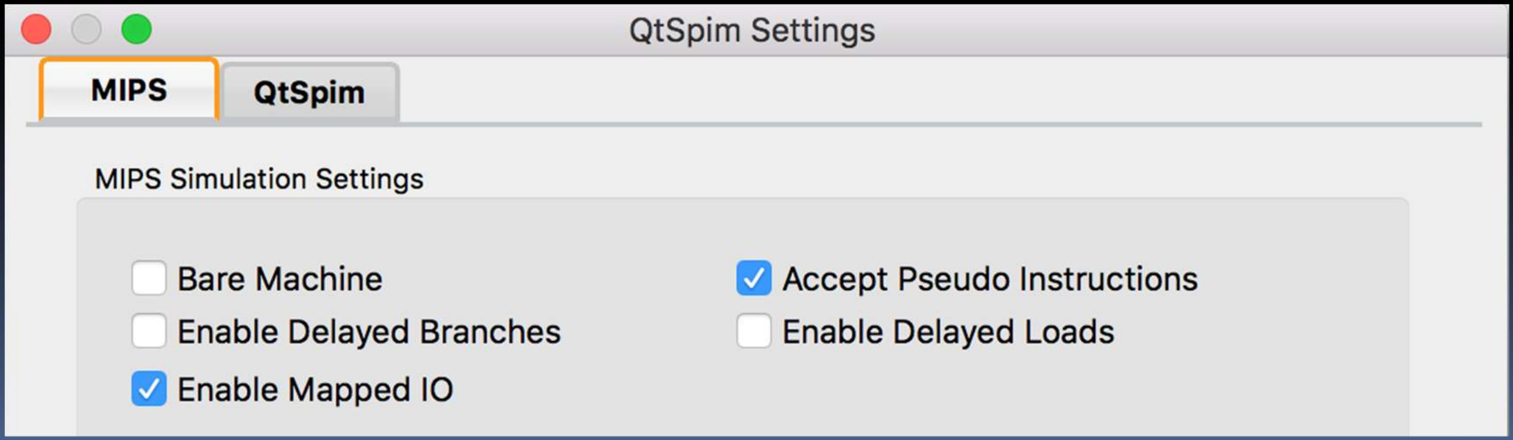
```
PC       = 400048
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 10010000
R2  [v0] = a
R3  [v1] = 0
R4  [a0] = 3
R5  [a1] = 7ffff694
R6  [a2] = 7ffff6a4
R7  [a3] = 0
R8  [t0] = 2345
R9  [t1] = 33667
R10 [t2] = 2245
R11 [t3] = 33767
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
```

```
User data segment [10000000]..[10040000]
[10000000]..[1000ffff]  00000000
[10010000]    00002345  00033667  00033767  00000000    E # . . g 6 . . g 7 . . . . . .
[10010010]..[1003ffff]  00000000


User Stack [7ffff690]..[80000000]
[7ffff690]    00000003  7ffff79b  7ffff790  7ffff779    . . . . . . . . . . . . y . . .
[7ffff6a0]    00000000  7fffffd6  7fffffb7  7fffff91    . . . . . . . . . . . . . . . .
[7ffff6b0]    7fffff5a  7fffff1e  7ffffeed  7ffffeb5    Z . . . . . . . . . . . . . . .
[7ffff6c0]    7ffffe9c  7ffffe78  7ffffe51  7ffffe3d    . . . . x . . . Q . . . = . . .
[7ffff6d0]    7ffffe30  7ffffe1b  7ffffe01  7ffffdd8    0 . . . . . . . . . . . . . . .
[7ffff6e0]    7ffffdbe  7ffffdaf  7ffffd63  7ffffd02    . . . . . . . . c . . . . . . .
[7ffff6f0]    7ffffceb  7ffffcdd  7ffffb95  7ffffb57    . . . . . . . . . . . . W . . .
[7ffff700]    7ffffb3c  7ffffb1f  7ffffad7  7ffffac5    < . . . . . . . . . . . . . . .
[7ffff710]    7ffffaad  7ffffa92  7ffffa6e  7ffffa45    . . . . . . . . n . . . E . . .
[7ffff720]    7ffffa27  7ffff9e6  7ffff9cf  7ffff9a7    ' . . . . . . . . . . . . . . .
[7ffff730]    7ffff988  7ffff974  7ffff961  7ffff952    . . . . t . . . a . . . R . . .
[7ffff740]    7ffff93c  7ffff916  7ffff8f1  7ffff8be    < . . . . . . . . . . . . . . .
[7ffff750]    7ffff89c  7ffff867  7ffff850  7ffff842    . . . . g . . . P . . . B . . .
[7ffff760]    7ffff828  7ffff816  7ffff7fe  7ffff7b8    ( . . . . . . . . . . . . . . .
[7ffff770]    00000000  00000000  35303300  656d2f36    . . . . . . . . . 3 0 5 6 / m e
[7ffff780]    79726f6d  6d617865  2e656c70  006d7361    m o r y e x a m p l e . a s m .
[7ffff790]    6f686353  452f6c6f  43004543  73552f3a    S c h o o l / E C E . C : / U s
[7ffff7a0]    2f737265  6e686f4a  636f442f  6e656d75    e r s / J o h n / D o c u m e n
[7ffff7b0]    4a2f7374  006e686f  646e6977  5f73776f    t s / J o h n . w i n d o w s _
[7ffff7c0]    63617274  5f676e69  66676f6c  3d656c69    t r a c i n g _ l o g f i l e =
[7ffff7d0]    425c3a43  69425456  65545c6e  5c737473    C : \ B V T B i n \ T e s t s \
[7ffff7e0]    74736e69  706c6c61  616b6361  635c6567    i n s t a l l p a c k a g e \ c
[7ffff7f0]    6f6c6973  6c696667  6f6c2e65  69770067    s i l o g f i l e . l o g . w i
[7ffff800]    776f646e  72745f73  6e696361  6c665f67    n d o w s _ t r a c i n g _ f l
[7ffff810]    3d736761  69770033  7269646e  5c3a433d    a g s = 3 . w i n d i r = C : \
[7ffff820]    646e6957  0073776f  52455355  464f5250    W i n d o w s . U S E R P R O F
[7ffff830]    3d454c49  555c3a43  73726573  686f4a5c    I L E = C : \ U s e r s \ J o h
[7ffff840]    5355006e  414e5245  4a3d454d  006e686f    n . U S E R N A M E = J o h n .
[7ffff850]    52455355  414d4f44  4a3d4e49  2d6e686f    U S E R D O M A I N = J o h n -
[7ffff860]    7470616c  5400706f  4f435456  4e4f4d4d    l a p t o p . T V T C O M M O N
```

```
Memory and registers cleared
Loaded: C:/Users/John/AppData/Local/Temp/qt_temp.116572
SPIM Version 9.1.9 of January 19, 2013
Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
```

QtSpim Simulator

- Link to download:
  - http://spimsimulator.sourceforge.net
- MIPS settings in the simulator:
  - Important to **not** have delayed branches or delayed loads selected under Settings.



MIPS Simulation Settings

☐ Bare Machine
☐ Enable Delayed Branches
☑ Enable Mapped IO
☑ Accept Pseudo Instructions
☐ Enable Delayed Loads

QtSpim – Config. Options

- A couple of things to configure:
  - The numerical representations of registers via the "Registers" menu option (decimal, hex, binary).
  - Whether you view user code and/or kernel code. Select Text Segment -> User text.

QtSpim – Quick How To

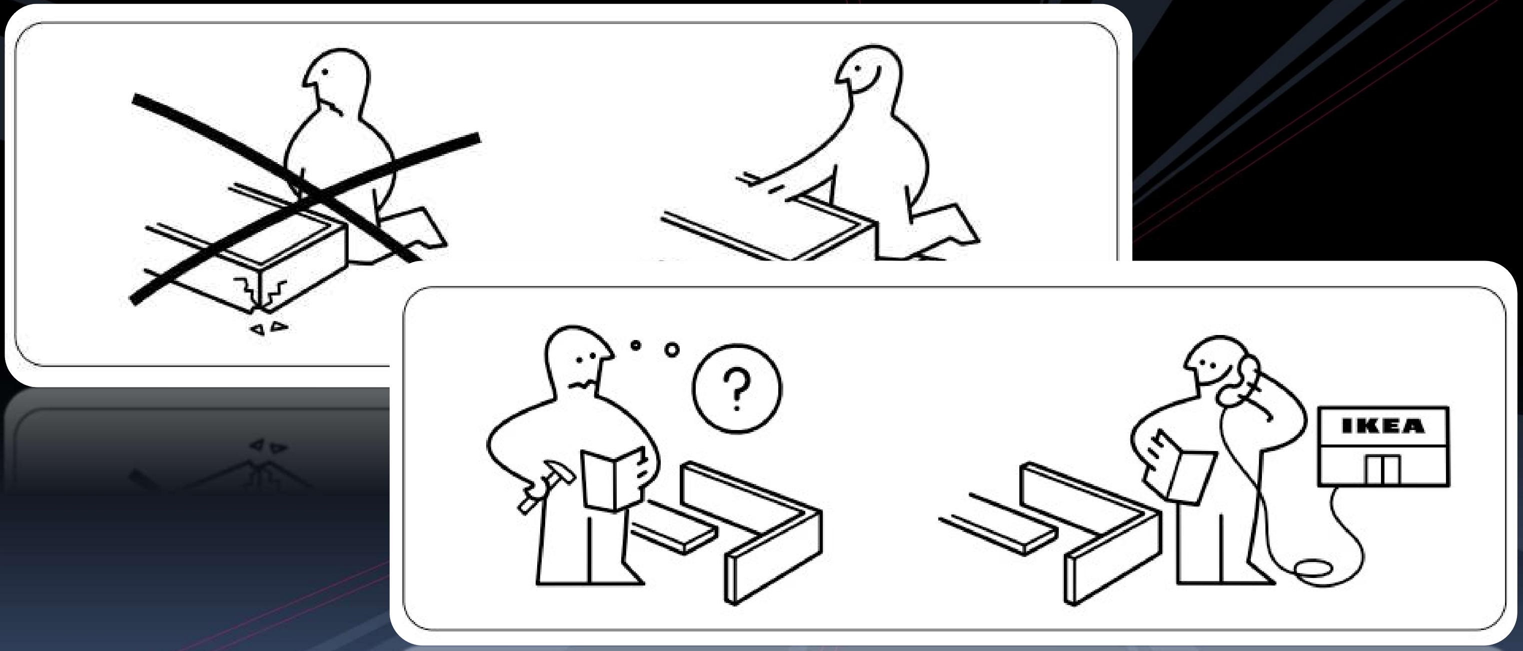
- Write a MIPS program (similar to the ones posted) in any text editor. Save it with `.asm` extension.
- In QtSpim select:
  - File -> Reinitialize and load a file
  - Single step through your program while observing (a) the Int Regs window and (b) the text window (user text).
    - As you step through, the highlighted instruction is the one about to be executed.

QtSpim Help => MIPS reference

- QtSpim help (Help -> View Help) contains
  - "Appendix A (Assemblers, Linkers, and the SPIM Simulator)" from *Patterson and Hennessey, Computer Organization and Design: The Hardware/Software Interface, Third Edition*

  - Useful reference for MIPS R2000 Assembly Language
    - Look at "Arithmetic and Logical Instructions".
    - We will also add other links to Portal under::
      - Course Materials -> General Course Information -> Textbook Readings

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.

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

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

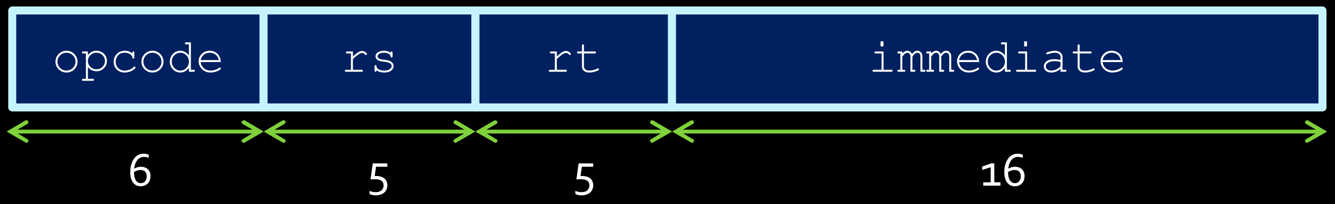
Branch instructions

- Alternate implementation using `bne`:

```
.text

main:   bne $t0, $t1, end     # check if $t0 == $t1
        ...                   # if $t0 == $t1, then
        ...                   # execute these lines

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

- Used to produce `if` statement behaviour.

Branch's immediate (`i`) value



opcode	rs	rt	immediate
6	5	5	16

- Branch statements are I-type instructions.
- The immediate value (`i`) is a 16-bit offset to add to the current instruction if the branch condition is satisfied.
  - 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)
  - 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 the PC is incremented first:

    `i` = (label location - (current PC)) >> 2

  - If the branch offset is calculated first:

    `i` = (label location - (current PC + 4)) >> 2

- For this course, we assume i is computed as:
  - `i` = (label - (current PC)) >> 2
  - Corresponds to the simulator we use for this course (QtSpim) → more on that later.

i in simulation

- Use a simple program in QTSpim 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 QtSpim, 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 `PC+4`
- How far can a processor branch? Are there any constraints?

Jump instructions

Instruction	Opcode/Function	Syntax	Operation
j	000010	label	pc = (pc & 0xF0000000) | (i<<2)
jal	000011	label	$31 = pc+4; pc = (pc & 0xF0000000) | (i<<2)
jalr	001001	$s	$31 = pc+4; pc = $s
jr	001000	$s	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.

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`	001001	$t, $s, i	$t = ($s < SE(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`)

If/Else statements

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

- An 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
```

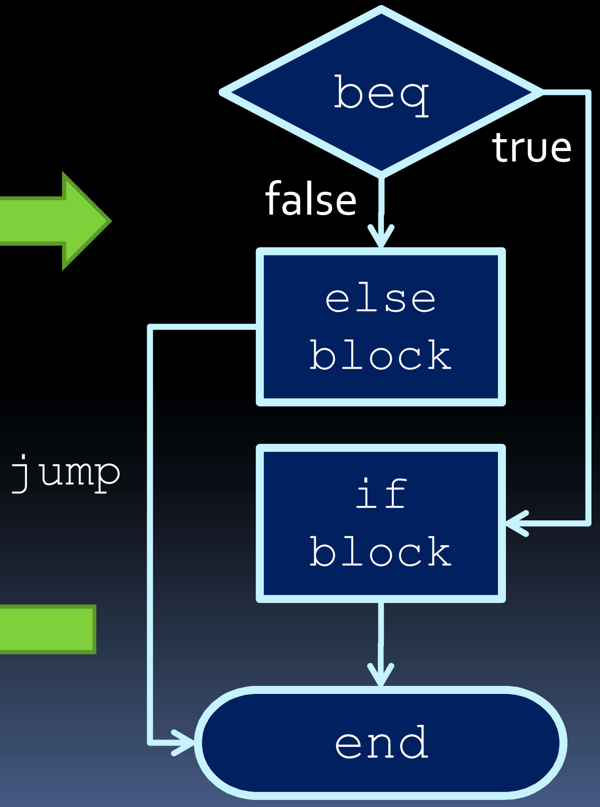
- Or branch on the else condition first:

```
#  $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:

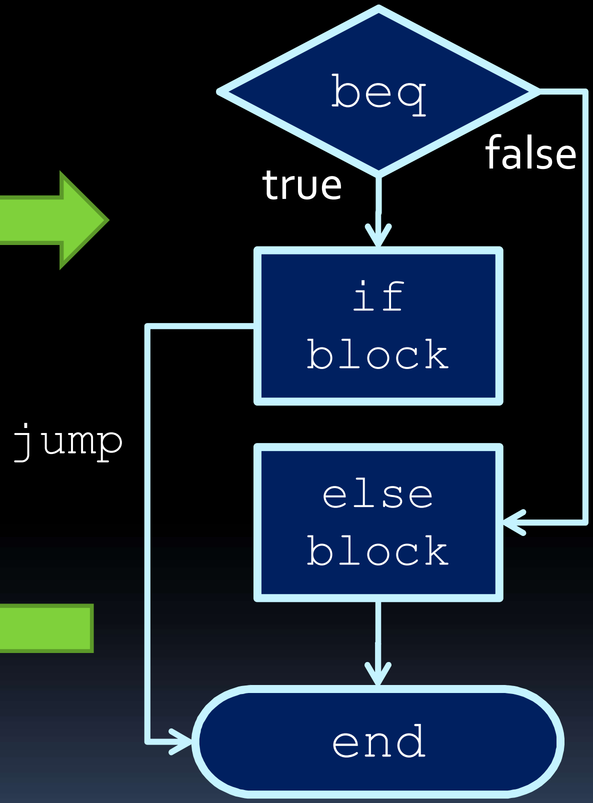
```
if ( i == j )
  i++;
else
  i--;
j += i;
```



```
#  $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 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 $ti, $ti, -1     # else-body: j--
END:   add $t2, $t1, $t3     # j = i + k
```

Multiple if conditions

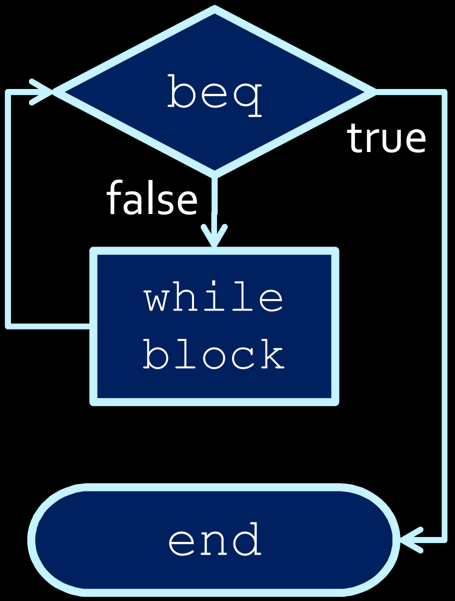
- How would this look if the condition changed?

```
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 ; i<100 ; i++ ) {
    j = j + i;
}
```

- This translates to:

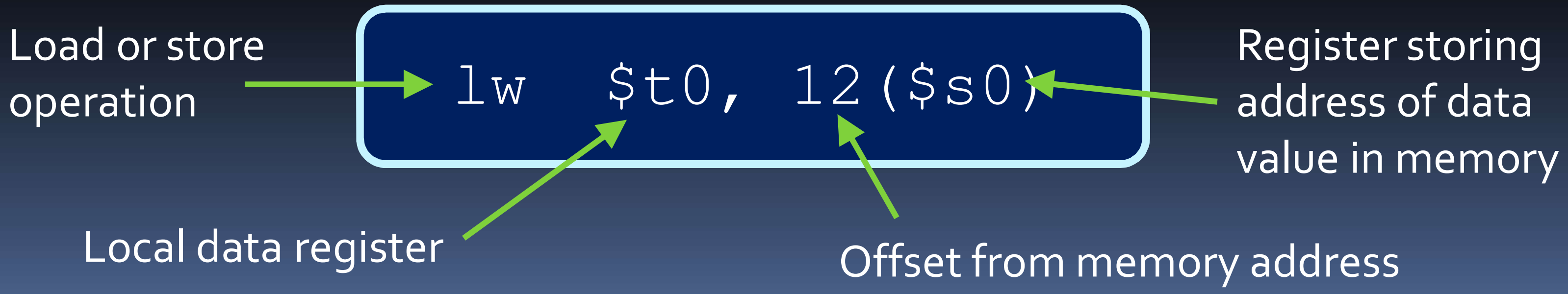
```
#  $t0 = i, $t1 = j
main:    add $t0, $zero, $zero        # set $t0 to 0
         add $t1, $zero, $zero        # set $t1 to 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE: 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

- Take out the initialization and update sections, and it’s 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:



```
lw  $t0, 12($s0)
```

Load or store operation

Register storing address of data value in memory

Local data register

Offset from memory address

Loads vs. Stores

- The terms “load” and “store” are seen from the perspective of the processor, looking at memory.

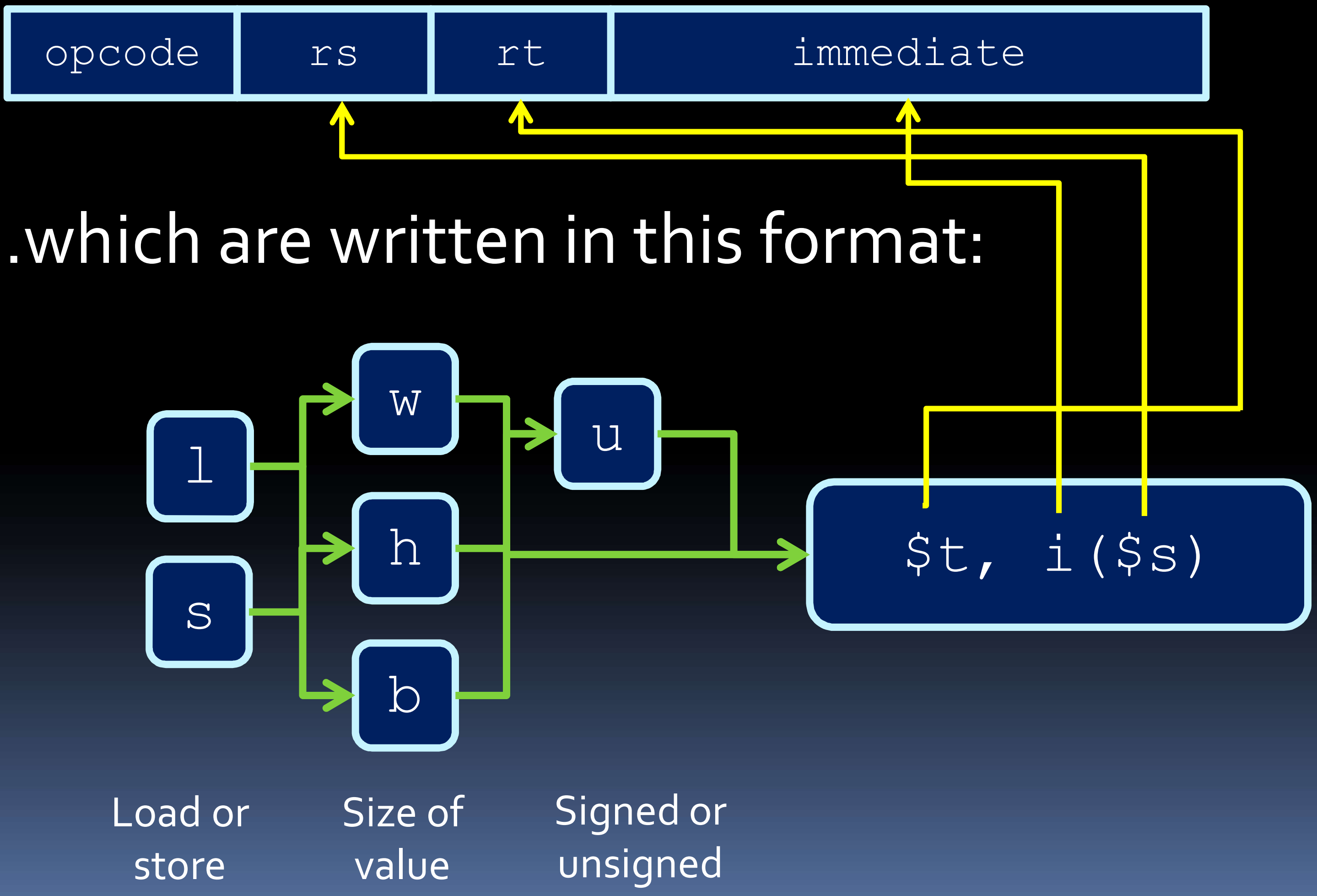
- Loads are read operations.
  - We load (i.e., read) from memory.
  - We load a value **from** a memory address into a register.
- Stores are write operations.
  - We store (i.e., write) a data value from a register **to** a memory address.
  - Store instructions do not have a destination register, and therefore do not write to the register file.

Memory Instructions in MIPS assembly

- Load & store instructions are I-type operations:

opcode	rs	rt	immediate

- …which are written in this format:

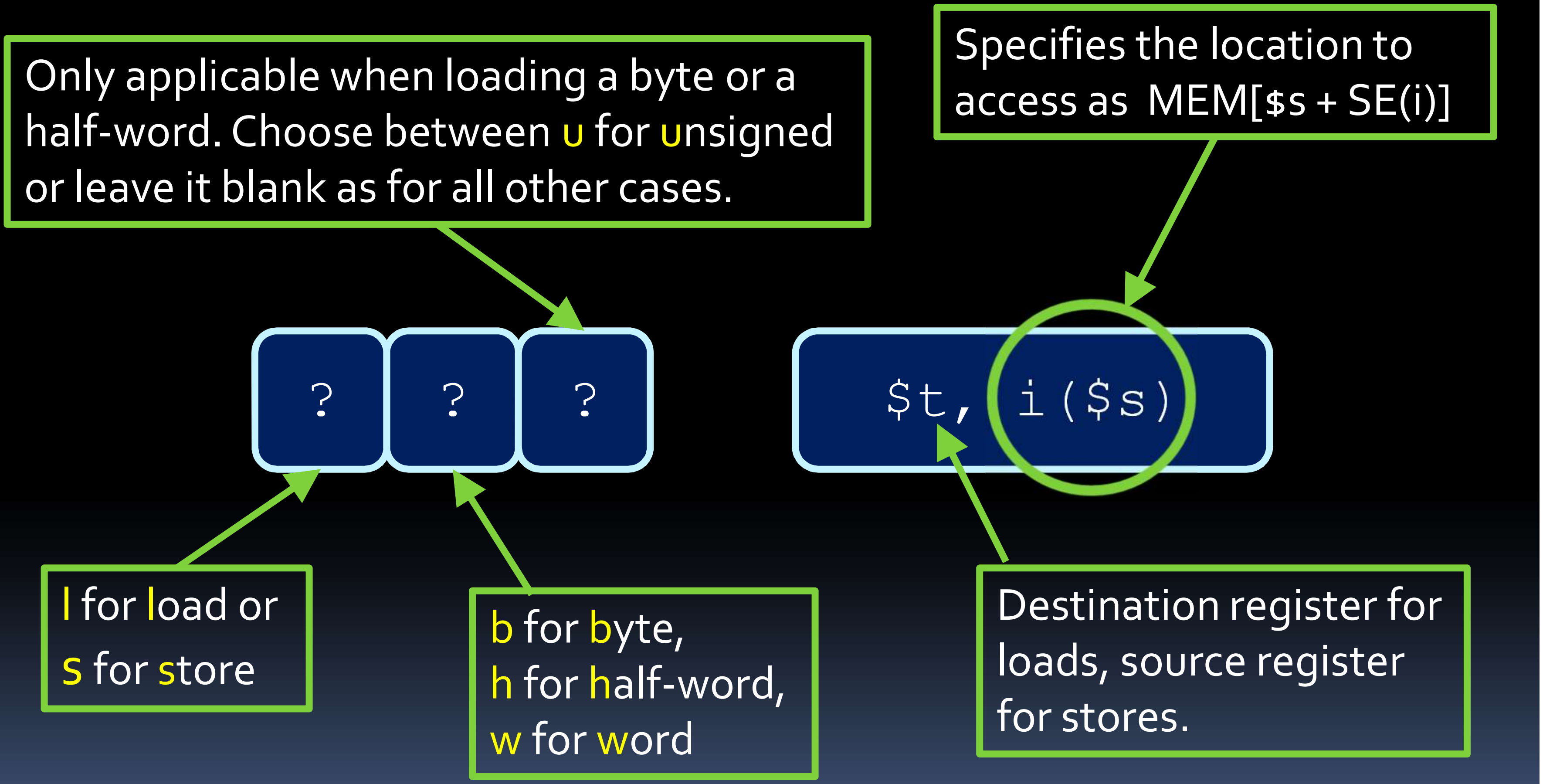


Load & store instructions

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)]

```
? ? ?    $t, i($s)
```

l for load or
s for store

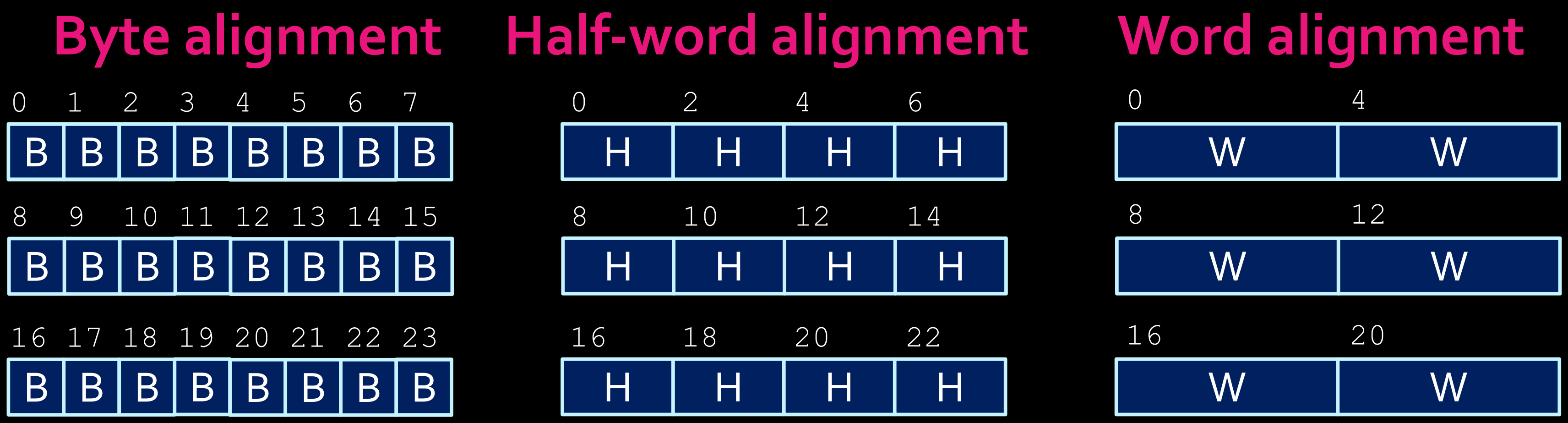
b for byte,
h for half-word,
w for word

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.

Memory alignment



Byte alignment

0 1 2 3 4 5 6 7

B B B B B B B B

8 9 10 11 12 13 14 15

B B B B B B B B

16 17 18 19 20 21 22 23

B B B B B B B B

Half-word alignment

0 2 4 6

H H H H

8 10 12 14

H H H H

16 18 20 22

H H H H

Word alignment

0 4

W W

8 12

W W

16 20

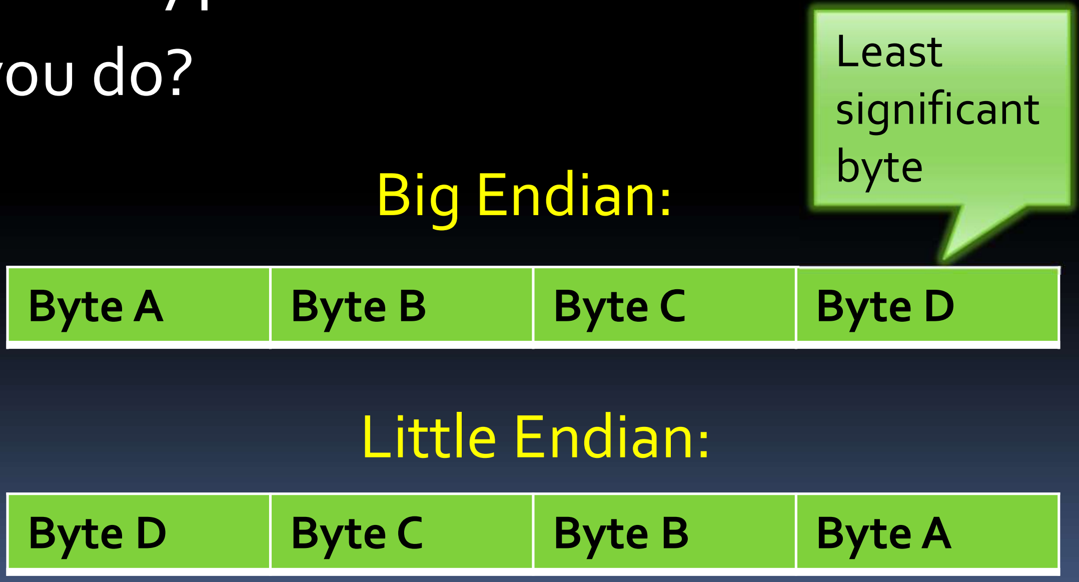
W W

- 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.
- <u>Next question:</u> How are the bytes within a word or half-word stored?

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



Least significant byte

Big Endian:

Byte A	Byte B	Byte C	Byte D

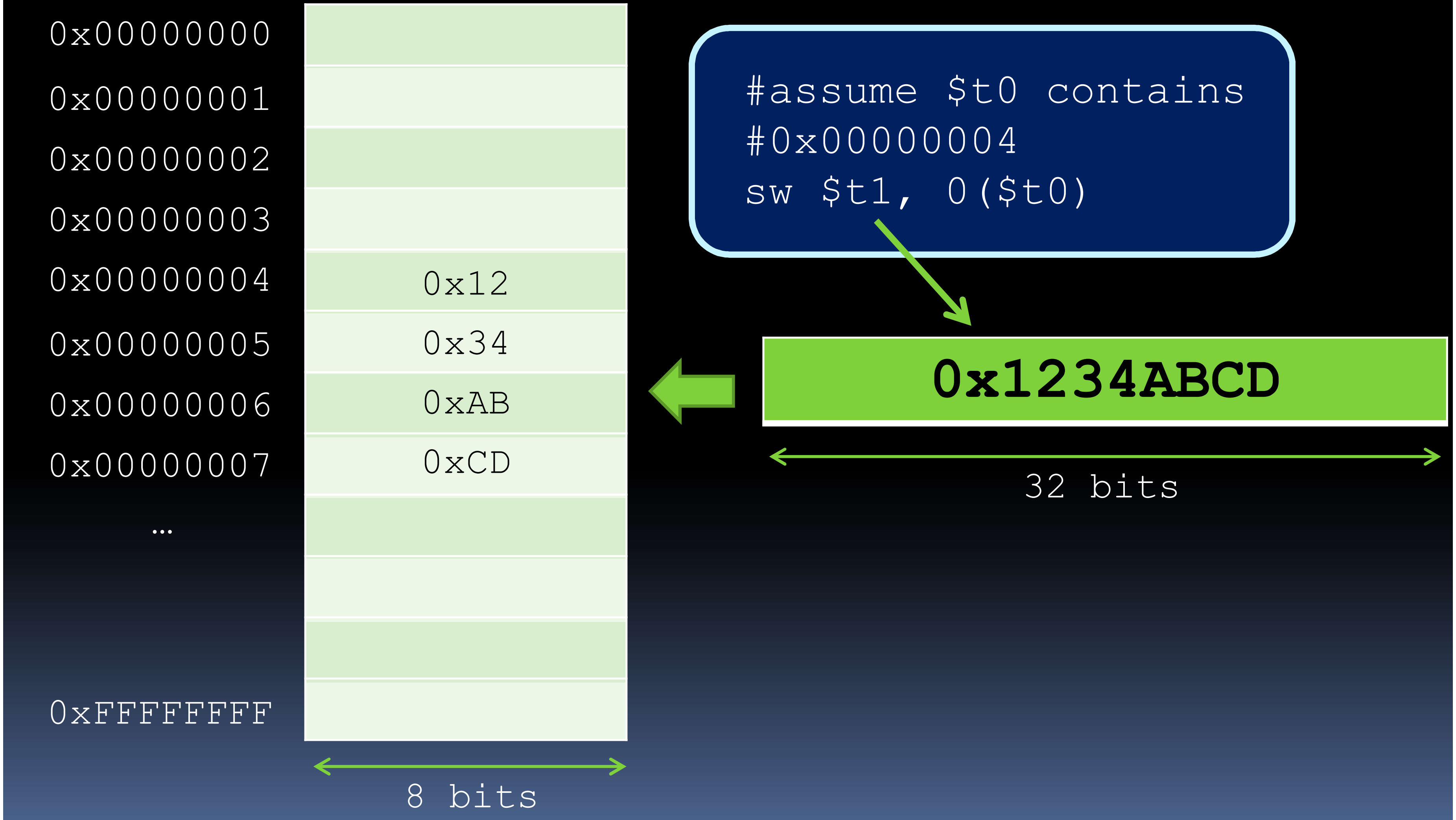
Little Endian:

Byte D	Byte C	Byte B	Byte A

Big Endian vs. Little Endian

- Big Endian
  - The most significant byte of the word is stored first (i.e., at address X). The 2nd 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 2nd 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	

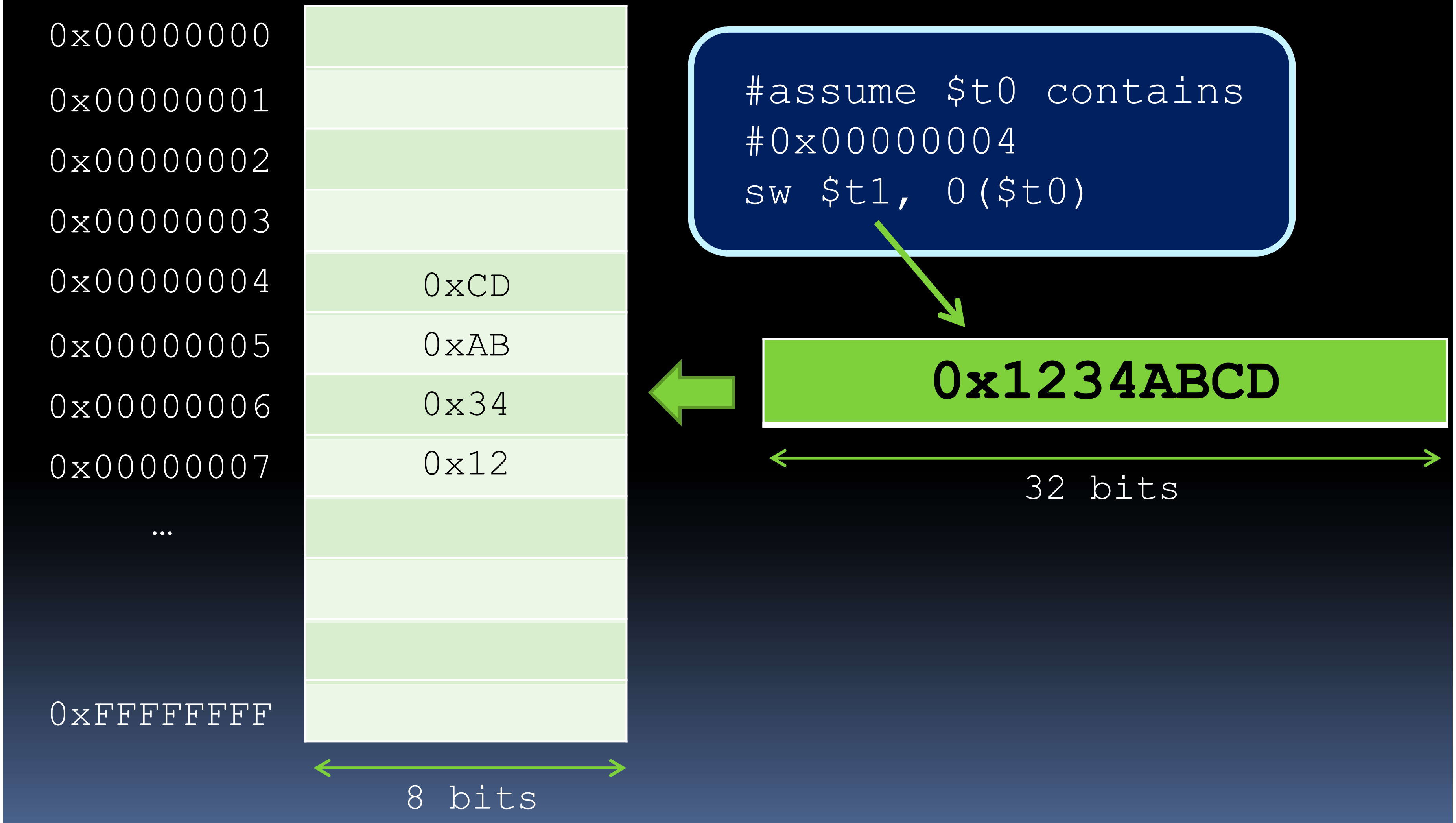
8 bits

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

0x1234ABCD

32 bits

Little Endian Example



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

8 bits

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

0x1234ABCD

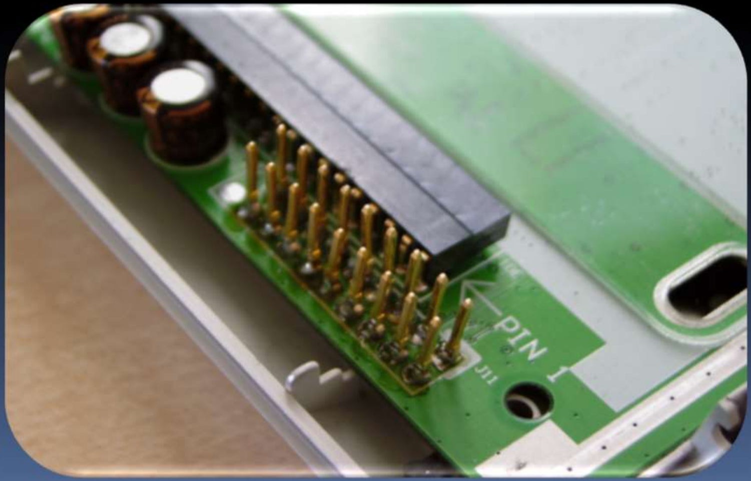
32 bits

MIPS Endianness

- MIPS processors are bi-endian, i.e., they can operate with either big-endian or little-endian byte order
- QtSpim 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 values are needed from a given memory location.
- Memory is also used to communicate with outside devices, such as keyboards and monitors.
  - Known as memory-mapped IO.
  - Invoked with a trap or syscall function.



It’s a trap!

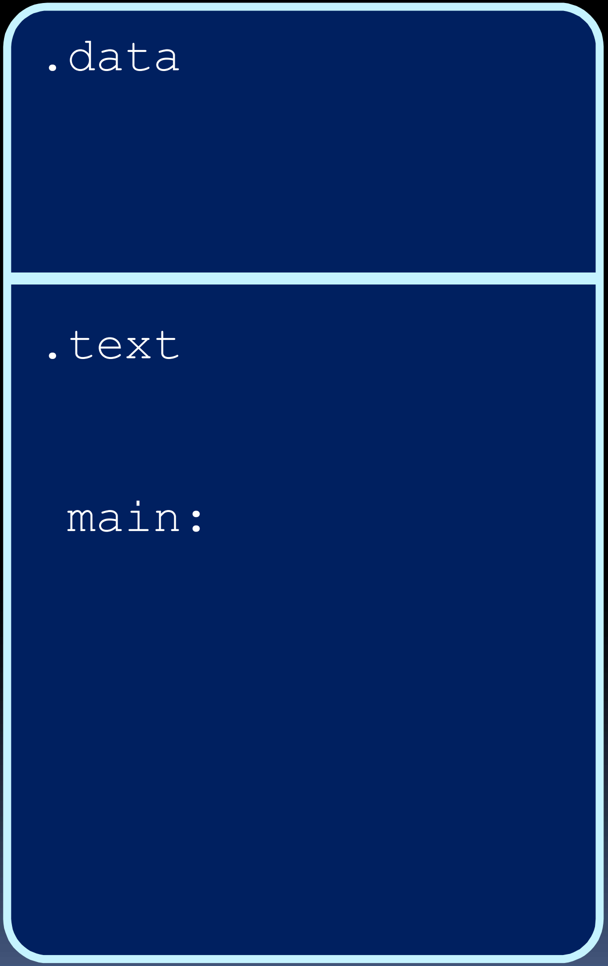
Instruction	Function	Syntax
`trap`	011010	i

- 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 ASCIIZ string to print
read_int	5	$2 is int read
read_float	6	$f12 is float read
read_double	7	$f12 (with $f13) is doubleread
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 Qtspim, use the Simple Machine under MIPS preferences (i.e., not the bare machine) and ensure Pseudo-instructions are 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 $at, immediate`** **# load upper immediate**
    - The "**immediate**" respresents the upper 16 bits of the memory address `label` corresponds to. These bits are loaded in the upper 16 bits of the dest. register. Lowest 16 bits are set to `0`.
    - Register `$at` (`$1`) is the register used by the assembler.

Instruction	Opcode/Function	Syntax	Operation
`lui`	`001111`	`$t, i`	`$t = i << 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` >= `$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, fetch the array values and store them in 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

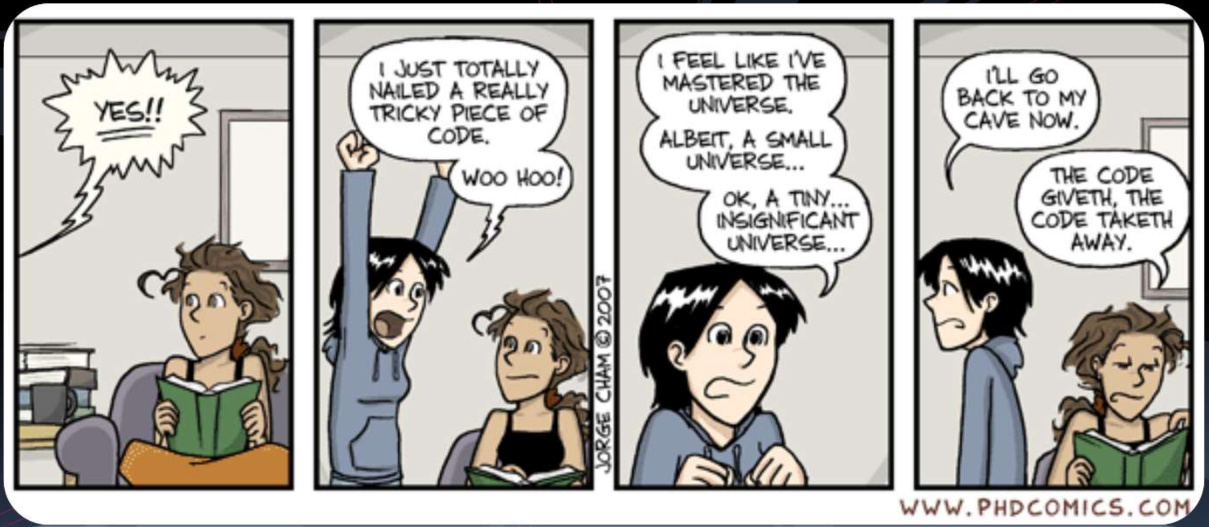
- How can we figure out the main purpose of this code?
- The `sw` lines indicate that values in `$t1` are being stored at `$t0`, `$t0+4` and `$t0+8`.
  - Each previous line sets the value of `$t1` to store.
- Therefore, this code stores the values 5, 13 and -7 into the struct at location `a1`.

```
            .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)
```

Struct program with comments

```
         .data
a1:      .space   12                #  declare 12 bytes
                                    #  of storage to hold
                                    #  struct of 3 ints
         .text
main:    addi     $t0, $zero, a1    #  load base address
                                    #  of struct into
                                    #  register $t0
         addi     $t1, $zero, 5     #  $t1 = 5
         sw       $t1, 0($t0)       #  first struct
                                    #  element set to 5;
                                    #  indirect addressing
         addi     $t1, $zero, 13    #  $t1 = 13
         sw       $t1, 4($t0)       #  second struct
                                    #  element set to 13
         addi     $t1, $zero, -7    #  $t1 = -7
         sw       $t1, 8($t0)       #  third struct
                                    #  element set to -7
```

Designing Assembly Code


YES!!
I JUST TOTALLY NAILED A REALLY TRICKY PIECE OF CODE.
WOO HOO!
JORGE CHAM © 2007
I FEEL LIKE I'VE MASTERED THE UNIVERSE.
ALBEIT, A SMALL UNIVERSE...
OK, A TINY... INSIGNIFICANT UNIVERSE...
I'LL GO BACK TO MY CAVE NOW.
THE CODE GIVETH, THE CODE TAKETH AWAY.
WWW.PHDCOMICS.COM

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

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

- 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.

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
j:           .space      4

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

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

```
             .data
i:           .space      4
j:           .space      4

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
```

Can you spot what was changed, and why?

Function Parameters

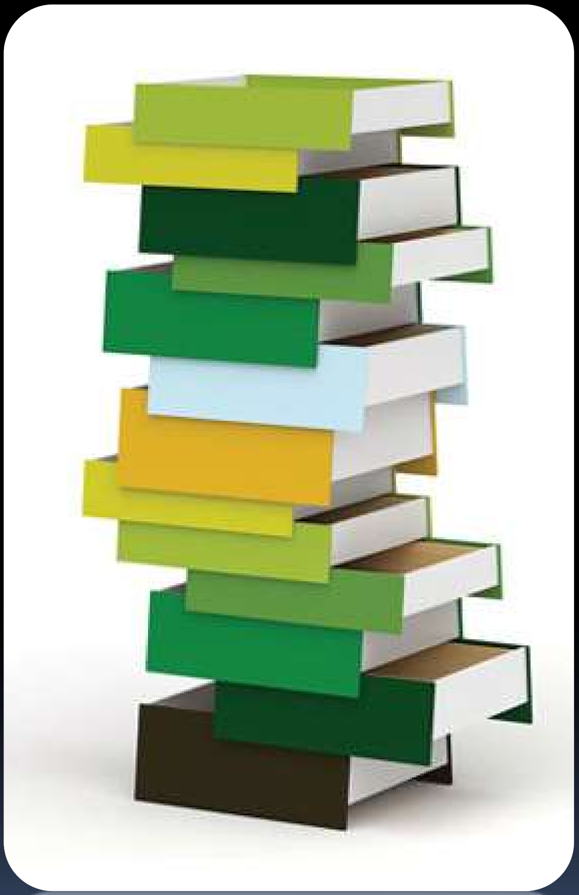
And why it’s crucial to understand the calling stack

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 code by defining the 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?

The Stack and the Stack Pointer

- A special register stores the stack pointer, which points to the *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 parameters) 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.



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()

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

- 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]`.

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()

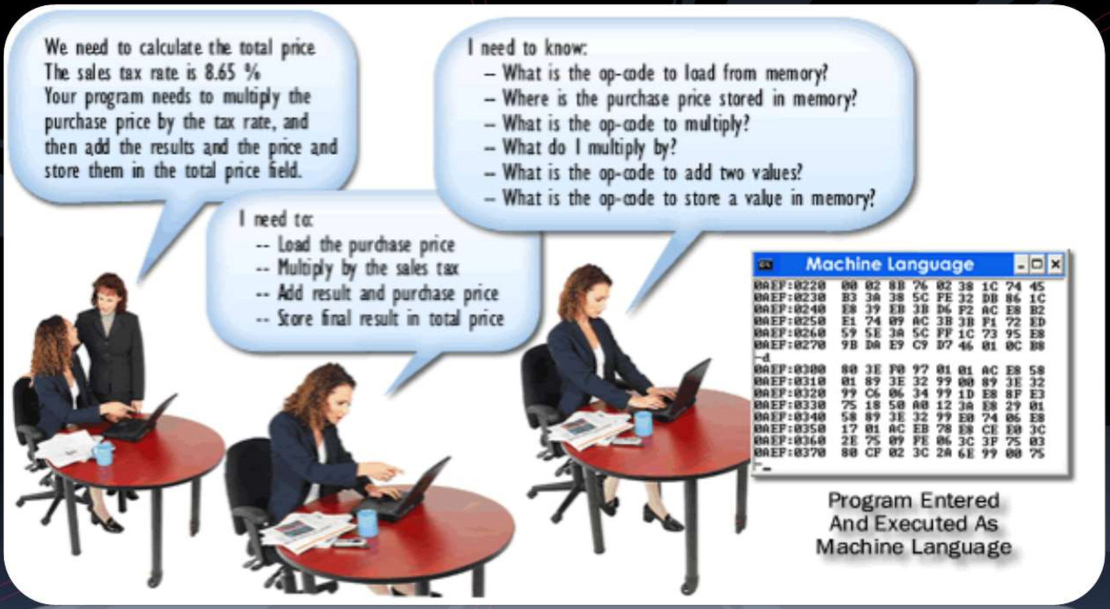
- Main algorithm:
  - What steps do we need to perform?
    - Get the location of `x[i]` and `y[i]`.
    - Fetch a character from `y[i]` and store it in `x[i]`.
    - Jump to the end if the character is the null character.
    - Otherwise, increment `i` and jump to the beginning.
    - At the end, push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

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

Translated strcpy program

```
strcpy:           lw     $a0, 0($sp)          # pop x address
                  addi   $sp, $sp, 4          #   off the stack
  initialization  lw     $a1, 0($sp)          # pop y address
                  addi   $sp, $sp, 4          #   off the stack
                  add    $t0, $zero, $zero    # $t0 = offset i
L1:               add    $t1, $t0, $a0        # $t1 = x + i
                  lb     $t2, 0($t1)          # $t2 = x[i]
                  add    $t3, $t0, $a1        # $t3 = y + i
main algorithm    sb     $t2, 0($t3)          # y[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
                  addi   $t0, $zero, 1        #   the top of
             end  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:
-- Load the purchase price
-- Multiply by the sales tax
-- Add result and purchase price
-- Store final result in total price
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

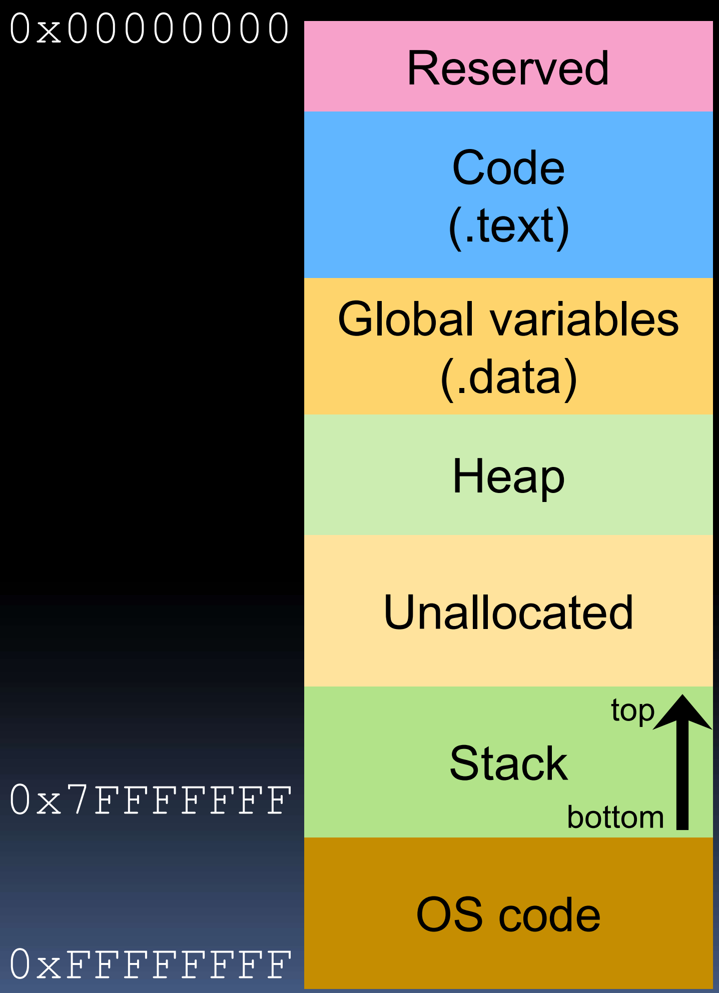
```
0AEF:0220  00 02 8B 76 02 38 1C 74 45
0AEF:0230  B3 3A 38 5C FE 32 DB 86 1C
0AEF:0240  E8 39 EB 3B D6 F2 AC E8 B2
0AEF:0250  E1 74 09 AC 3B 3B F1 72 ED
0AEF:0260  59 5E 3A 5C FF 1C 73 95 E8
0AEF:0270  9B DA E9 C9 D7 46 01 0C B8
-d
0AEF:0300  80 3E F0 97 01 01 AC E8 58
0AEF:0310  01 89 3E 32 99 00 89 3E 32
0AEF:0320  99 C6 06 34 99 1D E8 8F E3
0AEF:0330  75 18 50 A0 12 3A E8 29 01
0AEF:0340  58 89 3E 32 99 E0 74 06 E8
0AEF:0350  17 01 AC EB 78 E8 CE E0 3C
0AEF:0360  2E 75 09 FE 06 3C 3F 75 03
0AEF:0370  80 CF 02 3C 2A 6E 99 00 75
-_
```

Program Entered
And Executed As
Machine Language

How Functions Work

- To support functions we need to be able to:
  - communicate function arguments and return values
    - We'll use some registers and also the stack for this (stack is part of memory)
  - store variables local to that function and also ensure functions don't clobber useful data on registers
    - The stack will come to our rescue once more! Which means we need to know about calling conventions too.
  - return to the calling site (i.e., after the return statement execute the instruction after the one that did the function call)

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.

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.

How do we call a function?

- `jal FUNCTION_LABEL`
  - This happens **after** we've set the appropriate values to $a0-$a3 registers and/or pushed arguments to the stack.

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

- `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`.

How do we return from a function?

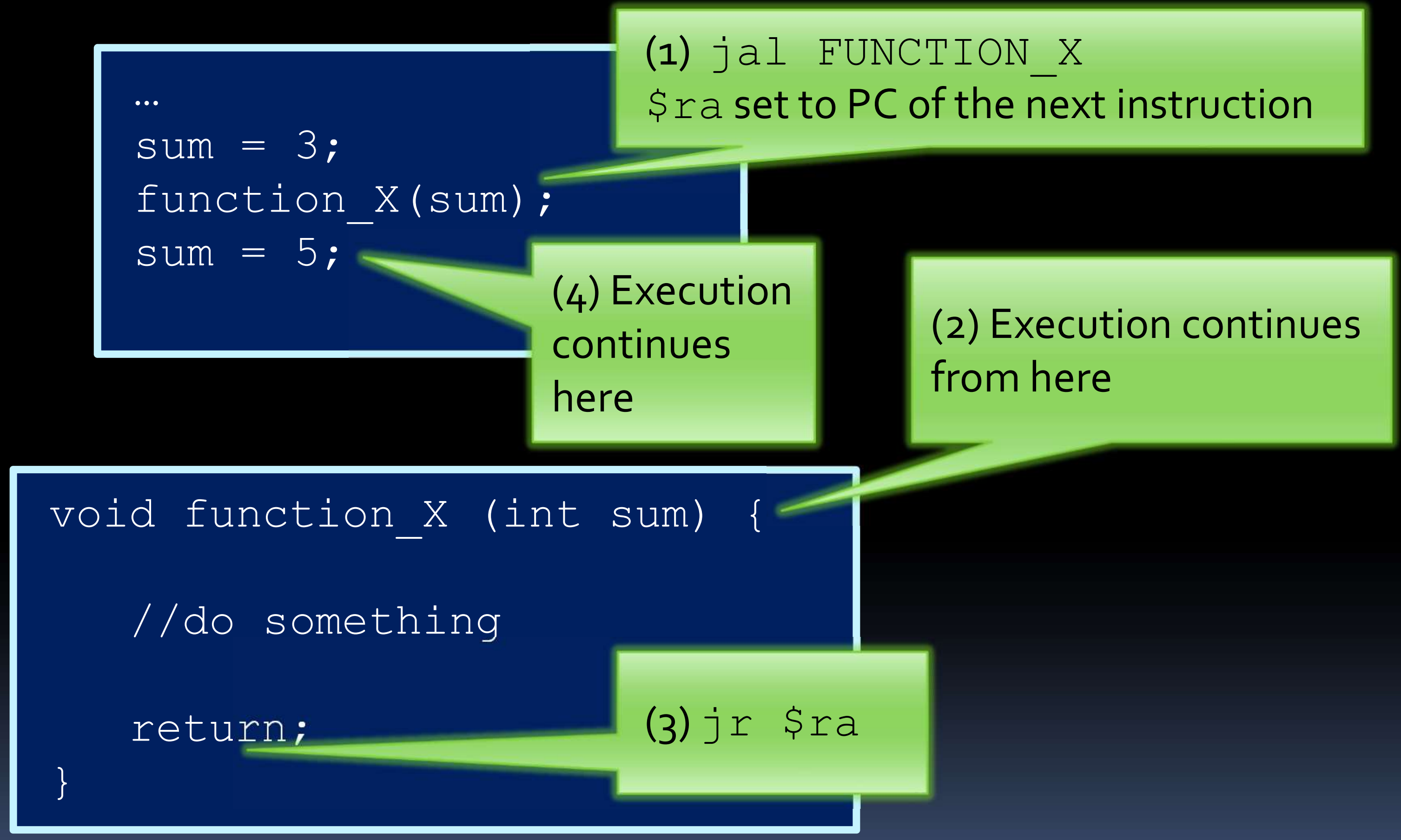
- `jr $ra`
  - The PC is set to the address in `$ra`.

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

- But how do we know what's in `$ra`?
  - `$ra` was set by the most recent `jal` instruction (function call)!

```
void function_X (int sum) {

    //do something

    return;
}
```

Function Calls – Cont’d



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

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

(4) Execution continues here

(2) Execution continues from here

```
void function_X (int sum) {

   //do something

   return;
}
```

(3) `jr $ra`

Function example! (functions_ex1.asm)

```
.data
RESULT:  .word 0

.text
main:   addi $t1, $zero, 20   # Simple demo for our
        addi $t2, $zero, 40   # function call arguments
```

What should I do before calling sum_function w/ arguments 20 and 40?

```
        jal sum_function # call the function
```

How can I store the return value in the memory location indicated by the label RESULT?

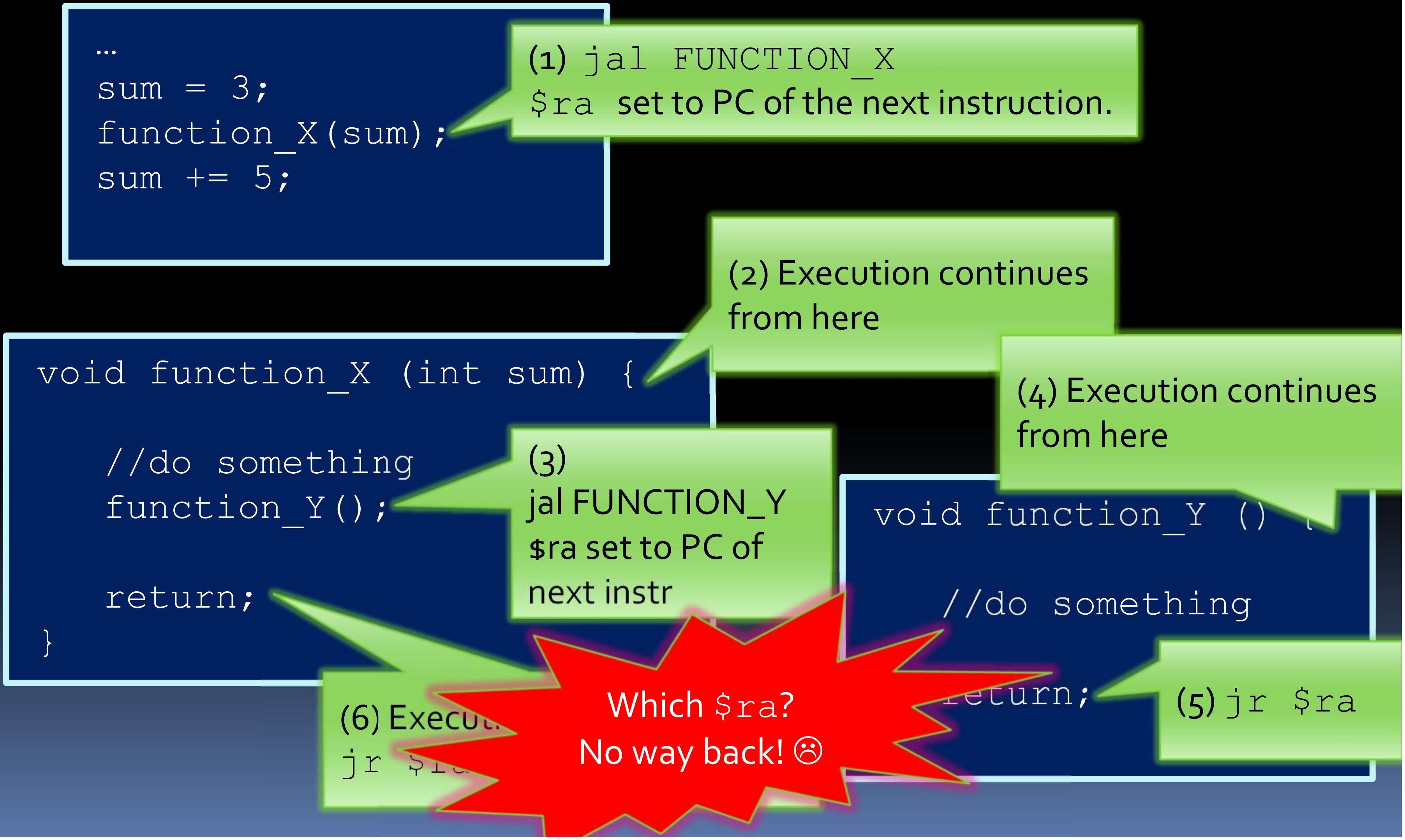
```
END:    j END  # Just added this here to show we're done.

sum_function:
```

Simple function. Add 2 numbers (values of the parameters) and return the result. How do we make this happen?

Function example! (functions_ex1.asm)

```
.data
RESULT: .word 0

.text
main:  addi $t1, $zero, 20  # Simple demo for our
       addi $t2, $zero, 40  # function call arguments
       add $a0, $t1, $zero  # Place arguments to $a0, $a1.
       add $a1, $t2, $zero  #  (as per convention)
       jal sum_function # call the function

       la $t3, RESULT   # store returned value to memory.
       sw $v0, 0($t3)
END:   j END  # Just added this here to show we're done.

sum_function:    add $v0, $a0, $a1
                 jr $ra
```

Nested Function Call Issue



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

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

```
void function_X (int sum) {

   //do something
   function_Y();

   return;
}
```

(2) Execution continues from here

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

```
void function_Y () {

   //do something

   eturn;
```

(4) Execution continues from here

(5) `jr $ra`

(6) Execu
`jr $r`

Which `$ra`?
No way back! ☹

The stack to the rescue!

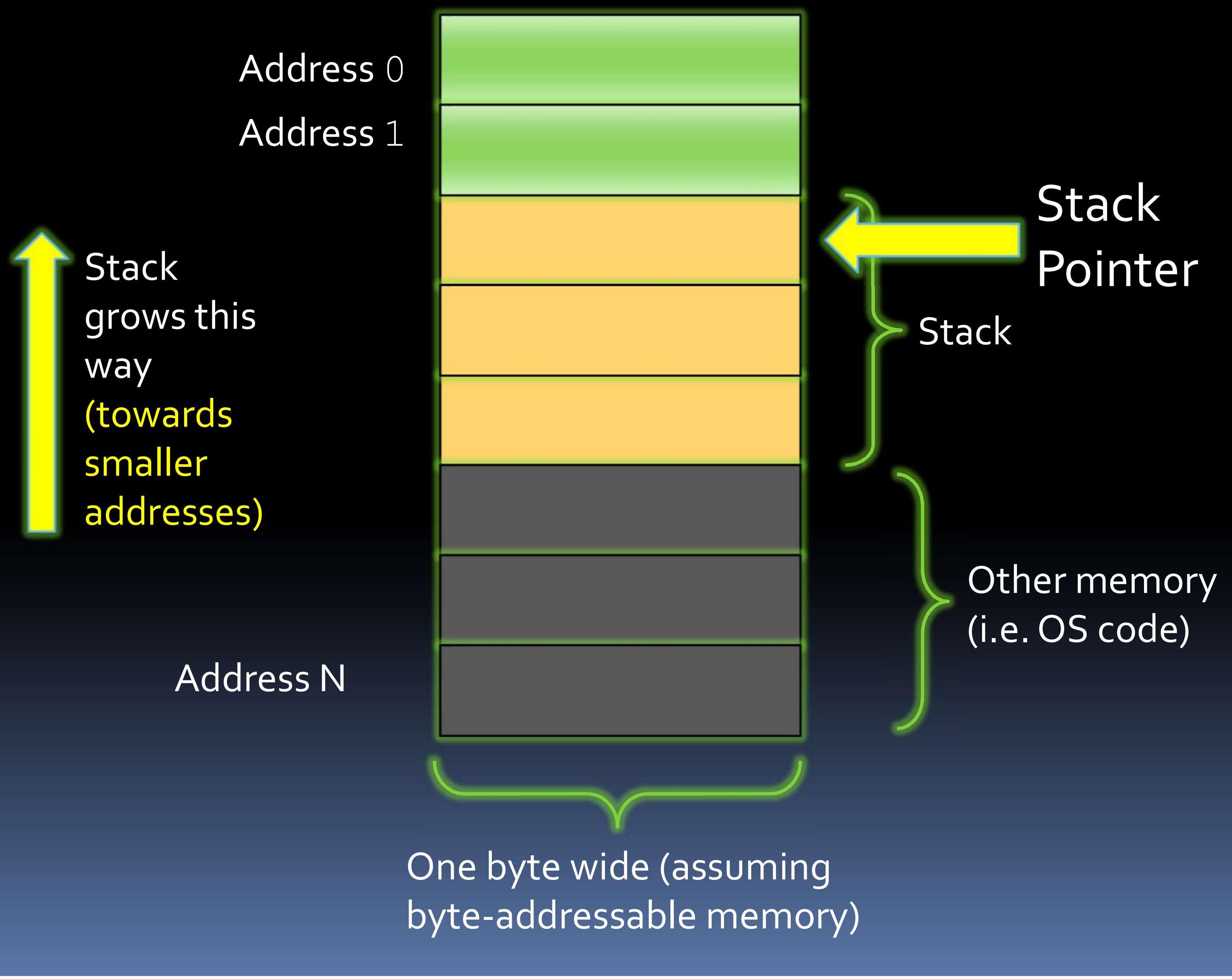
- What if you store `$ra` in the stack?
  - Different `$ra` values will exist in the stack over time.

- We can also use the stack to store*:
  - Function arguments
  - Function return values
  - And also to maintain register values (more on this later).

* *As mentioned before there are some predefined registers used for the function arguments and return values; the stack is used if this number is exceeded.*

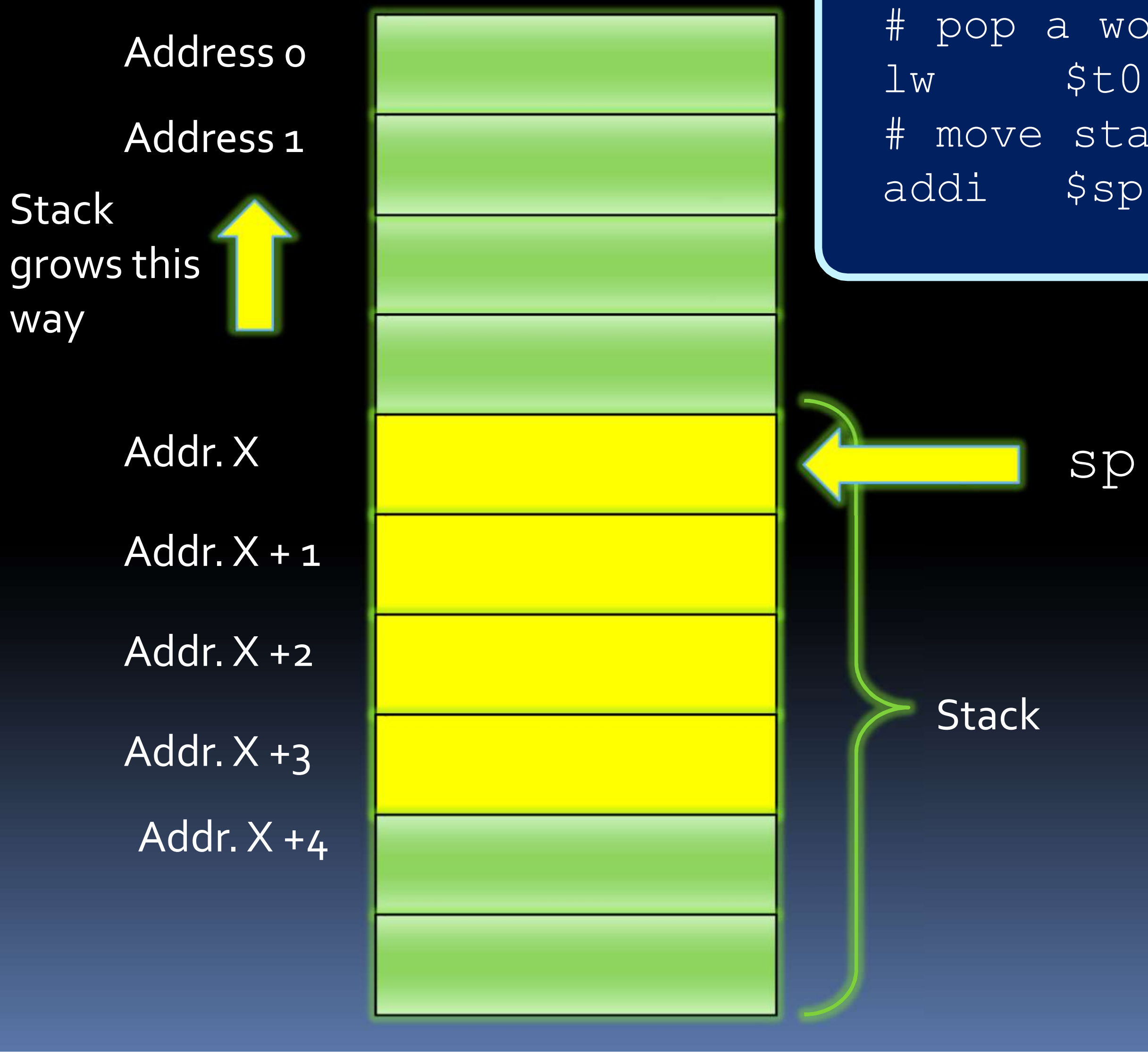
- *E.g., if there are more than 4 arguments.*

The Stack, illustrated



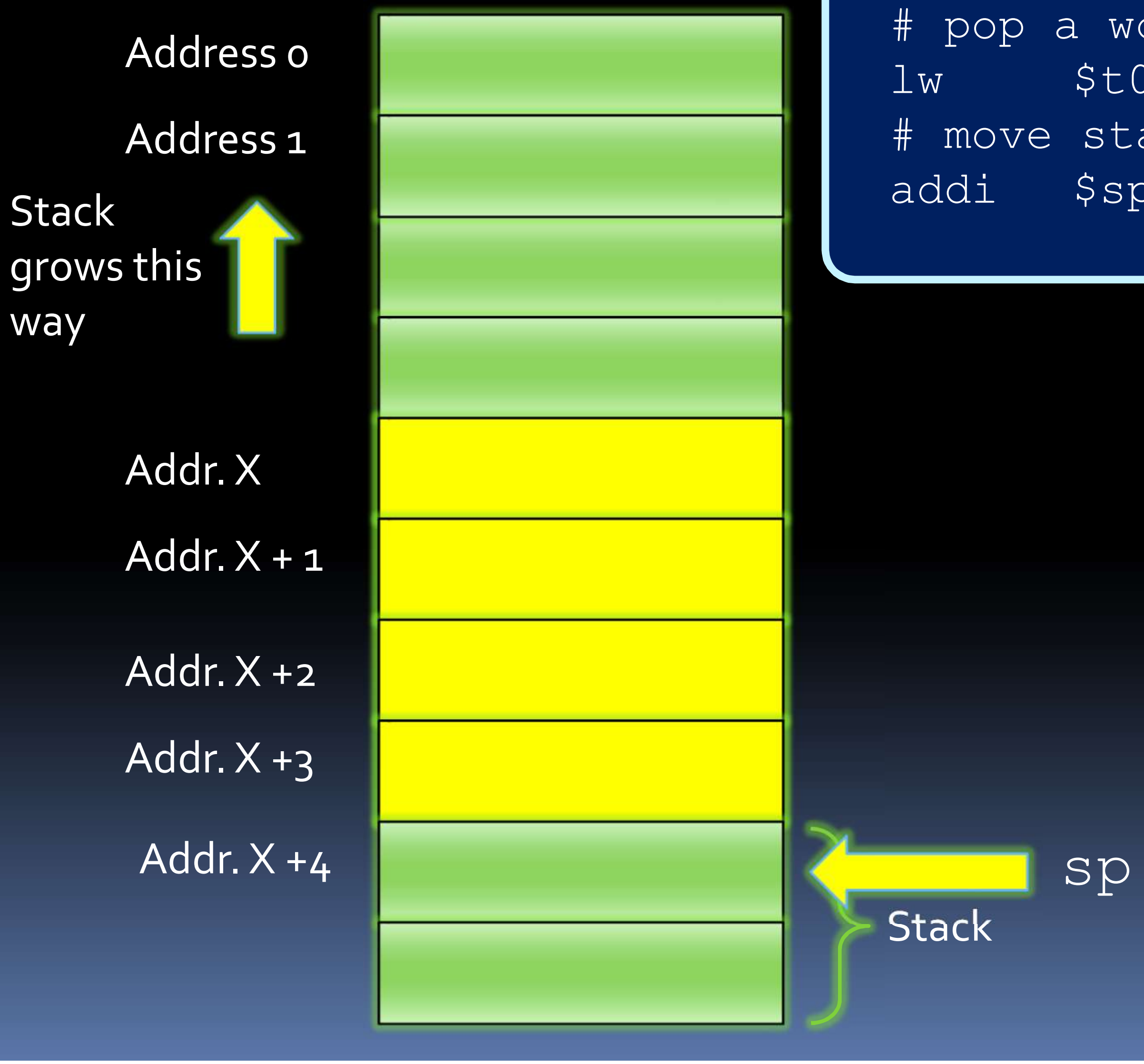
Popping Values off the stack - Before

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



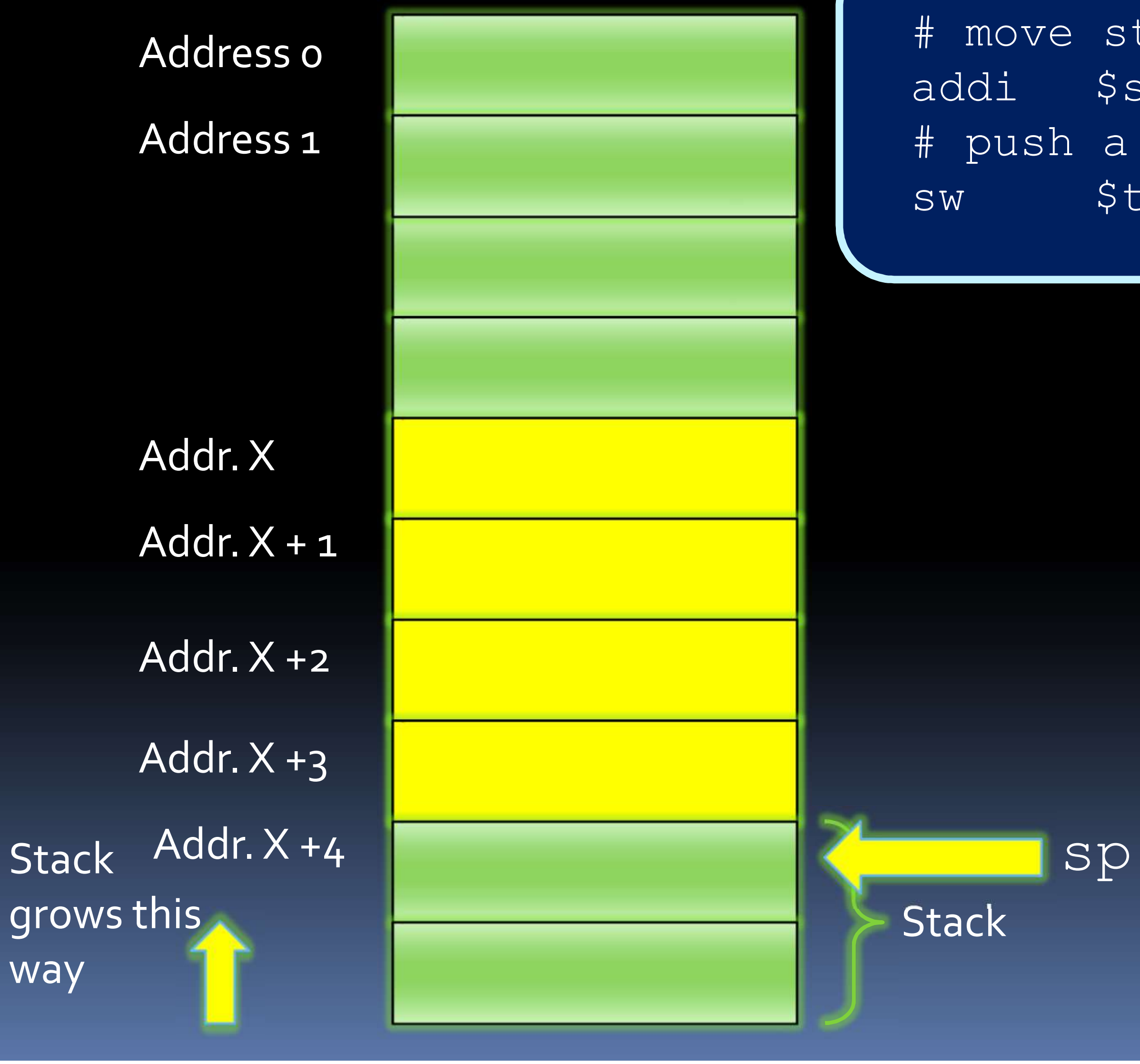
Popping Values off the stack - After

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



Pushing Values to the stack - Before

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



Address 0

Address 1

Addr. X

Addr. X + 1

Addr. X +2

Addr. X +3

Addr. X +4

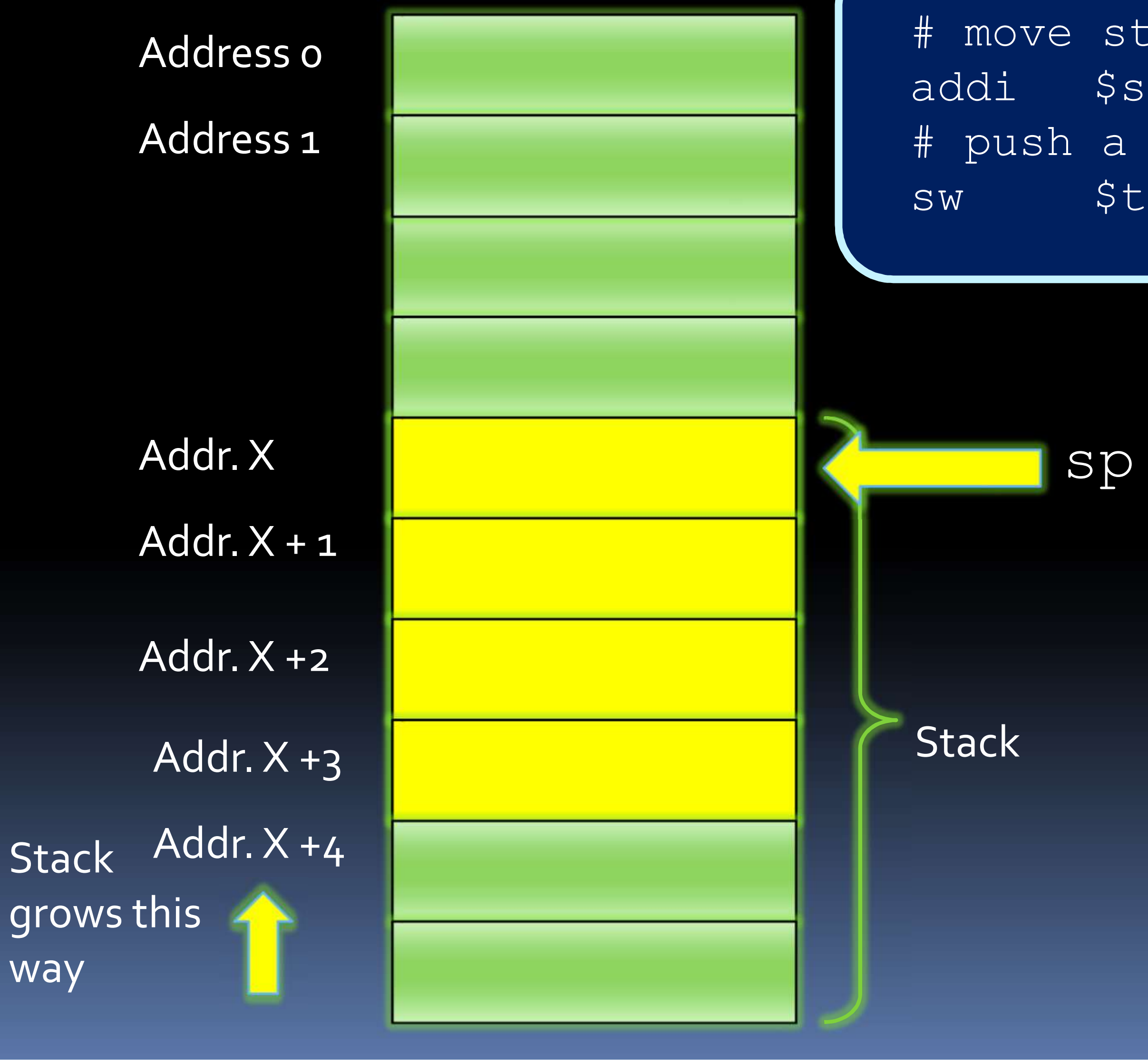
Stack grows this way

sp

Stack

Pushing Values to the stack - After

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



Stack Usage

- Pushing something onto the stack
  - 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:
  - 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

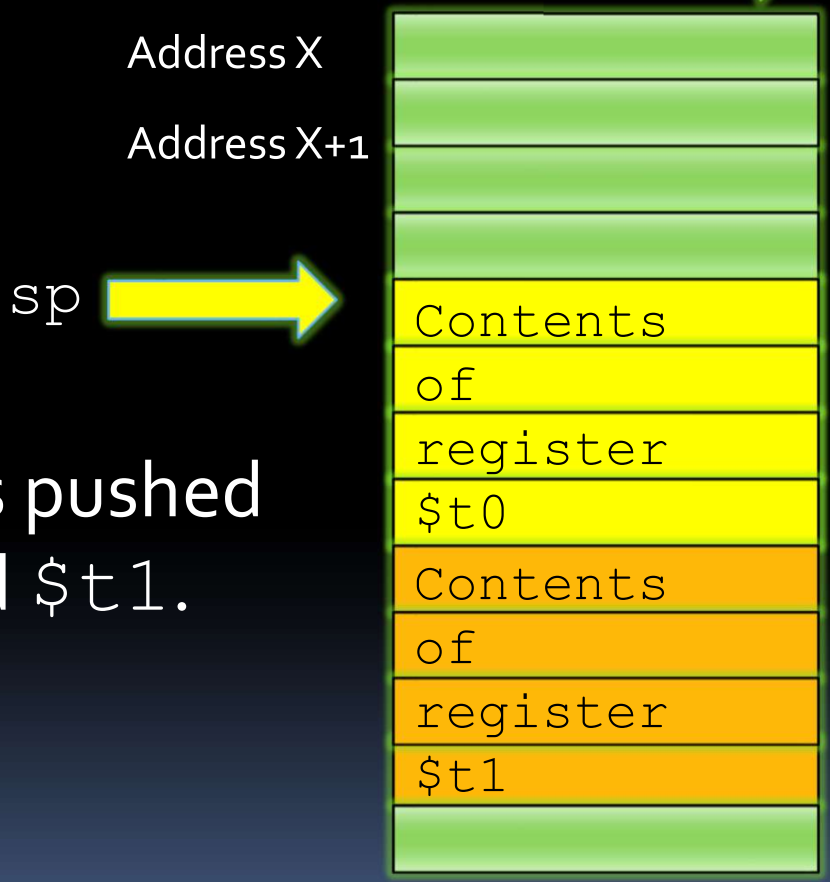
- Any space you allocate on the stack, you should later de-allocate.
- You should pop the items in the same order as you push them.
  - It might help to draw out an image of how your stack will look like.
- When pushing more than one item onto the stack, you can :
  - Either allocate all the space in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

Stack storage example

Figure shows stack *after* the push.

- Push contents of registers `$t0` and `$t1` onto the stack.

```
addi $sp, $sp, -8
sw $t0, 0($sp)
sw $t1, 4($sp)
```



- Restore stack values pushed to registers `$t0` and `$t1`.

```
lw $t0, 0($sp)
lw $t1, 4($sp)
addi $sp, $sp, 8
```

Alternative implementation

- Push contents of registers `$t0` and `$t1` onto the stack.

```
addi $sp, $sp, -8
sw $t0, 0($sp)
sw $t1, 4($sp)
```

```
# Alternative
addi $sp, $sp, -4
sw $t0, 0($sp)
addi $sp, $sp, -4
sw $t1, 0($sp)
```

- Are these two code snippets equivalent?
  - What is the element at the top of the stack (i.e., the element that `$sp` points to)?
    - $t0 for the original code snippet
    - $t1 for the alternative implementation
  - So the two code snippets are NOT equivalent from the perspective of what is on the stack.

The fixed alternative version

- Push contents of registers `$t0` and `$t1` onto the stack.

```
addi $sp, $sp, -8
sw $t0, 0($sp)
sw $t1, 4($sp)
```

```
# Alternative
addi $sp, $sp, -4
sw $t1, 0($sp)
addi $sp, $sp, -4
sw $t0, 0($sp)
```

- Restore values pushed to the stack to registers `$t0` and `$t1`.

```
lw $t0, 0($sp)
lw $t1, 4($sp)
addi $sp, $sp, 8
```

```
# Alternative
lw $t0, 0($sp)
addi $sp, $sp, 4
lw $t1, 0($sp)
addi $sp, $sp, 4
```

Back to Function Calls

- 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!

Need to know about the **caller vs. callee calling conventions.**

Calling Conventions

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

- 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`)

Register Saving Conventions

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

- 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!

Caller-Saved ($t0-$t9) vs. Callee-Saved ($s0-$s7) Registers

- A function (or code) that is a caller
  - Using $t0-$t9 and you care for their values?
    - Push them to the stack just before you make a function call and restore them immediately after the calling site.
  - Using $s0-$s7?
    - No action needed. It is the responsibility of the callee to ensure these registers are not modified.

- A function that is a callee
  - Using $t0-$t9?
    - No action needed. It it the responsibility of the caller to ensure there registers are not modified.
  - Using $s0-s7?
    - You need to ensure these registers are not modified.
    - If you plan to modify them, push them to the stack in the beginning of your function and restore them in the very end just before the jr $ra.

Note: If a function is both a caller and a callee, it will fall under both categories.

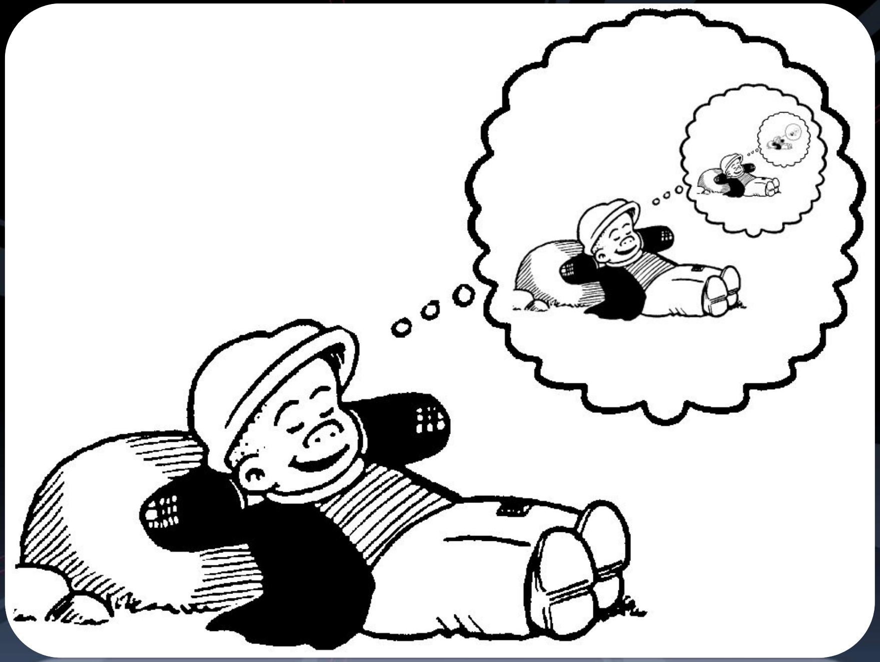
register_saving.asm (a template)

- Let’s go over this template first before we start talking about recursion!

- Note a recursive function (i.e., a function that calls itself) is both a caller and a callee.
  - Functions that call other functions are also both a a caller and a callee.

- To really see this principle at work, let’s examine how to create a recursive function.

Recursion in Assembly



Example: factorial(int n)

- Basic pseudocode for recursive factorial:

```
Base Case (n == 0)
    return 1
Get factorial(n-1)
    Store result in “product”
Multiply product by n
    Store in “result”
Return result
```

Recursive programs

- How do we handle recursive programs?
  - Still needs base case and recursive step, as with other languages.

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

  - Main difference: 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?

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!=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

n → $t0

n-1 → $t1

fact(n-1) → $t2

- 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 $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 $ra and $t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

factorial(int n)

n → $t0

n-1 → $t1

fact(n-1) → $t2

```
fact:        lw $t0, 0($sp)
             addi $sp, $sp, 4
             bne $t0, $zero, not_base
             addi $t0, $zero, 1
             addi $sp, $sp, -4
             sw $t0, 0($sp)
             jr $ra
not_base:    addi $sp, $sp, -4
             sw $t0, 0($sp)
             addi $sp, $sp, -4
             sw $ra, 0($sp)
             addi $t1, $t0, -1
             addi $sp, $sp, -4
             sw $t1, 0($sp)
             jal fact
```

factorial(int n)

n → $t0
n-1 → $t1
fact(n-1) → $t2

```
lw $t2, 0($sp)
addi $sp, $sp, 4
lw $ra, 0($sp)
addi $sp, $sp, 4
lw $t0, 0($sp)
addi $sp, $sp, 4
mult $t0, $t2
mflo $t3
addi $sp, $sp, -4
sw $t3, 0($sp)
jr $ra
```

Translated recursive program (part 1)

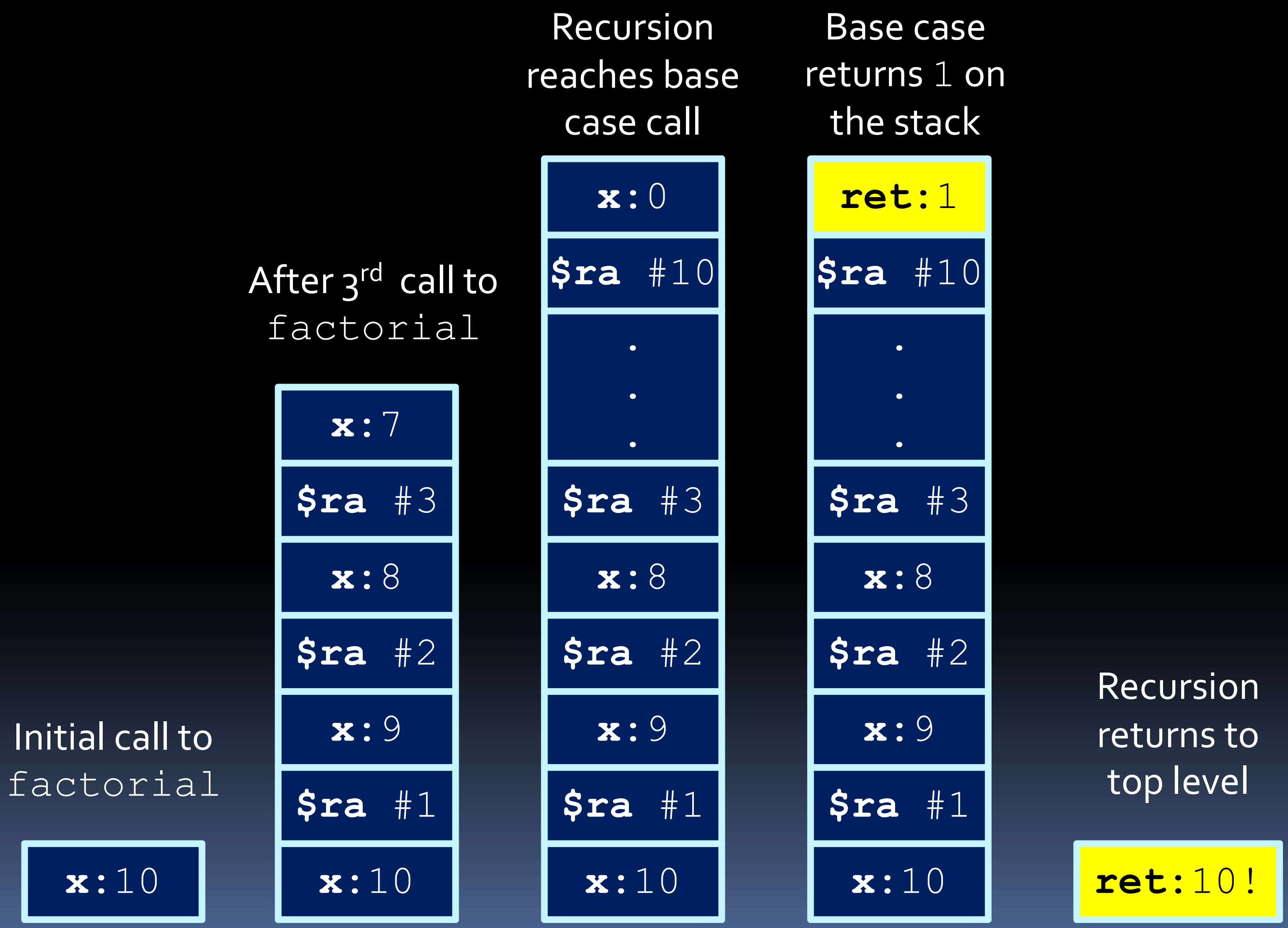
```
main:         addi   $t0, $zero, 10       # call fact(10)
              addi   $sp, $sp, -4         #   by putting 10
              sw     $t0, 0($sp)          #   onto stack
              jal    factorial            # result will be
              ...                         #   on the stack

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 program (part 2)

```
(continued from part 1 – returning from recursive call)
            lw     $t2, 0($sp)         # get 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



Initial call to factorial

x:10

After 3rd call to factorial

x:7
$ra #3
x:8
$ra #2
x:9
$ra #1
x:10

Recursion reaches base case call

x:0
$ra #10
.
.
.
$ra #3
x:8
$ra #2
x:9
$ra #1
x:10

Base case returns 1 on the stack

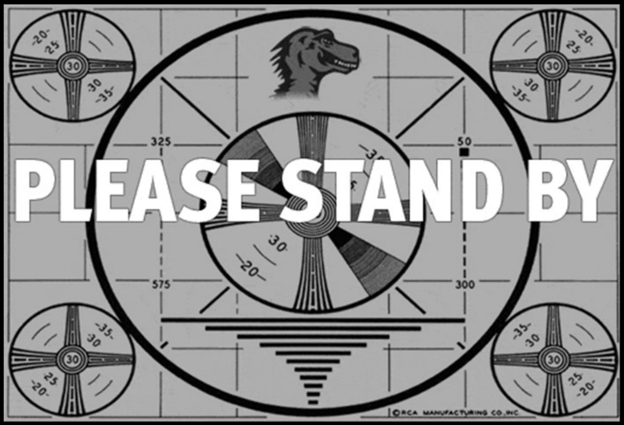
ret:1
$ra #10
.
.
.
$ra #3
x:8
$ra #2
x:9
$ra #1
x:10

Recursion returns to top level

ret:10!

A note on interrupts

- Interrupts take place when an external event requires a change in execution.
  - Example: arithmetic overflow, system calls (`syscall`), undefined instructions.
  - Usually signaled by an external input wire, which is checked at the end of each instruction.



A note on interrupts

- Interrupts can be handled in two general ways:
  - Polled handling: 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.
  - Vectored handling: 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 polled interrupt handling, the processor jumps to exception handler code, based on the value in the cause register (see table).
    - 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.

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